Move your code, not your data

Michael Giardino
Computing Systems Lab
Huawei Technologies
Zürich, ZH, Switzerland
michael.giardino@huawei.com

Siddharth Gupta
Computing Systems Lab
Huawei Technologies
Zürich, ZH, Switzerland
siddharth.gupta3@huawei.com

Lukas Humbel
Huawei Technologies
Zürich, ZH, Switzerland
lukas.humbel@huawei.com

Rene Mueller
Computing Systems Lab
Huawei Technologies
Zürich, ZH, Switzerland
rene.mueller@huawei.com

Anirban Nag
Computing Systems Lab
Huawei Technologies
Zürich, ZH, Switzerland
anirban.nag@huawei.com

Abstract

Memory requirements in the datacenter have increased dramatically, however current memory technologies are unable to keep up due to poor density scaling and pad-limited integration. Furthermore, such expensive memory resources are not efficiently utilized, in part, because of memory stranding issues. These factors have resulted in the emergence of disaggregated or far memory as a solution to increase capacity beyond a single node, and enable memory pooling to address underutilization. However, far memory comes with many challenges which may hinder its adoption. In this paper, we highlight these challenges ranging from high access latency, new failure domains, non-trivial data sharing mechanisms, and added infrastructure cost. We then enumerate a broad set of solutions to handle high access latency of far memory and introduce a taxonomy in terms of implementation complexity. Finally, we motivate an under explored solution—shipping code to data—using various function shipping techniques.

CCS Concepts

• **Computer systems organization → Heterogeneous (hybrid) systems**; **Distributed architectures**; • **Networks → Programming interfaces**.

Keywords

far memory, memory wall, function shipping

ACM Reference Format:
Michael Giardino, Siddharth Gupta, Lukas Humbel, Rene Mueller, and Anirban Nag. 2025. Move your code, not your data. In *4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25), March 30, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3723851.3723856

The authors are listed alphabetically.

This work is licensed under a Creative Commons Attribution International 4.0 License.

HCDS '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1470-2/25/03
https://doi.org/10.1145/3723851.3723856

1 Introduction

The gap between CPU and memory performance has been steadily widening [16], even as per-core CPU improvement has slowed [54]. This gap, often termed the "memory wall" [56], has long had significant implications in how we design systems and applications. The slowing of DRAM process scaling [27, 32] plays a large part, preventing the increase in performance and density seen by CPUs. Other challenges include the pad-limited physical connection of DDRx DRAM modules to CPUs [9], high power consumption (including regular refreshing), and the load of high-frequency parallel buses, limiting maximum frequency. The introduction of high-bandwidth memory primarily addresses bandwidth, while new memory technologies (e.g., PCM, ReRAM) attempt to improve density, persistence, and energy consumption. However, none of these proposed solutions scale the memory wall, and the growing data sizes only push larger capacity devices further up the memory hierarchy; previously to high-capacity non-volatile main memories such as Optane™ and now to a new class of memory, connected via non-traditional interconnects such as Compute Express Link (CXL).

In this paper, we examine the challenges introduced by far memory and introduce a taxonomy to help systematize disparate ways of remote memory integration. From this taxonomy, we experimentally motivate a less common approach, function shipping code to data, for dealing with memory latency.

2 Background

The strongest voice for disaggregation comes from the cloud. Poor utilization rates of resources, especially expensive DRAM, have led to proposals for the ability to allocate unused memory from remote nodes [10]. More radical architectures include entire CPU-less memory pools [25] or completely disaggregated systems with composable nodes made of memory, compute and storage. Cloud providers often show CPU utilization rates hovering around 40% [40, 53], while VM hosting servers report 15-30% unused physical memory [31, 35] with allocated but unused memory even higher around 30% [31, 35]. One major cause of this is the granularity of virtual machine instances which come with a fixed core-memory ratio, due to the physical layout of the underlying machine. Managed services such as software-as-a-service (SaaS) and function-as-a-service (FaaS) attempt to handle this problem by having providers manage the underlying infrastructure allocation. But the lack of customizability and vendor lock-in (especially in SaaS), and the high operational cost relative to infrastructure as a service (IaaS) (and on-premises hardware) can make this less desirable to customers [20]. While there have been proposals for more flexible FaaS allocations, most remain tied to a fixed core-memory ratio [37].

Besides bin packing of sub-physical machine sized VMs, there exist an important class of applications such as databases, analytics, scientific computing, and machine learning which require memory far beyond the scale of the largest physical node. While structured, parallel workloads such as those found in scientific computing are carefully designed, allocated, and scheduled, other more unstructured, dynamic, and memory intensive workloads may benefit the most from far memory. By moving main storage to far memory, table spaces can be shared among multiple database processes running on different nodes without requiring data replication [33]. Similarly, large index structures as used for nearest-neighbor search in vector databases and retrieval-augmented generation (RAG) can be shared in far memory [21, 24]. That said, even if these workloads can make use of far memory, the placement of data within memory is critical for performance [33].

