Far Memory for Data-Intensive Apps

Daniel Bittman, CTO
Pankaj Mehra, CEO (and Professor at OSU)

Elephance Memory

Agenda

- Context
- Technology
- Roadmap

Elephance does for Memory what S3 did for Storage

Elephance

A Breakthrough Platform that Delivers Memory-as-a-Service into the Data Infrastructure Market

Why?

High cost and poor utilization of memory, the last pre-virtualization resource in the data center

Why now?

Storage Class Memory (SCM) failed to become an alternative to expensive DRAM. Memory pooling is now regarded as the best solution. CXL gathering steam with PCIe 5 and Sapphire Rapids launch.

Why Elephance?

MemOS was built from the ground up to optimize efficient memory pooling. Fundamentally better^ than Hyperscaler and OEM solutions such as Google Zswap and Carbink, VMware AIFM, and Azure CompuCache.

✔ **Award-Winning* Foundation**

Core Technology: Rearchitected Memory Management

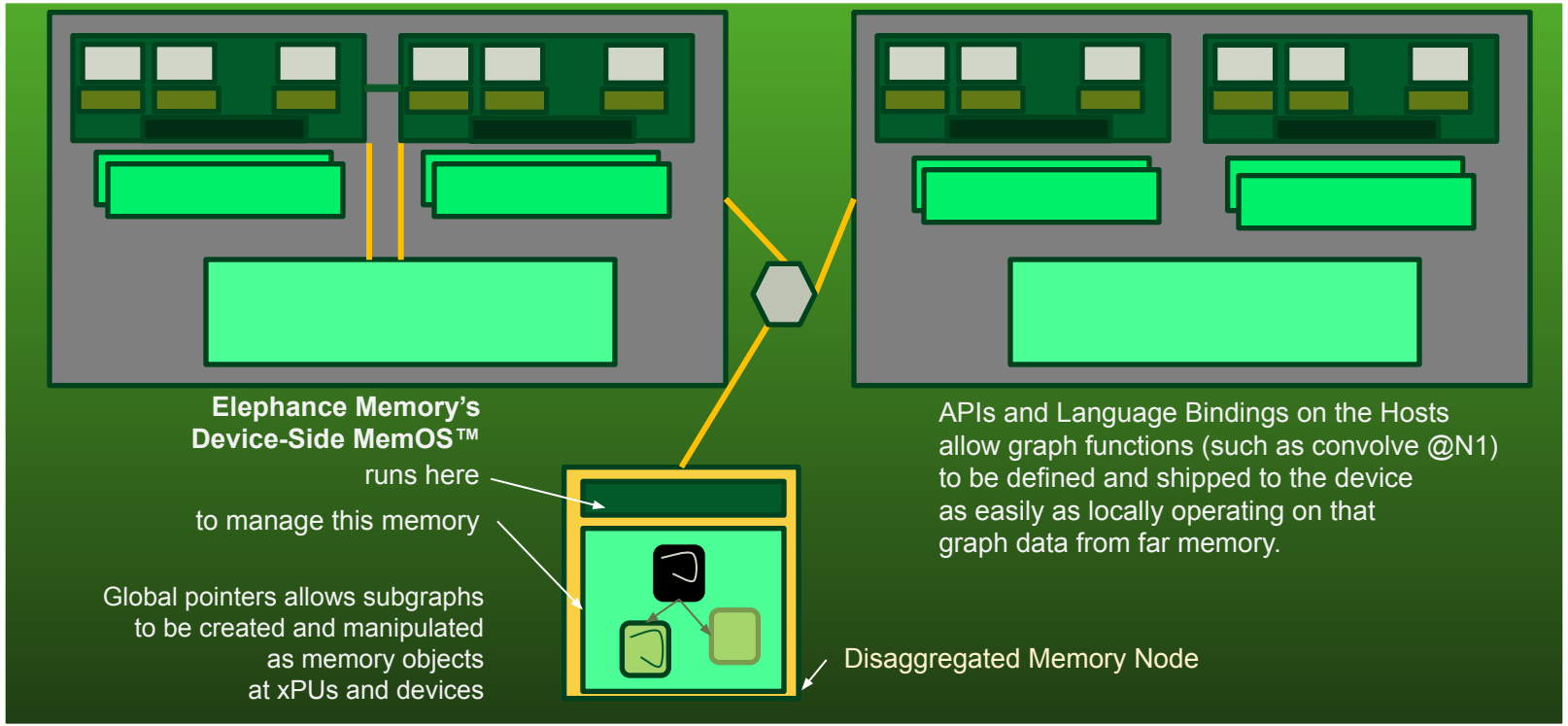
for

New Memory (3DXP, …), **New Interfaces** (**CXL** 2-3, PCIe 5-6)

*Usenix ATC 2020 Best Presentation

^**Pointers in Far Memory**, Miller, Neville-Neil, Mehra, Bittman, *ACM Queue*, June 2023

