Cross-Phase Blending: Why AI Fabrics Need Phase-Aware Transport

Soudeh Ghorbani
Microsoft Research & Rutgers University

Ying Zhang
Meta

Abstract

Datacenter congestion control is built around a local continuity premise: recent feedback from a flow is a useful summary for the next control action. AI training weakens this premise. Its communication is neither an unstructured stream nor a clean periodic signal. Collectives recur with recognizable structure, but host jitter, GPU variability, queueing, PFC, and multi-job interference move phase boundaries and change amplitudes. The result is a regime mismatch: a controller's recent-history state may describe the phase that just ended, not the phase that is starting.

We call the resulting failure *cross-phase blending*. It causes slow ramp-up into high-demand phases and slow release after them. This is not simply a matter of faster feedback: even one feedback horizon of stale state misallocates more bytes as link rates grow, and the penalty repeats at the same phase boundaries across thousands of training iterations. We argue that AI fabrics need *phase-indexed transport state*. PICCA is a transport architecture that lets existing congestion-control algorithms run over separate histories for recurring execution phases, using phase information from application hints, observation, or a global oracle. Production RoCE traces show phase structure with jitter. Preliminary ns-3 prototypes in which an oracle supplies phase information show the headroom: phase-indexed DCQCN and MLTCP reduce tail flow-completion time and boundary errors. We outline the challenges of capturing it without an oracle. The broader claim is architectural: future AI transports should remember by phase, not only by time.

CCS Concepts

• **Networks** → **Transport protocols**; **Data center networks**; *Network performance analysis.*



This work is licensed under a Creative Commons Attribution 4.0 International License.

HotNets '26, Salt Lake City, UT, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859954

Keywords

Congestion control, AI datacenters, distributed training, phase-indexed transport, collective communication, cross-phase blending

1 Introduction

AI training has turned datacenter networks into part of the computing substrate. As accelerators get faster and models become larger and more distributed, communication becomes harder to hide behind computation. Congestion control decisions therefore increasingly affect when a training phase finishes and when the next one can begin.

Congestion control is built on a simple abstraction: the recent past is a useful guide to the near future. A controller samples RTTs, ECN marks, queue occupancy, delivery rate, loss, INT feedback, or receiver grants; filters those observations over a short history; and updates a rate, window, or grant budget. DCTCP [1], DCQCN [19], TIMELY [10], Swift [7], HPCC [9], and receiver-driven transports [6, 11] differ in the signals they use and the control law they apply, but they share a state abstraction: one recent-history state per connection (flow or QP). This abstraction works when the underlying congestion process is *temporally continuous,* so smoothing recent samples reduces noise without changing the meaning of the estimate.

AI training traffic stresses this abstraction in a new way. A training iteration is not a stationary stream of independent flows. It is a sequence of execution phases: compute intervals, tensor-parallel collectives, data-parallel all-reduces, all-to-all exchanges, pipeline transfers, optimizer traffic, and background interference. These phases recur across iterations, but they can have very different communication demand on the same path or endpoint. A flow that was correctly conservative in one phase may be too slow in the next; a state that was correctly aggressive in one collective may be harmful after the fabric should have been released to another collective or job.

The important nuance is that AI traffic is *phase-structured*, not simply periodic. A rigid periodic model would say that the same burst shape returns at fixed wall-clock offsets. In practice, host scheduling, GPU kernel variation, PCIe effects, queueing, PFC, and interference move phase boundaries and change phase duration and amplitude. This places AI traffic between two familiar abstractions. Periodic mechanisms can

exploit a fixed clock; conventional feedback controllers can smooth locally continuous noise. AI traffic offers recurring structure, but with boundaries that drift.

This paper argues that the missing abstraction is *phase*. The controller should not ask only "what happened recently?" It should also ask "which execution phase am I in now?" If recent samples came from a different phase, they may be actively misleading. A single scalar history can then become a blend of incompatible regimes rather than a noisy estimate of the current one.

