Logical Memory Pools: Flexible and Local Disaggregated Memory

Emmanuel Amaro
VMware Research

Stephanie Wang
UC Berkeley

Aurojit Panda
NYU

Marcos K. Aguilera
VMware Research

Abstract

We propose *logical memory pools*, a memory disaggregation architecture for the emerging Compute Express Link (CXL) technology in datacenters. The key idea is to create a memory pool by carving out parts of the local memory in each server, rather than using a physical memory pool that is separate from servers. Logical pools provide significant benefits over physical pools, namely, lower cost, support for near-memory computing without extra hardware, and flexibility on designating whether memory is part of the memory pool or not. We demonstrate that logical pools can execute workloads that are unfeasible in physical pools, and that its faster access leads to better performance. Realizing logical memory pools poses five major challenges, which we believe can be overcome. Given the benefits of logical pools, we believe the CXL community should refocus efforts on logical, rather than physical memory pools.

CCS Concepts

• **Networks** → **Network architectures**; **Data center networks**; • **Computer systems organization** → *Distributed architectures*;

ACM Reference Format:
Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K. Aguilera. 2023. Logical Memory Pools: Flexible and Local Disaggregated Memory. In *The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23), November 28–29, 2023, Cambridge, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3626111.3628201

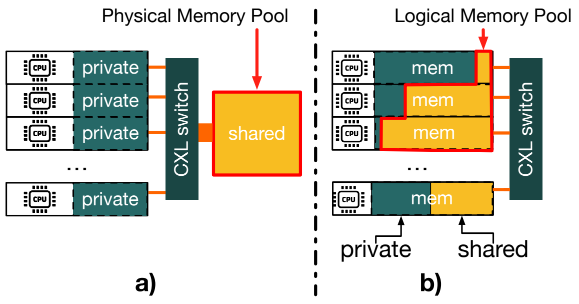

Figure 1: a) A Physical Memory Pool deployment. b) A Logical Memory Pool deployment.

1 Introduction

Memory disaggregation, which moves memory to pools accessible from multiple servers, improves memory utilization and reduces the cost of ownership of memory in data centers [2, 18, 38, 39, 43, 45]. Most prior work in this space focuses on *software* memory disaggregation, where software explicitly issues IOs to transfer data to and from the memory pool (*e.g.*, using RDMA). Recently, a faster approach—*hardware* memory disaggregation—is gaining traction due to the Compute Express Link (CXL) standard. CXL allows hosts to connect memory through the PCIe bus—including fabric-attached memory [30]—and processors can directly access such memory. Hardware memory disaggregation is faster than software because processors access memory using loads and stores, rather than IO requests. Load and stores are lighter weight, have lower latency, and can leverage processor mechanisms to hide memory latency, such as pipelining, out-of-order and speculative execution, and prefetching [17, 27]. CXL is also more promising than prior attempts at hardware memory disaggregation [1, 7, 16, 19, 25] because it has gained wide support from industry, from memory makers (Samsung, Micron, SK Hynix), to OEMs (*e.g.,* HPE, Dell), CPU manufacturers (Intel, AMD), and startups.

The current proposals for CXL memory disaggregation [8, 27, 29] are based on **physical memory pools** (Figure 1a), where the memory pool is physically separate from each server. Unfortunately, physical memory pools have four main drawbacks. First, they incur additional monetary cost due to the need for the memory pool hardware (power supply, motherboard and CPU, or custom ASIC or FPGA), physical space in the rack, and fabric ports on the switch to connect the pool. Second, even with CXL, physical pools incur a performance penalty, as memory accesses to the pool are slower than local memory by 3-10× [27, 30, 44]. Third, physical pools do not have CPUs, GPUs, or accelerators for near-memory computing; while one could imagine adding these to the pool, it would exacerbate cost. Fourth, physical pools impose a fixed ratio of local to pooled memory: once the system is deployed, this ratio is hard to adjust because it requires physically moving memory between servers and the pool. Meanwhile, recent work [27, 44] has shown that even with CXL, workloads can be highly impacted when their working set does not fit in local memory. Thus, to support a variety of workloads on the same deployment, we may be forced to overprovision local memory to provide good performance for all workloads, but this approach negates the main benefits of disaggregation.

