

The ACE Center for Evolvable Computing

Josep Torrellas[†], Minlan Yu[‡], José F. Martínez[§], Radu Teodorescu[¶], Zhiru Zhang[§], Zhengya Zhang^{||}, Tarek Abdelzaher[†], Mohammad Alian[§], Adam Belay^{**}, Rajesh Gupta^{††}, Tushar Krishna^{‡‡}, Arvind Krishnamurthy^{§§}, Christos Kozyrakis^{¶¶}, Charith Mendis[†], Subhasish Mitra^{¶¶}, Muhammad Shahbaz^{||}, Rachee Singh[§], Steven Swanson^{††}, Michael Taylor^{§§}, Mohit Tiwari^{|||}, Mengjia Yan^{**}
[†]University of Illinois [‡]Harvard University [§]Cornell University [¶]Ohio State ^{||}University of Michigan ^{**}MIT ^{††}UC San Diego
^{‡‡}Georgia Tech ^{§§}University of Washington ^{¶¶}Stanford University ^{|||}UT Austin

Abstract—This paper presents some of the main research contributions of the ACE Center for Evolvable Computing—an SRC/DARPA JUMP 2.0 center focused on systems and architectures for distributed compute. ACE’s goal is to devise novel technologies for distributed computing that substantially improve the energy efficiency and performance of applications. The research encompasses computing engines; memory and storage; communication and coordination; and security, privacy, and correctness.

I. INTRODUCTION: THE ACE VISION

The ACE Center for Evolvable Computing¹ is the SRC/DARPA JUMP 2.0 Center that focuses on systems and architectures for distributed compute. At a time of rapid escalation of energy consumption and deep technological disruption in distributed computing, ACE’s goal is to substantially improve the energy efficiency and performance of distributed computing systems. ACE envisions the future distributed computing infrastructure to be a seamless hierarchy of compute centers, from edge centers to geo-distributed mega-datacenters. For energy efficiency, the infrastructure will contain a large number of heterogeneous hardware accelerators, and tasks of small granularity will ship computation to where data is. In this environment, it is key to leverage hardware accelerators, since they compute in the most energy-efficient manner, and to minimize data movement, since it is the biggest source of energy waste.

To attain these goals, ACE argues for an *Evolvable* computing paradigm. The idea is that the new accelerator hardware, memory system interfaces, communication stacks, and security mechanisms must be designed for extensibility and composability. These systems should survive upgrades of their external environments, and be easily replaced by (and co-exist with) their own next-generation designs. These properties have served us well in the era of general-purpose processors; as we now move into an accelerator-intensive era, we need to retain them.

This article highlights the key projects and major accomplishments of ACE so far.

II. A COMPUTING BACKBONE BASED ON ACCELERATORS

ACE advances a vision of evolvable accelerator-centric computing where datacenters and other computing infrastructure integrate diverse, evolving accelerators. Rather than relying on a small set of specialized hardware, ACE embraces heterogeneity and continuous evolution, recognizing that many accelerator types must coexist to support emerging workloads.

For ease of communication, sets of heterogeneous accelerators are grouped into *ensembles*. This vision requires new methodologies to rapidly create, deploy, and reconfigure accelerators, while preserving programmability and efficiency. Central to ACE’s effort is an automated hardware–software co-design loop, where accelerator generation and compiler construction proceed in tandem, enabling rapid specialization, efficient code mapping, and low-overhead orchestration of accelerator ensembles.

Over the past few years, ACE has delivered the key building blocks toward this vision, including: design flows for rapid accelerator generation, automatic generation of compiler back-ends for new accelerators, and a framework for orchestrating fine-grained accelerators. In addition, ACE has designed diverse accelerators spanning tightly and loosely coupled systems for both dense and sparse workloads, and fast and accurate modeling tools for design-space exploration.

As shown in Figure 1, these advances come together in a feedback-driven workflow: accelerator modeling and prototyping inform automated accelerator generation and automated compilation creation, while compiler and system insights guide subsequent modeling refinements.

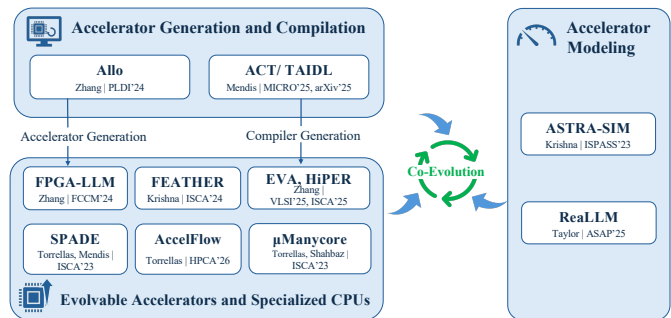


Fig. 1: Vision of the accelerator framework in ACE.

A. Evolvable Accelerators

Modern workloads are diverse and evolve rapidly. This exposes a key tension in accelerator design: hardware optimized for a given dataflow and layout is efficient, but quickly becomes mismatched as workloads change. Accelerators force a tradeoff between specialization and generality.

ACE addresses this challenge by developing a design flow for automated accelerator generation. *Allo* [1] is a high-level, Python-embedded accelerator design language and compiler stack based on the Multi-Level Intermediate Representation

¹<https://acecenter.granger.illinois.edu>.