We call this failure mode *cross-phase blending*. At an upward transition, stale low-rate history causes slow ramp-up into a high-demand phase. At a downward transition, stale high-rate history causes slow release, injecting excess traffic into a period that could have been used by another collective or job. The cost of this mistake grows with link speed. The controller cannot correct stale state until feedback from the new phase returns (typically at least an RTT), and that feedback horizon has a physical floor because signals must traverse the fabric. As links get faster, this fixed-time horizon contains more bytes and occupies a larger fraction of shrinking communication phases. Since training is iterative, this is a recurring cost through thousands of iterations rather than a one-time transient. Figure 1 illustrates the scaling effect. For a 1 MB communication phase, a 15μs feedback horizon covers 18% of the phase at 100 Gbps but 75% at 400 Gbps; the bytes sent under stale state grow from 187 KB to 750 KB. Since training repeatedly revisits the same communication phases, this stale-control region recurs across iterations. In §5, we show that with oracle phase information, phase indexing reduces p99 FCT by 22% at 200 Gbps and by 94% at 1600 Gbps, an estimate of the headroom available to practical phase sources. This is a training-time problem, not only a packet-level artifact. In synchronous training, the slowest exposed communication phase can gate the next iteration and remain a substantial fraction of batch time [14]. Thus, as faster links make stale control a larger fraction of the communication window, correcting the state at phase boundaries can translate into iteration-time savings.

Existing work recognizes that machine learning workloads have structure, but uses that structure in different layers. Cassini [12] and MLTCP [13] exploit iteration progress and collective timing to schedule or interleave communication. Modern datacenter transports improve the speed or precision of feedback using ECN [1, 17–19], delay [7, 10, 16], INT [9], delivery-rate estimates [3], receiver-driven grants or rate hints [6, 11, 18], or explicit network notifications of congestion and impending capacity release [2]. These directions are valuable, but they leave the transport history scalar. Our claim is orthogonal: even if a scheduler chooses a good time to run a collective, and even if the transport observes feedback quickly, the controller can still carry the wrong

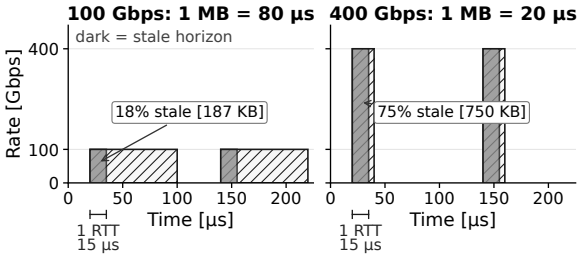


Figure 1: As link speeds increase, cross-phase blending affects a larger fraction of each communication phase. The stale-control region grows from 18% at 100 Gbps to 75% at 400 Gbps.

history into the handoff. AI fabrics need the transport state itself to be indexed by phase.

We propose PICCA, a phase-indexed congestion control architecture. PICCA is not a new congestion control algorithm (CCA). It is a state-management layer that lets an existing CCA run over phase-indexed histories. Instead of one recent-history state per flow or QP, PICCA maintains a small table of state slots, one per recurring phase. Each slot stores the native variables of the base controller: rate or window, RTT baseline, ECN history, pacing gain, receiver-grant budget, or other CCA state. On a phase transition, PICCA activates the slot associated with the new phase rather than adapting from the immediately preceding one; the base CCA then continues to react normally to live feedback.

Phase identifiers can come from runtime or collective-library hints, endpoint observation, or, at the oracle extreme, a global controller with a system-wide view of active collectives and topology (§4). This spectrum is intentional: PICCA is an architectural claim about what state should be remembered, not a commitment to one phase-discovery mechanism.

We ground the claim in three steps. First, we analyze a host-side Tx/Rx trace from a production RoCE AI fabric and show that the traffic is structured enough for phase history to be useful but jittered enough that a fixed-period controller is insufficient. Second, we formalize cross-phase blending and show how scalar history creates slow ramp-up and slow release at phase boundaries. Third, we estimate the headroom with ns-3 prototypes where an oracle gives DCQCN and MLTCP phase information, showing that phase-indexed state improves tail behavior and composes with progress-aware transport; we then outline the challenges of capturing it without one.

The goal is not to present a fully baked transport. The goal is to challenge a state abstraction that has become implicit in datacenter congestion control. AI fabrics are no longer carrying traffic whose next control action should be determined only by the most recent samples. They are carrying recurring, jittered execution phases. Future AI transports should therefore remember by phase, not only by time.

2 AI Traffic is Phase-Structured

This section establishes the empirical premise behind PICCA. We show three facts. First, the traffic generated by AI training is not an unstructured stream: communication phases recur with recognizable burst shapes. Second, this recurrence is not a rigid wall-clock period: burst start times, durations, and amplitudes vary across cycles. Host scheduling, GPU kernel variation, PCIe effects, queueing, PFC, and interference from other jobs introduce jitter in when phases begin, how long they last, and how much traffic they generate. Third, this combination matters for congestion control because it lies between two common abstractions. Periodic schedulers can exploit a fixed clock, and conventional feedback controllers can smooth locally continuous noise, but neither abstraction directly captures recurring phases whose boundaries move.