To address these drawbacks, in this paper we advocate for a new memory disaggregation architecture based on **logical memory pools** (Figure 1b). The key idea is to create a memory pool by carving out parts of the local memory from each server, rather than using a physically separate box. In this manner, we logically partition each server's memory into private and shared regions, where the union of all shared regions constitute the disaggregated memory. Similarly to physical pools, logical pools improve resource utilization. Unlike their physical counterparts, logical pools have lower cost, provide disaggregated memory with near-memory computing, and are more flexible. In more detail, logical pools are less costly because they avoid the extra hardware of the memory pool, its rack space, and its use of ports on the fabric switch (only the servers are connected to the switch). Logical pools support near-memory computations on disaggregated memory through three mechanisms: data placement, data migration (as in NUMA migration), and compute shipping. Near-memory computations do not require any extra hardware because servers already have powerful processors connected to the memory—not only CPUs, but possibly GPUs and other accelerators. With near-memory computation, both bandwidth and latency are significantly better than with CXL physical pools (e.g., see Table 1). Lastly, logical pools are flexible because the division of private and shared regions on each server can vary over time and per server, to dynamically address the needs of different workloads.

We present an early assessment of the benefits of logical pools over physical pools. Our findings suggest that logical pools provide higher disaggregated memory bandwidth and they accommodate a broader spectrum of workloads for a fixed total memory budget. As CXL fabrics for disaggregated memory are not yet available, we parameterize our experiments based on a slowdown of the disaggregated memory relative to local memory. Our evaluation indicates that logical pools can provide up to 4.7× higher bandwidth than physical pools. Furthermore, we highlight scenarios where the flexible ratio of logical pools enables workloads that are infeasible using physical pools when the total deployment memory is held constant.

	Latency (ns)	Bandwidth (GB/s)
Local memory	82	97
CXL remote memory	280 or 303	31 or 20

Table 1: Latency and bandwidth for different memory types. Local numbers are from our evaluation (§4). CXL numbers are from Pond [27] or an FPGA [44], respectively. In Pond, latency is estimated using a switch, and bandwidth is the maximum bandwidth of PCIe5 with 8 lanes. The FPGA has DDR4 memory connected locally to the host via PCIe5 with 16 lanes.

Given the many benefits of logical pools shown in this paper, we believe the CXL community should refocus its efforts from physical to logical memory pools.

Realizing logical memory pools creates five major challenges. First, we must design policies to choose the ratio of private to shared memory in each server, in addition to mechanisms to inform such policies by monitoring use. Second, to leverage near-memory computing, we need policies and mechanisms that migrate memory buffers to the servers that will most benefit from local access, or ship computations to the server with local access. Third, we need an efficient and flexible addressing mechanism for buffers in the logical pool that allows migration: as buffers can be shared, different servers may have pointers to the buffer being migrated and migration should not corrupt these pointers. Fourth, similarly to physical memory pools, we must balance scalability with cache coherence. Finally, we need mechanisms to handle server crashes, which take down part of the memory pool. These challenges must be overcome in a distributed, high-performance context.

The rest of this paper is organized as follows: §2 covers background and motivation. We describe our proposal in more detail in §3, and §4 focuses on the benefits of LMPs. We then discuss challenges associated with LMPs in §5, and §6 covers related work. We conclude in §7.

