Pacific Northwest
NATIONAL LABORATORY

PNNL-39616

AMAIS - Advanced Memory to Support AI for Science

Report

February 2026

Andres Marquez
Nathan Tallent
Sayan Ghosh
Roberto Gioiosa
Luanzheng Guo
Joshua Suetterlein
Joseph Manzano
Antonino Tumeo
Vito Castellana
Mahantesh Halappanavar
Kevin Barker
Jim Ang

U.S. DEPARTMENT of ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights**. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from
the Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062
www.osti.gov
ph: (865) 576-8401
fox: (865) 576-5728
email: reports@osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
or (703) 605-6000
email: info@ntis.gov
Online ordering: http://www.ntis.gov

PNNL-39616

AMAIS - Advanced Memory to Support AI for Science

Report

February 2026

Andres Marquez
Nathan Tallent
Sayan Ghosh
Roberto Gioiosa
Luanzheng Guo
Joshua Suetterlein
Joseph Manzano
Antonino Tumeo
Vito Castellana
Mahantesh Halappanavar
Kevin Barker
Jim Ang

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830
Pacific Northwest National Laboratory
Richland, Washington 99354

Abstract

The AMAIS (Advanced Memory to Support AI for Science) program addresses the growing imbalance between computation and memory performance in modern scientific and AI workloads—commonly known as the *memory wall*. The project introduces an integrated hardware–software co-design approach that leverages fabric-attached memory, near-memory compute, and advanced memory-centric software tools to enable scalable, high-performance execution of data-intensive applications. Key innovations include the Memory Archipelago architecture, the RRAPID software suite for accelerator synthesis and analysis, CXL-based platforms such as Two Sisters and Crete, and OCEAN, an open-source emulation environment that enables hyperscale experimentation with emerging memory fabrics. Through algorithm–architecture co-design, novel HLS methodologies, performance analysis frameworks, distributed runtimes, and new compiler technologies, AMAIS demonstrates how large, heterogeneous memory pools can support next-generation AI, HPC, and converged simulation workflows. This work establishes foundational methods and platforms for exploiting disaggregated memory systems at scale and provides early evidence of the transformative potential of CXL-enabled architectures for scientific computing.

Summary

The AMAIS program develops a comprehensive ecosystem of memory-centric technologies designed to overcome memory capacity, bandwidth, and latency constraints in AI and scientific computing. Beginning with investigations into memory augmentation for improved inferencing, AMAIS introduces architectures and tools that shift memory from a passive resource to an active, first-class component of system design.

A central contribution is the **Memory Archipelago**, a distributed memory architecture that integrates CXL-based fabric-attached memory with near-memory compute elements. This enables large-scale, heterogeneous memory pools suitable for simulation–AI convergence, retrieval-augmented generation, and long-context LLM inference. The hardware roadmap includes physical prototypes such as **Two Sisters**, **Crete**, and the upcoming **Abaco** system, alongside **OCEAN**, a scalable emulation platform enabling multi-host CXL 3.0 experimentation before hardware is widely available.

The **RRAPID Software & Tools Suite** provides core software foundations, featuring:

Active Memory: HLS methodologies and near-memory accelerator generation.

Performance Analysis (PA): Tools like MemGaze, MemFriend, and HOME for memory-centric profiling and modeling.

Kernels: Benchmarks, workload patterns, and ML/HPC case studies adapted for FAM.

Scale-Up: CXL software stacks including the MemForge allocator, CXL emulation frameworks, and system software for advanced memory hierarchies.

Scale-Out: Distributed runtimes (ARTS), compilers (CARTS), and synchronization mechanisms for multi-host shared memory.

Data Privacy & Security: New GPU memory, cache, and interconnect side-channel analyses and mitigations.

Theory: Scalable performance modeling using isoefficiency metrics and multi-tier pebble game frameworks for disaggregated memory.

Together, these components form a unified strategy for designing, analyzing, and deploying memory-augmented systems capable of supporting massive models, complex scientific workflows, and high-throughput data analytics. AMAIS demonstrates that CXL-enabled disaggregated memory—combined with co-designed algorithms, compilers, and runtimes—can fundamentally reshape the performance characteristics of next-generation scientific computing platforms.

Acknowledgments

This work was supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 76125: “AMAIS - Advanced Memory to support Artificial Intelligence for Science”. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

Transparency

Microsoft 365 Copilot, version bizchat.20260217.44.1, was utilized in the preparation of this report.

Contents

Abstract ii
Summary iii
Acknowledgments iv
1.0 Introduction 1
1.1 Memory Augmentation for Enhanced Inferencing 1
1.2 Vision 2
1.3 Project Overview 2
2.0 RRAPID Software & Tools Suite 3
2.1 Active Memory 5
2.1.1 Memory Centric Design Automation 5
2.1.2 Synthesis of Dataflow Accelerators 6
2.1.3 Algorithm–Architecture Co-Design for Memory-Efficient Transformer Inference 10
2.1.4 Introducing Memory Buffering Systems in High Level Synthesis 12
2.1.5 Conclusions 14
2.2 Performance Analysis 15
2.2.1 Measurement 16
2.2.2 Analysis 16
2.2.3 Architectural Modeling and Emulation 18
2.3 Kernels 20
2.3.1 Relevant workload patterns 21
2.3.2 Case studies 22
2.3.3 Summary 26
2.4 Scale-Up 27
2.4.1 System Software 27
2.4.2 Compiler 34
2.4.3 Co-Design and System Emulation 36
2.4.4 QEmu-CXL 38
2.5 Scale-Out 40
2.5.1 Overview 40
2.5.2 System Software for CXL-based FAM: Locks and Allocators 40
2.5.3 ARTS Runtime Development and CXL Integration 43
2.5.4 Compiler for ARTS: CARTS 45
2.5.5 Application Case Study: Gradient Boosted Decision Trees (GBDT) 48

2.6 Data Privacy and Security....51
2.6.1 GPU Covert and Side Channel Attacks in Memory....51
2.6.2 GPU Covert and Side Channel Attacks in Caches....52
2.6.3 GPU Covert and Side Channel Attacks in Fabrics....53
2.6.4 Harnessing ML Privacy via Memory Crossbars....54
2.6.5 Examining ML vulnerabilities....54
2.7 Theory....57
2.7.1 Isoefficiency Metric....57
2.7.2 Multi-level Pebble Games for Modeling Disaggregated-Memory Systems....62
3.0 Compute Platforms....69
3.1 Fabric-Attached Memory and CXL: A Paradigm for Scalable Computing....69
3.1.1 Hierarchy Overview....70
3.1.2 Protocol Implications....71
3.2 Physical Platforms....72
3.2.1 AMAIS Two Sisters....72
3.2.2 AMAIS Crete....74
3.3 Virtual Platforms....79
3.3.1 OCEAN: Open-source CXL Emulation at Hyperscale Architecture and Networking....79
4.0 Bibliography....91

Figures

Figure 1 RRAPID Software Suite & Tools....3
Figure 2: Example data flow graph, and corresponding execution traces with sequential dataflow, bufferless pipelined, and pipelined execution. Labels on nodes represent execution latency (control steps).....8
Figure 3: proposed High Level Synthesis flow....9
Figure 4: Throughput Evaluation of Selected Benchmarks.....9
Figure 5: ELiTeFormer attention block during generative inference. Hybrid linear attention and sliding-window sparse attention are executed in parallel for each head....10
Figure 6: Linear projection architecture and the specialized ternary PE. Weights are stored in compressed 8-bit frames encoding multiple ternary values. Each PE replaces conventional multiply-accumulate units with mask-based logic....11

Figure 7: HLS-generated accelerator systems: (a) Typical, and (b) Proposed buffering system design 13
Figure 8: Dot product: Memory channel occupancy with varying buffer sizes and channel latency 13
Figure 9 Software Contract 20
Figure 10 Synchronization Requirements 21
Figure 11 Memory Tradeoffs 21
Figure 12 Abstract Machine Learning Kernel 22
Figure 13 FAM STREAM Benchmark 23
Figure 14 FAM vs. Local Memory Kernels comparison 23
Figure 15 FAM MPI communication pattern 24
Figure 16 OSU MPI Benchmarks 25
Figure 17 FAM vs. Local Memory MPI LULESH Benchmark 25
Figure 18 Scale Out Overview 41
Figure 19 Shared Memory Invocation Snippet 41
Figure 20 Pearson Lock Performance 42
Figure 21 ARTS -- Abstract Runtime System 44
Figure 22 CARTS High-Level Flow 46
Figure 23 Isoefficiency -- perfect (blue), 10% super-efficient (orange) and 10% sub-efficient (green) 58
Figure 24 Sparse general matrix-matrix multiplication (SpGEMM) using the combinatorial approach 59
Figure 25 Isoefficiency for building the structure and values of C o traditional 2-scoket memory system 60
Figure 26 Isoefficiency calculated for computing the structure of C using the DRackSim simulator for different network bandwidths. 61
Figure 27 Multi-Level memory hierarchy and data movement costs in FAM. Assuming similar costs moving Blue to Red and Blue to Orange, smaller costs Orange to Red. Dissimilar costs Read vs Write. 65
Figure 28 Register/Memory Hierarchy and its Protocols 69
Figure 29 Two Sisters Machine 72
Figure 30 Crete Machine: ‘Big Crete’ Configuration 74
Figure 31 Crete Machine: 'Split Crete' Configuration 74
Figure 32 OCEAN Design Architecture 82
Figure 33 Two-host GROMACS on real CXL hardware vs. OCEAN. 88

1.0 Introduction

The **AMAIS – Advanced Memory for AI in Science** initiative began by addressing a critical challenge in AI applications: the **memory wall**, which limits performance and scalability for Machine Learning (ML) workloads. The initial goal was to broaden the range of ML algorithms and extend their applicability across diverse scientific domains.

To overcome this barrier, we introduced the concept of the **Memory Archipelago**—a co-design testbed for hardware and software stacks. This architecture is built on large, distributed memory pools, or "memory islands," that provide an additional memory tier and support **near-memory computing**. These islands can interact autonomously over a high-speed switch fabric without host intervention, enabling flexible and scalable memory solutions.

Building on this foundation, we developed **RRAPID** (Rapid Research, Agnostic Performance, Immediate Deployment), a suite of software technologies designed to complement Memory Archipelago architectures and accelerate research workflows.

In parallel, the hardware roadmap leverages **Compute Express Link (CXL)**, a high-speed CPU-device interconnect based on PCIe 5.x/6.x, enabling coherent memory access and unified memory for accelerators. Using these capabilities, we have built physical testbeds featuring large, disaggregated memory subsystems with embedded near-memory compute. Demonstrator systems such as **Two Sisters** and **Crete (Aegean architecture)** are paving the way for scalable architectures that unify near-memory computing with shared memory resources.

While these early architectures focused on research and design space exploration, the next-generation system—**Abaco (Bahamas architecture)**—now under contract, will deliver commercial-grade maturity. It will offer greater memory capacity, GPU integration, and support for massive ML model inference. These enhancements will enable **in-core memory computations** for large-scale converged applications and improve energy efficiency for scientific workloads. Complementing these physical testbeds, we are introducing **OCEAN**, an open-source emulation platform that accelerates software development and fosters collaboration within the broader research community.

1.1 Memory Augmentation for Enhanced Inferencing

AMAIS technologies are designed to **improve AI inferencing through Memory Augmentation**. By introducing an additional shared memory tier, these systems significantly expand the memory footprint available to hosts, increase overall memory bandwidth, and provide a substrate for distributed memory sharing across nodes. This augmentation enables ML frameworks to rethink **context engineering** as a foundational approach to memory management, allowing for richer and more comprehensive contexts that lead to improved inferencing outcomes.

This capability is particularly critical for **scientific computing**, where reasoning tools such as reflection, planning, and chain-of-thought play a key role in interpreting AI-driven insights. To support these workflows, AMAIS emphasizes **fast, semantically organized access to large**

information repositories, which is essential for scientifically accurate ML. This includes enabling **retrieval-augmented generation (RAG)**, managing large embedding databases and caches for rapid similarity and ranking analysis, and servicing multiple attention heads in large language models. Additionally, AMAIS provides infrastructure for **prompt engineering** and efficient handling of extensive knowledge repositories, ensuring that ML systems can deliver precise, context-aware results at scale.

1.2 Vision

Through close co-design among national laboratories, industry, and academia, this project aims to **redefine memory architectures for next-generation scientific computing with a strong emphasis on AI**—bridging the gap between simulation and machine learning at scale. By addressing the memory wall and enabling heterogeneous, high-performance systems, AMAIS lays the foundation for transformative advances in scientific discovery.

1.3 Project Overview

Given the close co-design collaboration with contractor Micron on the AMAIS project, the project has been architected into two distinct phases: a proof-of-concept (PoC) pre-hardware deployment phase (pre-PoC) and a post-delivery/post-acceptance phase (post-PoC). During the pre-PoC phase, the laboratory community concentrated on the development and integration of multiple software stack capabilities targeting AMAIS-class systems, utilizing the AMAIS demonstrator platform, Two Sisters, as a foundational testbed. As the project advances toward the final stages of Crete acceptance testing, AMAIS will transition into the post-PoC research phase. This subsequent phase will encompass the evaluation and refinement of application components referenced in Section 1.1 across OCEAN, Crete, and the forthcoming Abaco system -- projected for delivery end fiscal year 2026 -- through the conclusion of fiscal year 2027. The remainder of this report provides a comprehensive account of the AMAIS software stack achievements realized during the pre-PoC phase.

2.0 RRAPID Software & Tools Suite

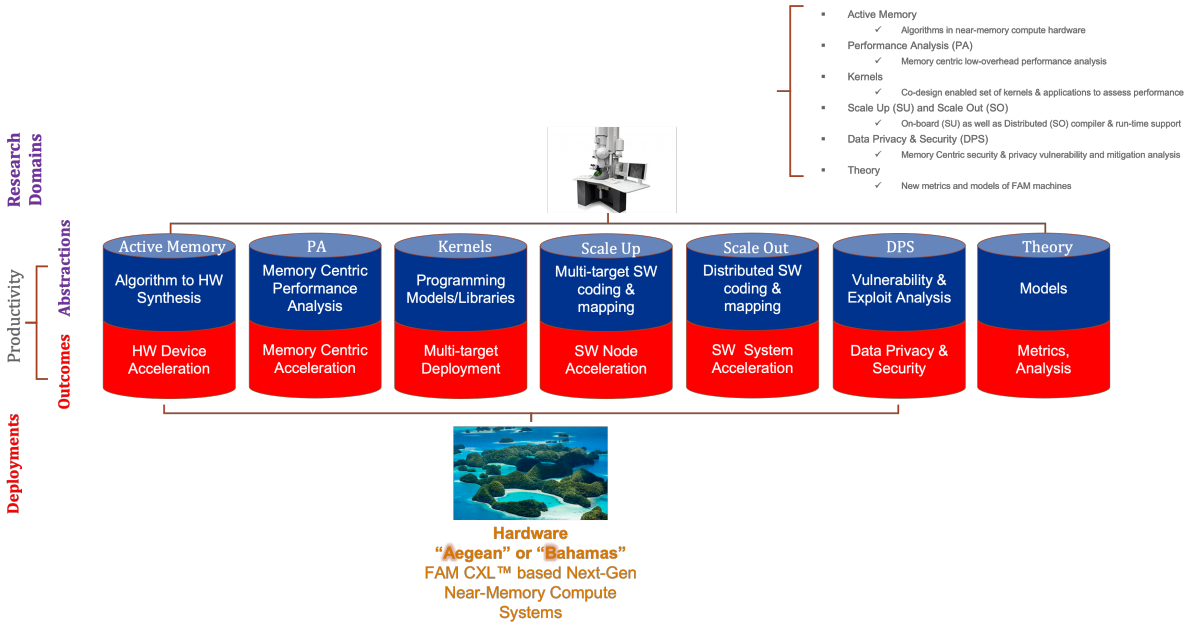


Figure 1 RRAPID Software Suite & Tools

The **RRAPID Software & Tools Suite** (Rapid Research, Agnostic Performance, Immediate Deployment) comprises seven thrust areas, each designed to deliver critical capabilities that underpin the success of AMAIS:

Active Memory Thrust

This thrust focuses on opportunities for Near-Memory and Distributed-Memory computing to minimize memory access latency, exploit high memory bandwidth, and reduce energy consumption otherwise wasted on data movement. Algorithms well-suited for deployment on programmable logic are prioritized, such as distributed locking for critical sections and similarity checks for vector databases.

Performance Analysis (PA) Thrust

PA adopts a memory-centric perspective, elevating memory to a first-class entity in performance analysis—contrasting with traditional actor-centric approaches. Existing memory analysis tools are often cumbersome and underutilized, limiting their effectiveness. PA aims to strike an optimal balance between analytical utility and overhead, enabling practical and efficient memory-centric performance evaluation.

Kernels Thrust

This thrust develops a comprehensive suite of kernels to benchmark and exercise AMAIS systems. It also extends PNNL's existing frameworks, such as Global Arrays, to AMAIS platforms and contributes to community-wide standardization efforts for MPI.

Scale-Up Thrust
Scale-Up investigates domain-specific compilation and execution strategies, leveraging PNNL's COMET and MCL frameworks. It also introduces new Rust-based memory allocators optimized for agnostic platform deployment. Additionally, this thrust drives the development of the OCEAN emulator platform, which complements AMAIS hardware by expanding accessibility and accelerating development productivity.

Scale-Out Thrust
Building on PNNL's expertise in dataflow architectures, compilers, and runtime systems, Scale-Out integrates CARTS (an OpenMP-based distributed compiler) with the ARTS runtime system for AMAIS machines. This thrust carefully orchestrates thread and data-block granularity, communication, and synchronization between compiler and runtime layers to enhance performance and energy efficiency.

Data Privacy & Security (DPS) Thrust
DPS elevates cybersecurity as a core design principle within the AMAIS portfolio. It addresses vulnerabilities introduced by accelerator-enabled memory, including covert channels, side channels, and row hammer attacks. Broader memory security concerns—such as transient attacks, CXL protocol integrity, and noise mitigation strategies for Memory Array Member Inference Attacks and differential privacy—are also explored.

Theory Thrust
This thrust develops abstract models and metrics for reasoning about AMAIS performance and efficiency. Composite metrics incorporating weak and strong scaling dimensions are considered suitable for this research. Furthermore, models originally devised for compiler register allocation and memory placement are extended to accommodate AMAIS-class architectures.

2.1 Active Memory

Modern scientific discovery increasingly relies on artificial intelligence (AI) and data-driven workflows that must process massive datasets efficiently. In many cases, performance is no longer limited by computation, but by the cost of moving data between memory and compute engines. This task addresses that bottleneck by developing new High-Level Synthesis (HLS) methodologies to design near-memory accelerators—custom hardware placed close to memory—within emerging Fabric Attached Memory (FAM) architectures.

Our objective is to establish a complete design flow from high-level application code to a deployed near-memory accelerator integrated within a Fabric Attached Memory system. This flow enables developers to rapidly generate specialized accelerators tailored to memory-intensive workloads. These include machine learning models operating on large parameter sets, graph and data analytics algorithms with irregular access patterns, and scientific AI pipelines that combine simulation, analytics, and learning. Rather than treating memory as passive storage, our approach co-designs computation and memory placement to reduce data movement, improve effective bandwidth utilization, and increase energy efficiency.

The research thrust advances three tightly integrated directions. First, we develop novel HLS methodologies and reusable hardware templates that automatically adapt accelerator structure to memory topology and workload characteristics. Second, we demonstrate acceleration strategies for AI-for-Science applications, focusing on irregular and data-centric algorithms. Third, we establish practical integration and deployment strategies for embedding these accelerators within Fabric Attached Memory systems, enabling scalable and composable near-memory computing.

The novelty of this work lies in unifying HLS methodology, memory-centric architecture design, and real scientific AI workloads into a cohesive and reproducible framework. The value added is a systematic approach that lowers the barrier to designing high-performance, memory-aware accelerators—bridging advanced hardware concepts with practical AI-driven scientific discovery.

2.1.1 Memory Centric Design Automation

High-Level Synthesis (HLS) raises the abstraction level of hardware design by enabling developers to describe accelerators in C/C++ while maintaining competitive performance. This approach has been highly effective for compute-dominated kernels—such as dense linear algebra and digital signal processing—where arithmetic throughput is the primary constraint and memory accesses are regular and predictable.

However, many emerging AI-for-Science workloads—including graph analytics, sparse machine learning models, particle simulations, and data assimilation pipelines—are dominated by irregular and latency-sensitive memory accesses. In these applications, performance depends less on raw compute capability and more on efficiently managing data movement and tolerating memory latency.

Conventional HLS-generated accelerators are typically organized around a centralized control structure known as a Finite State Machine (FSM). In practice, this means the accelerator executes a predefined sequence of steps: request data, wait for it to return, perform computation, then proceed to the next operation. While this control-driven model is effective for structured compute kernels, it limits memory-level parallelism and tightly couples computation progress to memory response timing.

This limitation becomes particularly important in CXL-based Fabric Attached Memory (FAM) systems. Unlike HBM, CXL FAM does not primarily provide ultra-high bandwidth; rather, it enables scalable, composable, and disaggregated memory capacity across the fabric. Access latency is higher than local DRAM, and bandwidth must be shared across links and devices. However, these systems support many concurrent memory transactions and large aggregate memory pools. To use them effectively, accelerators must issue and manage multiple outstanding requests and tolerate variable response times.

An FSM-centric accelerator often underutilizes these capabilities by issuing memory requests in a tightly sequenced manner. As a result, the performance bottleneck shifts from computation to memory orchestration, especially for irregular workloads deployed over fabric-attached memory tiers.

To address these challenges, we investigate alternative accelerator synthesis methodologies based on data-driven execution. In this model, computation units activate when data becomes available, allowing memory requests and processing to proceed concurrently. We complement this approach with reusable hardware IP templates designed for automated synthesis, incorporating buffering strategies, access aggregation, request parallelization, and latency-tolerant scheduling tailored for near-memory and CXL FAM environments.

The novelty of this approach lies in embedding memory-capacity-aware and latency-tolerant architectural principles directly into the HLS flow. Rather than relying on extensive manual code restructuring or tool-specific directives, our methodology systematically generates accelerators aligned with the characteristics of composable fabric-attached memory systems, enabling scalable and efficient AI-for-Science execution.

2.1.2 Synthesis of Dataflow Accelerators

Most HLS methodologies mainly rely on static execution schedules, translating into finite state machine with datapath architectures. This model is highly effective at leveraging instruction-level parallelism (ILP), but it depends on a centralized control unit to coordinate execution. However, FSMD becomes less efficient when handling concurrent, coarse-grained execution patterns—such as task-level or operator-level parallelism—commonly found in AI workloads like deep neural networks (DNNs). In such cases, the model can grow substantially more complex, potentially leading to increased hardware area usage and reduced performance efficiency. Similar considerations apply to memory intensive workloads. Memory operations on external memory such as DRAM, have indeed variable latency, thus preventing their concurrent scheduling, even in the absence of data dependencies. Dataflow models are an appealing solution

to address these limitations. Briefly, dataflow execution does not assume a static schedule, and it is instead controlled based on activations of operators, based on their dependencies.

The dataflow model inherently exploits parallelism by distributing control and enabling computations to proceed asynchronously across a network of connected operators. However, applying this model at the instruction level may result in significant area overhead, as each individual instruction requires its own activation mechanism.

For this reason, we have proposed a hybrid approach, enabling dataflow execution of bundles of instructions (e.g. operators of an AI model), each implemented as a FSMD architecture.

Starting from a relatively simple formulation, which allows for as-soon-as-possible execution, we have developed refined variants which introduce pipelined execution. Commercial tools exist which enable dataflow pipelined execution, but they are affected by severe limitations. They require specific directives to be injected in the source code, and the specification to comply with specific templates, e.g. Load-Compute-Store (LCS) sequences. In the LCS model, only the Load and Store stages are allowed to access external memory, while the Compute operates on local (streaming) buffers. When applicable (and it is not always the case), this solution is very efficient, but it still requires heavy code refactoring. Our solution is more general, because it does not require the input specification to comply with pre-determined pattern. Even more importantly, LCS approaches are limited by local memory availability, which our design overcome.

Our approach is based on dependencies analysis, which identifies producers and consumers operators. In the simplest formulation, producers notify completion to consumers, thus signaling them that they can start execution. Our refined formulations introduce backward communication, where consumers notify completion to producers, i.e. signaling that they can accept new data and execute again. This effectively enables pipelined execution. We have developed two variants of such formulation, one which require inter stage buffers, to store temporary data, and one which requires bufferless control logic. The bufferless variant, causes producers to stall til consumers complete execution. The other allows producer to execute multiple times, staging produced data into buffers, until these are full. Consumers retrieve input data from these buffers, freeing the corresponding entry once they complete. Figure 2 shows an example dependency graph, and the associated dataflow execution traces with the different formulation variants: sequential dataflow (v1), bufferless pipelined (v2), and pipelined with inter stage buffers (v3).

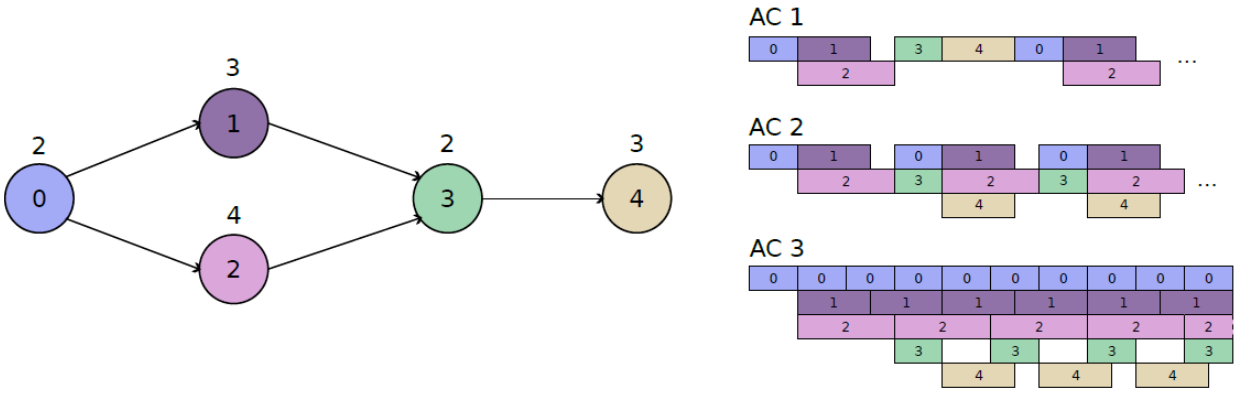


Figure 2: Example data flow graph, and corresponding execution traces with sequential dataflow, bufferless pipelined, and pipelined execution. Labels on nodes represent execution latency (control steps).

In our design, differently from LCS approaches, each operator (or "stage") is free to access external memory. This allows to support applications which process large amounts of data (larger than local memory) but also introduces a challenge: as multiple operators execute concurrently, they might attempt to access memory, using shared memory ports, at the same time. We address this issue by introducing an arbiter, which intercepts memory accesses, avoiding structural conflicts on the memory ports. Such arbiter heuristically assigns higher priority to consumers' memory access requests. Intuitively, pipelined execution greatly enhances throughput, allowing overlapping of multiple stages/operators. Still, being a shared resource, memory could represent a performance mitigating factor: when a memory access is in flight, it does prevent other operators from accessing memory, potentially causing pipe stalls. Reducing memory access latency is therefore crucial to reach optimal throughput and makes near memory acceleration an ideal solution. Alternatively, or better orthogonally, optimized memory subsystem solutions could help achieve superior performance, as discussed later.

As described so far, pipeline execution is only safe when different operators access non-overlapping memory regions. If not, producers could overwrite data not yet consumed, and vice versa. It is possible for specification designers to enforce this, but not convenient. The proposed synthesis flow addresses this issue.

When analyzing program dependencies, our front-end inspects the data flow with respect to memory, capturing information about direction (e.g. loads, stores, in-place updates), and aliasing (e.g. parsing restrict pointer qualifiers to assess non-aliasing). Alias analysis (i.e. assessing non-conflicting memory accesses) is further enhanced by exploiting high level information available in MLIR dialects, core representation of our toolchain frontend.

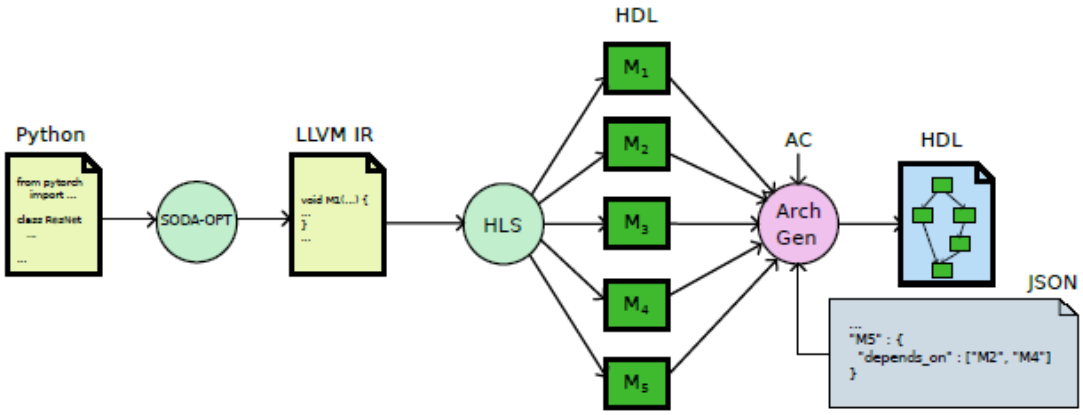


Figure 3: proposed High Level Synthesis flow.

Figure 3 demonstrates a high-level view of the proposed flow. Starting from a high-level specification, SODA-OPT, our optimizing front-end, performs dependency analysis, including memory data flow, generating a dependency graph. When failing in assessing aliasing, it does introduce artificial dependencies among "may-alias" operators (i.e. operators which might access, concurrently, overlapping memory regions). Using conventional HLS techniques, the flow synthesizes each operator as a standalone FSMD accelerator. Finally, the flow composes the final design, with dataflow control logic orchestrating pipelined execution across operators, and introducing the arbiter for accessing external memory.

