Towards Rack-as-a-Computer in Memory Interconnect Era with Coordinated Operating System Sharing

Yuxin Ren, Mingrui Liu, Hongbo Li, Chang Liao, Xiaojia Huang, Jianhua Zhang, Hanjun Guo, Yubo Liu, and Ning Jia
Huawei Technologies
China

ABSTRACT

Emerging memory interconnect (such as CXL and HCCS) promises rack-scale machine to become a reality, as the interconnect enables `load/store` accessible memory shared across the entire rack. However, the rack-scale shared memory poses two unique challenges on the operating system, primarily because of synchronization bottleneck and reliability issue. First, hardware cache coherence is not guaranteed, thus existing lock-based approach is ineffective to synchronize cross-node memory access. Second, memory faults significantly increase, and additional interconnect hops and switches expand fault surface and radius. As a result, current systems cannot efficiently leverage in-rack shared memory and instead manage rack resource in a disaggregated way, suffering from unnecessary networking/RDMA transmission overhead and redundant data copies.

This paper proposes FlacOS, a shared operating system for memory-interconnected rack-scale architecture. FlacOS fully exploits the scalability, elasticity, and capacity advantages of rack-scale machine through shared memory. FlacOS strategically extracts and places kernel data structures in the shared memory to achieve uniform and shared operating system functionalities within the rack. FlacOS co-designs lock-free synchronization algorithms and system-wide fault tolerance mechanism to simultaneously ensure high performance and reliability. Experiments using Redis on a physical 640-core rack machine illustrate that FlacOS achieves a latency reduction of 1.75-2.4 times compared to network-based solutions.

HotStorage '25, July 10–11, 2025, Boston, MA, USA

ACM ISBN 979-8-4007-1947-9/25/07.
https://doi.org/10.1145/3736548.3737832

CCS CONCEPTS

• **Software and its engineering → Operating systems**.

KEYWORDS

Operating System, Rack-scale, Memory Interconnect

ACM Reference Format:
Yuxin Ren, Mingrui Liu, Hongbo Li, Chang Liao, Xiaojia Huang, Jianhua Zhang, Hanjun Guo, Yubo Liu, and Ning Jia. 2025. Towards Rack-as-a-Computer in Memory Interconnect Era with Coordinated Operating System Sharing. In *The 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25), July 10–11, 2025, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3736548.3737832

1 INTRODUCTION

Emerging memory interconnect (such as CXL [10, 11], Gen-Z [17], and HCCS [62]) enables all computing resources within a rack to access global memory through `load/store` semantics. This significantly enhances the rack-level memory capacity, computing scalability, and resource elasticity, particularly providing the sharing capability over the substantial amount of global memory across the entire rack. Through shared memory, a single operating system (OS) is possible to uniformly manage all resources within the rack, thereby eliminating the overhead many of "data centers taxes", such as serialization, memory copying, and networking overhead, while greatly reducing operational and maintenance costs.

Unfortunately, existing systems [5, 18, 28, 45, 54, 65, 69] fail to efficiently leverage the rack-scale sharing capability. They either suffers from RDMA transmission overhead or exclusively reserve disaggregated memory to one node without sharing. This is because simply placing existing OS functionalities into shared memory to achieve global management is not feasible. Sharing OS across rack faces challenges primarily in synchronization and reliability. Shared global memory has higher latency, making frequent access on critical paths impractical. Moreover, rack-scale shared memory does not guarantee hardware cache coherence support [2, 14, 50].

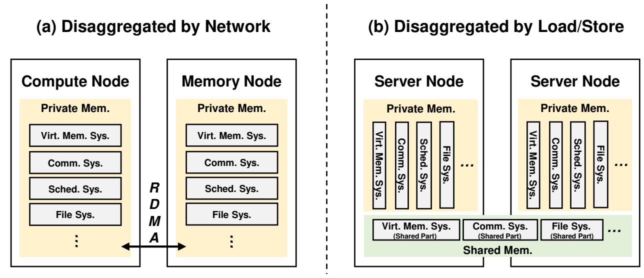

Figure 1: Comparison between existing RDMA-based disaggregated system and emerging shared rack-scale system.

