Declarative Memory Services

Jeronimo Castrillon
Technische Universität Dresden
jeronimo.castrillon@tu-dresden.de

Jana Giceva
Technische Universität München
jana.giceva@in.tum.de

Yu Hua
Huazhong University of Science and Technology
csyhua@hust.edu.cn

Kimberly Keeton
Google
kimkeeton@google.com

Akhil Shekar
University of Virginia
as8hu@virginia.edu

Kevin Skadron
University of Virginia
skadron@virginia.edu

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

The advent of new memory devices (e.g., high-bandwidth memory and processing-in-memory devices) and interconnects (e.g., Compute Express Link) brings a diverse landscape of modern memory systems with high performance and non-traditional features such as compression, encryption, replication and domain-specific as well as general-purpose computation. Meanwhile, it has become highly complex to program modern memory systems, leading to hand-crafted systems that are not sustainable as the hardware evolves.

Our vision is to build *declarative memory services* (DMS) where developers only need to specify the desirable properties supported by the devices without having to dictate how these properties are implemented using which device, which is performed by a DMS runtime, so as to hide the complexity and allow ample space for optimizations. We demonstrate the potential of DMS via real use cases and outline a research agenda towards realizing this vision.

1 INTRODUCTION

Data-intensive systems and applications must use memory efficiently to achieve high performance. For decades, the term "memory" has been synonymous with the properties of single-node DRAM: it is volatile, byte-addressable, orders of magnitude faster than storage, and its access is coherent only at the node level. Programming such conventional memory is tractable with various established building blocks, such as fast memory allocators [10, 18, 34], cache-conscious algorithms [41, 48, 56], and primitives such as `numactl` and `madvise` [19].

However, the development of memory devices has outpaced that of their programming models. For example, next-generation interconnects such as the Compute Express Link (CXL) [7] allow memory devices with diverse properties to coexist in a disaggregated system, often without forming a strict hierarchy [17, 67]. This trend is redefining "memory": it can be volatile or persistent; it may contain its own compute elements; its access is not always most efficient in a byte-addressable manner; and coherence guarantees can now extend across processors for certain (but not all!) memory regions. Overall, emerging memory technologies exhibit fundamentally different performance characteristics (e.g., latency, bandwidth, and access granularity) and capabilities (e.g., persistence, coherency, ordering guarantees, and even integrated compute).

These advanced properties break long-standing assumptions about address spaces, pointer validity, data placement, coherence, update semantics, and fault handling. For example, computation within the memory hierarchy must be kept coherent with CPU caches to prevent computing on stale data. This new landscape requires decoupling data and memory from the traditional, process-centric abstraction of a machine.

As a result, DBMS developers today must navigate a vast design space to leverage the unique properties of each memory device and make decisions regarding data placement and movement, coherence, synchronization, and fault handling. This complexity results in low programming efficiency, high maintenance and portability costs, and poor adaptivity that negatively impact performance. For example, an index specifically built and tuned for one generation of memory hardware may perform poorly on a newer generation.

Existing programming models force developers to work with low-level, imperative abstractions, such as `load`/`store`, and atomic instructions (or those built on top of them, such as latches). Developers must specify the implementation details (i.e., the "how") by understanding the properties of each device and applying them in hand-crafted code. In this paper, we argue that the programming model for future memory technologies should instead be declarative, allowing developers to specify *what* properties and features to use. To this end, we propose the vision of *declarative memory services* (DMS), following a layered approach. (1) An abstraction layer allows developers to declare the desired properties. (2) A calibration layer allows DMS to ascertain the underlying hardware capabilities. (3) A memory services layer then orchestrates program execution by identifying the optimal way to use each memory device.

With DMS, developers only need to specify high-level properties, such as parallelism and coherence requirements, at the memory block level and use a dataflow model to perform complex tasks. They no longer have to dictate how these operations and guarantees are realized. Instead, the DMS runtime transparently manages the implementation using various low-level mechanisms, such as offloading to processing-in-memory (PIM) devices and leveraging hybrid software-hardware support for coherence on partially coherent interconnects. As we will demonstrate later with practical use cases, DMS has the potential to free system developers from complex and manual decision-making, and presents many opportunities for performance optimizations.

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

This paper makes four contributions. ❶ We survey the landscape of emerging disruptive memory technologies to highlight the fundamental changes and complexities they introduce. ❷ We envision that new programming models for future memory technologies should evolve to be more declarative to ease development and maintenance. ❸ We propose the vision of declarative memory services (DMS) and demonstrate its potential with practical use cases. ❹ We elaborate on the challenges and present a research agenda to guide future work towards the full realization of DMS.

2 DISRUPTIVE MEMORY TECHNOLOGIES

Memory in modern computing systems is available in diverse configurations to offer properties as listed in Table 1.

