Pointers IN FAR MEMORY

A RETHINK OF HOW DATA AND COMPUTATIONS SHOULD BE ORGANIZED

ETHAN MILLER,
GEORGE NEVILLE-NEIL,
ACHILLES BENETOPOULOS,
PANKAJ MEHRA,
AND DANIEL BITTMAN

It is the best of times and it is the worst of times in the world of datacenter memory technology. According to IDC (International Data Corporation), DRAM (dynamic random-access memory) revenues exceeded $100 billion in 2022. Yet, the anticipated growth rate is hugging the zero line, and many producers either reported loss-making quarters or are rumored to do so soon. From the perspective of datacenter customers, by some estimates, the cost of renting memory ranges from $20 to $30 per gigabyte per year, for a resource that costs only $2 to $4 to procure outright. On top of this, SaaS (software as a service) end users, for example, are forced to rent all the memory that they will need up front. By some rough estimates, they then end up using less than 25 percent of that memory more than 75 percent of the time.[10]

CXL (Compute Express Link), a new technology emerging from the hardware side,[9] is promising to provide *far* memory. Thus, there will be more memory capacity and perhaps even more bandwidth, but at the expense of greater latency. Optimization will first, seek to keep memory in far tiers colder, and, second, minimize the rates of both access into and promotion out of these tiers.[1,5] Third, proactive promotion and demotion techniques being developed for far memory promote/demote whole objects instead of one cache line at a time to take advantage of bulk caching and eviction in order to avoid repeatedly incurring its long latency. Finally, offloading computations with many dependent accesses to a near-memory processor is already being seen as a way to keep the latency of memory out of the denominator of application throughput.[11] With far memory, this will be a required optimization.

CHASING POINTERS "NEAR" MEMORY

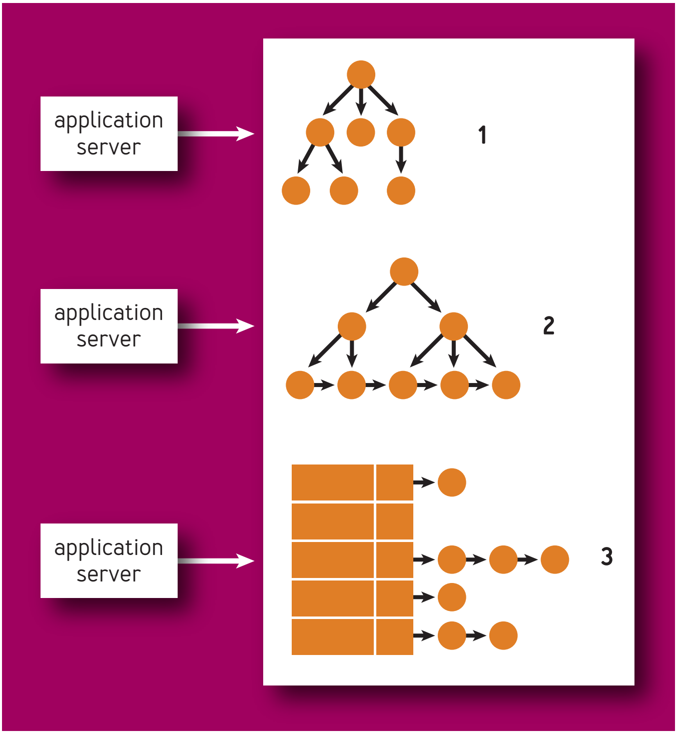
Applications that operate over richly connected data in memory engage heavily in pointer-chasing operations either directly (e.g., graph processing in deep-learning recommendation models) or indirectly (e.g., B+ tree index management in databases). Figure 1 shows an example of pointer-chasing applications in far memory: [1] graph traversal, [2] key lookup in a B+ index, and [3] collision resolution under open hashing.

Data from previous work[2] suggests that as data structures scale beyond the memory limits of a single host, causing application data to spill into far memory, programmers are forced to make complex decisions about function and data placement, intercommunication, and orchestration.