Consequently, complex functionalities are difficult to directly share. The reliability of shared memory is significantly compromised [14, 34, 66], as additional interconnect hops and switches increases fault probability and surface.

We propose FlacOS, a new operating system tailored for memory-interconnected rack-scale architecture. Our key design principle is to appropriately share kernel functionalities data structures in global memory, allowing the rack to operate and execute as a single multi-core machine. To efficiently address the challenges posed by cache incoherence and frequent failure, FlacOS incorporates co-designed synchronization and fault tolerance mechanisms. FlacOS comprises three core components. The lowest layer is a development toolkit, which offers common synchronization, memory management, and reliability mechanisms. Both applications and FlacOS utilize the toolkit to build their functionalities. Second, FlacOS kernel reconstructs core system modules by extracting shared data structures from node-local private data. Finally, we propose a new system-level fault tolerance abstraction which enables vertical system-application memory and states integration to achieve efficient fault isolation and recovery. We furthermore present a vision on the future rack-level serverless computing architecture as a production use case benefiting better scalability, density, and availability from FlacOS.

We implement and test FlacOS prototype on both simulated and physical rack-scale environments. With a two-node 640-core physical rack machine which uses HCCS [62] as memory interconnect, FlacOS achieves a latency reduction of 1.75-2.4 times compared to network-based solutions in Redis. On a simulated platform using virtual machines on top of a shared persistent memory, FlacOS improves container startup latency by 3.8 times.

The contributions of this paper include:

- We propose a coordinated and partially shared operating system architecture for memory-interconnected rack-scale machines.
- We propose fault box abstraction for system-level fault isolation and co-design synchronization mechanisms to deliver high performance and reliability simultaneously.
- We present the envision of future rack-level serverless architecture that utilizes the sharing capability to achieve high elasticity, availability, and density.

2 BACKGROUND

2.1 Memory Interconnect

The emerging memory interconnect (such as CXL [10, 11], Gen-Z [17], and HCCS [62]) tightly integrate rack-wide computing infrastructure and memory resource, thereby facilitating the previously envisioned rack-scale architecture (such as FireBox [3] and THE Machine [14]) gradually become a reality. Memory interconnect permits `load/store` accessible shared memory to all the nodes in the rack. Figure 1(b) depicts an abstract representation of the rack-scale architecture. Each node possesses a local memory, and a global memory is shared across the rack. This architecture shows several notable distinctions from fabric-centric computing [34].

- The rack consists of and connects general-purpose computing resources, rather than separate devices pools with little computing capacity. Thus, each node actively executes an independent OS instance.
- Execution entities on all nodes need to interact with each other via shared memory. However, memory interconnect supports basic atomic instructions [31, 66] but is not guaranteed to provide hardware cache coherence [2, 14, 50].

2.2 Challenges in Shared Memory

As shown in Figure 1(a), existing disaggregated systems [5, 45, 54, 69] ignore the rack-scale sharing capability, and rely on RDMA to integrate different pools of resource. However, placing kernel functionalities and data structures in the global memory to achieve rack-scale shared OS encounters two design challenges: synchronization and reliability.

- Rack-scale shared memory lacks hardware support for cache coherence [2, 14, 50]. Consequently, existing synchronization methods, such as locks, are difficult to effectively employ.
- The reliability of global memory is decreased. Current memory suffers frequent failures due to smaller transistor size in fabrication and manufacturing defects [13, 39, 55, 66]. Worse still, the multi-hop and interconnect switch further expands the fault surface. Therefore, system-wide fault tolerance is critical for rack-scale OS.