(MLIR) that enables rapid generation of specialized accelerators in a modular and composable way. *Allo* decouples algorithmic logic from hardware-specific optimizations (e.g., pipelining and memory banking), supports type-safe composition of parameterizable kernels, interoperates with PyTorch, and includes a built-in formal checker to verify functional equivalence before and after optimization passes. *Allo* has been demonstrated through the creation of a complex LLM inference accelerator on an FPGA, and various accelerators, achieving high performance, energy efficiency, and a 10× reduction in lines of code compared to manual RTL.

ACE has developed multiple configurable accelerators for a variety of workloads. One of them is *SPADE*, which targets mixed sparse-dense kernels. *SPADE* adapts dynamically to input sparsity patterns and achieves high utilization across a range of sparsities without the overhead of a fully custom sparse engine. A second accelerator is *FEATHER* [2], which addresses a key source of inefficiency in systolic array accelerators: even when dataflow is reconfigured, mismatches between dataflow and on-chip SRAM data layout cause systematic bank conflicts. To tackle this, *FEATHER* reconfigures both dataflow and layout at the same time, via a novel network-on-chip that reorders data in-place during reduction. *FEATHER* has been generated end-to-end using *Allo*.

EVA [3] is a programmable accelerator that targets edge-computing workloads. Many edge workloads use heterogeneous sets of compute kernels, and can evolve rapidly. *EVA* combines systolic array tiles (A-Tiles) with RISC-V CPU tiles (C-Tiles). A-Tiles support diverse kernels with high utilization, while C-Tiles provide general-purpose processing and enable rapid runtime adaptation. *EVA*'s versatile interconnect fabric—including local systolic interconnects, tile-level router networks, and inter-tile buses—supports efficient and programmable data movement. An *EVA* chip was fabricated in Intel 16 technology (16 mm², FP16). It operates at up to 1.5 GHz and achieves up to 1.54 TFLOPS and 708 GFLOPS/W.

ACE has also developed an extensive simulator infrastructure to evaluate accelerators. One instance is *ReaLLM*, which provides end-to-end modeling for cloud-scale distributed LLM inference accelerators.

B. Accelerator Ensembles

ACE envisions an accelerator ensemble as a heterogeneous set of accelerators in multiple chiplets on a package, affording manufacturers the ability to easily change the combination of accelerators in the package as the applications to accelerate evolve. Standard interfaces for chiplet-to-chiplet connectivity such as UCIe help make this vision a reality. To lower the barrier to chiplet integration and enable seamless interoperability across technology nodes, ACE is designing a synthesizable digital die-to-die interconnect compatible with UCIe. Its physical layer is specified in RTL and can be synthesized for different process nodes.

ACE has developed two versions of a chip to test this digital die-to-die interconnect. The first, mostly digital prototype, was taped out in 2025. The second one, with an all digital

design and improved portability will be taped out in August 2026. Compared to state-of-the-art die-to-die links, this design targets a lower data rate, but is compact, easy to generate, and energy-efficient.

To make ensembles of accelerators usable, it is necessary to develop techniques to identify which sections of an application have affinity for which types of accelerators, provide an API to generate the specialized code, and orchestrate the operation of the different accelerators at runtime with low overhead.

ACE introduced *AccelFlow* [4], an accelerator orchestration framework that manages multiple accelerators for microservices. Microservice environments suffer from “datacenter tax” caused by operations such as transmission control protocol (TCP), encryption, decryption, remote procedure calls (RPC), serialization, deserialization, compression, and decompression. Using hardware accelerators to speed-up these operations is challenging because these operations take tens of microseconds. Using a CPU core to orchestrate accelerators by invoking individual accelerators and receiving an interrupt when each accelerator completes has high overhead. In *AccelFlow*, CPU cores build software structures called *Traces* that contain a sequence of accelerator IDs. A trace can include branch conditions, whose outcomes determine the control flow inside the trace. A CPU core triggers accelerator execution by passing a trace to an accelerator. From then on, the accelerators in the trace execute in sequence without CPU involvement, passing data from one accelerator to the next, potentially executing branch conditions. The result is fast execution of microservices. Currently, identifying application sections for acceleration is performed manually, and generating the specialized code is performed semi-automatically. Future work involves automating these processes.

C. Automatic Compiler Construction for Accelerators

To make accelerators truly useful, they must be programmable with a high-level language. Unfortunately, most accelerators beyond GPUs (both academic and commercial) lack compilation support, which hinders their adoption by the community. Moreover, developing compilation frameworks takes a long time, especially relative to the rapid evolution rate of accelerator hardware.

To bridge this gap, ACE introduced the *Accelerator Compiler Toolkit (ACT) Ecosystem* [5]. Given the Instruction Set Architecture (ISA) of a tensor accelerator and its interface, *ACT* automatically generates a compiler backend and test oracles for the accelerator. This enables software to rapidly co-evolve with the hardware design.

Central to *ACT* is the tensor accelerator ISA definition language (*TAIDL*), which standardizes the development of semantics for accelerator ISAs. *TAIDL* models accelerator ISAs using high-level tensor Intermediate Representation (IR) constructs themselves. Given the *TAIDL* definitions for an accelerator, *ACT* automatically generates highly-scalable test oracles for the accelerator for software testing. These oracles are orders of magnitude faster than manually written versions, enabling rapid software testing.