Performance characteristics of far memory

By default, pointers (like the internode ones in figure 1) are defined in the virtual address space of the process that created them. Because of this, if left unoptimized, pointer-chasing operations and their dependent accesses can

FIGURE 1: **EXAMPLE OF POINTER CHASING APPLICATIONS IN FAR MEMORY**



overwhelm the microarchitecture resources that support memory-level parallelism (e.g., reorder buffers) even on a single CPU with local memory. With latencies that can range from 150ns to more than 300ns,[2] far memory further compounds this problem.

In a distributed setting, a simple-minded pointer-chasing offload without taking care of virtual-to-physical address translation results in chatty internode coordination with the parent process.[15] Effective optimization of pointer-chasing operations entails minimizing communication between the near-memory processor executing the traversal and the server running the parent process.

Developing far memory-ready applications

Evidence from HPC (high-performance computing) and database workloads points to the extreme inefficiency of pointer-rich sparse memory operations on CPUs and GPUs alike,[4,14] in some cases hitting less than one percent of peak performance. This leads applications to want to offload such work to near-memory processors. In the case of far memory, that near-memory processor is itself outside the translation context of the parent process of the pointer-rich data. Pointers therefore must make sense everywhere in these new heterogeneous disaggregated systems.

In order to lower infrastructure rent, cloud applications also wish to exploit disaggregated far memory as a fungible memory resource that can grow and shrink with the amount of data. Moreover, they want to independently scale their memory and compute resources. For example, database services want to flex compute up or down in proportion to query load. Pointer-rich data in far memory must be shareable at low overhead between existing and new compute instances.

PRIOR WORK ON FAR MEMORY

Pointers in traditional operating systems were valid only in the memory space of the parent process. Sharing pointer-rich data among processes, nodes, and devices therefore required serialization-deserialization. This limitation remained even when prior art was recently extended by taking an approach of tombstoning dangling references to data demoted to far memory using special pointers.[7,16] Those pointers could be dereferenced only from the original context of data creation, precluding independent scaling of memory and computation.

Global address spaces such as PGAS (partitioned global address space) support a limited form of global pointers that persist only for the lifetime of a set of processes across multiple nodes. NVM (nonvolatile memory) libraries such as PMDK (Persistent Memory Development Kit) support object-based pointers, but their "large" storage-format pointers are more than 64 bits long, and their traversal cannot be offloaded.

Commercial virtualization frameworks such as VMware's Nu proclets[13] can maintain only the illusion of global pointers by compromising security (by turning address space layout randomization off, for example).

Microsoft CompuCache[14] also supported global pointers, but by using a heavy database runtime atop full VMs even on disaggregated memory devices. All pointers, whether at hosts or in the CompuCache, are VM-local only. Pointer chasing across devices requires repeatedly returning to the host.

Teleport[15] supported pointer-chasing offload to remote memory but by directed, on-demand shipping of the virtual-to-physical translation context to the target locale of each function shipped.

Prior work on OS constructs for far memory is therefore missing a foundation of globally invariant pointers that can be shared with and dereferenced by any node or device in a cluster containing far memory.

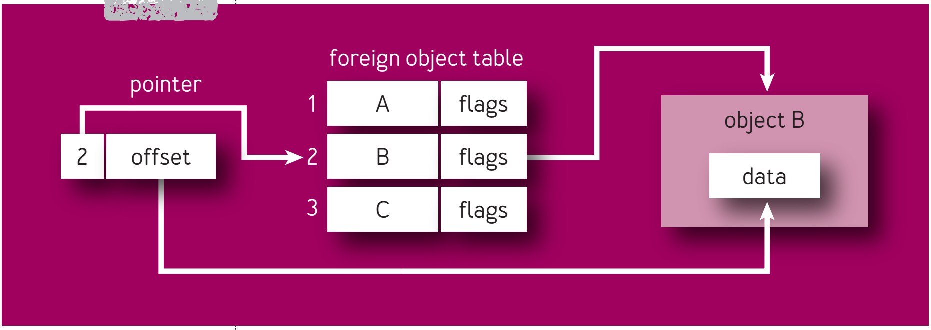
INVARIANT POINTERS

When organizing data at object grain, a globally invariant pointer must contain the ID of the object containing the target data, as well as an offset to that data. This object ID can be interpreted anywhere the pointer can be dereferenced. Ideally, invariant pointers should: (1) be no larger than 64 bits; and (2) permit access to partially resident objects. Existing approaches do not meet the first criterion (e.g., PMDK) or the second criterion (e.g., AIFM[12]); AIFM (application-integrated far memory) has a different pointer form for resident and nonresident objects. Providing truly globally invariant pointers, however, is necessary for offloading "run anywhere" code.