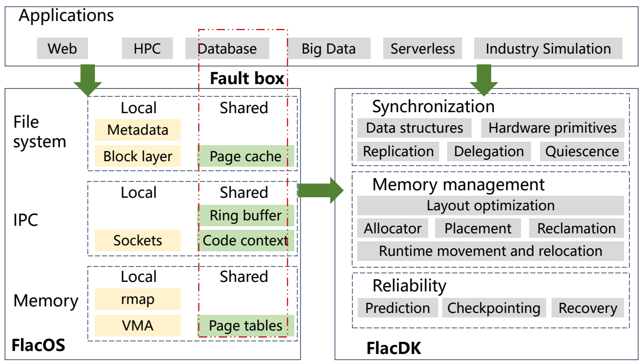

Figure 2: FlacOS architecture.

3 FLACOS DESIGN

3.1 FlacOS Architecture

FlacOS is a rack-scale OS that enables the entire rack to be operated as a single computer The primary objective of FlacOS is fully exploring the memory semantics and sharing capability of memory-interconnect rack architecture. Two principles guide FlacOS design.

- *Appropriately placing kernel functionalities and data structures in the shared global memory to eliminate network data transfer and redundant data copies within the rack.*
- *Co-design synchronization and fault tolerance mechanisms for non-cache-coherent shared data to deliver high performance and reliability simultaneously.*

Figure 2 presents the FlacOS overview, which contains three layers. FlacDK (§3.2) consists of a set of common mechanism and primitives to develop FlacOS system services and applications. FlacOS kernel implements critical functionalities in the shared global memory and coordinates with node-local OS instance. The current FlacOS prototype focuses on memory management (§3.3), file system (§3.4), and communication subsystem (§3.5), with a particular emphasis on analyzing which data structures should be allocated to global or local memory. Finally, FlacOS guarantees whole system reliability via a new system abstraction, fault box (§3.6).

3.2 FlacOS Development Kit

FlacDK particularly focuses on thee critical functionalities: synchronization, memory management, and reliability.

Synchronization. As discussed in §2.2, synchronization around rack-scale shared memory is challenging due to high memory access latency and potential the lack of hardware-guaranteed cache coherence. Therefore, the principle of FlacOS synchronization is to avoid using lock-based approach that causes heavy contention on a few of shared memory location. FlacDK provides libraries of three level synchronization primitives.

The lowest level library contains hardware specific operations that directly manipulate the global memory. These operations include atomic instructions, memory barriers, and CPU cache related instructions, such as cache flush, invalidation, and write back. The second library offers synchronization interfaces, such as locking and lock-free algorithms. The last library provides high-level concurrent data structures, such as vector, hash tables, ring buffer, and radix tree.

Especially, FlacDK leverages optimized lock-free synchronization that does not depend on hardware cache coherence.

- *Replication-based methods [4, 6, 25, 53].* This approach maintains a local replica in each node and a shared operation log to synchronize across nodes. In the common path, each node only accesses local replica to avoid contention. Modifications are logged and replayed in each node to achieve consistent and up-to-date states.
- *Delegation-based methods [15, 20, 48, 51].* This approach partitions data access between nodes, and each node exclusively manipulate a partition. When a node accesses other partitions, it sends requests to the owner node which performs the operation on behave of the requesting node.
- *Quiescence-based methods [12, 47, 60].* This approach employs read-copy-update (RCU) style synchronization to avoid in-place modification. Particularly, this method is efficient in non-cache-coherent shared memory as it converts tracking stale cache lines to parallel reference in RCU [49].

Memory management. FlacDK focuses on three aspects of memory management. 1) An object granularity allocator that needs to be incorporated into shared object synchronization and consider memory reclamation [47, 60]. 2) Optimization algorithms for object layout and allocation packing based on object hotness or liveness [26, 40]. 3) Runtime object movement and relocation mechanisms that reduce fragmentation, improve locality, and utilize memory tiering [8, 63].

