Programming Fully Disaggregated Systems

Christoph Anneser Lukas Vogel Ferdinand Gruber
Maximilian Bandle Jana Giceva
Technical University of Munich
firstname.lastname@in.tum.de

Abstract

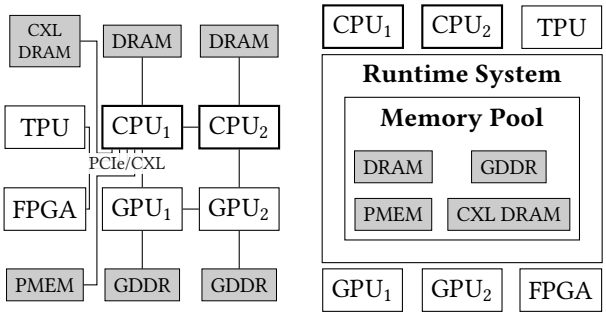
With full resource disaggregation on the horizon, it is unclear what the most suitable *programming model* is that enables dataflow developers to fully harvest the potential that recent hardware developments offer. In our vision, we propose to raise the abstraction level to allow developers to primarily reason about their dataflow and the requirements that need to be met by the underlying system in a declarative fashion. Underneath, the system works with typed memory regions and uses the notion of ownership that allows for more flexible memory management across the different compute devices and the tasks mapped onto them. This requires a holistic approach that crosses multiple layers of the system stack, opening exciting systems research questions.

ACM Reference Format:
Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. 2023. Programming Fully Disaggregated Systems. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595889

1 Introduction

With the ever-increasing demand for data, where the datasphere volume is expected to reach 175ZB by 2025 [50], we have reached the point where moving data is the dominating cost factor in data centers [34, 45]. Cloud providers race to serve the different requirements of modern workloads better but with pressure to achieve it in a more sustainable fashion [51]. To improve efficiency, data centers have evolved to more loosely coupled software-defined racks, where they disaggregate resources over fast network interconnects [52].

However, until recently, coherent memory remained tightly coupled, and servers had to be equipped with large memory capacities to serve peak workloads reliably. This overprovisioning is a considerable cost (50% of Azure's servers [5] and 40% of Meta's rack costs come from memory [40]) for a resource that could not be properly pooled. The average memory utilization reported by many cloud vendors remains low, typically in the range of 50-65% [38, 56]. Therefore, data centers could reduce costs by pooling different types of memory [9, 11, 21, 57] and compute devices [6, 13, 17–19, 30, 33, 47] by connecting them with fast networks [14, 45].



(a) Current Architectures **(b) Our Vision**

Figure 1: Moving from a compute-centric to a memory-centric architecture.

However, data and compute placement within these pools significantly impacts the overall system performance. For example, non-uniform memory accesses (NUMA) can slow down algorithms by up to 3× [39]. Similarly, a naïve data placement in a heterogeneous storage landscape can reduce a database system's performance by up to 3× [59].

Moreover, today, optimal placement has become an issue even *within* single processors. For example, take the recently introduced Intel's 4th Generation Intel® Xeon® Scalable Processors – codenamed *Sapphire Rapids* [7]. They have built-in encryption, compression, streaming, and high-bandwidth memory accelerators. Its most promising feature, however, is the adoption of Compute Express Links™ (CXL™) – an industry standard for cache-coherent interconnects for processors, memory expansion, and accelerators based on PCIe 5.0, which has been adopted by companies like Intel, AMD, ARM, Samsung, and NVIDIA, amongst others [9]. CXL enables us to first *scale-up* nodes by extending their compute and memory pools with 'pluggable' compute devices and DRAM/PMem expansion cards before we have to rely on more expensive 'scale-outs' to other compute nodes that

HOTOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595889

Table 1: Memory device properties as seen from a CPU.