Next, *ACT* automatically generates a sound and complete compiler backend for the accelerator that can be directly integrated with established ML compilers such as XLA. The backend yields high-performance code with low compilation overhead. The compiler backend generation is a combination of two stages: an instruction-selection stage and a memory-allocation stage. *ACT* automatically generates these transformations based on the *TAIDL* definitions. *ACT*-generated backends have higher compilation coverage than even industry-developed tooling. For example, the *ACT*-generated compiler backend targeting the AWS NKI interface achieves $2.1\times$ greater compilation coverage than AWS' neuronx-cc NKI compiler.

D. Specialized CPUs

While accelerators provide high efficiency for specialized kernels, CPUs remain indispensable for general-purpose code. ACE envisions that CPUs will differentiate for specific workloads due to the intense pressure for energy efficiency. One example is specialization for cloud-native workloads, namely microservice and serverless environments. Current processors are not a good match for these workloads. Current processors tend to have wide-issue cores, support hardware cache coherence across all cores, include advanced prefetching and branch prediction hardware to excel in long-running, predictable applications, and are designed to maximize the average performance of applications. In contrast, cloud-native workloads run services that have low Instruction Level Parallelism (ILP), do not share writable data with other services, run for very short times and often block on I/O, execute in a finely-interleaved manner with other services, and have strict tail latency requirements.

To execute cloud-native workloads efficiently, ACE proposed the *μ Manycore* [6]. Its architecture focuses on minimizing tail latency and on eliminating hardware structures that are unnecessary for cloud-native workloads. To minimize tail latency, *μ Manycore* adds hardware to alleviate bottlenecks that can add long latency to some service requests. Common bottlenecks are enqueueing, dequeuing, and scheduling service requests, context switching of services due to I/O, and contention in the on-package network. Consequently, *μ Manycore* chiplets include hardware support for enqueueing, dequeuing, and scheduling requests, as well as for saving and restoring process state in context switches. Further, to minimize network contention, *μ Manycore* uses a leaf-spine on-package interconnection network, which offers several redundant, low-hop connections between the same source and destination chiplet pairs. Finally, *μ Manycore* eschews package-wide hardware cache coherence, and is built out 8-16 core hardware cache-coherent domains. A cluster of *μ Manycores* reduces the tail latency of microservice requests by $10.4\times$ over a state-of-the-art system.

III. HETEROGENEOUS AND INTELLIGENT MEMORY AND STORAGE

In ACE's vision, the memory and storage tiers are active, partially intelligent heterogeneous layers of the distributed infrastructure. The premise is that the gains promised by accelerators cannot be realized if data movement remains the dominant cost: a trillion-parameter AI model requires terabytes of capacity for its weights alone, modern Key Value (KV) caches routinely exceed the High-Bandwidth Memory (HBM) capacity of a single GPU, and emerging workloads such as Retrieval-Augmented Generation (RAG) and hybrid vector searches are dominated by memory accesses. Meeting this reality requires a coordinated rethink of three layers at once: 1) the memory system itself, which must compose multiple technologies and be distributed across many nodes; 2) the management of these resources across competing tenants; and 3) the placement of compute alongside data, to alleviate movement costs.

ACE has made contributions across all three layers. Examples include a CXL-coherent storage architecture that pulls SSDs into the same shared memory plane as DRAM and lets them coordinate autonomously; a compute-enabled CXL memory expander that pushes data filtering into DRAM banks; and a variety of other memory and storage work.

A. Heterogeneous Memory Systems

Emerging computing systems increasingly break down the traditional processor-centric view of the system, where each processor is surrounded by fast SRAM caches and slower DRAM within the node. Today's systems increasingly rely on new far-memory tiers that sit beyond local DRAM, including SSD-backed far memory and compressed far-memory tiers that expand effective capacity by storing cold data in compressed form. At the same time, CXL is exposing memory as a resource that can extend across devices, nodes, and racks rather than remaining confined to one server. Non-volatile media and SSDs are moving close enough to the processor to participate more directly in the execution. Meanwhile, modern workloads place simultaneous pressure on capacity, bandwidth, energy, and tail latency in a way that no single memory technology can satisfy on its own. ACE's approach is to turn this heterogeneity into an asset: memory/storage systems that span technologies and topologies expose enough of their internal behavior to remain manageable by the software.

XFM [7] illustrates this direction on the memory side by introducing a new compressed far-memory tier into the system, and then using an accelerator in that tier to manage it. *XFM* shows that a software-defined far-memory layer built from compressed cold pages can serve as an intermediate tier for applications with compressible data and predictable access patterns. Compression and decompression operations are not ordinary processor work, but memory-side functions. *XFM* couples the compressed far-memory tier with a near-memory accelerator, which performs compression and decompression during DRAM refreshes—removing much of its interference with normal execution.