Twizzler[3] is an operating system that introduces globally invariant pointers by using a context local to the object in which the pointer is stored, shortening its representation while allowing any CPU that can read the pointer to fully resolve its destination. This is done using an FOT (foreign object table) that is part of each object in the system, ensuring that any individual object is self-contained.

An object's FOT contains identifying information for each foreign object that is the destination for a pointer in the object. Since these are stored in an ordered table, stored pointers use the index into the FOT as a stand-in for the full addressing information, a translation process shown in figure 2. This approach allows pointers to remain small: a 64-bit pointer can, for example, include a 24-bit "local" object ID and 40-bit offset. While this limits the number of foreign objects that can be referenced from a single object to 2^{24}, different objects have their own FOTs and can reference a different set of objects, so the total

FIGURE 2: **STORED POINTERS USE THE INDEX INTO THE FOT AS PLACEHOLDER**



number of objects in the system is limited only by the size of an object ID.

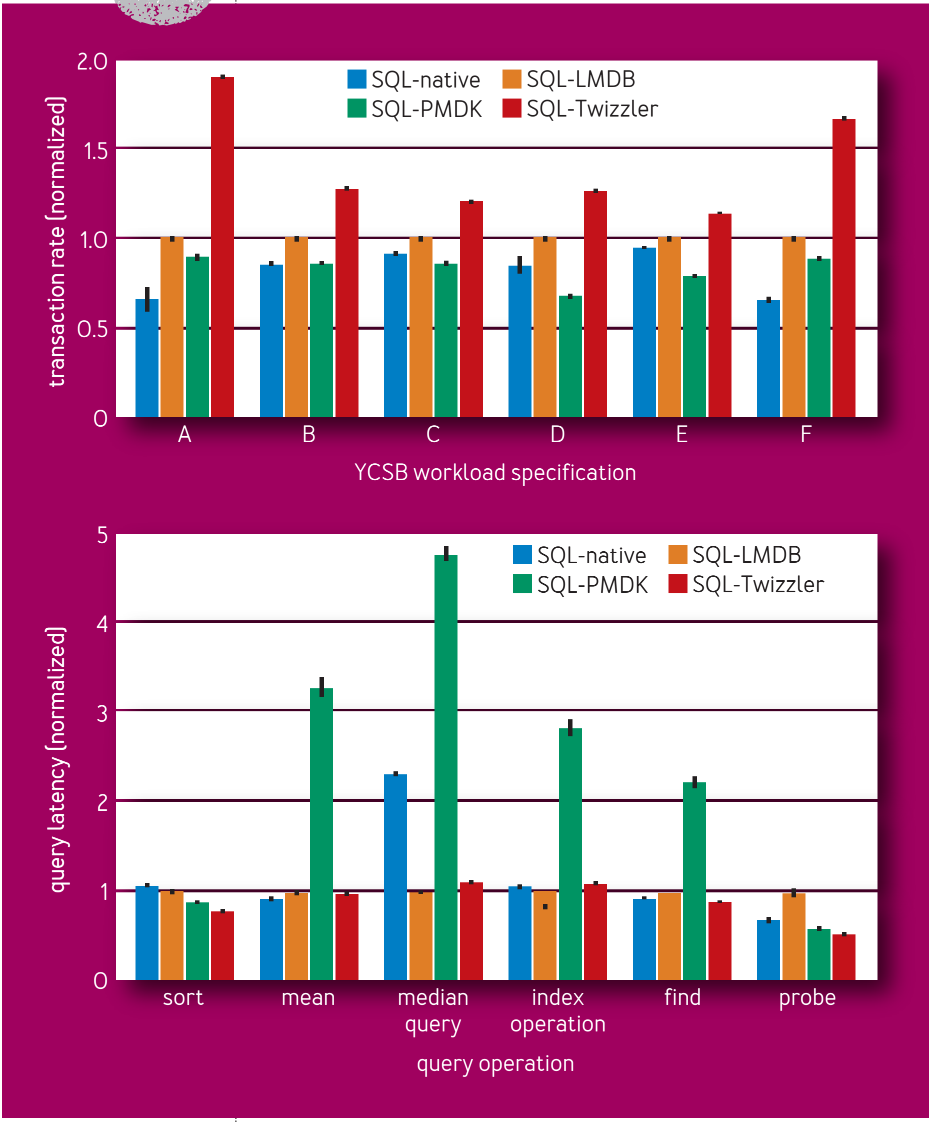
This approach also allows for a wide range of resolvers that translate identifying information in the FOT into an object ID. For example, the FOT might contain a static object ID or the equivalent of a file-system name to be resolved to an object ID by a name resolver. There is no requirement that a name resolve to the same object ID in different places: An object named */var/log/*syslog might resolve to different object IDs on different system nodes. Name resolvers themselves can be pluggable: The FOT needs only to identify the resolver in a way that any node in the system can run the resolver to return an object ID.