Passive Memory. Traditionally, memory is a passive component that serves as an intermediary storage layer between CPU caches (SRAM) and durable storage (e.g., SSDs) without inherent computation capabilities. Standard DDR DRAM is typically programmed using synchronous instructions such as `load`/`store`, atomics, and SIMD. Features like I/OAT [20] enable asynchronous memory operations to reduce data movement costs. Programming memory on a modern machine has become increasingly intractable. This complexity stems not only from well-understood NUMA effects in multi-socket CPUs but also from the black-box nature of CPU caches and newly observed chiplet effects [11]. For example, Data Direct I/O (DDIO) [21] delivers data from PCIe devices (e.g., NICs and NVMe SSDs) directly to a subset of CPU cachelines, requiring software to consume the data at the right time before the cachelines are evicted to realize any performance benefit. Similarly, applications seeking to partition the CPU cache must rely on vendor-specific intrinsics [44] or OS-level modifications [39]. Such idiosyncrasies of the modern memory devices and hierarchy make it challenging for applications and systems to achieve optimal performance.

Computational Memory. To overcome the data movement bottlenecks of modern workloads, the concept of computational memory has emerged, which integrates processing capabilities closer to or within the memory system [24]. This allows operations such as (de-)/encryption, compression, quantization, and even specialized analytics tasks such as filtering and similarity search to occur close to the data. This can boost performance and save energy. Two primary approaches exist: processing-near-memory (PNM) and processing-in-memory (PIM). PNM embeds logic outside the DRAM chips, but closely coupled with the memory module [23]. PIM embeds general-purpose or domain-specific computation logic in the DRAM, at either the bank-level [8, 16, 32] or even integrated with the individual subarrays making up the bank, using analog, e.g. [55] or digital logic, e.g. [36, 58, 68]. Although PNM reduces data movement overhead, placing the logic outside the DRAM dies misses the opportunity to leverage the high internal bandwidth of the DRAM architecture, e.g., all-bank parallelism. In contrast, PIM integrates the computation logic directly within the DRAM die. Emerging memory technologies such as ReRAM and PCM provide additional primitives such as massively parallel analog dot products [54]. However, PIM requires specialized and likely more costly memory designs with lower capacity. It may also increase contention for concurrent accesses to the memory interface. Both PNM and PIM potentially create challenges regarding coherence with the processor's cache hierarchy, and may even *increase* data movement if data first needs to be explicitly flushed from the cache to enable PIM-offloading, before moving it back to the CPU.

Table 1: Taxonomy of memory properties.

Property	Examples
Performance	Latency < 20 μs, bandwidth > 2GB/s
Volatility	Non-volatile, volatile
Active/Passive	Active (PIM) vs. normal DRAM
Location	Local, NUMA, remote (CXL or RDMA)
Granularity	Byte-addressable vs. page-based (and page size)
Coherency	Coherent, not coherent, partially coherent
Security	Encrypted vs. not encrypted
Compression	Types of compression
Ordering	Strong/weak memory model, explicit fencing, causal consistency
Fault Semantic	Atomicity, exactly once
Fault Model	Transient, partial fault

Tiered Memory Systems. Tiered memory is an important building block of modern data centers. Driven by escalating memory costs and the prevalence of latency-sensitive memory-intensive workloads, hyperscalers are rapidly integrating cheaper, slower memory tiers into the server memory hierarchy. These systems aim to free up fast, expensive primary memory by identifying infrequently accessed (cold) pages and relocating them to slower, cheaper secondary memory tiers (e.g., compressed memory [31], SSDs [66], non-volatile memory (NVM) [9] or CXL-attached memory [37]), while staying within a performance degradation target. Considerable attention has been paid to production tiered memory systems, new memory technologies, novel mechanisms to measure page temperature, and transparent operating system policies to migrate data to the appropriate tier in a timely fashion [2, 3, 6, 9, 13, 14, 31, 33, 37, 42, 49–51, 63, 64, 66, 69, 70, 72].

Resource Disaggregation. High-speed interconnects such as InfiniBand and CXL [7] have made it feasible to build diverse and flexible disaggregated memory pools from passive memory, computational memory and fast SSDs. A common deployment in today's data centers is remote direct memory access (RDMA), which minimizes CPU intervention for remote data accesses. In particular, InfiniBand provides ultra-low RDMA latency compared to Ethernet. CXL represents a further evolution, promising to fully decouple DRAM from the CPU. With a protocol built on top of PCIe, CXL enables low-latency, coherent communication between host CPUs and peripheral devices (e.g., accelerators). This facilitates flexible memory sharing, including fully disaggregated pools of heterogeneous memory types, and allows for intelligence in the memory controller to be used for offloading application logic [27]. Additionally, advances in NAND flash, such as 3D stacking, along with NVMe standards, have produced SSDs with bandwidth and latency approaching those of DRAM [17]. This progress has made it possible to provide byte-addressable memory devices backed by flash memory available in the form of memory expander cards over CXL, further reducing costs and increasing system memory capacity.

A Programmability Challenge. A significant programmability challenge arises because many of these new memory technologies deviate from the traditional von Neumann architecture. This necessitates the programming models and system software that support them. For example, the lack of ordering guarantees on RDMA writes renders it impractical for synchronization purposes [74]. While CXL-based solutions can overcome this limitation [71], they introduce new failure modes such as partial faults (e.g., what happens to a pointer and its object's consistency if a link fails), which the corresponding programming model must capture and handle. Although early prototypes have shown feasibility through hand-crafted optimizations, generalizing them requires automation, compiler support, appropriate abstractions, and robust runtime systems [25].