Reliability. FlacDK affords common mechanisms used for system fault tolerance. These mechanisms cover the entire fault handling process, including system monitoring, failure prediction, fault detection, checkpointing, and recovery. Ensuring reliability necessitates some form of redundancy, such as data or information redundancy. We intelligently combine these redundancies with synchronization mechanisms, minimizing both synchronization overhead and redundancy cost. For instance, redundant data can reuses replicas in replication-based synchronization. Data checkpointing can be incorporated with multiple object versions in quiescence-based synchronization. This integration requires to modify memory reclamation algorithm to account for both checkpointing period and pending references in concurrent execution and stale CPU cache. Additionally, operation logs used for synchronization about object updates can be utilized to achieve state rely during fault recovery.

3.3 FlacOS Memory System

Managing physical and virtual memory is the foundation in FlacOS to leverage shared memory. It requires new design for management operations and services, including page mapping, address translation, TLB shutdown, and deduplication. Furthermore, rack-scale shared memory naturally realizes the existing memory disaggregation capability. Thus, expensive memory services, such as swapping and compression [19, 59], are no longer needed, which significantly simplifies memory system. FlacOS partitions memory management structures between shared and local memory using the following methods.

Shared heterogeneous page table. The page tables are stored in global memory, enabling the address spaces sharing and multi-threading support across the entire rack. Moreover, FlacOS page tables are capable of indexing both local and global memory and unifies them into a single level address space. However, hardware MMUs must be adapted to access global memory, and page fault handling in FlacOS must be capable of allocating and loading pages into global memory.

Local data structures. Memory management control structures, such as `rmap` and `VMA`, are preserved within local memory of each node, because these structures are not accessed frequently. Furthermore, these structures can be efficiently synchronized atop of non-cache-coherent memroy [50].

3.4 FlacOS File System

Building a file system using global memory can fully leverage the memory performance advantages. Compared to the traditional block-based file systems, the software stack of memory file systems is much lighter. Our customers have identified some scenarios that require memory file systems, such as `RootFS` for containers, temporary data storage and shuffle in big data analytics, and data sharing and collective communication in HPC applications. In FlacOS, the file system divides its core data structures between shared and local memory based on the following principles.

Shared page cache. Page cache is critical for file system performance as it bridges the performance gap between memory and storage device. However, according to the analysis of our production cluster, page cache consumes a large amount of memory space. The main reason is that they have a lot of data duplication across multiple nodes, e.g., a large number of identical container images need to be stored between nodes in a cloud service. FlacOS places page cache into the global memory which enables all nodes to share a single page cache copy. Shared page cache introduces two benefits. First, it avoids each node maintaining redundant file page copies, thus significantly reducing rack-wide memory consumption. Second, the saved memory can be used to cache more files, effectively increasing the page cache capacity and file access performance. However, sharing page cache complicates cache management, such as cache missing handling and dirty data write-back. To solve these issues, we utilize mechanisms in [37, 38] that combines asynchronous handling and multi-version updates.

Local data structures. FlacOS keeps other parts of the file system in the local. We show several typical structures and why they are not suitable for sharing. First, metadata contains a large number of complex data structures (e.g., tree), while access patterns contain a large number of small random memory accesses. FlacOS keeps it locally to improve access efficiency, and uses bulk synchronization to reduce the overhead of cache consistency assurance. Second, the block layer is placed locally to be compatible with traditional non-memory semantic storage devices. Additionally, we expect to enhances journaling in FlacOS to simultaneously improve reliability and scalability by integrating it with synchronization mechanism [36].

3.5 FlacOS Communication System

Leveraging shared memory can greatly accelerate interprocess communication (IPC) in FlacOS, as it completely eliminate overhead of networking or RDMA.

Shared data buffer for zero-copy IPC. FlacOS IPC is compatible with domain sockets and supports communication between processes on different nodes. The data buffer is allocated in the shared memory, enabling zero-copy data transmission between nodes. Despite being frequently accessed in the data plane, the communication access pattern for these buffers remains relatively consistent, such as streaming or read-only access that do not simultaneously modify the buffer. Consequently, shared buffers can be easily synchronized across nodes via cache invalidation.