While the first access to a foreign object may be relatively slow, subsequent accesses are very fast, since the resolution to an object ID is cached. The system maps the object into the node's "guest physical" address space, leveraging MMU (memory management unit) hardware already in use for virtualization. It then maps the guest physical space in which the object resides into the guest virtual space for any processes that reference the object, using extended page tables to remove software from the CPU load/store path and allowing the system to run at memory speed. This is necessary for efficiency: Even minimal system software interaction on each load and store will slow the computation significantly.

Preliminary experiments[3] show that Twizzler's approach is effective at preserving low-latency pointer dereferencing for both intra-object and inter-object invariant pointers. On an Intel Xeon Gold CPU running at 2.3GHz, intra-object pointer dereferences take about 0.4ns, approximately the same time as "normal" dereferences. Cached inter-object pointer dereferences take 3.2ns, somewhat slower than intra-object dereferences but still sufficiently fast because relatively few such references are expected, given multi-megabyte objects. The first reference to a foreign object is slower, at 28ns, but still reasonable. If name resolution is more complex than interpreting a static full-length (128-bit) object ID, it would be longer still; however, these penalties are paid only once, regardless of how many times pointers from object A to object B are dereferenced in the same process.

Benchmarks on both microscale (in-memory key/value store) and macroscale (YCSB [Yahoo! Cloud Serving Benchmark] using different back ends) likewise show excellent performance for this approach. The top graph in figure 3 shows performance of the YCSB benchmark on SQLite using four back ends: the native SQLite back end; the LMDB (Lightning Memory-mapped Database) back end, which leverages mmap; our implementation of a PMDK

FIGURE 3: **PERFORMANCE AND LATENCY OF THE YCSB BENCHMARK**



back end, which uses a red-black tree under PMDK; and Twizzler, which uses a red-black tree with the invariant pointer approach.

The invariant pointer approach outperforms every other approach while providing the flexibility of "run anywhere" invariant pointers. The graph on the bottom of figure 3 similarly shows that these invariant pointers provide lower latency than other approaches because of the simplicity of the programming model and the low overhead for dereferencing pointers. PMDK, in particular, is significantly slower because its pointers are 128 bits long, requiring additional register space and memory operations to read and dereference.

It is important to note that the PMDK and Twizzler implementations are running the same back-end code, with changes made only to accommodate the different programming models; this shows the benefit of using 64-bit pointers local to an object context rather than 128-bit pointers, as PMDK does.

Elephance MemOS is a fork of Twizzler being developed to run on CXL far memory devices. It will be ported and optimized for the SoCs (systems-on-chip) used as controllers in CXL-disaggregated memory nodes.

