Technical Report 2026-02-01
Internetworking and Media Communications Research Laboratories
Department of Computer Science, Kent State University
http://medianet.kent.edu/technicalreports.html

White Paper: Infrastructure-Introspection for Self-Adaptive Applications in Accelerator-Dense Computing System

Javed I. Khan and Phil Thomas

Department of Computer Science, Kent State University

pthomas|javed@kent.edu

Abstract— Abstract— The emergence of heterogeneous accelerators, fabric-attached memory, and disaggregated computing has exposed limitations in traditional process-centric execution models. In modern accelerator-dense systems, execution viability is often determined by hardware conditions such as memory pressure, device contention, power constraints, and network dynamics rather than by operating-system process state. However, contemporary operating systems and cluster frameworks continue to couple execution identity to host-bound processes, containers, and virtual machines, limiting adaptability to changing resource conditions. This paper introduces a cooperative execution model that redefines execution identity around application-defined semantic execution units. Rather than relying solely on operating-system mechanisms, applications participate in placement, migration, and resource adaptation decisions using hardware telemetry and semantically valid execution boundaries. Computation can therefore pause, relocate, and resume across heterogeneous resources while preserving correctness. To support this approach, we present a framework for metric-driven execution rebinding and application-directed resource adaptation. Experimental evaluation on heterogeneous cluster environments demonstrates that semantically bounded execution mobility improves resource utilization, adaptability, and resilience compared to traditional process-centric approaches. The results suggest a new execution paradigm in which execution viability, rather than process persistence, becomes the primary abstraction for managing computation in future accelerator-rich system

I. INTRODUCTION

The rapid adoption of heterogeneous accelerators, fabric-attached memory, and disaggregated computing architectures has exposed fundamental limitations in traditional process centric execution models. In modern accelerator-dense systems, execution viability is increasingly determined by hardware-level conditions including accelerator memory pressure, device contention, power constraints, and fabric dynamics rather than solely by operating-system process liveness. Despite these changes, contemporary operating systems and cluster-management frameworks continue to define execution identity through host-bound processes, containers, and virtual machines, coupling execution continuity to operating-system state and externally imposed placement decisions. This mismatch introduces unnecessary migration overhead, obscures critical hardware telemetry from applications, and limits the ability of systems to adapt to dynamic resource conditions.

This paper argues that execution identity should be redefined around semantically meaningful, application-exposed execution units rather than operating-system processes. It introduces a cooperative execution model in which applications actively participate in placement, migration, and resource adaptation decisions through awareness of hardware-level telemetry and application defined execution boundaries. Within this model, computation can pause, relocate, and resume at semantically valid points, enabling fine-grained execution mobility across heterogeneous accelerators and distributed resources while preserving correctness. The proposed approach transforms migration from a heavyweight recovery mechanism into a lightweight, continuous capability for maintaining execution viability under changing hardware conditions.

Cite this document as : Javed I. Khan and Phil Thomas, *White Paper: Infrastructure-Introspection for Self-Adaptive Applications in Accelerator-Dense Computing System*, Technical Report 2026-02-01, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, February 2026, Available at: http://medianet.kent.edu/technicalreports.html

To support this vision, this paper defines a framework for metric-driven execution rebinding and application-directed resource adaptation that enables dynamic coordination between execution semantics and system-level telemetry. Through experimental evaluation on heterogeneous cluster environments, the research demonstrates how semantically bounded execution mobility can improve adaptability, resource utilization, and execution resilience compared to traditional process-centric approaches. The results establish a foundation for future computing systems in which execution viability, rather than process persistence, serves as the primary invariant governing computation across accelerator-dense and fabric-connected infrastructures.

Contemporary computing systems are undergoing a structural transformation. Domain-specific accelerators—GPUs, TPUs, NPUs, DPUs, FPGAs, and specialized inference engines—have evolved from peripheral co-processors into primary computational substrates. Simultaneously, memory disaggregation and fabric-attached architectures (e.g., CXL-based pooling) are dissolving traditional locality boundaries between compute and memory. In these environments, execution viability is increasingly governed by:

1) Accelerator memory capacity and residency pressure
2) Device-level scheduling and partitioning mechanisms
3) Power envelopes and thermal constraints
4) Interconnect topology (PCIe, NVLink, RDMA fabrics)
5) Fabric-level congestion and remote memory latency Forward progress is no longer determined solely by CPU scheduling or OS process liveness. It is constrained by heterogeneous, dynamic hardware-level conditions that evolve over time and vary across devices, nodes, and fabrics. Sustained execution is therefore a function of maintaining viability within these constraints rather than merely preserving process liveness. Yet the dominant execution abstraction remains process-centric.

Operating systems continue to define the process as the unit of execution identity, scheduling, isolation, and migration. Cluster managers operate over containers or virtual machines. Migration mechanisms preserve continuity by reconstructing OS-visible state. Even accelerator virtualization and unified memory systems retain host-bound process identity. These abstractions implicitly assume that execution is opaque, externally managed, and bound to a stable host context in which resource visibility and control are mediated entirely through system software layers. This paper argues that this abstraction boundary is fundamentally misaligned with modern heterogeneous hardware realities. As accelerator density increases and system behavior becomes dominated by devicelevel constraints and fabric dynamics, process-centric execution fails to provide the necessary expressiveness, adaptability, and efficiency required for sustained forward progress. More critically, it obscures the application from the very resource signals that determine its viability, preventing it from participating in decisions that directly impact its own correctness and performance.

OS–Hardware Mismatch

The misalignment between process-centric abstractions and accelerator-dense hardware manifests in four structural tensions.

1) Identity Inflation Process identity includes virtual address spaces, page tables, scheduler metadata, file descriptors, and device runtime context. When migration is required, these artifacts must be reconstructed—even when they are semantically irrelevant to the computation's logical state. As accelerator memory capacities and internal context sizes grow, this inflation increases migration complexity, amplifies data movement, and introduces fragility by coupling execution continuity to OS-level artifacts that are not intrinsic to the computation itself.
2) Policy Entanglement Execution placement policy is embedded within operating-system schedulers and cluster control planes. These components reason over CPU utilization, queue depth, or fairness objectives—but not over hardware-level safety constraints such as sustained GPU memory pressure, device-level partition exhaustion, or accelerator-specific contention domains. Execution may remain "runnable" from the OS perspective while being non-viable at the accelerator level. This disconnect reveals a fundamental limitation: placement decisions are made externally to the computation, without access to the semantic boundaries or internal structure necessary to make correct, timely decisions under dynamic hardware conditions.
3) Fabric Blindness Memory disaggregation and fabricattached accelerators introduce dynamic locality and remote access semantics. Data placement, access latency, and bandwidth availability become time-varying properties of the system rather than fixed characteristics. Yet execution continuity remains bound to host processes that assume stable locality. Hardware-level viability signals—such as congestion, remote memory latency, or interconnect contention—are observable but not

elevated into first-class execution control primitives. As a result, systems are unable to react to fabric-level dynamics in a manner that preserves both performance and correctness.