We have evaluated our approach on a set of diverse computational workloads, operating on streaming data. First, we consider the auto-encoder for Electron Energy Loss Spectroscopy (EELS), an essential tool for processing complex scientific datasets, particularly in material sciences. Next, we evaluate Graph Convolutional Networks (GCN), widely used for graph-based data analysis in machine learning tasks, focusing on node classification and graph learning. Additionally, we consider ResNet, a deep neural network model extensively employed in image recognition pipelines, from edge devices to server-side processing. Finally, we analyze Transformer layers, which are foundational to modern natural language processing (NLP) tasks and multimodal applications, including time series interpretation, language translation, reasoning, and vision-to-text processing.

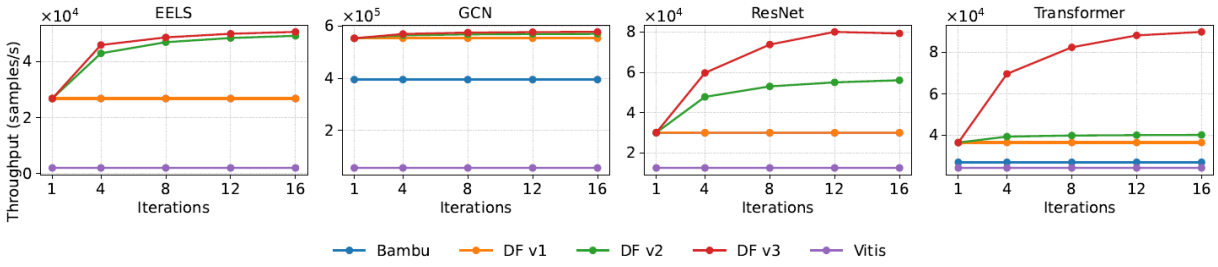


Figure 4: Throughput Evaluation of Selected Benchmarks.

Figure 4 shows the observed throughput of the selected benchmarks, synthesized on a Xilinx xc7z020 device, comparing the different dataflow formulations, against two baselines: Bambu HLS, and Vitis. Bambu HLS is a state-of-the-art HLS tool, Vitis is the commercial synthesis framework from Xilinx, and for the considered specification, they both generate FSMD design. We observe that the first dataflow formulation (DF v1), even in the absence of pipelining, allows to achieve superior results compared to the baselines. This is because it can execute concurrently independent operators, which the baselines' FSMDs instead serialize. The pipelined versions clearly provide even better results. While the introduction of buffers always results in the best performance overall, the improvement with respect to the bufferless formulation varies depending on the dependency structure. I.e. applications which feature regular chains of operators, such as Transformer, provide the most performance gains, since in this case the pipelined architecture with buffer can have the greatest number of operators concurrently executing.

2.1.3 Algorithm–Architecture Co-Design for Memory-Efficient Transformer Inference

Recent large language models (LLMs) are dominated by Transformer blocks whose computational and memory demands grow rapidly with context length. In generative inference, quadratic attention and the linear growth of the key–value (KV) cache creates severe memory pressure, limiting batch size and throughput in long-context scenarios. ELiTeFormer addresses these limitations through a tightly integrated algorithm–architecture co-design that simultaneously restructures the attention mechanism, the feed-forward network (FFN), and the hardware microarchitecture.

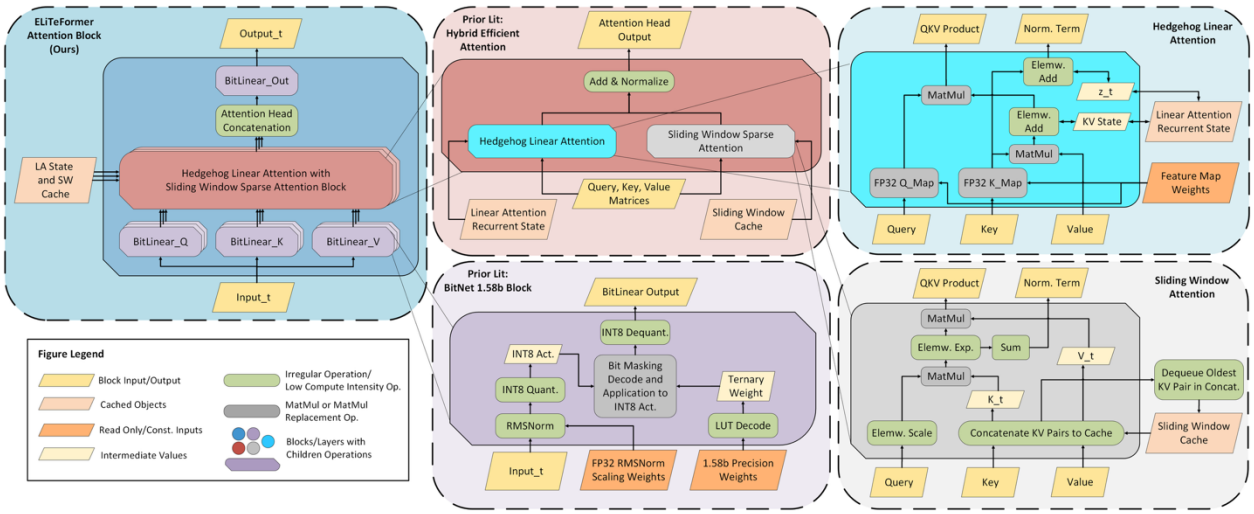


Figure 5: ELiTeFormer attention block during generative inference. Hybrid linear attention and sliding-window sparse attention are executed in parallel for each head.

At the algorithmic level, ELiTeFormer replaces quadratic attention with a hybrid linear attention mechanism combined with sliding-window sparsity, as illustrated in Figure 5: ELiTeFormer

attention block during generative inference. Hybrid linear attention and sliding-window sparse attention are executed in parallel for each head.. Linear attention reformulates the attention computation so that runtime and memory scale linearly with sequence length. During generative inference, this allows attention state to be maintained in a compact recurrent form. As a result, the KV cache footprint is reduced by approximately 12.8× compared to a LLaMA 3 baseline at long context lengths. For memory-centric systems, this reduction directly lowers memory capacity requirements and reduces data movement across memory tiers.

In parallel, ELiTeFormer adopts ternary quantization for linear projections in both attention and FFN blocks. Model weights are constrained to {−1, 0, +1}, achieving approximately 10× weight compression. The corresponding hardware realization is shown in Figure 6. Instead of implementing multiply-accumulate (MAC) units, the design replaces multipliers with mask-based selection and accumulation logic. Because ternary weights only require sign inversion, nullification, or pass-through of input values, DSP blocks are eliminated. Block-level simulations demonstrate up to 9.6× speedup for linear projections and up to 4.4× speedup for attention computations compared to conventional implementations.

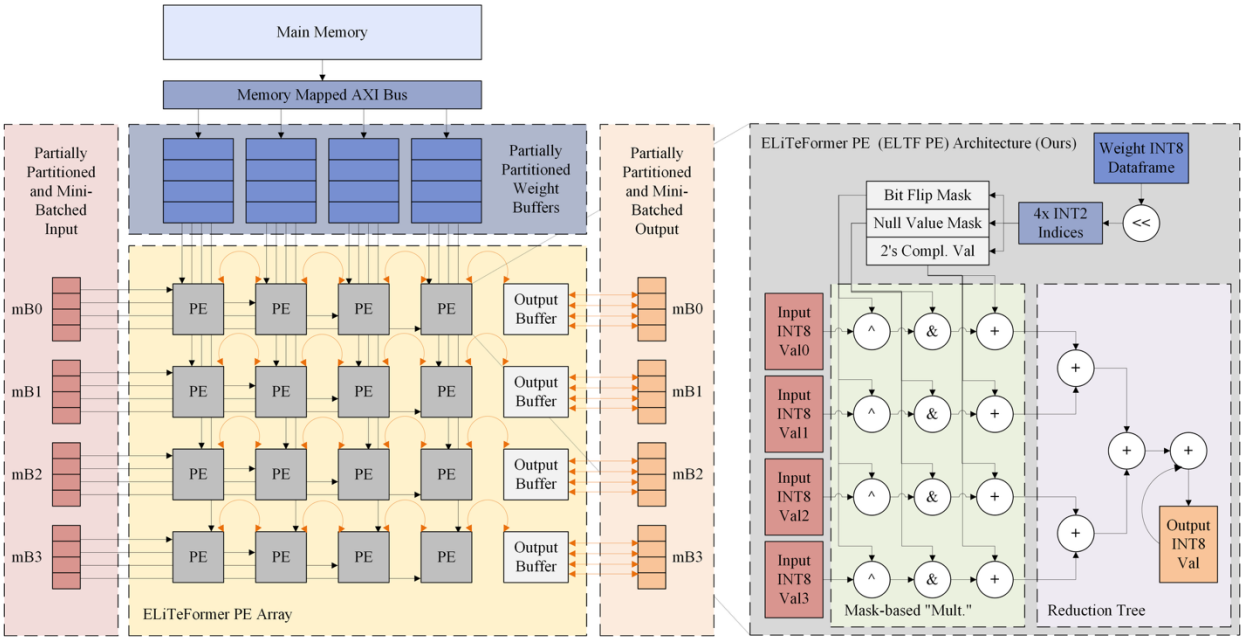


Figure 6: Linear projection architecture and the specialized ternary PE. Weights are stored in compressed 8-bit frames encoding multiple ternary values. Each PE replaces conventional multiply-accumulate units with mask-based logic.

The accelerator was synthesized using HLS flows and deployed on a Xilinx VCK5000 FPGA. End-to-end measurements show up to 3.9× lower latency and 3.2× higher energy efficiency than a similarly scaled LLaMA 3 implementation on an NVIDIA A100 GPU at long context lengths.

These results validate that algorithmic compression and architectural specialization compound to deliver system-level benefits.

The implications for CXL-based FAM systems are direct. CXL FAM primarily enables scalable and composable memory capacity across the fabric, but with higher latency and shared bandwidth relative to local DRAM. In such systems, KV cache growth and large model footprints directly increase fabric traffic and memory-tier pressure. The 12.8× KV cache reduction, enabled by the hybrid attention structure Figure 2, lowers capacity demands and reduces cross-fabric data movement during long-context inference. Similarly, the 10× model compression and multiplier-free datapath in Figure 3 reduce both memory footprint and compute energy, making the design well suited for CXL-attached FPGA accelerators deployed near fabric memory pools.

2.1.4 Introducing Memory Buffering Systems in High Level Synthesis

As mentioned in previous sections, HLS tools effectively optimize compute datapaths using the FSMD model, often adopting a load–compute–store paradigm that assumes timely data availability. In realistic systems, however, accelerators access off-chip DRAM through shared interconnects that introduce significant latency and contention. Even modest increases in memory latency can shift accelerators from compute-bound to memory-bound operation, causing them to stall for most execution cycles and dramatically degrading performance. Moreover, when multiple accelerators share the same memory channel, excessive memory transactions lead to congestion and poor overall system throughput. Existing HLS solutions either require manual code restructuring and pragma annotations to enable burst transfers or rely on fixed-parameter caches that are not tailored to application-specific access patterns. To overcome these challenges, we have investigated a transparent, compiler-assisted buffering system that automatically analyzes memory access direction and patterns (e.g., contiguous versus strided) and inserts parameterized on-chip buffers between the accelerator and DRAM. The prototype system leverages AXI4 burst transactions to amortize memory latency, reduce channel occupancy, and decouple computation from data movement without modifying the accelerator’s original design.

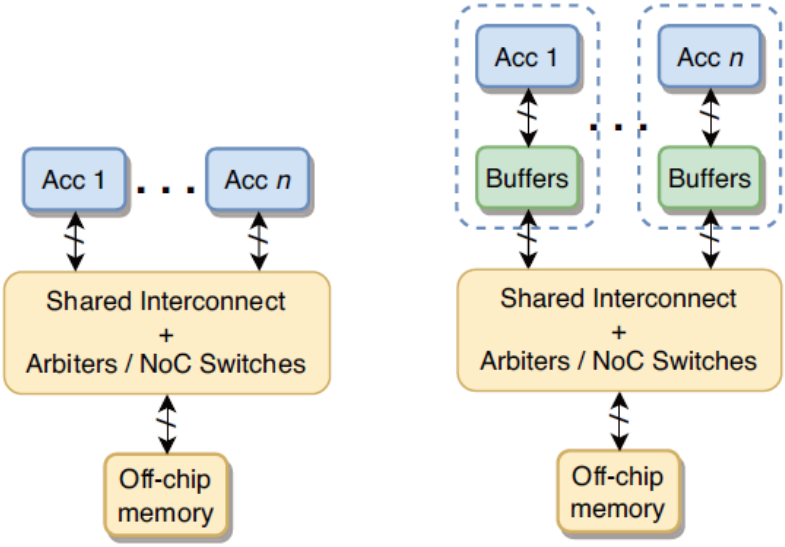


Figure 7: HLS-generated accelerator systems: (a) Typical, and (b) Proposed buffering system design

Figure 7 overviews the proposed design, compared to the conventional HLS-generated systems. Our solution interposes a dedicated buffer, residing in local memory, between the design modules (e.g. operators), and the shared memory. The buffers behave similarly to conventional caches: when a module accesses data residing in the buffer (this is verified analyzing target addresses), reads and writes occur in local memory. When an access targets an address outside the scope of the buffer, this triggers a burst memory transaction. This means that memory operations will always proceed at local memory latency, until there is a "miss" in the buffers. Buffers are individually sized; i.e. different modules could have buffers of different size. A two-step sizing heuristic balances on-chip memory constraints with performance gains, prioritizing strided and read-write accesses and iteratively refining configurations.

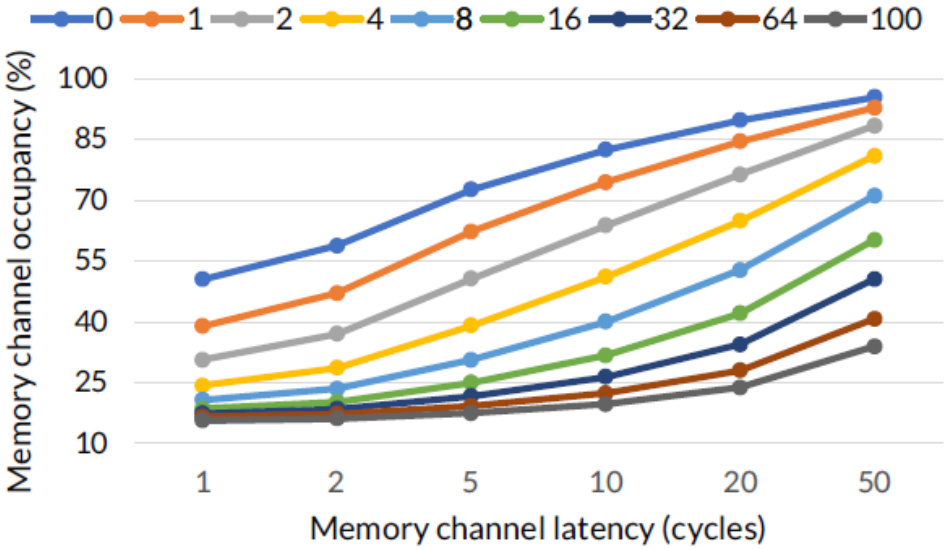


Figure 8: Dot product: Memory channel occupancy with varying buffer sizes and channel latency

Intuitively, bigger buffers could allow to achieve superior results, at the cost of higher on-chip memory requirements. Figure 8 studies the impact of buffers size and memory latency with respect to shared memory channels occupancy (the lower the better) for the dot-product kernel.

We have evaluated our solution on several PolyBench kernels synthesized with Bambu HLS and implemented on a Xilinx VCU128 FPGA. The study analyzes performance speedup, memory channel occupancy, and area overhead under varying buffer sizes and memory channel latencies (up to 50 cycles). Results show that without buffering, accelerators quickly become memory bound as latency increases, while appropriately sized buffers significantly reduce burst transactions and amortize latency. Larger buffers generally improve performance and reduce channel occupancy, especially for strided and read-write access patterns, though with increasing area cost. Using a 1KB on-chip memory constraint, the proposed heuristic-generated buffering system achieves an average 8.8× performance speedup and reduces memory channel occupancy by 53.2% compared to baseline designs. When compared against a 1KB direct-mapped cache implementation, the buffering system delivers 1.6× higher performance on average and 23.2% lower channel occupancy, demonstrating better adaptability to kernel-specific memory access patterns and more efficient use of burst transfers.

It is important to highlight that this solution has advantages in different shared memory configuration scenarios: when operating near memory, the buffering system will benefit from reduced memory latency; otherwise, it will help mitigating the overall impact of high latency memory operations.

2.1.5 Conclusions

This work advances a unified, memory-centric design methodology that rethinks High-Level Synthesis (HLS) in the context of emerging near-memory and Fabric Attached Memory (FAM) systems. By moving beyond conventional FSMD-centric accelerators and integrating dataflow-based execution models, latency-tolerant scheduling, and compiler-driven memory analysis, we demonstrate how accelerator architectures can be systematically aligned with the realities of high-latency, shared, and capacity-oriented memory fabrics. The proposed techniques—including hybrid dataflow–FSMD synthesis, arbiter-managed concurrent memory access, algorithm–architecture co-design for Transformer inference, and automated buffering—collectively reduce memory bottlenecks, improve throughput, and enhance energy efficiency without requiring burdensome manual code restructuring. Experimental results across AI-for-Science workloads, deep learning models, and PolyBench kernels show substantial gains in performance and memory channel utilization, validating the effectiveness of embedding memory awareness directly into the synthesis flow. Overall, this research establishes a scalable and reproducible framework for designing accelerators that treat memory not as a passive resource, but as a first-class architectural dimension—bridging high-level programming abstractions with practical, high-performance deployment in composable memory systems.

2.2 Performance Analysis

The memory wall phenomenon—where advances in processor performance significantly outpace those in memory subsystems—poses a fundamental challenge for contemporary computing systems. In memory-bound applications, memory subsystem behavior dominates performance, yet existing analysis approaches present significant limitations: detailed microarchitectural simulators require days to weeks to simulate modest workloads; hardware performance counters provide only aggregate statistics that obscure temporal and spatial access patterns; and scaled simulation approaches face challenges in capturing certain behaviors that emerge at larger scales. These limitations reflect a processor-centric design philosophy increasingly misaligned with memory-bound workloads where detailed understanding of memory access patterns, cache hierarchy interactions, and contention is critical for effective optimization.

We have developed an integrated framework for memory-centric analysis that enables effective hardware-software co-design. We describe practical trace collection techniques, including hardware-assisted processor tracing with minimal overhead and portable software-based instrumentation with statistical sampling. We present multi-perspective analysis methods that examine memory behavior from temporal, sequential, spatial, and relational viewpoints, revealing distinct optimization opportunities invisible in aggregate metrics. We detail an architectural modeling framework that uses sampled traces with temporal interpolation and confidence-based filtering to evaluate cache and memory configurations. Evaluation on representative benchmarks demonstrates that this framework achieves practical accuracy (L2 cache errors of 2.64%, confidence-filtered L3 errors of 9.92%, bandwidth errors of 7.33%) while providing substantial speedup (26.8×) over cycle-accurate simulation, enabling rapid design space exploration. We demonstrate how this integrated framework enables systematic identification of both hardware optimizations (memory controller tuning, bank partitioning, NUMA configuration) and software optimizations (data layout restructuring, prefetching strategies, memory-aware scheduling). Through this comprehensive treatment of the memory-centric analysis pipeline—from trace collection through architectural modeling to co-design application—we provide researchers and practitioners with practical techniques for addressing memory bottlenecks in contemporary computing systems.

We have developed an integrated framework for memory-centric analysis that brings together trace collection techniques, multi-perspective analysis methods, and architectural modeling approaches.

Trace collection techniques: We describe both hardware-assisted processor tracing with minimal overhead and portable software-based instrumentation with statistical sampling, analyzing trade-offs in overhead, portability, and resolution.

Multi-perspective analysis methods: We present four complementary analytical perspectives—time-centric, sequence-centric, location-centric, and spatial affinity analysis—each revealing distinct optimization opportunities through examination of memory behavior from different viewpoints.

Architectural modeling framework: We detail HOME, an architectural modeling framework, which uses sampled traces with temporal interpolation and confidence-based filtering to evaluate cache and memory configurations, demonstrating practical accuracy while providing substantial speedup over cycle-accurate simulation.

2.2.1 Measurement

To reason about memory systems, it is necessary to understand how application instructions within processors issue memory requests, the temporal intensity and distribution of these requests over time, the memory addresses and regions they access, and the data lanes and memory devices they utilize. Capturing this information requires collecting metadata beyond just memory addresses: instruction addresses enable identification of which code segments generate specific access patterns, timestamps facilitate temporal analysis of memory pressure and contention, and novel analyses reveal the spatial distribution of accesses across the memory hierarchy and interconnect. This metadata-rich approach provides the foundation for comprehensive memory-centric analysis.

To achieve this level of understanding, we leverage fine-grained traces of memory accesses. Because exhaustive tracing is often impractical due to prohibitive storage and performance overheads, we focus on sampled access traces. A sampled trace consists of a series of sequences and gaps. Each sequence is a stream of back-to-back memory accesses that fills a fixed-length buffer. A gap represents the sampling period, measured in terms of either time or memory access counts. Each entry in the sequence captures information about the memory address, instruction address, timestamp, and issuing core. The time and space overhead of sampled traces can be controlled by adjusting the buffer size and sampling period parameters.

There are two primary methods for collecting these traces: a portable approach that uses dynamic instruction instrumentation and an extremely lightweight version that leverages processor tracing hardware. Each method presents distinct trade-offs between overhead, portability, and resolution.

A key result is MemGaze, memory analysis tool that combines high-resolution trace analysis and low overhead measurement, both with respect to time and space. Prior methods incurred time slowdowns of O (100×). In contrast, MemGaze collects high-resolution word-level memory access traces with typical overhead of 10-35%. The highest resolution supported is back-to-back sequences. MemGaze leverages emerging Processor Tracing support to collect data. It achieves low overhead in space and time by leveraging sampling and various methods of hardware support for collecting traces. MemGaze provides several post-mortem trace processing methods, including multi-resolution analysis for locations vs. operations; accesses vs. spatio-temporal reuse, and reuse (distance, rate, volume) vs. access patterns.

2.2.2 Analysis

Memory access patterns can reveal performance bottlenecks and optimization opportunities when viewed from complementary angles. We have developed several different analytical perspectives for examining traces of sampled access sequences. In particular, we analyze traces from the perspectives of: (a) memory access intervals fully ordered by time, enabling

identification of temporal phases with poor locality; (b) hot sequences of memory accesses, both within time and grouped by sequence similarity, revealing recurring access patterns; (c) memory locations and allocated regions within the address space, identifying objects with poor spatial locality; and (d) related pairs of memory locations, whether spatially contiguous or not, quantifying spatial-temporal affinity for prefetching and caching optimization. Each analytical perspective provides distinct insights into memory behavior. Time-centric analysis reveals when performance issues occur, sequence-centric analysis identifies what code patterns cause issues, location-centric analysis pinpoints which data structures exhibit poor locality, and affinity analysis quantifies how memory locations relate to each other in the access stream.

2.2.2.1 Time-centric

The time-centric analysis organizes the trace as an execution interval tree, or a hierarchy of time intervals where each interval is a fully ordered sequences of memory accesses.

2.2.2.2 Time-Sequence-Centric Analysis

It is often extremely informative to analyze memory locality with respect to hot sequences—frequently recurring patterns of memory accesses examined both within specific time windows and grouped across time by their similarity. A hot sequence represents a set of accesses that recurs frequently throughout execution, indicating a common execution path through the program that generates a characteristic memory access signature.

Importantly, the sequences we identify are approximate rather than exact matches. That is, a sequence should recur and maintain its general ordering pattern, but only with respect to its dominant access pattern, allowing for minor variations in execution. We permit gaps and sudden starts/stops in the pattern: gaps account for conditional branches within hot sequences that may skip certain accesses depending on runtime conditions, while sudden starts/stops account for the fact that samples may capture only a prefix or suffix of a hot sequence when sampling windows do not align perfectly with the sequence boundaries.

2.2.2.3 Location-Centric Analysis

Data locality analysis can be categorized into two fundamental approaches: actor-centric and location-centric. In actor-centric analysis, memory accesses and locality are analyzed from the perspective of an executor, such as individual threads or processors, tracking what each actor accesses and when. In contrast, location-centric analysis adopts the perspective of the memory system itself, which may be shared by many actors, and studies the resulting locality characteristics from the viewpoint of memory locations. This perspective is particularly valuable for understanding cache behavior and memory hierarchy utilization, as it directly models how the memory system experiences access patterns.

2.2.2.4 Spatial Affinity

To provide a comprehensive view of spatial-temporal locality beyond single-location reuse analysis, we introduce spatial-temporal affinity metrics that quantify locality relationships between pairs of memory locations. These metrics address the question: “What location j is likely to be accessed within the vicinity of an access to location i?”” This pair-wise perspective is

particularly valuable for understanding hardware prefetching opportunities (which benefit from predictable access sequences) and cache sharing behavior (which benefits from nearby locations being accessed close together in time). A key result is the MemFriend tool. MemFriend's multi-resolution analysis identifies significant memory segments and simultaneously prunes the analysis space such that time and space complexity is modest. MemFriend creates signatures, selectable at 3D, 2D, and 1D resolutions, that provide novel insights and enable predictive reasoning about application performance. The results aid data layout optimizations, and data placement decisions.

2.2.3 Architectural Modeling and Emulation

We have developed trace-based architectural emulation and performance evaluation.

2.2.3.1 DRAM modeling

To support rapid, actionable contention analysis, we introduce HOME (Hierarchy-Oriented Memory Evaluation), a lightweight framework that processes sparse PTWRITE traces by temporally interpolating sampled events through timestamp scaling to restore time-varying memory access pressure. HOME then applies configurable cache and memory models to evaluate the impact of memory slowdown across the memory hierarchy. It applies confidence conditions to identify and flag under-sampled runs that may produce unreliable results.

A significant challenge in trace-based architectural analysis is the impact of random sample drops, which can affect both accuracy and hardware prediction performance. As established in prior work the accuracy of input samples is critical for hardware predictions, as models must accurately capture the previous states of the memory and cache hierarchies to produce reliable projections. To mitigate the impact of sample drops, HOME introduces two complementary techniques: temporal interpolation to restore access pressure and confidence conditions to flag potentially unreliable analysis regions.

HOME can instantiate different levels of the memory hierarchy, with modeling approaches tailored to the optimization objectives of each level. Cache modeling typically employs event-driven simulation, where the primary focus is on optimizing cache miss rates and hit patterns. In contrast, memory system modeling uses time-driven approaches, as the major optimization goals are bandwidth utilization and access latencies. This separation of modeling approaches reflects the fundamental difference in how these hierarchy levels impact performance: caches are capacity-limited structures where hit/miss behavior dominates, while memory systems are bandwidth-limited resources where queueing and timing effects dominate.

2.2.3.2 Recommending combined DRAM and CXL allocation policies

Compute Express Link (CXL) enables memory capacity and bandwidth expansion beyond DRAM limits by offering cache-coherent load/store instructions on byte-addressable PCIe-attached media. However, CXL's higher latency necessitates strategic data placement between DRAM and CXL tiers. Current options include burdensome manual allocation for CXL in DAX mode or the promising alternative of utilizing a memory-interleaving policy for NUMA's weighted allocator. The optimal policy (peak throughput), however, varies with access patterns,

DRAM vs. CXL bandwidths, and working set size. Currently, identifying that policy requires an exhaustive search through time- and resource- intensive experiments or repeated simulations.

We developed MemProphet, a framework to rapidly analyze, identify, and reason about CXL memory performance bottlenecks. MemProphet leverages HOME-CXL, a fast, CXL-aware emulator based on a "run-once, analyze-many" methodology. Using a single replay of a single sampled execution trace, MemProphet emulates performance of diverse DRAM-CXL configurations and identifies optimal allocation strategies, avoiding repeated application execution. For NUMA interleaving, it determines the ideal DRAM-CXL policy. For DAX mode, MemProphet identifies code most appropriate for CXL allocations by mapping performance-critical metrics to their corresponding hot sequences and static code regions. MemProphet has both low online and offline costs, achieving up to 20x speedup over state-of-the-art simulators and emulators, while providing accurate, actionable insights for effective CXL memory management configured either in NUMA or DAX mode.

2.2.3.3 Reasoning about power

Managing the energy consumption and power efficiency of parallel applications is a significant issue. As emerging applications continue to push against the memory wall of modern machines, the growing imbalance between compute and data movement creates new opportunities to intelligently tune CPU power consumption. Unfortunately, existing frequency and voltage scaling techniques do not adequately capture fine-grained changes in memory access behavior, rendering the compute/data access imbalance invisible to the components of the system that could capitalize on it, thus leaving potential power savings on the table.

We have developed MemPower, a flexible, model-based approach to exposing compute/data movement imbalance that characterizes the fine-grained memory behavior of parallel workloads. This characterization then informs our automated software framework which can statically instrument the application binary with model-determined voltage/frequency transitions that balance fine-grained changes in memory access behavior with the costs of hardware transitions. Using MemPower, we demonstrate a reduction in EDP of 6% to 42% on a range of HPC benchmarks with minimal impact on execution time when compared to the standard OS/hardware-managed power control mechanism.

2.3 Kernels

The goals of the AMAIS MLS thrust are to 1) develop/utilize high-level distributed-memory programming abstractions for *rapid prototyping* of data-intensive machine learning applications, 2) devise *efficient heuristics and methods* to optimize data movement for application scenarios (especially sparse and irregular applications that are memory-access driven) considering latest advances in high-capacity standards-based memory infrastructure, and 3) perform *quantitative evaluation* to assess systems performance at different scales. For these cases, we consider the abstract software/hardware modules below to aid the co-design of next generation of hardware architectures. Currently, the difference in best case aggregate bandwidths between scale-up and scale-out is about 10-100x considering proprietary chip-to-chip interconnects or integrated optics, at least another order of magnitude in bandwidth (with reduced latencies due to zero copy) can be realized through contemporary disaggregated memory using open CXL standards.