2 Background and Motivation

2.1 Memory Disaggregation

Many prior research efforts show the potential of memory disaggregation with high-performance network fabrics. For example, CFM leverages OS-level mechanisms to improve job throughput using far memory [2]; MIND employs programmable switches for rack-scale disaggregated memory management [26]; and AIFM provides libraries for applications to use far memory with high performance [39]. These efforts rely on software memory disaggregation: software inititates requests to access disaggregated memory and acknowledges completions. For instance, using RDMA, application libraries or the OS must post memory access requests to network queues; the NIC then adds completions to completion queues, which software drains. This process is slow and poorly aligned with CPU architectural features. Further, this problem is common to most network transports, including TCP, where CPU overheads are even larger, exacerbating the performance impact of software memory disaggregation.

An emerging technology, Compute Express Link (CXL), enables hardware memory disaggregation. While memory disaggregation is not novel [1, 25], CXL has garnered robust industry support, which can lead to broader adoption and impact. For example, recent Intel and AMD CPUs support early CXL versions. By integrating CXL ports directly into CPUs, processors can directly address disaggregated memory, enabling CPU architectural features like caching, prefetching, pipelining, and speculative execution. As a result, hardware memory disaggregation reduces CPU overheads, lowers latency, and increases throughput compared to previous software approaches. Early performance evaluations on CXL prototype exist [27, 44]; Table 1 summarizes them and compares them to local memory. Bandwidth is expected to be 4-10× lower, and latency 3-5× higher, than local memory. Thus, while hardware disaggregated memory performs significantly better than software disaggregated memory, it is still expected to be slower than local memory.

2.2 CXL-Based Memory Pools

We now discuss additional CXL details relevant to memory pools. CXL is a family of low latency, high throughput I/O bus architectures designed to interconnect hosts, accelerators, and memory devices. It uses the ubiquitous PCIe electrical interface and defines three protocols: CXL.io, CXL.cache, and CXL.mem. CXL.io provides I/O semantics and is implemented by all CXL devices because it is used by the control path. CXL.cache implements a cache synchronization protocol for device and CPU caches, and CXL.mem provides a transactional interface between CPUs and CXL memory devices.

CXL defines three device-types: Type-1 and Type-2 are accelerators, and Type-3 are memory devices. Our focus is on Fabric-Attached Memory (FAM) Type-3 devices that use CXL.mem, and can be accessed by multiple hosts. FAMs can expose one or more disjoint memory regions. Two enhanced versions of FAMs are defined: Global FAMs that use Port Based Routing (PBR) allowing them to scale to a rack; and Shared FAMs that allow a single memory region to be concurrently accessed from multiple hosts.

In this paper, we use the term *memory pool* to refer to Global Shared Fabric-Attached Memory. We assume CXL (or similar technology) will support Shared FAMs and the ability for one server to access physical memory provided by DIMMs on the motherboard of another server. CXL allows coherency for Shared FAMs to be implemented in hardware with multi-host support (*i.e.,* via an Inclusive Snoop Filter and a Back-Invalidation protocol), or through software-managed coherency.

3 Logical Memory Pool

This section first positions Logical Memory Pools (LMPs) in the context of memory disaggregation by delineating shared and unique capabilities compared to physical memory pools. An architecture overview of LMPs follows.

3.1 Key Capabilities

LMPs shares three capabilities with CXL physical pools. First, servers in both types of pools use the standard CPU load-store memory interface to a global fabric address space. Second, servers can access more memory than locally present. Third, the memory pool serves as shared memory for servers.

LMPs provide two unique capabilities not offered by CXL physical pools. First, all memory in an LMP can be accessed at local memory speeds by the server hosting it, delivering the key performance advantage of LMPs. Second, servers can dynamically adjust their ratio of private to shared memory, providing flexibility to adapt to the working set sizes of active applications. Recent work has shown the importance of fitting the working set into memory with local access speeds, even in CXL environments [27, 44]. However, an advantage of physical memory pools over LMPs is that the former permit a more independent scaling of memory and compute.

