DmRPC: Disaggregated Memory-aware Datacenter RPC for Data-intensive Applications

Jie Zhang[†]
Zhejiang University
Hangzhou, China
carlzhang4@zju.edu.cn

Xuzheng Chen[†]
Zhejiang University
Hangzhou, China
chenxuz@zju.edu.cn

Yin Zhang
Zhejiang University
Hangzhou, China
zhangyin98@zju.edu.cn

Zeke Wang[*]
Zhejiang University
Hangzhou, China
wangzeke@zju.edu.cn

***Abstract*—Modern datacenter applications are increasingly being built using a *microservices* architecture. These microservices communicate with each other using datacenter RPCs. RPC's pass by value semantics incur redundant data movement along the network, especially for data-intensive applications. Naively introducing a shared global address space to datacenter RPC does not work as it would couple microservices and require microservices to handle data consistency, significantly complicating the development and deployment of applications. Fortunately, the modern datacenter is embracing disaggregated memory (DM). In a DM-enabled datacenter, servers running the microservices can be all connected to one global disaggregated memory pool, thus the pass by value semantics can be replaced by pass by reference. However, prior work on DM requires complicated synchronization primitives to share data across physical machines, so naively adopting them to datacenter RPC would harm microservices' agility and modularity.**

To this end, we present DmRPC, a DM-aware datacenter RPC for data-intensive datacenter applications to our knowledge. First, DmRPC introduces a DM-aware shared global address space to provide the semantics of pass by reference to datacenter RPC, thus alleviating the redundant data movement issue. Second, DmRPC adopts a `copy-on-write` mechanism to avoid complicating application logic to handle data consistency while guaranteeing high performance. We have applied DmRPC to two different implementations of DM, one is network-based (DmRPC-net) while the other is CXL-based (DmRPC-CXL). Our evaluations on synthetic 7-tier microservices workloads show that DmRPC-net (or DmRPC-CXL) achieves 4.2× (or 8.3×) higher throughput and achieves 1.1× (or 1.7×) lower average latency than that of the baseline, respectively. On a widely used microservice benchmark DeathStarBench, DmRPC-net can achieve 3.1× higher throughput and 2.5× lower average latency than the baseline.

I. Introduction

Modern datacenter applications have been using the microservices programming model to improve their agility, elasticity, and modularity [26], [26], [35], [39], [41], [72]. In particular, microservices break complex monolithic logic into many fine-grained and loosely-coupled services, and microservices communicate with each other over Remote Procedure Calls (RPC) such as eRPC [37], DaRPC [59], gRPC [1]. However, compared with monolithic design, the main issue of introducing microservices is to bring much additional network data movement [9]. The underlying reason is that the *pass by value* semantic of RPC results in redundant data movement. Many microservices do not even touch the transferred RPC arguments or only touch a small portion of them, they just forward the arguments to the next microservice by calling a new RPC. For example, an application-level load balancer only forwards the RPC requests to an idle server without accessing the content in the requests (∼60% traffic in the data center would go through a load balancer [56]). The issue of redundant data movement becomes more severe for large RPC arguments, which is common in the modern datacenter. For example, the commodity block storage service uses RPC to transfer large data blocks (tens to hundreds of KBs) [28], [49].

Introducing a global address space to datacenter RPC can alleviate the issue of redundant data movement. A global address space shared by all microservices can provide a *pass by reference* semantic, thus the redundant movement only involves small references (metadata that acts like a pointer). When a microservice really needs to access the data, it uses the reference to fetch the pointed data through the network. However, such a global address space directly contradicts the semantics of RPC, and couples the RPC caller and callee. Microservices that share the same memory need to synchronize with each other to keep data consistent. This requires additional logic in the microservice to handle the sharing, complicating the development of the microservices.

Thus, instead of using RPC directly, some domain-specific applications tend to be built on top of specialized frameworks [50], [52], [57], such as Apache Spark [73] for data processing, and Distributed Pytorch [45] for machine learning. The framework is usually integrated with an in-memory data store service. Instead of directly transferring large data in RPC, the caller process copies the data to the data store service and sends the returned reference. When a remote callee process needs to access the data, it uses the reference to fetch the copied data from the caller's data store to its own data store through the network. The copied data in the callee's data store is unchangeable, it has to copy again from the data store to the process's heap space. The two copies eliminate the need to handle data consistency issues. Wang et al. [65] propose to introduce this mechanism to general RPC. However, the two extra copies and inefficient communication with the data store service result in poor performance. Adopting this mechanism to datacenter RPC is unacceptable due to the

[†]Equal contribution

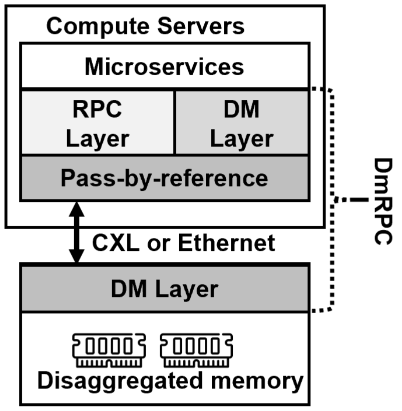


Fig. 1: Architecture of DmRPC

datacenter microservices' requirements for high throughput and low latency [35], [50].

Fortunately, modern datacenters are embracing memory disaggregation. Memory disaggregation allows computing servers to be paired with different amounts of disaggregated memory (DM) flexibly according to their demands, thus improving total memory utilization. Many mainstream datacenters are developing their memory disaggregation prototype, for example, Microsoft Azure [43], Meta [47]. With CXL, memory disaggregation in production is thought to be coming in the near future. We observe that a memory disaggregation datacenter has the potential to enable RPC to enjoy the benefits of $pass\ by\ reference$ without sacrificing programming simplicity and performance. However, prior works [6], [22], [32], [53], [62], [71] on DM all require complicated synchronization primitives to support sharing data across physical machines, thus simply adopting them to datacenter RPC would harm microservices' agility and modularity.

To this end, we propose DmRPC, a DM-aware datacenter RPC for data-intensive datacenter applications. First, it provides a DM-based shared global address space, enabling $pass\ by\ reference$ semantics for large object transfer in RPC, thus alleviating the problem of redundant data movement in the nested RPC calls. Meanwhile, it adopts $pass\ by\ value$ semantics for small object transfer like prior RPC systems, to avoid memory management overhead. Second, DmRPC adopts a `copy-on-write` mechanism to delay the copy to when necessary and reduce the copy amount by copying at the granularity of pages. The `copy-on-write` mechanism eliminates the need to handle data consistency in the microservice logic, thus avoiding complicating the development and deployment of microservices.

Figure 1 shows the overall architecture of DmRPC. We have applied DmRPC to two DM implementations, one is network-based (DmRPC-net) and the other is CXL-based (DmRPC-CXL). We use a synthetic 7-tier microservices application and a widely used microservice benchmark (i.e., DeathStarBench [26]) to evaluate DmRPC. In DeathStarBench, DmRPC-net achieves 3.1$\times$ higher throughput and 2.5$\times$ lower average latency than the baseline. In summary, we make three key contributions.

1) We present a DM-aware datacenter RPC for data-intensive datacenter applications.
2) We present DM-based $pass\ by\ reference$ semantics to address the problem of redundant data movement.
3) We present a `copy-on-write` mechanism along with DM to provide programming simplicity for RPC without sacrificing performance.

II. Background

A. Datacenter RPC

Modern datacenter applications are decomposed into deep hierarchies of microservices [19]. Compared with monolithic design, the microservice-based design provides the benefits of agility, modularity, and scalability.

Microservices communicate with each other using Remote Procedure Call (RPC), which has been a key building block of distributed systems ever since the early 80s [14], [59]. Datacenter RPC has been designed to saturate the network bandwidth in the datacenter infrastructure (tens or hundreds of Gbps) and provide low latency (several microseconds) [35], [37]. Microservices choose RPC as the communication layer mostly because of its simplicity. There is no shared state between an RPC caller and an RPC callee. The arguments and return values in RPC are $passed\ by\ value$, which completely hides the details among callers and callees. This design greatly simplifies the development and deployment of datacenter applications without sacrificing microservices programming model's agility and modularity.