PROGRAMMING WITH MEMORY OBJECTS AND INVARIANT POINTERS

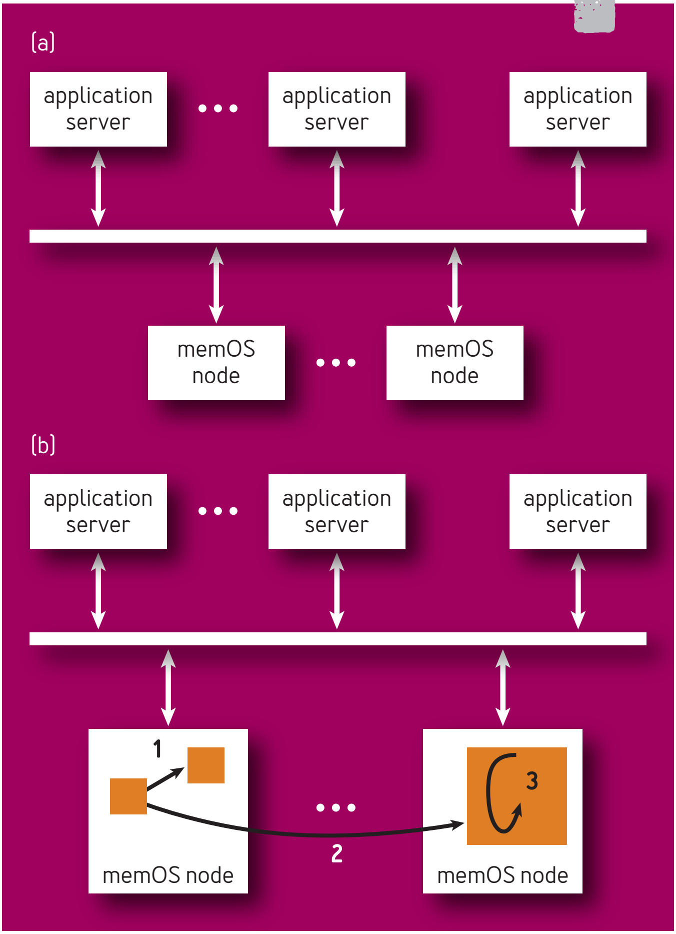
For software developers, what does memory disaggregation mean and how will systems be built around it? The architecture of such systems will aim to hide the details from the majority of programmers, so their code will not need to change to run on these new systems.

There are three ways in which systems can be built to provide disaggregated memory: application libraries, modification to the operating system's memory system, and changes beneath the operating system at the hardware layers, as seen in figure 4. In the figure, a set of application servers is connected to a set of MemOS nodes over a shared bus. Pointer-rich application data in far memory lives on MemOS nodes. Pointers can be: (1) inter-object and on the same device, (2) inter-object across devices, or (3) intra-object.

It is likely that the first way that disaggregated memory will be made available will be through application libraries linked directly into the application, seen in (A) at the top of figure 4. The memory shim acts as a specialized memory allocator that knows how to handle remote memory using a MAP (memory access protocol). The MAP may depend on a current technology such as RDMA (remote direct memory access), or may be something newer such as CXL3.

Many languages, such as Python, which depend on the C library for memory, will be able to use the memory shim to handle memory for objects in the language, freeing the Python programmer from having to know anything about disaggregated memory. For languages such as C and C++, which handle pointers directly, the programmer will have to work with the memory shim APIs in order to manage remote memory. The prevalence of Python and similar managed memory languages in big data and machine-learning applications means that programmers in those fields can use disaggregated memory in a transparent way, no matter where the memory shim is located in the software stack.