3.2 Architecture

In LMPs, a server's private memory is exclusively accessed by its processors and maintains local system state, such as the OS and process control blocks, as well as per-process state like stack, heap, and text sections. In contrast, shared regions are globally accessible by any server, and constitute the disaggregated memory, creating a NUMA-like system with different access latencies (local and remote). We envision LMPs providing 10–100 TB of shared memory while enabling its servers to achieve high memory utilization, thus reducing total cost of ownership.

Cache coherence across multiple servers can present a scalability challenge, as has been previously observed in distributed shared memory [1, 5, 25]. Therefore, LMPs do not assume cache coherence for all shared memory. Instead, it provides a small amount (a few GBs) of coherent memory that can be used for coordination and synchronization. Limiting the amount of coherent memory lessens the likelihood of filling

Remote link	Uncore freq.	Min lat.	Max lat.	Bandwidth
Link0	2.2Ghz	163ns	418ns	34.5GB/s
Link1	0.7Ghz	261ns	527ns	21.0GB/s

Table 2: Minimum and maximum latency under load, and bandwidth, achieved for two emulated CXL links. Link0 is the default UPI configuration. Link1 is a slowed down UPI link for which we decrease the remote CPU's uncore frequency to 0.7Ghz.

CXL's Inclusive Snoop Filter and allows tracking coherence at a granularity finer than a cache line to avoid false sharing.

As LMPs present a load-store interface on a global address space, it is important to have an efficient addressing scheme. Local references, or references whose global address resolve to the same server, avoid CXL traffic. On the other hand, accesses to remote references, or references that resolve to another server, induce fabric traffic. Accessing remote references will have higher latency and lower throughput compared to local references, as discussed in §2. Thus, in order to enhance data locality, the addressing scheme should permit data migration across servers. That is, migrating a buffer should not invalidate its address.

Implementing LMPs requires a per-server runtime and an application library for allocating, controlling, and setting up disaggregated memory access—for example, by mapping a range of virtual addresses to memory in the pool. Furthermore, the runtime must execute at least two background tasks: one for adjusting the size of shared regions to minimize remote accesses, and another to find opportunities for buffer migration.

We discuss the challenges associated with shared cache coherent regions, memory addressing schemes, and policies for adjusting size of shared regions and migration in §5.

4 Benefits

We preliminarily evaluate the benefits of LMPs over physical pools, including their lower entry barrier, and superior flexibility and performance as demonstrated by microbenchmarks.

4.1 Evaluation setup

Testbed. We use a two-socket server with Intel Xeon Gold 5120 CPUs, each with 14 cores at a fixed frequency of 2.2Ghz. The system has 192GB DRAM, with 96GB per NUMA node. Nodes are connected with two bidirectional UPI links [20].

Remote links. While a CXL 3 fabric is required for both logical and physical memory pools (see §2.2), this technology is not yet available. Therefore, we emulate the fabric using UPI links. Given that CXL fabrics are expected to underperform UPI links due to longer wires, and the necessity for re-timers [27], we artificially decrease the remote socket's uncore frequency to slow down memory accesses over UPI. We label the slowed down link as Link1, and use Link0 as the baseline UPI link. A characterization of both links' read latency under load is provided in Table 2. We posit that Link0 represents an upper bound for future CXL remote performance while Link1 is a closer approximation.

Memory pool configurations. In our microbenchmarks, we consider one logical memory pool and two physical memory pool setups, each with 4 servers and a total memory budget of 96GB. In the physical pool setups, the memory pool has 64GB, and each server keeps 8GB of local memory. The first physical pool setup (Physical cache) uses local memory as cache for the pooled memory, while the second one (Physical no-cache) does not use its local memory as cache. Caching incurs an upfront memcpy() overhead but provides faster subsequent reads. In contrast, the LMP setup (Logical) uniformly distributes the 96GB, assigning 24GB to each server. Although the specific memory capacities we employ are small compared to real deployments, we are more interested in the ratios of pooled to local memory.