3 CASE STUDIES

To highlight the challenges and motivate the need for DMS, we examine three representative scenarios: (1) building disaggregated B+-trees, (2) leveraging PIM for filtering in data-parallel data analytics pipelines, and (3) improving buffer cache efficiency using tiered memory.

3.1 Case 1: Manually Disaggregated B+-Trees

Disaggregated memory enables independent scaling of compute and memory resources, offering high resource utilization and virtually unlimited, elastic memory capacity for database systems. Consequently, DBMS architectures must be restructured to effectively leverage separate compute and memory pools. We use RDMA-based disaggregated memory systems here as a concrete example. The memory pool consists of a number of servers with large memory and weaker CPUs, whereas the compute pool features servers with larger-scale multicore CPUs (e.g., 10s to 100s of cores per server) but limited memory. Tree nodes reside in the memory pool and can be accessed using one-sided or two-sided RDMA from compute servers. This creates a large design space.

Coherence and Caching. Due to the long latency to access the memory pool, it is desirable to cache frequently accessed tree nodes in compute servers. Yet the application may access and update the tree from any compute server and coherence is not guaranteed across servers by RDMA, so the developer must consider coherence between compute servers. A naive implementation would incur many coherence messages between compute servers, adding much overhead [40]. The programmer also has to decide on the caching policies by answering such questions as *should the compute side cache both inner and leaf nodes, or only certain types?, what kind of eviction policy should be employed?* and so on.

Data Placement and Replication. On the memory side, a key design decision is data placement strategies. This again opens up a vast tradeoff space:*how should the tree nodes be partitioned across memory servers?, whether/when to use replication?*, etc.

Offloading. Just like the compute side features some memory, the memory pool also features some compute that could be leveraged. This leads to the non-trivial design decision of when, what, and how to offload tree operations to the memory side to reduce data transfer costs between the compute and memory pools without overloading the memory servers.

Existing programming models provide only low-level primitives like RDMA read/write. Without lifting the abstraction level, disaggregated B+-trees had to hardcode their designs. For example, Sherman [65] hardcoded its caching strategy to cache only inner nodes. DEX [40] took a step further to cache leaf nodes, and used compute-side logical partitioning to sidestep coherence. It also had to hardcode data partitioning and offloading strategies.

3.2 Case 2: Accelerating Filtering with PIM

Typical data analytics pipelines consist of various tasks such as join, aggregation and filtering. PIM is ideally suited for such data-parallel tasks because it can exploit the massive parallelism inherent in DRAM at a much higher bandwidth than what is exposed on the memory channel. Filtering predicates, for example, are highly data-parallel and can greatly benefit from PIM-based compute offloading, for example, at a bank-level as proposed by Membrane [57]. However, for a data analytics pipeline to leverage PIM for filtering, the developer must explore a large design space due to several architectural challenges.

Data Placement. It is desirable to maximize bank-level parallelism during table filtering. This requires the developer to evenly distribute table data across all DRAM banks. However, existing virtual memory page allocation policies offer limited control over physical address mapping, making such a distribution difficult to guarantee. Hardware solutions may limit system flexibility and scalability. OS solutions may be difficult to extend when new hardware designs become available. Ideally, a dynamic data placement strategy tailored for PIM is needed, that partitions memory into PIM and non-PIM regions and enables fine-grained control over physical address ranges.

Coherence. Given that DRAM can now be accessed by both offloaded, on-chip operations and the CPU, a developer would need to carefully reason about the consistency between CPU caches and main memory. For example, they have to make decisions such as *when should a cacheline be flushed to reflect a new update?, which flushes can be potentially avoided to improve performance?* and so on. Such decisions affect both performance and correctness.

Offloading. The compute tasks and associated data flow in a query plan, and potential parallel execution require careful coordination and optimization to avoid introducing wasteful data movement and costly synchronization. Offloading decisions must holistically optimize data movement and compute efficiency across the entire dataflow.

3.3 Case 3: Memory Tiering in Buffer Cache

The three-tier buffer cache ideas presented in HyMem [59] and Spitfire [73] show that one can implement more efficient and flexible buffer managers for tiered memory/storage hierarchies (DRAM, persistent memory, SSDs) with the use of pointer swizzling. Vmcache [35] introduced a less-invasive buffer manager to translate page identifiers into pointers by leveraging the page tables used by virtual memory. Unlike file-backed mmap, this allows the DBMS to retain control over page faults and the eviction policies. In either case, the DBMS can place objects in pages with variable sizes and in vmcache it can give hints about which pages are no longer needed and can thus be evicted with no penalties using primitives like

`madvise(MADV_DONTNEED)`. The vmcache architecture was later also extended to integrate CXL-memory extensions in the memory/storage hierarchy [52].

Data Caching. However, data that needs to be read or spilled can have different properties. For example, data can be read just once and then discarded (e.g., sequential read with a scan) or can be accessed repeatedly and at random (e.g., when building or probing a hash table). The former data pages should not be placed in the buffer cache at all, thereby avoiding cache pollution, whereas the latter needs to remain present for as long as the hash table is used. These insights cannot be easily communicated to the buffer manager with existing interfaces.