2.1 Enabling Technologies

While there have been distributed memory systems over the years, the current industry is centering around a few primary technologies. With the expansion of GPUs and other accelerators for AI, we have seen a proliferation of interconnect technologies that attempt to go beyond traditional PCI-attached devices. AMD's Infinity Fabric and NVIDIA NVLink both integrate GPUs more closely with the CPUs and have seen topologies expand beyond a simple hub and spoke design. The most broadly researched standard is the Compute eXpress Link (CXL) [50]. There are several standards of CXL, each with increasing capabilities. Initial versions offered some limited concept of coherence and no sharing, while later versions added pooling (2.0) and sharing (3.0) and a model that is closer to symmetric coherence.

Our analysis aims to be interconnect agnostic in the sense that we are interested in a superset of available, proposed, and potential features that would allow for the pooling of memory. Thus, we will be focusing on these more advanced interconnect technologies that could be described as *fabrics* in which memory is not only connected directly to a CPU, but is accessible within a rack (inter-node) or beyond. These specific features of individual technologies vary significantly and are constantly evolving, thus a detailed comparison is beyond the scope of this paper. However, we can identify certain capabilities that are necessary or beneficial for memory pooling. The most relevant features include byte addressability, load-store access, sharing, coherence, and peer-to-peer communication. We will examine the application of these in the next section and explore the challenges introduced in Section 3.

2.2 Terminology

Before we examine problems posed and opportunities presented by modern memory systems, it is important to examine the ecosystem, and in doing so, define our terms.

In this paper, a **tiered memory system** is defined as one that has multiple memories beyond last-level cache (LLC) and before block storage that with different *performance* characteristics. These differences are due multiple memory technologies connected to the CPU memory controllers such as a DRAM-NVMM system and/or a non-traditional CPU-DRAM topology such as a system with CPU- and CXL-attached memory. **Heterogeneous memory** is closely related however we will use it to describe systems only if they contain different physical memory technologies (e.g., DDR4 vs DDR5 vs HBM vs PCM) but not memories of the same technology but with different latency or bandwidth characteristics (e.g., NUMA or CXL-attached DRAM). This distinction is helpful for identifying energy and density differences between systems.

We use the term **far memory** to describe byte-addressable load-store memories that are not directly connected to a specific CPU's memory controllers, including cross-socket NUMA memory, CXL-attached Type-3 devices and memory-semantic SSDs. *Disaggregated* and *pooled* memory are often used interchangeably, however we will try to make a distinction between these terms for clarity. **Disaggregated memory** is a form of CPU-less far memory that can be allocated by one or more nodes via CXL, RDMA, etc., also sometimes referred to as zero-CPU NUMA (zNUMA) [19, 35]. **Pooled memory** can be either disaggregated or unallocated on a traditional node and can be allocated by systems via network or interconnect.

These systems can be further distinguished by the scope of addressing, level of sharing, and coherence. On one extreme, we have traditional NUMA systems which share a single address space, can share arbitrary memory between CPUs, and caches are kept coherent. A pointer passed from one core to another can be successfully dereferenced, and indeed the process itself can be migrated since they share an operating system. Such systems are very challenging to engineer and scale. At the other extreme, we have a traditional distributed system which shares nothing, where all communications are explicit and done via RDMA, RPC, etc. The traditional service-based cloud operates on systems such as these, and while the communication mechanisms themselves have become very efficient, they still contribute to the datacenter tax and their usage is only growing [46].

Future systems, especially as they expand to the rack scale, will likely lie somewhere in-between, with configurable domains of addressing and coherence. Whatever the future system looks like, there will be significant challenges in integration.

3 Far Memory Challenges

While there are good reasons for expansion via tiered memory, we discuss the primary areas in which problems arise. The *latency* of local DRAM is already difficult or impossible to hide, and as with NVMM, far memory latency is even higher. When we build shared-memory systems on top of a higher latency, scalable fabric, there are significant challenges to correctness and consistency. The common system assumption that memory is reliable no longer holds, and different *failure modes* need to be handled. Finally, the *cost/benefit* of far memory is still unclear.

3.1 Hiding latency

With the enormous bandwidth available in modern interconnects, the primary performance challenge with far memory for general

Table 1: Number of instructions necessary to hide latency

Memory Type	Latency	6-wide 2 GHz	6-wide 3.4 GHz	6-wide 4 GHz
L1/L2	<10 *ns*	120	204	240
L3 [13, 15]	30 *ns*	360	612	720
DDR5 [13]	100 *ns*	1,200	2,040	2,400
Cross socket [12]	200 *ns*	2,400	4,080	4,800
CXL [35]	300 *ns*	3,600	6,120	7,200
Optane™ [45]	350 *ns*	4,200	7,140	8,400
Intra Rack CXL	400 *ns*	4,800	8160	9600
Context Switch	1 *μs*	12,000	20,400	24,000
Inter Rack CXL	10 *μs*	120,000	204,000	240,000

purpose computing is latency. Fortunately, most fundamental structures of modern CPUs exist to hide memory latency. Caches, out-of-order execution, multi-wide issue, simultaneous multithreading (SMT), branch predictors, and prefetchers are all present to hide memory latency. It is well-understood that these features cannot hide the "killer microsecond" [4] from low-latency I/O. However, as research has shown, they are not enough to hide the latency of remote memory [35]. In fact, we argue that given the multi-wide issue and high frequency of modern CPUs, Table 1 shows that even large reorder buffers (e.g., Sapphire Rapids [11] has 512 ROB entries) are insufficient to hide even an LLC miss, let alone a remote memory access. The killer microsecond is now sub-microsecond and shrinking.