Name	Bw.	Lat.	Gran.	Attached	Sync	Persist.
Cache	++	++	1 B	CPU	✓	✗
HBM	++	+	64 B	CPU	✓	✗
DRAM	+	+	64 B	CPU	✓	✗
PMem	○	○	256 B	CPU	✓	✓
CXL-DRAM	○	○	64 B	PCIe	✓/ ✗	✓/ ✗
Disagg. Mem.	○	–	?	NIC	✗	✓/ ✗
SSD	–	–	4 KiB	PCIe	✗	✓
HDD	––	––	4 KiB	SATA	✗	✓

would require the implementation of more complex consistency protocols [38]. Furthermore, CXL-based compute devices can coherently access and cache host CPU memory, enabling new data and compute placement combinations but making optimal placement decisions much more complex, as Figure 1a shows.

We believe that in such a heterogeneous hardware landscape, existing programming models are not suitable anymore. Traditionally, a developer has to explicitly place data on a memory device and specify which accelerator performs the computation. In particular, this explicit data placement requires the developer to be aware of various memory types' different properties, as shown in Table 1. For example, to optimize applications and data placement, developers must consider access latencies, their granularities (bytes or logical blocks), and how devices are physically attached. Otherwise, they will be unable to fully unlock the potential of these exciting, emerging hardware platforms. Unsurprisingly, this topic is being discussed in several recent proposals on how to write programs for scale-out cloud systems [20, 29, 61] and how to do memory-tiering at warehouse scale [22, 40].

Recent work suggested that we should switch away from CPU- or process-centric architectures to overcome the complexity of disaggregated systems and thus allow developers to primarily focus on their application logic [22, 23, 28, 53, 54, 60]. For example, by lifting the abstraction level, Vogel et al. proposed a new framework enabling developers to get declarative control over data movement in heterogeneous, disaggregated environments [58], while HetCache co-optimizes data placement by taking different memory, compute devices, and queries into account [43].

In this paper, we ask '*what should be the appropriate programming model for implementing various dataflow frameworks in the era of full resource disaggregation.*'

2 Vision

This section presents our envisioned programming model and runtime system that would enable the writing of scalable code that leverages modern hardware with disaggregated compute and memory pools.

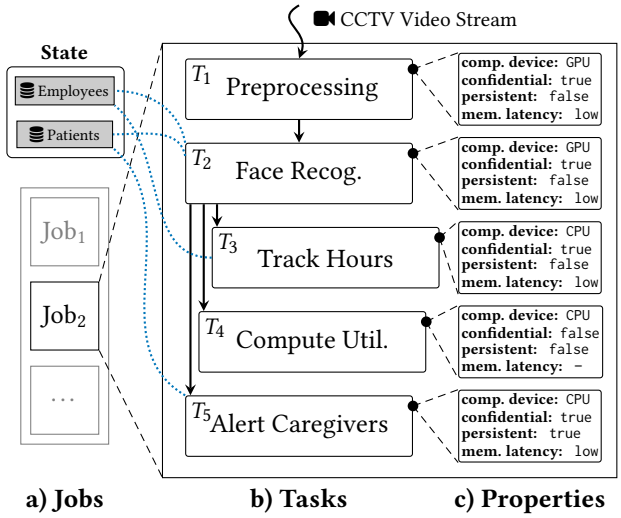


Figure 2: Example dataflow system of a hospital. Jobs consist of tasks that form a directed acyclic graph. Properties can be attached to tasks and dataflows.

2.1 Foundations

Data-intensive applications like database systems [42], machine learning frameworks [3, 4], or large-scale data analytics platforms [1, 2] can often be generalized to *dataflow systems*. To introduce the concepts of our approach, therefore, we rely on their well-known architecture, where applications launch *jobs* that consist of *tasks*. Tasks represent computational units, and connecting arrows between tasks represent the dataflow and its direction. Connected tasks form a directed acyclic graph.

Example. Figure 2 shows an example of such a job (2a) consisting of 5 tasks (2b): A hospital might have a CCTV camera recording entering and leaving persons (T_1) using GPU-accelerated face recognition connected to an employee and patient database (T_2). This information is then used to track the working hours of the employees (T_3), feed a public website displaying the utilization of the emergency ward (T_4), and alert caregivers if a confused patient exits the hospital and does not reappear after a grace period (T_5).