Compression and Encoding. Pages that hold base data are often stored in a columnar format following the decomposition storage model (DSM) or partition attribute across (PAX) [1], while those holding intermediate data [28] are stored in a row-major format following the N-ary storage model (NSM). Cooling data pages in a remote memory pool or in storage often involves additional encoding/compression that depends on the data format/types and the capabilities of the target hardware platform [30]. Again, with a lack of transparency, the buffer manager would have to resort to a more generic compression scheme.

Synchronization and Coherency. Some DBMS data structures are used for fine-grained coordination among worker threads, while others are thread-local. Different scopes of coordination need different properties from the underlying memory. For example, in the presence of CXL-based memory pools, one would prefer to keep the data structures used for synchronization in the coherent part of the memory, and those that still need to be shared but without coordination in the non-coherent part of the pool [4]. Thread-local data structures should remain close to the worker that exclusively uses them.

3.4 The Case for Declarative Memory Services

Through the experience of manually disaggregating B+-trees, leveraging bank-level filter units, and using tiered memory for the buffer cache, we observe several major drawbacks of existing programming models: (1) The resulting design and implementation can be highly complex, requiring non-trivial concurrent and distributed programming skills. (2) The design decisions are fixed once the system is implemented and only provide optimal performance for the assumed devices, limiting portability and leaving little room for runtime adjustment. For example, it can be challenging to port DEX or a data analytics pipeline built for Membrane to to different hardware, because even slight changes in features might render the existing optimizations inefficient. Furthermore, changes to application-level logic may interfere with the memory system design and optimizations. (3) By design, manual approaches only provide localized optimizations that benefit a single data structure or application. For example, the effectiveness of offloading a task to PIM heavily depends on three factors: the end-to-end task graph and where these tasks can be efficiency executed; the system's memory configuration; and the size, location, and potential data-movement costs of the data structures being operated upon. Although the first item is often known during development/compilation, the last two typically are not. Hardcoded decisions will often not result in optimal performance, so delaying the decision to runtime can ensure more efficient and scalable performance by exploiting near-data processing when appropriate. In both case studies, offloading also does not consider other DBMS components. (4) Finally, handcrafted systems may miss important features, making them partial solutions. For example, little work has considered availability and replication while designing indexes for disaggregated memory.

Given these complexities, we argue that now is a good time to embrace a declarative model, which we refer to as DMS, for *declarative memory services*. With DMS, developers specify the desired properties, which are then captured by a compiler and runtime that directly orchestrates the execution using available hardware. This enables separation of concerns related to the execution runtime, and *decouples the physical design of an application and its logical functionality*. The benefit is multi-fold:

- Better optimization across heterogeneous memory devices.
- Decoupling device-specific logic from the high-level code.
- Simplified programming for future, unknown architectures.

For example, the developer can start with a monolithic B+-tree and annotate the source code with desired properties (such as those listed in Table 1). Tree nodes can be marked as `cacheable` and/or `coherent`, and B+-tree operations functions can be annotated with `offloadable` when active memory devices (e.g., PIM) are available. Similarly for data analytics, with DMS, developers can describe optimal runtime offloading strategies based on specific data characteristics, such as table sizes or cardinality, minimizing the need for low-level, hardware-specific programmer intervention. Finally, a buffer cache built with DMS can delegate much work previously done manually to DMS at runtime. For example, the developer can declare which data structures require coherence and specify the desirable encoding scheme for a certain data type. DMS then employs the corresponding memory services to satisfy these requirements.

Based on the available hardware resources, the DMS runtime interprets the annotations to guide decisions related to data placement, replication, coherence, caching, offloading, etc. This enables high adaptability to evolving hardware, ensuring systems built with DMS remain robust and future-ready. Next, we discuss the DMS vision in detail and outline a research agenda to fully realize it.

4 VISION: DECLARATIVE MEMORY SERVICES

We propose a layered approach to realizing DMS. As Figure 1 shows, the first layer is an abstraction layer that allows applications to specify their desirable properties (e.g., those listed in Table 1), including whether certain memory regions should be persistent, encrypted, coherent, etc. The second layer is the calibration layer shown in the bottom of Figure 1, which discovers and keeps track of the properties of the available hardware devices. Between these two layers is the memory services layer which is "the runtime" that orchestrates the execution of the applications.

4.1 Declarative Abstraction Layer

We adopt logical region-based abstraction [4] and dataflows to provide declarative programming for an application that can be a low-level data structure (e.g., a B+-tree) or a full end-to-end system (e.g., a DBMS or machine learning framework). Developers can choose to work with either or both of them: the former provides