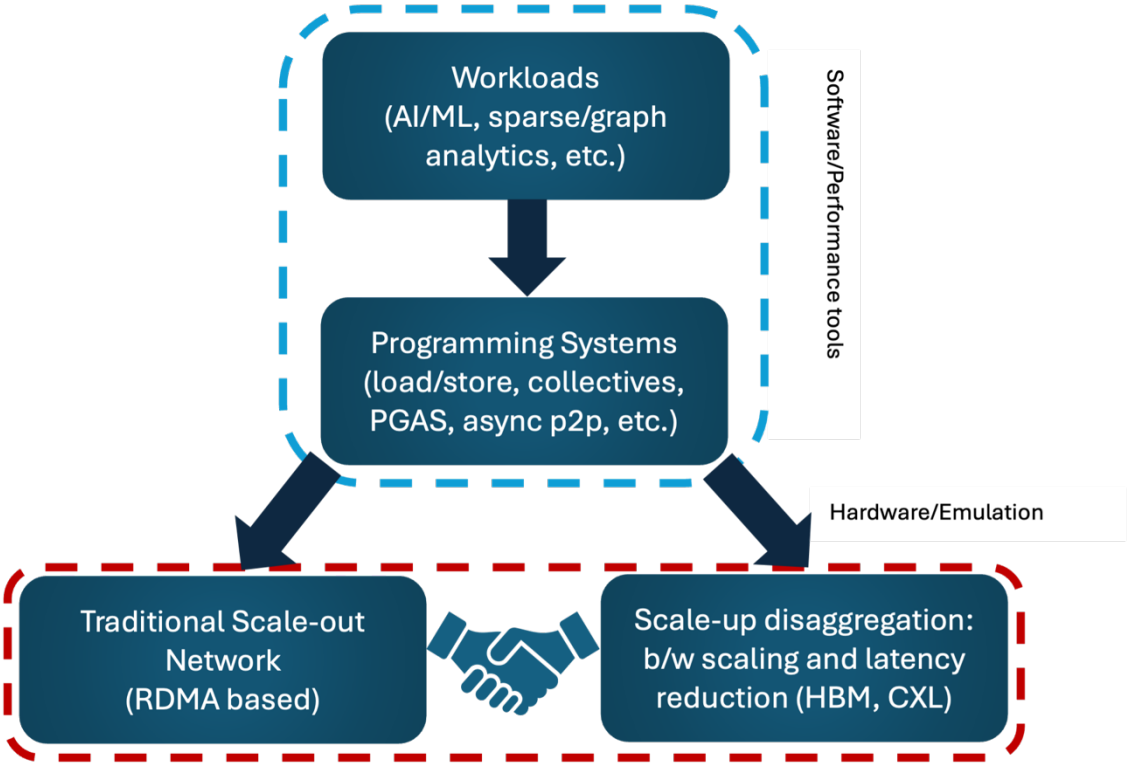


Figure 9 Software Contract

Contemporary thread and process-based programming models can support memory disaggregation with two changes to the source code considering direct-access persistence models (DAX) provisioning of external memory: 1) allocate explicitly on CXL-based Fabric Attached Memory (FAM) using mmap-like memory mapping interfaces, and 2) handle possible cache coherency by explicitly invalidating local caches prior to accessing data on FAM (this is relevant

when multiple hosts are connected to CXL memory pool via switch, standard memory consistency models only apply to single-host domains).

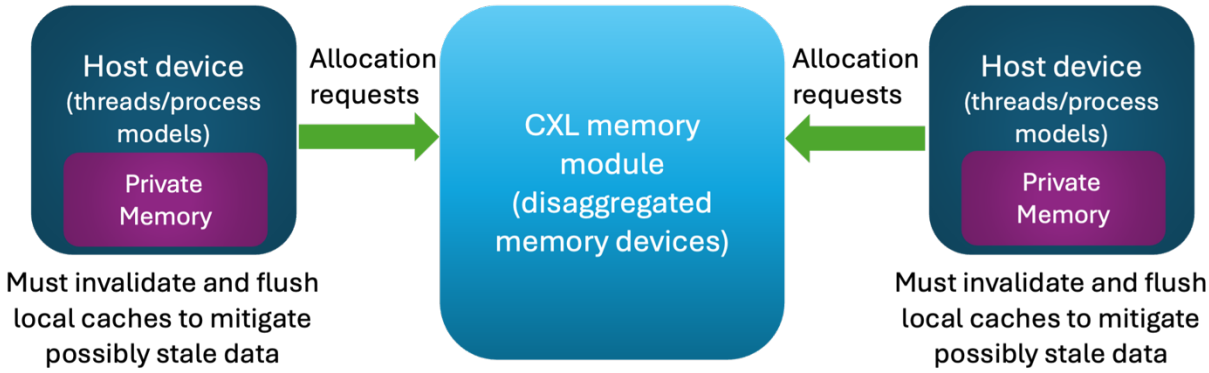


Figure 10 Synchronization Requirements

2.3.1 Relevant workload patterns

Our workloads are governed by the abstract hardware/emulation model above. We consider the application workloads to span multiple nodes, using threads, processes or mixed models. For instance, threads within a node and process-based model across nodes for data movement (via one-sided RDMA-based approach or via RPC/active messages), with options to access private and disaggregated memory. For software, we have been developing lightweight interfaces to build different application scenarios and proxies, and such interfaces currently depends on standardized models such as MPI or OpenMP. These interfaces can generate application traces and profiles, which can be used in simulators and detailed performance assessments. Our simple on-node performance model (no network communication) is shown below, complex instructions increase front-end stalls, whereas long-latency memory operations increase backend-stalls, throttling the front-end. The premise here is that the application scenarios considered would be able to drive the bandwidth available on node (using all the available processing cores across NUMA regions) and balancing stalls and memory accesses (optimizing data reuse, large caches are better), until saturation (at that point, distributed-memory scaling would be justified).

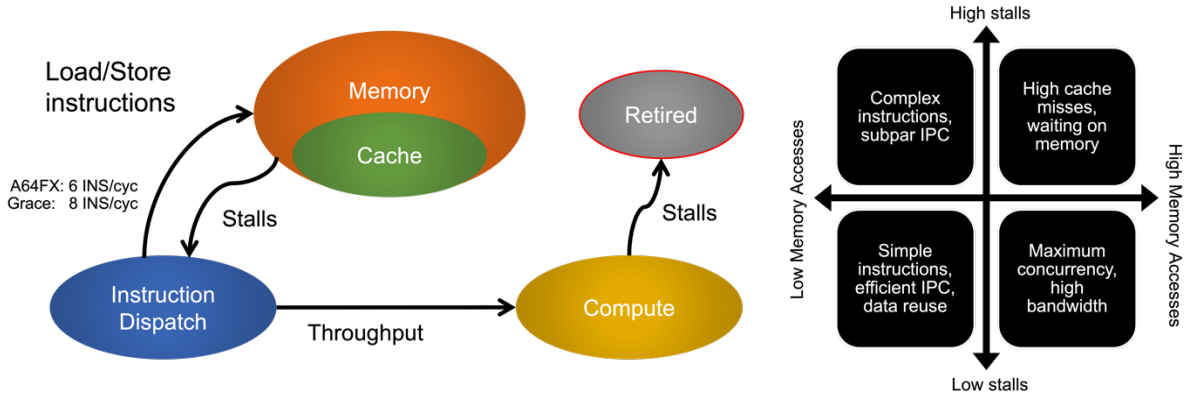


Figure 11 Memory Tradeoffs

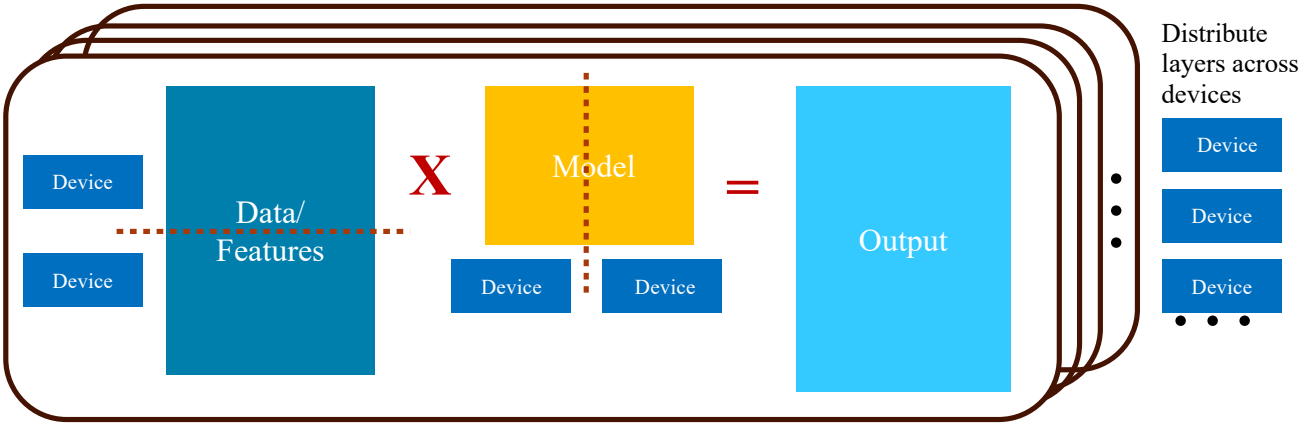


Figure 12 Abstract Machine Learning Kernel

To express application patterns, consider the following abstract parallel machine learning kernel, implying distribution of data and model across devices (hybrid parallelism).

Such hybrid parallelism is the mainstay of modern machine learning, since just subdividing data across one dimension may not be sufficient for sustainable strong scalability (for instance, Meta's recent Llama model training incorporates 4D parallelism). Deploying such models assumes optimal collective performance and sufficient memory capacity and bandwidth. For instance, contemporary Large Language Model (LLM) inference workloads increasingly rely on access to fast and high-capacity memory for managing Key-Value caches for the associated tokens. On the other hand, for irregular memory-access driven cases (e.g., graph analytics), distribution exacerbates communication, and thus having capacity is beneficial in reducing memory access related stalls (and avoiding communication altogether by exploiting disaggregated scale-up).

2.3.2 Case studies

Preliminary evaluations on two-sisters (two-socket Sapphire Rapids with 8GB of global shared memory with CXL switch emulator) for single-host multithreaded CPU-memory bandwidth assessments via STREAM benchmark (higher bandwidth is better) indicate subpar bandwidth (among different FAM allocations, reading from FAM yields the best bandwidth), possibly due to limited PCIe bandwidth relative to DDR.

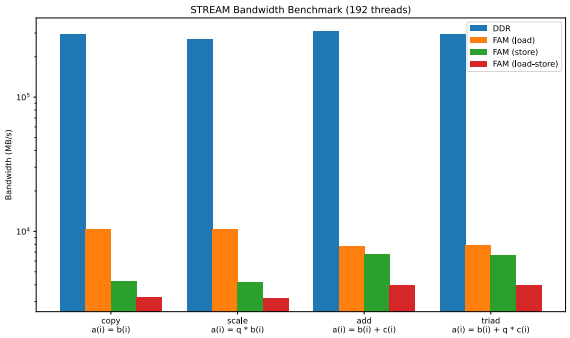


Figure 13 FAM STREAM Benchmark

However, the negative effects of FAM on single host is nuanced when considering real application workloads. Since graph applications demonstrate irregular memory accesses and relatively low arithmetic operations, they are ideal workloads for CXL memory. The GAP Benchmark Suite (GAPBS includes optimized parallel (using OpenMP) implementations of common graph algorithms supporting a wide variety of scientific applications; we use variants of five graph kernels in our analysis: Breadth First Search (BFS), PageRank (PR), Connected Components (CC), Betweenness Centrality (BC) and Triangle Counting (TC). Since graph workloads frequently accesses the original graph (i.e., vertices, edge weights, degrees, etc.), and seldom updates it, we allocate the graph data structure on FAM. This requires a one-time graph construction cost (as demonstrated through STREAM performance, stores are twice more expensive than loads), which is about twice more expensive than local-DDR. As shown below, performances between local-DDR and FAM are competitive; since this is a single-host workload like STREAM, coherency is hardware-managed. GAPBS kernels are traversal based and iterative, frequently accessing noncontiguous locations of the input graph, unlike STREAM, whose access pattern is contiguous.

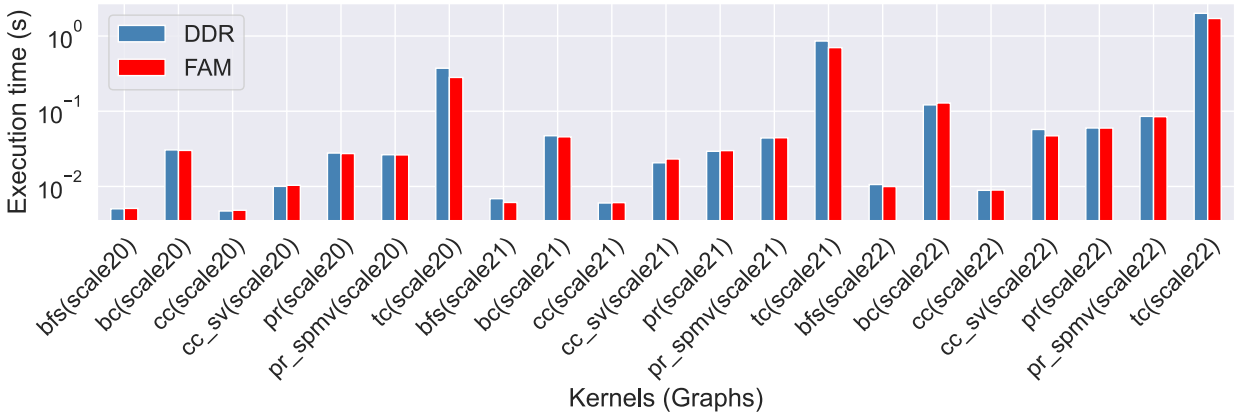


Figure 14 FAM vs. Local Memory Kernels comparison

The scope of performance improvements with FAM expands when considering multiple hosts; in such a situation, it would be possible to exploit “zero copy” across processes, where explicit communication or data movement over RDMA can be avoided with appropriate modifications in the application code. In practice, most distributed-memory applications exploit both host DDR

and FAM leading to several combinations, as shown below. These combinations will have distinct cache invalidation requirements depending on specific load/store patterns. Typically, considering arbitrary process pairs across hosts, we can represent the data movements or sharing potential (through "zero copy"), which can act as a guide for general application porting.

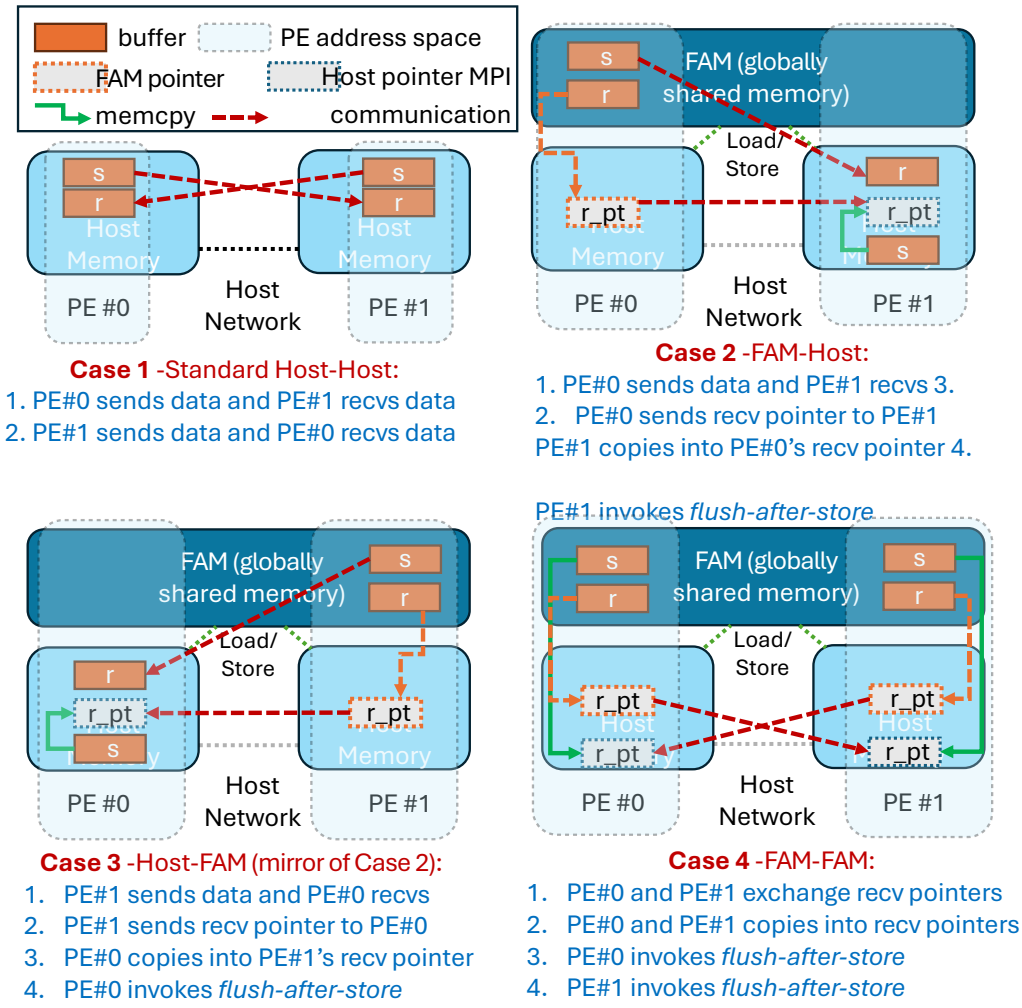


Figure 15 FAM MPI communication pattern

Below, multi-host assessments using the OSU microbenchmark (latency and bi-directional bandwidth) between two processes on two hosts shows linear scalability due to direct accesses, as opposed to purely MPI-based code path (traversing the RDMA NIC). While ideal FAM-FAM latencies/bandwidth is best, software managed coherence (cache flush) reduces the performance.

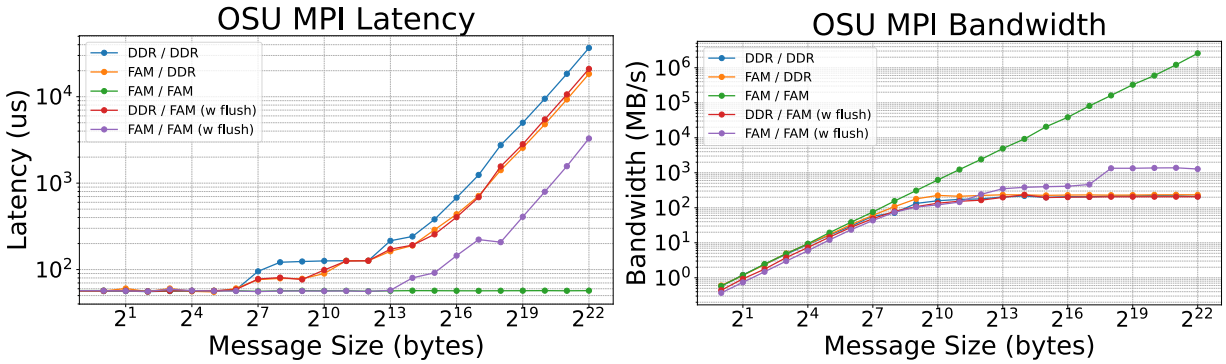


Figure 16 OSU MPI Benchmarks

On real-world MPI-based application workloads, the performance enhancements are subdued due to frequent cache coherence invocations. There is no standard way to mitigate this effect, and we anticipate designing newer programming models or adapting existing runtimes such as MPI or SHMEM with specific functions to handle the FAM memory kind.

LLNL LULESH mini-application is from the area of shock hydrodynamics and used extensively for software-hardware co-design of HPC systems. We use the FAM-FAM allocation strategy (Case 4 above), requiring allocation of the send buffers on FAM, and 26 receive-side buffers (pertaining to the neighbors in a 3D process grid) for holding the pointers (or base address of remote buffers), allowing direct access (and reducing about 100 LoC after eradicating the MPI_Recv calls). The results are shown below (in terms of figure-of-merit, higher is better), and indicates competitive performances between DDR and FAM. This bears further investigation and could be an artifact of the FPGA-enabled CXL switch emulation adopted in the current platform and an artifact of over-flushing to facilitate cache coherence.

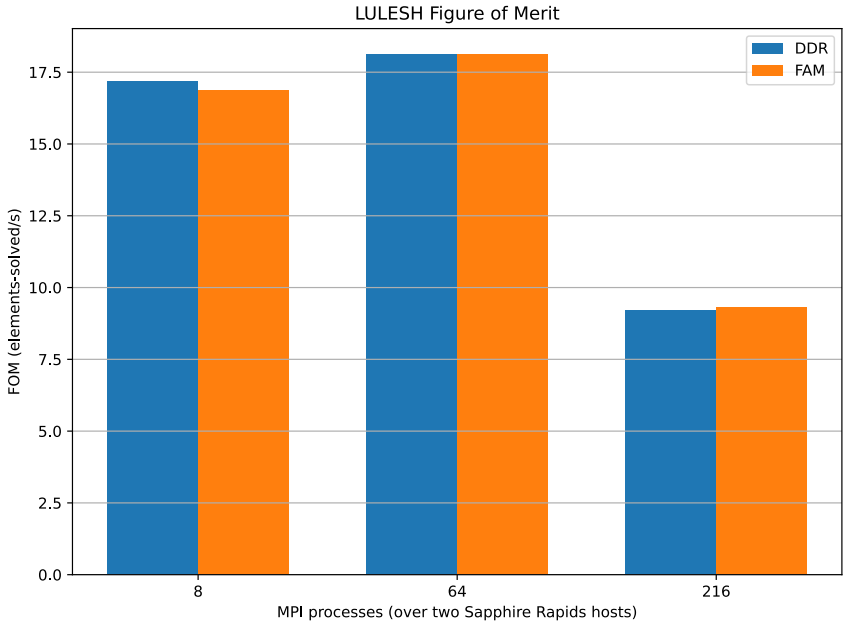


Figure 17 FAM vs. Local Memory MPI LULESH Benchmark

2.3.3 Summary

CXL memory pooling is ideal for capacity-driven nearest neighborhood or random accesses; as such, large scale graph analytics or inference use cases can leverage such system integrations. Our evaluations suggest that non-contiguous short reads in single-host graph analytics and database workloads can benefit from direct-access CXL disaggregated shared-memory pools, via minimal code changes, and minor impact on the performance bottom line. Although, distributed-memory models such as MPI would require nontrivial adaptations to port workloads on switch-based CXL, for managing coherency efficiently. Since MPI is a widely portable programming model, supporting CXL memory kinds, and exposing semantics for coherence or implementations handling coherence internally (upon detecting specific CXL device types) would propel the adoption of disaggregated memory systems in HPC.

2.4 Scale-Up

2.4.1 System Software

2.4.1.1 Hardware-Software Co-Development for Emerging CXL Architectures

The exponential growth of data-intensive workloads, from High-Performance Computing (HPC) simulations to AI-driven analytics, has exposed limitations in conventional memory architectures. These workloads share a common need: large, unified memory pools accessible via shared-memory programming paradigms. CXL addresses this by providing high-speed, low-latency interconnects between CPUs, memory devices, and accelerators, leveraging the PCIe physical layer while introducing protocols for coherence and flexibility.

Despite its promise, CXL hardware prototypes often support only subsets of CXL protocols, and the accompanying software stack is limited to basic validation. This mismatch creates a bottleneck: by the time hardware is ready, software lags, delaying deployment and preventing valuable feedback from influencing hardware design. In this work, we argue for co-development, where hardware and software evolve together, reducing time-to-deployment and enabling real-time feedback loops that inform design choices.

To operationalize co-development, the authors introduce a framework integrating:

A full-system CXL emulator built on QEMU,

MemForge, a Rust-based memory allocator for CXL-attached devices,

High-level APIs for CXL system programming.

This combination allows developers to build and test applications leveraging CXL memory even before physical hardware is available. The emulator abstracts low-level hardware details, while MemForge provides intuitive abstractions for memory management, exposing familiar data structures like vectors and matrices. Crucially, MemForge includes introspective telemetry for profiling and debugging, supporting performance tuning and correctness verification.

2.4.1.2 Emulation System: QEMU as the Backbone

QEMU serves as the foundation for the emulation environment, offering dynamic binary translation to run guest operating systems on a host machine. Its PCIe emulation infrastructure extends naturally to CXL, enabling virtualized components such as:

CXL Host Bridges for traffic routing,

CXL Root Ports linking hosts to devices or switches,

CXL Switches for complex topologies and resource pooling,

CXL Memory Devices (Type 3) for volatile or persistent memory experiments.

This flexibility allows developers to emulate configurations ranging from direct-attached memory to multi-device setups connected via CXL switches. In this study, two configurations are evaluated:

Direct-attached memory: a single memory expander linked to the host,

Switch-based topology: two memory devices connected through a CXL switch.

By leveraging QEMU, developers can build and validate CXL-aware drivers, system software, and applications without costly physical prototypes, accelerating readiness for real deployments.

2.4.1.3 MemForge: A Rust-Based Allocator for CXL Memory

MemForge is designed to manage CXL-attached memory efficiently while ensuring safety and concurrency. Rust's ownership model and compile-time borrow checker reduce runtime overhead by shifting consistency checks to compile time, a critical advantage at scale, where thousands of nodes may share terabytes of memory.

Internally, MemForge implements the buddy system algorithm, which recursively splits large contiguous blocks into smaller chunks to satisfy allocation requests, minimizing fragmentation. Upon deallocation, adjacent free blocks are merged to restore larger blocks, improving future allocation efficiency. MemForge supports multiple backing strategies:

File-backed memory for development on non-CXL systems,

Device-backed memory for CXL environments.

Applications can adopt MemForge with minimal changes, thanks to its compatibility with Rust's standard allocator trait. For example, simply replacing vec::with_capacity() with vec::with_capacity_in() in an application enables seamless integration. MemForge also offers robust tracing and logging capabilities via Rust's log and tracing crates, with optional OpenTelemetry integration for distributed environments. Trace data can be visualized using tools like Perfetto, aiding performance diagnosis and debugging.

Experimental Results

The evaluation spans three representative kernels:

- GEMM (General Matrix Multiply): ubiquitous in HPC and machine learning,
- SpMM (Sparse Matrix-Dense Matrix Multiply): critical for graph analytics and GNNs,
- Transformer kernel: simulating attention mechanisms and feedforward layers in LLMs.

Experiments run on QEMU (v9.2.90) with CXL extensions, using virtual machines configured with 8 vCPUs, 8GB DRAM, and either:

- A 4GB direct-attached CXL memory expander, or
- Two 2GB memory devices connected via a CXL switch.

Both configurations operate in Direct Access (DAX) mode to bypass the I/O stack. Rust nightly (v1.88.0) is used for allocator features.

Although QEMU lacks cycle-accurate timing, relative trends are informative. Across all benchmarks, MemForge introduces negligible runtime overhead compared to Rust's standard allocator. For small inputs, allocation costs dominate, but as input sizes grow and computation becomes the bottleneck, performance converges across configurations. The switch-based setup incurs slightly higher latency than direct-attached memory, reflecting expected overhead from additional routing.

Tracing capabilities further enhance observability. For example, SpMM traces reveal two waves of allocation and deallocation events corresponding to matrix format conversions. Visualization in Perfetto highlights thread activity and timing, enabling fine-grained performance analysis.

Conclusions and Future Directions

The study demonstrates that hardware-software co-development is not only feasible but essential for emerging technologies like CXL. By combining QEMU-based emulation with MemForge's allocator and APIs, developers can prototype applications, validate designs, and provide feedback before hardware reaches maturity. Future work will enhance timing accuracy through cycle-accurate models, extend support for cxl. cache and integrate accelerator devices into the co-development framework. These advancements will further align hardware and software trajectories, accelerate deployment and unlock CXL's full potential for HPC, AI, and data analytics.

2.4.1.4 MemForge Mesh

MemForge Mesh is a distributed shared-memory architecture that explicitly separates coordination from data access. Coordination is managed by a lightweight authority service, whereas data access occurs through direct memory mappings on each participating node. The design intentionally eliminates network-based data movement: the authority never manipulates data, and applications do not invoke the authority during performance-critical execution paths. Instead, all nodes map a common backing store (e.g., a shared file or a device dax device) and employ leases issued by the authority to regulate access.

A typical deployment consists of:

- A shared backing store (such as a file in a shared directory or a /dev/dax* device exposed via CXL).
- A single authority process responsible for metadata management and lease arbitration.
- Node-local agents interfacing with the backing store.

Applications interact through a concise API that either embeds the agent or connects via inter-process communication (IPC). This API provides typed handles for memory mapping and lease-controlled access.

MemForge Mesh presents the following key properties:

- Data-plane locality: Reads and writes occur via mmap or DAX mappings, eliminating remote procedure calls (RPC) from the data path.
- Coordination without data transfer: The authority exclusively manages object creation and lease operations, ensuring predictable scalability.
- Pluggable persistence: A unified abstraction supports file-backed, in-memory, and DAX storage, applying alignment and flush semantics consistently.

From an algorithmic perspective, MemForge Mesh operates as a shared-address allocator combined with a lease coordinator. Upon object creation, the authority assigns an offset and length within the backing store, which the agent caches locally. Applications then map these offsets directly for read/write operations. Leases enforce concurrency control, permitting multiple readers while serializing writers. Durability is explicit: applications invoke flush operations to guarantee persistence. MemForge Mesh is not a distributed filesystem; rather, it provides precise control over allocation, mapping, and ownership. This design is particularly suited for high-performance computing (HPC) workloads and domain-specific runtimes where explicit control and predictable latency are essential.

The following sections describes MemForge Mesh's key components:

MemForge-Core

MemForge-Core constitutes the foundational layer of the system. It does not execute servers, open sockets, or perform memory mappings; instead, it defines the invariants that ensure global consistency. This component provides alignment utilities and allocator primitives employed throughout the stack, particularly within the authority's offset assignment logic.