Declarative programming. As described in Section 1, recently introduced hardware platforms (such as Sapphire Rapids [7]) have built-in accelerators and support CXL1.1, allowing to *scale-up* one node with more accelerators and coherent memory expansion. From a programmer's perspective, implementing and optimizing applications, such as the hospital's dataflow, for modern hardware becomes increasingly complex and time-consuming. One viable option for developing high-performance dataflow applications is to introduce a new abstraction layer. This abstraction would hide the details of compute and memory devices during the application's *development* and defer the compute and memory

placement decisions to *runtime*. Declarative programming concepts could allow developers to focus on the application logic (*what*) rather than *how* it is executed on a specific platform.

Common patterns. Today's dataflow applications share common patterns and often have similar requirements. For example, processing sensitive user data (i.e., T_2) requires them to implement security standards, including data encryption. Jobs and tasks could be either streamed or processed in batches. Machine learning-related tasks benefit from hardware acceleration. Implementing such requirements for each dataflow system individually and optimizing it to run on disaggregated systems is time-consuming and error-prone.

Properties for dataflow systems. Instead, a programming model should enable developers to attach common properties to their dataflow applications at different granularities. In Figure 2c), each task has some properties: While the video feed's confidentiality might depend on the country, the employee and patient database, and the tagged and cross-referenced persons are confidential. Furthermore, the video feed itself is not latency-sensitive, but since image recognition is computationally intensive, it requires low-latency memory (from the view of the GPU) to allow for real-time face recognition. The alerting task (T_5) has to store missing patients persistently, as a system crash would otherwise mean they might be forgotten. Furthermore, by attaching the property *confidentiality* to the tasks T_1–T_3 and T_5 in Figure 2, the application developer can indicate that the processed data is sensitive and must not be visible to other tasks or jobs. Another recurring pattern is the *materialization* of output data, as is the case for materialized views in database systems or the neural network's weights after training, making it another good candidate for a property being attached in dataflow systems.

Requesting properties. Current disaggregated systems introduce various memory devices, each having different properties regarding latency, bandwidth, persistency, and others (cf. Table 1). Deploying dataflow systems that serve thousands of jobs in parallel on such complex hardware landscapes with multiple physical memory devices makes efficient memory management more challenging, especially when tasks are deployed on different compute devices and the performance-critical inter-task communication is being implemented via message-passing over shared memory [41]. Therefore, the physical memory devices should be made transparent to applications that instead request memory based on the required properties. For example, the application could specify whether the allocated memory should be persistent and what latencies or bandwidths are acceptable.

Ownership Chunks of memory requested in such a way would then have a clear *owner* (i.e., a task, a job, or the whole application) allowing us to reason about the lifetime of chunks of memory and be aware of when we can re-assign it to new tasks. We could, thus, implement a reusable optimizer for various dataflow systems' data placement.

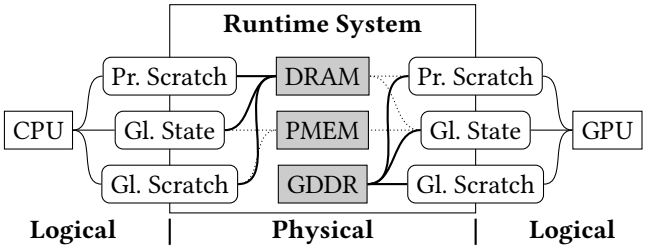


Figure 3: Mapping logical Memory Regions to physical memory depends on the compute device.

Summary. Given the challenges listed above, we need a programming model that enables application developers to utilize modern, disaggregated hardware platforms more efficiently. Such a model ideally enables the developer to attach commonly seen properties to tasks and facilitates managing disaggregated memory declaratively, which makes not only the application *development* more efficient but also the *applications* themselves by automatically co-optimizing data placement and the overall resource utilization.