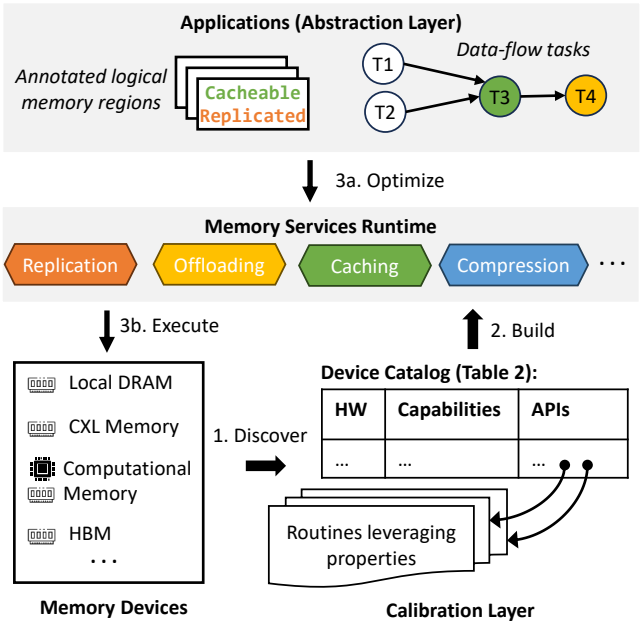

Figure 1: Overview of DMS. The calibration layer (1) discovers hardware capabilities and indexes them in a device catalog along with each device's available APIs. The APIs are used to (2) build a series of common memory services to be used by the applications with annotations and desired dataflow. DMS runtime then (3a) jointly optimizes and (3b) executes the applications judiciously using the available devices.

logical memory regions that allow developers to specify desired properties at the memory block level. This is most useful for programming low-level components, such as a B+-tree for a DBMS. The latter models compute tasks and associated logical data transfers into a dataflow graph for execution, which can be useful for building systems such as data analytics pipelines where the application can be expressed as a set of operators, each carrying certain properties such as latency service level objectives (SLOs).

Logical Memory Regions. We categorize memory into *logical* regions to abstract away specifics of the underlying memory devices. A developer would just need to specify the qualitative and/or quantitative properties for interacting with the data stored in the region. For example, if a region is used to keep states that require synchronization across threads, one could mark it as `[shared, coherent]`; if certain SLOs are desired, one could specify such properties as `[latency < 1`μ`s, read bandwidth >= 2GB/s]`. The memory services layer (described later) will map these requirements to the devices using device-specific APIs, which can transparently perform the necessary operations such as spilling/aging and relevant transformations to satisfy the target semantics.

Going back to the case studies, the B+-tree developer may declare a newly allocated tree node to be cacheable with SLOs: `[cacheable, latency < 10`μ`s] LeafNode *n = allocate(...);`. DMS can leverage compute-side DRAM for caching using predefined caching implementations maintained by the calibration layer (described later). In other scenarios such as robust query processing, one annotates the requirements to the private state with performance SLOs (e.g., a hash join's hash table will be consumed by a downstream operator at a rate of X cycles/tuple). In case of insufficient on-device memory, DMS could transparently spill the state's content to local storage [29], remote DRAM or object storage to avoid out-of-memory errors or memory-to-storage performance cliffs.

Table 2: Device catalog in the calibration layer.

Hardware	Capabilities	APIs
Local DRAM	Coherence. Byte-addressable.	`dram-load`, `dram-store`, `dram-dsa`
CXL DRAM	Partial coherence. Byte-addressable.	`cxl-load`, `cxl-store`
Membrane	Compute. Byte-Addressable.	`pim-load`, `pim-store` `pim-offload`

Dataflows. On top of logical memory regions that target individual accesses or tasks, DMS exposes to the developer a dataflow programming model to explicitly model (1) the compute tasks and (2) the *logical* data transfers into the dataflow graph that involve a series of compute and memory accesses. The developer can mark the pipelines of operations that can be potentially merged (task or operator fusion) if the data transfer allows for streaming the data. In case of pipeline breakers, one can explicitly mark the state (e.g., the hash table). These tasks can then be annotated with the aforementioned properties. For example, at the top of Figure 1, $T3$ and $T4$ are respectively marked to be `cacheable` and `offloadable`, which will then be guaranteed by the caching and offloading services using proper devices. In the previous PIM filtering example, to fully utilize the fine-grained data-level parallelism and high internal bandwidth enabled by the PIM device's (Membrane) filtering units across memory banks, the developer can provide annotations that describe the data layout and structure. These annotations guide the system in organizing data to achieve optimal placement, thereby enhancing bank-level parallelism. Additionally, they help maintain a coherent view of data between PIM units and the host, which is critical for ensuring correctness after every update operation. Building on these capabilities, a high-level task graph representation of queries can expose opportunities for operator fusion, allowing multiple filter operations to be offloaded and executed concurrently within the memory system, thereby enhancing performance beyond what traditional CPU-based execution can achieve.

4.2 Calibration Layer

The calibration layer serves as the input to the memory services. It (1) discovers new hardware devices, (2) keeps track of their properties and (3) provides accesses to a set of APIs, each of which implements the features of the hardware. This can be done by maintaining a device catalog, as shown in Figure 1 (bottom right) and Table 2. For example, local DRAM (as seen from the perspective of local CPUs) provides coherence, byte-addressability and can be accessed using `load`/`store` instructions as well as asynchronous data streaming accelerator (DSA) primitives on certain platforms [20].

Meanwhile, PIM devices can support both normal `load`/`store` accesses and offloading, which are implemented using corresponding APIs (`pim-load`, `pim-store` and `pim-offload`). They may also advertise more device-specific capabilities, specifying the data types they support, such as numeric (integer and/or floating-point) vs. string values, and the compute they can perform, such as comparisons, support for regular expressions or simpler, non-regex predicates. These APIs are implemented and maintained by DMS, which internally will require low-level programming with hardware primitives. They are akin to the physical implementations of relational operators: for example, comparing CXL-based vs. PIM-based APIs for the same data access operations is similar to comparing hash join vs. sort-merge join operators. The memory services will then use these APIs to orchestrate the application's execution.