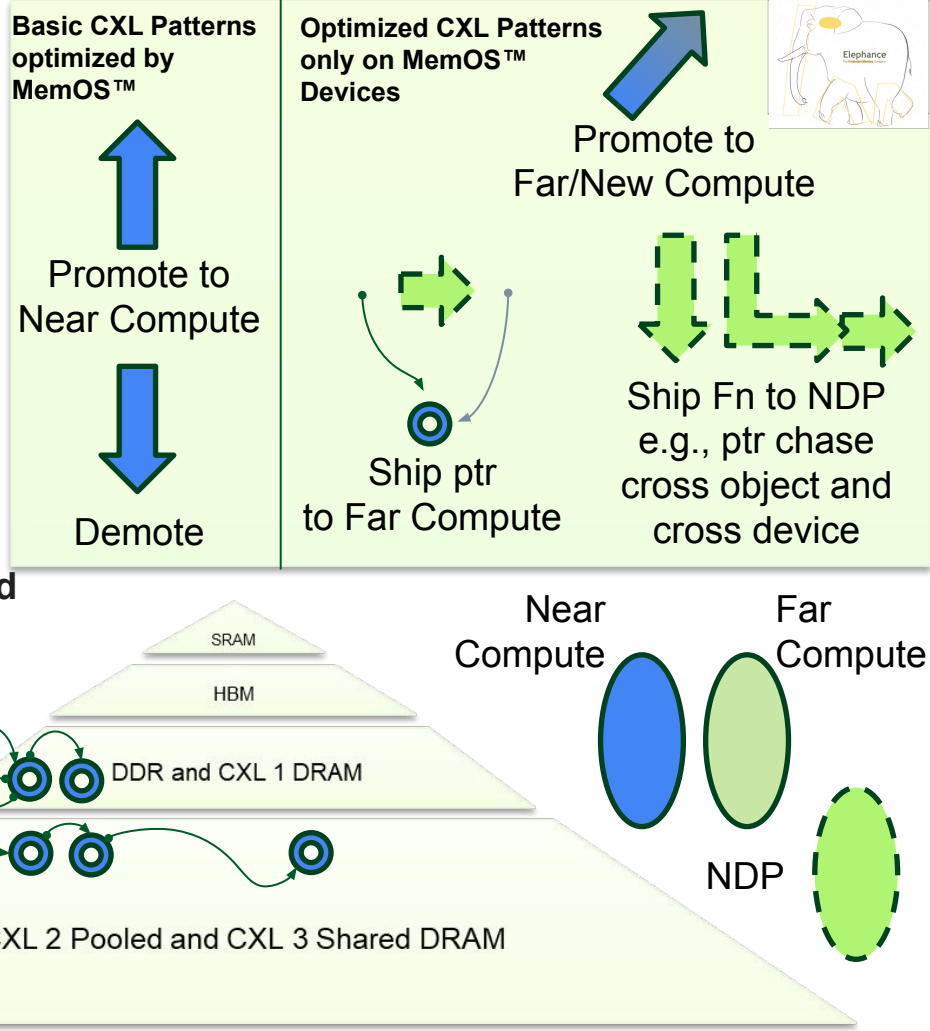
Elephance MemOS™: An “Operating System” for Disaggregated Memory Nodes



*__Taming Memory with Disaggregation__, Pankaj Mehra, IEEE Computer (Sep 2022)

Unique Enablers of Memory PaaS

1. **High Performance Flexible Remoteable Pointers**
   - Native 64b pointers
   - Minimize swizzling overheads
   - Valid everywhere in space and time
2. **Easy to Flex Compute Up and Down**
   - New compute can just declare and use
   - Minimize [de]serialization overheads
   - No need to keep compute fired up
3. **μServices point to data in DMN, not copy it around**
   - Existing microservices benefit right away
   - Minimize/Eliminate unnecessary copying, resulting cache pollution, BW abuse
   - Minimize [de]serialization overheads
4. **Memory Objects**
   - Hyperscaler-friendly proactive, grouped promotion/demotion of objects instead of cacheline-at-a-time data or PTE-at-a-time metadata



Our Core Value Propositions

V1. N-S

Memory Objects make efficient use of caches, TLBs, MMU resources
Challenger: Folio

V2. E-W

Better scale-out with Global Pointers; speedier independent scaling of Memory and Compute

V3. NDP

No need to go back to Host for manipulating graph data (e.g. RAG indices) on device

V6. XPI

Unprecedented parallel compute by CxDs chasing pointers across index shards or offloaded sorts, shuffles, transposes

V5. Open Formats II

Graph and Vector data in the same memory object. Elephance contribution back to Arrow?