AutoSSD [8] extends a similar philosophy to the storage side. Current all-SSD RAID arrays offer high aggregate throughput but suffer from long tail latencies, driven primarily by background operations performed by the flash translation layer—e.g., garbage collection. Approaches that schedule around busy SSDs from the host side run into structural limits: the host’s view of SSD state is stale, the host cannot reach into device-internal queues, the scheduling itself contends with primary CPU work, and host scheduling capacity does not scale as devices are added. *AutoSSD* removes the host from the loop. Each SSD publishes its own busy state via a flag in CXL-coherent shared memory, and busy SSDs forward incoming requests directly to non-busy peers without host CPU mediation. The result is a substantial reduction in tail latency. *AutoSSD* points toward a vision where storage devices are participants in an intelligent fabric.

B. Near- and In-Memory/Storage Acceleration

Even in a well-managed heterogeneous memory/storage system, at some point, the energy and latency cost of moving data to compute units dominates everything else. A promising way past that ceiling is to bring computing to the data. Accordingly, ACE devotes a task to pursuing intelligent memory and storage (IMS), which embeds small compute elements in SRAM, DRAM, NVM, and SSD storage. The goal is to attain a reusable memory and storage substrate capable of near-data computation whose hardware investment can be amortized across different workloads.

DReX [9] is a compute-enabled CXL memory expander that co-designs algorithm and hardware for dense data retrieval. This workload searches a body of dense vectors and returns the top-k instances most relevant to the user query. *DReX* is implemented as a CXL Type-3 device whose internal memory has eight LPDDR5X packages. There are simple compute engines in the DRAM banks and a near-memory accelerator outside of the DRAM. The system combines two ideas. An initial search is run within the DRAM banks themselves, which have a tailored data layout. This step prunes vectors that are unlikely to be among the top-k for the query without ever leaving the DRAM array. Then, the near-memory accelerator performs exact similarity scoring on the small set of survivors and returns truly accurate top-k results to the host. On a representative dataset, *DReX* delivers an 85% reduction in time-to-first-token relative to the state-of-the-art CPU algorithm. This work was performed with the PRISM Center.

LongSight [10] reuses the *DReX* substrate to speed-up transformer attention in LLMs. As context windows in transformer-based LLMs have grown into the millions of tokens, the attention phase now dominates inference latency, and the KV cache dominates HBM footprint. *LongSight* observes that the same filtering algorithm that *DReX* uses to prune embedding vectors can also prune irrelevant keys in attention, since the underlying operation in both cases is a query-against-many-vectors dot product. *LongSight* therefore stores the KV cache in *DReX*’s CXL and offloads sparse attention to it. Crucially, repurposing *DReX* for attention requires no changes to its

in-DRAM filtering units and only a small expansion of the near-memory accelerator’s scratchpad. With a single GPU and *DReX* device, *LongSight* supports context lengths of up to one million tokens, substantially improving throughput and reducing per-token latency over the state of the art.

ACE has made several other contributions to IMS systems. Some examples are *FloatAP*, which extends in-SRAM associative processing to full floating-point support, and *PIMsynth*, which synthesizes different digital and analog processing-in-memory architectures from a common Verilog front-end.

IV. FINE-GRAINED COMMUNICATION AND COORDINATION

Modern scale-out applications—from large-scale LLM training to latency-sensitive inference serving, and multi-user microservice and serverless environments—share a common reality: communication and coordination are major bottlenecks. LLM training routinely occupies over 10,000 GPUs, where collective communication can consume significant execution time; LLM serving requires tens of accelerators per model, with memory bandwidth and inter-node transfers on the critical path of every token; and microservices demand rapid, fine-grained resource management across a shared infrastructure. As computation becomes accelerated by specialized hardware, and memory/storage improve efficiency through near-memory/storage compute, communication bandwidth and latency require increased attention.

ACE addresses the challenge of delivering high-performance, energy-efficient, and adaptive communication and coordination through co-design: joint optimization across network fabrics, memory systems, compute engines, and application-level algorithms. ACE’s main work spans three multi-tenant cloud workloads: LLM training, LLM inference, and microservice and serverless environments. It also includes a robust effort on benchmarking and simulation at scale. Figure 2 shows the different components of a datacenter where ACE has made contributions.

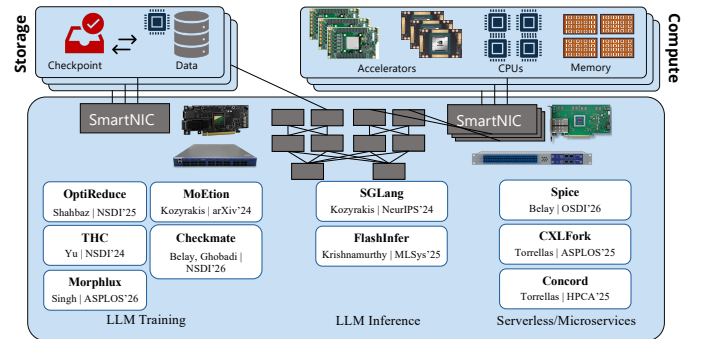


Fig. 2: Components of a datacenter where ACE contributed.

A. LLM Training: Performance and Fault Tolerance

Training LLMs stresses every layer of the datacenter stack. A single training run for a frontier model routinely occupies hundreds of thousands of accelerators for weeks or months,

processing diverse data types (text, image, video, code) across diverse model architectures (dense transformers, mixture-of-experts (MoE), multi-modal pipelines). At large scale, communication overhead grows superlinearly with cluster size, and failures shift from rare exceptions to routine events—Meta reports GPU failures approximately every two hours during large training jobs. ACE’s training efforts span two fronts: performance and resilience.