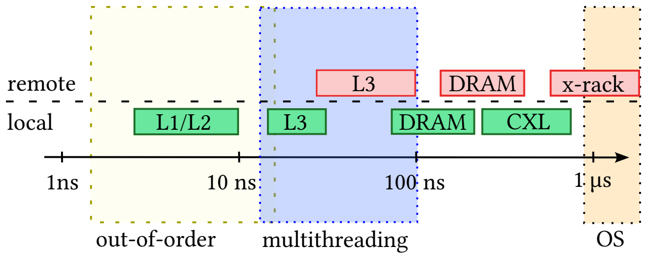

Figure 1: Various local (green) and remote (red) members of the memory hierarchy spans a broad range of latency. The yellow box shows the latency ranges well handled by out-of-order execution. The blue box shows the limits of multithreading techniques while the orange box shows where the OS begins to take over. The gap around DRAM (local and remote) and CXL memory is the focus of this paper.

Figure 1 shows the various techniques available for different latency bands in modern systems. The difficulty of hiding this latency is demonstrated clearly in Li *et al.* which shows that moving data from local to standard NUMA latency (78 ns vs. 142 ns) slowdowns of up to 50% are seen [35]. When latencies are increased from 115 ns to 255 ns (lowest observed latency of real CXL1.1 hardware [52]), slowdowns can approach 100%. Increasing the number of threads through SMT [22, 29] may hide more latency than reordering, but it comes back to the ratio between potential instruction issue rate and memory latency. If a processor can issue hundreds or thousands of instructions before a memory request is serviced, it is hard to find enough parallelism, rendering many of these structures unhelpful for far memory. Large LLCs (passing 1 GB in the highest end processors [3]) are popular additions to modern CPUs for both power and performance reasons. However, for many of the big data applications that would make use of far memory, even if the primary working set fits in LLC the secondary set cannot and thus, larger caches offer diminishing returns [48, 49]. Moreover, these enormous LLCs are, in reality, physically distributed among each core, giving optimal latency to a slice of 10-20 MB, forcing any larger working sets to spill to higher-latency neighboring caches [1, 15].

If the required data is predictable, it can be prefetched into the cache to be ready when it is needed. Even if the CPU can accurately identify prefetches many hundreds or thousands of instructions away from use, there still are questions of timeliness and cache capacity. In cloud workloads [23] and in higher latency NVMM [5], hardware prefetchers can even harm performance.

3.2 Sharing

Another significant hurdle in far memory systems is the challenges of sharing data. CXL, for example, has gone through many augmentations of its sharing abilities over various versions. Even though CXL ostensibly supports rack-scale interconnection, it is not clear how these devices will integrate or scale to the rack level.

Especially between homogeneous CPUs, there is an implicit contract that shared memory viewed by two different cores will have some model of consistency. With low latency and a relatively small number of nodes, hardware-based coherence offers significant advantages in efficiency and programmability. However, previous research in rack-scale coherence has shown significant limitations as the number of nodes grow and, most importantly, the amount of shared mutable data increases. The more writers and readers to a particular shared memory region, the greater the pressure on coherence mechanisms. If, due to scaling difficulties, hardware-based coherence is abandoned in favor of software techniques of consistency, this shifts a significant burden onto the application designer.

3.3 Failure

An often overlooked issue in modern pooled memory systems (but a first-order concern in large-scale distributed ones) is how to handle failures [14]. Load-store shared-memory systems such as a standard multi-socket NUMA system have an implicit assumption that memory is reliable. Therefore, if a portion of physical memory is not available or a load never returns, there are few mechanisms to handle this gracefully either in hardware or by the OS. Transient errors may be handled by ECC but permanent ones are dealt with, at best, by the OS via hot(un)plug [38] and terminating the affected processes. In the worst case, the result is a kernel panic that brings the whole node down. From the perspective of software, the failing system exhibits a fail-stop behavior. Large, mission critical or massively parallel systems rely on redundant execution, coding techniques, or software-managed resiliency for failures, e.g., by using replication and a consensus protocol [8]. A system that uses disaggregated or pooled memory can be composed of multiple components which may fail independently. The likelihood of a failure increases with system scale. Furthermore, when a only subset of the compute or memory components fail, a partial application or VM failure can occur, which is more difficult to handle than a fail-stop failure. Partial failures may also come in the form of dropped/delayed network packets (not uncommon even in well-engineered modern datacenters [36]). While networking protocols are fully capable of handling such situations, the hardware load-store unit and memory manager are not, introducing several complex failure modes. However, the manner in which pooled memory is proposed in the cloud often neglects these challenges, leaving the mechanisms required to recover unclear.