Microbenchmark. We measure the bandwidth used by a multi-core server as it performs an aggregation on a large vector in disaggregated memory. More precisely, one server computes the sum of a vector using 14 cores, where each core sums part of the vector. We repeat this process 10 times and report the average bandwidth. We run the experiment using the two UPI link configurations (Link0 and Link1) and we consider four vector sizes: 8GB, 24GB, 64GB, 96GB.

4.2 Benefit 1: Lower Entry Barrier

We first examine the cost implications of LMPs compared to physical pools by comparing the total required resources for each deployment. While both require a fabric switch and a fabric adapter per server, physical pools demand additional components such as a power supply, motherboard, and CPUs or custom ASICs/FPGAs to function as the memory pool. Also, physical pools require extra rack space and additional switch ports. Further, provisioning the switch↔pool link with the same capacity a server↔switch link can create incast problems at the physical pool, demanding either a higher-capacity link or multiple links (thick orange line in Figure 1(a)). In contrast, LMPs use existing rack infrastructure and do not require extra rack space or fabric ports. Although incast problems are possible with LMPs, they have three ways to prevent it: data placement, data migration, and compute shipping.

Focusing on memory, we contrast LMPs with physical pools in two scenarios. The first scenario provides an equal amount of *disaggregated* memory to both deployments. Here, the physical deployment is more costly due to its need for supplementary memory to function as local memory for each server. In the second scenario, both deployments have equal *total* memory, and so the physical deployment needs to delegate

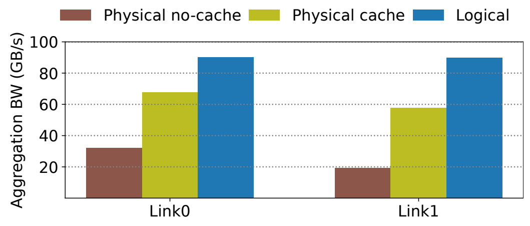

Figure 2: 8GB vector sum.

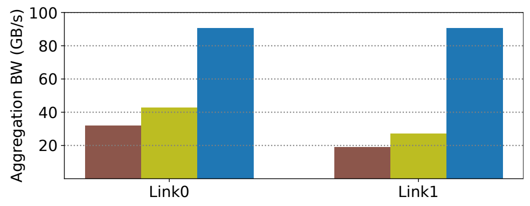

Figure 3: 24GB vector sum.

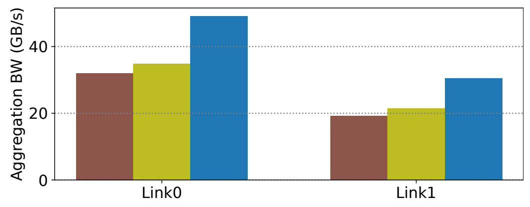

Figure 4: 64GB vector sum.

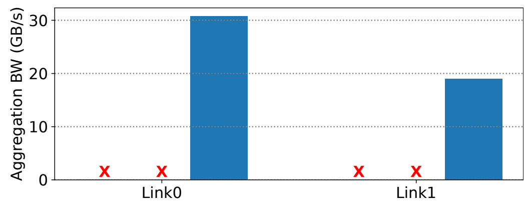

Figure 5: 96GB vector sum.

some memory to the memory pool. This causes servers to end up with less local memory than LMP servers. From these two scenarios, we find that LMPs are better than physical pools either economically (first scenario) or operationally (second scenario).

4.3 Benefit 2: Enhanced Memory Performance

Accessing disaggregated memory in LMPs is at least as fast as accessing a physical pool in all cases. Moreover, when an LMP server accesses memory in the pool that resolves locally, LMPs provide faster access. This advantage is illustrated in Figure 2 and Figure 3 for the vectors of size 8GB and 24GB, which fit entirely in the local memory of one LMP server. We see that LMPs deliver up to 4.7× improved bandwidth compared to Physical no-cache for both 8GB and 24GB vectors, and up to 3.4× compared to Physical cache for the 24GB vector.