Conceptually, MemForge-Core serves as the formal specification for size and alignment constraints. When the authority allocates a range within the backing store, it guarantees proper alignment and adherence to power-of-two constraints. These rules are encapsulated in reusable functions. For example, the LayoutEx type represents a size-plus-alignment descriptor, forming

the basis for reasoning about allocation. Functions such as align_up_pow2 implement rounding behavior, while order_for_len determines allocation strategies.

The authority employs a monotonic bump allocator with alignment guarantees. This approach yields deterministic, O(1) allocations that are contiguous, simplifying reasoning in the data plane. Although memory reclamation is not supported, this trade-off is acceptable for large persistent backings and structured workloads, where simplicity and predictability are paramount.

Despite its modest size, MemForge-Core is critical: any modification to its alignment logic directly influences object layout and mapping correctness. For DAX backings, alignment is a strict requirement; thus, the correctness of these utilities determines whether mappings succeed.

MemForge-Backing

MemForge-Backing defines the interface between the system and the operating system. It introduces the Backer trait, which provides a unified abstraction for mapping regions of a backing store into memory. Core operations include mapping regions as read-only or read-write, unmapping regions, flushing ranges, and optionally issuing persistence barriers.

The file-based backer offers portability by leveraging memmap2 for shared mappings and maintaining a registry of active maps for explicit unmapping. Its flush operation employs msync semantics, sufficient for file-backed persistence. The mock backer implements the same trait atop an in-memory vector for testing purposes.

The DAX backer is specialized for persistent memory devices. Device-DAX mappings impose strict alignment constraints, which the kernel exposes via sysfs (e.g., /sys/class/dax or /sys/bus/dax/devices). The DAX backer retrieves device size and alignment from sysfs and rejects mapping requests that violate these constraints. It also treats msync failures such as EINVAL as no-operations, reflecting the unique behavior of DAX devices.

Critically, the backer enforces data-plane invariants. While the authority may approve an object of size X, if the mapping targets a DAX device requiring 2 MiB alignment, the backer will reject misaligned requests. Consequently, higher-level components expose create_with_align variants to ensure compliance with backing constraints from the outset.

In summary, MemForge-Backing constitutes the data-plane contract, defining how bytes on disk or device become memory pointers. It underpins the system's promise of zero-copy data access.

MemForge-Mesh-Proto

The proto package specifies the control-plane schema governing metadata operations between agents and the authority. By design, the schema includes only metadata (object identifiers, offsets, lengths, classes, and lease operations) excluding any data payloads.

This architectural decision ensures that the authority remains lightweight, scalable, and stable. By constraining the protocol to metadata, MemForge Mesh avoids transferring large payloads across the network and maintains a clear separation of responsibilities. The proto package thus serves as both a technical artifact and a design boundary, delineating the authority's scope.

Generated using tonic, the protocol integrates seamlessly with Rust services. Client stubs are employed by agents, while server stubs are implemented by the authority. Developers rarely modify these definitions during routine use; they become relevant primarily when evolving coordination semantics.

MemForge-Mesh-Server (Authority)

The authority server represents the metadata nucleus of MemForge Mesh. Its responsibilities are twofold: allocating ranges within the shared backing and arbitrating leases. It does not perform data mapping or access and never interacts directly with the backing store.

Upon receiving an object creation request, the authority allocates an offset and length within a fixed capacity using a monotonic bump allocator. This allocator aligns the requested size (minimum 4 KiB or higher if specified), assigns a contiguous range, and advances the bump pointer. This strategy ensures fast, deterministic allocation. The resulting metadata entry (comprising offset, length, class, and version) is stored in a directory indexed by object identifier.

Leases implement the consistency mechanism. A client requests read or write leases, which the authority grants based on current state. Read leases may be concurrent, whereas write leases are serialized. Version numbers facilitate detection of stale access and ordering enforcement. Although the current implementation is intentionally simple, the lease API provides a foundation for more sophisticated policies.

Operationally, the authority is a singleton service with minimal state. It can be restarted if backed by an external metadata store; however, in its current in-memory form, it should be treated as ephemeral. Its scalability derives from its exclusive focus on metadata, rendering it robust under high data throughput.

MemForge-Mesh-Agent

The agent functions as the node-local orchestrator for data access, mediating between the authority and application code. Its primary role is to translate global object identifiers into local memory mappings. When an application opens or creates an object, the agent retrieves metadata from the authority, caches the offset and length, and invokes the backer to establish the mapping.

Two operational modes are supported:

Embedded: The agent resides within the application process, suitable for testing and demonstrations.

Daemon with Proxy: The agent executes as agentd on each node, exposing a local IPC socket. Applications connect via AgentProxy, enabling multiple processes to share mappings, caches, and leases without redundant operations.

Algorithmically, the agent maintains a directory cache mapping object identifiers to metadata tuples (offset, length, class, version). This cache minimizes repeated authority lookups. Lease acquisition is explicit: prior to mapping read or write access, the agent requests a lease from the

authority. Leases are encapsulated in guard objects that release upon scope termination, aligning with Rust's ownership semantics.

The agent also manages backing selection, supporting file-based, DAX, or automatic detection via BackingKind. This flexibility permits heterogeneous deployments without altering application code. DAX constraints are enforced by delegating to the DAX backer, which rejects misaligned mappings. Consequently, components targeting DAX must request aligned sizes during object creation.

MemForge-Mesh (User API)

The memforge-mesh crate provides the application-facing API, exposing typed handles (e.g., DmaHandle<T>) and higher-level abstractions (DmaBox, DmaVec) that facilitate interaction with distributed shared memory in Rust.

A DmaHandle<T> represents a globally identified object within the backing store. Applications create or open handles and subsequently map them for read or write operations. Mapping methods return guard objects coupling mapping lifetimes with leases, ensuring coordinated access. This design adheres to RAII principles: upon guard destruction, the lease is released, and the mapping invalidated.

Two features warrant emphasis:

Alignment control: The create with align method enables callers to specify alignment requirements, essential for DAX compliance.

Explicit flush: The API provides flush operations, allowing applications to enforce persistence guarantees, a critical requirement for persistent memory.

The user API abstracts the authority entirely, operating through the agent or proxy. This indirection ensures consistent coordination and mapping logic, preserving API stability across deployments and facilitating long-term production use.

Expected Workflow

The interaction sequence is as follows:

- Application invokes object creation or opening via the MemForge-Mesh API.
- Agent/Proxy consults the authority, caches metadata, and acquires leases.
- Backer maps the object's offset and length into local memory.
- Application performs read/write operations, flushes if necessary, and releases leases.

The authority remains the definitive source of metadata, while the backing store constitutes the data plane. The agent mediates these components, ensuring consistent allocation and lease state across processes.

Closing Perspective

MemForge Mesh delivers a minimal yet rigorous framework for distributed shared memory. Its explicit coordination model and lightweight architecture afford expert users the control and predictability required for high-performance and persistent memory workloads, without imposing the complexity of conventional distributed storage systems.

2.4.2 Compiler

2.4.2.1 A Compiler Framework for Linear Algebra on CXL-Attached Memory

The growing memory demands of data-intensive applications such as scientific simulations, machine learning, and graph analytics have pushed traditional server architectures to their limits. When datasets exceed the capacity of local DRAM, systems often resort to storage devices like NVMe or hard drives, which introduce orders of magnitude higher latency. Fabric-attached memory (FAM) offers a middle ground, providing lower latency than storage while offering larger capacity than DRAM. Among emerging standards, Compute Express Link (CXL) has gained traction as an interconnect technology that enables byte-addressable access to disaggregated memory pools via load/store semantics. This capability makes CXL an attractive solution for memory-intensive workloads, but its higher latency compared to DRAM, typically two to five times greater, poses challenges for efficient utilization.

The work introduces extension to the COMET designed to mitigate these challenges and enable efficient use of CXL-attached memory for linear algebra computations. COMET combines a domain-specific language (DSL) for tensor algebra with a multi-level intermediate representation (IR) compilation pipeline and optimization strategies such as tiling and double buffering. The goal is to allow programmers to specify memory placement decisions at a high level while the compiler automatically generates optimized code that hides CXL latency.

This illustrates the performance gap between DRAM, disk storage, and FAM using a matrix multiplication benchmark. Storing data on disk results in execution times thousands of seconds longer than DRAM, while FAM offers intermediate performance. This demonstrates the potential of FAM as a scalable memory solution, provided its latency can be effectively managed. Existing approaches to leveraging remote memory fall into three categories: kernel-based solutions that migrate pages transparently, library-based approaches that expose explicit APIs, and compiler-based strategies that automate code generation based on high-level annotations. COMET adopts the compiler-based approach, aiming to balance ease of use with performance.

COMET is built on the MLIR framework, which supports progressive lowering from high-level abstractions to low-level code. The workflow begins with a DSL that allows users to write tensor-based programs and annotate tensors with an allocator attribute indicating whether they should reside in DRAM or FAM. For example, in a matrix multiplication kernel, input tensors can be placed in FAM while the output tensor remains in DRAM. This annotation-based approach gives programmers explicit control over data placement without sacrificing the simplicity of high-level programming. The compiler processes these annotations through multiple IR layers. At the top, the Tensor Algebra dialect captures the semantics of operations using Einstein notation, preserving mathematical structure for algebraic optimizations. The Index

Tree dialect extends support for sparse tensors and masking operations. Mid-level dialects such as MemRef and SCF introduce concrete memory references and loop structures, enabling optimizations like tiling and double buffering. Finally, the code is lowered to LLVM IR, where memory allocations are translated into calls to either the CXL allocator or standard malloc, and machine code is generated using LLVM's backend optimizations.

To address CXL's higher latency, COMET implements two key optimizations: tiling and double buffering. Tiling partitions input tensors into smaller, cache-friendly chunks, improving data locality and enabling prefetching. By analyzing tensor semantics and loop structures, the compiler can predict access patterns and prefetch tiles from FAM into DRAM before they are needed. Double buffering overlaps computation with data movement by maintaining two buffers per compute worker: one for computation and one for prefetching. An auxiliary worker thread handles prefetching, while compute workers process data in parallel. When both prefetching and computation complete, buffers are swapped, allowing continuous overlap of data transfer and computation. This approach hides latency and maximizes throughput.

2.4.2.2 Evaluation and Results

We evaluate COEMT using dense matrix multiplication (GEMM) on a CXL prototype system comprising a dual-socket Intel Xeon Gold processor and an FPGA-based memory accelerator connected via PCIe 5.0. The FAM device provides 32 GB of CXL-attached memory. Experiments compare four configurations: DRAM-only, FAM-only, FAM with tiling, and FAM with double buffering. On 16384×16384 matrices using 28 threads, naïve FAM execution achieves only 0.30× the performance of DRAM, reflecting a 3.3× slowdown. Tiling improves performance to 0.86× of DRAM, while double buffering delivers a 2.50× speedup over DRAM, demonstrating the effectiveness of latency-hiding optimizations.

Thread scalability experiments show that double buffering consistently outperforms other configurations across thread counts, except at very low concurrency where synchronization overhead dominates. Profiling results reveal that double buffering improves instructions per cycle (IPC) by 12.6× compared to naïve FAM execution, primarily due to reduced L2 miss rates and more predictable prefetching. Hardware counters confirm that stalls shift from FAM-bound to DRAM-bound, indicating successful prefetching. Sensitivity analysis on tile size shows that optimal performance occurs with tiles sized to fit in the 2 MB L2 cache (typically 256–512 elements), simplifying autotuning.

2.4.2.3 Conclusion and Future Directions

COMET demonstrates that compiler-based approaches can effectively bridge the gap between high-level programming and low-level memory optimizations in heterogeneous memory systems. By combining a DSL, multi-level IR pipeline, and targeted optimizations, COMET enables applications to exploit large CXL-attached memory pools without incurring prohibitive latency penalties. The evaluation shows that double buffering can transform FAM from a performance liability into an asset, achieving speedups over DRAM by overlapping computation and data movement. Future work will focus on automatic tile size selection based on cache hierarchy analysis, support for additional linear algebra operations, and hybrid allocation strategies that dynamically balance data placement between DRAM and CXL memory. The

authors also plan to extend COMET to GPUs and accelerators, broadening its applicability to diverse computing environments.

2.4.3 Co-Design and System Emulation

2.4.3.1 Performance Study of CXL Memory Topology

The rapid evolution of data-intensive applications has placed unprecedented demands on memory systems. High-Performance Computing (HPC) workloads and Large Language Model (LLM) inference tasks exemplify this trend, requiring architectures that deliver both high bandwidth and low latency. Traditional memory configurations, while foundational, often fail to meet these requirements, leading to bottlenecks that compromise scalability and efficiency. Against this backdrop, Compute Express Link (CXL) emerges as a transformative interconnect technology, promising flexible and coherent memory sharing across heterogeneous computing environments. This study delves into the performance implications of various CXL memory topologies, with a particular emphasis on the role of CXL switches.

CXL introduces three protocols (CXL.io, CXL.cache, and CXL.mem) multiplexed over PCIe, enabling dynamic memory allocation and coherent data sharing between CPUs, GPUs, and other accelerators. While the potential benefits of CXL are clear, its practical deployment raises critical questions. Direct-attached memory configurations, for instance, risk memory stranding and fail to exploit CXL's full capabilities. Conversely, CXL switches promise scalability by connecting multiple hosts to multiple memory devices, but they introduce overhead and routing complexity. Understanding these trade-offs is essential for system architects seeking to optimize performance in modern data centers.

This work sets out to evaluate the performance impact of different CXL memory topologies under two representative workload classes: HPC and LLM inference. We aim to:

Quantify performance variations across topologies.

Identify congestion effects introduced by CXL switches.

Propose a latency-based cost model to assess Total Cost of Ownership (TCO) benefits.

Through a combination of emulation and limited machine-level validation, the study provides actionable insights for designing CXL-enabled infrastructures.

The integration of CXL.mem and CXL switches represents a paradigm shift in memory architecture. CXL.mem facilitates memory expansion and sharing, reducing bottlenecks in data-intensive applications. However, commercial CXL.mem devices remain scarce, necessitating simulation-based approaches. To bridge this gap, we employ CXLMemSim, a lightweight emulator that injects realistic latency and bandwidth characteristics into unmodified applications. By modeling reorder buffer behavior and cache-line states, CXLMemSim enables accurate profiling of memory access patterns across diverse topologies.

Simulation Results

The evaluation platform comprises a dual-NUMA Intel Xeon system running Ubuntu 22.04 with kernel 5.15. The study explores four distinct topologies:

- Direct-attached CXL
- Switch-attached CXL
- Switch-attached with mixed paths
- Single switch connecting multiple CXL devices

Workloads include BT (dense linear algebra) and LU (sparse linear algebra) from the NPB-OMP benchmark suite for HPC, and Mistral-7B inference using Intel's IPEX-LLM library for LLM evaluation. Each workload is executed across varying OpenMP thread counts, with process pinning via numactl to control NUMA locality.

The results reveal a nuanced interplay between topology, thread count, and workload characteristics. For HPC workloads, congestion effects become evident as parallelism increases. In topology-3, for example, BT execution time rises from 2406.86 seconds at one thread to 2433.29 seconds at eight threads, indicating that the benefits of parallelism are negated by switch-induced bottlenecks. LU exhibits similar trends, with execution time fluctuating but generally increasing under higher thread counts. These findings underscore the importance of topology-aware scheduling: blindly scaling thread counts can degrade performance when switch congestion dominates.

LLM inference workloads exhibit a comparable pattern, albeit with distinct nuances. For Mistral-7B, topology-3 initially benefits from parallelism, reducing execution time from 461 seconds at one thread to 439 seconds at four threads. However, performance deteriorates beyond this point, climbing back to 458 seconds at twelve threads. Topology-2 fares worse, showing congestion effects almost immediately as thread count increases. These observations highlight the complexity of optimizing LLM deployments in CXL-connected environments, where memory access patterns and topology characteristics interact in unpredictable ways.

Machine-Level Validation

To corroborate emulator findings, We conducted STREAM benchmark tests on a real CXL switch topology. Bandwidth scales nearly linearly up to 32 threads but plateaus thereafter, with some operations (Copy, Scale) even declining. This validates the emulator's prediction: beyond a certain concurrency threshold, switch saturation imposes hard limits on scalability.

Latency-Based Cost Model and TCO Analysis

Beyond performance metrics, we introduce a latency-based cost model to estimate TCO savings from CXL adoption. The model partitions execution time into segments corresponding to data residency in main memory, CXL memory, and SSD. By normalizing SSD latency to one and incorporating measured relative latencies for MMEM (0.1) and CXL (0.125), we derive an

analytical model to compute server count reductions under capacity-bound workloads. For example, with a main-to-CXL memory ratio of two and congestion thresholds factored in, the model predicts a 34.25% reduction in server count and a 27.67% TCO saving—even assuming a 10% cost premium per CXL-equipped server. This analysis provides a practical framework for data center operators weighing the economic viability of CXL deployment.

Conclusion

CXL technology holds immense promise for addressing the memory challenges of modern computing. However, its benefits are contingent on careful topology design and congestion management. By analyzing the performance trade-offs and economic implications of CXL memory topologies, this study equips system architects with the knowledge needed to harness CXL effectively. The study's findings carry several implications for system design:

Topology Selection Matters: Performance varies significantly across configurations; one-size-fits-all approaches are untenable.

Congestion Awareness Is Critical: Switch saturation can negate parallelism benefits, necessitating topology-aware scheduling and resource allocation.

Cost Modeling Is Essential: Economic considerations should guide procurement decisions, particularly for capacity-bound workloads like LLM inference.

Validation Remains Key: Emulation offers valuable insights, but real-world measurements are indispensable for fine-tuning deployment strategies.

Future research should continue refining emulation tools, exploring multi-switch fabrics, and developing adaptive scheduling algorithms to fully realize CXL's potential in HPC and AI-driven environments.

2.4.4 QEmu-CXL

QEmu is a system emulator widely used in the scientific and engineering communities. Recently, CXL extensions have been added to QEmu to emulate CXL-based systems, including systems featuring direct-attached memory pools or memory devices attached to CXL switches. We leverage the jic23/qemu repository on GitLab, a specialized fork of the QEMU project, primarily intended for continuous integration (CI) testing and experimental development related to Compute Express Link (CXL) emulation. QEMU itself is a widely used, open-source machine emulator and virtualizer that can emulate complete hardware systems in software. It supports dynamic binary translation for performance and integrates with hypervisors such as KVM and Xen to achieve near-native CPU performance when hardware virtualization is available. This makes QEMU a versatile tool for running operating systems designed for one architecture on a different host architecture, as well as for providing user-space API virtualization for Linux and BSD environments.

The jic23/qemu fork focuses on extending QEMU's capabilities to emulate emerging CXL features. CXL is an open interconnect standard designed to enable coherent memory sharing between CPUs, accelerators, and memory devices. Hardware availability for CXL is still limited,

so emulation plays a critical role in early software enablement, specification validation, and development of kernel and firmware support. This repository provides a platform for testing CXL device models, switch topologies, and fabric management features without requiring physical hardware.

Key aspects of this fork include support for CXL memory devices (Type 3), host bridges, and switches, along with mechanisms for mapping host physical address spaces to emulated CXL components. It enables developers to experiment with advanced features such as dynamic capacity devices, mailbox command interfaces, and tunneling of fabric management commands. These capabilities are essential for building and validating orchestration tools and kernel features that will eventually manage real CXL fabrics. The repository also serves as a staging ground for patches that rework QEMU's internal handling of CXL mailboxes and introduce switch-level command interfaces, which are part of the CXL 3.0 specification.

From a software architecture perspective, this fork retains QEMU's modular design, which includes subsystems for CPU emulation, device models, and I/O virtualization. It leverages QEMU's Tiny Code Generator (TCG) for dynamic translation and supports plugins for tracing and profiling, which are useful for feeding memory access traces into external simulators for performance modeling. Developers can configure complex virtual topologies, including multi-host scenarios with distributed memory servers, to emulate realistic deployment environments.

2.4.4.1 Support for Co-Design

We leverage QEmu for the co-design and co-development studies performed across the various work summarized in this report. Specifically, both MemForge and MemForge Mesh described in Sections Hardware-Software Co-Development for Emerging CXL Architectures and MemForge Mesh, respectively, use QEmu for hardware-software co-design and co-development.

2.5 Scale-Out

2.5.1 Overview

The Scale-out task is a primary component of the AMAIS (Advanced Memory for AI Science) project at Pacific Northwest National Laboratory (PNNL). Our mission is to evaluate Compute Express Link (CXL) based Fabric-Attached Memory (FAM) as a high-performance alternative to traditional network interconnects in the exascale ecosystem.

Modern scientific discovery has become increasingly data-driven, requiring the integration of theory, simulation, and experimental results with advanced artificial intelligence (AI) and machine learning (ML). This paradigm shift requires unprecedented levels of memory; experimental and simulated data can quickly consume petabytes of DRAM, while the latest ML models contain trillions of parameters requiring terabytes of accelerator memory. To support this vision of **integrated science**, large-scale clusters require significantly larger pools of fast memory and more efficient methods for sharing data across compute resources.

While memory accelerators enabled by CXL are poised to provide the large, shared memory needed to handle these growing footprints, a fundamental research challenge remains: **actually, scaling out**. While CXL-based communication offers the potential for significantly lower latency and higher throughput than traditional message-passing or RDMA-based protocols, realizing these gains requires the implementation of the correct synchronization and abstraction mechanisms. We turn to **Distributed Asynchronous Many-Task (AMT)** runtimes and **Partitioned Global Address Space (PGAS)** memory models as the solution. These abstractions lend themselves uniquely well to addressing memory pools far larger than the capacity of a single node, providing a framework to manage fine-grained data sharing without the overhead of legacy software stacks. In this work, we demonstrate how this architectural approach, implemented via the **ARTS runtime**, can effectively scale scientific ML and Department of Energy (DOE) mini applications across disaggregated CXL resources. Moreover, we are introducing the **CARTS compiler framework** which is designed to take OpenMP codes and effectively map them to distributed environments via the ARTS runtime systems. We have promising results with this framework, beating some OpenMP implementation by **7x** in single node runs.

2.5.2 System Software for CXL-based FAM: Locks and Allocators

To leverage CXL-based FAM without native hardware atomic support in early prototypes, we addressed the fundamental challenges of managing memory that is physically disaggregated but logically shared across sovereign CPU nodes.

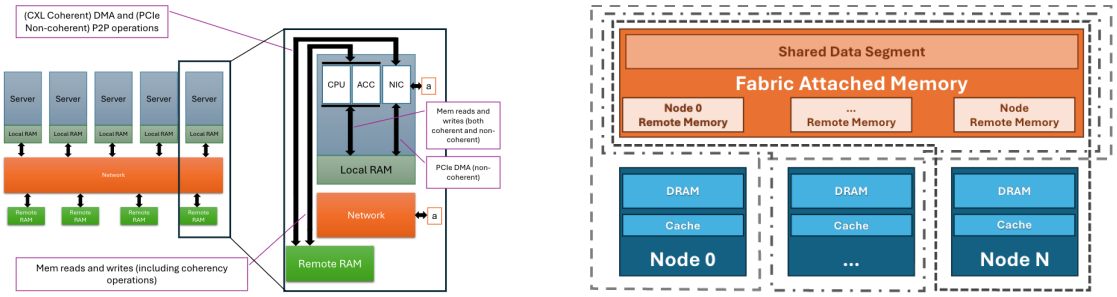


Figure 18 Scale Out Overview

2.5.2.1 Programmatic Shared-Memory Allocators

In this work, we build upon the initial implementations of global memory allocators provided by our hardware and software partners. While these early versions established the foundational ability to share data across processes and nodes, their implementation via manual string-based lookups and raw byte-block management lacked an inherent notion of objects or their initialization (i.e., C++ object orientation). This baseline infrastructure was a necessary first step, but it proved insufficient for complex parallel programs, as it required developers to manually bridge each node to specific memory regions and lacked a programmatic approach to dynamic allocation. Moreover, it provided no structural guarantees regarding the readiness of an allocation for sharing, which often resulted in error-prone manual synchronization. Furthermore, a more programmatic approach will be essential for future compiler integration as the software stack matures to support standard C++ features.

To address these limitations, we developed a high-level, programmatic framework that makes disaggregated memory feel like a natural extension of the local heap. For parallel programs operating in Single Program, Multiple Data (SPMD) mode, we implemented an automatic naming convention. By ensuring nodes attempt to allocate or access memory in a deterministic order across the fabric, the system generates synthetic object names (e.g., obj1, obj2), effectively removing the burden of manual string management from the developer.

```
SharedFam<int> num(10);
int main(int argc, char * argv[]) {
    if(famRank == 1) {
        assert(*num == 10);
        *num = 5;
        num.producerFlush();
    }
    if(famRank) {
        while(1) {
            num.consumerFlush();
            if(*num == 5)
                break;
        }
    }
    return 0;
}
```

Figure 19 Shared Memory Invocation Snippet

This framework is supported by a specialized ranking convention—functioning independently of MPI—that designates a single "owner" node responsible for the initial allocation and subsequent initialization of a shared object. To ensure memory is never accessed in an inconsistent or unallocated state, the system employs state-aware synchronization: while the owner node

prepares the data, the remaining nodes in the cohort automatically wait for the allocation to signal its availability. The resulting interface adopts a design similar to C++'s std::shared_ptr, allowing developers to manage shared FAM resources using familiar object-oriented semantics while the system handles the underlying complexities of cross-node availability and initialization.

2.5.2.2 Distributed Synchronization (Software-Based Locks)

While CXL 3.x specifications provide for hardware coherency via the write-back-invalidate mechanisms, the feasibility of vendor implementation remains a significant uncertainty. Consequently, we formulated our synchronization strategy around software-based coherency using a Tournament of Peterson Locks.

This design allows N threads to contend for a critical section with O(log(N)) complexity while requiring only atomic read/write semantics, avoiding the need for advanced hardware intrinsics currently absent in Type 3 device implementations. To manage the cache-coherency gap between nodes, we manually inserted producer and consumer flush operations. While functional, evaluation on our prototype (112 threads across two nodes) revealed a 1000x overhead compared to single-thread baselines, emphasizing that synchronization costs must be amortized by sufficiently large critical sections.

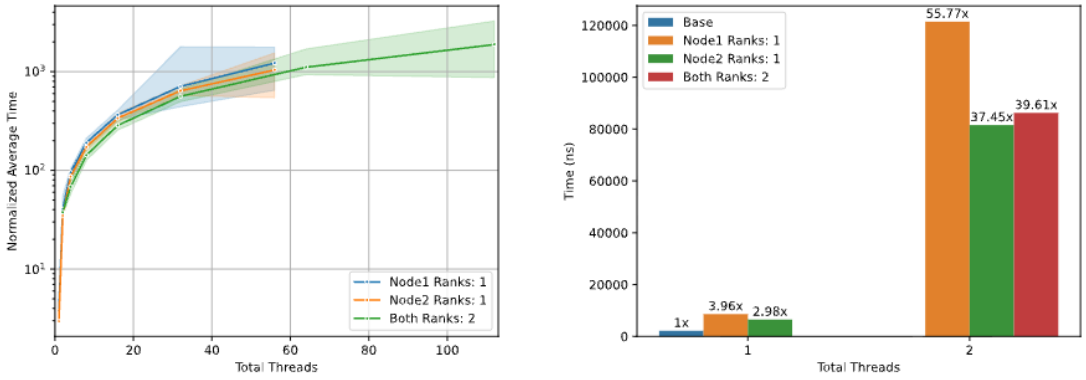


Figure 20 Pearson Lock Performance

To evaluate the feasibility of this approach, we conducted testing on a two-node CXL prototype. Our findings indicate that while functional mutual exclusion can be achieved across a disaggregated fabric using only atomic reads and writes, the performance profile is heavily influenced by the lack of hardware-backed atomics. In our environment, local DRAM latency on Xeon Gold 5420 nodes was approximately 100 ns, whereas the CXL-FAM interface introduced roughly 250 ns of latency. Despite this 2.5x increase in raw access time, the true performance bottleneck emerged from the software-managed coherency required for mutual exclusion. The necessity of manual flush operations results in a significant performance penalty; at a high contention level of 112 threads, we observed a normalized average time for lock acquisition that was approximately 1000x to 3000x higher than a single-threaded baseline. These results

emphasize that in current disaggregated architectures, synchronization costs must be carefully amortized by utilizing sufficiently large critical sections.[1]

Comparative Analysis with Standard Lock Algorithms

As part of our exploration, we analyzed why standard high-performance locks—such as Ticket, Anderson, and MCS—are suboptimal when decoupled from hardware-backed vertical memory hierarchies and applied to disaggregated CXL memory:

Ticket Locks: While memory-efficient, they rely heavily on a single now-serving variable. In a CXL environment without hardware coherency, every thread spinning on this variable generates excessive FAM traffic and contention, negating the benefits of the load/store interface and saturating the fabric bandwidth.

Anderson's Queue Lock: This algorithm improves contention by assigning each thread a unique slot in an array to spin on. While this effectively manages contention, it significantly increases the memory footprint (O(numProcs)) and still requires hardware-level atomics to manage the shared index—intrinsics that are currently unavailable in first-generation CXL hardware.

MCS Locks: Often the gold standard for local shared-memory synchronization, MCS locks use a linked list for local flag spinning. However, in a CXL testbed, these locality benefits are negated because the flags must be globally visible to successors in FAM to ensure cross-node visibility. Every "local" spin becomes a global flush-and-read operation, effectively transforming the MCS lock into a more complex version of the Anderson lock with significantly higher allocation and management overhead.

2.5.3 ARTS Runtime Development and CXL Integration

The Abstract Runtime System (ARTS) is an Asynchronous Many-Task (AMT) runtime designed to manage massive parallelism by expressing applications as a Directed Acyclic Graph (DAG) of fine-grained tasks. Our recent development has focused on evolving this core architecture to natively support CXL-based fabrics as a first-class execution tier.

Asynchronous Many-Task runtimes represent a significant departure from traditional bulk-synchronous programming models. By decomposing applications into thousands of fine-grained tasks rather than a handful of coarse-grained processes, AMTs can effectively overlap computation with communication. This capability is particularly vital in CXL-based environments; while CXL offers lower latency than traditional networks, access to disaggregated FAM still incurs a performance penalty compared to local DRAM. AMTs leverage a dataflow execution model to ensure that tasks are only scheduled when their specific data dependencies are satisfied. This allows the runtime to dynamically hide latency by switching to ready tasks while others wait for FAM-resident data, providing a high-level abstraction that automates complex coordination and load balancing across heterogeneous resources.