Trace and measurement context. We analyze a representative host-side Tx/Rx trace from a production RoCE AI fabric of the kind characterized in prior work [5]. That fabric is built as a dedicated RDMA training network using two-tier leaf–spine zones, with hosts attached through high-speed RNICs and workloads spanning ranking, vision, and LLM training jobs. The trace records outgoing (Tx) and incoming (Rx) traffic intensity at fine time granularity on a host/NIC, exposing the burst structure that would be hidden by coarse aggregate link counters. We use this trace as a minimal example: the point is not that every job has the same period, but that the traffic observed by the transport alternates among recurring communication regimes.

Figure 2 summarizes the result. In the time domain, Tx and Rx both contain sharp bursts separated by lower-demand intervals. The bursts recur, but not at exactly fixed offsets: their boundaries drift, and their duration and intensity vary across cycles. A rigid periodic model would predict that the same burst shape reappears at fixed wall-clock offsets. The trace instead shows recurring structure with jittered boundaries. This distinction matters because an emerging class of AI-specific mechanisms exploits iteration periodicity; such mechanisms are useful, but a controller that assumes exact periodic alignment can be brittle when phases arrive early, late, shorter, longer, or with different amplitude.

The lag- and frequency-domain views make the same point from a different angle. The autocorrelation function (ACF) compares the trace to time-shifted copies of itself: sharp peaks at multiples of a period indicate a rigid clock, while broad or decaying lobes indicate approximate recurrence with drift. The power spectrum decomposes the trace into frequency components: a narrow spectral line would indicate a nearly periodic signal, while energy spread around the peak indicates timing and amplitude jitter. In Figure 2, the ACF has recurring but broad, decaying lobes, and the PSD has a workload heartbeat near f_0 whose energy is spread rather than concentrated in a narrow line. Thus, the traffic is structured enough for phase history to be useful, but not regular enough for a fixed-period controller to be sufficient.[1]

The phase-domain view exposes why this structure is problematic for conventional congestion control. By phase domain, we mean aligning samples by the recurring execution phase they belong to, e.g., a low-demand compute interval, a high-rate collective, or another communication regime, rather than by absolute wall-clock time. If the same phase reappears across iterations, the controller would ideally reuse the history learned for that phase. A conventional history-based controller does not do this: it maintains a single time-ordered state for a connection. When the workload crosses a phase boundary, recent samples may come from the previous regime, while the next control decision applies to a different regime. This violates the local-continuity premise of most congestion controllers: the recent past is no longer a noisy version of the near future, but evidence from a different phase. §3 formalizes this as cross-phase blending and quantifies the resulting transition error.

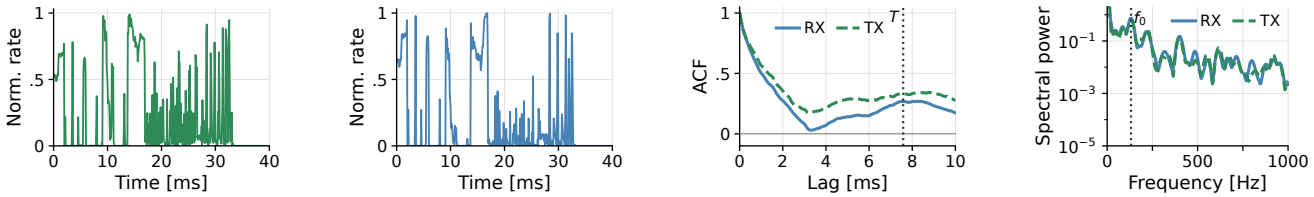
3 Failure Mode: Cross-Phase Blending

Congestion control is built around a simple feedback premise: recent measurements are useful for the next control decision. Controllers aggregate RTT samples, ECN marks, queue occupancy, loss, or receiver grants over short windows to filter noise and guide the next rate or window update. This works when the recent past and near future belong to the same underlying traffic regime. AI training traffic breaks this premise. A training job moves through a sequence of execution phases, each with different communication demand. Compute, tensor-parallel collectives, data-parallel all-reduces, pipeline-parallel transfers, optimizer traffic, and interference from other jobs can all induce different phase-conditioned sending rates on the same path or endpoint. The relevant question for the controller is therefore not only "what happened recently?" but "which phase am I in now?"