2.2 Mapping to Disaggregated Systems

While the envisioned programming model abstracts from specific memory devices and instead lets the application specify what properties the requested memory must have, we need a runtime system that maps logical requests to the physical hardware in the background.

Memory devices. As shown in Table 1, various devices are already contributing to the pool of disaggregated memory, with more being added in the future. Each device has different properties concerning latency, bandwidth, coherency, and persistency. The mapping from a task's memory request and its declared properties must therefore be matched to the underlying hardware, which leads to three challenges:

(1) The 'optimal' memory device depends on the compute device executing the task *and* the type of memory accesses it performs (e.g., random vs. sequential, read- or write-intensive accesses, or access granularity). Figure 3 visualizes this problem: 'fast and local' scratch memory might preferably be DRAM when the task runs on a CPU. For tasks running on a GPU, however, GDDR provides better latency and bandwidth, although with less capacity.

(2) Tasks might share memory: The preceding task's output could become the succeeding task's input. If both tasks run on different compute devices, their shared memory must be addressable by both (e.g., via `CXL.mem`) or copied after the first task is done. Therefore, data placement depends not only on one task but also on the interaction of multiple tasks.

Table 2: Common Memory Regions.

Name	Properties	Purpose
Global State	{coherent, sync}	Syncing tasks
Global Scratch	{coherent, async}	Data exchange
Private Scratch	{noncoherent, sync}	Thread-local data

(3) Depending on how far memory is physically away, we want to expose different interfaces. In the case of *near* memory that provides low access latency, we would prefer synchronous `loads/stores` to reduce the task's execution time. If memory is 'far away', we should switch to an asynchronous interface that fetches memory in the background. For example, accessing CXL-attached memory will result in high latency comparable to accessing DRAM on a different NUMA socket. Asynchronous accesses improve the accelerator's utilization and overall throughput.

We propose three concepts to mitigate these problems:
(1) Memory regions. Since the properties of a device change depending on the task's point of view (i.e., by which compute device it is executed), we use the concept of *Memory Regions* to abstract from physical devices. A Memory Region is a *logical view* on a physical device: It guarantees some set of properties specified by a task (e.g., low latency, persistency) *relative to* the executing compute device. At runtime, the system maps the Memory Region to a physical device satisfying its properties. Memory Regions are thus *declared* and identified by their *properties*, not by their *location*, unlike traditional approaches. We group properties that are often used together and name the resulting Memory Region to express commonly used abstractions in programming – Table 2 describes three frequently used Memory Regions and Figure 4 shows how our dataflow system uses them: All threads of a task have their Private Scratch and hold a reference to a Global State and Global Scratch. Memory regions for dataflow systems for a *single* device have already been used in the past. Broom [25], for example, introduces memory regions and ownership to track lifetimes and, therefore, to remove the garbage collector. We build on this approach by generalizing memory regions to *multiple* devices.
(2) Memory ownership. To facilitate inter-task communication, we introduce the concept of *memory ownership*: Each chunk of allocated memory is either

- *exclusively* owned by a task. This applies if it is just task-local scratch space or handed over to the next task after completion. Here, consistency guarantees and memory ordering can be relaxed.
- or it *shares* the ownership with other tasks that may run concurrently. This puts additional requirements on the Memory Region, i.e., being cache-coherent or having strict memory ordering.

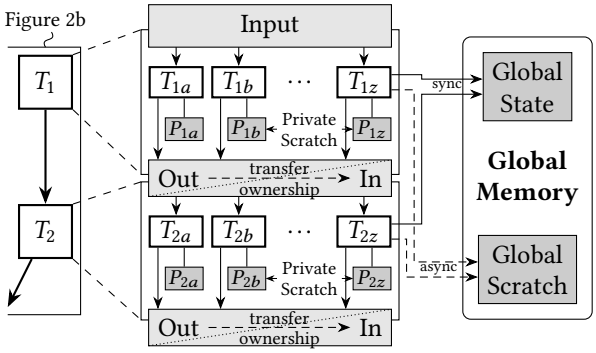


Figure 4: Tasks and Typed Memory Regions.