Additional challenges arise when memory is shared among different VMs or processes. A shared memory region may remain allocated even after all VMs or processes accessing it have terminated. For example, an in-memory database system may keep its shared buffer pool alive when all engine processes are shut down during migration and maintenance operations. Hence, shared memory can outlive the compute processes and effectively behaves like persistent memory, even though the underlying DRAM is volatile. We may learn lessons from previous research into NVMM such as Optane™ [55]. One of the major insights from NVMM is that having a easy-to-understand border between "memory" and "storage" (usually crossed with an explicit syscall e.g., `write()` and `sync`), is taken for granted in system design, and, thus, when persistence is brought across this memory-storage frontier, complex crash consistent mechanisms must be developed [58]. In the same way much (but not all [47]) early optimism into NVMM neglected the difficulty of integrating persistent memory into existing systems, we must plan for the certainty of failure in pooled memory systems.

Fortunately, far memory itself does not necessarily have to come with all aforementioned challenges. Consider a VM that allocates remote memory for expansion: if that memory is not shared with any other VM and released when the VM is terminated, the failure behavior of the remote memory is akin to that of local memory in a NUMA-system; if the far memory node goes down, the VM simply fails. In case the VM host crashes, the hypervisor of the memory node will eventually notice the failure and free the allocated memory. Note that the same applies even in a scenario in which far memory is shared between two VMs as long as they are in the same failure domain, i.e., if one fails so does the other. When using far memory, it essential to consider the relevant failure domains—as in any distributed system.

3.4 Costs

Hyperscalers observe that DRAM costs (and associated energy) approach 40% of total datacenter usage [42]. This motivates the need not to waste or "strand" memory in the cloud, but, we argue, it makes a strong argument *against* disaggregated memory (i.e., CPU-less memory nodes). This is part of the case made by Levis *et al.* against CXL memory pooling [34], in which they argue that the monetary cost of DRAM is already very high so if one adds specialized CXL hardware, including switches, the cost goes even higher, approaching the cost of a normal server. However, if next-generation interconnects are able to replace existing Ethernet-based rack networking infrastructure, the additional cost of interconnection will be borne anyway.

A reasonable counterargument can be made that *much of this memory already exists* in the form of older DDR3 and DDR4 DIMMs, with similar capacity but no longer compatible with state-of-the-art processors. These DIMMs represent a significant amount—nearly half by some estimates [41]— of the embodied carbon of a server, not to mention the CAPEX. Some have suggested that these older technology memories could constitute a lower-performance tier of memory [6], but, as we examined in Section 3.1, hiding DRAM latency is already a challenge, made worse by greater distance and older technology. One alternative is to pair more efficient, simpler cores, possibly in a manycore format [39, 51], with this older memory, allowing for the execution of memory-intensive functions local to the memory, discussed in more detail in Section 5.

4 Taxonomy

To clarify the potential solutions available for using far memory, we present in Figure 2 a taxonomy of techniques available to system designers for integrating far memory. From left, the primary distinction is whether the application manages far memory directly. If it does not, it may fall to the operating system (first branch) or hypervisor (second branch) to transparently allocate, measure hotness and, migrate pages. Such approaches include Linux' automatic NUMA balancing [26], TPP [42], Infiniswap [18] and others [17, 28, 57] which rely on various forms of *hot page detection* to correctly move pages between memory regions. Most of these techniques can be implemented at both OS or hypervisor level, but their effectiveness may vary. Implementation in the hypervisor enables memory overcommitment at the cloud level. It also provides a global view and enables, for example, finding the globally coldest page. Observing processes, on the other hand, allows better understanding of application access patterns.

Our taxonomy distinguishes cloud provider services (often marketed as *X*-as-a-service) from the hypervisor. Thus, if far memory is transparent to the OS and the hypervisor, it can still be used for infrastructure. If the application is aware of different regions of memory, then we argue that the primary bifurcation occurs between whether the far memory has load-store semantics or uses some form of direct memory access. When load-store access is provided, the remaining question is whether it uses load-store on *local memory* (by moving the execution to data) or the application steers allocations (using e.g., `pmalloc` or `madvise`).

In general, the complexity of application integration and performance both increase (as shown on the right side of Figure 2) from bottom to top. Allowing the provider to make use of far memory requires no knowledge from the software developer (nor indeed the operating system), but the improvements can be low and may even hurt performance (in the case of overprovisioning). On the other hand, steering memory allocation or decomposing applications into shippable functions increase application complexity, but yield larger gains.

All of these techniques in some form or another have shown value, but with various tradeoffs. For the remainder of this paper, we focus on a single area we believe has been underexamined, namely function shipping.