V4. Open Formats I

Arrow data format support derives from Rust, allows vector NDP w/o copying
Arrow compute format, also from Rust, allows offload w/o cross-compilation

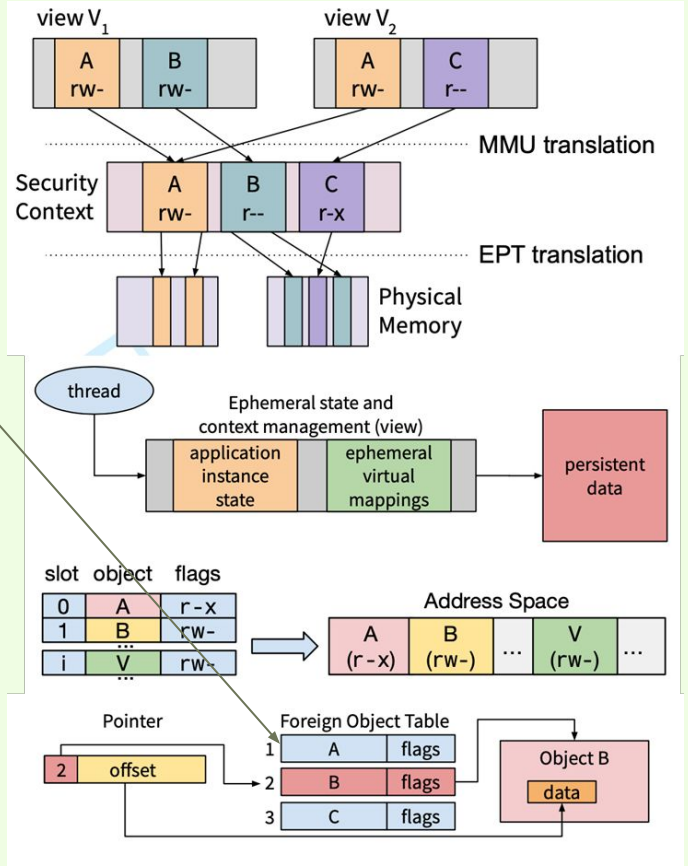
Invariant Pointers for Disaggregated Memory
Invariant under N-S or E-W data movement, meaningful rendezvous with function shipped to device

MemOS under the hood

Translation context disaggregated from process, moved into Memory Objects with 128b ids that are themselves easily embedded inside **64b pointers** that traverse a large address space

Efficient representation and fast manipulation of local pointers (intra-object) in 1s nanoseconds

Scalable persistent representation and fast manipulation of non-local pointers (linked data) using a Foreign Object Table for indirection in every object within 10s nanoseconds



MemOS delivers a Breakthrough in Memory Usage Efficiency

MemOS reduces the cost of operating 1000 EC2 16GB instances by > 40%: **$130/hr → $75/hr**

Memory on Demand

Savings due to renting lower-GB Compute Instances

Service uses <25% of its memory >75% of the time

1,000 16GB EC2 = $130/hr.
~40% for memory = $53/hr.
3/4th memory not rented

$40 Savings

+

Independent Scaling

Savings by flexing between minimum and peak Compute

80:20 [>80% of the time, a query services sees <20% of peak load]

$58 Savings

=

"Pooled" Memory

Replacing "trapped" memory with far memory

− Cost to provide 75% of memory 25% of the time: $17/hr. @$0.006/GB/hr

− Elephance MemoryPaaS: $26/hr (@60% Margin)

Net Savings:
$55/hr; $478K Annual

MemoryPaaS

- A customer operating 1000 EC2 16GB instances saves $87K/yr. in memory rent alone
- Saves $560K/yr. from flexing down
- Total IaaS OpEx can decrease from $130/hr. to $58.5/hr.
- Elephance's other benefits will more than cover the loss of performance due to far memory's latency:
  1. Better RPC [HotOS'21] 2. Pointer-chasing offload for GPUs 3. Improved caching

Savings due to renting lower-GB Compute Instances

Memory on Demand

Service uses <25% of its memory >75% of the time

1,000 16GB EC2 = $130/hr.
1/3rd for memory = $40/hr.
3/4th memory not rented ($30/hr.)

Savings by flexing between minimum and peak Compute

Independent Scaling

80:20 [>80% of the time, a query services sees <20% of peak load]

($64/hr.) from flexing down

"Far" Memory

Customer's cost for memory rented: $22.5/hr.

Elephance MemoryPaaS: ($5.5/hr.)

Cost to provide 75% of memory 25% of the time: $17/hr.
@$0.006/GB/hr

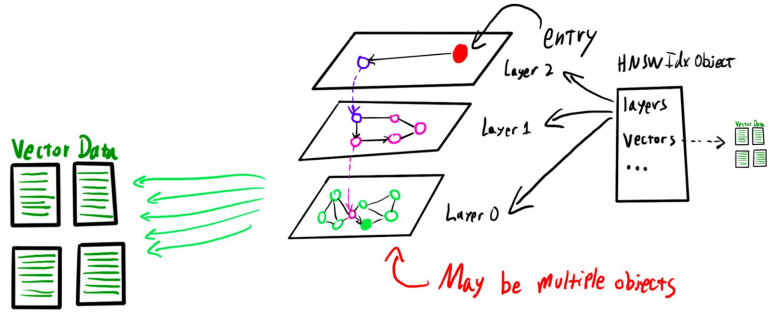
Implied ARR of $20K ($4K net) per TB of memory disaggregated at 25% occupancy SLA

Example: Retrieval Augmented Generation

- AI example that includes lookup process and the use of several graph-based and vector-based data structures that need to loaded from persistent state and used.