Note that memory being owned *exclusively* by a task does not mean it can only ever be owned by one thread of execution. As Figure 4 shows, ownership can be *transferred*, i.e., a reference to the memory chunk can be passed to the next task (the "`out`" becomes the "new `in`"), which is similar to `C++`'s move semantics. This explicit ownership model enables us to always allocate the most suitable memory per thread of execution. In contrast, in a traditional disaggregated system, users must choose placement, which increases complexity, especially as more kinds of memory become available.
(3) Access interfaces. It is beneficial to address different Memory Regions by different access modes to improve resource utilization. When accessing global memory, we might benefit from an asynchronous model where we can interleave computation with memory accesses. Memory Regions, thus, should expose different interfaces to access data.

2.3 Programming Model

After introducing the abstractions of Memory Regions, ownership, and interfaces, we now switch back to the application level and discuss integrating these concepts into the motivating dataflow example.
Runtime system. To implement the envisioned programming model, we need a runtime system that is responsible for (1) determining at runtime which physical memory device best fits each task's declared requirements, (2) allocating the Memory Regions that tasks have requested, (3) de-allocating Memory Regions after the last owning task finishes, (4) and resource-aware task scheduling.
Abstracting memory regions. As discussed in the previous section and Table 2, dataflow systems share common memory usage and access patterns and have similar requirements. The following Memory Regions should be pre-defined by the programming model:

- **Private Scratch** is memory local to each thread of the task. Since it is not shared, it may have relaxed coherence guarantees. As demonstrated in Figure 4, each task's thread has its own private scratch space ($P_{1a} \ldots P_{1z}$), which is only

Table 3: How applications may use memory regions.

	Priv. Scratch	Glob. State	Glob. Scratch
DBMS	operator state (hashtables, …)	synchronization (latches, …)	(temp) indexes, caches
ML/AI	model training state	metadata, worker state	input data, cached transf. data
HPC	node-local working mem.	job metadata, node states	object/blob storage
Streaming	cache/buffer (send, recv.)	cluster/worker state	result/data cache

alive during its execution. It stores intermediate results not part of the task's output. Private scratch is visible to only one thread and not transferable.

- **Global State** is memory global to the application and shared between tasks to synchronize tasks and threads. It, thus, has to provide strict coherence guarantees and strong memory ordering but is expected to be slow as it has to be accessible from all compute devices.
- **Global Scratch** can pass data between tasks that are not connected. Passing data in such a way is helpful when two tasks do not depend on each other but may use an intermediate result from another task (such as a bloom filter) to speed up its processing. The Global Scratch exposes an asynchronous interface as threads should not block on `load/store` on slow memory.

Moving data. Data will be passed between tasks via Global Scratch memory or to the next task in the dataflow. For the second case, we need a concept of input and output as shown in Figure 4. The input consists of the data set the current task should operate on and is generated by the preceding task. The output is the data the task produces, i.e., the next task's input. Input and output can be modeled as Memory Regions, which the active task owns. Thanks to our concept of memory ownership, the output memory of the preceding task can directly become the input memory of the next task if it is addressable by the compute devices of both tasks. The runtime system allocates input and output memory so that handover is just a memory ownership transfer, and physical data movement is minimized.

2.4 Mapping Application Types

Different application types can be easily mapped to our proposed architecture. We illustrate four types in Table 3 and describe two in more detail.

Database systems internally represent queries as relational operator trees where the output of one operator becomes the input of the following operator, which nicely maps onto dataflow systems. Each operator must keep track of its state in private scratch (e.g., a group hash table for aggregation operators) and synchronize with other concurrently running operators via latches in the global state. Furthermore, some operators can re-use (transient) results of earlier operators stored in the global scratch space (e.g., a hash join might re-use a hash index created by an aggregation operator).