Shared code context for migration-based RPC. Remote procedure call (RPC) represents a special form of IPC focusing on control flow transfer between services using function call semantics. FlacOS optimizes RPC through thread migration model [16, 41, 58], where the client invokes the server code by switching address space without switching the thread. To enhance efficiency and flexibility, FlacOS places the invoked service code context within shared memory for the efficient sharing of RPC services among nodes. The shared context also empowers fast process migration between nodes and efficient scaling up to support service elasticity [61, 68]. Furthermore, shared context can be part of thread runtime snapshot for fast thread creation [46] and optimized runtime sharing [7].

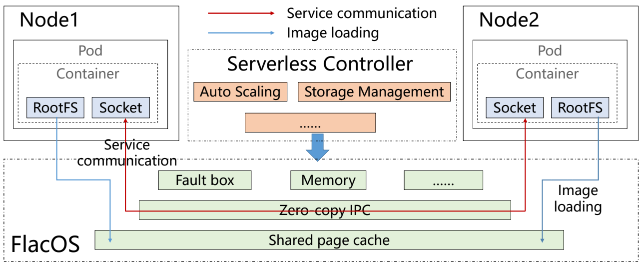

Figure 3: Serverless architecture on top of FlacOS.

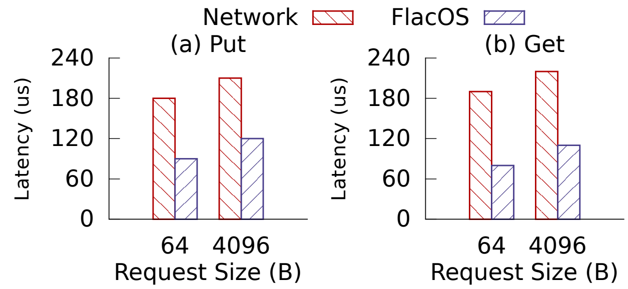

Figure 4: Redis results.

Local data structures. Socket structures that maintain communication metadata are stored in the local memory. FlacOS employs the replication-based method to synchronize metadata across nodes to achieve fast and reliable connection establishment and destination addressing..

3.6 System-wide Reliability

The key of addressing the increasing memory faults involves minimizing the failure radius and achieving rapid recovery. Separately enhancing individual component is ineffective to ensure fault tolerance over the entire system. FlacOS proposes vertical fault box and adaptive redundancy to improve system reliability.

Fault box. We propose fault box, a new abstraction for system level fault isolation. Unlike existing systems which horizontally aggregate the states of different applications together, a fault box vertically consolidates a single application's memory and status based on the application execution flow. Thus, falut box allows the complete state set of an application to be manipulated at once without triggering different system components and independent state recovery. For example, a fault box encompasses the page table, context, communication buffer, stack, and heap of an application. This prevents a single failure from propagating to multiple applications and enables efficient migration and recovery

Adaptive redundancy. Based on user configuration and task criticality, FlacOS adaptively employs different degree of reliability methods, such as periodic check-pointing [27, 52], partial replication [9, 70], and n-modular execution [21, 57].

4 CASE STUDY

4.1 Serverless Computing

We demonstrate the use of FlacOS to reconstruct the architecture of serverless computing, an important customer scenario. Our customers report three top issues in existing serverless: high (cold) startup latency during elastic scaling [7, 30, 35, 64], performance interference under high container density [1, 46], and communication cost between services (chains) [32, 42].

We present the envision of future rack-level serverless architecture based on FlacOS in Figure 3. FlacOS shared page cache and file system enable rack-wide container hot startup by sharing image and runtime data. Services are interacted using FlacOS IPC via shared memory, avoiding cross-node networking overhead. Serverless control plane utilizes the scheduling, fault tolerance, and sharing capability in FlacOS to achieve high elasticity, availability, and density.