- HNSW Index over vector storage with embeddings and data chunks available via lookup table.

- HNSW Nodes can be grouped by object, splitting the graph across objects is easy and enables the workload to move dynamically between hosts



Example: Retrieval Augmented Generation

- MemOS provides quick load / access time to complex data structures: much faster to map an HNSW index in memory objects than to load from persistent files.

	Load
HNSW (File)	27 ± 3 ms
HNSW (MemOS)	0.2 ± 0.1 ms
Vector DB	7 ± 4 ms

- Impact of invariant pointers is minimal due to aggressive caching and compiler optimization

virtual addresses

more flexible

HNSW	baseline	1.0 ± 0.02
	invariant	1.01 ± 0.02
	w/ Arrow	0.78 ± 0.01
K-means	baseline	1.0 ± 0.1
	invariant	1.0 ± 0.1

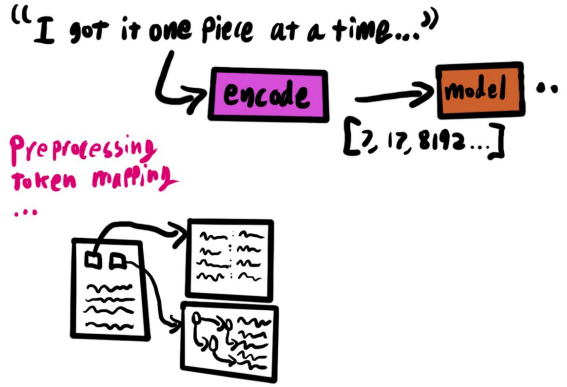
(normalized to baseline)

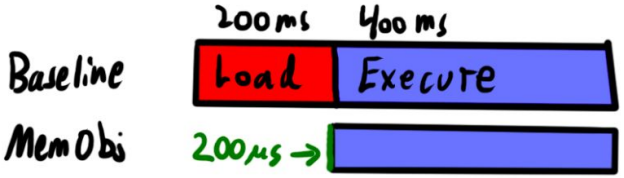
FAISS: 3% ~ 9% overhead

- Load, search, and access times significantly faster than vector databases due to directly accessing data structures in-memory

Example: RAG and Tokenization

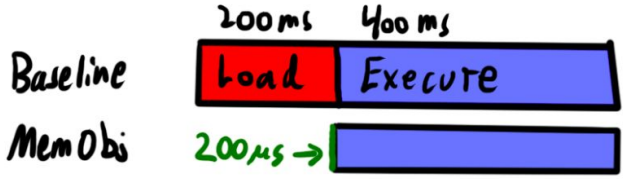
- Tokenization is a vital part of the pipeline, loading takes work.
- Store tokenizer state in memory objects as a collection of primitive, but persistent and invariant, data structures.
- Significantly faster to load and more flexible, enables elasticity and pooling of even more data





Performance improvements: low-level and high-level

- Focus on memory mapping as data access with improved semantics for varying consistency models, re-use of page tables structures, and large mappings.



- Significantly reduced kernel interposition in the data plane.

Fewer Page faults
Less Address Space Management
Reduced copying / wasted work

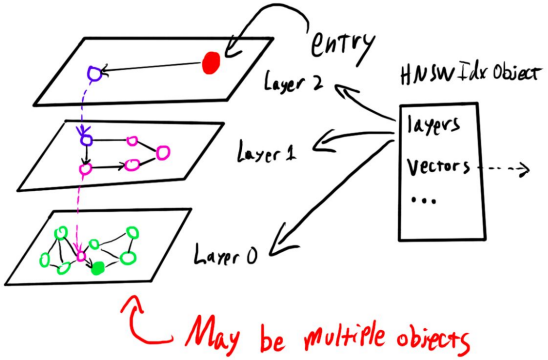
- Simplification of layers, reduce parallel virtual in-memory state that mirrors persistent state.

~10,000 faults → 80
No impact on runtime

Where do objects come from?

- Memory objects can be created directly or via a library or data structure.

  Example: HNSW index builds objects internally as part of node allocation.



- Key: application- and data-structure specific-semantics inform object creation, data layout, and optimization.

  This is abstracted from the programmer, while still enabling direct, low-level control.