[1] https://www.memsys.io/wp-content/uploads/ninja-forms/5/MEMSYS24_Suetterlein-2.pdf

2.5.3.1 Architectural Foundations: Tasks and Data Blocks

In the ARTS model, task execution is governed by a dataflow-inspired consistency model (CDAG), where work is scheduled only when all signal and data preconditions are met. Data is managed through a global address space partitioned into Data Blocks, each identified by a Globally Unique ID (GUID). This GUID-based system is critical for CXL integration, as it facilitates transparent data access across the cluster, allowing the runtime to treat disaggregated FAM's shared data segment as a logical extension of system memory.

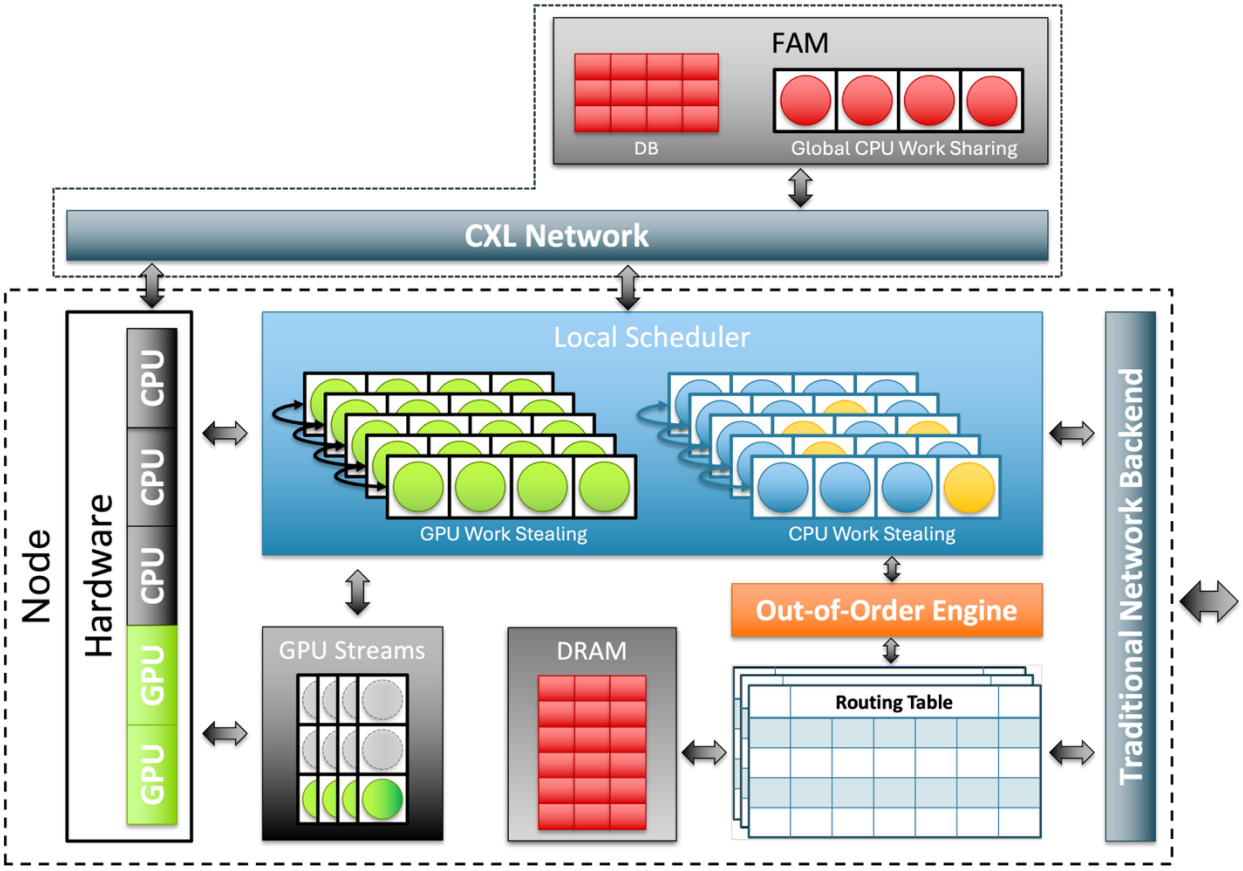


Figure 21 ARTS -- Abstract Runtime System

2.5.3.2 CXL-Enable Task Sharing

We have augmented the native ARTS intra-node scheduler—which traditionally uses a work-stealing paradigm with local deques—to support global task sharing. By leveraging our shared-memory allocator and software locks, we implemented a global queue residing in CXL memory. Workers now push work to this remote queue when local load exceeds specific thresholds, allowing other nodes to participate in the workload via FAM. Because CXL queue access incurs significantly higher costs due to software synchronization, we are currently optimizing heuristics to balance task granularity against cross-node latency.

2.5.3.3 Native Tiered Memory and PGAS Integration

A major advancement is the integration of CXL directly into the native ARTS memory model. By treating CXL-attached memory as a primary tier, the runtime manages FAM-resident objects with the same lifecycle and visibility guarantees as local DRAM. This deep integration allows the runtime to maintain the Partitioned Global Address Space (PGAS) abstraction for the user while performing data-centric scheduling—effectively placing tasks on nodes that have the lowest-latency path to the required FAM-resident Data Blocks.

2.5.4 Compiler for ARTS: CARTS

As part of our scale-out initiative, we have developed CARTS[1], an OpenMP compiler that translates OpenMP programs into a distributed runtime environment. The initial version of CARTS is publicly available at [https://github.com/pnnl/CARTS], with a larger release forthcoming that will incorporate additional distributed optimizations and support for multiple allocation schemes.

Key features of the framework include:

- A ***14-stage compiler pipeline*** built on MLIR-based dialects, enabling advanced metadata analysis and seamless transformation of OpenMP into ARTS executables.

- ***Multiple partitioning strategies*** based on static access pattern analysis, including element-wise, blocked/tiled, and stencil methods with automatic halo region management (termed Ephemeral Slice Dependencies).

- ***Innovative runtime extensions*** such as support for Ephemeral Slice Dependencies, persistent synchronization holders, and compiler-guided memory acquisition.

- ***A comprehensive benchmark suite of 25 tests*** to evaluate correctness and performance compared to standard OpenMP implementations.

The next phase of this research will focus on integrating memory-allocation-centric optimization passes for CXL-enabled runtime components, ensuring cohesive, system-wide performance improvements. The following sections highlight key aspects of the CARTS toolchain.

1 https://llvm.org/devmtg/2025-03/slides/carts.pdf

2.5.4.1 CARTS Internals In-Depth

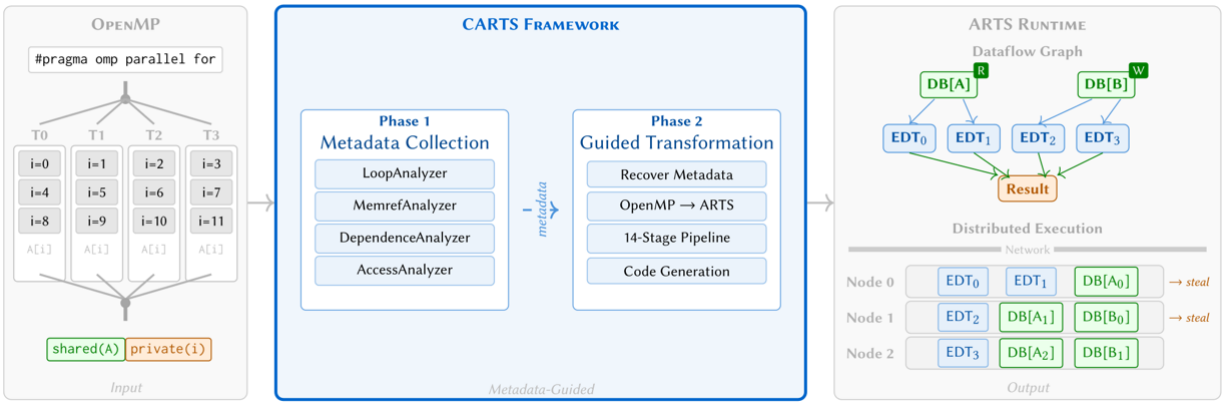


Figure 22 CARTS High-Level Flow

Analyzing parallel programs poses significant challenges, so our initial analysis focuses on the sequentialized version of the code. Once the front end produces the MLIR OpenMP dialect, CARTS sequentializes the program and leverages PolyGeist[1] and other MLIR tools to analyze memory access patterns and loop characteristics (Figure 22). This metadata is then incorporated during the translation from OpenMP to ARTS. Next, concurrent constructs are decomposed into task graphs that closely map to ARTS concurrency units, known as Event-Driven Tasks (EDTs). Synchronization constructs and dependencies are systematically mapped to either events or memory constructs, depending on their role in the control and data flow. For instance, data dependencies are represented as memory constructs ("data blocks"), while control synchronization is managed through events.

Memory constructs are further optimized by selecting the appropriate access mode (read-only or write) and decomposing them to maximize concurrency and storage efficiency while minimizing synchronization overhead. Partitioning strategies—elementwise, striped, blocked, stencil, or coarse—are determined based on this decomposition and access patterns (

Table 1).

[1] https://github.com/llvm/Polygeist

Table 1 Different access pattern and the resulting data block granularity. The third column represents applied cases

Pattern	***Resulting Granularity***	***Applied***
Read Only, single	Coarse	No partition needed
Mixed direct and indirect access	Block	Indirect full range
Stencil/indexed	Stencil	Neighbor access
Uniform Direct	Block	Regular array
Multi node access	Block/Element Wise	Distributed execution
Non-uniform	Coarse	Safety fallback
Offset mismatch (Access can overlap)	Coarse	Data-dependency Access

Additional optimizations include:

- *Tagging read-only arrays* to avoid unnecessary synchronization

- *Tagging write-only arrays* to eliminate redundant data fetching

- *Hoisting acquire and release operations* from loop nests to reduce remote access

- *Synchronization and concurrent unit fusion* for read-only regions

- *Vectorization support* for internal access patterns

Despite its advantages, CARTS has some limitations typical of early-stage compiler toolchains. All function calls within parallel regions must be inlined to enable accurate metadata collection, so recursive and external library functions are handled conservatively. Multidimensional arrays must be declared as static arrays or arrays of arrays; linearized access (e.g., A[i*M+j]) and complex strided patterns are not correctly analyzed. Additionally, support for OpenMP features is not comprehensive limitations include a lack of support for advanced atomic operations beyond basic addition, OpenMP runtime functions, deep nested parallelism (beyond two levels), and target offloading.

2.5.4.2 Current Preliminary Results

Table 2 List of OpenMP kernels that are translated to ARTS

Benchmark Name	Preliminary Speedup
kastors-jacobi/jacobi-for	1.34x
kastors-jacobi/jacobi-task-dep	1.11x
kastors-jacobi/poisson-for	0.81x
kastors-jacobi/poisson-task	0.83x
Montecarlo/ensemble	1.88x
ml-kernels/activations	1.38x
ml-kernels/batchnorm	3.65x
ml-kernels/layernorm	1.55x
ml-kernels/pooling	1.85x
polybench/2mm	2.87x
polybench/3mm	2.27x
polybench/atax	1.94x
polybench/bicg	2.23x
polybench/convolution-2d	1.32x
polybench/convolution-3d	2.77x
polybench/correlation	1.80x
polybench/gemm	**7.26x**
polybench/jacobi2d	**0.41x**
seissol/volume-integral	2.69x
specfem3d/stress	1.51x
specfem3d/velocity	1.3x
sw4lite/rhs4sg-base	1.33x
sw4lite/vel4sg-base	1.33x
stream	0.98x
Polybench/seidel-2d	1.19x

Table 2 summarizes preliminary CARTS results on a single AMD-based node using 64 threads, compared with the standard OpenMP implementation. CARTS delivered promising outcomes on 21 out of 25 benchmarks, achieving a maximum speedup of 7.26x on GEMM and encountering the largest slowdown (0.41x) on Jacobi.

2.5.5 Application Case Study: Gradient Boosted Decision Trees (GBDT)

While the architectural improvements in software synchronization and runtime tasking provide the necessary foundation for disaggregated execution, their real-world utility is best evaluated through a high-impact, memory-intensive workload. Gradient Boosted Decision Trees (GBDTs) serve as an ideal vehicle for this validation. As a vital machine learning technique for classification and regression, GBDTs are notoriously memory-bound due to the irregular, random memory access patterns required to construct feature histograms.

In traditional systems, these patterns frequently lead to severe cache thrashing and diminished parallel efficiency, often forcing a compromise between the strict capacity limits of local DRAM and the high communication latencies of network-distributed training. By implementing GBDT on CXL-FAM, we aim to demonstrate how a disaggregated, fabric-attached memory pool can alleviate these pressures—effectively transforming an application traditionally limited by the "memory wall" into a scalable, high-performance workload.

2.5.5.1 GBDT Algorithm Overview

Gradient Boosting is an ensemble learning technique that combines multiple "weak" learners—typically simple decision trees—into a single "strong" learner through an iterative process. At each step, a new tree is trained to predict the pseudo-residuals (errors) of the existing ensemble, effectively performing gradient descent in the functional space to minimize a loss function.

Our implementation is based on the XGBoost variant, which enhances the standard GBDT formulation by adding a regularization term to the objective function to prevent overfitting. XGBoost uses a second-order Taylor expansion of the loss function, which requires computing first-order gradients and second-order Hessians for each data point. The training process involves greedily splitting nodes by finding feature values that maximize the gain across sub-trees. This requires building feature histograms (sketches) at every node to estimate optimal split points, a process that is highly memory-intensive and dominates overall performance.

2.5.5.2 Memory Layouts and Optimization Strategies

To evaluate the impact of CXL's direct load/store interface, we explored three distinct memory layouts, each offering different trade-offs in cache utilization and data movement:

Row-based Layout: Implemented as a row-major vector of vectors. Features are stored contiguously within a row, and nodes are split by duplicating data into new vectors. This approach simplifies scanning but increases memory pressure due to duplication.

Column-based Layout: Utilizes a column-major scheme where features are contiguous across rows. This layout enables effective histogram caching during binning, as a single feature column can be processed for all rows in a block, though it still requires data duplication across splits.

Quicksort-based Layout: This layout stores training data in a single, contiguous one-dimensional array, along with an auxiliary index array. Splitting is treated as a quicksort-style pivot operation on the indices rather than the data itself. This approach is highly memory efficient as it avoids data duplication, preserving a single read-only copy of the massive dataset in FAM.

2.5.5.3 GBDT Future Work

The implementation of GBDTs on CXL-FAM demonstrates that disaggregated memory catalyzes a fundamental shift toward in-place, index-manipulation algorithms, such as our Quicksort-based layout, which reduces data movement compared to traditional duplication-heavy patterns. Our current implementation uses OpenMP to parallelize key portions of the application. Our future work will use ARTS and CARTS to evaluate this application's

performance. Ultimately, with this work, we hope to demonstrate that a dataflow-aware runtime and compiler stack can transform memory-bound applications such as GBDT into scalable workloads capable of processing massive datasets previously deemed computationally impractical.

2.6 Data Privacy and Security

Building secure and resilient computational platforms requires integrating cybersecurity principles from the very beginning—covering inception, design, development, and deployment of new machines within the AMAIS portfolio. Proactively identifying vulnerabilities and implementing mitigation strategies at each stage is essential to reducing risk and ensuring compliance with evolving security standards.

This priority becomes even more critical for next-generation architectures that introduce a large, persistent memory tier to support ML acceleration. Such systems enable shared memory access and deliver compute and memory elasticity, but they also expand the attack surface and introduce new vectors for potential exploitation. Addressing these challenges demands a holistic approach that combines robust access controls, continuous monitoring, and adaptive security frameworks to safeguard sensitive data and maintain operational integrity.

2.6.1 GPU Covert and Side Channel Attacks in Memory

Graphics Processing Units (GPUs) have become integral components of computing systems across all scales. In this work, we introduce novel side-channel attacks targeting Graphics DDR (GDDR) memory chips. While prior research has demonstrated similar attacks on CPU memory, these techniques do not directly apply to GPU memory due to fundamental differences in GDDR microarchitecture, GPU memory controllers, and the GPU computational model.

To address this gap, we reverse-engineer the mapping of physical addresses to GDDR bank addresses and show that traditional row buffer timing attacks are ineffective on these systems because of row buffer management policies. Instead, our approach exploits the **Refresh Management (RFM)** feature, originally designed to mitigate Rowhammer vulnerabilities. We identify a timing leakage in RFM where repeated accesses to the same bank trigger refresh events, resulting in measurable variations in access latency.

Leveraging this leakage, we construct covert channel attacks on shared GPUs, achieving bandwidths exceeding **50 KBps per bank** with an error rate as low as **0.03%**. Furthermore, we demonstrate two end-to-end side-channel attacks on discrete GPUs with GDDR6:

Application Fingerprinting

3D Object Rendering Fingerprinting in Blender

These attacks achieve F1 scores of up to **95%** and **98%**, respectively. Additionally, we implement three side-channel attacks on GPU-based SoCs using LPDDR5 memory—application fingerprinting, web fingerprinting, and video fingerprinting—each delivering high F1 scores.

Finally, we present a **Denial-of-Service (DoS)** attack that exploits RFM blocking to degrade application performance by an average factor of **4.8×**.

2.6.2 GPU Covert and Side Channel Attacks in Caches

2.6.2.1 Cross-Component GPU Covert Channels

Graphics Processing Units (GPUs) are now ubiquitous across modern computing platforms—from mobile devices and personal computers to high-end server-class systems. With the growing demand for graphics and video workloads, recent processors often integrate GPUs on the same chip as CPUs. This integration introduces shared resources, creating potential avenues for microarchitectural attacks between the CPU and GPU.

In this work, we examine covert channel attacks that exploit either shared microarchitectural components (such as last-level caches) or shared contention domains (e.g., interconnect buses). We demonstrate two reliable covert channels:

Cache-Based Channel: Leveraging the shared LLC cache in Intel's integrated GPU architectures.

Contention-Based Channel: Exploiting the ring bus connecting the CPU and GPU to the LLC.

This represents the first demonstrated microarchitectural attack crossing the CPU-GPU boundary. Such cross-component channels pose unique challenges due to heterogeneous computation models and asymmetric memory hierarchies. To overcome these, we exploit GPU parallelism to boost communication bandwidth without relying on a common clock. Our cache-based channel achieves 120 kbps with a 2% error rate, while the contention-based channel delivers up to 400 kbps with an error rate of 0.8%. Additionally, we present a proof-of-concept prime-and-probe side-channel attack that probes the full LLC from the GPU.

2.6.2.2 Multi-GPU Systems and Emerging Threats

The deep learning revolution has been powered by GPUs and specialized accelerators, enabling computationally intensive training and inference at scale. As workloads grow, multi-GPU machines have become essential in high-performance computing (HPC) and cloud data centers. These systems are often shared among multiple users, making security a critical concern.

We investigate the vulnerability of Nvidia DGX multi-GPU platforms to covert and side-channel attacks. These machines feature multiple discrete GPUs interconnected via NVLink and PCIe. Through reverse engineering of the cache hierarchy, we show that an attacker on one GPU can induce contention on the L2 cache of another GPU. Using this insight, we develop:

Covert Channel Attack: Achieving up to 3.95 MB/s bandwidth across two GPUs.

Prime-and-Probe Attack: Allowing an attacker to infer cache hit/miss patterns on a remote GPU.

This capability enables a range of side-channel attacks; we demonstrate a proof-of-concept application fingerprinting attack on a remote GPU with high accuracy. Our findings establish, for the first time, that multi-GPU systems are susceptible to microarchitectural attacks, underscoring the need for future research to strengthen their security.

2.6.3 GPU Covert and Side Channel Attacks in Fabrics

Graphics Processing Units (GPUs) are essential accelerators across a wide range of applications, including computer vision, Extended Reality (XR), language models, healthcare, and more. As datasets continue to grow, experiments on a single GPU can span days, driving widespread adoption of multi-GPU systems to accelerate these workloads. In parallel, security and privacy concerns for both CPUs and GPUs have gained increasing attention. Conventional I/O interconnects such as PCIe have already been shown to be susceptible to side-channel attacks.

This work demonstrates that modern high-speed **multi-GPU interconnects** are likewise vulnerable to **covert** and **side-channel** attacks arising from **congestion** on these links. By monitoring data transfer rates over the shared **NVLink** protocol, adversaries can infer information about other users' activity. Building on this observation, we design a **covert channel** spanning two GPUs that achieves **notable bandwidth**, and we introduce a **side-channel attack** that enables profiling of applications executing across multiple GPUs.

In this **Wild and Emerging Ideas (WEI)** paper, we present initial results of these attacks on Nvidia multi-GPU systems. To our knowledge, this is the **first attack** on multi-GPU interconnects that exploits **congestion-based timing leakages**.

Implications for Fabric-Attached Memory

The vulnerabilities identified in multi-GPU interconnects have direct implications for emerging **fabric-attached memory (FAM)** architectures. FAM systems rely on high-speed fabrics to provide disaggregated memory resources across multiple compute nodes, often sharing interconnects like NVLink or custom fabrics. Congestion-based timing leakages in these fabrics could enable attackers to infer memory access patterns, application behavior, or even sensitive data movement across nodes. As FAM becomes a cornerstone for scalable AI and HPC platforms, securing these fabrics against covert and side-channel attacks is critical. This would require proactive measures such as traffic obfuscation, isolation policies, and hardware-level QoS enforcement to prevent information leakage through shared interconnects.

2.6.4 Harnessing ML Privacy via Memory Crossbars

Deep Learning, widely applied across diverse domains, relies on powerful architectures such as Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs). Specialized accelerators have been developed to handle the compute-intensive Multiplication-and-Accumulation (MAC) operations inherent in these models. However, the limitations of conventional digital CMOS hardware have driven interest in **Non-Volatile Memory (NVM)-based accelerators**, particularly **Resistive RAM (RRAM)**.

RRAM, organized in crossbar arrays, enables in-memory MAC computations, significantly reducing data movement and delivering substantial time and energy savings for DNN workloads. In this study, we empirically examine RRAM crossbars, which perform analog computations but exhibit inherent non-idealities due to device characteristics, access transistors, and parasitic resistances. These imperfections lead to approximate outputs, posing reliability challenges for general-purpose applications. Interestingly, DNNs and CNNs are highly fault-tolerant, making them well-suited for acceleration through in-situ computing with RRAM crossbars.

Recent research suggests that beyond performance gains, RRAM crossbars may also enhance **robustness against adversarial examples**. Our work investigates whether these non-idealities can improve resilience against **privacy attacks**, specifically **Membership Inference Attacks (MIAs)**, which aim to extract sensitive information from trained models. Given the growing use of sensitive datasets—such as genomic data, personal images, clinical records, and location traces—and the prevalence of cloud-based training, ensuring privacy in machine learning systems is critical.

Traditional defenses like **differential privacy** and **output obfuscation** often degrade model utility and require retraining. In contrast, RRAM crossbars inherently disrupt the statistical features exploited by privacy attacks, providing privacy benefits without additional computation or hardware constraints. Using the **PytorX framework**, this research represents the first exploration of **machine learning privacy through RRAM crossbar implementation**, offering a novel pathway toward trustworthy and efficient AI systems.

2.6.5 Examining ML vulnerabilities

2.6.5.1 Advancing Chart Understanding in Vision–Language Models

Chart interpretation remains a critical yet underexplored challenge for Vision–Language Models (VLMs). Existing benchmarks typically assess performance on clean, well-structured figures, but they fail to reflect real-world complexity where charts may be corrupted, partially occluded, or paired with reasoning-intensive questions.

To address this gap, we introduce a new evaluation framework that systematically tests VLM robustness under chart distortions and reasoning-heavy question formats. Unlike prior benchmarks focused on surface-level question answering, our design incorporates multiple-choice, exam-style questions inspired by Korea's CSAT English section. These questions

explicitly require identifying incorrect statements about charts, compelling models to move beyond simple fact matching and demonstrate precise reasoning—even under visual degradation.

Our contributions are threefold:

- Benchmarking State-of-the-Art VLMs, revealing systematic hallucination patterns such as fabricated values and trend misinterpretation under degraded conditions.
- Introducing CHART NOISe (Chart Hallucinations, Answers, and Reasoning Testing on Noisy and Occluded Input Selections)—a dataset of 1,500 CSAT-style chart questions paired with diverse corruptions and occlusions, enabling controlled robustness analysis.
- Proposing Baseline Mitigation Strategies, including front-end quality filtering and occlusion detection, to reduce reasoning errors.

Together, these efforts establish CHART NOISe as the first rigorous testbed for evaluating chart reasoning robustness in real-world scenarios.

Implications for Enterprise Analytics and Decision-Making
For organizations that rely on automated chart interpretation—such as government, financial analytics, operational dashboards, and business intelligence systems—this research highlights critical vulnerabilities in current VLM-based solutions. Inaccurate reasoning under noisy or incomplete visual data can lead to flawed insights and poor strategic decisions. By adopting evaluation frameworks like CHART NOISe, enterprises can better assess model reliability, implement robustness safeguards, and ensure that AI-driven analytics maintain accuracy even in imperfect real-world conditions. This is essential for compliance, risk management, and maintaining trust in data-driven decision-making.

2.6.5.2 Detecting Harmonized Image Edits in the Era of Generative AI

The rapid evolution of deep learning-based content generation and manipulation techniques—collectively known as **Generative AI**—has fueled the development of tools capable of producing highly realistic images and seamlessly editing existing ones. As these technologies advance, distinguishing authentic images from AI-generated or edited content becomes increasingly difficult, raising critical concerns for **digital forensics**, **media authenticity**, and **content verification**.

A particularly challenging class of manipulations involves the **insertion and harmonization of objects** within images. Harmonization ensures that inserted objects blend naturally with their surroundings, making detection by traditional forensic models extremely difficult—especially when lighting and texture adjustments are skillfully applied. While harmonization has historically been a complex task, modern deep-learning approaches have made it highly effective at fooling the human eye. However, these methods may still leave subtle artifacts, as observed in other editing techniques.

To address this gap, we propose **Disharmony**, a Harmonization Detection Model specifically designed to identify image edits based on **lighting inconsistencies**. Our study targets edits such as object insertion followed by harmonization, manual photo manipulation using tools like Photoshop, and edits generated by advanced deep learning models, including diffusion-based techniques.

Our contributions are as follows:

- Evaluation of Existing Harmonization Methods: We tested current forensic models on harmonized edits and confirmed their limitations, reinforcing the need for specialized detection approaches.

- Aggregated Training for Robust Detection: We trained a segmentation model on a dataset combining three harmonization techniques—neural network-based light adjustment, physics-based methods, and handcrafted approaches—achieving significant improvements over existing forensic models.

- Sensitivity Analysis and Generalization: We conducted a sensitivity study demonstrating that Disharmony extends beyond the harmonization methods used in training, offering broad applicability for detecting diverse image edits.

This research introduces a novel solution to the growing challenge posed by sophisticated Generative AI capabilities, marking an important step toward reliable forensic detection of harmonized image edits.

Implications for Enterprise Security and Compliance
For enterprises, the ability to detect harmonized image edits is critical for **preventing misinformation**, **protecting brand integrity**, and ensuring compliance with regulatory frameworks governing digital content authenticity. In sectors such as government, finance, healthcare, and media, undetected image manipulations can lead to reputational damage, legal exposure, and operational risks. By integrating detection models like Disharmony into content verification pipelines, organizations can strengthen **trust in digital assets**, support compliance with standards such as **GDPR** and **ISO/IEC 27001** and mitigate the risk of AI-driven disinformation campaigns. This capability is essential for maintaining transparency and accountability in enterprise workflows.

2.7 Theory

2.7.1 Isoefficiency Metric

The shift from traditional, siloed data center architectures to decoupled, pooled memory systems—specifically Fabric Attached Memory (FAM) and Compute Express Link (CXL)—represents a tectonic shift in high-performance computing (HPC) and AI infrastructure. Historically, the tight coupling of compute and memory has resulted in memory stranding, where a large fraction of DRAM remains underutilized because its host CPU is compute-bound. This inefficiency represents a massive drain on the Total Cost of Ownership (TCO). Memory disaggregation has emerged as an attractive path to mitigate these challenges through efficient pooling of memories. The cornerstone of this transition is CXL 3.0, a PCIe-based protocol that facilitates cache-coherent memory pooling. By allowing heterogeneous compute nodes to access a shared DRAM pool with DRAM-like latencies, CXL enables the dynamic scaling of memory capacity and bandwidth.

The performance delta between local DRAM and fabric-attached memory necessitates a rigorous approach to modeling the trade-offs between computation and data movement. Thus, adoption of memory pooling faces a strategic challenge for the lack of mature analytical tools. Without robust frameworks to model performance, multi-million-dollar investments in CXL-enabled hardware are being provisioned without analysis, risking failure to meet return on investment targets due to unanticipated data movement overheads. To resolve these architectural uncertainties, we must look beyond basic speedup metrics. This synthesis evaluates two fundamental frameworks: *Isoefficiency*, which provides a macro-level scalability metric for algorithm-architecture combinations, and *Red-Blue Pebble Game modeling*, which provides the granular, task-based insight required to manage data movement in multi-tenant environments.

2.7.1.1 Isoefficiency as a Scalability Metric for Disaggregated Systems

In disaggregated systems, traditional speedup metrics fail because they do not account for the necessary growth in problem size required to amortize the increased latency of remote memory access. *Isoefficiency* provides a unified metric that balances hardware parameters (e.g., CXL interconnect speeds), algorithmic design, and input characteristics. Intuitively, isoefficiency measures the rate at which the input sizes need to grow relative to the growth in computing resources to keep parallel efficiency constant. It measures the degree to which a problem size must increase relative to the number of processors to maintain a constant efficiency. We posit that isoefficiency is a better metric to evaluate systems and system designs with disaggregated memory. Isoefficiency provides a consistent metric to measure the relative performances of different systems across three major components: (*i*) Hardware parameters and system (or design) configurations, (*ii*) algorithmic design choices (computation, communication, etc.), and (*iii*) input characteristics. Relative performances can be measured with a single metric while varying across different components when the systems scale with number of processing units and other components. Since isoefficiency focuses on the scalability of a system when computing resources are added to the system, it provides a unified metric for relative comparisons.