1) *Improving Training Performance*: The workhorse of training is collective communication primitives (e.g., AllReduce, AllGather, and ReduceScatter). ACE has made several contributions to make these operations faster and more energy-efficient. Traditional implementations of collectives assume homogeneous, fully connected links, but modern GPU clusters increasingly feature direct-connect fabrics—optical circuit switches, patch-panel circuits, or switchless physical links—where each node communicates with only a small number of peers at any given time. As a result, existing algorithms are suboptimal.

ACE developed an algorithmic framework that jointly synthesizes both the network topology and its communication schedule for direct-connect collectives. The approach starts from small, provably optimal base topologies and iteratively expands them to large-scale deployments using a family of expansion techniques, while a novel polynomial-time schedule generation algorithm produces efficient communication schedules. A Pareto-efficient search then identifies the best topology-schedule pair for a given cluster size, degree constraint, and workload. On a 12-node reconfigurable GPU testbed, the synthesized topologies reduced collective communication time by over 30% for DNN training; large-scale simulations demonstrate up to an order of magnitude reduction.

ACE observes that the strict reliability guarantees of communication protocols (e.g., retransmissions, in-order delivery) add significantly to tail latency in AllReduce due to congestion and stragglers. However, ML models are resilient to partial gradient loss, so perfect reliability is unnecessary. Hence, ACE introduced *OptiReduce*, which speeds up AllReduce by allowing early completion once the 99th percentile of data has arrived, and by relaxing the need to perform retransmissions.

To reduce the bandwidth of collective communication, prior work exchanges gradients in compressed form. Unfortunately, this requires the receiving node to decompress, aggregate, and recompress gradients—a slow process that negates much of the bandwidth benefits. ACE has proposed *Tensor Homomorphic Compression (THC)*, which eliminates this overhead through homomorphic compression: compressed gradients are directly summable without decompression. On a 64-GPU cluster, *THC* combined with *OptiReduce* achieve 1.3–2.2× faster time-to-accuracy compared to state-of-the-art baselines, with no loss in model quality.

2) *Enhancing Training Resilience*: GPU hardware failures account for nearly 50% of all training interruptions at scale. Each failure requires pausing training, migrating the state to spare hardware, repopulating the communication library state,

and reloading the most recent checkpoint. Today’s collective libraries treat collective communication groups as static. Hence, if the topology changes, the state needs to be recreated from scratch.

ACE addresses recovery overhead across the full stack. At the physical layer, ACE introduces *Morphlux*, a programmable photonic fabric that enables *in-place* replacement of failed accelerators. Rather than migrating the whole training job to spare hardware, the photonic interconnect is reprogrammed in microseconds to route around the failed GPU and use a replacement GPU. After the failure, we enable the GPUs to reconstruct the state of the collective library during execution, minimizing downtime. An end-to-end prototype demonstrated recovery from failures within seconds, compared to minutes with the state-of-the-art.

At the framework level, ACE rethinks the traditional bottleneck of checkpointing. *MoEtion* introduced *sparse checkpointing* for MoE models. Rather than saving the full model state, *MoEtion* incrementally checkpoints subsets of experts across iterations. A sparse-to-dense reconstruction algorithm restores logically-consistent dense checkpoints at recovery time. *MoEtion* reduces per-iteration checkpoint size by 55%. Another design, *Checkmate*, provides zero-overhead checkpointing by reusing the gradients exchanged during data-parallel AllReduce: programmable switches multicast gradients to a shadow CPU cluster that performs the checkpoint.

B. LLM Inference: Scheduling, Memory, and Programmability

The challenge of LLM inference is that different phases stress different resources. Prefill is compute-bound, while decode is memory-bandwidth-bound. Different model architectures (dense transformers, MoE models, multi-modal pipelines) and different serving scenarios (single-turn, multi-turn with KV cache reuse, RAG) exercise the compute, memory, and communication subsystems differently.

ACE’s inference work spans five layers of the stack. At the model efficiency layer, ACE developed quantization and approximate attention techniques that reduce per-request compute and memory consumption (e.g., *Atom* and *GEAR*). At the model attention layer, ACE developed approximation optimizations that significantly enhance the scalability of the attention operator (e.g., *Tactic*, *TurboAttention*, *FLuRKA*, and *LongSight*). At the scheduling and resource allocation layer, ACE built prediction-aware schedulers that exploit request characteristics, such as embedding-based output-length predictors, to improve GPU utilization and reduce tail latency (e.g., *TRAIL*) as well as systems that exploit intra-GPU task-level parallelism (e.g., *NanoFlow*). At the memory management layer, ACE addressed the KV cache bottleneck through intelligent offloading to CPU DRAM and network-accelerated memory access across the scale-up domain (e.g., *NEO*, *Aqua*, and *DReX*). Finally, at the distribution layer, ACE tackled cross-node orchestration challenges, including disaggregated prefill-decode architectures, SmartNIC-accelerated inter-node communication, and cluster-wide resource allocation via

decomposition-based optimization. Several of these techniques achieve multiple-fold speedups over commercial systems.