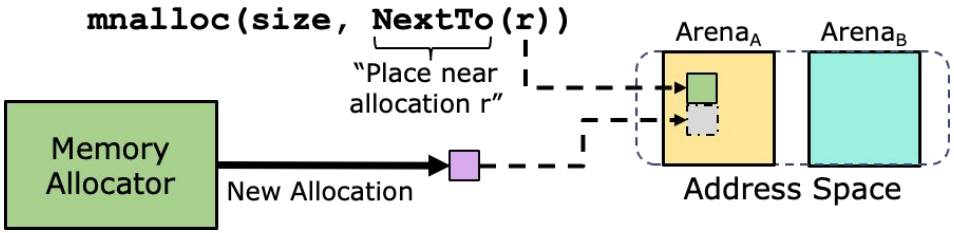
```
struct GraphNode {
    // ...
    vector: ArrowBuffer<UInt32Type>,
}
fn arrow_sum(object: Object<GraphNode>) -> Option<u32> {
    let buf = object.base().vector.buffer();
    let sc = arrow::buffer::ScalarBuffer::<u32>::new(buf, 0, LEN);
    let prim_array = arrow::array::PrimitiveArray::<UInt32Type>::new(sc, None);
    arrow::compute::sum(&prim_array)
}
```

Semantic Data Placement

- Use application semantics to group memory allocations
  - Application semantics are program characteristics observed statically or dynamically
- Memory placement controlled through specialized memory allocator
- Example `Placement: NextTo`
  - Place one allocation near another within the same memory arena (backed by Memory Objects)

“Mnemonic Allocator”

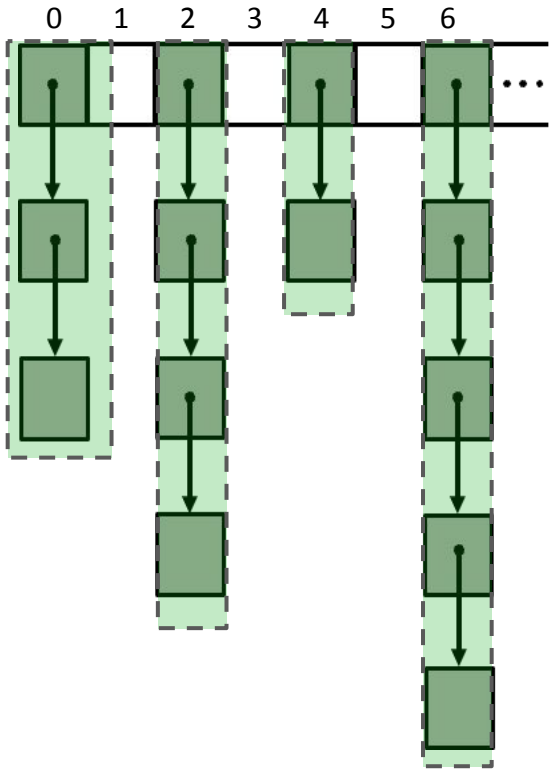
```
mnalloc(size, Placement)
```

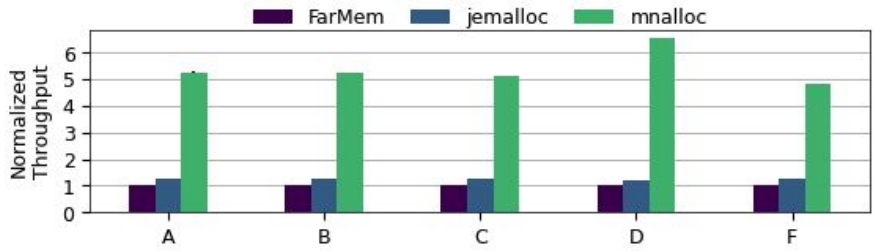


Example: Key-Value Store

Placement with **mnalloc**

NextTo co-locates memory allocations





- Performance Evaluation
  - YCSB: mix of reads, writes, and updates
  - **Improves performance by 3.84-5.34x**
    - Compared to jemalloc
- Full details, and tiered memory study in paper:

Allen Aboytes and Pankaj Mehra. "Improving Performance on Tiered Memory with Semantic Data Placement." *IEEE Computer Architecture Letters* (2025).

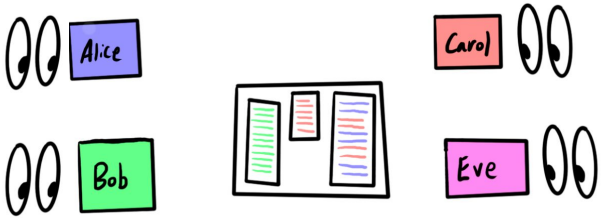
Security and Interoperability

- AI applications ushering in a huge range of new security challenges, but also the same old ones at a new scale.

- MemOS is written in Rust, but not just to do it! Rust enables us to:
  - Take advantage of recent *strict provenance* work to improve safety of data-centric, in-memory programming.
  - Leverage the ownership model to efficiently track live memory, ownership, and shared immutable access.