4) Coarse-Grained Migration Semantics Existing migration techniques—virtual machine live migration, container restart, and checkpoint/restart—treat migration as heavyweight reconstruction of OS-visible execution context. These approaches incur significant latency, redundant data movement, and disruption to execution flow. They do not support fine-grained, semantically bounded rebinding of computation, where only the logically necessary state is transferred and execution resumes at welldefined boundaries. Migration is therefore treated as an exceptional event rather than a continuous mechanism for maintaining execution viability.

The problem is therefore deeper than scheduler design. It is the definition of execution identity itself.

Research Gap and Core Question

Despite extensive work in operating system scheduling, cluster management, and migration techniques, existing systems continue to treat execution as an opaque, externally managed entity. Prior approaches focus on improving scheduling heuristics, optimizing resource allocation, or reducing migration overhead, but they fundamentally preserve the assumption that the application is unaware of and uninvolved in execution placement decisions. This creates a critical gap: while modern hardware exposes increasingly rich and dynamic signals regarding execution viability—such as accelerator memory pressure, device contention, and fabriclevel congestion—these signals are not accessible to, nor actionable by, the application itself. As a result, execution decisions are made without incorporating the entity that has the most accurate understanding of its own computational structure and correctness constraints. This paper addresses the following core question: How can execution be redefined such that applications are able to observe hardware-level conditions and actively participate in fine-grained decisions regarding execution placement, migration, and resource adaptation across heterogeneous systems? This paper asserts that execution in heterogeneous, accelerator-dense systems should be defined in terms of semantically meaningful, applicationexposed execution units that enable dynamic, metric-driven relocation and resource adaptation. By allowing applications to observe hardware-level conditions and participate directly in placement and resource management decisions, systems can achieve more efficient, robust, and adaptable execution than is possible under traditional process-centric, externally managed models. In current systems, execution identity is defined externally by the operating system as an opaque process, and all placement, scheduling, and migration decisions are imposed upon it without knowledge of its internal semantic structure. This dissertation challenges that assumption by proposing that execution identity should instead be defined in terms of semantically meaningful, application-exposed execution units. These units represent the smallest boundaries at which computation can safely pause, relocate, and resume while preserving correctness. By exposing these boundaries, the application becomes an active participant in execution management rather than a passive entity subject to external control. Execution units provide a mechanism through which applications can surface their internal structure, enabling systems to reason about relocation, placement, and resource utilization at a level aligned with the computation itself. Crucially, this model extends beyond relocation to encompass dynamic, applicationdriven resource adaptation. Applications are no longer limited to consuming resources assigned by external schedulers; instead, they are capable of observing hardware-level conditions and actively participating in the expansion, contraction, or redistribution of resources in response to runtime needs. Within this model, execution is no longer statically bound to a host process nor solely governed by OS-level schedulers. Instead, it is dynamically rebindable across nodes and heterogeneous accelerators, with movement occurring at semantically valid boundaries and driven by real-time system telemetry. The role of the system evolves from enforcing placement to coordinating with application-defined execution units, enabling a cooperative control plane in which hardware conditions and application semantics jointly determine execution behavior. This redefinition transforms migration from a heavyweight, reactive mechanism into a lightweight, continuous capability intrinsic to execution itself. It enables fine-grained adaptation to hardware variability, improves resource utilization across heterogeneous accelerators, and provides a foundation for systems in which execution viability—not process persistence—is the primary invariant. Contributions of this Dissertation This paper makes the following primary contributions. First, it introduces a new execution abstraction based on semantically meaningful, application-defined execution units that decouple execution identity from traditional process-centric representations. Second, it presents a cooperative control model in which applications actively participate in execution placement and resource management decisions. Third, it defines a mechanism for fine-grained, metric-driven execution mobility across nodes and heterogeneous accelerators.

Fourth, it demonstrates how application-driven resource adaptation enables more efficient utilization of acceleratordense systems. Finally, it establishes a foundation for rethinking execution in disaggregated and heterogeneous environments, in which execution viability is maintained through continuous coordination between application semantics and system-level telemetry.

II. EXECUTION ABSTRACTION

This section formalizes the execution abstraction underlying Fabric-Introspective Execution Capsules (FIECs). The goal is to define an execution identity independent of operatingsystem process state while preserving correctness under crossnode rebinding governed by hardware-level viability constraints. Execution is defined in terms of semantically complete execution boundaries and a partitioned global state model. This formulation enables execution to be transferred across substrates using only bounded semantic state, without migrating or reconstructing operating-system or device runtime context. Rebinding decisions therefore operate on semantic execution units rather than on processes, virtual machines, or containers.

A. Explicit State Partition

Let the total execution state of a running computation be:

$$\Gamma = S_{\text{semantic}} \cup S_{\text{os}} \cup S_{\text{device}}. \qquad (1)$$

Here S_{semantic} represents computation-critical state required for forward semantic progress, S_{os} operating-system–visible state (e.g., process identifiers, page tables, scheduler metadata, file descriptors), and S_{device} device and runtime artifacts (e.g., GPU register files, driver context, internal scheduling state). Traditional migration mechanisms preserve execution continuity by transferring or reconstructing subsets of S_{os} and S_{device}. In contrast, execution identity is defined exclusively by S_{semantic}. Observation. For a broad class of deterministic or controlled accelerator workloads, the final observable output depends only on S_{semantic}. Operating-system and device state ($S_{\text{os}} \cup S_{\text{device}}$) may influence execution timing or intermediate behavior but do not affect final observable output at semantic execution boundaries. This separation enables execution continuity through bounded semantic state transfer without operating-system process reconstruction, provided execution between boundaries is deterministic or externally controlled and externally visible effects occur only at boundary completion.