Introduced by Grama et al., isoefficiency evaluates the scalability of parallel systems and algorithms by analyzing the required growth in problem size N relative to the number of processors P to maintain a constant efficiency E. Intuitively, a proportional growth in work with growth in number of processors is considered as an isoefficient system. Efficiency E is given as the ratio of the speedup S to the number of processors P, where speedup S is the ratio of the execution time on a single processor T_s to the execution time on p processors T_p. The isoefficiency $E(p)$ determines the rate at which the problem size must increase to keep efficiency a constant, as described by the equation $N = K.E\ (p)$, where K is constant. This function provides insights on the scalability of a parallel system. We provide a hypothetical illustration for a problem that scales with number of processors in Figure 23. The three lines in the figure depict when efficiency scales linearly, super-linearly and sub-linearly.

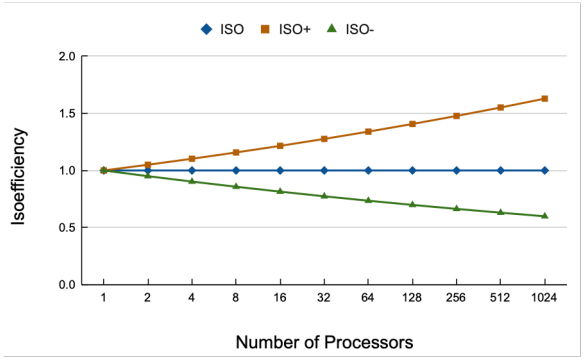


Figure 23 Isoefficiency -- perfect (blue), 10% super-efficient (orange) and 10% sub-efficient (green)

Empirical Evaluation: We conducted evaluation using sparse general matrix–matrix multiplication (SpGEMM) as the computational kernel. For enabling different memory footprints, we use R-MAT as the generative model for building the matrices. We performed experiments on a traditional 2-socket shared-memory system and a simulator for disaggregated memory system. Our goal was to select an algorithm that would not only stress the memory system but also include sufficient floating-point operations. We therefore selected sparse general matrix–matrix multiplication (SpGEMM) as the first algorithm for demonstrating that isoefficiency is a viable metric for evaluating systems. Since our goal is not to implement the best algorithm for SpGEMM, we adopt a simplistic but work-efficient approach for SpGEMM using the combinatorial approach of Brualdi and Cvetkovic.

Given two sparse matrices A and B, we build a special data structure called a König digraph, $G = (W \cup R \cup K,\ E_1 \cup E_2)$. Each row in A is represented with a "white" vertex (W) in G. The "gray" vertices (R) represent both the columns of A and rows of B, and the "black" vertices (K) represent the columns in B. The nonzero elements in A and B are represented by directed and weighted edges in E_1 and E_2 respectively. A simple 3 × 3 matrix example is provided in Figure 24. The nonzero values in $C = A \cdot B$ can be computed by performing two-hop walks starting from vertices in W and ending with vertices in K. For example, the nonzero value for $C\ (1,\ 2)$ in Figure 24, will consist of a traversal from $w_1 \rightarrow r_1 \rightarrow k_2$ and $w_1 \rightarrow r_3 \rightarrow k_2$. The time complexity of this algorithm can be expressed as $O\ (n\Delta^2)$, where n is the number of rows in A and Δ is the maximum degree for vertices in W and R. While this is a loose bound, we note that the amount of work performed is proportional to the number of nonzero in A and B.

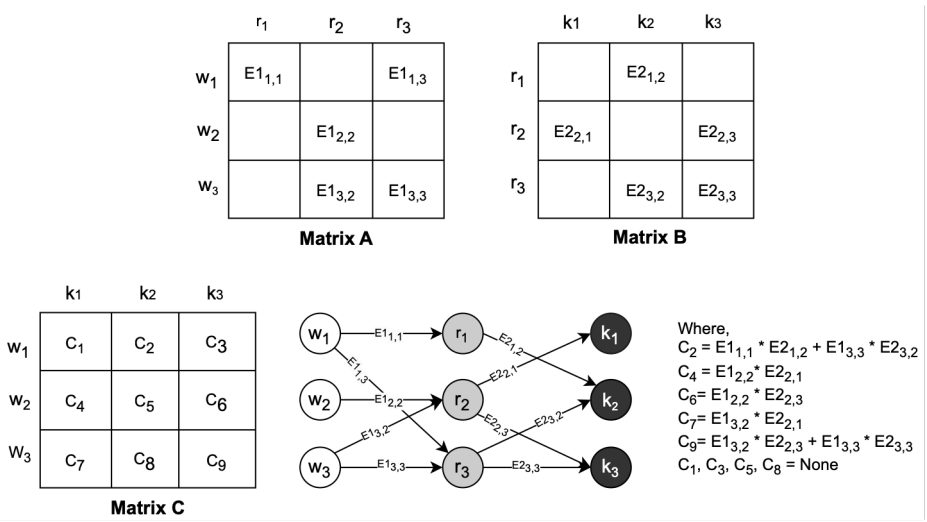


Figure 24 Sparse general matrix-matrix multiplication (SpGEMM) using the combinatorial approach

We implemented SpGEMM in C++ using OpenMP with all data structures using 64-bit precision. We use a compressed sparse representation for Konig digraph˘ G, which provides efficient access to nonzero for each row and column in A and B respectively. Unlike a serial implementation, parallel SpGEMM is nontrivial without first constructing the nonzero in C, assuming that the structure of C is dense (either fully or in blocks). In our implementation, we first perform a symbolic walk to determine the exact location of each non-zero value in C and then perform the actual computation. Due to the nature of our implementation, which is not highly optimized, the construction of C takes considerably more time than calculating the values of C and does not scale well. The 2-socket shared-memory system that we used in this work is comprised of AMD EPYC 7282 16-core processor, utilizing the x86_64 architecture. It supports both 32-bit and 64-bit operation modes with a little-endian byte order. The system has 32 CPUs, organized as 16 cores per socket across 2 sockets, with each core operating on single thread, and two NUMA nodes (node 0: CPUs 0-15, node 1: CPUs 16-31). The processor runs at a frequency of approximately 2.8 GHz with a BogoMIPS value of 5589.58 and supports AMD-V virtualization technology. The memory hierarchy includes 32KB of L1 data- and L1 instruction-cache per core, 512Kb of L2 cache per core, and 16MB of shared L3 cache.

We present the compute time for two specific steps: (i) Computing the structure of matrix C, and (ii) computing the values of matrix C. Further optimizations are required for computing the structure of C. As we observe, parallel efficiency number are very low indicating that this step does not scale well. We also note that we do not explicitly construct the König digraph since matrices A and B are stored in row-wise and column-wise formats. Computing the nonzero values of matrix C after it's compressed sparse structure is built scales well proportional to the number of processors and doubling of the number of problem size. To double the amount of work, we double the number of nonzero in matrices A and B. We note that the number of nonzero is proportional to the matrix size (we use a fixed ration of 8× as noted in §4.2). By

increasing the value of $SCALE$ by one, we double the matrix size (2^{SCALE}), and consequently, the problem size. Similarly, by increasing the value of $SCALE$ by two quadruples the work. We note that by using the probability values of (0.25, 0.25, 0.25, 0.25) for R-MAT generator, we expect to have a normal distribution for the number of nonzero for each row (and column).

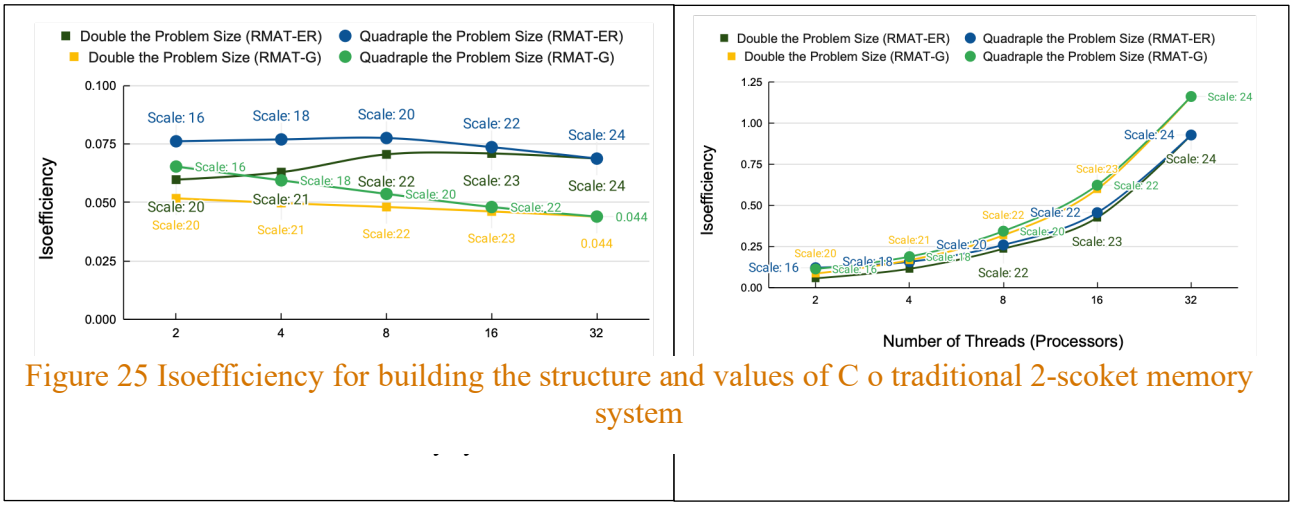


Figure 25 Isoefficiency for building the structure and values of C o traditional 2-scoket memory system

We observe in Figure 25-left that building the structure of C is not an efficient due to memory allocation and irregular memory accesses. Since there are no computations, this step also lacks efficient approach to amortize memory access costs. We can also observe that quadrupling the problem sizes is more efficient than doubling the problem sizes. Further, RMAT-ER scales better than RMAT-G since load imbalances are expected to be smaller for RMATER (due to differences in degree distributions). In contrast, calculating the nonzero values after the structure has been built parallelizes efficiently. We observe better scaling for larger problems ($SCALE \geq$ 22), as summarized in Figure 25-right. Further, we do not observe noticeable difference between RMAT-ER and RMAT-G. We can observe the stark differences in the isoefficiency values for the two kernels – building structure and computing values. To study the impact on a disaggregated memory system, we used a state-of-the-art simulator that we describe next. We refer you to Develapalli et al. for further details on experimental results.

Simulation Results: We conducted empirical evaluation using the DRackSim simulator. Although DRackSim can simulate a large-scale system, we configure it for one compute and remote memory node. The compute node models a multi-socket CPU system with an out-of-order pipeline at compute cores and a multi-level cache hierarchy for CPU-driven memory simulation. The memory manager is configured to allocate memory at a 50:50 ratio in local and remote memory, alternatively for each page. In the simulation, the remote memory node is connected to the compute node through a 10, 30 or 50 Gbps interconnect having 35ns of total latency for (de)packetization/processing. The interconnect uses a queue-based modeling that takes 64-byte memory request packets from the compute node and forwards them to the memory node. In a large-scale CXL3.0-enabled system, a compute node will only get a small share of the switch bandwidth. Therefore, we perform simulations at multiple interconnect bandwidths to validate the proposed metric for changing resource availability in a real system. However, the remote memory access at the memory node is always performed at cache line granularity (64B), which sends a 128-byte response packet, including cache line data. The DRAM simulation at local and remote memory is handled by integrating DRAMSim2, which instantiates multiple 2400MHz DDR4 (19.2Gbps) components each for local and remote memory. The thread context switching time is not considered in the simulator.

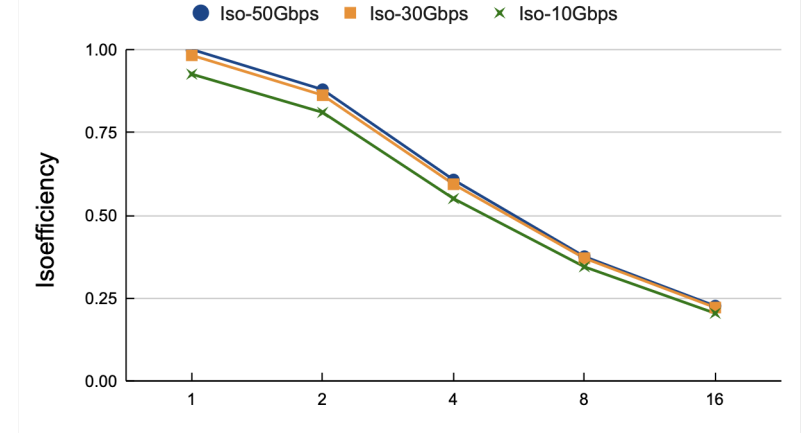


Figure 26 Isoefficiency calculated for computing the structure of C using the DRackSim simulator for different network bandwidths.

Since we had difficulties in simulating complete SpGEMM execution, we only present the results for computing the structure of *C* in this section. Results for two different settings of the network bandwidth are summarized in Table 3. The isoefficiency values for three different bandwidth settings are presented in Figure 26. Since the numbers for 50 Gbps and 30 Gbps are very similar, we omit them from Table 3. We can observe that the isoefficiency values successively decrease, and therefore, indicate that the growth in problem size needs to be more than linear to maintain a constant isoefficiency.

Table 3 Results from DRackSim simulator. Compute time is calculated in millions of cycles

#Threads	SCALE	M-Cycles	Isoefficiency
50Gbps			
1	10	112	1
2	11	153	0.88
4	12	224	0.61
8	13	369	0.38
16	14	650	0.23
10 Gbps			
1	10	121	1
2	11	166	0.36
4	12	247	0.12
8	13	402	0.04
16	14	720	0.01

We posit that isoefficiency provides a *unified metric* for evaluating scalability of a system along different aspects: algorithms, input, and hardware features. However, further evidence using larger systems is required to conclusively show the use of isoefficiency as a unified metric for evaluating large, disaggregated memory systems. In the next section, we continue the discussion on analytical models for evaluation using the popular red-blue pebble game framework, which we also propose as a practical approach for scheduling memory movement in disaggregated memory systems.

2.7.2 Multi-level Pebble Games for Modeling Disaggregated-Memory Systems

Analytical models to predict the performance of an application under different configurations of local and remote memory will enable the design of efficient disaggregated memory systems, as well as help in accurately benchmarking experimental systems. Towards this end, we propose the Red-Orange-Blue Pebble (ROBP) game, an adaptation of the Red-Blue Pebble game, to model disaggregated-memory systems, where orange pebbles represent pooled (fabric-attached) memory. We evaluate ROBP using a greedy algorithm with approximation guarantees that prioritizes computation-ready nodes while minimizing memory transfers and present a novel reinforcement learning (RL)-based scheduler that dynamically optimizes pebbling strategies to reduce memory transitions. We conducted extensive empirical evaluation using representative applications of increasing complexity and demonstrate that the greedy approach is computationally efficient to evaluate a broad set of real-world scenarios, and that the RL-based algorithm adapts to workload patterns, improving memory efficiency in disaggregated systems. The lower theoretical bounds established by the pebble games for simplistic workloads, and the empirical results for complex workloads, will provide critical insights to system designers as well as application developers in adopting disaggregated memory technologies.

Red-Blue Pebble Game: The Red-Blue Pebble Game, introduced by Hong and Kung, is a theoretical framework used to analyze trade-offs between computation and memory usage in DAG-based algorithms. It models computation in a system with:

- P1: A limited number of red pebbles, representing fast memory (e.g., cache or local memory)

- P2: An unlimited number of blue pebbles, representing slow memory (e.g., remote or secondary storage)

The game is played on a DAG, where a vertex can only be computed (i.e., executed) if all its immediate predecessors have red pebbles. The objective is to compute all vertices in the DAG using the fewest possible memory transfers while respecting memory constraints.

The standard Red-Blue Pebble Game follows these rules:

- R1 (Input) A blue pebble can be placed on any vertex at any time.
- R2 (Computation) A vertex can be computed if all its predecessors have red pebbles.
- R3 (Memory Placement) A red pebble can be placed on any vertex that already has a blue pebble.
- R4 (Output) A blue pebble can be placed on any vertex that has a red pebble.
- R5 (Deletion) A red or blue pebble can be removed from any vertex at any time.

A transition is an ordered pair of configurations, where the second configuration follows from the first using one of the rules. An execution is a sequence of configurations where each successive pair forms a transition. A complete execution of the DAG begins with the initial configuration and ends with the terminal configuration. By analyzing pebble placement and transitions, we can derive lower bounds on the I/O complexity of different algorithms, offering insights into memory-efficient execution strategies.

The Red-Blue Pebble Game, introduced by Hong and Kung, provides a theoretical framework to analyze I/O complexity in DAG-based computations by modeling the trade-off between computation and memory usage. It establishes lower bounds on memory transfers for various algorithms, such as FFT (Ω (n log$n$/log S)) and matrix multiplication ($\Omega(n3/S)$), helping optimize memory constrained execution. Demaine and Liu studied the complexity of computing optimal cache size vs. memory transfer trade-offs, showing the computational hardness of optimizing red-blue pebbling. Gleinig and Hoefler further extended the model to large DAGs, refining I/O lower bounds for large-scale computations. Recent studies have focused on extending red-blue pebbling to multi-processor and distributed computing models. Kwasniewski et al. revisited parallel matrix-matrix multiplication, demonstrating how optimized pebbling strategies can significantly reduce communication costs in HPC environments. Zhang et al. applied red-blue pebbling to communication lower bounds in CNNs, highlighting its role in optimizing deep learning workloads. Böhnlein et al. generalized red-blue pebbling to multi-processor architectures, proposing a greedy heuristic to balance memory usage, computation, and communication. In this work, we adapt their greedy scheduling strategy for single-processor execution and extend it to the Red-Orange-Blue Pebble (ROBP) game, explicitly incorporating fabric-attached pooled memory as a distinct tier. Table 4 provides a summary of key works on this topic. To the best of our knowledge, our work is the first such effort for emerging disaggregated-memory systems that are poised to see increasing adoption.

Table 4 Previous studies that analyze red-blue pebble-game formulations

Paper	2-level/ multi-level	Processors (Single/Multiple)	Algorithm Games	Application Diversity	Empirical Evaluation
Jia-Wei and Kung	2-level	Single	S-paritioning for I/O lower bounds	FFT, matrix multiplication, sorting networks, snake-like mesh	None
Savage	Multi-level	Single	S-partition across L levels of memory	General computations	None
Elango et al.	Multi-level	Multiple	S-partitioning for red-bluewhite	CDAGs for matrix ops, FFT, composite loops	Theoretical (decomposition and case analysis)
Carpenter et al.	2-level, multi-level	Single	TSP-based approximation algorithm	Diamond graphs, Jacobi graphs	I/O cost vs pebbles
Liu	2-level	Both	No algorithm	Trees, pyramid DAGs, gadgets	Theoretical proofs
Demaine and Liu	2-level	Single	No algorithm	General DAGs (gadgets)	Theoretical bounds using SAT reduction
Kwasniewski et al.	2-level	Multiple	Greedy X-partitions	Matrix-Matrix Multiplication	I/O reduction and speedup with peak performances
Papp and Wattenhofer	2-level	Single	Greedy heuristics (no concrete strategy)	Gadgets, grid DAGs	Theoretical proofs
Zhang et al.	2-level	Single	Pseudocode for I/O lower bounds	CNN convolutions	None
Gleinig and Hoefler	2-level	Multiple	Proposed I/O rules	Trees and Random DAGs	No.of I/Os vs memory size and no.of vertices (tightness and linear scaling)
Bilardi and Stefani	2-level	Single	Boundary complexity	q-pyramids, q-trees	No (only theoretical bounds)
Böhnlein et al.	2-level	Multiple	Greedy	No specific details	Theoretical bounds (no experimental result)
Sobczyk	2-level	Single	Partial computations (approximation and hardness)	Graphs, sparse arithmetic circuits, matrix-vector operations	None

Red-Orange-Blue (Ours)	Multi-level	Single	Greedy, RL	Parallel prefix sum, FFT, Sparse matrix-matrix DAGs, Deep learning Model CDAGs (Fine and Coarse)	Optimal Moves, I/O cost

Red-Orange-Blue Pebble Game: We propose an extension to this game by also adding a limited number of orange pebbles representing pooled (fabric attached) memory (Figure 27). The ROBP game adds additional rules to the RBP game for movement if data between red and orange, and orange and blue nodes, to represent disaggregated memory. For the proposed red-orange-blue pebble (ROBP) game, we propose the following rules:

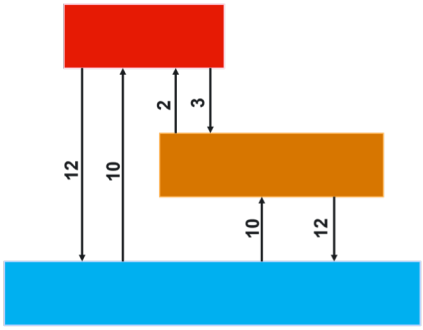


Figure 27 Multi-Level memory hierarchy and data movement costs in FAM. Assuming similar costs moving Blue to Red and Blue to Orange, smaller costs Orange to Red. Dissimilar costs Read vs Write.

- R1 (Input): Assign blue pebbles on all source nodes or inputs (vertices with no predecessors).
- R2 (Movement): An orange pebble can be placed on any vertex that already has a blue pebble.
- R3 (Movement): A red pebble can be placed on any vertex that already has a blue or an orange pebble.
- R4 (Computation): A vertex can be computed if all its immediate predecessors have red pebbles.
- R5 (Movement): An orange pebble can be placed on any vertex that has a red pebble.
- R6 (Output): A blue pebble can be placed on any vertex that has a red pebble.
- R7 (Termination): The game terminates when no further moves can be made, or when all destination nodes (vertices with no successors) are computed and placed in memory.

Scheduling Algorithms: To analyze and optimize memory usage in disaggregated-memory systems, we employ two distinct scheduling strategies for the Red-Orange-Blue Pebble Game (ROBPG): a greedy heuristic-based approach and a reinforcement learning (RL)-based adaptive scheduling method. The greedy scheduling algorithm, adapted from Böhnlein et al., follows a priority-based heuristic to minimize memory transfers by efficiently placing and transitioning pebbles. It prioritizes computation-ready nodes within a directed acyclic graph (DAG) using a min-heap, where nodes are ranked by resolved dependencies and available red pebbles. In cases of ties, the heuristic favors deeper nodes in the DAG to prevent future recomputation. When

resource constraints arise, the algorithm strategically converts pebbles between red, orange, and blue to optimize memory utilization and computational efficiency.

In contrast, our RL-based approach formulates the pebbling problem as a Markov Decision Process (MDP), where an agent learns optimal scheduling strategies through Q-learning. The state representation encodes the current memory allocation (red, orange, and blue pebbles on vertices), while the action space consists of placing, removing, and transitioning pebbles between memory tiers. The agent learns an optimal scheduling policy by interacting with the environment, where actions are selected using an ϵ-greedy policy that balances exploration (random actions) and exploitation (choosing the best-known action based on Q-values). Rewards are assigned based on memory efficiency, with penalties for excessive memory transitions and positive rewards for completing computations with minimal overhead. The Q-values are updated iteratively using the Bellman equation (Algorithm 2), refining the agent's decisions over multiple training episodes.

While the greedy scheduler is computationally lightweight and provides fast scheduling, it lacks the ability to adapt to workload variations and dynamic memory constraints. In contrast, the RLbased method is more adaptive but incurs a higher computational cost during training, requiring multiple episodes to converge. In practice, the greedy approach is preferable for structured DAGs with predictable memory access patterns, whereas the RL-based scheduler performs better in scenarios with complex dependencies and dynamic memory constraints.

Algorithm 1 Greedy Pebble Game Scheduling (RBPG & ROBPG)

Require: Directed Acyclic Graph (DAG) $G = (V, E)$
Require: Maximum red pebbles R_{max}, Maximum orange pebbles O_{max} (for ROBPG)
Ensure: All $v \in V$ are computed with minimal memory transfers

```
Initialize source nodes with blue pebbles
Initialize min-heap H with nodes based on priority
while H is not empty do
    Extract highest-priority node v from H
    if all predecessors of v have red pebbles then
        if |Red_Pebbles| < R_max then
            Place a red pebble on v
        else if ROBPG and |Orange_Pebbles| < O_max then
            Convert an unused red pebble to orange
            Place a red pebble on v
        else
            Convert an unused orange pebble to blue
        end if
    end if
    Update dependencies and push next eligible nodes into H
end while
Finalize all sink nodes by converting red/orange to blue
```

Algorithm 2 Reinforcement Learning-Based Pebble Scheduling (RBPG & ROBPG)

Require: Directed Acyclic Graph (DAG) $G = (V, E)$, Maximum red pebbles R_{max}, Maximum orange pebbles O_{max} (ROBPG), Episodes E
Ensure: Trained RL agent learns optimal pebbling strategy with minimal transitions

```
Initialize Q-table, exploration rate ε, learning rate α, discount factor γ
for each episode in E do
    Initialize environment state S (all nodes uncomputed)
    while not all nodes computed do
        Select action A using ε-greedy policy:
    • Explore (random action) with probability ε
    • Exploit (best Q-value action) otherwise
        Apply action A to update S → S'
        Compute reward R based on:
    • Negative cost for memory moves (pebble transitions)
    • Large positive reward if all nodes are computed successfully
        Update Q-value using Bellman equation:
    Q(S,A) ← Q(S,A) + α (R + γ max Q(S',A') − Q(S,A))
        Set S ← S'
    end while
    Decay exploration rate ε ← max(ε_min, ε · decay)
end for
Return trained Q-table with optimal actions
```

Empirical Evaluation: We selected five representative workloads with increasing complexity: Parallel Prefix Sum, Fast Fourier Transform (FFT), Sparse Matrix-Matrix Multiplication (SpMM), and a Deep Learning Model (ResNet), evaluated in both coarse-grained and fine-

grained variants. All workloads were executed under a simulation setting utilizing Python implementations of both the Red-Blue and Red-Orange-Blue pebbling models, employing a greedy approach as described in the methodology. For each experiment, the theoretical minimum number of red pebbles (min_red_pebbles) required was computed based on the maximum in-degree of nodes within the DAG plus one, following the method outlined in previous work. However, due to the limitations inherent in the greedy strategy, the actual number of red pebbles utilized—termed "working red pebbles"—often exceeded this minimum. These working red pebbles were determined empirically through iterative trial-and-error. Additionally, we compute the total number of moves required for each scenario and compute the associated costs according to the predefined cost metrics detailed in Figure 3.1. These metrics provide a quantitative basis for assessing and comparing the efficiency and performance trade-offs inherent in each pebbling strategy across all evaluated workloads.

We refer the reader to Devulapalli et al. for detailed presentation of experimental results and only provide a couple of examples in this report. Total moves (an approximation of the optimal moves) for the parallel prefix sum DAGs using both Red-Blue greedy and Red-Orange-Blue greedy pebbling algorithms and the corresponding working red pebble counts for each problem size are summarized in Table 5. Thus, using prefix sum as a simple workload, we observe that a minimum amount of disaggregated memory is required to obtain performance gains relative to traditional systems. Similarly, the total move counts for the Fast Fourier Transform (FFT) workloads and the corresponding working red pebble configurations for each problem size are provided in Table 6.

Table 5 Total number of moves computed using the greedy algorithm for different parallel prefix sum DAGs

Different Configurations	Problem Size														
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
No.of Nodes	3	7	15	31	63	127	255	511	1023	2047	4095	8191	16383	32767	65535
No.of Edges	2	6	14	30	62	126	254	510	1022	2046	4094	8190	16383	32766	65534
Min Working Red Pebbles	3	4	5	6	8	10	12	15	18	21	22	27	31	35	42
RB with Min Reds	4	11	26	57	119	245	499	1008	2029	4074	8169	16356	32736	65500	131029
RB with 2x Min Reds	4	8	21	51	111	235	487	993	2011	4053	8147	16329	32705	65465	130987
ROB Min Red + 64x Red for Orange	4	11	26	57	119	245	499	1008	2029	4756	10834	22792	47104	95992	193834
ROB Min Red + 128x Red for Orange	4	11	26	57	119	245	499	1008	2029	4074	9426	21064	45120	93752	191146
ROB Min Red + 256x Red for Orange	4	11	26	57	119	245	499	1008	2029	4074	8169	17608	41152	89272	185770
ROB Min Red + 512x Red for Orange	4	11	26	57	119	245	499	1008	2029	4074	8169	16356	33216	80312	175018

Table 6 Total number of moves computed using the greedy algorithm for different FFT DAGs

Different Configurations	Problem Size											
	2	4	8	16	32	64	128	256	512	1024	2048	4096
No.of Nodes	4	12	32	80	192	448	1024	2304	5120	11264	24576	53248
No.of Edges	4	16	48	128	320	768	1792	4096	9216	20480	45056	98304
Min Working Red Pebbles	3	6	11	21	41	81	146	275	634	1281	2179	4169
RB with Min Reds	6	19	54	140	344	816	1903	4334	9607	21248	46974	102328
RB with 2x Min Reds	6	16	43	119	303	735	1757	4059	8973	19967	44795	98159
ROB Min Red + 1x Red for Orange	6	19	57	163	423	1039	2508	5833	12948	28927	65145	143143
ROB Min Red + 4x Red for Orange	6	19	54	140	344	816	2070	5008	11046	25084	58608	130636
ROB Min Red + 8x Red for Orange	6	19	54	140	344	816	1903	4334	9607	21248	46974	113960

Across all evaluated workloads—Parallel Prefix Sum, FFT, SpMM, and ResNet models (both coarse- and fine-grained), the greedy pebbling approach demonstrated clear trends. The Red-Blue strategy with increased red pebble counts consistently achieves the lowest total cost due to reduced transitions (data movement). However, the Red-Orange-Blue approach, while relying on fewer red pebbles, shows significant cost reductions as orange pebble allocations increase. Yet, for most workloads, even large allocations of orange pebbles (e.g., 512×) only approach but do not fully match the efficiency of Red-Blue configurations with 2× red pebbles. This highlights a key limitation of the greedy approach: while computationally efficient, it does not optimally balance transitions and pebble moves, especially under constrained red pebble conditions. The magnitude of cost reduction varies across workloads, with fine-grained models benefiting more from increased pebble flexibility, while irregular workloads like SpMM remain more sensitive to pebble allocations. These insights establish the greedy strategy as a practical baseline for evaluating pebbling efficiency across diverse computational graphs. We aim to enhance the reinforcement learning based algorithm in our future work.