B. Disaggregated Memory

Memory disaggregation can be categorized into network-based and CXL-based types.

1) Network-based Memory Disaggregation: Network-based memory disaggregation connects compute servers with memory servers using network interconnect. The compute server can access the memory in the remote servers through RDMA, TCP/IP, or other network protocols. The network latency of RDMA and user-space networking (powered by DPDK [21]) can be as low as several microseconds. As such, the latency penalty of accessing remote memory can be greatly amortized compared with traditional kernel TCP's milliseconds latency. According to the granularity of the memory, there are two types of network-based memory disaggregation.

Page-based memory disaggregation. Page-based memory disaggregation [8], [10], [31], [32], [42] manages remote memory at the granularity of page. Part of works [8], [10], [31] leverage virtual memory systems to intercept paging requests underneath the kernel swap daemon ($kswapd$). When there is a page fault, it evicts recently unused local pages and fetches remote-needed pages. In this way, the remote memory is transparent to user applications. Other prior works [11], [32] use explicit user APIs to write modified local data to the remote memory node or read remote data to local memory.

Object-based memory disaggregation. Rather than tying swapping to virtual memory abstraction of pages, AIFM [58]

TABLE I: Comparisons of different data sharing methods

Approaches	Sharing Semantics	Performance	Mutability	Programming
Traditional RPC [1], [37], [59]	Pass-by-value	Low	Mutable	Simple
DSM Model [6], [16], [22], [32], [44], [53], [54], [62], [71]	Pass-by-reference	High	Mutable	Complex
Distributed In-Memory Data Store [50], [73]	Pass-by-reference	Low	Immutable	Simple
DmRPC (Ours)	Pass-by-reference	High	Mutable	Simple

ties swapping to individual application-level memory objects. Compared with page-based swapping, object-based swapping is more fine-grained and does not suffer from IO amplification when accessing small objects. The disadvantage is that users have to use provided primitives in their application code to ensure that evacuation handlers can evacuate unused objects correctly and efficiently.

2) *CXL-based Memory Disaggregation:* Both page-based and object-based memory disaggregation exchange data through the network, thus the performance could be limited by network performance. For example, its access latency through the network usually reaches several to hundreds of microseconds. Instead, the emerging Compute Express Link (CXL) provides a completely different approach. With CXL, hosts and DM are connected through CXL links, which build upon the physical and electrical interfaces of PCIe 5.0/6.0. The CXL memory appears like a CPU-less NUMA node, hosts can access the CXL memory through sheer load/store instructions, and the access latency can be as low as hundreds of nanoseconds [43], [47], [60].

CXL memory. There are two types of CXL memory that can be exposed to multiple hosts. The first is called LD-FAM (Logical Devices Fabric-Attached Memory). LD-FAM partitions a physical CXL memory device into up to 16 logical devices. Each logical device can be exposed to a host with a separate Device Physical Address (DPA). The second is called G-FAM (Global Fabric-Attached Memory). G-FAM provides a highly scalable memory resource accessible by all hosts ($\sim$100s to $\sim$1000s) within a CXL fabric. G-FAM has only one DPA space that is common across all hosts. To create shared memory, different hosts allocate a contiguous address range of their physical address space, this address range is mapped to the DPA of the G-FAM. G-FAM device would be in charge of the address translation from each host's physical address to DPA. In this way, different hosts can access the same content in G-FAM.

III. Motivation and Trends

A. Issues of Traditional Global Address Space

Introducing a global address space for datacenter RPC can alleviate the problem of redundant data movement through the network. The microservices call RPC to send the reference of the data to a remote microservice. Once the remote microservice needs the data, it uses the reference to fetch the local data. The reference is transferred in the nested RPC calls on behalf of the large data itself, thus alleviating the problem of redundant data movement across the network. There are mainly two approaches to implementing a global address space for datacenter RPC. And Table I summarizes their characteristics.

Distributed shared memory. Distributed shared memory (DSM) [12], [16], [38], [44], [54] provides the illusion of a single globally shared address space across physically distributed machines [54]. However, introducing DSM to general RPC has been proven impractical [65], because adopting the DSM model would greatly harm the agility and modularity of microservices. In particular, introducing DSM into datacenter RPCs significantly complicates the microservice logic and couples the caller and callee. Microservices logic has to handle shared memory management and complicated data consistency. And that is why the DSM model has not been adopted by any datacenter RPC.

Distributed in-memory data store. Ray [50], [66] integrates a distributed in-memory data store service. When transferring large data, the caller copies the entire data into the data store service and a shared reference is returned. Then caller passes the reference to a remote node through an RPC call. When needed, the remote node communicates with the caller's data store service, using the reference to retrieve the entire copy to its local data store. Instead of directly using this immutable copy, this copy must be copied from its local data store to the user's heap. Sharing an immutable data copy greatly simplifies the user logic, each user does not need to handle data consistency. Wang et al. [65] propose that general RPC should also be integrated with such a distributed data store to provide the semantics of $pass\ by\ reference$. However, we identify that adopting this solution for datacenter RPC incurs poor performance due to two main reasons. First, the caller has to copy the data to the data store service. The callee fetches the data from the caller's data store service to its local data store service through the network and then has to copy the data from the local data store service to the user's heap space. The introduced communication and two additional copies are inefficient. Second, even if the callee only needs to access a small portion of a shared copy, the entire copy has to be transferred from the caller to the callee. Our evaluation result in Section VI-D indicates that DmRPC can provide 21.6$\times$/5.6$\times$ than Spark/Ray when transferring 32KB raw data blocks between two servers (each equipped with a 100 GbE NIC) using a single thread.

B. DM-based Global Address Space

There is one important trend that motivates this paper. That is, the modern datacenter is actively embracing disaggregated memory. With the release of the CXL standard, mainstream datacenters [43], [47] have started early deployment of DM. From hardware to software vendors, memory disaggregation is thought to be a standard facility of the datacenter in the near future. We argue that DM should help introduce a global address space for datacenter RPC while addressing the issues of the DSM model and distributed in-memory data store. However, directly adopting current DM solutions can not address these issues.

Limitation of existing DM solutions. Many of the prior works on DM [8], [10], [31], [58] do not allow memory sharing across different processes, while others [6], [22], [32], [53], [62], [71] support sharing memory adopting a DSM-like model. This requires the process to handle data consistency, thus greatly complicating the microservice logic. For example, Remote Region [6] provides barriers, mutexes, and doorbells. Clio [32] uses $rlock$, $runlock$, and $rfence$ to synchronize data. FaRM [22] uses distributed transactions to manage data consistency. Adopting these approaches to microservices significantly complicates user logic, which is unacceptable for microservices.

IV. Overview of DmRPC

To address the problem mentioned in Section III, we propose DmRPC, a DM-aware datacenter RPC that supports a global address space without sacrificing programming simplicity or performance.

A. Design Goals

There are three main design goals of DmRPC.

G1: Minimizing redundant data movement along network when using datacenter RPC. Deeply nested datacenter RPC incurs redundant data movement along the network, especially for large RPC arguments. Instead of $pass\ by\ value$, DmRPC shall provide a $pass\ by\ reference$ mechanism to reduce redundant data movement.

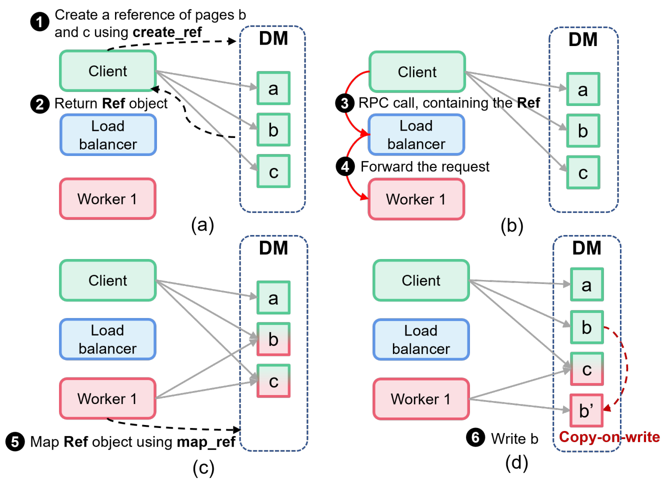
G2: Abstracting complex user logic away from handling data consistency. Traditional distributed shared memory (DSM) model [12], [38], [44], [54], [65] exposes too many underlying details such as consistency to users, which hinders the agility and modularity of microservices. Our design intends to provide $pass\ by\ reference$ semantics while keeping high programmability.