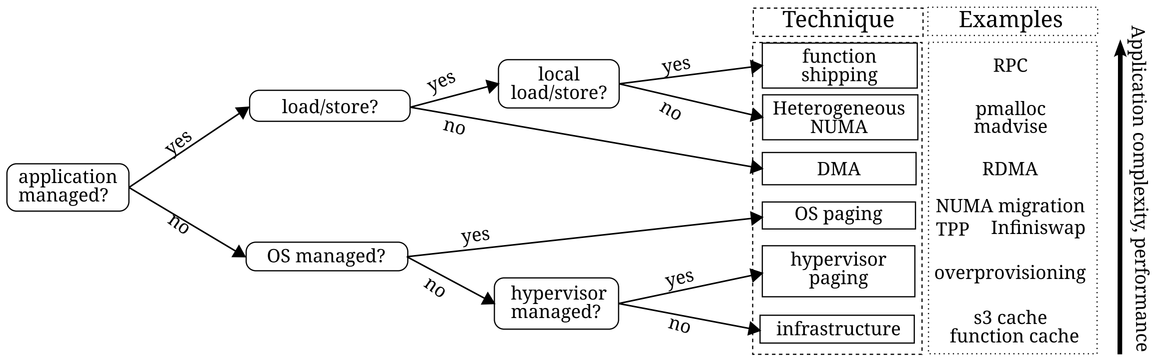

Figure 2: The solutions for far memory can be placed in a taxonomy. Square boxes show techniques for managing far memory and the right box gives examples. Application complexity but also potential performance increase from bottom to top.

5 Function Shipping

Traditional data-locality aware computing distributes subsets of data among a collection of nodes. In the case of database sharding or map-reduce, knowledge of both where the data is located and which data will be accessed, must be known at the time of task scheduling, i.e., before execution. In some cases, this assumptions hold, and thus this technique performs well. However, there exist many classes of applications in which *a priori* knowledge of the data being accessed is not available. In such cases, job scheduling is suboptimal and necessitates higher-latency remote accesses. As memory latency increases, if it cannot be hidden by prefetching (as in the case of data dependent loads) the performance degrades heavily. At this point, the best solution may be to dynamically send the execution to data. A possible solution is processing in memory (PIM), which moves operations *into* memory but for general purpose computing it has many drawbacks. The physical structure of the memory spreads data across many ranks, banks, and channels, so the placement of compute logic is not obvious, as tasks besides memsets and bitwise operations become very complex. Related, the semantics, consistency, and correctness of such operations are unclear. Finally, DRAM is already expensive and adding custom (but limited) logic will only drive up the price.

In such scenarios, we propose *function shipping* using stateless serverless functions [30] or a stateful continuations [44] to migrate execution to the data. The ability to move closer to the data allows for memory intensive (i.e., low operations per byte) to become significantly more efficient.

In order to obtain this flexibility, there are several possibilities available. On the one extreme, specific memory intensive functions could be accelerated near memory without interrupting or allocating resources on the remote node. For example, gathers, scans, or reductions could occur in a specialized accelerator on the remote node. It can have a highly specialized architecture, while not interrupting the remote CPUs. The drawback, however, is limited functionality, especially if this is a fixed-function processor, and the addition of data types or operations may be difficult or impossible. A more programmable accelerator could be imagined, something resembling a specialized CISC processor with new functions that can be updated via microcode (e.g., Amazon Trainium [7]). This still limits functionality and requires writing code for the specific (mutable) ISA or relying on an API or library, but could be justified for commonly used functions.

On the other extreme is simply sending the entire virtual machine to the data as is done in live migration, allowing for transparent movement of execution to code with no programmer input. This naïve approach has several obvious problems, such as the size of the VMs data and the latency incurred by partial migrations, leaving critical data structures on the remote node. The data of an entire virtual machine may be much too large to quickly migrate, especially if the operation is relatively short. If all the VM's memory is not migrated, what remains may be critical and latency sensitive, slowing down the operating system. Allocation of sufficient vCPUs would also be required by the hypervisor.

A more pragmatic solution lies somewhere between, in which specific functionality is available in the form of either a stateless function or a stateful continuation. This allows for the invocation or migration of functionality in a lightweight modern sandbox such as a virtual machine (e.g., microVM, unikernel), container, or runtime (e.g., Wasm, GraalVM). While this still requires "core stealing" from available CPUs on the remote node, given these operations are seldom compute intensive (and are thus memory bound), they could be handled by a much smaller vCPU allocation than a virtual machine. Moreover, CPU resources would not have to be used the entire time memory is allocated, allowing for more efficient use of resources (in the case of cloud providers) and lower cost (in the case of users). In the case of using reusing older technologies in a memory blade, simple, low-power cores could be used to run these memory-intensive functions. Whatever the solution used, the shared address space and load-store access allows for a fault-free fall back to (albeit slower) NUMA-like access.