LMPs maintain a performance advantage even when data size exceeds a server's local memory capacity. For instance, the 64GB vector does not fully fit in the cache of Physical cache, or in the local memory of an LMP server. However, an LMP server can access 3/8 of the vector locally. As Figure 4 shows, this leads to Logical providing 42% higher bandwidth than Physical cache on Link1. Moreover, the slower the remote link, the better the performance of LMPs relative to physical pools. This is because as remote memory gets slower, the unaffected performance of local memory for Logical delivers a higher advantage.

While our microbenchmarks focused on bandwidth, a similar analysis applies for latency, where LMPs would outperform the physical pool. This is because the maximum remote loaded latency is 2.8× and 3.6× higher than maximum loaded local latency, when using Link0 and Link1 links, respectively.

4.4 Benefit 3: Near-memory Computing

If we distribute the sum across LMP servers, then each server could access different parts of the vector locally. Thus, LMPs can use computation shipping to further enhance performance through near-memory computing so that all memory accesses are local. The end result is an even larger performance improvement than reported above (not shown). Computation shipping requires a more sophisticated runtime and possibly application changes, but it can be done in LMPs using server CPUs. In contrast, with physical pools, computation shipping either is infeasible or requires additional processing hardware, exacerbating its cost.

4.5 Benefit 4: Memory Flexibility

As we mentioned before, LMPs are ratio-flexible as servers can adjust the size of their private and shared regions to adapt to workload requirements. This is illustrated with the experiment using the 96 GB vector in Figure 5. With LMPs, each server can contribute all of its memory to the logical pool, allowing the deployment to fit the vector. The physical pool, however, has too little memory in the pool (64GB) to fit the vector and cannot run the workload. It is impossible to reconfigure it short of physically moving memory DIMMs from servers to the memory pool.

5 Challenges

There are five main challenges in realizing LMPs.

Cache coherence. For scalability reasons, we do not expect LMPs to provide cache coherence for most of the shared memory, but they should provide a few GBs of cache coherent shared memory (coherent memory) for synchronization purposes (§3.2). However, achieving high-performance even for a small amount of coherent memory is non-trivial. A coherence engine must interpose on all accesses to coherent memory, a requirement that can potentially slow down local accesses. We can address this by leveraging approaches that place coherence engines in fabric switches to minimize the cost of interposition. In addition, applications can leverage prior work on scalable coordination mechanisms to reduce coherence traffic on coherent memory, such as NUMA-aware coordination [6, 9–11, 24].

Sizing the shared regions. Striking a balance in the size of shared memory regions is critical, as demonstrated by the findings in §4.5. Oversizing the shared regions can negatively affect performance of local workloads if the local memory is monopolized by remote servers. On the other hand, undersizing the shared region can render the LMP insufficient for the application needs. Thus, the challenge is to make disaggregated data to tightly fit in the LMP's shared regions. Finding this balance can be formulated as a global optimization problem that is solved periodically. The objective is to maximize the number of local accesses while prioritizing high-value applications.

Locality balancing. Similar to NUMA balancing in multi-socket machines, LMPs need to periodically migrate data between servers to maximize the number of local accesses. Balancing is even more important for LMPs than multi-socket systems, because the difference between fast and slow accesses is greater in LMPs. Yet balancing between servers is harder since it is a distributed systems problem: we need new mechanisms to identify slow accesses (NUMA systems unmap memory to cause page faults, but this is too slow for LMPs), and new policies to decide what data to migrate (NUMA systems can rely on the kernel to drive policy). For the former, a simple solution is to use performance counters to profile accesses. For the latter, one could use access bits to identify hot remote data for each server.

Address translation. To provide locality balancing, LMPs must support logical addresses and an efficient scheme to translate logical addresses to their physical address (a server and a physical address within the server). A traditional approach is to use a directory that maps logical addresses to physical locations (*e.g.,* a multi-level page table with TLBs) but this approach is too inefficient for our use, because all servers need access to the directory when translating addresses, and this would incur slow remote accesses. A better solution is to translate in two steps: first, map a logical address to a server, then map the address within the server. The first step uses coarse-grained maps, which can be globally accessible, while the second step is more fine grained and can be resolved locally within the target server.