B. Fabric-Introspective Execution Capsule (FIEC)

A Fabric-Introspective Execution Capsule (FIEC), denoted U_i, is a semantically complete execution unit that can execute on any compatible substrate while preserving observable output and boundary state:

$$\text{Exec}_A(U_i) \equiv \text{Exec}_B(U_i) \qquad (2)$$

with respect to observable output and boundary state, where:

- I_i is a read-only input region,
- O_i is a write-only output region,
- $\sigma_i \subseteq S_{\text{semantic}}$ is boundary-complete semantic state sufficient for resumption,
- H_i encodes hardware-independent execution metadata. Execution proceeds as an ordered sequence of capsules:

$$E = \langle U_0, U_1, \ldots, U_n \rangle.$$

A capsule represents the minimal semantically complete execution unit whose state is sufficient to resume execution on any compatible substrate without reconstructing operatingsystem or device runtime state. Capsules satisfy the following invariants:

- Determinism (or controlled execution): identical (I_i, σ_i, H_i) produce identical (O_i, σ_{i+1}) under deterministic or externally controlled execution.
- Isolation: capsules access only I_i and O_i and do not depend on external mutable state.
- Boundary Completeness: σ_i contains all semantic state required for resumption on a compatible substrate.
- Forward Independence: U_i does not depend on future capsules and can execute independently given (I_i, σ_i, H_i).

These invariants ensure that execution continuity depends only on S_{semantic}, while S_{os} and S_{device} do not affect observable correctness at semantic boundaries.

C. Substrate Compatibility

Substrate Compatibility. Two substrates A and B are compatible, denoted $A \sim B$, if for any FIEC U_i:

$$\text{Exec}_A(U_i) \equiv \text{Exec}_B(U_i)$$

with respect to observable (O_i, σ_{i+1}). Compatibility requires preservation of instruction semantics, memory consistency guarantees, and numerical determinism at capsule boundaries. It does not require identical device runtime state, operatingsystem configuration, or accelerator model.

D. Hardware Viability Constraints

Execution viability is governed by hardware-observable signals. A capsule may be rebound from substrate A to B when:

1) $A \sim B$ (compatibility),

2) sustained hardware pressure violates viability on A, 3) a semantic boundary σ_i has been reached. Return rebinding occurs only if:

$$M(t) < M_{\text{low}} \; \forall t \in [t_1, t_1 + \tau].$$

This hysteresis policy prevents oscillation, ensures bounded rebinding frequency, and guarantees that decisions occur only at semantic execution boundaries. Safety. If $A \sim B$ and the FIEC invariants hold, rebinding preserves final observable output and execution ordering. Liveness. If a compatible substrate remains available and capsules eventually terminate, execution completes. Correctness therefore depends only on S_{semantic} and substrate compatibility, not on operating-system process continuity.

E. Architectural Implication

By defining execution over S_{semantic}, semantic execution continuity is decoupled from operating-system process identity. Hardware-level signals become execution viability constraints, while the operating system remains responsible for isolation, protection, and coarse-grained resource allocation. Execution identity therefore becomes a property of semantic state progression rather than of a specific operating-system process.

III. SYSTEM ARCHITECTURE

Section 2 defined Fabric-Introspective Execution Capsules (FIECs) and the separation between semantic and nonsemantic state. This section describes the architecture that realizes this abstraction without modifying the operating system.

The core design principle is strict role separation between the operating system and the FIEC runtime:

- The operating system enforces isolation, protection, virtual memory, and device mediation.
- The FIEC runtime governs semantic execution continuity, hardware introspection, and rebinding policy.

This separation decouples execution identity and placement policy from operating-system process context, allowing placement to be governed by hardware-level viability constraints without kernel modification or OS-level migration mechanisms.

A. Architectural Overview

The architecture separates semantic execution identity, hardware viability evaluation, and rebinding policy across application logic, runtime control, hardware telemetry, and operatingsystem services.

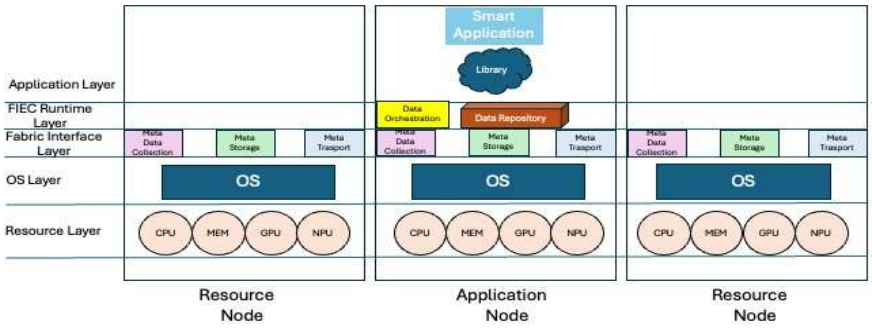

Fig. 1. FIEC architecture showing the application node and resource nodes connected through the Fabric Interface Layer and FIEC runtime.

Figure 1 illustrates the system across an application node and multiple resource nodes, organized into four layers:

1) Application Layer. Expresses computation as a sequence of FIECs with explicit semantic boundaries.
2) FIEC Runtime Layer. Maintains semantic state, enforces boundaries, evaluates viability, and performs crossnode rebinding.
3) Fabric Interface Layer (FIL). Exposes normalized hardware telemetry for viability assessment.
4) Operating System and Resource Layer. Provides isolation, memory management, and device mediation while exposing hardware resources as execution substrates.

The application node hosts runtime control and semantic execution management, while resource nodes supply execution substrates and export hardware-observable state through the FIL. Execution identity is defined by semantic capsule state rather than by any operating-system process. Unlike VM migration or checkpoint/restart systems, execution continuity is defined exclusively by semantic state, and rebinding occurs only at semantic execution boundaries.

B. Fabric Interface Layer

The FIL exposes hardware-semantic signals required for viability assessment. Rather than raw device state, it exports a normalized hardware vector:

$$H(t) = \langle M(t), P(t), L(t), C(t), \ldots \rangle$$

where $M(t)$ is memory pressure, $P(t)$ power or thermal constraints, $L(t)$ fabric latency, and $C(t)$ resource contention or utilization.

This abstraction enables rebinding policy to operate on hardware viability rather than device-specific metrics.

C. Policy Separation

Traditional systems embed placement and migration policy within operating-system schedulers or orchestration frameworks. Here, the operating system provides protection and resource mediation, while execution identity and placement policy are implemented in the FIEC runtime.