5.1 Function Shipping Cost

To motivate this approach, we examine the overhead of function shipping using various techniques. We emulate a remote memory system using a two-socket NUMA system with local memory latency of ≈110 ns and remote (cross-socket) latency of ≈258 ns. We wrote a testing framework (≈5k C++ LoC) that uses the multi-process model via POSIX shared memory, allowing for the sharing of memory and the communication between cores via POSIX

Table 2: Function Shipping Examples

Shipping method	shipping overhead	min. number of remote accesses
thread migration	62 μs	419
FS RPC w/o code	52 μs	351
FS RPC w/ native code shipped	115 μs	788
FS RPC w/ Wasm code shipped	8,300 μs	57,448

message queues. We evaluate a microbenchmark consisting of a randomly shuffled linked list, with the dataset on the remote node.

Table 2 shows the results of the tested function shipping methods, their overheads, and the number of equivalent remote memory accesses required to match the overhead of migration. For a baseline, we use normal OS-controlled thread migration which takes ≈62 μs (or the equivalent of 419 remote accesses). We note that this baseline necessitates *both nodes running the same operating system instance* and thus is not likely in a disaggregated system which may consists of multiple independent nodes (and operating systems). The basic communication overhead of an RPC in which the code already is running in a thread on the remote node is 52 μs. This is equivalent to having a remote application running all the time. The next version ships the function as a dynamically loaded shared object (.so). It loads the .so into shared memory on the local node, and then sends a reference to it to the remote node, which then begins executing it (via implicit transfer). This requires 115 μs but has the advantage of allowing for the execution of arbitrary functions on a remote node. Finally, to move closer to a serverless technique, we provide Wasm-based sandboxing. This begins by sharing the code as a Wasm module and, as in the previous example, sending a reference of its location to the remote node, which creates a Wasm instance and executes the function on the remote data. This has the largest overhead (8,300 μs), however it provides strong isolation guarantees.

In these examples, function shipping can be surprisingly cheap, which can be offset by a relatively small number of remote accesses. When shipping native code, performance is equal to local execution (because it now is) and shipping Wasm provides nearly identical performance to native code (110 ns vs. 113 ns per access).

5.2 Shipping Query Plans

We now show more complete example, shipping query plans in DuckDB [43]. We use the same NUMA machine from the previous experiment with a remote memory latency approximately 2.5× that of local accesses. Table 3 shows the end-to-end performance of a Group-By aggregation of various selectivity and cardinality using local memory, remote memory, and query plan shipping via Substrait [2] (see Figure 3). *Shipping* includes the time for shipping the query plan, the query execution, and the serialization of query results. *Table local* is the ideal speedup if the table is placed in the local NUMA node. As one would expect, performance is best when the table is local, outperforming remote access anywhere from a few percent to over 90%, depending on data characteristics.

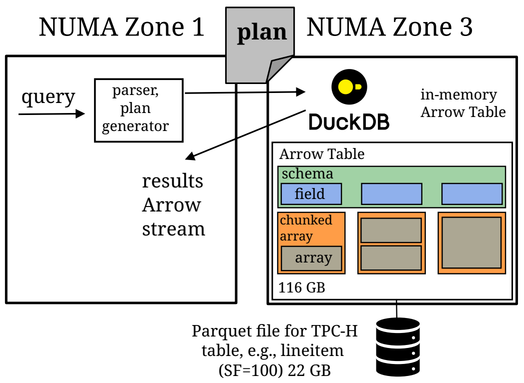

Figure 3: We simulate the Arrow table on a remote node (NUMA zone 3) and ship the query plan via Substrait.

Table 3: Remote Access vs. Near Data Processing in DuckDB

sel.	groups	end-to-end time: table location remote	end-to-end time: table location local	end-to-end time: plan ship	speedup over remote by local	speedup over remote by ship
25%	1	4.2 s	3.5 s	3.6 s	1.2×	1.2×
25%	1 k	4.2 s	3.6 s	3.6 s	1.2×	1.2×
25%	250 k	7.1 s	6.8 s	9.9 s	1.0×	0.7×
4%	1	3.4 s	1.8 s	1.8 s	1.9×	1.9×
4%	1 k	3.4 s	1.8 s	1.8 s	1.9×	1.9×
4%	250 k	3.9 s	2.4 s	2.8 s	1.6×	1.4×
4%	2.5 M	4.6 s	2.8 s	14.9 s	1.6×	0.3×

In most cases, the plan shipping is only slightly slower than a purely local execution, outperforming remote access by 1.2-1.9×, with the decrease in performance due to the costs of shipping and (de-)serialization of plans and query results. However when the cardinality grows, the performance of shipping the query plan decreases, at the extreme, degrading below simply accessing the data remotely due to the high cost of (de-)serialization of the results.

This experiment demonstrates that shipping compute in the form of entire query plans is possible and can achieve performance equivalent to data-local execution.

6 Conclusions

In this paper, we set out to summarize and systematize the problems introduced by far memory and the associated solutions to integrating it. If even a handful of the wide variety of proposed systems and applications succeed, system designers will require many different techniques for dealing with their challenges. Moving data to code has long been the primary target of researchers, but we believe that function shipping should also be a major focus of systems researchers, and we present a motivational example using a modern workload. These complex systems will need a variety of solutions, some repurposed from the past and some entirely new.