G3: Introducing a trivial performance overhead. The performance of datacenter RPC directly determines the overall application performance [41]. It is unacceptable to sacrifice nontrivial performance for programming simplicity.

B. Design Overview

Figure 1 shows the architecture overview of DmRPC. DmRPC targets DM-enabled datacenter, it intends to provide a global address space to accelerate data-intensive RPCs while abstracting user logic away from handling data consistency and providing high performance. Microservices running in different compute servers communicate with each other using RPC. Each microservice can access the disaggregated memory through the CXL link or Ethernet, according to the type of memory disaggregation. To minimize redundant data movement, it adopts a $pass\ by\ reference$ mechanism to enable microservices to share DM region without complicating the application logic or sacrificing performance.

DmRPC is orthogonal to the prior works on disaggregated memory. We have applied DmRPC to two different implementations of DM: DmRPC-net and DmRPC-CXL. DmRPC-net is based on the network and adopts a non-transparent interface where applications access disaggregated memory via explicit API calls (i.e., $rread$, $rwrite$). DmRPC-CXL is based on CXL and applications access disaggregated memory through naive $load$ and $store$ instructions.



1 Create a reference of pages b and c using create_ref
2 Return Ref object
Client
Load balancer
Worker 1
DM
a
b
c
(a)
3 RPC call, containing the Ref
4 Forward the request
(b)
5 Map Ref object using map_ref
(c)
6 Write b
b'
Copy-on-write
(d)

Fig. 2: An example of a high-level view of DmRPC.

To achieve **G1**, we introduce a DM-aware global address space to datacenter RPC. When a microservice intends to share a large data in a DM region, instead of sending the data in an RPC call, it uses the RPC to send a reference (called Ref) of the memory region. The Ref object is small (several bytes), and is transferred along the RPC chain on behalf of the large data. The $pass\ by\ reference$ semantics greatly reduce redundant data movement. At the time a microservice needs to access data, it uses the Ref object to map the shared disaggregated memory to its virtual address space, thus manipulating the data.

A naive solution to achieve **G2** is to copy the entire data, and then only allow the remote microservice to access the copy. In this way, there is no need for different microservices to handle data consistency issues because each of them owns a different memory region. However, the copy could incur performance degradation, dissatisfying **G3**. To resolve this, we design a `copy-on-write` layer on top of the DM. The `copy-on-write` layer alleviates the performance degradation in two ways. First, it delays the actual copy to the time when a microservice writes the shared memory region, avoiding data copy for `read-only` accesses. Second, the `copy-on-write` is at the granularity of pages. Pages that have not been written would not be copied, further reducing the copying overhead. In this way, we achieve **G2** and **G3**, abstracting microservices logic away from handling data consistency, more importantly, without sacrificing performance.

Figure 2 gives a high-level view of DmRPC mechanism to show how DmRPC works.

(a) Microservice *Client* calls *create_ref* (❶), then a *Ref* object pointing to these pages would be returned (❷).
(b) *Client* sends the *Ref* object to *Load balancer* using an RPC call (❸). The *Load balancer* forwards the request to one unloaded worker, i.e., *Worker* 1 (❹).
(c) *Worker* 1 calls *map_ref* to map the remote contents to its virtual address (❺). Now pages b and c are shared between *Client* and *Worker* 1, and both of them can read pages b and c.
(d) When one microservice writes to the shared page (❻), assuming it's *Worker* 1, a `copy-on-write` would be triggered and the content of the page would be copied to a new page. The write would be applied to the new page, and the virtual address of the *Worker* 1 would point to the new page. The `copy-on-write` logically guarantees each microservice owns a private memory region.

Size-aware transfer. *pass by reference* avoids large data transfer in data-intensive RPCs. However, DmRPC manage data at the granularity of pages, creating reference for small data objects may overwhelm the system. As such, DmRPC only uses *pass by reference* semantics for large objects, and uses *pass by value* semantics for small objects. DmRPC would automatically choose the appropriate mode based on the parameter object size, while users are not aware of the two different modes.

TABLE II: Programming APIs

ralloc(size)	Allocating *size* bytes disaggregated memory, and return a *remote_addr*
rfree(remote_addr)	Deallocates *remote_addr*.
create_ref(remote_addr, size)	Return an *Ref* object pointing to the associated pages. The memory region would be marked as `read-only`, any writes would trigger `copy-on-write`.
map_ref(ref)	Map the pages pointed by *Ref* to a DM virtual address and return the *remote_addr*.
rwrite(remote_addr, local_addr, size)	Write *size* bytes data of the local memory *local_addr* to the disaggregated memory *remote_addr*. This function is specific to DmRPC-net. In DmRPC-CXL, the user can directly operate on the disaggregated memory.
rread(remote_addr, local_addr, size)	Read *size* bytes data from the disaggregated memory *remote_addr* to local memory *local_addr*. Similar to *rwrite*, *rread* is specific to DmRPC-net.

C. Application Programming Interface

Table II demonstrates the APIs of DmRPC. Both DmRPC-net and DmRPC-cxl use the same APIs for allocating, freeing, and sharing disaggregated memory. The difference is how they read/write the disaggregated memory, through explicit APIs or naive *load/store* instructions. *rwrite* and *rread* only appear

```
1  /* @Client microservice */
2  int *r_addr = (int*)ralloc(len*sizeof(int));
3  //Fill the disaggregated memory
4  rwrite(r_addr,local_buf,len*sizeof(int));
5  //Create a reference of r_addr
6  Ref ref = create_ref(r_addr,len*sizeof(int));
7  RPC_LB(ref); //Call load balancer microservice
8  rfree(r_addr);
9
10 /* @Load balancer microservice */
11 //Forwards requests without touching arguments
12 RPC_LB(Ref ref){
13   if(worker_1_is_idle){
14     RPC_Worker_1(ref); //Call Worker 1's microservice
15   }else{
16     RPC_Worker_2(ref); //Call Worker 2's microservice
17   }
18 }
19
20 /* @Worker 1 microservice */
21 RPC_Worker_1(Ref ref){
22   //Map ref to local virtual address that maps to DM
23   int* r_addr = (int*)map_ref(ref);
24   //Read from DM to local buffer
25   rread(r_addr,local_buf,len);
26   //Working on local memory: aggregating the content
27   int sum = 0;
28   for(int i=0;i<len;i++){
29     sum += local_buf[i];
30   }
31   rfree(r_addr);
32 }
33
34 /* @Worker 2 microservice */
35 RPC_Worker_2(Ref ref){
36 ...
37 }
```

Listing 1: An example of programming with DmRPC

in DmRPC-net, other APIs appear in both DmRPC-net and DmRPC-CXL.

Listing 1 is a programming example of using DmRPC:

Client. The *Client* first allocates a disaggregated memory buffer (Line 2) and fills the buffer (Line 4). Then it calls *create_ref* (Line 6). This would mark this memory buffer as a `read-only` region. It then returns a *Ref* object pointing to the `read-only` region. Instead of sending the actual data to the remote microservice *Load balancer*, *Client* only sends the *Ref* object to the *Load balancer* using an RPC call (Line 7).

Load balancer. *Load balancer* is in charge of forwarding the requests to remote workers. If *Worker* 1 is idle, it forwards the request to *Worker* 1 by calling *RPC_Worker_1* (Lines 13-14). Otherwise, it forwards the request to *Worker* 2. Note that *Load balancer* would not touch the data in the DM buffer, it simply forwards the *Ref* object, saving both network bandwidth and memory bandwidth of the server that *Load balancer* runs on.

Workers. Workers simply aggregate the requested data. It first calls *map_ref* to map the remote buffer to its virtual address (Line 23). After that, the worker can use the virtual address to aggregate the DM buffer (Lines 26-29).