D. Boundary-Constrained Rebinding

Rebinding is permitted only at semantic boundaries σ_i. At each boundary, the runtime evaluates:

1) Substrate compatibility,

2) Hardware viability constraints from $H(t)$, 3) Declarative FIEC policies.

Rebinding transfers only boundary-complete semantic state and does not transfer kernel or device runtime state. Execution resumes using only semantic state.

E. Decentralized Control

The architecture does not require a global scheduler. Rebinding decisions emerge from interactions between FIEC constraints, hardware signals, and compatible substrates, reducing coordination overhead and aligning placement with hardware viability.

F. Hysteresis and Stability

To prevent oscillation under transient pressure, the runtime employs hysteresis thresholds:

$$M_{\text{high}} > M_{\text{low}}.$$

Execution becomes non-viable only when sustained pressure exceeds M_{high} for dwell interval τ, and return rebinding requires sustained recovery below M_{low}. This ensures bounded rebinding frequency and stability.

G. Architectural Implications

The architecture shifts execution identity from operatingsystem processes to semantic execution units governed by hardware-level viability constraints. Execution continuity is defined by semantic state progression rather than operatingsystem process context, enabling cross-node rebinding without kernel modification.

IV. IMPLEMENTATION AND EXPERIMENTAL METHODOLOGY

This section describes the experimental platform, runtime implementation, and trigger-driven evaluation used to validate hardware-introspective execution. The experimental design targets three properties: (1) execution continuity independent of host-local process identity, (2) relocation driven by hardware viability constraints, and (3) migration cost bounded by semantic state size rather than process or memory footprint. The trigger regimes isolate these properties under controlled conditions. The runtime rebinds execution when sustained hardware signals violate declared viability constraints. Evaluation therefore focuses on three controlled hardware-trigger regimes:

- Trigger 1 — Static Memory Pressure
- Trigger 2 — Static GPU Pressure
- Trigger 3 — Oscillating GPU Capability

Each regime represents a distinct hardware-viability condition observable through the Fabric Interface Layer (FIL) and evaluates execution continuity under memory pressure, accelerator pressure, and time-varying capability.

A. Hardware Platform

Experiments run on commodity x86 servers connected via low-latency Ethernet. Each node provides:

- Multi-core x86 CPU,
- NVIDIA A40 GPU (48GB),
- Ubuntu 22.04 LTS, • NVMe-backed storage.

Nodes run independent operating systems with no shared filesystem, container checkpoint/restore, or VM migration. All coordination, telemetry, and rebinding logic reside in user space. This isolates runtime behavior from operating-system migration mechanisms. Experiments use a three-node cluster:

- Node A (initial substrate),
- Node B (remote$_1$),

- Node C (remote$_2$).

Nodes are physically homogeneous in processor, accelerator, and memory configuration. Controlled heterogeneity is introduced only in Trigger 3 through runtime modulation of GPU memory and power limits.

B. Runtime Components

The runtime consists of the Fabric Interface Layer, Execution Controller, and Capsule Manager.

a) Fabric Interface Layer: FIL samples hardware telemetry including CPU utilization, system memory, GPU memory, and GPU power (nvidia-smi). Metrics are normalized and written to a lightweight SQLite store containing the latest snapshot (CPU idle, free memory, GPU usage, GPU power). Only the most recent state is required for decisions; historical traces are logged separately for evaluation.

b) Execution Controller: The controller detects sustained threshold violations and performs rebinding using hysteresis. Decisions occur only at semantic boundaries, preserving correctness without transferring OS or device state.

c) Capsule Manager: The Capsule Manager maintains minimal semantic state (S_{semantic}) required for suspension and resumption. Only boundary-complete semantic state is transferred; process, memory, and device runtime state are not.

d) Runtime Control Components: The runtime is a lightweight control plane consisting of a controller, metadata store, capsule repository, node agents, and execution launcher. The controller evaluates viability and triggers rebinding; the metadata store tracks telemetry and placement; the capsule repository stores semantic state. Node agents handle telemetry, execution, and transfer. The execution launcher instantiates capsules on the destination using only semantic state, ensuring that continuity is governed by semantic progression rather than host-local process identity.

C. Trigger 1 — Static Memory Pressure A substrate is non-viable when:

$$M_{\text{sys}}(t) > M_{\text{high}}$$

for dwell interval τ. Rebinding occurs only at semantic execution boundaries. Two workloads are evaluated:

a) Container Relocation: A containerized workload saturates CPU and memory. When pressure exceeds the threshold, execution suspends and resumes on a remote node as a single semantic unit.

b) Conjugate Gradient (CG): The CG solver executes with iteration-aligned capsule boundaries. Each iteration forms a capsule; relocation preserves monotonic solver progress.

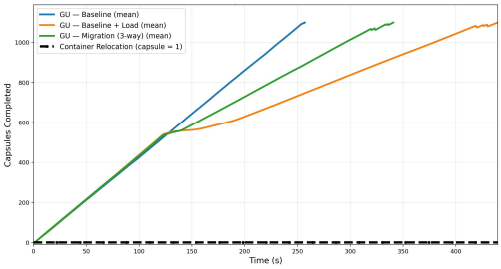

Fig. 2. Capsules completed versus time under Trigger 1 (Static Memory Pressure). The CG workload progresses through multiple capsules, while the container relocation experiment appears as a single capsule. Curves show mean progression across five runs for Baseline, Baseline+Load, and Migration (3-way).

Figure 2 shows reduced progress under memory contention and recovery under migration. The container relocation experiment moves a single semantic unit, while CG migrates across iteration boundaries. Migration therefore restores forward progress without requiring process reconstruction.

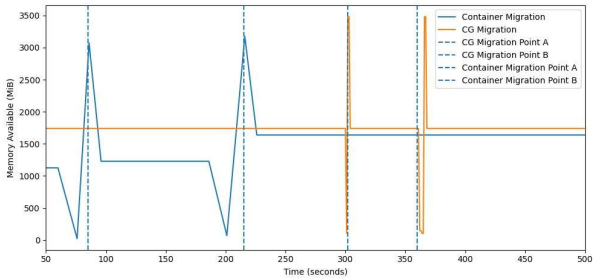

Fig. 3. Available system memory versus time under Trigger 1. Vertical dashed lines indicate rebinding events triggered by sustained memory pressure.

Figure 3 shows pressure-driven rebinding and recovery on the source node. Sustained system memory pressure triggers relocation only at semantic boundaries, indicating that movement is governed by hardware viability rather than by operating-system scheduling state.