3.0 Compute Platforms

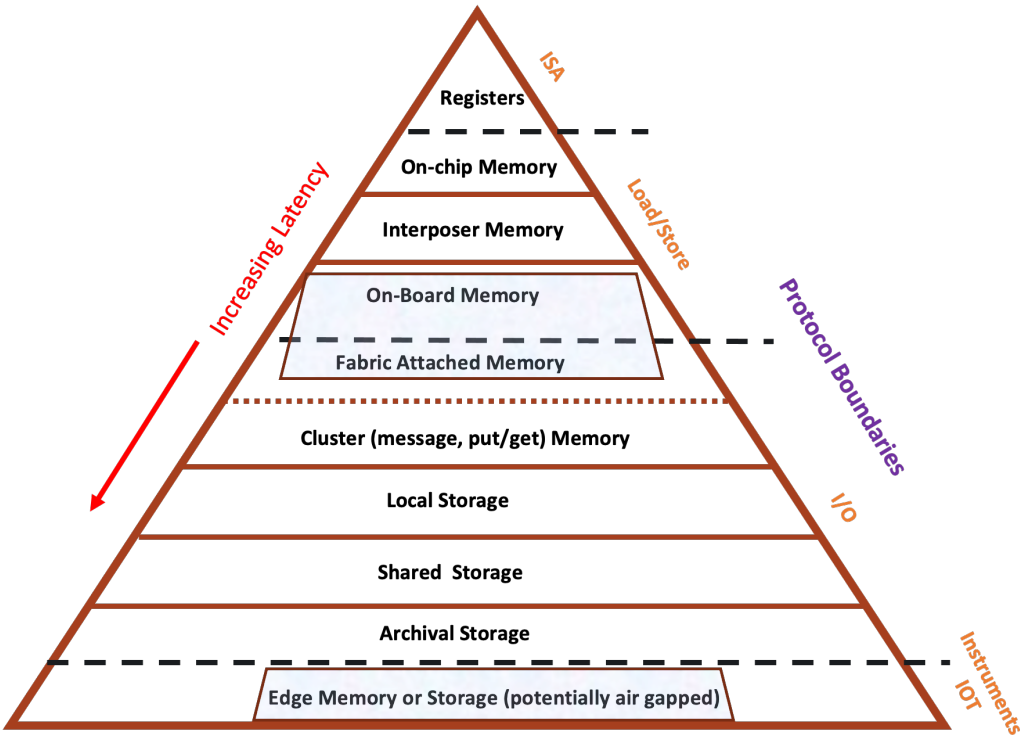


Figure 28 Register/Memory Hierarchy and its Protocols

3.1 Fabric-Attached Memory and CXL: A Paradigm for Scalable Computing

Fabric-Attached Memory (FAM) represents a fundamental shift in memory architecture, moving beyond traditional Actor-centric designs to enable disaggregated memory pools accessible across multiple compute nodes. Instead of memory being physically tied to a single processor, FAM leverages high-speed interconnects to provide shared, elastic memory resources that can be dynamically allocated based on workload demands. This approach is particularly valuable for large-scale AI, HPC, and cloud environments where memory capacity and bandwidth are critical bottlenecks.

Compute Express Link (CXL) is the enabling technology for this transformation. Built on top of PCIe, CXL introduces several protocols for cache-coherent access between CPUs and accelerators.

Together, these protocols allow heterogeneous components—CPUs, GPUs, and specialized accelerators—to share memory, reducing data movement and improving performance. By supporting memory pooling and tiered hierarchies, CXL makes it possible to scale compute and

memory independently, enabling elastic AI/ML training environments, high-throughput analytics, and cost-efficient resource utilization.

3.1.1 Hierarchy Overview

3.1.1.1 Control Space

Each CXL device exposes a register interface for configuration and control, typically mapped into the host's address space via PCIe BARs.

These registers manage device state, link negotiation, and protocol operations such as memory read/write requests and coherence messaging.

In emulation environments (e.g., QEMU-based CXL 3.0 models), these registers are critical for simulating Type-3 memory devices and switches, handling protocol messaging and ACPI integration for OS recognition.

3.1.1.2 Memory Hierarchy Extensions

Host Memory remains the primary tier, but CXL introduces pooled and fabric-attached memory (FAM) accessible across multiple hosts (Figure 28).

CXL.mem Protocol enables direct load/store access to remote memory with implicit coherence actions, reducing reliance on explicit software synchronization.

CXL.cache Protocol supports cache-coherent access for accelerators, allowing them to share CPU memory without redundant copies.

CXL.io Protocol provides standard PCIe-like semantics for configuration and non-coherent transfers.

3.1.1.3 Fabric-Level Coordination

CXL switches allow multiple hosts to share memory pools without routing through main memory, improving latency and bandwidth.

Advanced synchronization primitives—such as atomic operations (compare-and-swap, fetch-and-add) and hardware barriers—are implemented at the fabric level to support distributed locking and fairness policies.

3.1.2 Protocol Implications

3.1.2.1 Coherence Management

Registers participate in coherence messaging, ensuring that memory operations across hosts maintain consistency. This is essential for workloads using shared-memory programming paradigms.

Performance and Scalability:
Studies show that topology choices (direct-attached vs. switch-based) significantly affect latency and bandwidth for HPC and AI workloads. Protocol overhead and congestion patterns influence cost models and scalability.

Synchronization Challenges:
Classical locking mechanisms (e.g., Peterson-style locks) can incur severe slowdowns—up to 1000×—in multi-host scenarios without hardware atomic primitives, underscoring the need for protocol-level synchronization support.

3.2 Physical Platforms

3.2.1 AMAIS Two Sisters

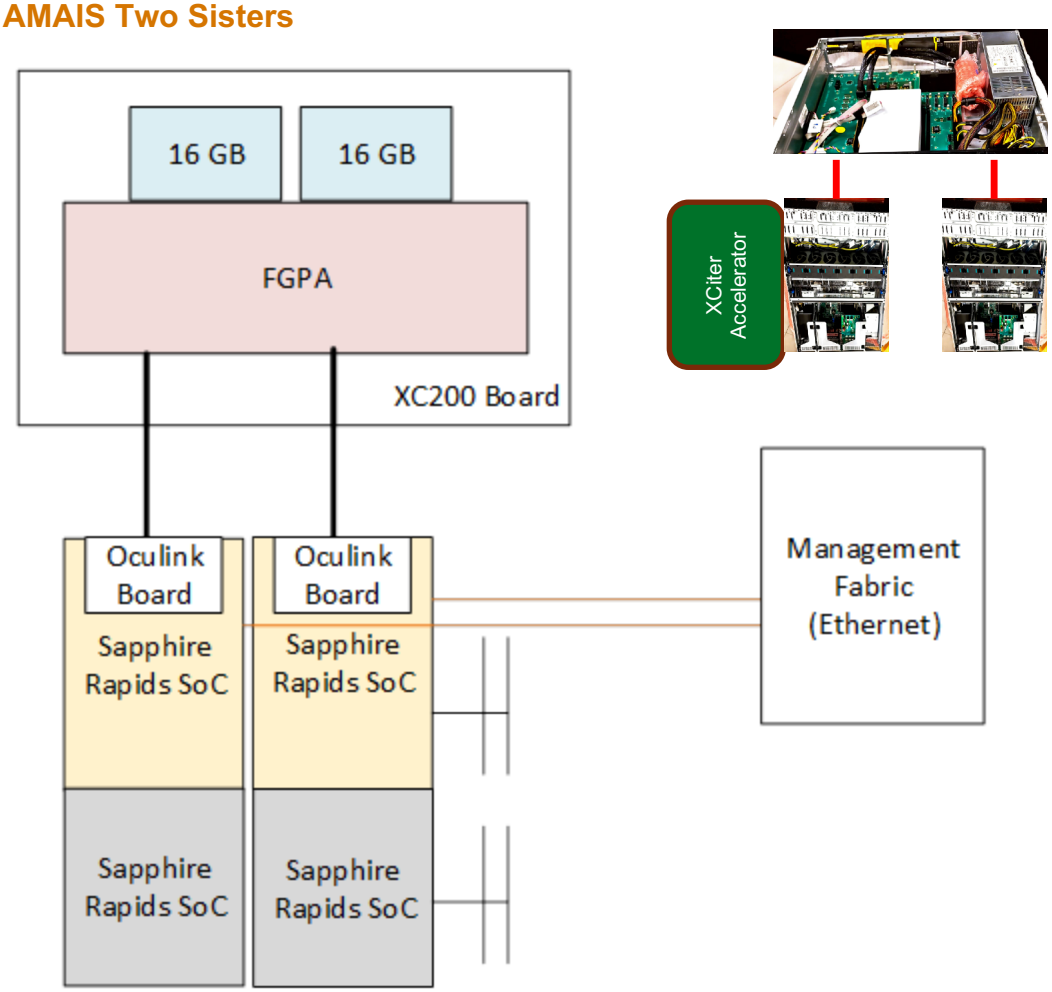


Figure 29 Two Sisters Machine

3.2.1.1 Two Sisters Overview

Installation Date: October 1, 2023

The AMAIS Two Sisters platform is designed as a Compute Express Link (CXL) switch emulation environment, supporting advanced memory and interconnect research. Key components include:

Core Specifications

CXL Specification: Version 2.x implemented on Xilinx Virtex UltraScale+ VU13P FPGA

Fabric-Attached Memory (FAM): 32 GB DDR4

Hosts: Two Intel Sapphire Rapids processors

Memory Module: Emulates CXL Type-3 device behaviors

Interface: CXL 1.1 over PCIe Gen4 speeds

FPGA: VU9P-class for initial module; VU13P-class for advanced configurations

Form Factor: Plug-in CEM card (DW, ¾ length)

Features: Memory-side cache and CXL-attached memory (32 GB DDR4)

Base Hardware: Micron FPGA development board (custom R&D ECN upgrade to VU13P)
https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/hpc-single-board-accelerators/sb-852

xCiter 2.0 DMC Module: Implements CXL Type-2 device behaviors

FPGA: VU13P-class

Interface: CXL 1.1 (PCIe Gen4 speeds)

Form Factor: Plug-in CEM card (DW, ¾ length)

Components: Threading Engine (TE), Command Manager (CM), memory-side cache

Memory: 32 GB DDR4 attached via CXL

Base Hardware: Micron FPGA development board (R&D ECN upgrade to VU13P)
https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/hpc-single-board-accelerators/sb-852

CXL-to-CXL Bridge: Connects current CXL production servers to the CXL switch

FPGA: VU13P-class

This architecture enables CXL switch emulation, memory pooling, and coherent interconnect testing, providing a foundation for research into fabric-attached memory, CXL protocol behaviors, and next-generation disaggregated computing.

3.2.2 AMAIS Crete

Crete comes in two logical configurations

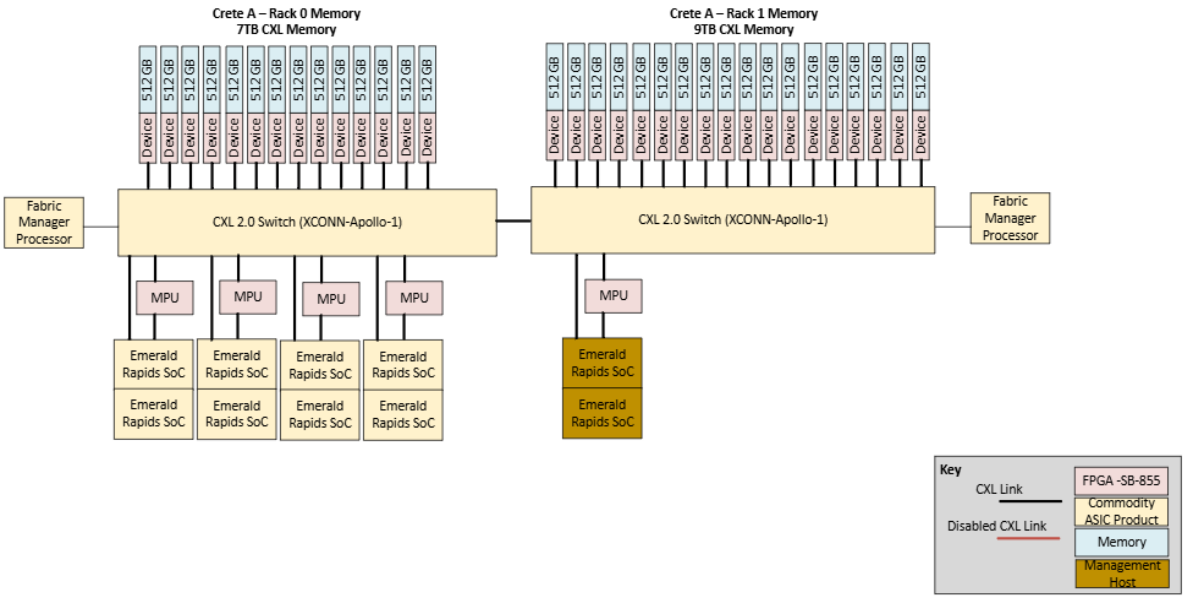


Figure 30 Crete Machine: 'Big Crete' Configuration

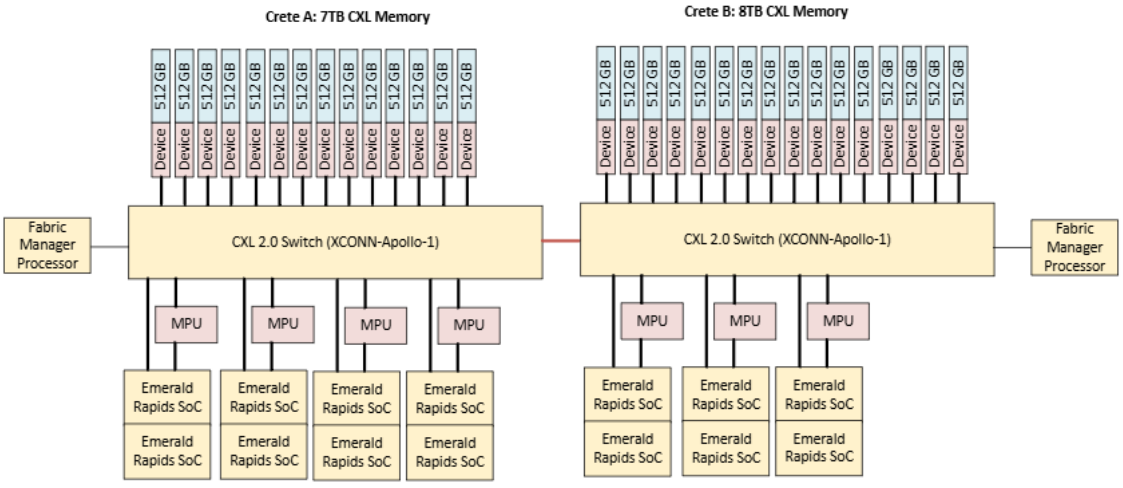


Figure 31 Crete Machine: 'Split Crete' Configuration

3.2.2.1 Crete Overview

Installation Date Phase-A: June 1, 2025

One sub-system installation (single switch)

Installation Date Phase-B: February 1, 2026

Full system installation (two switches)

NAS staging

Installation Date Phase-C: TBD

GPU accelerators

The AMAIS **Crete** platform is a high-capacity, multi-host CXL testbed designed to explore **memory pooling**, **near-memory compute acceleration**, and **CXL protocol scalability** for next-generation AI and HPC workloads.

The **AMAIS Crete** platform supports two distinct configurations to accommodate different research and development needs

Big Crete Mode

In this configuration, an **asymmetric top/bottom switch topology** enables four hosts connected to the top switch to access **all Fabric-Attached Memory (FAM) modules**. This setup is ideal for exploring **large in-core computations** and high-capacity memory pooling scenarios (Figure 30).

Split Crete Mode

Here, the two switches operate as **logically independent units**. The left switch supports four hosts, while the right switch provides three hosts with FAM access. This configuration is optimized for **development productivity**, allowing multiple **logical FAM silos** to run concurrently for parallel testing and workload isolation (Figure 31).

Both configurations support **active memory**, where each FAM module is equipped with a **programmable logic device** for **near-memory compute acceleration**, enabling advanced operations close to the memory layer for improved performance and reduced data movement.

Core Specifications

CXL Memory Devices: 15 Micron SB855 units

Capacity: 512 GB per device

Near-Memory Compute Acceleration: AMD VU13P FPGA integrated for advanced compute offload

Protocol Bridges: 4 Micron SB855 CXL MPU Protocol Bridge Units

CXL Switch Fabric: One XCONN XC50256 **CXL 2.0 Switch Chassis**

Management and Control: i.MX8 controller for system orchestration

Compute Infrastructure

Servers: Four MiTAC Fox Creek Pass Servers (M50FCP2UR208)

Equipped with **5th Generation Xeon Platinum 8568CXL CPUs** (CXL-enabled)

Host Memory: 1 TB Micron DDR5 per server (4800 MT/s, MTC40F2046S1RC48BA1)

Local Storage: 3.4 TB SSD per server

2× Micron 7450 MTFDKCB1T9TFR drives per node

Expansion and Management

CXL Expansion: 4 XCONN CXL Card Expansion Chassis for modular scaling

Management Server: Dell PowerEdge R360 for centralized control

Shared Memory Management: Micron's RAPID Multi-host Shared Memory Allocator for dynamic allocation

Networking and Power

Network Fabric: 100 Gb Ethernet switch (48 ports) for high-speed interconnect

Serial Console Access: Lantronix SLC8016 for host debug output

Power Infrastructure:

5 Rack PDUs

4× 2U 12V 120A AC/DC power supplies

1× 1U 12V 60A AC/DC power supply

Cabling

MCIO CXL Cabling:

35 meters total

20 cables (1 m)

10 cables (1.5 m)

3.2.2.2 Strategic Role of Crete in AMAIS AI and HPC Initiatives

Crete provides a large-scale, multi-host environment for evaluating CXL-based memory pooling, fabric-attached memory, and near-memory compute acceleration under real-world AI and HPC workloads. Its architecture supports elastic resource allocation, enabling researchers to test high-bandwidth interconnects, shared memory protocols, and scalable compute fabrics. This platform is critical for AMAIS's mission to design disaggregated, high-performance systems that meet the demands of large language models, data-intensive analytics, and next-generation scientific computing.

3.3 Virtual Platforms

3.3.1 OCEAN: Open-source CXL Emulation at Hyperscale Architecture and Networking

Compute Express Link (CXL) is a new technology that lets computers attach extra memory through fast links, instead of only relying on the memory chips (DRAM) plugged directly into the motherboard. In theory, this should help us build systems with much more memory, share memory across many machines, and move data more efficiently between them. These abilities are very attractive for modern workloads such as large scientific simulations, large language models (LLMs), and big distributed databases, all of which work with massive amounts of data.

However, so far, most experiments have not shown a significant performance advantage for CXL over traditional DRAM-only systems when running on just one or a few servers. Existing studies either use small "proxy" applications (simplified kernels that approximate parts of real workloads) on a single server, or they look at realistic distributed applications (such as MPI programs, LLM workloads, or distributed analytics) but can only run them at very small scale, because deploying CXL hardware or a CXL emulator at large scale is still practically unavailable. Under these small-scale conditions, CXL often looks slower than local DRAM, especially for compute-intensive programs where every extra nanosecond of memory latency matters.

This outcome makes sense. On a single machine, for programs that mostly spend their time doing calculations, low memory latency is critical, and DRAM connected directly to the CPU is still faster than CXL-attached memory. In these cases, CXL is unlikely to win. For memory-intensive programs, such as distributed database, the story changes: a CPU can support many more CXL devices via its available PCIe lanes and CXL switches (a single PCIe 5.0 switch can accommodate up to 144 lanes and 72 ports allowing for 71 CXL devices connected for x2 lanes per device; the CXL 3.0 Fabrics allow for multi-level switching enabling up to 4096 CXL devices connected within a single fabric) than it can via DDR5 DIMMs through its limited DIMM slots as many as 12 DIMM slots per CPU allowing for maximum of 12 DDR5 devices. More importantly, such switching fabric enables data transfers between devices without involving the host CPU.

By spreading ("interleaving") data across local DRAM and multiple CXL devices, a system can achieve significantly higher total memory bandwidth—on the order of 33–60% higher—than a DRAM-only system. For example, the CENT work shows a design that connects 32 CXL memory devices to a single CPU in a hierarchical way to support LLM inference. That system reaches 2.3× higher throughput than a state-of-the-art GPU solution, mainly because of its higher internal memory bandwidth with the connected 32 CXL devices.

Even so, the real advantage of CXL is likely to show up at much larger scale, when many servers are involved. In today's clusters, moving data between machines usually means sending messages over the network (e.g., MPI calls, RDMA operations). Each message involves copying data, packaging it, sending it, unpacking it, and possibly storing more copies on the receiving side. As datasets and models grow, these data movements become a major bottleneck. CXL

changes the game by making it possible for multiple machines to share a large pool of memory, accessible with normal memory reads and writes. Instead of each node holding its own copy of the data and constantly sending messages, nodes can work directly on shared data in the CXL memory pool. This "zero data copy" approach can, in principle, reduce the number of copies, shrink memory footprints, and cut communication overhead. For example, collective communication operations could operate on shared buffers instead of sending messages back and forth; LLM inference nodes could share a single key–value (KV) cache; and distributed databases could keep global indexes in one shared pool rather than replicating them everywhere.

To conclude, there are several key reasons why we haven't seen big performance wins from CXL at scale yet. *First*, most large-scale applications are still designed for the traditional model where each node has its own private memory and talks to others through messages. Their distribution and collective operations have not yet been redesigned to take advantage of a shared CXL memory pool. *Second*, real CXL hardware is still new and expensive, and it has not been deployed widely in top supercomputers or major data centers. Most available systems only have several CXL-enabled nodes, which is too small to study the real benefits of pooled memory across many machines. *Third*, current CXL emulation tools are either very low-level and slow (cycle-accurate simulators) or limited to simple, single-host and single-node scenarios. None of them can emulate a large, disaggregated CXL memory pool across hundreds of virtual nodes in a way that is fast enough to run real applications. *Finally*, CXL devices themselves are also costly and often lag the latest specification in terms of features. This makes it hard for small companies, universities, and many research groups to get enough hardware to experiment with CXL at large scale. It is practically impossible today to benchmark hundreds of nodes with real CXL hardware or to try out new CXL 3.x features on production systems.

Because of these constraints, what we need is a flexible, scalable CXL emulation platform. Such a platform should run on regular clusters or cloud infrastructure, but emulate many CXL-enabled hosts, memory devices, and fabric switches. It should support the latest CXL 3.x features, allow users to configure different topologies and latency/bandwidth settings, and provide an interface where system and application developers can implement and test CXL-aware algorithms and data structures. With this kind of "elastic" CXL emulator, we can finally study how CXL behaves at large scale, redesign collective and data-intensive operations to exploit shared pooled memory and evaluate performance and scalability before large CXL deployments are available in production systems.

This report presents such an emulation platform and aims to demonstrate how it can be used to explore the potential of CXL for future large-scale applications.

We develop this thrust as an open-source project to amplify impact beyond the immediate research team and actively seek outreach and community-building opportunities, which has been approved by the PNNL IP Legal/Commercialization office.

3.3.1.1 Related Work

Table 7 Emulation Categories

Category	Work
Software-Based Emulation	QEMU, OCEAN
Software-Based Simulation	CXL-DMSim, CXLMemSim, ESF, DRackSim
Hardware-Based Emulation	emucxl, Pond
Hardware Prototyping	DirectCXL, TPP

CXL Emulation and Simulation. Existing works related to CXL simulations can be divided into four categories as listed in the above Table 7. Initial efforts in CXL emulation include adding support in QEMU for basic CXL 2.0 features. That work enabled single-host experimentation with a CXL memory device, but it did not cover multi-host fabrics or the new capabilities of CXL 3.0. NUMA-enabled hardware has been used in to emulate CXL memory expansion. CXL-SIM provides a cycle-accurate simulation of CXL, useful for micro-architectural studies, but it is far too slow to run realistic software workloads or OS-level experiments. QEMU has basic support for CXL 2.0 device emulation, which allows a virtual machine to recognize a simple CXL Type-3 memory device, but this lacks multi-host capabilities and does not incorporate new CXL 3.0 features. Cycle-accurate simulators like CXL-DMSim or general architectural simulators like gem5 can model CXL devices at a low level, but they are extremely slow and impractical for full software stack development and evaluation. There have also been FPGA-based prototype platforms, but those tend to be limited in scale and costly to develop and use. Our work builds on these ideas by providing a high-performance, full-system emulator for CXL 3.0, bridging the gap between simplistic emulation and detailed simulation.

Memory Disaggregation. Memory disaggregation has been explored in various contexts. LegoOS is an operating system designed for a disaggregated data center, which includes disaggregated memory accessed over RDMA. Hydra provides a software-based coherent shared memory across a cluster using custom protocols on commodity hardware. These projects demonstrate the benefits and challenges of breaking the traditional one-machine, one-memory model. OCEAN differs in that it leverages the emerging CXL standard to implement disaggregation, rather than relying on bespoke software protocols. This means that software developed on OCEAN could potentially run on any CXL 3.0-compliant hardware in the future, benefiting from hardware support for coherence and memory sharing.

3.3.1.2 System Design

To address these challenges, this project proposes to design and build, from the ground up, an open-source full CXL 3.0 emulation platform, called **OCEAN** after **O**pen-source **C**XL **E**mulation **A**t Hyperscale Architecture and **N**etworking. The project then proposes to use it as the basis for application-driven memory hierarchy research, a memory-extension software stack, and CXL hardware co-design. Particularly, we will present a comprehensive software emulation

framework for CXL 3.0 with full multi-host support. The platform will ultimately target hyperscale and cloud environments, implementing key elements of the CXL 3.0 specification such as memory device emulation with multiple Host-managed Device Memory (HDM) decoders, a configurable Fabric Manager, flexible switch topologies for multi-host, and coherent memory pooling.

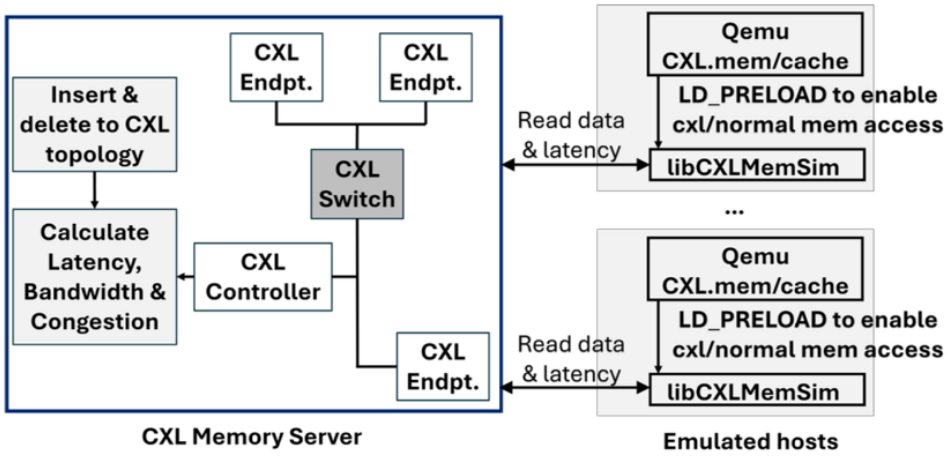


Figure 32 OCEAN Design Architecture

We illustrate the proposed OCEAN emulation design architecture in Figure 32. Each QEMU instance acts as a CXL-enabled host. All CXL memory accesses issued by a VM are intercepted and redirected to a central memory server. The server responds with the data, while also injecting latency based on topology, link distance, and switch depth. A distributed Fabric Manager daemon coordinates between the memory server and the VMs. This design allows us to emulate coherent memory sharing across up to 16 hosts. By just intercepting the CXL memory accesses from VM instances, OCEAN outperforms cycle-level simulators in terms of simulation speed. This allows workloads to run to their completion at near native speed instead of only simulating a few regions of interest which is a common practice in cycle-accurate simulators. On top of this emerging platform, the project will explore how real applications—ranging from NWChem to LLM workloads and distributed databases—can exploit CXL-based memory pooling, and how the emulator can evolve into a deployable software solution for multi-server memory extension, performance modeling, and eventual production. We delve into the OCEAN design details in two-fold: CXL 3.0 device emulation and multi-host coordination.

CXL 3.0 Device Emulation

Memory Device Emulation: OCEAN emulates CXL Type-3 memory devices with full compliance to the CXL 3.0 specification. Each emulated memory device supports features that mirror a real CXL 3.0 memory expander. For example, the device can expose up to 32 Host-managed Device Memory (HDM) decoders, allowing flexible mapping of memory regions into host physical address spaces.

Switch Emulation: CXL fabric switches are key components for enabling multiple hosts to share devices. OCEAN includes a virtual CXL switch model that supports a range of advanced

features. The switch can be configured with up to 256 virtual ports, which allows it to connect many hosts and devices in an emulated fabric. We implement hardware-based routing tables in the switch, including support for routing modes like multicast and anycast to efficiently handle one-to-many and many-to-one communication patterns.

Fabric Manager: CXL 3.0 defines a Fabric Manager (FM) entity responsible for orchestrating the memory fabric. In OCEAN, we implement a distributed Fabric Manager that runs partly in the host software and partly as logic within the emulated devices. The Fabric Manager performs automatic discovery of the topology: when the emulated system boots, it discovers all hosts, memory devices, and switches present in the fabric and builds a global connectivity map. It then handles resource allocation, deciding how much of the shared memory pool is assigned to each host and managing the mapping of global memory addresses to device-local addresses.