A notable ACE project is *SGLang* [11], which provides a programmable inference engine that co-optimizes the execution of structured language model programs—workloads where a single user request involves multiple LLM calls interleaved with control flow, branching, and external tool invocations. *SGLang* introduces RadixAttention for automatic KV cache reuse across structured program executions, a compressed finite state machine for constrained decoding, and an optimized runtime that batches across program boundaries.

ACE also introduced the *FlashInfer* [12] kernel library, which provides the high-performance GPU primitives necessary for serving LLM models. *FlashInfer* organizes LLM inference into four operator families—Attention, GEMM, Communication, and Sampling—and exposes each through lightweight, composable APIs with just-in-time compilation that adapts kernels to diverse model architectures and KV cache layouts. Its vector-sparse attention kernels achieve high bandwidth, and its block-sparse formats optimize memory access patterns for irregular KV cache structures.

SGLang and *FlashInfer* have become foundational infrastructure for the open-source LLM serving ecosystem. *SGLang* is a de facto standard in high-performance LLM serving. *FlashInfer* is integrated into vLLM, *SGLang*, and NVIDIA's TensorRT-LLM, and NVIDIA now releases its most performant inference kernels directly through *FlashInfer*.

C. Serverless and Microservice Environments

In serverless and microservice environments, millisecond-granularity stateless functions must be instantiated on demand, scaled elastically, and coordinated with other functions. These environments present multiple performance challenges, including function cold start delays, burstiness, and frequent stalls due to storage accesses. To support these environments with high performance, ACE has made multiple contributions.

ACE introduced *Spice* [13] to minimize the overhead of function cold start. *Spice* redesigns the Linux kernel's checkpointing subsystem to alleviate or eliminate the bottlenecks of conventional snapshot/restore—many system calls for metadata restore, repeated memory mappings, and copy-on-write page faults. *Spice* introduces: a libOS-based metadata restore service that operates entirely in userspace, an object format for efficiently representing snapshotted page tables, and a new kernel interface that pre-installs page table entries and asynchronously prefetches memory pages. The result is a 10–15× reduction in cold-start latency over state-of-the-art checkpointing systems.

In rack-scale deployments, bursty conditions cause many instances of the same function to run concurrently on different nodes. To enable low-overhead remote forking of functions across nodes, ACE introduced *CXLfork* [14]. *CXLfork* checkpoints a function's state into CXL-attached memory with low overhead once. Then, it forks the function in other nodes with near-zero serialization and zero copy. Children processes in remote nodes access read-only pages directly from CXL

memory, and seamlessly cache their data in their local caches; when they write to a page, they trigger a copy-on-write. *CXLfork* reduces the local memory consumption by 87% over prior mechanisms, enabling more functions to reside in memory.

The performance of functions is severely limited by frequent accesses to global storage. To mitigate this overhead, data can be cached in the memory of the nodes that execute the functions. Unfortunately, existing caching schemes are suboptimal: they either restrict an object to be cached in a single, fixed node, or use a cache coherence protocol that requires many messages. To address these limitations, ACE proposed *Concord*, a software cache coherence protocol for distributed systems tailored to serverless environments. It allows multiple copies of the same object to be cached in different nodes concurrently, while keeping them coherent with low overhead. Compared to the state-of-the-art, *Concord* speeds up execution by 2.4×.

These three systems are interfaced together, demonstrating that high-performance execution of fine-grained functions is achievable through coordinated hardware-software co-design.

D. Benchmarking and Simulation at Scale

ACE has developed multiple tools to evaluate large clusters. Two popular ones are *Chakra*, a graph-based representation of distributed AI workloads that includes standard execution traces, and *ASTRA-sim*, a versatile simulator of distributed ML architectures.

V. SECURE, PRIVATE AND CORRECT ACCELERATORS

New system architectures, increasingly rich in heterogeneous accelerators, call for a rethink of security and correctness mechanisms explicitly targeted to accelerators. In addition, emerging AI-enabled applications increasingly process large volumes of sensitive data. Hence, it is important to understand the new security threats that they introduce and how to defend against them. ACE addresses these challenges by discovering, analyzing and mitigating emerging threats in AI-centric applications; designing new trusted execution solutions for evolvable accelerator-rich systems; and developing new frameworks for cross-stack verification of security and correctness. Figure 3 shows the ACE architecture for security and correctness.

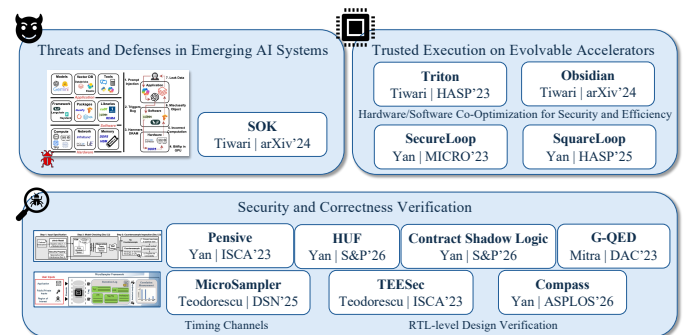


Fig. 3: The ACE architecture for security and correctness.