4.2 Prototype Evaluation

We implement a prototype of some components in FlacOS, including memory management, IPC, and shared page cache. We test FlacOS prototype on both real and simulated platform. The real rack platform comprises two nodes, and each node is a Kunpeng 920 server which has 4 80-core NUMA, leading to a total of 640 cores in the rack. The two nodes are connected by HCCS memory-interconnect [62] to achieve shared memory between them. The simulation environment runs two virtual machines sharing a piece of persistent memory [23], which allows us to emulate the access latency associated with memory-interconnect. We conduct two experiments.

We run Redis tests on the Kunpeng rack platform, with Redis server and client running on separate nodes and interacting via FlacOS IPC. We compare FlacOS with a networking-based approach, where the client and server communicates using TCP/IP stack over a direct-connected Ethernet. Figure 4 compares FlacOS latency against networking of both set and request of two request sizes. The majority of the overhead in the networking method comes from software overhead, including buffer allocations, data copies, and stack processing. Thanks to direct access to the shared memory, FlacOS avoids the most software overhead and reduces the latency by 1.75-2.4 times compared to networking.

We next test container startup latency on the simulated platform using a 4GB Pytorch image. After the first node starts up a container, the second node starts another container of the same image. We focus on the startup latency on the second node. Without FlacOS, the second suffers a complete cold start which requires to load image from registry and takes 21.067s. In FlacOS, its shared page cache stores container image in the shared memory during the first node's startup. Thus, the second node directly loads image from shared memory, reducing the startup latency to 5.526s. We also measure the hot startup latency which is 3.02s. Hot startup is faster than FlacOS cold startup because cold startup still needs to download image metadata, such as manifest.

5 OPEN CHALLENGES

There are some open challenges that need hardware-software co-design, and we leave them in the future work of FlacOS.

Device sharing and aggregation. Managing devices within a rack faces three issues. 1) *Global naming and addressing.* We expect devices export a single name and address across the whole rack. For instance, all nodes have the same IP address and block namespace. This will considerably simply maintain operations in the rack. 2) *Device sharing.* A device should be available to all nodes to realize flexible request distribution and flow scheduling [29, 67]. This requires device drivers and DMA buffers to reside in shared global memory. 3) Device aggregation. In addition to sharing, a node is also expected to access all devices, even if they are attached to other nodes. This is similar to multi-rail RDMA capability [33, 43] that increases parallelism for individual I/O or flows.

Rack-wide interrupt. Existing memory-interconnect lacks efficient inter-rack interrupt support and needs to support following interrupt types. 1) *IPI.* It is necessary to be extend inter-processor interrupt to cores located in different nodes. 2) `mwait`. Global memory should be capable of triggering interrupt similar to `monitor/mwait` instructions [22], which is crucial for fast event notification and convenient debugging. 3) *Interrupt routing.* External interrupts from devices should be able to be routed to any core in any node, achieving `irq_balance` facility [24] in rack-wide.

System Bootstrapping. Bootstraping a rack-scale computer requires more integration of BIOS functionality and global shared memory. For example, data structures holding hardware description, such as memory topology and bus hierarchy, can be stored in shared memory to advertise available hardware resources to FlacOS via FDT [44] or ACPI [56].