2) Multi-Host Coordination

Coherency Protocol: Maintaining cache coherency across multiple hosts is one of the most challenging aspects of CXL 3.0. OCEAN implements the full CXL 3.0 coherency protocol to keep memory consistent across all hosts and devices. We support the complete set of cache states (the Modified, Exclusive, Shared, Invalid (MESI) protocol states, along with any CXL-specific extensions), so that caches in different hosts correctly track ownership and sharing of memory lines. To reduce unnecessary coherence traffic, OCEAN uses distributed snoop filters: each host and switch can maintain a directory of which other hosts might have a given cache line, allowing the system to limit broadcast snoops to only the relevant nodes. We also optimize direct peer-to-peer communication between hosts for certain transactions. For example, if one host needs to access data that is currently in another host's cache, OCEAN can facilitate a direct transfer between those hosts (via the CXL switch) without going out to main memory, which improves latency.

Synchronization Primitives: Beyond basic memory sharing, multi-host applications often require efficient synchronization across hosts. OCEAN provides a set of global synchronization primitives built into the emulated hardware. For instance, we implement atomic operations (like compare-and-swap, fetch-and-add, etc.) that operate on the shared memory and are globally visible to all hosts, which simplifies building synchronization algorithms across nodes. We also provide a fast barrier primitive in hardware: hosts can enter a barrier and the emulated CXL fabric will only release them once all participants have reached the barrier, which is useful for coordinating phase-based computations across nodes. Additionally, a distributed lock mechanism is available, whereby hosts can attempt to acquire a named lock implemented in the Fabric Manager; this lock mechanism ensures fairness (to avoid starvation) and has configurable policies (like FIFO or priority-based) to match different use cases.

3.3.1.3 Implementation

1) Extensions to QEMU: We modified QEMU (version 10.0) to incorporate full CXL 3.0 emulation support. In total, our extensions to QEMU amount include approximately 15,000 lines of new C code. Key modifications include:

Device Models: We implemented new device models under QEMU's hw/cxl subsystem to model CXL 3.0 Type-3 memory devices and switches. These models handle the device register interface, CXL protocol messaging (e.g., memory read/write requests, coherence messages), and integration with QEMU's PCIe and ACPI subsystems (so that the virtual BIOS/OS can recognize and configure the CXL devices).

Memory Backend: We developed a custom memory backend in QEMU to support the dynamic capacity feature of CXL 3.0. This backend interacts with the host operating system's memory subsystem to allocate and free memory regions on the fly, and it ties into the CXL device model to expose these regions to the guest hosts. It also supports persistent memory (if available on the host) to emulate scenarios where CXL memory might be backed by non-volatile memory.

Switch Implementation: We added a virtual CXL switch device in QEMU that can be instantiated to connect multiple virtual hosts to multiple devices. The switch implementation uses QEMU's internal networking infrastructure as a starting point but extends it to model CXL routing, switching latency, and bandwidth constraints. It also provides hooks for the fabric management protocols (allowing the Fabric Manager module to run discovery and configuration tasks through the switch).

2) Kernel Modules: To complement the QEMU-level changes, we implemented several Linux kernel modules to support CXL 3.0 features in the guest OS (*Linux 6.16-rc7* in our setup):

CXL Driver Extensions: We extended the Linux CXL kernel driver to recognize and work with our emulated CXL 3.0 devices. This includes support for new device capabilities like GFAM and MH-SLD. The driver can enumerate the fabric, map global memory into the Linux memory manager, and handle *hotplug* events (when memory is added or removed by OCEAN).

Memory Management: We wrote a kernel module that acts as a custom memory allocator for CXL memory regions in the guest. It communicates with the OCEAN Fabric Manager to request memory when the guest OS wants to use CXL memory, and it ensures that memory is returned to the pool when freed. This module maintains the mapping between the guest's notion of physical pages and the global memory addresses.

Coherency Module: We developed a small kernel module to assist in handling coherency protocol messages in the guest. While QEMU primarily handles coherency at the device level, this module helps integrate the coherence events with the guest OS (for instance, if a cache line owned by the CPU needs to be written back or invalidated due to a remote access, the module can coordinate with the CPU's cache flush mechanisms).

3) User-Space Components: Several user-space tools and daemons are included in OCEAN to manage the CXL fabric and assist developers:

Fabric Manager Daemon: A central user-space service (written in C++) runs on the host to perform high-level fabric management. It communicates with the QEMU instances (for low-level control) and uses out-of-band channels to exchange information about topology and resource status. This daemon provides a shell or RPC interface for issuing management commands (like adding a device, partitioning memory among hosts, injecting errors, etc.).

CLI Tools: We provide a set of command-line interface tools for configuring and monitoring OCEAN. For example, there are tools to list the current fabric topology, check how much memory each host is using, manually trigger a memory rebalancing, simulate a device failure, or switch from shm/TCP/RDMA mode. These tools are implemented in Python and C, and they interface with the Fabric Manager Daemon and kernel modules.

Libraries: For application developers, OCEAN develops a user-space library (with C/C++ APIs) that simplifies using CXL-shared memory. For instance, an application can use our library to allocate a region of shared memory that is visible to other hosts or to create a global mutex or atomic variable. Under the hood, the library interacts with the kernel modules to carry out these operations. This allows developers to prototype and test distributed applications that exploit CXL features without dealing with low-level details.

4) Transport Layer: To support various deployment scenarios—ranging from single-server functional testing to high-performance multi-node clusters—we decoupled the CXL protocol logic from the underlying transport mechanism. OCEAN implements a modular transport layer that supports three distinct modes: Shared Memory (SHM), TCP/IP, and RDMA.

Shared Memory (SHM): This mode is optimized for single-node deployments where all guest VMs (hosts) and the memory server process reside on the same physical machine. We utilize POSIX shared memory segments to create a zero-copy data path. The memory server allocates the backing store for the CXL devices in shared memory, which is then mapped directly into the virtual address space of the QEMU instances. CXL flits are passed via circular ring buffers implemented in shared memory, synchronized using process-shared pthread mutexes and condition variables. This mode provides the lowest latency (≈ 1µs) and is ideal for functional verification and developing coherency protocols without network overhead.

TCP/IP: For ease of deployment across standard data center networks without specialized hardware, OCEAN provides a TCP-based transport. In this mode, CXL transaction layer packets (TLPs) are serialized and encapsulated into standard TCP packets. While this introduces higher latency due to kernel stack overhead and data copying, it ensures compatibility with virtually any networked environment (including public clouds). We use this mode primarily for testing fabric management and discovery protocols across large-scale topologies where strict timing fidelity is less critical than topological complexity.

RDMA: To achieve performance fidelity in multi-node setups, OCEAN implements an RDMA backend using the ibverbs API. This mode maps CXL load/store semantics to RDMA READ/WRITE operations. When a guest OS issues a read to an emulated CXL region, the QEMU instance issues an RDMA READ to the remote memory server, bypassing the remote CPU entirely. For coherency messaging and interrupts, we utilize RDMA SEND/RECV verbs to ensure low-latency delivery. By leveraging hardware offload, the RDMA transport allows OCEAN to sustain high bandwidth (near line-rate) and maintain low latency even when emulating hosts are distributed across different physical servers, making it the preferred mode for performance evaluation.

3.3.1.4 Evaluation

1) Experimental Setup: We compare the performance of OCEAN against three alternatives. First, we use a traditional RDMA-based memory disaggregation approach as a baseline, where memory is shared over an InfiniBand network using RDMA. Second, we compare to QEMU's existing CXL 2.0 emulation (without multi-host support) to highlight the improvements in our CXL 3.0 implementation. Third, we reference the expected performance of native CXL 3.0 hardware as projected by the specification and vendor estimates in order to gauge the fidelity of our emulation.

2) Microbenchmarking

Memory Latency: We first measure basic memory access latency under different scenarios on OCEAN. For local memory access (where a host accesses its own directly attached memory via CXL, analogous to non-uniform memory access within a node), OCEAN achieves a latency of roughly 1µs for a random read, compared to an expected ~250 ns on real CXL 3.0 hardware. For remote memory accesses that traverse a CXL switch (one host accessing memory that is primarily attached to a different host or a shared device), the latency is about 1.1µs in OCEAN, versus an estimated 400 ns on real CXL 3.0 hardware. In the worst-case cross-host scenario, which involves both a CXL switch traversal and a coherence action (e.g., data being owned by another host's cache), we observe about 1.2µs latency on OCEAN, compared to around 600 ns projected for actual hardware. These results show that while OCEAN introduces some overhead, it remains within 3x of the expected real hardware latencies.

Memory Bandwidth: We use the STREAM benchmark to evaluate memory bandwidth in various configurations. On a single host with a dedicated CXL memory device, OCEAN achieves a sustained bandwidth of 8.4 GB/s for sequential access patterns, which is about 85% of the theoretical maximum for a 100Gbps RDMA. In a multi-host scenario, we had four hosts simultaneously reading from a shared memory pool; OCEAN delivered an aggregate bandwidth of 6.2 GB/s across the hosts, indicating that our emulation handles contention and parallel access efficiently.

Coherency Overhead: To quantify the overhead of maintaining coherence across hosts, we crafted microbenchmarks where multiple hosts frequently read and write to the same cache lines. In OCEAN, the additional latency for coherence-related communication (snoop requests, invalidations, etc.) averages about 120 ns per affected operation. In a stress scenario where many writes to shared data occur in a short time, we observed up to a 15% drop in throughput due to the coherence traffic and cache invalidations, which is in line with what we expect on a real system might experience under similar conditions. We also examined the effect of false sharing (when two hosts accidentally access different data that resides on the same cache line): in such cases, performance was degraded by roughly 8% in our tests. This indicates that careful data placement or padding might be necessary in software to avoid false sharing when using a CXL-based shared memory system, just as it is in tightly coupled multiprocessors.

Litmus Test: To validate the correctness of OCEAN's coherency protocol implementation, we conducted extensive testing using memory consistency litmus tests like store-buffer, atomic operations, and tiering tests. These tests are designed to expose subtle bugs in multiprocessor

memory systems by exercising specific ordering scenarios that reveal whether the system properly implements its claimed consistency model.

For the Store-Buffer (SB) litmus test, which is particularly sensitive to memory ordering violations, we ran 1 million iterations across 4 hosts sharing a CXL memory region. In a weakly-ordered system, both threads can observe the old values (resulting in a "00" outcome). In our evaluation, the "00" outcome never occurred, as expected. But the other three cases all happened with the "11" outcome dominating. This distribution indicates that OCEAN correctly models the relaxed consistency semantics of CXL, showing eventual consistency behavior consistent with the specification rather than strict sequential consistency.

We also validated message passing scenarios using CXL atomic operations. In these tests, one host writes data followed by a flag set using an atomic store, while another host spins on the flag using atomic loads before reading the data. Across 10 million iterations, we observed zero violations of causality—the receiving host always observed the data write when the flag was set, confirming that OCEAN properly orders atomic operations with respect to regular memory accesses.

For tearing detection, we performed concurrent 64-bit writes and reads to unaligned addresses that span cache line boundaries. Without proper atomic guarantees, such operations can result in torn reads where parts of old and new values are mixed. Our tests confirmed that OCEAN correctly implements the CXL specification's requirements: aligned 64-bit operations remained atomic (0% tearing rate), while unaligned operations showed the expected tearing behavior (approximately 3.2% tearing rate under high contention), matching what would occur on real CXL hardware.

These litmus test results demonstrate that OCEAN faithfully reproduces the memory consistency guarantees of CXL 3.0, providing researchers with a reliable platform for developing and testing concurrent algorithms designed for CXL-based systems.

3) Cast Studies

Case Study 1: Genomics Pipeline Acceleration. This case study involves a genomics data processing pipeline, GROMACS. The GROMACS pipeline typically consists of multiple stages (such as read mapping, variant calling, etc.) where each stage produces intermediate data that the next stage consumes. Traditionally, these intermediate results are communicated between stages using MPI collective operations or written to parallel file systems, which becomes a bottleneck for large genomics datasets. We implemented a four-stage genomics pipeline across 4 hosts using OCEAN. A 100GB CXL *shared memory* region was allocated to serve as a high-speed buffer for intermediate data. Each stage of the pipeline used CXL-aware "memcpy" operations to write its output data directly into the designated regions of the shared memory pool. The next stage (running on a different host) could then access the data immediately through direct memory reads, bypassing traditional MPI communication patterns.

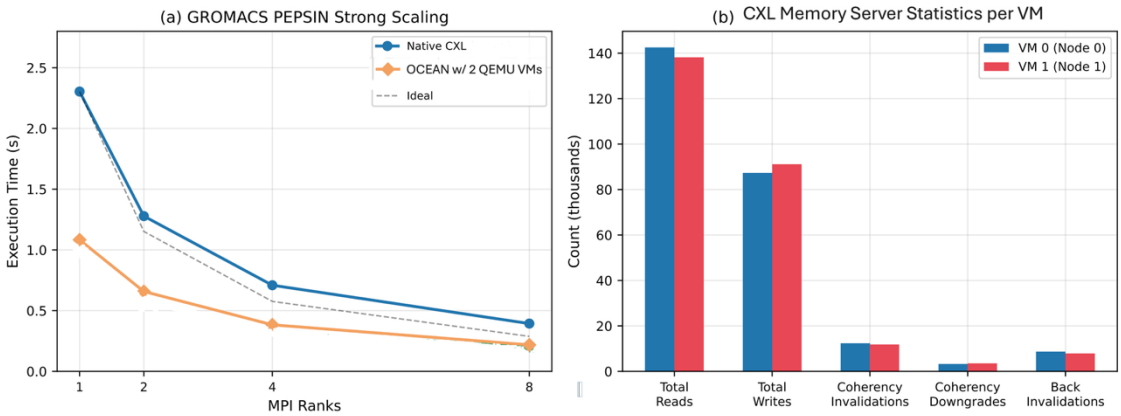


Figure 33 Two-host GROMACS on real CXL hardware vs. OCEAN.

This design replaced MPI Send/MPI Recv operations with simpler memcpy calls to CXL memory addresses with atomic memory to notify the readiness of the data, eliminating the overhead of message passing protocols. As a result, we measured that the inter-stage communication latency dropped from an average of 45ms (using MPI over InfiniBand) to just 42ms using CXL memcpy operations. The pipeline avoided approximately 12 GB of redundant data copies that would have occurred with traditional MPI buffering. The synchronization between stages was handled through CXL atomic operations rather than MPI barriers, further reducing coordination overhead. When one stage completed processing a data chunk, it simply sets a flag in shared memory that the next stage could poll with minimal latency. Overall, OCEAN reduced the total processing time of the pipeline by about 15%. This case study highlights how CXL's memory sharing can benefit data-intensive applications by replacing complex message passing protocols with simple memory operations, leveraging the speed and simplicity of shared memory semantics for what traditionally required explicit communication libraries.

To compare OCEAN with real hardware, we ran GROMACS with native CXL and with OCEAN across two hosts while varying the number of MPI ranks. Figure 2 shows the results. Across all MPI ranks, OCEAN consistently delivers lower execution time than the Native CXL configuration. CXL memory server statistics collected from each VM demonstrate that reads, writes, and coherence events are relatively balanced between nodes, suggesting that the runtime preserves locality and avoids excessive invalidation traffic.

Case Study 2: Distributed Training of a Large Language Model. The other case study demonstrates OCEAN using a large-scale distributed machine learning workload. We trained a language model (LLaMa-2) with approximately 7 billion parameters across 8 hosts. Each host was equipped with 8 GPUs, and all hosts shared a 512 GB memory pool via OCEAN to hold the model's parameters. In addition, about 100 GB of training data (input batches) at any given time was stored in the shared memory for quick access by all GPUs. In this configuration, OCEAN significantly accelerated training compared to a conventional setup. We observed a roughly 42% reduction in the time to train one epoch of the model, primarily due to faster parameter synchronization. The communication overhead between nodes was reduced by about 65% because instead of exchanging parameters or gradients over the network, the hosts directly read and wrote to the shared memory pool. The scaling of training throughput with the number of

hosts was near-linear up to 8 hosts, indicating that OCEAN efficiently handles the multi-host coordination for this workload. This case study shows that OCEAN can effectively support memory-intensive, communication-heavy workloads like distributed deep learning, providing performance benefits even in the absence of actual CXL hardware.

Note that in both case studies the latency injection has been always enabled, although the latency injection only introduces a modest performance overhead.

3.3.1.5 Future Steps

Specifically, the team will carry out three core tasks in the next step.

Reconstructing Bluesky for a CXL Emulation System. This thrust focuses on turning the OCEAN prototyping emulator into a more accessible CXL emulation system on Bluesky and then stress testing it with demanding, real-world workloads. The first objective is to deploy the full prototyping emulation stack onto Bluesky, a PNNL homemade cluster of 32 compute nodes. In this setup, QEMU-based virtual machines will be launched across multiple compute nodes as emulated CXL hosts, while a dedicated VM running the CXL memory server will reside on a separate node to model a shared, fabric-attached memory pool. The distributed Fabric Manager and topology-aware latency injection mechanisms will be configured to span these VMs, allowing us to emulate realistic multi-host CXL 3.0 fabrics directly on Bluesky.

The specific tasks upon the readiness of the Bluesky-based CXL emulation system will include (1) testing and benchmarking data-intensive and communication-intensive applications on the CXL emulation platform and exploring performance bottlenecks and how these applications can be redesigned to exploit CXL-style memory pooling, such as NWChem, LLM workloads (e.g., KV-cache–heavy inference and RAG pipelines), the Tigon distributed database, MPI-based HPC workloads; (2) optimizing and developing new underlying parallelization and communication operations to best exploit CXL's zero–data-copy, shared-memory semantics. For instance, developing pointer exchange capabilities in the message passing level within the new CXL-advanced MPI for one-sided communication; and (3) providing controlled access to the CXL emulation platform as a service to external users and industry partners (e.g., Micron), addressing potential issues, and collecting feedback.

Reinventing CXL Emulation towards a lightweight CXL software solution. This thrust focuses on evolving beyond the OCEAN prototyping emulator to develop a lightweight CXL-like memory extension software stack suitable for multi-server and cloud environments. Unlike the prototyping emulator, which is designed for full CXL 3.0 feature coverage (e.g., 32 HDM decoders, a fully featured Fabric Manager, rich switch topologies, and detailed latency injection) and for exploring architectural correctness, this lightweight solution emphasizes flexibility, simplicity, and direct integration with existing cloud and virtualized infrastructures. It is intended as a pragmatic software that makes CXL-style memory pooling available as a system component.

To support deployment across multiple machines and cloud instances, this requires refactoring the Fabric Manager beyond its prototyping role into a distributed or hierarchical service that can handle dynamic pool creation, host enrollment, and policy-driven allocation and isolation across

many clients. Maintaining a consistent global view of pool membership, address space layout, and access policies in the presence of dynamic joins, leaves, and pool reconfigurations is a key new challenge.

We will address it by designing a lightweight control protocol between Fabric Manager components and participating hosts to ensure safety and consistency at scale. The lightweight solution will streamline how CXL-like memory is exposed to guest VMs and containers. We will provide minimized device models and guest OS drivers that present the pooled memory as a standard memory resource while internally routing accesses to the shared pool. We will design the entire stack for cloud-native deployment. The memory server, Fabric Manager, and host-side components will be packaged into a lightweight hypervisor service that can be instantiated and managed by a cloud orchestrator. This represents a clear difference from the prototyping emulator, which is oriented toward manual or scripted experiments, and moves toward a reusable, service-oriented component that can be integrated into real cloud workflows.

Stretch Goals: CXL Type 2 Device Emulation and CXL memory pool-informed chip design. The final thrust comprises stretch goals that extend the platform to heterogeneous devices and hardware co-design.

The first extension is to add support for CXL Type 2 device emulation, focusing on GPUs from vendors such as NVIDIA, AMD, and Intel. This would entail building GPU device models that can access the emulated CXL pools, either directly or via host-mediated transfers, and integrating with existing GPU software stacks so that selected allocations can be backed by CXL memory instead of or in addition to host DRAM.

The other extension is to connect the emulator and its performance models to chip design workflows using tools such as Bambu. The idea is to collect detailed traces of memory activity from emulated CXL environments—address streams, timing information, types of transactions, and inferred access patterns—and translate these into inputs for architectural exploration.

3.3.1.6 Outreach Activities

We will actively seek outreach and community-building opportunities to amplify impact beyond the immediate research team. We will actively seek new collaborations with universities and industrial partners in memory, systems, HPC and cloud computing, engaging them both as early users of the emulator and as co-design collaborators on CXL-aware algorithms and applications.

We have established the CXL Emulation Workgroup, created a webpage and a LinkedIn page of it, and announced the launch of it at SC’25. We will expand it by contributing test cases, sharing benchmarks and artifacts, and onboarding new academic and industry members through regular meetings and discussions on future roadmap.

We will organize periodic tutorials and hands-on workshops at premier venues such as IPDPS and SC to advertise the emulator platform and best practices for CXL-based system development.

4.0 Bibliography

Agostinelli, Victor, Nicolas Bohm Agostini, and Antonino Tumeo. 2025. “UltraFormer: An Efficient Transformer for FPGAs.” *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 4, 274–274. https://doi.org/10.1109/FCCM62733.2025.00034.

Agostini, Nicolas Bohm, Serena Curzel, Vinay Amatya, et al. 2022. “An MLIR-Based Compiler Flow for System-Level Design and Hardware Acceleration.” *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, October 30, 1–9. https://doi.org/10.1145/3508352.3549424.

Ankur Limaye, Nicolas Agostini, Vito Castellana, et al. 2023. “Designing Intelligent Memory Interfaces to Improve Accelerator-Based System Performance.” Paper presented at EEE/ACM 42nd International Conference on Computer-Aided Design (ICCAD 2023), San Francisco, CA, USA. October 29.

Barone, Claudio, Rishika Kushwah, Ankur Limaye, et al. 2024. “To Cache or Not to Cache? Exploring the Design Space of Tunable, HLS-Generated Accelerators.” *Proceedings of the International Symposium on Memory Systems*, September 30, 210–18. https://doi.org/10.1145/3695794.3695815.

Dagostini, Jessica I., Joseph B. Manzano, Tyler Sorensen, and Scott Beamer. 2025. “miniGiraffe: A Pangenomic Mapping Proxy App.” *2025 IEEE International Symposium on Workload Characterization (IISWC)*, October 12, 339–52. https://doi.org/10.1109/IISWC66894.2025.00036.

Devulapally, Anusha, Mahantesh Halappanavar, Bruno Jacob, Vijaykrishnan Narayanan, and Andres Marquez. 2025a. “Revisiting Pebble Games for Modeling and Efficient Use of Disaggregated Memory Systems.” *Proceedings of the International Symposium on Memory Systems*, October 6, 8–14. https://doi.org/10.1145/3767110.3767118.

Devulapally, Anusha, Mahantesh Halappanavar, Bruno Jacob, Vijaykrishnan Narayanan, and Andres Marquez. 2025b. “Revisiting Pebble Games for Modeling and Efficient Use of Disaggregated Memory Systems.” *Proceedings of the International Symposium on Memory Systems*, October 6, 8–14. https://doi.org/10.1145/3767110.3767118.

Devulapally, Anusha, Mahantesh Halappanavar, Amit Puri, Vijaykrishnan Narayanan, and Andres Marquez. 2024. “Using Isoefficiency as a Metric to Assess Disaggregated Memory Systems for High Performance Computing.” *Proceedings of the International Symposium on Memory Systems*, September 30, 192–97. https://doi.org/10.1145/3695794.3695812.

Dhruv Gajaria. 10/24. “Bringing It HOME: Analyzing Contention Hotspots across the Memory Hierarchy with Low Overhead.” Poster. Supercomputing.

Gajaria, Dhruv, and Andrés Márquez. 2025. “HOME: A Hierarchy-Oriented Memory Evaluation Framework for Fast Contention Analysis.” *Proceedings of the International Symposium on Memory Systems*, October 6, 213–25. https://doi.org/10.1145/3767110.3767123.

Ghosh, Sayan. 2022. “Improved Distributed-Memory Triangle Counting by Exploiting the Graph Structure.” *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, September 19, 1–6. https://doi.org/10.1109/HPEC55821.2022.9926376.

Gioiosa, Roberto, Bo Fang, Lenny Guo, and Andres Marquez. 2025. “Hardware-Software Co-Development for Emerging CXL Architectures.” *Proceedings of the International Symposium on Memory Systems*, October 6, 1–7. https://doi.org/10.1145/3767110.3767113.

Herrera Guaitero R.A., J.B. Manzano Franco, J.D. Suetterlein, and A. Marquez. 2025. “CARTS: Enabling Event-Driven Task and Data Block Compilation for Distributed HPC.” Paper presented at LLVM-CGO. March 1.

Hou, Shiyue, Nathan R. Tallent, Li Wang, and Ningfang Mi. 2024. “Performance Analysis of Data Processing in Distributed File Systems with Near Data Processing.” *2024 International Symposium on Networks, Computers and Communications (ISNCC)*, October 22, 1–6. https://doi.org/10.1109/ISNCC62547.2024.10758994.

Islam, Md Shohidul, Sankha B. Dutta, Andres Marquez, Ihsen Alouani, and Khaled N. Khasawneh. 2024. “Harnessing ML Privacy by Design Through Crossbar Array Non-Idealities.” *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 25, 1–2. https://doi.org/10.23919/DATE58400.2024.10546651.

Kang, Yan, Sayan Ghosh, Mahmut Kandemir, and Andrés Marquez. 2024. “Impact of Write-Allocate Elimination on Fujitsu A64FX.” *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, January 11, 24–35. https://doi.org/10.1145/3636480.3637283.

Kang, Yan, Sayan Ghosh, Mahmut Kandemir, and Andrés Márquez. 2024. “Studying CPU and Memory Utilization of Applications on Fujitsu A64FX and Nvidia Grace Superchip.” *Proceedings of the International Symposium on Memory Systems*, September 30, 198–207. https://doi.org/10.1145/3695794.3695813.

Kilic, Ozgur O., Nathan R. Tallent, Yasodha Suriyakumar, Chenhao Xie, Andres Marquez, and Stephane Eranian. 2022. “MemGaze: Rapid and Effective Load-Level Memory Trace Analysis.” *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, September, 484–95. https://doi.org/10.1109/CLUSTER51413.2022.00058.

L’Abbate, Ryan, Anthony D’Onofrio, Samuel Stein, et al. 2024a. “A Quantum-Classical Collaborative Training Architecture Based on Quantum State Fidelity.” *IEEE Transactions on Quantum Engineering* 5: 1–14. https://doi.org/10.1109/TQE.2024.3367234.

Lee, Hyungro, Milan Jain, and Sayan Ghosh. 2022. “Sparse Deep Neural Network Inference Using Different Programming Models.” *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, September 19, 1–6. https://doi.org/10.1109/HPEC55821.2022.9926362.

Limaye, Ankur, Nicolas Bohm Agostini, Claudio Barone, et al. 2025. “A Synthesis Methodology for Intelligent Memory Interfaces in Accelerator Systems.” *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, January 20, 1016–22. https://doi.org/10.1145/3658617.3697553.

Palmer, Bruce, Sayan Ghosh, and Andrés Márquez. 2024. “Distributed-Memory Sparse Deep Neural Network Inference Using Global Arrays.” *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, September 23, 1–7. https://doi.org/10.1109/HPEC62836.2024.10938434.

Peng, Zhen, Polykarpos Thomadakis, Jacques Pienaar, and Gokcen Kestor. 2025. “LiteForm: Lightweight and Automatic Format Composition for Sparse Matrix-Matrix Multiplication on GPUs.” *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, July 20, 1–14. https://doi.org/10.1145/3731545.3731574.

Ravan Nazaraliyev, Yicheng Zhang, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. 2025. “Not so Refreshing: Attacking GPUs Using RFM Rowhammer Mitigation.” *34th USENIX Security Symposium*, July 10.

Shin, Philip Wootaek, Jack Sampson, Vijaykrishnan Narayanan, Andres Marquez, and Mahantesh Halappanavar. 2025a. “Disharmony: Forensics Using Reverse Lighting Harmonization.” *2025 IEEE/CVF Winter Conference on Applications of Computer Vision Workshops (WACVW)*, February 28, 708–17. https://doi.org/10.1109/WACVW65960.2025.00085.

Shin, Philip Wootaek, Jack Sampson, Vijaykrishnan Narayanan, Andres Marquez, and Mahantesh Halappanavar. 2025b. “Losing the Plot: How VLM Responses Degrade on Imperfect Charts.” Version 1. Preprint, arXiv. https://doi.org/10.48550/ARXIV.2509.18425.

Subasi, Omer, Sayan Ghosh, Joseph Manzano, Bruce Palmer, and Andrés Marquez. 2024. “Analysis and Benchmarking of Feature Reduction for Classification under Computational Constraints.” *Machine Learning: Science and Technology* 5 (2): 020501. https://doi.org/10.1088/2632-2153/ad3726.

Suetterlein, Joshua, Joseph Manzano, and Andres Marquez. 2024. “Synchronization for CXL Based Memory.” *Proceedings of the International Symposium on Memory Systems*, September 30, 178–85. https://doi.org/10.1145/3695794.3695810.

Suriyakumar, Yasodha, Nathan R. Tallent, Andres Marquez, Karen L. Karavanic, and Ozgur O. Kilic. 2024. “MemFriend: Understanding Memory Performance with Spatial-Temporal Affinity.” *Proceedings of the International Symposium on Memory Systems*, September 30, 270–84. https://doi.org/10.1145/3695794.3695820.

Wu, Jianbo, Jie Liu, Gokcen Kestor, Roberto Gioiosa, Dong Li, and Andres Marquez. 2024. “Performance Study of CXL Memory Topology.” *Proceedings of the International Symposium on Memory Systems*, September 30, 172–77. https://doi.org/10.1145/3695794.3695809.

Zhang, Yicheng, Ravan Nazaraliyev, Sankha Baran Dutta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2024. “Beyond the Bridge: Contention-Based Covert and Side Channel Attacks on Multi-GPU Interconnect.” *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*, May 16, 35–36. https://doi.org/10.1109/SEED61283.2024.00014

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354

1-888-375-PNNL (7665)

www.pnnl.gov