Failure domains. Failures of memory may occur in both LMPs and physical pools, albeit in slightly different ways. With LMPs, memory failures come from host crashes while with physical pools it comes from memory pool crashes. To handle failures, LMPs can take advantage of similar solutions proposed for physical pools, such as failure masking through replication or erasure coding [49], or failure reporting to application through exceptions.

6 Related Work

We propose a new CXL-based disaggregated memory architecture, but our work is related to work in several other areas.

NUMA. NUMA (Non-Uniform Memory Access) is a type of computer architecture for systems with many CPU sockets, where each socket has its own local memory, and sockets can access another socket's memory, albeit with higher latency. While NUMA architectures are typically used within a server, LMP's non-uniform memory access characteristics are similar to those in the NUMA architecture. Therefore, we believe work done for NUMA (*e.g.,* NUMA migration, NUMA-aware data structures, NUMA locks, etc.) will be useful for LMPs.

Remote memory systems using RDMA. RDMA (Remote Direct Memory Access) allows a host to access memory on another host without involving the remote processor. Unlike CXL (and LMP), RDMA requires the host to issue NIC IOs to read and write remote memory instead of loads and stores (as in LMPs). Remote memory systems encapsulate these IOs in user libraries for accessing the remote memory (*e.g.,* FaRM [12, 13], scale-out NUMA [35], scale-out cc-NUMA [15], RackOut [36], FaSST [23], Storm [37]). RDMA based systems resemble LMPs in its memory flexibility, and we believe some RDMA-based techniques can be carried over to LMPs to benefit key-value stores (*e.g.,* [22, 33]), databases (*e.g.,* [40, 48]), HPC (*e.g.,* [14, 21, 46]), distributed file systems (*e.g.,* [31, 32, 47]), and many other systems.

Software memory disaggregation. The recent proliferation of fast local area networks has led to many systems that use the memory of a remote host ("far memory") to extend the local memory capacity through a *software* solution. This idea can be implemented using transparent paging (CFM [2], Infiniswap [18]) or runtime libraries (AIFM [39], Carbink [49]).

DSM. There is much work on DSM (Distributed Shared Memory) systems (*e.g.,* [1, 3, 4, 25, 28, 34, 41, 42]), whose goal is to emulate a shared memory system using a distributed system. This goal turned out to be elusive due to the difficulty of getting reasonable performance, especially due to the cache-coherence requirement of such systems. DSM systems have had both software and hardware implementations, and disaggregated memory systems are most similar to hardware DSMs. From the work on hardware DSMs, we learn that cache coherence is hard to scale well.

7 Conclusion

Hardware memory disaggregation is finally becoming a reality with CXL. The CXL community is currently targeting physical memory pools to achieve memory disaggregation, but we believe they should refocus to provide instead logical memory pools due to its many advantages, as shown in this paper.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. 1995. The MIT Alewife machine: Architecture and performance. *ACM SIGARCH Computer Architecture News* 23, 2 (1995), 2–13.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *European Conference on Computer Systems*. 1–16.

[3] Cristiana Amza, Alan L. Cox, Shandya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer* 29, 2 (Feb. 1996), 18–28.

[4] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. 1990. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In *ACM Symposium on Principles and Practice of Parallel Programming*. 168–176.

[5] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient distributed memory management with RDMA and caching. 11, 11 (2018), 1604–1617.

[6] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *ACM Symposium on Principles and Practice of Parallel Programming*. 157–166.

[7] CCIX Consortium. Accessed 2023/01/26. CCIX. (Accessed 2023/01/26). https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf.

[8] cxl [n. d.]. Compute Express Link (CXL). ([n. d.]). https://www.computeexpresslink.org.