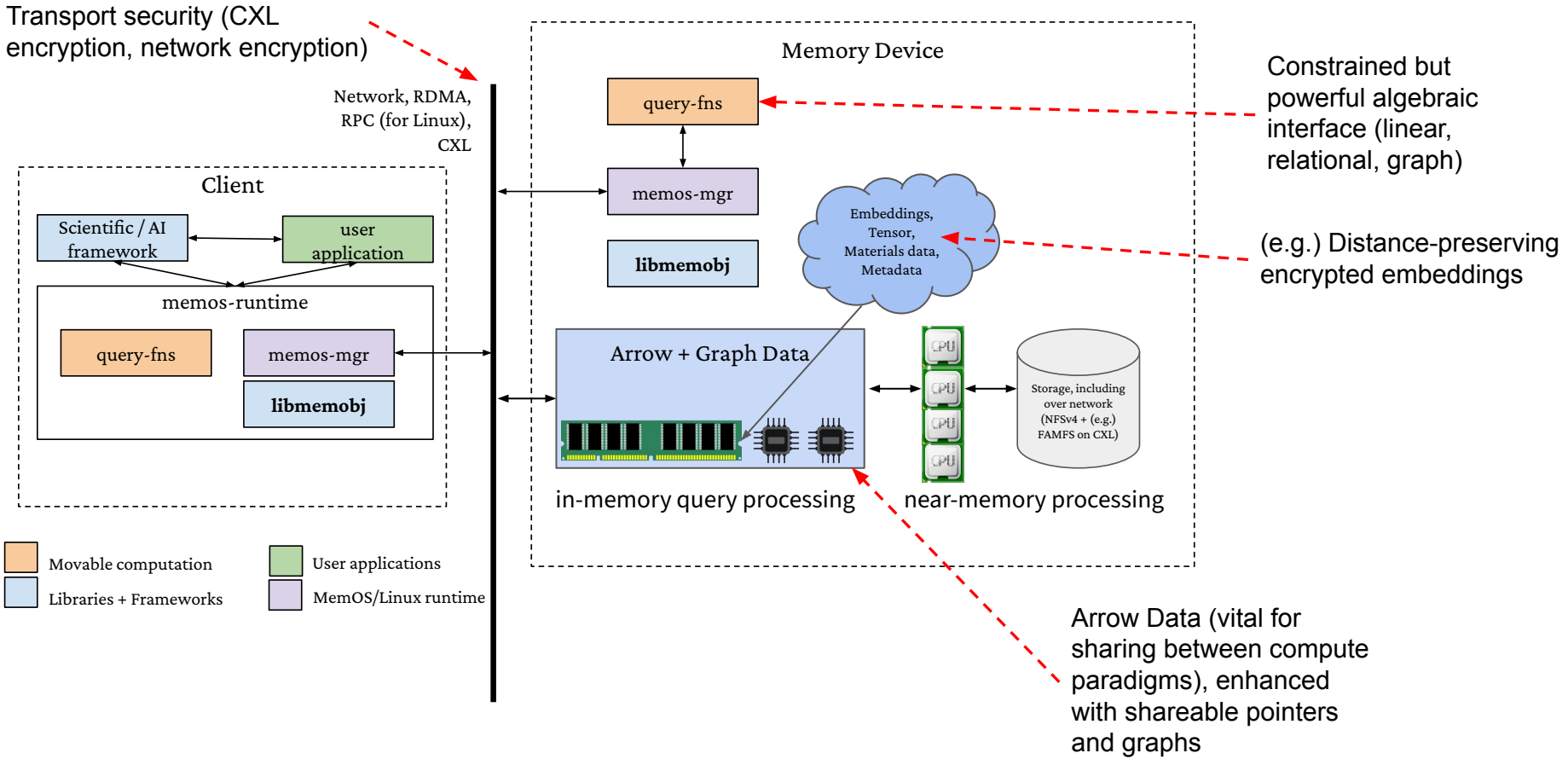
- Interoperability through open formats, contribute back open formats for graph data.

- Data sharing via existing protocols and standards: pNFS, FamFS, standard files



MemOS and Memory Objects

Now delivered in a User Mode Library for Linux (libmemobj)



Roadmap

State of the Art

- Variable length objects increasingly displacing block semantics in storage
- Greater memory efficiency from compaction and compression for AI and General Compute virtualized

Memory Objects

- Elephance style Memory Objects with optimizations for Arrow Tensor data and Graph Data
- Maturing Arrow beyond dense matrix and some relational data to cover all three: Tensor, Graph and Relational
- Graph and Sparse Tensor are both necessary

Parallel Memory Objects

- Parallel Objects that are aware of partition functions such as DFS numbering on graphs and Hilbert Space partitioning on high dimensional tensors
- Hilbert-aware Arrow seen as the ultimate in portability and performance by customers

Vendor Considerations vs Customer Considerations

Customer-specific

Integrate into a unique topology a wide range of accelerators
Secure, Virtualize, Provision
Allow tenants to program infrastructure through highly constrained (usually containerized) interfaces
Offloading to old GPUs not just computation mem or storage

Cross-vendor

Common Framework APIs for

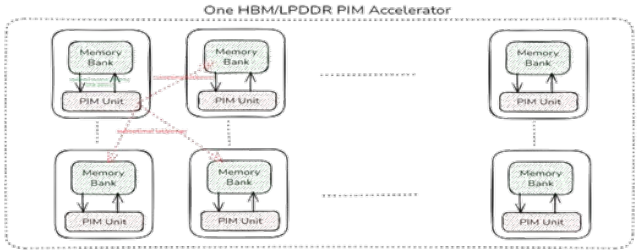
1. Discovery and Config of Comp Mem Devices
2. Function and Data Placement
3. *Cost*-Based Optimization of Offloads