4.3 Memory Services Layer

On top of the APIs maintained by the calibration layer's device catalog, we build a series of memory services that perform the heavy lifting of application execution. The range of memory services (types and amounts) is jointly determined by the variety of available devices and the application's needs, as specified by the user in the declarative abstraction layer.

We envision the memory services layer as a runtime system (software layer) that the DBMS or any other application running on top would interact with. Some of its memory services are provided when invoked (e.g., memory allocation), while others would run in the background to ensure efficient resource usage and SLO enforcement. On allocation, DMS assists with data placement of the logical memory region that best matches the desired properties. For background services we envision lightweight metadata tracking on how the regions are used (e.g., in the context of CXL-root complex where data could be shared/accessed by different devices) as well as data movement, data tiering, data/logical region garbage collection, and data transformations. On a dataflow level, when a logical region is marked with the option for offloading, the runtime engine that schedules the tasks should interact with the memory services, to check if the compute tasks in the pipeline are supported (and at what cost).

In the B+-tree example, the necessary services include caching and data placement. The caching service caches memory regions marked `cacheable` at the compute-side local DRAM, while the data placement service distributes tree nodes across memory servers. These two services remove the need for DEX and Sherman to manually handle compute-side caching and data placement. Additionally, an offloading service can abstract away offloading decisions and implementations from DEX itself, simplifying programming.

In the PIM-based filtering example, instead of requiring programmers to explicitly specify offloading decisions to use Membrane for filtering, a runtime offloading service can dynamically determine whether to offload filter operations based on the size of the columns involved, which typically are only known at runtime. For instance, offloading filters on smaller dimension table columns may incur overheads that outweigh any performance gains.

Finally, for the buffer cache the existing examples already confirmed the benefits of using hints from the DBMS as opposed to opaque OS-provided tiered memory services that try to infer the usage intent of the application running above (with hotness tracking, etc.). But, with DMS we can extended it further. For example, the developer could specify how the workload plans to use the data (e.g., just reading once sequentially for large scans, or many times at random for heavy synchronization among the workers) to guide the data placement better. One could even annotate the likely access patterns for specific applications that are non-intuitive to an opaque storage manager [15]. For example, B+-tree probes follow strict parent-child paths, which motivated DEX to hand-craft a specific path-based caching policy that avoids prematurely evicting inner nodes; with DMS such manual processing can be replaced by a declarative annotation. Similarly, the developer could tailor the spilling process based on the bandwidth requirement from the upstream operators (specified declaratively) and adjust it based on the hardware capabilities of the spill target environment [29]. When knowing the internal structure of the page (e.g., PAX or NSM), one could also hint to the runtime system so a more suitable encoding/compression scheme is chosen when "cooling" the data down the hierarchy.

5 RESEARCH CHALLENGES

Realizing DMS presents challenges and research questions.

Challenge 1 (Calibration): Device Characterization. Beyond straightforward properties such as access granularity (e.g., byte- vs. page-level), more subtle characteristics, such as memory performance under different load levels and consistency guarantees, are much harder to capture and express accurately. A stop-gap solution is for expert DMS developers to hardcode APIs based on the needs from memory services. It remains a challenge to devise a framework that allows the device catalog to self-evolve with the hardware and the requirements from memory services.

Challenge 2 (Abstraction and Memory Services): Mapping Properties to Services. The same service could be built in different ways. For example, both memory servers and PIM devices can implement offloading—*which implementation should we use?* One could implement two variants of the same service—then *how should we decide which service to use?* An interesting direction is to determine this automatically with the help of application hints.

Challenge 3 (Memory Services): Guaranteeing SLAs. It can be particularly challenging to enforce quantitative SLAs (e.g., the average access latency should be within 100ns) as the workload and hardware evolve. The memory services layer can monitor metrics at runtime and adjust if the SLO is not being met, e.g., by migrating to different memory services. Some of these SLAs could overlap or conflict with each other. In multi-tenant scenarios, for example, each tenant may specify a target metric via DMS. Their metrics can conflict (e.g., two tenants respectively prioritize tail latency and throughput) or present joint optimization opportunities (e.g., by sharing intermediate data). The memory services layer should be able to reconcile these conflicts and perform joint optimizations.

Challenge 4 (Calibration and Memory Services): Extensibility and Composability. As new memory devices continue to emerge, DMS should be extensible to be able to capture new device capabilities. This requires defining forward-compatible interfaces, which is an active area [26, 46] and relevant work could be adopted.

Challenge 5 (All Layers): Deployment of DMS. DMS needs to gather program requirements at compile and/or runtime from multiple servers. This indicates a per-server local DMS component plus a global service for the whole system. The availability of the device catalog is also critical. The challenge is to provide high availability for DMS itself while maintaining negligible overhead.

Challenge 6 (All Layers): Correctness and Debugging. Another challenge is ensuring that DMS-based execution indeed satisfies user-defined semantics. Debugging and diagnostic tools also need to be built to allow developers to explore how the program will execute and why an SLO was missed. Tackling these challenges will require revisiting techniques for breakpoints/single-stepping, record-replay [47], telemetry, and debugging declarative queries.