6 CONCLUSION

This paper introduces a new operating system called FlacOS that is designed for memory-interconnected rack-scale machine. FlacOS utilizes in-rack shared memory to reconstruct system functionalities, including physical and virtual memory, file system, and IPC. We suggest that rack-scale reliability should be ensured with system-wide memory and states management instead of enhancing individual system components. Thus, FlacOS proposes fault box as a new system abstraction that vertically integrates memory of an application originated from multiple system services. We demonstrates a rack-level serverless architecture based on FlacOS to enjoy performance, elasticity, availability, and density benefits. We test FlacOS prototype using both simulated and physical rack-scale machine. With a 640-core physical rack machine, FlacOS achieves 1.75-2.4 times lower Redis request latency than network-based solutions. On a VM-based simulated platform, FlacOS improves container startup latency by 3.8 times. We believe that FlacOS paves the way for enabling OS management and control over shared memory of – and augmenting the capabilities of – the increasing prevalence of rack-scale machines.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and feedback We also thank our colleagues in Huawei for their support, including but not limited to Guangxing Deng, Qiang Li, Ruilin Li, Shixin Liu, Mingtian Zhang, Liang Li, Yi Chen, Songping Yu, and Rui Xiang. Yubo Liu is the corresponding author.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20).*

[2] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K. Aguilera. 2023. Logical Memory Pools: Flexible and Local Disaggregated Memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets'23).*

[3] Krste Asanovic. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14).*

[4] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective Replication and Sharing in an Operating System. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21).*

[5] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21).*

[6] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17).*

[7] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From warm to hot starts: leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21).*

[8] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*.

[9] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*.

[10] CXL [n.d.]. CXL Specifications: https://www.computeexpresslink.org/download-the-specification.

[11] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* 56, 11, Article 290 (July 2024), 37 pages.

[12] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* (2012).

[13] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. *CoRR* abs/2102.11245 (2021). arXiv:2102.11245 https://arxiv.org/abs/2102.11245

[14] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS'15)*.

[15] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*.

[16] Bryan Ford and Jay Lepreau. 1994. Evolving Mach 3.0 to A Migrating Thread Model. In *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*.

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

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

[19] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*.

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*.

[21] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

[22] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2021. A case against (most) context switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)*.

[23] Intel. 2021. 3D XPoint DCPMM. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.

[24] Irqbalance. [n.d.]. The new official site for irqbalance. http://irqbalance.github.io/irqbalance/

[25] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. 2015. Shoal: Smart Allocation and Replication of Memory For Parallel Programs. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*.

[26] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. KLOCs: kernel-level object contexts for heterogeneous memory systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[27] Oren Laadan and Jason Nieh. 2007. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *2007 USENIX Annual Technical Conference (USENIX ATC'07)*.

[28] 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. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*.

[29] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. 2024. Flow Scheduling with Imprecise Knowledge. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*.

[30] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC'22)*.

[31] libfam-atomic [n.d.]. Fabric Attached Memory Atomics libary: https://github.com/FabricAttachedMemory/libfam-atomic.

[32] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. 2024. FUYAO: DPU-enabled Direct Data Transfer for Serverless Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[33] Jiuxing Liu, A. Vishnu, and D.K. Panda. 2004. Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*.

[34] Ming Liu. 2023. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*.

[35] Yubo Liu, Hongbo Li, Mingrui Liu, Rui Jing, Jian Guo, Bo Zhang, Hanjun Guo, Yuxin Ren, and Ning Jia. 2025. FlacIO: Flat and Collective I/O for Container Image Service. In *23rd USENIX Conference on File and Storage Technologies (FAST'25)*.

[36] Yubo Liu, Hongbo Li, Yutong Lu, Zhiguang Chen, Nong Xiao, and Ming Zhao. 2020. HasFS: optimizing file system consistency mechanism on NVM-based hybrid storage architecture. *Cluster Computing* 23, 4 (Dec. 2020), 2501–2515.

[37] Yubo Liu, Yutong Lu, Zhiguang Chen, and Ming Zhao. 2020. Pacon: Improving Scalability and Efficiency of Metadata Service through Partial Consistency. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS'20)*.

[38] Yubo Liu, Yuxin Ren, Mingrui Liu, Hongbo Li, Hanjun Guo, Xie Miao, Xinwei Hu, and Haibo Chen. 2024. Optimizing File Systems on Heterogeneous Memory by Integrating DRAM Cache with Virtual Memory Management. In *22nd USENIX Conference on File and Storage Technologies (FAST'24)*.