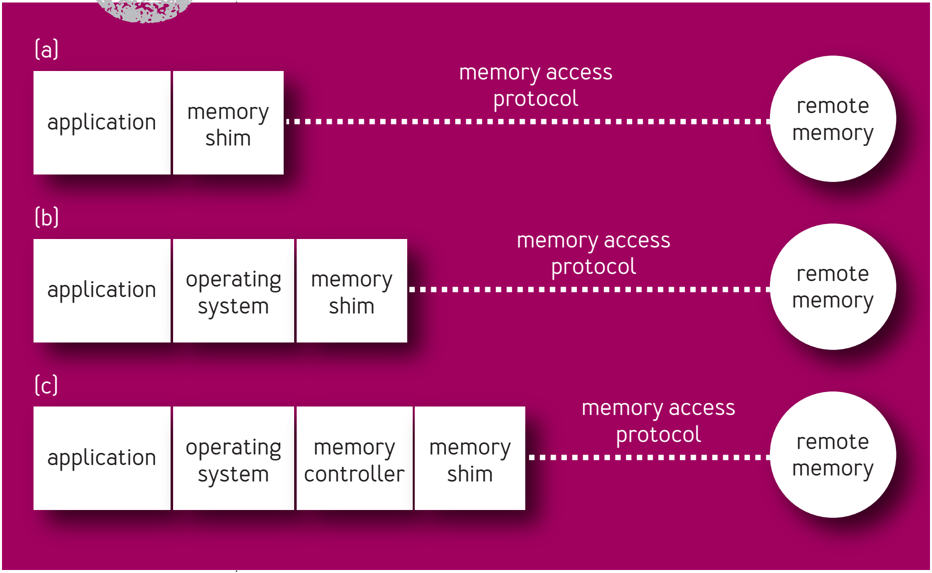
FIGURE 4: **EXAMPLE MEMOS DEPLOYMENT**



Extending the operating system's virtual memory system to integrate with the memory shim is the next logical place to interpose disaggregated memory in the stack, seen in (B) in figure 5. Again, the specific MAP is not exposed to the kernel developer, only the memory shim APIs. The Linux operating system already has HMM (heterogeneous memory management),[8] which is a natural place to slot in the memory shim. Once the shim is integrated into the operating system itself, all applications can use disaggregated memory transparently without modifications to their source code or linking with specialized libraries.

The deepest that far memory can be placed in the stack is in the hardware itself. Memory controllers integrated in CPUs from Intel and AMD are already starting to support early versions of CXL disaggregated memory. Future, more featureful controllers will present memory

FIGURE 5: **EXTENDING THE OS'S VIRTUAL MEMORY SYSTEM**



to the operating system both locally and remotely in a transparent manner, but, like the other two cases, will require a MAP to be interposed between the hardware and the remote memory. The protocol in this instance will be CXL 3.

While putting the memory shim into hardware will likely result in the highest bandwidth, lowest latency, and maximum portability, there are reasons to continue to use a memory shim as a linked library into the software. First and foremost is the level of control that linking directly to the memory shim gives to the programmer. Once such functionality is embedded into the operating system or the memory controller, application programmers will lose control and visibility into the remote memory system. While many will be happy not to have to manage memory on their own, applications will remain where such control is a feature. Novel memory architectures for distributed memory must first be tried in software, and some may be too specialized ever to be implemented in hardware.

Consider a memory system where pointers are globally invariant, which will be possible with MemOS but is not yet common in pointer-based systems. Building and debugging such a system in software makes it possible to rapidly iterate on the design—impossible in a memory controller and certainly more difficult to debug in the operating system. Applications that can use globally invariant pointers have distinct advantages because computation can take place on any node without the application having to know where a pointer might reside. Furthermore, it will be possible to move code, rather than data, to achieve computational efficiency—again, because no matter which compute node a pointer resides on, the pointer itself is the global handle that computation depends on, rather than an address in local memory, as things stand today.

KEY TAKEAWAYS

Effectively exploiting emerging far-memory technology requires consideration of operating on richly connected data outside the context of the parent process. Operating-system technology in development offers help by exposing abstractions such as memory objects and globally invariant pointers that can be traversed by devices and newly instantiated compute. Such ideas will allow applications running on future heterogeneous distributed systems with disaggregated memory nodes to exploit near-memory processing for higher performance and to independently scale their memory and compute resources for lower cost.

References

1. Al Maruf, H., et al. 2023. TPP: transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* 3, 742–755; https://dl.acm.org/doi/10.1145/3582016.3582063.
2. Berger, D. S., et al. 2023. Design tradeoffs in CXL-based memory pools for public cloud platforms. *IEEE Micro* 43(2), 30–38; https://dl.acm.org/doi/abs/10.1109/MM.2023.3241586.
3. Bittman, D., et al. 2020. Twizzler: a data-centric OS for non-volatile memory. In *Proceedings of the Usenix*

Annual Technical Conference; https://dl.acm.org/doi/pdf/10.5555/3489146.3489151.