6 RELATED WORK

Prior work has proposed to decouple application semantics from actual implementations. Stateful DataFlow multiGraph [5] proposes a data-centric intermediate representation for code semantics and transformations. This allows optimizing code for different architectures, but still requires performance engineers to interactively modify SDFGs based on program structure, hardware and intermediate performance results. XMem [60] also allows programmers to specify memory properties, which are summarized by a global attributes table and implemented using HW/SW co-design. XMem focuses on low-level DRAM access performance in a single node, whereas DMS provides declarative programming to support various modern memory technologies (beyond DRAM) in both single-node and disaggregated environments. Similar to DMS, prior proposals such as X10 [43] have attempted to decouple data placement and scheduling to a runtime in the context of partitioned global address spaces [12]. DMS further offloads the mapping of services to devices to the runtime, expanding the optimization space. Anneser et al. [4] proposed a vision of a memory-centric, declarative programming model for fully disaggregated systems; DMS adopts its logical memory regions in the abstraction layer. The unified memory framework [61] allows users to manage multiple memory pools characterized by different attributes or using different hardware resources; these pools can be adopted by DMS as building blocks. AIFM [53] proposes application-integrated APIs to make "far" (remote) memory transparent to developers. Beehive [38] allows developers to write synchronous code and automatically transforms it to asynchronous code to extract more throughput. HetCache [45] proposes proportional and access-path-aware optimizations across CPU, GPU and NVMe devices. Earlier systems such as Dryad [22] have proposed to leverage the dataflow model. Data Pipes [62] allows developers to declaratively and explicitly specify data movement between devices, and leaves the choice of data movement primitives to a runtime. DMS takes a step further to allow applications to specify higher level properties without explicitly specifying data movement operations between devices.

7 SUMMARY

We have made the case for declarative programming on disruptive memory technologies, and proposed the vision of declarative memory services (DMS). DMS allows developers to express properties that are enforced by a set of memory services. This decouples programming abstractions and device-specific implementation details to improve portability, frees developers from dealing with complex low-level details per memory device and opens up ample opportunities to build fast and future-proof data systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We are thankful to Schloss Dagstuhl and the organizers and attendees of Dagstuhl Seminar 25151 (Disruptive Memory Technologies). This work was partially supported by an NSERC Discovery Grant (RGPIN-2019-04620), the National Science Foundation under grant no. PPoSS-2217071, PRISM, one of seven centers in JUMP2.0, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, the German Research Council (DFG) through the HetCIM-II project (502388442), CRC/TRR 404-Active 3D (528378584), and NSFC (62125202 and U22B2022).

REFERENCES

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*. 169–180.

[2] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.

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

[4] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. 2023. Programming fully disaggregated systems. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 188–195.

[5] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. Article 81, 14 pages.

[6] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, et al. 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)*. 77–95.

[7] Compute Express Link Consortium. 2024. Compute Express Link (CXL) Specification. https://computeexpresslink.org/wp-content/uploads/2024/11/CXL-Specification_rev3p2_ver1p0_2024October2_evalcopy.pdf

[8] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*. 1–24.

[9] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 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, Volume 3 (ASPLOS 2023)*. 727–741.

[10] Jason Evans. 2006. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of the BSDCan Conference*.

[11] Alessandro Fogli, Bo Zhao, Peter R. Pietzuch, Maximilian Bandle, and Jana Giceva. 2024. OLAP on Modern Chiplet-Based Processors. *Proc. VLDB Endow.* 17, 11 (2024), 3428–3441.

[12] High Performance Fortran Forum. 1994. High Performance Fortran Language Specification (Part III). *SIGPLAN Fortran Forum* 13, 3 (Sept. 1994), 22–55.

[13] 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)*. 649–667.

[14] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 692–708.

[15] Alexander Halbuer, Christian Dietrich, Florian Rommel, and Daniel Lohmann. 2023. Morsels: Explicit Virtual Memory Objects. In *Proceedings of the 1st Workshop on Disruptive Memory Systems, DIMES 2023, Koblenz, Germany, 23 October 2023*. ACM, 52–59.

[16] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*. 372–385.

[17] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes. In *Conference on Innovative Data Systems Research (CIDR 2022)*, Vol. 22. 1–8.

[18] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 257–273.

[19] IEEE and The Open Group. 2016. The Open Group Base Specifications Issue 7, IEEE Std 1003.1. (2016).

[20] Intel. 2025. Intel Data Streaming Accelerator (Intel DSA). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html.

[21] Intel Corporation. 2025. Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. 59–72.

[23] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. 2022. Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM. *IEEE Micro* 42, 1 (2022), 116–127.

[24] Asif Ali Khan, João Paulo C. de Lima, Hamid Farzaneh, and Jeronimo Castrillon. 2024. The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview. *CoRR* abs/2401.14428 (2024).

[25] Asif Ali Khan, Hamid Farzaneh, Karl F. A. Friebel, Clement Fournier, Lorenzo Chelini, and Jeronimo Castrillon. 2024. CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24) (ASPLOS '24)*.