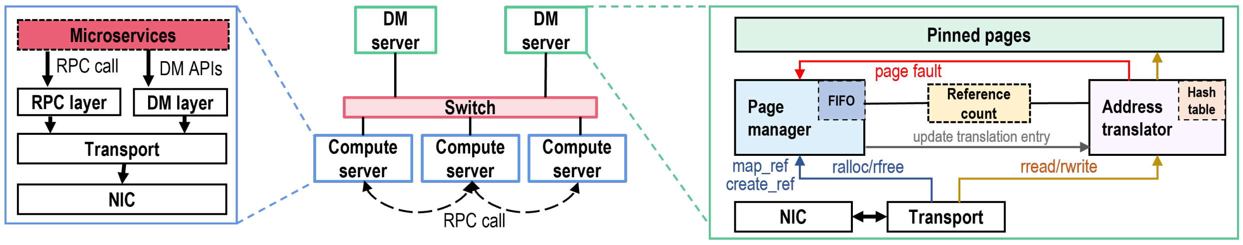


Fig. 3: DmRPC-net architecture

V. Design

A. DmRPC-net

Figure 3 shows the architecture of DmRPC-net. Both compute servers and DM servers are regular servers. Each server is equipped with an Ethernet NIC and connected to a top-of-rack (ToR) switch.

Generally, microservices processes running at compute servers communicate with each other through $RPC\ layer$, a modified C/C++ RPC library based on prior work eRPC [37]. Microservices running in the compute servers can manipulate disaggregated memory in the DM servers through $DM\ layer$. Both RPC layer and DM layer are based on DPDK [21]. Our networking protocol is founded upon the UDP and the network reliability is handled in the RPC layer just like eRPC [37]. Apart from the local virtual memory address space, each process has its own remote virtual address space. Each process has a unique global PID across all compute servers and the global PID is assigned by our software running on DM servers. The combination of the global PID and the per-process remote virtual address is called DM virtual address, which can be used to address a unique disaggregated memory region in the DM servers.

The DM server in DmRPC-net is in charge of serving the requests from the microservices running at compute servers. The DM server software runs on top of Linux, the software architecture of the DM server consists of two components: $Page\ manager$ and $Address\ translation$. In the following, we present the design details of each component.

1) Page Manager: $Page\ manager$ is in charge of managing the memory in the DM server and the DM virtual address range. At the initialization time, $Page\ manager$ allocates a large memory and pins it. The pinned memory is used as the disaggregated memory. $Page\ manager$ manages the pinned memory at the granularity of pages (the page size is changeable, 4 KB in our case). It manages the pinned pages in a $FIFO$. $Page\ manager$ uses reference counting to track the number of processes sharing the page. Each pinned page has a 4-byte integer that records the reference count. These reference counts are stored linearly in the memory. At the time a page is mapped to a virtual address, its reference count value is initialized to one. In the following, we detailedly discuss how to handle $ralloc/rfree/create\_ref/map\_ref$.

ralloc. For each process leveraging the DM, $Page\ manager$ maintains a VA allocation tree that records allocated VA ranges, similar to the Linux vma tree [51]. When a process allocates a new DM buffer, $Page\ manager$ looks up the VA tree to find an unused virtual memory address and returns it to the process. When the process first writes to a DM virtual address, a page fault would be triggered. Then $Page\ manager$ would pop an unused page from the $FIFO$ and maps the page to the DM virtual address that triggers the page fault.

rfree. When a microservice releases the memory by calling $rfree$, $Page\ manager$ notifies $Address\ translator$ to clear the translation entries in $hash\ table$. Then $Page\ manager$ looks up the corresponding reference count. If the value is one, $Page\ manager$ pushes the freed page to the $FIFO$. Otherwise, it just decrements the reference count by one.

create_ref. After receiving the request, $Page\ manager$ adds the reference counts of the corresponding pages by one. Then $Page\ manager$ generates a unique key and constructs a key-value tuple, the value stores the pinned virtual addresses of pages to be shared. The key-value tuple would be stored in a map. At last, $Page\ manager$ returns an Ref object, containing the key and the identifier of the DM server.

map_ref. After receiving the request, $Page\ manager$ uses the key in the Ref object to look up the map and gets the pinned virtual addresses of the shared pages. Then it allocates an unused DM virtual address and maps it to the shared pages, notifying $Address\ translator$ to store the translation entries in the $hash\ table$. At last, $Page\ manager$ returns the DM virtual address to the microservice.

2) Address Translator: In DmRPC-net, there are two address translations for each DM access.

1, Software-based translation. The first translation is handled by $Address\ translator$ software and it translates DM virtual address to the virtual address of the pinned memory. All processes' translation entries are stored in a single in-memory $hash\ table$. If there is no required translation entry in the $hash\ table$, $Address\ translator$ would notify $Page\ manager$ to handle the page fault.

2, MMU-based translation. The second translation is implicit and done by the memory management unit (MMU). It translates the pinned memory's virtual address to the physical address.

Since all the DM memory is pinned, OS-level page faults would not be triggered during the two translations. The overhead of the software-based address translation is minor compared with the network transfer time. Our evaluation shows that the first software-based translation only accounts for 0.17% of the total DM access time.

It is also possible to skip the software-based translation by modifying OS and letting MMU translate the DM virtual address directly to the physical address. We leave this improvement for future work.

How to serve a read request. After receiving a read request, $Address\ translator$ looks up the $hash\ table$ to retrieve the virtual address of the pinned pages. Then it directly returns the content in the pinned pages without checking the reference count.

How to serve a write request. After receiving a write request, $Address\ translator$ first looks up the $hash\ table$ to retrieve the virtual address of the pinned pages. Then $Address\ translator$ reads the reference count of the pinned pages. If the reference count value is larger than one (i.e., the page is shared by other processes), a page fault occurs. $Page\ manager$ pops a new page from the $FIFO$ and copies the content of the old page to the new page. The write request would be applied to the new page. Then $Page\ manager$ minus the reference count of the old page by one. At last, $Address\ translator$ updates the corresponding entry in the $hash\ table$, mapping the DM virtual address to the new page's virtual address.

B. DmRPC-CXL

DmRPC-CXL is based on CXL 3.0 specification [18]. Different from DmRPC-net, processes in DmRPC-CXL can use naive $load/store$ instructions to access data in the DM through the CXL link.

Design challenge. Design of DmRPC-CXL raise a new challenge: CXL memory does not have processing power. In DmRPC-net, we implement a centralized `copy-on-write` layer in the DM server. However, the CXL memory has no such processing power itself.

Fortunately, CXL 3.0 specification allows each host to perform arbitrary ISA-supported atomic operations on its connected CXL memory [18]. We leverage this feature to implement a distributed `copy-on-write` layer. Figure 4 shows the architecture of DmRPC-CXL. Each compute server is connected to an Ethernet switch and a CXL switch. Microservices communicate with each other through the RPC layer on top of the Ethernet network. And microservices can leverage the CXL memory through the CXL switch.

1) CXL Memory Management: DmRPC-CXL is based on G-FAM (details in Section II-B2). Each computer server linearly maps a contiguous range of its physical address to the DPA of the G-FAM device. The majority of the physical memory of a G-FAM device is used as CXL physical pages, while the remaining memory records the reference count of these CXL physical pages. These physical pages and reference counts are visible to all hosts in the fabric.

Coordinator server. There is a coordinator server in the fabric, which is in charge of managing the ownership of all CXL physical pages among all compute servers. It communicates with compute servers using a reliable network protocol. Each computer server reserves the ownership of some free CXL physical pages, when the number of free pages is less than a threshold, the compute server requests more free pages from the coordinator server. Similarly, when the number of free pages is larger than a threshold, the compute server gives back the ownership of free pages to the coordinator server. Reserving free pages in compute servers significantly reduces the overhead of coordinating the ownership of CXL physical pages.

The DM layer in each compute server is in charge of managing the CXL physical pages it owns, allocating/freeing CXL memory, handling page faults, translating virtual address to CXL physical address, and handling `copy-on-write`. The DM layer mainly runs in the kernel space.

2) CXL Page Fault Handler and Address Translation: The CXL page fault handler manages all the free CXL physical pages it owns using a $FIFO$.