[39] Jialun Lyu, Marisa You, Celine Irvene, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, Savyasachi Samal, Ioannis Manousakis, Lisa Hsu, Preetha Subbarayalu, Ashish Raniwala, Brijesh Warrier, Ricardo Bianchini, Bianca Schroeder, and Daniel S. Berger. 2023. Hyrax: Fail-in-Place Server Operation in Cloud Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*.

[40] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[41] Gabriel Parmer. 2010. The Case for Thread Migration : Predictable IPC in a Customizable and Reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT'10)*.

[42] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*.

[43] Ying Qian and A. Afsahi. 2006. Efficient RDMA-based multi-port collectives on multi-rail QsNet/sup II/ clusters. In *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*.

[44] Jaworowski RafaÅĆ. 2010. Flattened Device Trees for Embedded FreeBSD. In *BSDCan* (Ottawa, Canada).

[45] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. 2024. Scaling Up Memory Disaggregated Applications with SMART. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[46] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. 2020. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*.

[47] Yuxin Ren, Guyue Liu, Gabriel Parmer, and Björn B. Brandenburg. 2018. Scalable Memory Reclamation for Multi-Core, Real-Time Systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'18)*.

[48] Yuxin Ren and Gabriel Parmer. 2019. Scalable Data-structures with Hierarchical, Distributed Delegation. In *Proceedings of the 20th International Middleware Conference (Middleware'19)*.

[49] Yuxin Ren, Gabriel Parmer, and Dejan Milojicic. 2020. Bounded incoherence: a programming model for non-cache-coherent shared memory architectures. In *Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'20)*.

[50] Yuxin Ren, Gabriel Parmer, and Dejan Milojicic. 2020. Ch'i: Scaling Microkernel Capabilities in Cache-Incoherent Systems. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'20)*.

[51] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*.

[52] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. 2012. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*.

[53] Ori Shalev and Nir Shavit. 2006. Predictive log-synchronization. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06)*.

[54] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.

[55] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

[56] UEFI Forum, Inc. [n.d.]. *Advanced Configuration and Power Interface (ACPI) Specification Version 6.4, https://uefi.org/sites/default/files/resources/ACPI_Spec_6_4_Jan22.pdf*.

[57] Jonas Vinck, Bert Abrath, Bart Coppens, Alexios Voulimeneas, Bjorn De Sutter, and Stijn Volckaert. 2022. Sharing is caring: secure and efficient shared memory support for MVEEs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22)*.

[58] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. 2015. SPeCK: a kernel for scalable predictability. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*.

[59] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*.

[60] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*.

[61] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*.

[62] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* 41, 5 (2021), 67–75.

[63] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving program locality in the GC using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*.

[64] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[65] Yekang Zhan, Haichuan Hu, Xiangrui Yang, Shaohua Wang, Qiang Cao, Hong Jiang, and Jie Yao. 2024. RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths. In *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC'24)*.

[66] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*.

[67] Zhiyu Zhang, Shili Chen, Ruyi Yao, Ruoshi Sun, Hao Mei, Hao Wang, Zixuan Chen, Gaojian Fang, Yibo Fan, Wanxin Shi, Sen Liu, and Yang Xu. 2024. vPIFO: Virtualized Packet Scheduler for Programmable Hierarchical Scheduling in High-Speed Networks. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM'24)*.

[68] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. 2023. BeeHive: Sub-second Elasticity for Web Services with Semi-FaaS Execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*.

[69] Yijie Zhong, Minqiang Zhou, Zhirong Shen, and Jiwu Shu. 2024. UniMem: redesigning disaggregated memory within a unified local-remote memory hierarchy. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'24)*.

[70] Diyu Zhou and Yuval Tamir. 2022. RRC: Responsive Replicated Containers. In *2022 USENIX Annual Technical Conference (USENIX ATC'22)*.