[26] Abigale Kim, Marco Slot, David G Andersen, and Andrew Pavlo. 2025. Anarchy in the Database: A Survey and Evaluation of Database Management System Extensibility. *Proc. VLDB Endow.* 18, 6 (2025), 1962–1976.

[27] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Taranov Konstantin, Dejan Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Conference on Innovative Data Systems Research (CIDR 2022)*, Vol. 22.

[28] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. 3753–3766.

[29] Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. 2024. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance. *Proc. ACM Manag. Data* 2, 6 (2024), 238:1–238:27.

[30] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.

[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 (ASPLOS '19)*. 317–330.

[32] Sukhan Lee, Shinhaeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*. 43–56.

[33] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 17–34.

[34] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. In *Programming Languages and Systems*. 244–265.

[35] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25.

[36] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 556–569.

[37] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 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*. 574–587.

[38] Quanxi Li, Hong Huang, Ying Liu, Yanwen Xia, Jie Zhang, Mosong Zhou, Xiaobing Feng, Huimin Cui, Quan Chen, Yizhou Shan, and Chenxi Wang. 2025. Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 167–187.

[39] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2009. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Article 14, 12 pages.

[40] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proc. VLDB Endow.* 17, 10 (June 2024), 2603–2616.

[41] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[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 Languages and Operating Systems, Volume 3*. 742–755.

[43] Josh Milthorpe, V. Ganesh, Alistair P. Rendell, and David Grove. 2011. X10 as a Parallel Language for Scientific Computation: Practice and Experience. In *2011 IEEE International Parallel & Distributed Processing Symposium*. 1080–1088.

[44] Khang T Nguyen. 2016. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html

[45] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. In *Conference on Innovative Data Systems Research (CIDR 2023)*.

[46] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (June 2023), 2679–2685.

[47] Andrew Quinn and Peter Alvaro. 2025. Deterministic Record-and-Replay. *Commun. ACM* 68, 5 (April 2025), 32–34.

[48] Jun Rao and Kenneth A. Ross. 2000. Making B+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. 475–486.

[49] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.

[50] Amanda Raybuck, Wei Zhang, Kayvan Mansoorshahi, Aditya K Kamath, Mattan Erez, and Simon Peter. 2023. MaxMem: Colocation and Performance for Big Data Applications on Tiered Main Memory Servers. *arXiv preprint arXiv:2312.00647* (2023).

[51] 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, Volume 1*. 351–367.

[52] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. 2024. A Three-Tier Buffer Manager Integrating CXL Device Memory for Database Systems. In *40th International Conference on Data Engineering, ICDE 2024 - Workshops, Utrecht, Netherlands, May 13-16, 2024*. 395–401.

[53] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[54] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. 2020. Memory devices and applications for in-memory computing. *Nature Nanotechnology* 15, 7 (2020), 529–544.

[55] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 273–287.

[56] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. 510–521.

[57] Akhil Shekar, Kevin Gaffney, Martin Prammer, Khyati Kiyawat, Lingxi Wu, Helena Caminal, Zhenxing Fan, Yimin Gao, Ashish Venkat, José F. Martínez, Jignesh

Patel, and Kevin Skadron. 2025. Membrane: Accelerating Database Analytics with Bank-Level DRAM-PIM Filtering. arXiv:2504.06473

[58] Farzana Ahmed Siddique, Deyuan Guo, Zhenxing Fan, Mohammadhosein Gholamrezaei, Morteza Baradaran, Alif Ahmed, Hugo Abbot, Kyle Durrer, Kumaresh Nandagopal, Ethan Ermovick, Khyati Kiyawat, Beenish Gul, Abdullah Mughrabi, Ashish Venkat, and Kevin Skadron. 2024. Architectural Modeling and Benchmarking for Digital DRAM PIM. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. 247–261.

[59] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 1541–1555.

[60] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B. Gibbons, and Onur Mutlu. 2018. A case for richer cross-layer abstractions: bridging the semantic gap with expressive memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. 207–220.

[61] Sergei Vinogradov, Igor Chorazewicz, Piotr Balcer, and Rafal Rudnicki. 2024. Unified Memory Framework. In *Proceedings of the Future of Memory and Storage*. https://github.com/oneapi-src/unified-memory-framework

[62] Lukas Vogel, Daniel Ritter, Danica Porobic, Pinar Tözün, Tianzheng Wang, and Alberto Lerner. 2023. Data Pipes: Declarative Control over Data Movement. In *Conference on Innovative Data Systems Research (CIDR 2023)*.

[63] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 79–94.

[64] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 161–179.

[65] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*. 1033–1048.

[66] 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)*. 609–621.

[67] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 307–323.

[68] Lingxi Wu, Rasool Sharifi, Ashish Venkat, and Kevin Skadron. 2022. DRAM-CAM: General-Purpose Bit-Serial Exact Pattern Matching. *IEEE Computer Architecture Letters* 21, 2 (2022), 89–92.

[69] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 19–35.

[70] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 817–833. https://www.usenix.org/conference/atc24/presentation/xu-dong

[71] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS '25)*. 689–702.

[72] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *Symposium on Operating Systems Design and Implementation*.

[73] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David E. Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2195–2207.

[74] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proc. ACM Manag. Data* 1, 2, Article 131 (June 2023), 26 pages.