AI/ML applications must first transform and preprocess the input data (e.g., parsing, sampling, and feature extraction) before training a model on accelerators. This can also be modeled as a dataflow system, as demonstrated by Cachew [26]. Cachew stores the transformed data in a cache (global scratch) and uses a dispatcher accessing worker states (global state) to assign tasks running on accelerators (private scratch).

3 Discussion

Our proposed memory-centric programming model radically changes how applications and developers interact with memory in the disaggregated cloud. They should no longer have to deal with the complexity of handling different memory devices, which is further complicated by emerging technologies like CXL. Instead, memory should be requested declaratively based on desired properties like latency or bandwidth.

The way forward. Our programming model requires a runtime system (RTS) that should abstract away hardware-specific details of memory accesses and does the bookkeeping regarding ownership and the lifetime of regions. Furthermore, the RTS needs to make the deployment decisions on mapping tasks and memory onto the disaggregated resources. To make this come true, we need to address several challenges for which we begin the discussion in this section:

(1) Who oversees the management and utilization of the disaggregated resources?
(2) How do we make optimal deployment decisions?
(3) How to enforce deployment policies at runtime?
(4) Where should the RTS/control plane be placed?
(5) What support from the underlying system stack is needed on the critical data path?
(6) How can we make our concept of memory regions easy to use in general-purpose programming languages?
(7) How can we combine declarative and imperative principles in one programming model?
(8) What are the potential limitations of our approach?

Implementing the programming model and the runtime system is a non-trivial endeavor that calls for a holistic approach, crossing multiple layers of the systems stack. In the following paragraphs, we discuss how we can address these challenges and use prior work from the systems, compiler, and database communities to set the stage for our proposal.

Challenges 1-3: What is required from the RTS? Our RTS needs to manage memory resources –typed with different properties (cf. Figure 2)– of multiple machines or even cluster wide. Jobs and tasks request memory regions from the RTS that it then maps to physical memory (cf. Figure 3).

The allocation of memory regions goes beyond the capabilities of already known single-host memory management [46] and existing distributed managed runtimes [2]. With multiple, coherently accessible memory tiers, the RTS must manage memory regions based on pages or objects and their placement. Both approaches are actively researched by the systems community and have different implications on performance and scalability [48, 62]. To optimize the placement of memory regions, we can build on recent work that used pointer tagging to track the hotness of pages or objects and to implement remotable pointers that either point to objects in local or in remote memory (pointer swizzling) [37, 40, 48, 62].

Our RTS must also schedule and map tasks to different types of devices using cost models that consider topology and access paths [49] to optimize for concurrently running jobs. Therefore, it must know or predict the resource utilization of memory and compute devices. Scheduling also requires reusing results to avoid unnecessary copying [60] and lowering to different types of hardware. Thankfully, such cost models for optimization and lowering tasks to multiple devices are already well-known in the database and compiler communities [24, 35, 54]. Furthermore, new approaches using MLIR, such as LingoDB [31], have shown that it is feasible to provide the compiler with various statistics to make cost-based transformations and data and task placement decisions.

Challenges 4-5: What layer supports the RTS memory deployment? The RTS provides memory regions, but without some levels of abstraction, the complexity of handling disaggregated memory is just moved to the application. In our vision, the core responsibility of the operating system (OS) is mapping RTS-requested memory into the address space of our proposed tasks. The *processor-centric* design could lead to host congestion [10] and become a bottleneck in the future and the concept of memory ownership of today's OSes are not suitable anymore [53] because ownership is now globally managed by the RTS [23]. Thus, in the disaggregated cloud, OSes should be built *memory-centric*, like our jobs and tasks. Of course, this is a simplification of OS memory management, leaving out many aspects (e.g., the memory the OS requires for managing devices). We are not the first to propose such a shift in OS design and can rely on previous research [23, 53].