[9] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. 2021. Clof: A compositional lock framework for multi-level NUMA systems. In *ACM Symposium on Operating Systems Principles*. 851–865.

[10] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[11] David Dice, Virendra J Marathe, and Nir Shavit. 2012. Lock cohorting: a general technique for designing NUMA locks. *ACM SIGPLAN Notices* 47, 8 (2012), 247–256.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Symposium on Networked Systems Design and Implementation*. 401–414.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Ed Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles*. 54–70.

[14] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 553–560.

[15] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *European Conference on Computer Systems*. 1–15.

[16] genz [n. d.]. Gen-Z consortium. ([n. d.]). https://genzconsortium.org.

[17] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *USENIX Annual Technical Conference*.

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *Symposium on Networked Systems Design and Implementation*. 649–667.

[19] Intel. Accessed 2023/01/26. Intel Rack Scale Architecture. (Accessed 2023/01/26). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[20] Intel. Accessed 2023/06/29. Intel Xeon Processor Scalable Family Technical Overview. (Accessed 2023/06/29). https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[21] Nusrat Sharmin Islam, Dipti Shankar, Xiaoyi Lu, Md Wasi-Ur-Rahman, and Dhabaleswar K Panda. 2015. Accelerating I/O performance of big data analytics on HPC clusters through RDMA-based key-value store. In *International Conference on Parallel Processing*. 280–289.

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *ACM Special Interest Group on Data Communications*.

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Symposium on Operating Systems Design and Implementation*. 185–201.

[24] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and practical locking with shuffling. In *ACM Symposium on Operating Systems Principles*. 586–599.

[25] James Laudon and Daniel Lenoski. 1997. The SGI Origin: a ccNUMA highly scalable server. *ACM SIGARCH Computer Architecture News* 25, 2 (1997), 241–251.

[26] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. Mind: In-network memory management for disaggregated data centers. In *ACM Symposium on Operating Systems Principles*. 488–504.

[27] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[28] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* 7, 4 (Nov. 1989), 321–359.

[29] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture*.

[30] Ming Liu. 2023. Fabric-Centric Computing. In *Workshop on Hot Topics in Operating Systems*. 118–126.

[31] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *USENIX Annual Technical Conference*.

[32] Shaonan Ma, Teng Ma, Kang Chen, and Yongwei Wu. 2022. A Survey of Storage Systems in the RDMA Era. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[33] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference*.

[34] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference*. 291–305.

[35] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[36] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The case for RackOut: Scalable data serving using rack-scale systems. In *ACM Symposium on Cloud Computing*. 182–195.

[37] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. 2019. Storm: a fast transactional dataplane for remote data structures. In *ACM International Conference on Systems and Storage*. 97–108.

[38] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *Symposium on Networked Systems Design and Implementation*. 181–198.

[39] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *Symposium on Operating Systems Design and Implementation*. 315–332.

[40] André Ryser, Alberto Lerner, Alex Forencich, and Philippe Cudré-Mauroux. 2022. D-RDMA: Bringing Zero-Copy RDMA to Database Systems. In *Conference on Innovative Data Systems Research*.

[41] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. 1996. Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 174–185.

[42] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. 1994. Fine-grain Access Control for Distributed Shared Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–306.

[43] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Symposium on Operating Systems Design and Implementation*.

[44] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. *arXiv preprint arXiv:2303.15375* (2023).

[45] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *Symposium on Networked Systems Design and Implementation*. 161–179.

[46] Hao Wang, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K Panda. 2013. GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation. *IEEE Transactions on Parallel and Distributed Systems* 25, 10 (2013), 2595–2605.

[47] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *USENIX Conference on File and Storage Technologies*, Vol. 19. 221–234.

[48] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1637–1650.

[49] Yang Zhou, Hassan Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Jack Turner, David E Culler, Hank Levy, and Amin Vahdat. 2022. Carbink: Fault-tolerant Far Memory. In *Symposium on Operating Systems Design and Implementation*.