We call the resulting failure mode *cross-phase blending*. A conventional controller maintains one recent-history state across all phases: a sending rate and congestion summary in DCQCN [19]; a congestion window or pacing state in window/rate controllers such as Swift [7]; an RTT or delay baseline/gradient in delay-based controllers such as TIMELY [10]; or utilization and delivery-rate estimates in HPCC [9] and rate-recursive controllers. The variables differ, but the abstraction is the same: feedback from all recent samples updates one state object.[2] When the workload transitions from phase p_i to phase p_{i+1}, state learned in p_i is carried into p_{i+1},

[1] The markers T and f_0 indicate the dominant heartbeat.

[2] Note that although state may be maintained separately for different connections or paths, each history combines observations across phases.



(a) Tx traffic has recurring bursts, but their timing and duration vary across cycles.

(b) Rx traffic has recurring bursts, but their timing and duration vary across cycles.

(c) ACF: recurring structure with decaying, jittered lobes.

(d) PSD: a workload heartbeat near f_0 is broadened by phase jitter.

Figure 2: AI traffic is phase-structured, not strictly periodic: bursts recur, but their timing and duration vary.

even if the two phases require different *ideal control states*. Cross-phase blending occurs when the controller initially acts on state from the previous phase rather than the state appropriate for the active phase.

Cross-phase blending creates two operationally distinct transition errors. When the next phase has a higher target than the previous one, scalar state is biased downward and the controller ramps up too slowly; we call this *slow ramp-up*. When the next phase has a lower target, scalar state is biased upward and the controller releases too slowly; we call this *slow release*. Slow release is not harmless in shared AI fabrics. A lower-rate phase for one collective is often the scheduling window for another collective, either from 3D parallelism within the same job or from a different job. When traffic remains queued, stale high-rate state can allow excess bursts into a period intended for other communication, disrupting the interleaving that scheduling systems such as Cassini [12] and MLTCP [13] attempt to create.

These errors create a recurring *boundary debt*. A controller cannot learn that the correct state has changed until feedback from packets sent in the new phase returns, which takes at least one round-trip time between the sender and the congestion point: a packet must reach the congestion point, and the resulting feedback must return to the sender's controller.[3] This horizon has a floor: feedback must traverse the fabric, whose propagation delay is bounded by the speed of light in the fiber. As AI fabrics move from 100 Gbps to 400 Gbps, 800 Gbps, and beyond, a fixed interval of stale control misallocates more bytes for the same relative rate error. And because AI training repeatedly revisits the same phase boundaries across thousands of iterations, this debt is not a one-time transient; it recurs at every boundary of every iteration where scalar state blends incompatible phases.

Figure 3 illustrates this effect on the Tx trace. The scalar curve is a diagnostic EWMA over Tx rate: all samples update one temporal history, regardless of phase. It lags at both upward and downward phase transitions, creating repeated boundary debt. The phase-indexed diagnostic uses the same EWMA but keeps separate histories for recurring phases, allowing it to track the active phase-conditioned target more closely. §4 turns this diagnostic into a transport architecture.

The key point is that faster feedback alone does not remove cross-phase blending. Even if feedback arrives sooner or more frequently, an estimator that smooths all samples into the same scalar history still blends observations across phase boundaries. The result is a fictional intermediate state that corresponds to no actual execution phase. Decreasing the smoothing window reduces the duration of the error but increases noise sensitivity; increasing the window stabilizes the estimate but worsens boundary lag. This is a fundamental tension for scalar temporal estimators.

This suggests a different state representation. Instead of maintaining a single controller state $\hat{s}_t$, e.g., a rate, window, RTT baseline, or ECN summary, the controller can maintain phase-indexed state $\hat{s}_\phi$, where ϕ identifies the current execution phase. Upon entering a recurring phase, the controller can reuse the state learned from previous visits to that phase rather than adapting from the immediately preceding phase. In the ideal case, the controller does not need to rediscover that phase p_{i+1} has a different target from p_i; it indexes the state associated with p_{i+1} directly.

We quantify cross-phase blending using *transition error*. Let τ index a detected phase boundary, W be the number of samples in the first feedback horizon after the boundary, $\hat{s}_t$ be the controller's estimate or control value, and $\mu_{\phi(t)}$ be the active phase-conditioned target:

$$E_{\text{transition}}(\tau) = \frac{1}{W} \sum_{t=\tau}^{\tau+W-1} \left| \hat{s}_t - \mu_{\phi(t)} \right|.$$

This metric reports average absolute boundary error. We separately report underestimation and overestimation as slow ramp-up and slow-release error, respectively.