D. Trigger 2 — Static GPU Pressure

This trigger evaluates execution relocation driven by sustained GPU memory pressure. A substrate is considered nonviable when $M_{gpu}(t) > G_{high}$ for a sustained dwell interval τ. Execution rebinding is evaluated only at semantic execution boundaries. Two accelerator workloads are evaluated to demonstrate capsule-based relocation under GPU memory pressure.

a) Matrix Multiplication (AxB): Dense FP64 matrix multiplication is decomposed into fixed-size tiles, each representing a semantically complete capsule. Sustained GPU memory pressure triggers relocation of capsule execution to a compatible remote node at capsule boundaries.

b) Blocked LU/QR Factorization: Blocked LU and QR factorizations execute using tile-based decomposition with strict dependency ordering. Each tile update forms a capsule boundary, allowing relocation under GPU pressure while preserving strict semantic ordering and monotonic factorization progress.

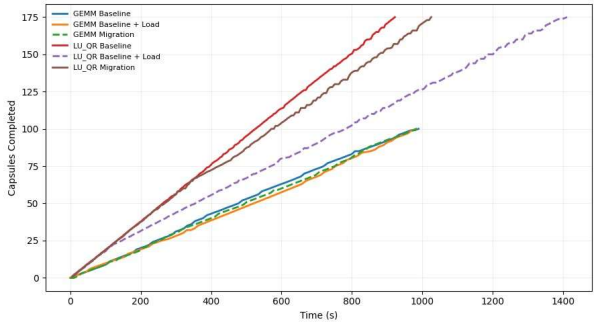

Fig. 4. Capsules completed versus time under Trigger 2 (Static GPU Pressure) for GEMM and LU/QR. Migration restores progress under sustained GPU memory pressure.

Figure 4 shows capsule progress under sustained GPU memory pressure. Baseline+Load reduces capsule completion rates due to GPU memory contention. When migration is enabled, execution relocates to a substrate with available GPU memory, restoring forward progress toward baseline behavior. Both workloads maintain monotonic capsule progression across relocation events, demonstrating that execution identity is preserved across capsule rebinding and that useful work is not discarded under pressure.

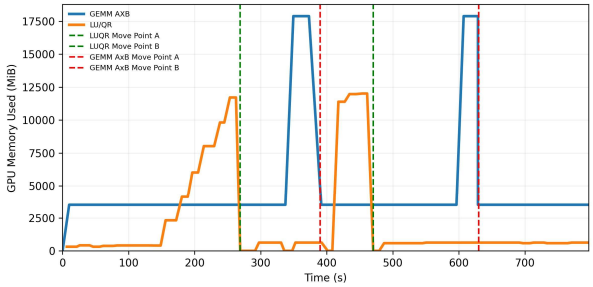

Fig. 5. GPU memory usage over time for GEMM and LU/QR during migration experiments averaged across five runs. Vertical dashed lines mark migration trigger points where sustained GPU memory pressure exceeded the viability threshold.

Figure 5 shows GPU memory pressure leading to relocation events. Sustained GPU memory pressure drives the system beyond the configured viability threshold, triggering capsule rebinding at semantic execution boundaries. After relocation, GPU memory usage drops on the source node while execution continues on the destination substrate. The restoration of forward progress occurs without transferring device state or restarting execution, indicating that GPU-bound workloads can be relocated using only semantic boundary state. This contrasts with GPU virtualization and checkpoint-based approaches, where device context reconstruction or full memory transfer is required. These results demonstrate that capsule execution can be repeatedly relocated across nodes under sustained GPU memory pressure while preserving forward progress and avoiding process or device-state migration.

E. Trigger 3 — Oscillating GPU Capability

GPU memory and power limits are modulated using sinusoidal functions, creating time-varying capability envelopes across nodes. Capsules are rebound to the most viable substrate at semantic boundaries.

TABLE I

GPU MEMORY ENVELOPE MODULATION PARAMETERS

Node	Memory Amplitude (Δ_k)	Period (T_k)
N0	8GB	600s
N1	7GB	760s
N2	6GB	890s

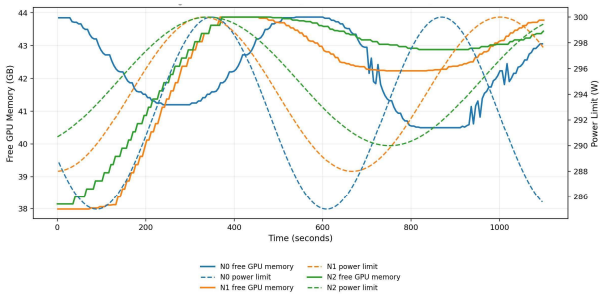

Fig. 6. Cyclic GPU capability envelopes across nodes. Controlled modulation of GPU memory availability and power limits creates time-varying substrate viability conditions.

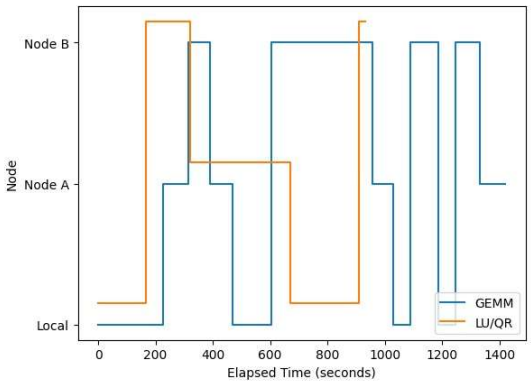

Fig. 7. Execution location versus time under Trigger 3. Step transitions indicate capsule rebinding events across substrates.

Execution relocates as viability envelopes shift. GEMM migrates more frequently due to shorter capsules, while LU/QR shows longer residency periods. These traces demonstrate stable repeated rebinding under time-varying hardware capability without restarting the application.

F. Multi-hop Rebinding

Execution migrates across multiple nodes (A → B → C) as viability changes. Figure 7 shows local, remote, and remoteto-remote transitions, demonstrating that execution continuity remains bound to semantic boundary state rather than to any particular host.

G. Stability and Hysteresis

Rebinding uses hysteresis to avoid oscillation. A node becomes non-viable only after sustained threshold violation and returns only after sustained recovery, ensuring bounded rebinding frequency even under fluctuating hardware conditions.

H. Migration State Size

Only boundary-complete semantic state is transferred, so migration cost scales with capsule state size rather than memory footprint.