Memory allocation. When a process allocates a CXL memory buffer, the DM layer finds an unused virtual address range from the VMA tree. Then the DM layer sets an unused bit of vm_flags in the vm_area_struct, indicating that this virtual address should be mapped to CXL memory (we call this type of virtual address as CXL virtual address). At last, the DM layer returns the CXL virtual address to the process. At this time, no CXL physical pages are mapped to this virtual address.

Address translation. There are two address translations in DmRPC-CXL. The first translation translates CXL virtual address to CXL physical address and is done by the MMU in the compute server. The page table entries of this translation relation are written in the same page table storing regular page table entries. Such that this translation can be offloaded to the MMU of the compute server. The second translation translates CXL physical address to the device physical address (DPA) of the CXL memory device and is done by the CXL device itself.

3) Copy-on-write: The `copy-on-write` in DmRPC-CXL relies on two data structures. The first is the permission flags of a page [2], stored in each page table entry. The permission flags can indicate whether the page is a `read-only` page. The second is the reference count of each CXL physical page. When a CXL physical page is mapped to a CXL virtual address, its $ref\ count$ would be initialized to one. If there are not enough free CXL physical pages in the $FIFO$ when handling page faults, the DM layer would request some free pages from the coordinator server.

create_ref. After receiving a request from the process, the DM layer would add the reference counts of corresponding CXL physical pages by one using an atomic manner. Then the DM layer marks these pages as `read-only` by modifying corresponding page table entries. After that, the DM layer returns all physical pages' addresses as a reference.

map_ref. After receiving a request, the DM layer would find an unused CXL virtual address from the vma tree, mapping the CXL virtual address to the physical address of the CXL pages contained in the reference. The DM layer inserts these mappings into the page table, setting these pages as `read-only`. Then the DM layer returns the CXL virtual

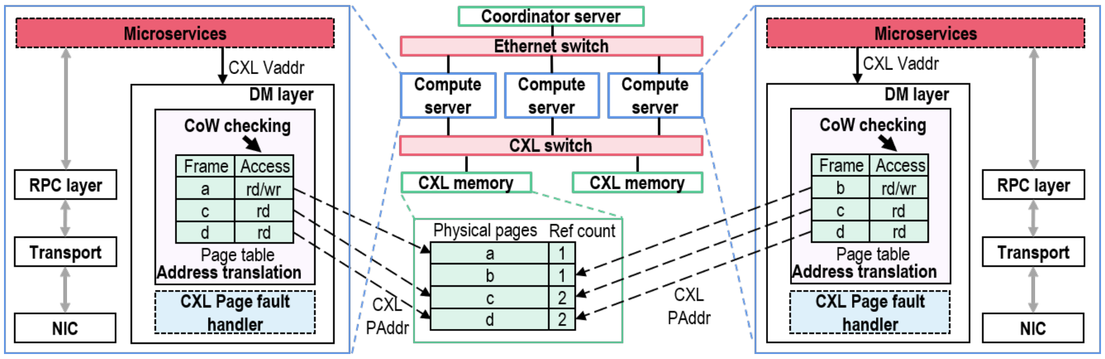


Fig. 4: DmRPC-CXL architecture

address to the microservice process. The process can now access the shared memory using this CXL virtual address.

How to serve a *load* operation. Serving a *load* operation on CXL memory is completely the same as regular memory. First, the MMU translates the CXL virtual address to CXL physical address. Second, the physical address is used to retrieve the data from the cache system, or the memory if there is a cache miss.

How to serve a *store* operation. In the beginning, the MMU would translate the CXL virtual address to CXL physical address. There are three possible situations.

1) If the virtual address has not been mapped with a CXL physical page, a page fault occurs. The DM layer then pops an unused CXL physical page from the $FIFO$, mapping it to the CXL virtual address.
2) If the page is `read-only`, a page fault occurs, the DM layer would look up the $ref\ count$ of the CXL physical page. If the reference count is larger than one, a `copy-on-write` would be triggered. The DM layer allocates a new CXL physical page and copies the content of the old page to the new page. Then the DM layer updates the page table entry, pointing the CXL virtual address to the new CXL physical page and marking the page as "writable and readable" by modifying the permission flags in the page table entry. At last, the DM layer minuses the reference count of the old CXL physical page by one using an atomic instruction. If the reference count is one, the DM layer simply modifies the permission flags of the page table entry to "writable and readable". The write would be applied to the CXL physical page to which the page table entry points.
3) If the page is "writable and readable", no page fault occurs. The write would be applied to the pointed CXL physical page.

Memory release. When a microservice releases the shared CXL memory by calling $rfree$, the DM layer first deletes the corresponding local page table entry. Then it looks up the corresponding $ref\ count$. If the value is one, the DM layer pushes the freed physical pages to the $FIFO$. Otherwise, the DM layer simply minus the $ref\ count$ by one using an atomic manner. In DmRPC-CXL, a shared CXL physical page is owned by many processes on different compute servers. The process that frees the page lastly is in charge of the reclamation of this page. The DM layer would add the freed CXL physical pages to the $FIFO$. If the number of free pages in the $FIFO$ is larger than a threshold, the compute server would return ownership of some free pages to the coordinator server.

In DmRPC-CXL, most accesses to CXL memory would not incur additional overhead and can leverage the cache system in the compute server. Only the first write to a `copy-on-write` page would require additional memory access (reference count). This merely degrades the performance of accessing CXL memory.

VI. Experimental Evaluation

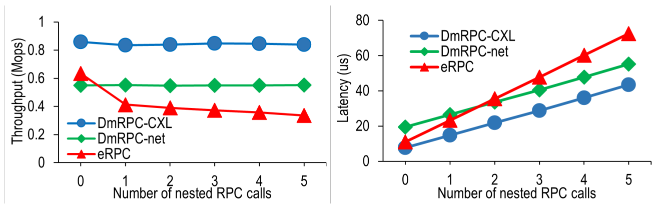
A. Experimental Setup

Our experimental platform consists of eight AMAX XP04A201G servers. Each has two 24-core 2.2 GHz Xeon Silver 4214 CPUs, 128 GiB (4x32 GiB) 2400 MHz DDR4 memory. Each server is equipped with a 100 GbE Mellanox ConnectX-5 NIC [48]. They run Ubuntu 18.04 (Linux 4.15.0-20-generic) with hyper-threading.

eRPC baseline. This baseline employs the state-of-the-art RPC library eRPC [37] to allow microservices to communicate with each other in different servers. eRPC is a general-purpose RPC library that runs on commodity CPUs and traditional datacenter networks based on either lossy Ethernet or lossless fabrics. The microservices use $pass\ by\ value$ semantics to send large data to each other.

DmRPC-net. DmRPC-net uses eRPC [37] as the communication layer, the difference is that microservices can leverage disaggregated memory in the DM server using our $DM\ lib$. The microservices call RPC to send the Ref object of the large data instead of the large data itself. We implement the global disaggregated memory pool using two servers. When each microservice needs to allocate a remote memory buffer, its allocation request would be forwarded to one of the memory servers in a round-robin manner.

DmRPC-CXL. Currently, there is no commodity CXL switch/retimer to build a CXL-based memory pool. To emulate a CXL-based memory pooling system, we need to emulate the higher latency from both the CXL memory's higher latency and the CXL switch's latency. Recent work [60] uses a real



(a) Throughput (b) Latency

Fig. 5: Performance under the different numbers of nested RPC calls.

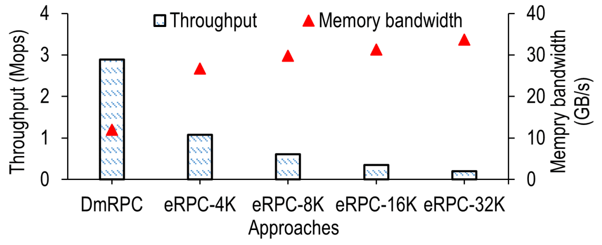


Fig. 6: Performance of load balancer

experimental FPGA-based CXL memory to measure the CXL memory access latency, which is about 2.2× higher than the local socket DDR access, where the local DDR latency is around 75 nanoseconds [33], [67]. The authors also claim that future commercial CXL memory access latency would be lower since their current long access latency can be partly attributed to its FPGA implementation. However, our CXL emulation experiment still uses this overestimated 2.2× value as the expected CXL latency (165 ns).