[3] "Sub-RTT" feedback [2] refers to a delay shorter than the typical end-to-end sender–receiver RTT. Direct switch notifications avoid the detour through the *receiver*, but the complete reactive loop still includes sender-to-congestion-point and congestion-point-to-sender propagation, i.e., an RTT between the sender and the congestion point.

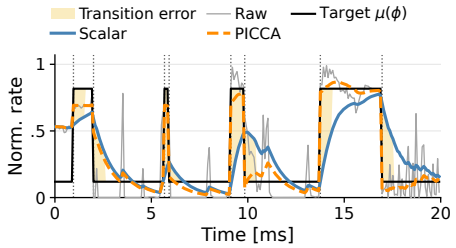


Figure 3: Cross-phase blending creates transition lag: scalar history adapts too slowly at phase boundaries, while phase-indexed history preserves per-phase state.

4 PICCA: A Phase-Indexed Transport Architecture

The failure mode in §3 suggests a different abstraction for AI congestion control. PICCA is not a new congestion control algorithm (CCA). It is a state-management architecture that lets an existing CCA run over phase-indexed histories. The base CCA still decides how to react to ECN marks, RTT samples, delivery-rate estimates, receiver grants, or INT feedback; PICCA changes which history those signals update and which state the CCA consults at a phase boundary.

Concretely, conventional transports maintain one recent-history state per connection. PICCA replaces that state with a small table of phase slots, $\mathcal{T}[\phi]$ = base-CCA state for phase ϕ, where each slot contains the variables the underlying CCA already uses: rate or window, RTT baseline, ECN history, pacing gain, delivery-rate estimate, timeout state, or receiver-grant budget. At each control interval, PICCA selects an active phase identifier $\phi(t)$, credits feedback to the phase whose packet elicited it, and asks the base CCA to act from $\mathcal{T}[\phi(t)]$. On a phase transition, PICCA checkpoints the old slot and activates the slot associated with the new phase, rather than adapting from the immediately preceding phase. This design separates three concerns.

1. Identifying phases. PICCA needs a phase identifier, but the identifier need not be a perfect semantic label. It only needs to separate recurring regimes whose control histories should not be blended. Phase information can come from a spectrum of sources. Local hints from the training runtime or collective library can label traffic by collective type, pipeline stage, communicator, message-size class, or a coarser communication-heavy phase. These hints describe what the application is doing, but they do not fully define the network regime: the same all-to-all may need different control state depending on which other collectives or jobs are active. At the other extreme, a global phase oracle can assign identifiers using a system-wide view of active collectives, topology, and sharing relationships. Such an oracle is closer to a scheduler or collective coordinator than to a collective communication library alone. Observation-driven detection can refine either source using send rate, delivered rate, RTT gradient, ECN marks, PFC events, burst shape, destination set, or inter-arrival structure.

Table 1: Example seedings of CCA state from a phase target rate R.

CCA	Seeded fields
DCQCN [19]	`m_rate` = R, `targetRate` = R; $\alpha \approx 1 - R/C$
HPCC [9]	`curRate` = R; local/per-hop utilization $u \approx R/C$
DCTCP/Reno	`cwnd` = $R \cdot$ RTT
Swift [7]	`rate` = R; `cwnd` = $R \cdot$ RTT/MTU

2. Indexing CCA state. PICCA turns the state variables of the base CCA into phase-indexed variables: $(\hat{s}, x, r, q, \ldots) \rightarrow (\hat{s}_\phi, x_\phi, r_\phi, q_\phi, \ldots)$. Equivalently, a minimal phase-indexed estimator updates only the slot of the phase that sent the sampled packet (at time t_s), $\hat{s}_{\phi,t} = (1-\alpha)\hat{s}_{\phi,t-1} + \alpha s_t, \quad \phi = \phi(t_s)$, and then invokes the base control law using the active slot. The key architectural point is that phase indexing changes the unit of memory, not necessarily the control law. A PICCA wrapper can therefore sit alongside DCQCN-, HPCC-, Swift-, or receiver-grant-style controllers, preserving their feedback semantics while preventing unrelated phases from overwriting each other's histories. When phase information includes a target rate R (e.g., from a global oracle), PICCA translates R into the native state of the base CCA rather than replacing the CCA. Table 1 shows examples. After this seeding step, the CCA resumes normal feedback control around the phase-aware anchor.