TABLE II

REPRESENTATIVE TRANSFERRED SEMANTIC STATE SIZES

Workload	State Size	Notes
Container Relocation	~50–150 KB	Container boundary state
CG Solver	~20–80 KB	Iteration state
GEMM	~10–40 KB	Tile descriptors/results
LU/QR	~30–120 KB	Tile and dependency state

Transferred state remains small (KB–hundreds of KB), far below application memory size. This distinguishes capsule rebinding from process or VM migration, where migration cost scales with memory footprint or device state. Across all triggers, execution continuity remains independent of process identity, relocation is driven by hardware viability, and migration cost remains bounded

by semantic state size. These results support the feasibility of hardware-governed execution continuity without reconstructing operating-system process context.

V. RELATED WORK

This work relates to process migration and checkpointing, accelerator virtualization, memory disaggregation, cluster scheduling, and hardware-aware runtime systems. We position our contribution through the lens of execution identity and state transfer, rather than migration mechanism or scheduling policy alone.

A. Process Migration and Checkpointing

Transparent process migration has long been studied for load balancing and fault tolerance

[Mosberger and Peterson(1996)], [Barak and La'adan(1999)].

Checkpoint and restart systems extend these ideas to modern distributed and containerized environments [Ansel et al.(2009)], [Liu et al.(2018)], [CRIU Project([n.d.])].

These systems preserve execution continuity by capturing and reconstructing operating-system process state, including memory mappings, file descriptors, scheduler metadata, and runtime-managed resources. Execution identity therefore remains tied to the operating- system process, and migration transfers or reconstructs OS-visible and device-visible state.

In contrast, our model defines execution identity exclusively over semantic execution state and transfers only boundarycomplete semantic state. Execution continuity is therefore achieved through semantic state transfer rather than process reconstruction.

B. GPU Virtualization and Accelerator Remoting

A substantial body of work explores GPU virtualization and accelerator sharing in clusters. Systems such as GViM and vCUDA virtualize accelerator access across virtual machines [Gupta et al.(2011)], [Shi and Chen(2012)], while Pegasus improves multi-tenant GPU scheduling and isolation [Wu et al.(2020)]. Remote GPU frameworks such as rCUDA and GridCuda expose accelerators over the network [Duato et al.(2010)], [Kato et al.(2012)].

These approaches decouple device access from locality, but execution remains anchored to host process identity. Continuity is preserved by maintaining or reconstructing OSvisible and device-visible state. Our abstraction instead treats accelerator remoting and virtualization as substrate capabilities within a compatibility relation, while execution identity is defined only by semantic execution state.

C. Memory Disaggregation and CXL Systems

Recent systems investigate fabric-attached and pooled memory architectures. Infiniswap enables remote memory swapping over RDMA fabrics [Gu et al.(2017)], and CXL-based systems such as Pond demonstrate hardware-supported memory pooling [Li et al.(2023)].

These systems expose memory viability constraints not reflected in traditional operating-system scheduling abstractions, but execution remains process-bound. Our model instead treats sustained hardware-level signals, such as memory pressure, as execution-governing viability constraints and redefines execution continuity at semantic execution boundaries.

D. Operating-System Architecture and Resource Disaggregation

Several systems have explored restructuring operatingsystem abstractions to better match modern hardware heterogeneity and disaggregated resources. The Barrelfish multikernel architecture treats the operating system as a distributed system to address scalability and hardware diversity [Baumann et al.(2009)], while LegoOS explores operatingsystem support for hardware resource disaggregation across fabric-connected components [Park et al.(2017)].

These systems decouple operating-system abstractions from physical resource location, but execution identity remains tied to operating- system processes and kernel-managed execution contexts. In contrast, our work leaves the operating system unchanged and introduces an execution abstraction in which execution identity is defined by semantic execution state rather than operating-system process context.

E. Cluster Scheduling and Resource Management

Large-scale cluster schedulers such as Sparrow and Firmament explore low-latency and data-aware task placement [Ousterhout et al.(2013)], [Grandl et al.(2016)], while systems such as Heracles and Quasar optimize resource efficiency and quality-of-service in multi-tenant clusters [Lo et al.(2015)], [Delimitrou and Kozyrakis(2014)].

These systems reason over tasks, containers, virtual machines, or processes as placement units, and relocation typically relies on restart, checkpoint, or virtual-machine migration semantics. Our approach instead operates below scheduler granularity and above device context, where execution identity is defined by semantic execution units rather than schedulervisible tasks.

F. Task- and Object-Based Runtime Systems

Runtime systems such as Ray [Moritz et al.(2018)], Charm++ [Kale and Krishnan(1993)], and actor-based models provide abstractions for distributed and migratable computation.

These systems support task or object mobility, but migration reconstructs runtime-managed state within operating-system processes. Our model instead defines execution identity at semantic execution boundaries and transfers only semantically complete state across substrates.

G. Hardware-Aware, Cross-Layer Scheduling

Several systems incorporate hardware signals into runtime or scheduling decisions. Heterogeneous task schedulers leverage NUMA awareness and accelerator proximity [Kaldewey et al.(2012)], while carbon- aware frameworks adapt placement based on energy signals [Gao et al.(2022)].

These approaches integrate hardware observations into placement decisions but continue to operate over process- or task-bound execution units. Our model instead uses hardware introspection to govern execution viability and cross-node rebinding of semantic execution units.

H. Summary

Prior systems improve migration efficiency, accelerator virtualization, scheduling intelligence, or hardware-aware placement, but they preserve operating-system processes, virtual machines, or runtime-managed tasks as the unit of execution continuity. As a result, relocation typically requires transferring process or device state, restarting execution, or reconstructing runtime context, which limits responsiveness under sustained hardware pressure. This work instead proposes an OS-transcending, hardware introspective execution abstraction in which execution identity is defined by semantic execution state, hardware-observable constraints govern execution viability, and cross-node rebinding transfers only boundarycomplete semantic state rather than operating-system or device runtime state. Across all trigger regimes, the evaluation demonstrates that execution can relocate under memory and accelerator pressure while preserving forward progress, maintaining monotonic execution, and keeping migration cost bounded by semantic state size. These results show that execution can adapt to dynamic hardware conditions in acceleratorrich clusters without process migration or device-state reconstruction, enabling execution placement to follow hardware viability rather than remain anchored to a specific host.

VI. DISCUSSION

The evaluation demonstrates that execution identity can be decoupled from operating-system process context and governed instead by hardware-introspective viability constraints across three regimes: memory-driven rebinding, GPU-driven capsule rebinding, and dynamic hardware capability variation.