The next step is to emulate the CXL switch latency. Prior work [43] designs a CXL external memory controller and emulates a 32 sockets memory pool, the introduced switching latency is around 87 nanoseconds. A recent industrial presentation [3] also indicates that the CXL switch would introduce around 100 nanoseconds of extra latency. As such, we use 165 ns as the CXL memory latency and 100 ns as the CXL switch latency. The memory latency of the emulated CXL memory pool should be around 265 ns. We use a two-socket server in the emulation. To increase the memory access latency, we let all threads run in the same socket and only allocate memory from another socket. However, cross-socket memory latency is around 125 ns, which is still far lower than 265 ns. As such, we manually reduce the server uncore frequency (from 2400 MHz to 800 MHz) which could increase the memory access latency to around 265 ns. All the DmRPC-CXL results in this work are measured under the 265 ns memory latency.

B. Effect of DM-aware Global Address Space

We examine the effect of our proposed DM-aware global address space, which provides $pass\ by\ reference$ semantics to reduce redundant data movement along the network. we introduce two applications to validate the benefits of global address space.

Nested RPC calls. In the nested RPC call application, the client calls an RPC with a 4 KB size array as the argument, and the called microservice directly passes the array to the next microservice without using it. After several repeated RPC calls, the final microservice aggregates the array and returns the result. Each microservice uses a single thread.

Figure 5a illustrates the achieved throughput with different chain lengths. We have three observations. First, DmRPC-net's throughput is higher than that of eRPC (except for only 1 RPC call), because eRPC introduces unnecessary redundant memory copies between microservices while DmRPC-net does not due to its DM-aware global address space. Second, DmRPC-CXL achieves higher throughput than the other two implementations. This is because the last microservice in DmRPC-CXL accesses the shared data through the CXL link, instead of fetching data through the network. Compared with the network link, the CXL link provides much lower latency and higher throughput. Third, the throughput of DmRPC-net and DmRPC-CXL merely change over the number of nested RPC calls because both pass reference between middle RPC calls due to its DM-aware global address space, while eRPC's throughput decreases when the number of nested RPC calls increases because eRPC uses $pass\ by\ value$ semantics between any two RPC calls.

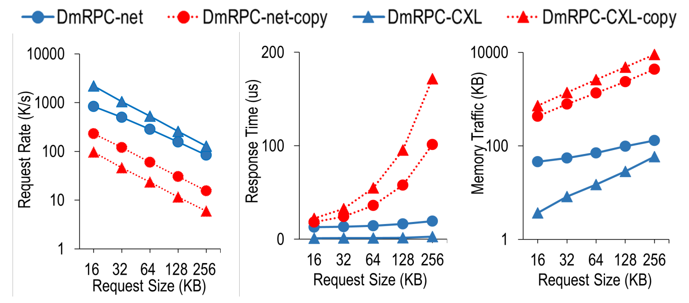
Figure 5b illustrates the average latency of different approaches. We observe that DmRPC-CXL and DmRPC-net have lower latency than eRPC, when the number of nested RPC calls is larger than 1, indicating the efficiency of DM-aware global address space that enables $pass\ by\ reference$ semantics.

Application-layer load balancer (LB). Application-layer LB forwards the requests with arguments to the unloaded server. We build an LB microservice that simply forwards the incoming requests with arguments to the unloaded remote servers through RPC calls. We use three servers to generate requests to the LB server, one server as the LB server that forwards the requests, and three servers to receive the forwarded requests. The increased network bandwidth introduces bottlenecks to the CPU, memory, and PCIe interconnect of machines that run data mover applications [56].

Figure 6 demonstrates the throughput and memory bandwidth occupation on the load balance server. We observe that DmRPC delivers the highest throughput, and occupies the lowest memory bandwidth, because DmRPC only forwards references, rather than data, in the LB server due to its DM-aware global address space, and eRPC forwards data (4K to 32K) in the LB server. The results indicate that DM-aware global address space can 1) alleviate the memory bandwidth pressure on LB servers that run data mover applications; and 2) achieve a higher request rate on these servers.

C. Effect of the Copy-on-write Mechanism

A naive approach to avoid complicating microservice logic would be sharing a copy of the original data, thus decoupling the caller and the callee. However, implementations ended with $-copy$ adopt the unconditional copy approach to bring much memory bandwidth pressure to DM, because these



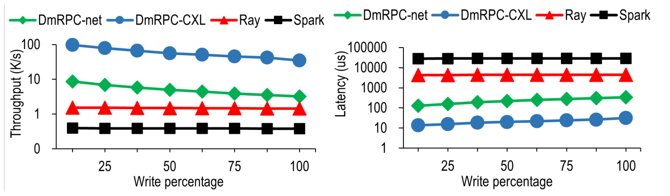
(a) Achieved request rate (b) Response time (c) Memory traffic for each request

Fig. 7: Effect of `copy-on-write`

implementations copy the original data between microservices. Thus implementations marked without $-copy$ adopt a `copy-on-write` mechanism to address the issue of the unconditional copy. We examine the effect of `copy-on-write` mechanism, in terms of performance of *create_ref*, where *create_ref* function call would return the *Ref* object that represents the original memory region. For DmRPC-net implementations, we use one CPU core in a single memory server and ensure the client can send requests fast enough to fully saturate the processing ability of the memory server. For DmRPC-CXL implementations, we use only one client thread.

Figure 7a and Figure 7b show the achieved request rate and response time when calling *create_ref*. We observe that DmRPC-CXL (or DmRPC-net) achieves up to 22.8× (or 7.3×) higher request rate than that of DmRPC-CXL-copy (or DmRPC-net-copy), and DmRPC-CXL (or DmRPC-net) achieves up to 63.6× (or 5.2×) lower response time than that of DmRPC-CXL-copy (or DmRPC-net-copy), this is because `copy-on-write` mechanism avoids the unnecessary copies between microservices. As evidence, we also use Intel Performance Counter Monitor (PCM) [34] to show the average disaggregated memory traffic under these implementations. Figure 7c illustrates the average memory traffic per request under different request sizes. We observe that DmRPC-CXL and DmRPC-net have significantly lower memory traffic than DmRPC-CXL-copy and DmRPC-net-copy, respectively.

For DmRPC-net, highly concurrent requests are addressed through two strategies: 1) Load-balanced distribution across multiple memory servers, where each microservice's remote memory allocation requests are currently routed in a round-robin fashion. 2) Concurrent requests received in a single memory server will be dispatched to its different CPU cores, each responsible for managing a portion of the memory. Regarding DmRPC-CXL, due to the absence of centralized computing power, concurrent requests are handled by atomic operations in the client. Different threads can operate on different memory addresses concurrently. Different threads can operate on the same memory address sequentially which is guaranteed by the CXL-supported atomic operations.



(a) Throughput (b) Latency

Fig. 8: Performance comparison with Ray/Spark

D. Comparison with Ray/Spark

In this section, we compare DmRPC with Ray and Spark, which integrate a distributed in-memory data store service. Ray is equipped with an in-memory data store service called Plasma. In Section III, we discuss the inefficiency of adopting an in-memory data store service. Spark has a similar data store service that is part of its BlockTransferService [4].

In this subsection, we compare the performance of Ray and Spark using a micro-benchmark. The caller microservice creates a reference of a large raw data block (32 KB), and then sends the reference to a remote microservice using an RPC call. The raw data block avoids the influence of data serialization. The remote microservice writes the shared data that the reference points to. We let the remote microservice write different percentages of the shared data. Figure 8a and Figure 8b show the achieved throughput and latency when using a single thread. We have two observations.

First, DmRPC-CXL (DmRPC-net) achieves up to 62× (or 5.6×) higher throughput than that of Ray and achieves up to 315× (or 34.2×) lower latency. The performance of Spark is even worse. The underlying reason is that distributed data store is inefficient for their two unconditional data copies and inefficient communication with the data store service. Besides, Ray and Spark are not specially designed for fine-grained microservices.