References

[1] 2023. wrBench: Comparing Cache Architectures and Coherency Protocols on ARMv8 Many-Core Systems. *Journal of Computer Science and Technology* 38, 6

(2023), 1323–1338.

[2] Substrait Management Commitee 2025. *Substrait: Cross-Language Serialization for Relational Algebra*. Substrait Management Commitee. https://substrait.io

[3] Advanced Micro Devices. 2023. *AMD EPYC 9684X*. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9684x.html

[4] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (mar 2017), 48–54. doi:10.1145/3015146

[5] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-bench: benchmarking persistent memory access. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2463–2476. doi:10.14778/3551793.3551807

[6] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (mar 2023), 30–38. doi:10.1109/MM.2023.3241586

[7] Nafea Bshara. 2024. AWS Trainium: The Journey for Designing and Optimization Full Stack ML Hardware. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 4. doi:10.1145/3620666.3655592

[8] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1849–1862. doi:10.14778/3229863.3229872

[9] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. 2023. A Case for CXL-Centric Server Processors. arXiv:2305.05033 [cs.AR] https://arxiv.org/abs/2305.05033

[10] Ho-Ren Chuang, Karim Manaouil, Tong Xing, Antonio Barbalace, Pierre Olivier, Balvansh Heerekar, and Binoy Ravindran. 2023. Aggregate VM: Why Reduce or Evict VM's Resources When You Can Borrow Them From Other Nodes?. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 469–487. doi:10.1145/3552326.3587452

[11] clamchowder. 2021. Popping the Hood on Golden Cove. https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/

[12] clamchowder. 2023. Core to Core Latency Data on Large Systems. https://chipsandcheese.com/2023/11/07/core-to-core-latency-data-on-large-systems/

[13] clamchowder. 2023. Sapphire Rapids: Golden Cove Hits Servers. https://chipsandcheese.com/2023/03/12/a-peek-at-sapphire-rapids/

[14] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (Koblenz, Germany) *(DIMES '23)*. Association for Computing Machinery, New York, NY, USA, 23–30. doi:10.1145/3609308.3625268

[15] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 17 pages. doi:10.1145/3302424.3303977

[16] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (May 2024), 33–39. doi:10.1109/MM.2024.3373763

[17] Michael Giardino, Kshitij Doshi, and Bonnie Ferri. 2016. Soft2LM: Application Guided Heterogeneous Memory Management. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 1–10. doi:10.1109/NAS.2016.7549421

[18] 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)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[19] Minho Ha, Junhee Ryu, Jungmin Choi, Kwangjin Ko, Sunwoong Kim, Sungwoo Hyun, Donguk Moon, Byungil Koh, Hokyoon Lee, Myoungseo Kim, Hoshik Kim, and Kyoung Park. 2023. Dynamic Capacity Service for Improving CXL Pooled Memory Efficiency. *IEEE Micro* 43, 2 (2023), 39–47. doi:10.1109/MM.2023.3237756

[20] David Heinmeier Hansson. 2022. *Why we're leaving the cloud*. https://world.hey.com/dhh/why-we-re-leaving-the-cloud-654b47e0

[21] Jon Hermes, Josh Minor, Minjun Wu, Adarsh Patil, and Eric Van Hensbergen. 2024. UDON: A case for offloading to general purpose compute on CXL memory. arXiv:2404.02868 [cs.ET] https://arxiv.org/abs/2404.02868

[22] Jason Howard. 2023. The First Direct Mesh-to-Mesh Photonic Fabric. In *2023 IEEE Hot Chips 35 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–17. doi:10.1109/HCS59251.2023.10254719

[23] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. 2024. Limoncello: Prefetchers for Scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 577–590. doi:10.1145/3620666.3651373

[24] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

[25] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-centric Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers* (Portland, OR, USA) *(ROSS '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 1 pages. doi:10.1145/2768405.2768406

[26] Linux kernel development community. 2024. Documentation for NUMA Balancing. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#numa-balancing

[27] Kanguk Kim, Youngwoo Son, Hoin Ryu, Byunghyun Lee, Jooncheol Kim, Hyunsu Shin, Joonyoung Kang, Jihun Kim, Shinwoo Jeong, Kyosuk Chae, Dongkak Lee, Ilwoo Jung, Yongkwan Kim, Boyoung Song, Jeonghoon Oh, Jungwoo Song, Seguen Park, Keumjoo Lee, Hyodong Ban, Jiyoung Kim, and Jooyoung Lee. 2023. 14nm DRAM Development and Manufacturing. In *2023 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 1–2. doi:10.23919/VLSITechnologyandCir57934.2023.10185314

[28] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. 2019. Page migration support for disaggregated non-volatile memories. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 417–427. doi:10.1145/3357526.3357543