Challenges 6-7: How to get the developer on board and ease adoption? Until now, there is no consensus on handling the ownership and lifetime of memory objects and streams across different devices, and popular AI/ML frameworks handle them differently [8]. Furthermore, developers should not face the complexity of modern memory technology [61] and instead should request memory declaratively. This declarative approach is a paradigm shift for many programming languages (PL), where memory is managed manually or by a language runtime [46]. Consequently, the PL should either (1) allow programmers to provide different versions of code targeting different memory types or (2) provide a central compilation service that JIT compiles the programmer's declarative description of memory accesses. The latter –a mixture of declarative and imperative code– is actively researched [44, 55] and could be adapted for our approach.

Challenge 8: What are the potential limitations? Raising the abstraction level leads to new questions our approach does not yet solve: (1) How can we *debug*, *profile*, and *optimize* dataflow applications with multiple abstraction layers for performance when the runtime system hides performance-relevant details? Fortunately, the system community has already shown that – despite intricacies and difficulties – debugging [32] and profiling [16] across multiple abstraction layers is possible. (2) Legacy applications might not adopt a new programming model requiring significant source code modifications. A similar approach has been recently proposed by the Mojo programming language, which is a superset of Python and uses declarative programming to enable hardware acceleration with GPUs and FPGAs for AI and ML workloads. (3) How to mitigate faults and report them to the user? Network errors, corrupted memory, and planned and unplanned node faults such as kernel updates or power outages are common in data centers having thousands of interconnected compute and memory devices. If not handled properly, failures may lead to data loss and force applications to stop and restart. Therefore, our programming model and its runtime system must implement suitable mechanisms that guarantee fault tolerance *and* are compute- and storage-efficient. Several ideas have been recently discussed by the systems community, including replication-based approaches [12, 27, 53] and the striping of memory pages across multiple memory nodes [36]. The runtime system could also implement a combination of erasure-coding, one-sided remote memory accesses and compaction, and off-loadable parity calculations, as it is used by Carbink, a state-of-the-art approach for fault-tolerant far memory [62].

Conclusion. With our envisioned programming model, application developers can fully utilize emerging new hardware more easily without being concerned about the specifics of the underlying hardware or the complexity of memory coherency models. Building a distributed RTS is a complex task requiring support from the systems, compiler, and language community (cf. Legion [15]). However, the advantages of our proposal in terms of complexity reduction, resource utilization, and flexibility will make this effort worthwhile.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was funded by the German Research Foundation (DFG) within the SPP2037 under grant no. Ke 401/22-2.

References

[1] 2013. PrestoDB. https://prestodb.io/ Last Accessed: 2023/07/10.

[2] 2014. Apache Spark. https://spark.apache.org/ Last Accessed: 2023/07/10.

[3] 2015. Tensorflow. https://www.tensorflow.org/ Last Accessed: 2023/07/10.

[4] 2016. PyTorch. https://pytorch.org/ Last Accessed: 2023/07/10.

[5] 2020. CXL and GEN-Z iron out a coherent interconnect strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/ Last Accessed: 2023/07/10.

[6] 2020. Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/ Last Accessed: 2023/07/10.

[7] 2023. 4th Generation Intel® Xeon® Scalable Processors. https://ark.intel.com/content/www/us/en/ark/products/series/228622/4th-generation-intel-xeon-scalable-processors.html Last Accessed: 2023/07/10.

[8] 2023. Apache Arrow. https://github.com/apache/arrow/pull/34972 Last Accessed: 2023/07/10.

[9] 2023. Compute Express Links. https://www.computeexpresslink.org/download-the-specification Last Accessed: 2023/07/10.

[10] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David E. Culler, and Amin Vahdat. 2022. Understanding host interconnect congestion. In *HotNets*. ACM, 198–204.

[11] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference*. USENIX Association, 775–787.

[12] 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 *EuroSys*. ACM, 14:1–14:16.

[13] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[14] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, István Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. 2020. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *SIGCOMM*. ACM, 782–797.

[15] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *SC*. IEEE/ACM, 66.

[16] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. Profiling dataflow systems on multiple abstraction levels. In *EuroSys*. ACM, 474–489.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95.

[18] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *FAST*. USENIX Association, 29–41.