Second, with the increase of the write percentage, the throughput of DmRPC-CXL and DmRPC-net decreases, and their latency increases, while Ray's and Spark's throughput and latency merely change, because they unconditionally copy the original data in the caller process to callee process, while DmRPC-CXL and DmRPC-net only do one copy when a process writes to the data thanks to the `copy-on-write` mechanism. The results indicate that the DmRPC can take advantage of avoiding unnecessary data copies.

E. Evaluation on Synthetic Workload

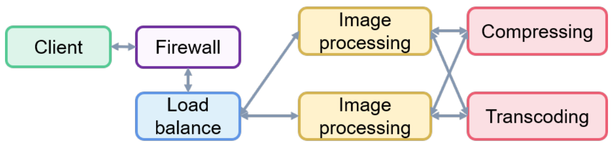
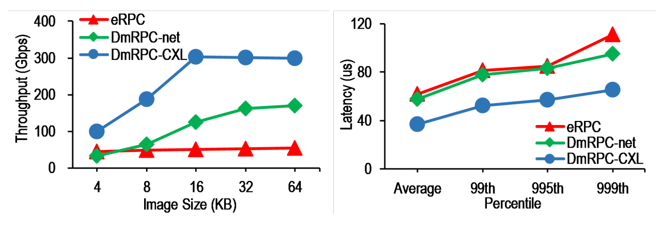


Fig. 9: Cloud image processing microservices architecture

We evaluate DmRPC on a synthetic application built with microservices. We design 7-tier microservices that implement



(a) Throughput (b) Latency

Fig. 10: End-to-end throughput and latency of different approaches

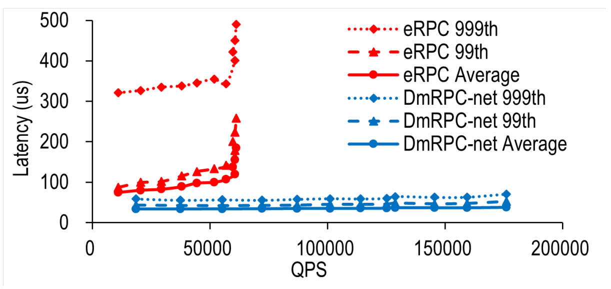


Fig. 11: DeathStarBench: average latency and 99th/999th tail latency under different request rates

a Cloud Image Processing application, shown in Figure 9. The *Client* issues image processing requests. The *Firewall* microservice checks the permission and blocks unauthorized requests. The *Load balance* forwards the requests to one of the servers that run the *Image processing* microservice using a round-robin way. The *Image processing* microservice would parse the request and call the *Transcoding* to transcode the image or call the *Compressing* to compress the image. Finally, *Image processing* returns back the processing result to *Client*.

For the eRPC baseline, we use seven servers, each microservice runs on a server. For DmRPC-net, there are two servers used as DM servers. To keep the number of servers used the same as the eRPC baseline, we allow one server to run multiple microservices if the server has spare network bandwidth and CPU cycles. Although these microservices are running on the same machine, RPC calls between them would still be forwarded to a network switch for fair comparison. For DmRPC-CXL, we further deploy the microservices that touch the image data on the same socket in the same server. To use the same number of servers as the other baselines, we scale the results of a single server to simulate the performance of multiple servers.

Figure 10a illustrates the throughput comparison of different approaches. We have two observations. First, when increasing the image size, both DmRPC-net and DmRPC-CXL can significantly increase their throughput while eRPC cannot, because DmRPC introduces global address space and `copy-on-write` mechanism to reduce unnecessary data copies. Compared with eRPC, DmRPC-net and DmRPC-CXL increase the achievable throughput by $4.2\times$ and $8.3\times$, respectively. Second, DmRPC-CXL reaches peak throughput (around 300 Gbps) when the image size is larger than 8 KB.

We observe that when increasing the image size, both DmRPC-net and DmRPC-CXL can significantly increase their throughput while eRPC cannot, because DmRPC introduces global address space and `copy-on-write` mechanism to reduce unnecessary data copies. Compared with eRPC, DmRPC-net and DmRPC-CXL increase the achievable throughput by $4.2\times$ and $8.3\times$, respectively. When the image size is large ($\geq$32 KB), the throughput of DmRPC-CXL levels off. Our further evaluation shows that the system throughput increases almost linearly with the number of used CPU cores. When all CPU cores (12 per socket) are used, the UPI [5]/network throughput in each server is far lower (1.6×/40.1×) than the available bandwidth. This indicates that the system throughput is bounded by the number of used CPU cores.

Figure 10b demonstrates the average, 99th, 99.5th, and 99.9th percentile latency of different approaches. The image size is fixed to 4 KB. We have two observations. First, DmRPC-CXL shows the lowest latency because large data is never transferred through the network. Second, DmRPC-net can slightly reduce the latency compared with eRPC. The reason is that, compared with DmRPC-net, DmRPC-CXL not only reduces redundant data movement but also enables accessing data through the CXL link, instead of the network. Compared with eRPC, DmRPC-net and DmRPC-CXL reduce the average latency by $1.1\times$ and $1.7\times$, respectively.

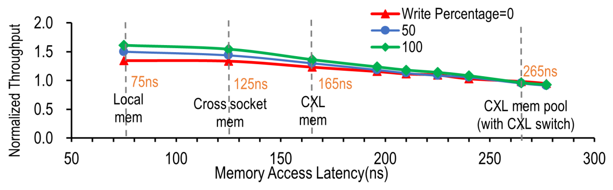
F. Evaluation on DeathStarBench

DeathStarBench [26] is a widely used open-source benchmark suite designed to evaluate the performance of microservice. DeathStarBench has three open-source applications, among which the social network framework is the most widely used [24], [25], [27], [60], [75]. We use the mixed workloads of social network framework to compare DmRPC-net to eRPC. Following the setting of prior work [60], we let the mixed workload to be consists of 60% *read-home-timeline*, 30% *read-user-timeline*, and 10% *composing-post*. The workload shows a simulated use case of a real social network, where overall, most users read the posts composed by a few users. In DeathStarBench, all requests traverse at least three data mover services (load balancer, proxy, and php-fpm), these three services just forward the request without touching the data. Traffic in *read-user-timeline* even traverses five data mover services.

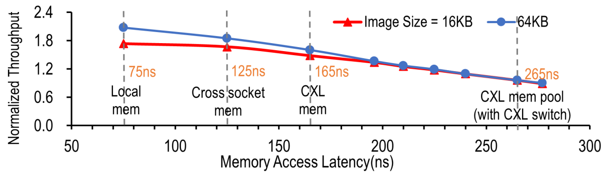
We evaluate eRPC and DmRPC-net on DeathStarBench, all microservices are deployed on three servers. Figure 11 shows the latency under different request rates. We have two observations. First, DmRPC-net's achievable request rate is $3.1\times$ higher than eRPC. Second, under the same request rate, the average latency, 99th tail latency, and 999th tail latency of DmRPC-net is much lower than that of eRPC. The main reason for the performance gain is that two proposed RPCs allow to simply forward the data without touching the data, while eRPC needs to touch the data.

G. Discussion on CXL Emulation

As mentioned in VI-A, DmRPC-CXL's performance is measured under 265ns cross-socket memory access latency. To further understand the latency influence on the DmRPC-CXL's performance, we re-run two experiments and measure the DmRPC-CXL performance under different memory access latency by tuning uncore frequency.



(a) Micro-benchmark throughput



(b) Cloud image processing throughput

Fig. 12: DmRPC-CXL normalized throughput under different memory access latency.

Figure 12a shows micro-benchmark (the same as experiment in Section VI-D) throughput of DmRPC-CXL under different memory access latency. Figure 12b shows the cloud image processing (the same as the experiment in Section VI-E) throughput of DmRPC-CXL under different memory latency.

In both experiments, we observe that the throughput slightly decreases when the memory latency increases. We believe that our chosen memory access latency (265 ns) can well emulate the performance of the future CXL memory pool (with CXL switch). Even if the future CXL switch has slightly longer access latency, we believe our main conclusion drawn from our emulation still holds.

VII. Related Work

To our knowledge, DmRPC is the first work digging into the potential benefits that disaggregated memory can bring to datacenter microservices. We contrast DmRPC and existing works in the following aspects.