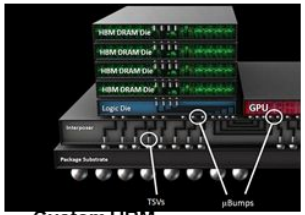
Vendor-specific

Multiple generations and flavors (PNM, **PIM**, **PUD**, AIM) of Comp Mem Devices, Comp Storage Devices, DPUs and even standalone accelerators on fabrics

Wide variety of accelerated operations

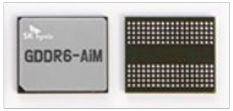
Exposed as Algebraic Kernels that can be *offloaded* by Hosts to operate *locally* on Comp Mem devices for benefits in power, performance and cost





Custom HBM

CXL Computational Memory (CMS)



Processing-in-Memory (AiM)

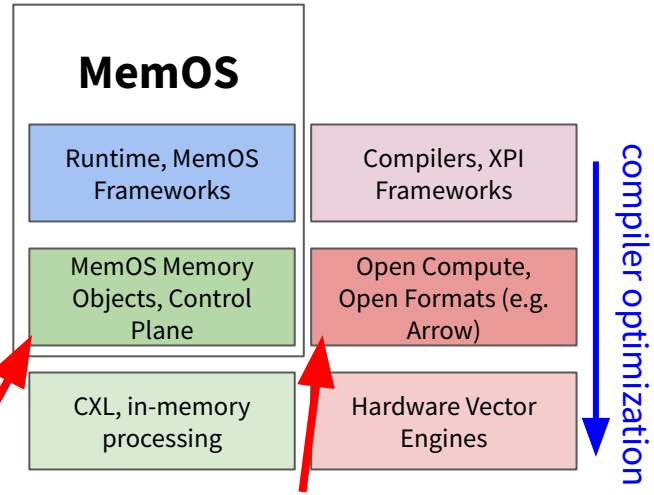
Elephance MemOS

Parallelism
at all Layers

enable domain-specific parallelism optimizations

parallel: data sharing, cluster-level parallel (offload)

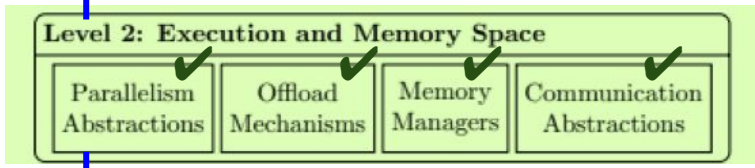
parallel: vector engines, memory sharing



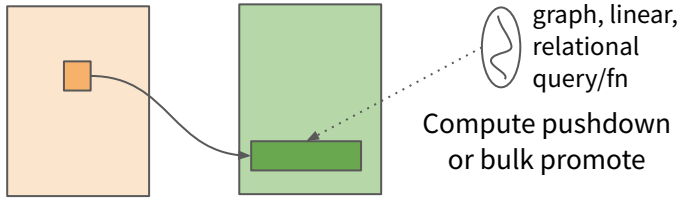
zero ld/st overhead, zero copy, zero serialization

targeted optimization, portability, heterogeneity

Expose parallelism to layer 3



Exploit parallelism from layer 1



less control plane traffic, less memory tracking, batched memory movement, smart caching + prefetching

1,000x improvement in AI model load time due to fluid movement and invariant data (Arrow Arrays + Invariant Graphs)