[19] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *MICRO*. IEEE Computer Society, 7:1–7:13.

[20] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. 2021. New Directions in Cloud Programming. In *CIDR*. www.cidrdb.org.

[21] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*. USENIX Association, 401–414.

[22] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David E. 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 *ASPLOS (3)*. ACM, 727–741.

[23] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan S. Milojicic. 2015. Beyond Processor-centric Operating Systems. In *HotOS*. USENIX Association.

[24] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. 1996. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *VLDB*. Morgan Kaufmann, 378–389.

[25] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *HotOS*. USENIX Association.

[26] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In *USENIX Annual Technical Conference*. USENIX Association, 689–706.

[27] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *NSDI*. USENIX Association, 649–667.

[28] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 417–433.

[29] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*. ACM, 1–12.

[31] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401.

[32] Timo Kersten and Thomas Neumann. 2020. On another level: how to debug compiling query engines. In *DBTest@SIGMOD*. ACM, 2:1–2:6.

[33] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP*. ACM, 756–771.

[34] Peter M. Kogge and John Shalf. 2013. Exascale Computing Trends: Adjusting to the "New Normal"' for Computer Architecture. *Comput. Sci. Eng.* 15, 6 (2013), 16–26.

[35] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020).

[36] Youngmoon Lee, Hassan Al Maruf, Mosharaf Chowdhury, and Kang G. Shin. 2019. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *CoRR* abs/1910.09727 (2019).

[37] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE Computer Society, 185–196.

[38] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*.

[39] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*. www.cidrdb.org.

[40] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *ASPLOS (3)*. ACM, 742–755.

[41] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.

[42] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[43] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU-acceleration for Memory-Efficient Analytics. In *CIDR*. www.cidrdb.org.

[44] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015.

[45] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Muhammad Mukarram Bin Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM*. ACM, 66–85.

[46] Paula Pufek, H. Grgic, and Branko Mihaljevic. 2019. Analysis of Garbage Collection Algorithms and Memory Management in Java. In *MIPRO*. IEEE, 1677–1682.

[47] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*. IEEE Computer Society, 13–24.

[48] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *OSDI*. USENIX Association, 315–332.

[49] Enrico Russo, Maurizio Palesi, Salvatore Monteleone, Davide Patti, Giuseppe Ascia, and Vincenzo Catania. 2022. MEDEA: A Multi-objective Evolutionary Approach to DNN Hardware Mapping. In *DATE*. IEEE, 226–231.

[50] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. 2018. The digitization of the world from edge to core. *Framingham: International Data Corporation* 16 (2018).

[51] Ian Schneider. 2022. Building low-carbon computer systems: when does carbon diverge from cost? [Talk]. https://youtu.be/W7uTbxCxmPg Last Accessed: 2023/07/10.

[52] Boris M Shabanov and Oleg I Samovarov. 2019. Building the software-defined data center. *Programming and Computer Software* 45 (2019), 458–466.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2019. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *USENIX ATC*. USENIX Association.

[54] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. 2022. Towards a fully disaggregated and programmable data center. In *APSys*. ACM, 18–28.

[55] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow.* 15, 5 (2022), 1119–1131.

[56] 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 *EuroSys*. ACM, 30:1–30:14.

[57] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *VLDB J.* 29, 6 (2020), 1223–1241.

[58] Lukas Vogel, Daniel Ritter, Danica Porobic, Pınar Tözün, Tianzheng Wang, and Alberto Lerner. 2023. Data Pipes: Declarative Control over Data Movement. In *CIDR*. www.cidrdb.org.

[59] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2662–2675.

[60] Stephanie Wang, Benjamin Hindman, and Ion Stoica. 2021. In reference to RPC: it's time to add distributed memory. In *HotOS*. ACM, 191–198.

[61] Yiying Zhang, Ardalan Amiri Sani, and Guoqing Harry Xu. 2021. User-defined cloud. In *HotOS*. ACM, 33–40.

[62] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *OSDI*. USENIX Association, 55–71.