A. Emerging Threats and Defenses in AI-Centric Applications

Emerging compound AI inference pipelines integrate multiple LLM models, ML techniques, and external tools. Such systems increasingly operate on sensitive inputs and data. They offer expanded attack surfaces, including AI algorithms as well as the software and hardware components they use. While current research often examines these components in isolation, combining cross-layer attack observations can enable powerful end-to-end attacks.

ACE takes a systematic and comprehensive look at different software and hardware attacks applicable to compound AI systems. We have collected attack gadgets across stack layers, and created a comprehensive repository of attack primitives. In ACE's *SoK* framework, we demonstrate multiple end-to-end attack chains by composing cross-stack attack gadgets. Defending against these attacks necessitates vertically-integrated security primitives. These primitives must be deployed within isolated execution environments, and be anchored by hardware platforms that provide Trusted Execution Environment (TEE) guarantees.

B. Efficient Trusted Execution for Evolvable Accelerators

Designing security primitives for accelerators faces the challenge of minimizing their performance overhead. This is because they closely interact with key parts of the architecture, such as the scheduling of work to the accelerators or the memory system behavior. To achieve both security and performance/energy efficiency, ACE developed design-space exploration tools. *SecureLoop* [15] is the first systematic exploration and optimization framework to tackle this problem. It introduces crypto-aware scheduling of work and analytical optimization of authentication-block assignment. *SecureLoop* can find schedules for secure DNN designs with up to 33% speedup and 50% improvement in energy-delay product. As a follow-up, *SquareLoop* improves the fidelity of *SecureLoop*. It considers a broader authentication-block design space, more precise memory modeling, and a more advanced search mechanism that co-searches the mapping and authentication-block strategy. Taken together, these works show that security for accelerators should be incorporated into the architecture exploration loop itself.

The *Sesame* [16] secure inference accelerator explored multi-tenant isolation mechanisms such as on-chip resource partitioning, memory-traffic shaping, and zeroization of sensitive data to enforce temporal and spatial isolation. While these security primitives provide strong isolation guarantees, they often impose substantial performance overheads. In practice, in a request, only a subset of tokens typically contains user-sensitive information. Selectively activating security primitives only for these private data regions enables significant performance improvements. The *Triton* [17] compiler provides an API that allows developers to annotate private data subsets. *Triton* then employs taint-tracking-based dataflow analysis to separate the computation of private and public data. Special instruction flags are used to signal the hardware to enable on-chip protection mechanisms only when required.

C. Security and Correctness Verification

ACE has developed a suite of scalable and robust verification frameworks that improve the security, correctness, and trust of the next generation computing systems. These frameworks demonstrate that joint verification of software and hardware is crucial, that formal verification can prove security and correctness of large-scale designs, and that detailed visibility into the normally hidden microarchitectural state enables identification of new vulnerabilities. This section discusses some of frameworks.

1) *Co-verification of TEE Software and Hardware*: A TEE is a set of hardware extensions that provide security guarantees for applications running on untrusted operating systems. The security of TEEs is threatened by a variety of microarchitectural vulnerabilities, which have led to many attacks. While various approaches for verifying TEE designs have been proposed, they do not generally extend to jointly verifying the security of the underlying microarchitecture. To address this limitation, ACE developed *TEESec*, a pre-silicon framework for discovering microarchitectural vulnerabilities in the context of TEEs. *TEESec* jointly and systematically tests the TEE APIs and underlying microarchitecture against leakage across isolation boundaries. *TEESec* uncovered 10 distinct vulnerabilities in two open-source RISC-V out-of-order processors.

2) *Leakage Detection in Constant Time Execution*: Constant time programming is a principal line of defense against timing side-channel attacks. Unfortunately, it relies on assumptions about the underlying microarchitectural implementation that are frequently incorrect or incomplete, leading to exploits. To address this challenge, ACE developed *MicroSampler* [18], a dynamic leakage-detection framework to identify secret-dependent microarchitectural behavior that can lead to side-channel leakage in security-critical software. *MicroSampler* runs the constant-time code to be verified on a cycle-accurate register-transfer level (RTL) simulation of the target system and builds a detailed representation of microarchitectural state captured at cycle granularity. *MicroSampler* then uses statistical analysis to measure any existing correlation between microarchitectural state and data values that are identified as sensitive (e.g., encryption keys). *MicroSampler* is able to reveal multiple vulnerabilities in constant-time encryption code, including a previously-unknown vulnerability in OpenSSL.

3) *Formal Verification of Hardware Designs*: Formal verification using model checkers offers rigorous security guarantees. However, most existing model-checking methods, if used off-the-shelf, do not align with the complexity and scale of realistic hardware. Hence, ACE has extensively focused on techniques for compute engine verification, both at early-stage and at RTL design, and both for security and correctness.

• *Formal Verification of Early-stage Architectural Designs*. ACE showed that it is feasible to leverage a designer's insights to identify the aspects of a design that are essential to achieve security goals, and to formulate them in ways that verification tools can use. ACE introduced *Pensieve* [19] to demonstrate this idea for timing side channels—a type of information

leakage where secrets influence the timing of the execution. *Pensieve* explicitly exposes timing signals as a modeling principle where microarchitectural designs are described as a modular composition, and RTL-level hand-shaking signals are lifted as interfaces across modules to make timing signals explicit. With this modeling principle, designers are guided to focus on capturing all relevant design details that determine timing behavior. *Pensieve* discovered new attack variants and vulnerabilities that were missed by manual examination.