Across all regimes, semantic execution remains strictly monotonic, rebinding footprint remains bounded, and no OSlevel migration mechanisms are invoked. Multi-hop experiments (Local → Remote_1 → Remote_2 → Local) demonstrate that execution identity remains stable under repeated rebinding, including remote-to-remote transitions.

These results suggest that execution continuity need not be tied to operating-system processes, virtual machines, or runtime-managed tasks. Instead, execution identity can be defined at semantic execution boundaries and governed by hardware-level viability constraints.

A. Repositioning the OS–Architecture Boundary

Traditional systems anchor execution identity within kernelmanaged abstractions such as processes, containers, and virtual machines. These abstractions simultaneously serve as isolation boundaries and execution continuity anchors. The results in this work demonstrate that these roles can be separated.

In the proposed architecture, the operating system enforces isolation, protection, and scheduling fairness; execution units define semantic continuity; and hardware-introspective signals govern execution viability. Execution continuity therefore does not depend on host affinity, process lifetime, or schedulervisible metadata, but on semantic boundary state.

This repositions execution continuity as an architectural abstraction above the kernel but below cluster schedulers, refining the OS–hardware boundary rather than replacing it.

B. Viability as a Safety Constraint

A central conceptual shift in this work is treating hardwarelevel signals as viability constraints rather than performance hints. Sustained system memory pressure or accelerator memory exhaustion defines viability boundaries beyond which continued execution on a substrate risks failure or severe degradation.

Rebinding occurs only when sustained pressure exceeds configured thresholds for a dwell interval, and return rebinding requires sustained recovery below lower thresholds. This hysteresis model prevents oscillatory behavior and ensures bounded rebinding frequency under repeated pressure events or dynamic capability variation.

Rebinding therefore functions as proactive preservation of semantic execution continuity under sustained hardware constraints rather than reactive failure recovery.

C. Implications for Fabric-Attached Memory and CXL

CXL-based memory pooling systems demonstrate that memory locality is no longer static, but process-centric execution assumes stable, host-bound memory mappings and treats memory pressure primarily as a scheduling problem.

The memory-driven rebinding experiments show that execution can follow sustained memory pressure across nodes without transferring full process images. Because only semantic boundary state is transferred, rebinding cost scales with boundary state size rather than resident memory size.

This suggests that pooled-memory and fabric-attached memory systems may benefit from semantic-boundary-aware execution, where execution relocation follows viability constraints rather than host allocation decisions.

D. Compatibility Across Heterogeneous Substrates

The compatibility relation defined earlier ensures that rebinding preserves instruction and memory semantics across substrates. In the prototype, identical GPUs simplify compatibility, but heterogeneous deployments may require explicit capability descriptors, numerical precision constraints, and runtime verification of accelerator features.

The multi-hop rebinding experiments demonstrate that compatibility is a semantic relation rather than a locality constraint. Execution identity persists across successive compatible substrates even when execution moves between multiple remote nodes.

E. Relationship to Cluster Scheduling

Cluster schedulers such as Borg andOmega [Verma et al.(2015)], [Schwarzkopf et al.(2013)] operate at process or container granularity and optimize placement, utilization, and fairness, but they do not redefine execution identity. The architecture evaluated in this work operates below process granularity and above device context. It can coexist with existing schedulers by introducing hardware-introspective rebinding as a lower-level execution safety mechanism rather than a scheduling policy.

In this layered model, schedulers manage resource allocation, execution units manage semantic continuity, and hardware signals enforce execution viability. This separation preserves compatibility with existing cluster infrastructure while refining the abstraction boundary between scheduling, execution identity, and hardware-level constraints.

F. Broader Implications

Across all scenarios, execution continuity remains stable under repeated hardware-driven substrate transitions. Only bounded semantic state is transferred, and no OS-visible migration occurs.

These results suggest that process identity is an implementation artifact rather than a semantic necessity. As acceleratorrich and fabric-connected systems continue to scale, execution viability will increasingly depend on hardware-level constraints rather than static process state.

By defining execution continuity at semantic boundaries, systems can adapt to dynamic hardware conditions without incurring large migration overheads or kernel-level complexity. OS-transcending, hardware-introspective execution therefore refines the abstraction boundary between software semantics and architectural resource constraints.

VII. LIMITATIONS

The evaluation demonstrates correctness, bounded rebinding state, and control stability across memory-driven, GPU-driven, and dynamically varying hardware viability regimes. However, both the abstraction and the prototype have limitations that constrain the scope of current validation.

A. Compatibility Under Heterogeneous Accelerators

All experiments use identical NVIDIA A40 GPUs, which simplifies the compatibility relation ($A \sim B$) by ensuring identical instruction semantics, numerical precision behavior, and device-memory characteristics. In heterogeneous environments, compatibility enforcement becomes non-trivial. Differences in:

- Numerical precision behavior (e.g., FP32 vs. FP64),
- Instruction-set extensions,
- Memory consistency guarantees, • Device-level partitioning semantics, may invalidate naive rebinding.

While the formal model permits heterogeneous substrates through compatibility constraints, the prototype does not implement automatic capability negotiation or validation. Compatibility enforcement at deployment scale therefore remains an open systems problem.

B. Telemetry Granularity and Reaction Latency

The Fabric Interface Layer samples telemetry at fixed 10second intervals. Hysteresis thresholds ensure stability, but rebinding latency is bounded by sampling resolution plus semantic boundary completion time.

Under bursty workloads, coarse sampling may delay rebinding, while aggressive sampling increases monitoring overhead. The evaluation validates correctness under sustained pressure rather than optimal reaction latency. Adaptive or event-driven telemetry policies are not yet implemented.

C. Control-Plane Scalability

The prototype employs a centralized controller to aggregate telemetry and trigger rebinding. This simplifies correctness reasoning but does not evaluate cluster-scale deployment.

At larger scales, telemetry aggregation may become bandwidth- intensive, decision latency may increase, and distributed consistency mechanisms may be required. The architectural model does not mandate centralization, but scalability beyond small clusters was not evaluated.

D. Workload Structure Assumptions

The evaluation workloads use deterministic, tiledecomposed numerical kernels (GEMM, CG, and LU/QR).

These workloads provide:

- Explicit semantic boundaries,
- Bounded capsule state,
- No external side effects, • No persistent device-resident state.