Disaggregated memory. Memory disaggregation decouples computing and memory, improving the total memory utilization rate. There are mainly two categories of memory disaggregation. The first category [6], [8], [10], [13], [17], [22], [23], [31], [32], [36], [42], [53], [55], [58], [61], [62], [64], [68], [71] is designed for Ethernet networks. The second category [29], [43], [47], [63] aims at CXL-memory. These works focus on alleviating the performance degradation of using disaggregated memory since it offers higher latency than local DRAM. In contrast, DmRPC is orthogonal to these works. We focus on exploring the benefit of introducing disaggregated memory to datacenter RPC. And DmRPC can be applied to both network-based disaggregated memory and CXL-based disaggregated memory. [46] reduces memory occupation by detecting pages that have the same content. In contrast, DmRPC aims at reducing unnecessary data movements.

Datacenter RPC. The optimization of small RPC wire protocol originates with Birrell and Nelson [15], who introduce implicit-ACK. Sprite RPC [70] and eRPC [37] directly use raw datagrams and performance re-transmissions only at clients. The Direct Access File System [20], [74] uses RDMA in RPCs. In contrast, DmRPC focuses on optimizing RPC for DM-enable datacenter.

Global shared address space. Distributed shared memory (DSM) [12], [38], [44], [54], [65] provides a logical global shared address space across physically distributed processes. FaRM [22] and Anna [71] impose a consistency model to manage shared memory. Adopting these approaches to datacenter RPC significantly complicates the user logic, coupling the caller and the callee. In DmRPC, the microservices don't need to handle data consistency, each microservice that shares the same memory region is decoupled with each other. Ray [50], [66] leverages an in-memory data store service to share an immutable copy, avoiding complicating user logic at the cost of performance loss. In contrast, DmRPC is specific to the DM-enabled datacenter, it leverages `copy-on-write` to avoid complicating microservices logic while keeping performance.

Remote fork with `copy-on-write`. The remote fork mechanism has been widely used in virtual machine fork and migrations to reduce unnecessary copy [7], [30], [40], [69]. For example, MITOSIS [69] maps a child container's virtual memory to its parent container's physical memory without checkpointing the memory. The `copy-on-write` semantics in the remote fork avoid copying the entire memory when forking new containers, instead, the copy only occurs when a write happens. In contrast, DmRPC uses fork semantics to avoid unnecessary data movement when invoking RPCs.

VIII. Conclusion

Modern datacenter services are decomposed into deep hierarchies of microservices, which introduces much redundant data movement along the network. Motivated by the trend that the modern datacenter is actively embracing disaggregated memory, we propose DmRPC. First, it introduces a DM-based global shared space, thus providing $pass\ by\ reference$ semantics to microservices to minimize redundant data movement. Second, it adopts a `copy-on-write` mechanism, providing programming simplicity while keeping high performance. The experimental results show that in DeathStarBench, DmRPC-net can achieve 3.1× higher throughput than eRPC while achieving 2.5× lower average latency.

Acknowledgement. We would like to thank Yizhou Shan and Jialin Li for their valuable feedback on our work. The work is supported by the following grants: the National Key R&D Program of China (NO.2022ZD0119301), the Program of Zhejiang Province Science and Technology (2022C01044), Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010). Zeke Wang is the corresponding author.

References

[1] grpc. https://grpc.io/, 2022.

[2] Paging permission flags. https://wiki.osdev.org/Paging, 2022.

[3] CXL Memory Disaggregation and Tiering. In *SDC*, 2023.

[4] Spark BlockTransferService. https://books.japila.pl/apache-spark-internals/storage/BlockTransferService/, 2023.

[5] Intel® xeon® processor scalable family technical overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html/, 2024.

[6] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *ATC*, 2018.

[7] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. Vmflock: Virtual machine co-migration for the cloud. In *HPDC*, 2011.

[8] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *ATC*, 2020.

[9] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *SOCA*, 2016.

[10] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[11] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *HotCloud*, 2020.

[12] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *PPoPP*, 1990.

[13] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. Reconsidering os memory optimizations in the presence of disaggregated memory. In *ISMM*, 2022.

[14] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *TOCS*, 2(1):39–59, 1984.

[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. *VLDB*, 2018.

[17] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, 2021.

[18] Compute Express Link. CXL™ 3.0 Specification. https://www.computeexpresslink.org/download-the-specification, 2022.

[19] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *ASPLOS*, 2019.

[20] Matt DeBergalis, Peter F Corbett, Steven Kleiman, Arthur Lent, Dave Noveck, Thomas Talpey, and Mark Wittle. The direct access file system. In *FAST*, 2003.

[21] DPDK. Data Plane Development Kit. https://www.dpdk.org, 2024.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI*, 2014.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, 2015.

[24] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS*, 2020.

[25] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *ASPLOS*, 2021.

[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[27] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS*, 2019.

[28] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets rdma. In *NSDI*, 2021.

[29] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with directcxl. In *ATC*, 2022.

[30] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Secure live migration of sgx enclaves on untrusted cloud. In *DSN*. IEEE, 2017.

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, 2017.

[32] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS*, 2022.

[33] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. Shuhai: A tool for benchmarking high bandwidth memory on fpgas. *TC*, 2022.

[34] Intel. Intel® performance counter monitor. https://github.com/intel/pcm, 2022.

[35] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: a distributed platform for building microservices in the cloud. In *EuroSys*, 2018.

[36] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[37] Anuj Kalia, Michael Kaminsky, and David G Andersen. Datacenter rpcs can be general and fast. *NSDI*, 2019.

[38] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. *Distributed Shared Memory: Concepts and Systems*, 1994.

[39] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *ATC*, 2019.

[40] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[41] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *ASPLOS*, 2021.

[42] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *SOSP*, 2021.

[43] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, 2023.

[44] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP*, 1988.

[45] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[46] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International*

Symposium on High-Performance Comp Architecture, pages 1–12. IEEE, 2012.

[47] 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. In *ASPLOS*, 2023.

[48] Mellanox. ConnectX®-5 En Card Product Brief. https://www.mellanox.com/sites/default/files/relateddocs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf, 2017.

[49] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *SIGCOMM*, 2022.

[50] 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*, 2018.

[51] Paul Movall, Ward Nelson, and Shaun Wetzstein. Linux physical memory analysis. In *USENIX Annual Technical Conference, FREENIX Track*, pages 23–32, 2005.

[52] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[53] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *ATC*, 2015.

[54] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[55] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. Thymesisflow: a software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *MICRO*, 2020.

[56] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The benefits of general-purpose on-nic memory. In *ASPLOS*, 2022.

[57] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*, 2015.

[58] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. Aifm:high-performance, application-integrated far memory. In *OSDI*, 2020.

[59] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *SoCC*, 2014.

[60] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. In *MICRO*, 2023.

[61] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *ATC*, 2020.

[62] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *SOSP*, 2017.

[63] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. Evaluating emerging cxl-enabled memory pooling for hpc systems. *arXiv preprint arXiv:2211.02682*, 2022.

[64] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *OSDI*, 2020.

[65] Stephanie Wang, Benjamin Hindman, and Ion Stoica. In reference to rpc: it's time to add distributed memory. In *HotOS*, 2021.

[66] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In *NSDI*, 2021.

[67] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Shuhai: Benchmarking high bandwidth memory on fpgas. In *FCCM*, 2020.

[68] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. Fpganic: An fpga-based versatile 100gb smartnic for gpus. In *ATC*, 2022.

[69] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast rdma-codesigned remote fork for serverless computing. 2023.

[70] Brent B Welch. The sprite remote procedure call system. Technical report, 1986.

[71] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. Anna: A kvs for any scale. *TKDE*, 33(2):344–358, 2019.

[72] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan RK Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. Orca: A network and architecture co-design for offloading us-scale datacenter applications. *arXiv preprint arXiv:2203.08906*, 2022.

[73] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[74] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, and Zeke Wang. Smartds: Middle-tier-centric smartnic enabling application-aware message split for disaggregated block storage. In *ISCA*, 2023.

[75] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *ASPLOS*, 2021.