- *Verification of RTL Designs.* Formal verification of accelerator hardware for correctness at the RTL level is a time-consuming and error-prone process. ACE advanced the state-of-the-art of this field with the *G-QED* [20] verification framework. *G-QED* can verify interfering accelerators—i.e., accelerators that can produce a different result for a given input depending of the input's context within a sequence of inputs. Given a hardware accelerator with its set of actions, *G-QED* verifies the accelerator without requiring detailed knowledge about its design. *G-QED* demonstrates broad applicability, and proves soundness and completeness guarantees. The effectiveness and practicality of *G-QED* has been demonstrated through a variety of accelerators and processor cores. Also, in an industrial case study, *G-QED* demonstrated higher verification productivity, improving from 370 person-days using an industrial flow to 21 person-days using *G-QED*.

ACE has also contributed with formal verification techniques for security at the RTL level. Some of the contributions are *Contract Shadow Logic*, *Compass*, and *HUF*.

D. AI-Assisted, Formally Verified Design

LLMs can rapidly generate RTL code, promising to compress chip development cycles, cost, and effort. However, the designs produced by even frontier models typically contain subtle bugs that are difficult to fix. To improve the designs produced by LLMs, ACE has explored *AI-Boosted Chip Design (ABCD)*. The idea is to invoke an LLM to generate an initial design of the accelerator and then, repeatedly, execute *G-QED* to find and characterize any bugs in the design, and invoke the LLM to revise the design, given the previous design and the bugs found. Initial experiments show that this approach is promising.

VI. CONCLUDING REMARKS

ACE has other projects not presented here. In particular, there are several prototyping efforts that combine multiple ACE technologies. One of them is an accelerator ensemble with 3 chiplets in a package: an upgraded *EVA* accelerator, an SRAM-based compute-in-memory accelerator, and an FPGA. A second prototype is a large GPU cluster with both photonic and electrical networks showcasing the techniques of Section IV.

REFERENCES

- [1] H. Chen *et al.*, “Allo: A Programming Model for Composable Accelerator Design,” in *PLDI*, 2024.
- [2] J. Tong *et al.*, “FEATHER: A reconfigurable accelerator with data reordering support for low-cost on-chip dataflow switching,” in *ISCA*, 2024.
- [3] J. Zhu *et al.*, “EVA: A 16mm² 1.54TFLOPS Tiled-Based Accelerator for Evolvable Edge Computing,” in *VLSI*, 2025.
- [4] J. Stojkovic *et al.*, “AccelFlow: Decentralized Orchestration of Distributed Accelerators in Microservice Environments,” in *HPCA*, 2026.
- [5] C. Mendis, “The ACT Ecosystem,” <https://act-compiler.github.io>.
- [6] J. Stojkovic *et al.*, “μManycore: A Cloud-Native CPU for Tail at Scale,” in *ISCA*, 2023.
- [7] N. Patel *et al.*, “XFM: Accelerated Software-Defined Far Memory,” in *MICRO*, 2023.
- [8] M. Shen *et al.*, “AutoSSD: CXL-Enhanced Autonomous SSDs for Low Tail Latency,” in *HPDC*, 2025.
- [9] D. Quinn *et al.*, “DReX: Accurate and Scalable Dense Retrieval Acceleration via Algorithmic-Hardware Codesign,” in *ISCA*, 2025.
- [10] —, “LongSight: Compute-Enabled Memory to Accelerate Large-Context LLMs via Sparse Attention,” in *MICRO*, 2025.
- [11] L. Zheng *et al.*, “SGLang: Efficient Execution of Structured Language Model Programs,” in *NIPS*, 2024.
- [12] Z. Ye *et al.*, “FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving,” in *MLSys*, 2025.
- [13] B. Holmes *et al.*, “Rethinking process snapshots for near-warm serverless cold starts,” in *OSDI*, 2026.
- [14] C. Alverti *et al.*, “CXLfork: Fast Remote Fork over CXL Fabrics,” in *ASPLOS*, 2025.
- [15] K. Lee *et al.*, “SecureLoop: Design Space Exploration of Secure DNN Accelerators,” in *MICRO*, 2023.
- [16] S. Banerjee *et al.*, “Sesame: Software defined enclaves to secure inference accelerators with multi-tenant execution,” *arXiv preprint arXiv:2007.06751*, 2020.
- [17] —, “Triton: Software-defined threat model for secure multi-tenant ML inference accelerators,” in *HASP*, 2023.
- [18] M. Ghaniyoun *et al.*, “MicroSampler: A Framework for Microarchitecture-Level Leakage Detection in Constant Time Execution,” in *DSN*, 2025.
- [19] Y. Yang *et al.*, “Pensieve: A Modular Approach for Security Evaluation of Microarchitectural Designs,” in *ISCA*, 2023.
- [20] S. Chattopadhyay *et al.*, “G-QED: Generalized QED Pre-silicon Verification beyond Non-Interfering Hardware Accelerators,” in *DAC*, 2023.