4. Dongarra, J. 2021. A not so simple matter of software. ACM Turing Award lecture; https://www.youtube.com/watch?v=cSO0Tc2w5Dg.
5. Duraisamy, P., et al. 2023. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* 3, 727–741; https://dl.acm.org/doi/10.1145/3582016.3582031.
6. Hsieh, K., et al. 2016. Accelerating pointer chasing in 3D-stacked memory: challenges, mechanisms, evaluation. IEEE 34th International Conference on Computer Design (ICCD), 25–32; https://ieeexplore.ieee.org/document/7753257.
7. Jennings, S. 2013. The zswap compressed swap cache. LWN.net; https://lwn.net/Articles/537422/.
8. Kernel Development Community. Heterogeneous memory management. The Linux Kernel 5.0.0; https://www.kernel.org/doc/html/v5.0/vm/hmm.html.
9. Mehra, P., Coughlin, T. 2022. Taming memory with disaggregation. *Computer* 55(9), 94–98; https://ieeexplore.ieee.org/document/9869614.
10. Michelogiannakis, G., et al. 2022. A case for intra-rack resource disaggregation in HPC. *ACM Transactions on Architecture and Code Optimizations* 19(2), 1–26; https://dl.acm.org/doi/10.1145/3514245.
11. Rodrigues, A., Gokhale, M., Voskuilen, G. 2019. Towards a scatter-gather architecture: hardware and software issues. In *Proceedings of the International Symposium*

on Memory Systems, 261–271; https://dl.acm.org/doi/10.1145/3357526.3357571.

12. Ruan, Z. et al. 2020. AIFM: high-performance, application-integrated far memory. In *Proceedings of the 14th Usenix Symposium on Operating Systems Design and Implementation (OSDI)*; https://dl.acm.org/doi/pdf/10.5555/3488766.3488784.
13. Ruan, Z., et al. 2023. Nu: achieving microsecond-scale resource fungibility with logical processes. In *Proceedings of the 20th Usenix Symposium on Networked Systems Design and Implementation (NSDI);* https://www.usenix.org/system/files/nsdi23-ruan.pdf.
14. Zhang, Q., et al. 2022. CompuCache: remote computable caching using spot VMs. 12th Annual Conference on Innovative Data Systems Research (CIDR); https://www.cidrdb.org/cidr2022/papers/p31-zhang.pdf.
15. Zhang, Q., et al. 2022. Optimizing data-intensive systems in disaggregated data centers with Teleport. In *Proceedings of the International Conference on Management of Data (SIGMOD);* https://dl.acm.org/doi/10.1145/3514221.3517856.
16. Zhou, Y., et al. 2022. Carbink: fault-tolerant far memory. In *Proceedings of the 16th Usenix Symposium on Operating Systems Design and Implementation (OSDI);* https://www.usenix.org/system/files/osdi22-zhou-yang.pdf.

Ethan Miller *has been a member of the technical staff at Pure Storage since 2009. He is a professor emeritus in computer science and engineering at the University of California, Santa Cruz, where he held the Veritas Presidential Chair in Storage; founded the Center for Research in Storage Systems; and*

led the development of the Ceph distributed file system, the Twizzler operating system. He was a member of the RAID project during his PhD work at UC Berkeley. His research interests include non-volatile memory systems, security and reliability for storage systems, and scalable and long-term storage systems.

George Neville-Neil *works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His interests are computer security, operating systems, networking, time protocols, and the care and feeding of large code bases. He is the author of* The Kollected Kode Vicious *and a co-author of* The Design and Implementation of the FreeBSD Operating System. *He earned his bachelor's degree at Northeastern University, Boston. His software was deployed in NASA's missions to Mars, as part of VxWorks. He is an avid bicyclist and traveler who currently lives in New York City.*

Achilles Benetopoulos *is a PhD student at UC Santa Cruz, working at the intersection of distributed systems, databases, and programming languages. Previously, he spent a few years working as a software engineer up and down the stack for several companies.*

Pankaj Mehra, *founder of Elephance Memory, has held executive technology and management positions in memory and storage industries since 2013. He was previously VP of product planning at Samsung, a senior fellow at SanDisk and Western Digital, and a distinguished technologist at Hewlett-Packard. Pankaj has held faculty and visiting positions at IIT*

Delhi, UC Santa Cruz, and the IBM TJ Watson Research Center. He is an author/inventor with more than 100 books, papers, and patents about servers, storage, interconnects, and AI.

Daniel Bittman *is a co-founder of Elephance Memory and principal maintainer of the Twizzler operating system. He received his PhD in CS from UC Santa Cruz, studying under Peter Alvaro and Ethan Miller. When not doing operating systems research, he can be found hiking in the mountains.*

CONTENTS