[29] Georgios K. Konstadinidis, Hongping Penny Li, Francis Schumacher, Venkat Krishnaswamy, Hoyeol Cho, Sudesna Dash, Robert P. Masleid, Chaoyang Zheng, Yuanjung David Lin, Paul Loewenstein, Heechoul Park, Vijay Srinivasan, Dawei Huang, Changku Hwang, Wenjay Hsu, Curtis McAllister, Jeff Brooks, Ha Pham, Sebastian Turullols, Yifan Yanggong, Robert Golla, Alan P. Smith, and Ali Vahidsafa. 2016. SPARC M7: A 20 nm 32-Core 64 MB L3 Cache Processor. *IEEE Journal of Solid-State Circuits* 51, 1 (2016), 79–91. doi:10.1109/JSSC.2015.2456902

[30] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 81–92. doi:10.1145/3620678.3624648

[31] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/3297858.3304053

[32] Mark LaPedus. 2019. *DRAM Scaling Challenges Grow*. Semicondutor Engineering. https://semiengineering.com/dram-scaling-challenges-grow/

[33] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. 2023. Elastic Use of Far Memory for In-Memory Database Management Systems. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 35–43. doi:10.1145/3592980.3595311

[34] Philip Levis, Kun Lin, and Amy Tai. 2023. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 18–24. doi:10.1145/3626111.3628195

[35] 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, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[36] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. 2021. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 33–48. doi:10.1145/3447786.3456227

[37] Changyuan Lin and Mohammad Shahrad. 2024. Bridging the Sustainability Gap in Serverless through Observability and Carbon-Aware Pricing. In *Proceedings of the 3nd Workshop on Sustainable Computer Systems* (Santa Cruz, USA) *(HotCarbon '24)*. Association for Computing Machinery, New York, NY, USA. https://hotcarbon.org/assets/2024/pdf/hotcarbon24-final164.pdf

[38] Linux Kernel Documentation. 2024. *Memory Hot(Un)Plug (6.10.0-rc6).* https://docs.kernel.org/admin-guide/mm/memory-hotplug.html

[39] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. 2012. Scale-out processors. *SIGARCH Comput. Archit. News* 40, 3 (jun 2012), 500–511. doi:10.1145/2366231.2337217

[40] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data).* 2884–2892. doi:10.1109/BigData.2017.8258257

[41] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. 2023. Myths and Misconceptions Around Reducing Carbon Embedded in Cloud Platforms. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems* (Boston, MA, USA) *(HotCarbon '23).* Association for Computing Machinery, New York, NY, USA, Article 7, 7 pages. doi:10.1145/3604930.3605717

[42] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Lanuages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023).* Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[43] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19).* Association for Computing Machinery, New York, NY, USA, 1981–1984. doi:10.1145/3299869.3320212

[44] John C Reynolds. 1993. The discoveries of continuations. *Lisp and symbolic computation* 6 (1993), 233–247. doi:10.1007/BF01019459

[45] Daniel Robinson. 2020. Optane Persistent Memory vs Optane SSDs - confused? Then read on. https://blocksandfiles.com/2020/10/08/optane-persistent-memory-vs-optane-ssds-confused-then-read-on/

[46] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23).* Association for Computing Machinery, New York, NY, USA, 498–514. doi:10.1145/3600006.3613156

[47] Margo Seltzer, Virendra Marathe, and Steve Byan. 2018. An NVM Carol: Visions of NVM Past, Present, and Future. In *2018 IEEE 34th International Conference on Data Engineering (ICDE).* 15–23. doi:10.1109/ICDE.2018.00011

[48] Rathijit Sen and Karthik Ramachandra. 2018. Characterizing Resource Sensitivity of Database Workloads. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 657–669. doi:10.1109/HPCA.2018.00062

[49] Amna Shahab, Mingcan Zhu, Artemiy Margaritov, and Boris Grot. 2018. Farewell My Shared LLC! A Case for Private Die-Stacked DRAM Caches for Servers. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 559–572. doi:10.1109/MICRO.2018.00052

[50] Debendra Das Sharma and Ishwar Agarwal. 2022. *Compute Express Link 3.0 Standard.* Technical Report.

[51] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. 2023. μManycore: A Cloud-Native CPU for Tail at Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23).* Association for Computing Machinery, New York, NY, USA, Article 33, 15 pages. doi:10.1145/3579371.3589068

[52] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24).* Association for Computing Machinery, New York, NY, USA, 818–833. doi:10.1145/3627703.3650061

[53] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20).* Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. doi:10.1145/3342195.3387517

[54] Amanda Tomlinson and George Porter. 2023. Something Old, Something New: Extending the Life of CPUs in Datacenters. *SIGENERGY Energy Inform. Rev.* 3, 3 (Oct. 2023), 59–63. doi:10.1145/3630614.3630625

[55] Emmett Witchel. 2024. Challenges and Opportunities for Systems Using CXL Memory. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24).* Association for Computing Machinery, New York, NY, USA, 2. doi:10.1145/3620666.3655590

[56] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24. https://dl.acm.org/doi/pdf/10.1145/216585.216588

[57] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19).* Association for Computing Machinery, New York, NY, USA, 331–345. doi:10.1145/3297858.3304024

[58] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 1029–1046. https://www.usenix.org/conference/osdi20/presentation/zhang-wen