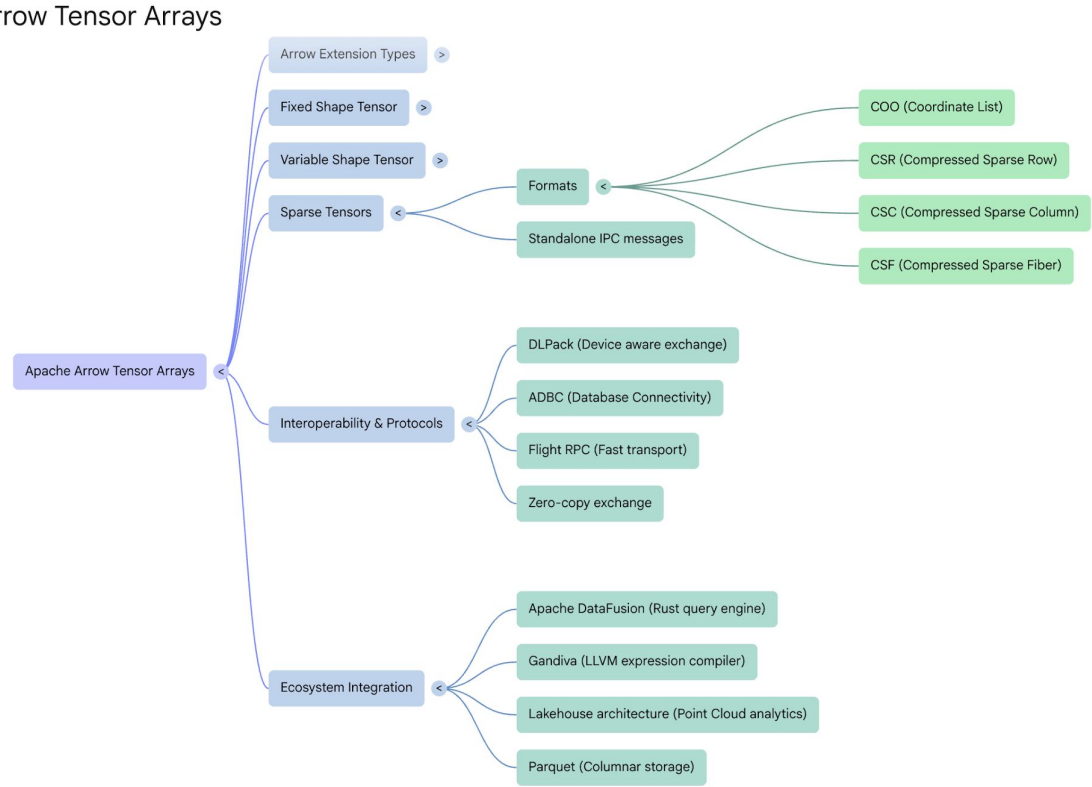
contains an *invariant pointer*

Direct, zero-copy arrow integration

```
struct GraphNode {
    // ...
    vector: ArrowBuffer<UInt32Type>,
}
fn arrow_sum(object: Object<GraphNode>) -> Option<u32> {
    let buf = object.base().vector.buffer();
    let sc = arrow::buffer::ScalarBuffer::<u32>::new(buf, 0, LEN);
    let prim_array = arrow::array::PrimitiveArray::<UInt32Type>::new(sc, None);
    arrow::compute::sum(&prim_array)
}
```

Apache Arrow Ecosystem Rapidly Maturing; Gaps remain

- Arrow language-agnostic columnar layout enables **zero-copy data exchange** between programs & across system boundaries
- Recent Progress atop Arrow Extn Formats
  - CSF exploits Trie plus CSR for high-d data
  - DLpack brings device-aware data interchange to TF, JAX, pyTorch, Numpy, CuPy
    - Oddly, JAX seen as *target* of tensor op offload
  - DataFusion brings Arrow's efficiency to "Agentic AI use of tools" via ADBC, for SQL/DF queries against tables containing tensor data
- Contribution is needed
  - "Compute Functions" stagnated
  - Substrait expression trees stalled
  - Graph formats see no significant progress



Proposed Contributions and Anticipated Benefits

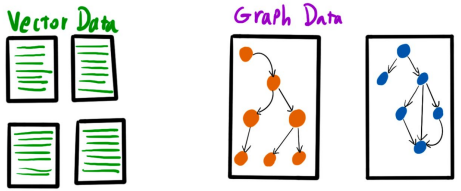
- **Compute Functions**
  - 3 new broad families of "Fixed Functions" in SNIA terminology
  - Each is an Algebra Engine
    - Computes Expression Trees of Ops from its algebra
    - Operates on stored or received open-format Arrow [extn] data
    - And produces or transmits open-format Arrow [extn] data
  - **Compute Functions will thus cover the largest and fastest growing *computable* data types: Linear Algebra, Relational Algebra, Graph Algebra**

- **Expression Trees**
  - Vastly expand Apache Substrait beyond trees of SQL SPJ ops. Add:
    - scalar/vector/tensor operations (both Block/Randomized variants)
    - **Open Format Partitioning Ops**
      - Hilbert Numbering for Linear Algebra as used by HPC
      - Hash partition key generators from Hadoop, Spark, S3, etc.
      - DFS numbering, RSB operator, etc. for graphs
    - Graph Data and Compute
  - **Vs Named Fn mess, unbounded generality of Programmable Fns**

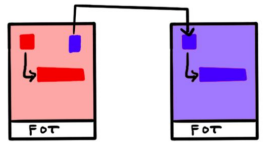
- **Graph Data and Compute**
  - (New) Memory Objects
    - Efficient processing and transfer of ptr-rich data across address spaces within a namespace without onerous cache coherence or SerDe
  - 3 powerful ISO-standard GQL ops in Search-Accumulate family
    - Set/Bag Accumulate, Histogram Accumulate, Heap Accumulate
    - DFS, Random Walk, and Best First
  - To efficiently cover data/fns. of **Knowledge Graphs and Context, GNNs, Recommender Systems, Physical AI & Robotics**

- **Security Benefits: Constrained Interfaces for Intent-Based Security via Rewriting of Algebraic Queries**
- **Performance Benefits: Exploit decades of work on mature cost-based optimization for offload**

Conclusion



- Both applications and emerging hardware demand new programming models for accessing in-memory data



- MemOS enables a data-centric programming model focused on parallelism and developer-friendly flexibility in building and sharing complex, large data with low coordination.

- Looking for investment, partners, and collaboration!

ELEPHANCE MEMORY

elephance memory.com

MemOS