Applications with fine-grained synchronization, external I/O dependencies, long-lived device-resident state, or irregular control flow may require additional abstraction layers to define boundary-complete execution units. The model is therefore most directly applicable to data-parallel or decomposable longrunning workloads.

E. Security and Governance Assumptions

Cross-node rebinding transfers semantic state between hosts. The prototype assumes a trusted cluster environment and does not implement:

- Cryptographic state authentication,
- Integrity protection,
- Multi-tenant policy enforcement, • Cross-administrative-domain governance.

These mechanisms are orthogonal to the abstraction but would be required for production deployment.

F. Performance Scope

The evaluation emphasizes correctness, bounded rebinding footprint, and hysteresis stability rather than global performance improvement. Memory-driven rebinding preserves forward progress under sustained system memory pressure, GPUdriven rebinding demonstrates correctness under sustained accelerator pressure and repeated multi-hop rebinding, and dynamic capability modulation demonstrates stable execution under continuously varying hardware capability.

Large-scale throughput characterization, energy analysis, or cross-accelerator performance comparisons were not performed.

G. Kernel Non-Modification Tradeoffs

The architecture intentionally avoids kernel modification to isolate execution identity from operating-system process state and simplify deployment. Deeper kernel integration could improve telemetry fidelity, reduce rebinding latency, and enable coordinated scheduling decisions, but tighter OS integration risks reintroducing process-bound execution identity. Balancing closer OS collaboration with preservation of semantic execution identity remains an open design question.

These limitations do not invalidate the central result: execution continuity can be defined independently of operatingsystem process context and governed by sustained hardwarelevel viability constraints. Realizing the abstraction at production scale will require advances in compatibility enforcement, control-plane scalability, and secure semantic state transfer.

VIII. CONCLUSION

Accelerator-rich, fabric-connected systems increasingly expose hardware-level constraints—system memory pressure, accelerator memory exhaustion, power limits, and interconnect variability—that directly determine execution viability. However, prevailing execution models continue to anchor semantic continuity to operating-system process identity, coupling execution to host-bound OS state.

This paper introduced OS-transcending, hardwareintrospective execution units, a cross-layer execution abstraction that separates semantic execution continuity from operating-system process context. Execution identity is defined by bounded semantic boundary state, while sustained hardware-level signals govern cross-node rebinding across compatible substrates.

We formalized global state partitioning, defined a substrate compatibility relation, and established safety and liveness under boundary-constrained rebinding. A user-space prototype requiring no kernel modification demonstrated memorydriven, GPU-driven, and dynamically varying hardware viability regimes, including repeated multi-hop rebinding across nodes. Across all experiments, execution remained strictly monotonic, rebinding footprint remained bounded, and no OSlevel migration or process reconstruction occurred.

These results demonstrate that execution continuity can be defined independently of operating-system process state and governed instead by sustained hardware-level viability constraints. As fabric-attached memory, accelerator specialization, and heterogeneous compute fabrics continue to evolve, execution placement increasingly becomes a hardware-viability problem rather than solely a scheduler-placement problem.

OS-transcending, hardware-introspective execution therefore refines the system abstraction boundary, enabling execution to follow sustained hardware viability across substrates rather than remaining anchored to operating-system process identity.

REFERENCES

[Ansel et al.(2009)] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 1–12.

[Barak and La'adan(1999)] Amnon Barak and Oren La'adan. 1999. The MOSIX Multicomputer Operating System. *Future Generation Computer Systems* 13, 4 (1999).

[Baumann et al.(2009)] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS¨ Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 29–44.

[CRIU Project([n.d.])] CRIU Project. [n.d.]. Checkpoint/Restore In Userspace (CRIU). https://criu.org. Accessed 2026-02-15.

[Delimitrou and Kozyrakis(2014)] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 127–144.

[Duato et al.(2010)] Jose Duato, Antonio J. Pe´ na, Federico Silla, Rafael˜ Mayo, and Enrique S. Quintana-Ort´ı. 2010. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*. 224–231.

[Gao et al.(2022)] Peter Gao, Adam Wierman, and Zhenhua Liu. 2022. Carbon-Aware Computing for Data Centers. In *Proceedings of the 13th ACM International Conference on Future Energy Systems (e-Energy)*.

[Grandl et al.(2016)] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[Gu et al.(2017)] Juncheng Gu, Youngqiang Li, Yifeng Zhou, and Mosharaf Chowdhury. 2017. Infiniswap: Fast Remote Memory Swapping in Data Centers. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[Gupta et al.(2011)] Vishal Gupta, Ada Gavrilovska, and Karsten Schwan. 2011. GViM: GPU-Accelerated Virtual Machines. In *ACM Symposium on Cloud Computing*.

[Kaldewey et al.(2012)] Tim Kaldewey, Ganesh Ananthanarayanan, and Andrew S. Birrell. 2012. Heterogeneous Task Scheduling for Accelerators. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*.

[Kale and Krishnan(1993)] Laxmikant V. Kale and Sanjeev Krishnan. 1993. Charm++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 91–108.

[Kato et al.(2012)] Satoshi Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. 2012. TimeGraph and GridCuda: GPU Virtualization. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*.

[Li et al.(2023)] Huaicheng Li, Daniel S. Berger, Thomas Moscibroda, and Christina Delimitrou. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[Liu et al.(2018)] Haikun Liu, Hai Jin, and Xiaofei Liao. 2018. Container Migration for Cloud Computing: A Survey. In *IEEE International Conference on Communications (ICC)*.

[Lo et al.(2015)] David Lo, Lingjia Cheng, Ramanathan Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. 450–462.

[Moritz et al.(2018)] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 561–577.

[Mosberger and Peterson(1996)] David Mosberger and Larry L. Peterson. 1996. Making Paths Explicit in the Sprite Process Migration System. In *USENIX Winter Technical Conference*.

[Ousterhout et al.(2013)] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*.

[Park et al.(2017)] Yongjun Park, Sangeetha Seshadri, Juhyung Lee, Tianyi Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 69–87.

[Schwarzkopf et al.(2013)] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 351–364.

[Shi and Chen(2012)] Liang Shi and Hao Chen. 2012. vCUDA: GPU

Virtualization for High Performance Computing. In *IPDPS Workshops*.

[Verma et al.(2015)] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 18:1–18:17.

[Wu et al.(2020)] Jialin Wu, Zhihao Jia, Yongwei Wu, and Ion Stoica. 2020. Pegasus: Coordinated Scheduling for GPU Sharing. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.