3. Handling jitter. PICCA does not assume a rigid periodic schedule. A phase may begin earlier or later than expected, last for a different duration, or have a different amplitude because of model layer, placement, or interference. PICCA treats phase state as a prior, not a hard prediction: when a recurring phase begins, the controller starts from the associated state, then corrects it with fresh feedback.

PICCA's benefit therefore depends on whether history from previous occurrences provides a better starting point than the immediately preceding phase, rather than on an exact mapping from phases to network conditions. Changes in competing traffic or routing can invalidate that history. A practical implementation should therefore age or invalidate stale entries and use the base CCA's default initialization when no reliable history is available. Practical designs can also share information across related phases, impose per-destination or per-priority budgets, and cap aggregate rate across phase slots to preserve fairness and stability. PICCA is complementary to AI schedulers and progress-aware transports such as Cassini [12] and MLTCP [13]: they decide *when* communication should run, while PICCA remembers *how* recurring phases should be controlled. §5 estimates the headroom with a global oracle that seeds base-CCA state at boundaries; §6 asks how practical sources capture it.

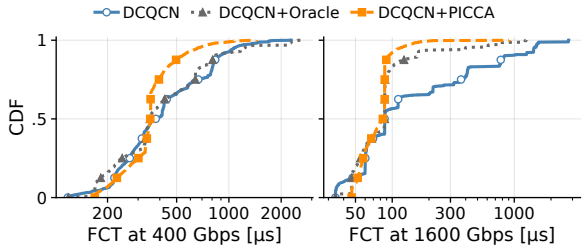
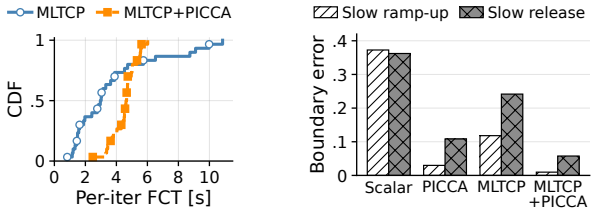


Figure 4: With oracle phase information, PICCA reduces tail message FCTs, while the same oracle without phase-indexed state leaves a long tail.



(a) MLTCP+PICCA. **(b) PICCA cuts boundary error.**

Figure 5: PICCA reduces MLTCP iteration tails (a) and reduces boundary error on the trace (b).

5 What If We Knew the Phase?

We instantiate PICCA in two preliminary prototypes. The first is an RDMA/ns-3 prototype that pairs a global PhaseOracle with DCQCN to test whether phase-aware warm-starts improve end-to-end collective performance. The second is an MLTCP-style prototype and trace-driven analysis that isolates the state-representation effect and tests whether PICCA composes with progress-aware transport.

Prototype. Our RDMA prototype implements the oracle extreme of §4. As a simulation free of practical noise such as host jitter, stragglers, and unseen tenants, it approximates an upper bound on PICCA's gain with a cluster planner. At simulator initialization, the PhaseOracle scans each rank's Chakra execution trace, walks the dependency DAG to identify collective flow sets and expected start times, and runs a max-min sharing model over the topology to produce a time-indexed phase calendar. At QP creation and at each phase boundary, the simulated RDMA hardware queries the oracle for the active phase and target rate R for that flow. A CCA-specific translator then seeds the base controller state, e.g., DCQCN rate/alpha, a window or cwnd, or HPCC utilization state, before normal ECN, RTT, INT, or delivery-rate feedback resumes. Thus, the prototype changes the starting state at recurring phase boundaries; it does not disable congestion feedback.

The closest deployable analogue is a cluster planner. Compiler and runtime components can expose a communication calendar, pipeline schedulers know microbatch order, and ProcessGroupNCCL already exposes per-collective start/end events, so a planner could assign phase identifiers and target shares for the NIC to apply. In production, however, it is an imperfect oracle: it sees the plan, not the execution, and host jitter, stragglers, and tenants outside its view move the boundaries and targets it predicts (§6).

End-to-End Simulation. We evaluate PICCA on top of DCQCN in a 16-NPU, 4-leaf, 2-spine fabric with 4 downlink NPUs per leaf, 2 uplinks to the spines and 1 µs per-hop latency. The workload runs 4 collectives concurrently on all 16 NPUs: three Ring-AllReduces (20 MB per iteration, 2 ms of compute between iterations) and one AllToAll-Direct (20 MB per iteration, 1 ms of compute between iterations). The AllToAll-Direct is launched roughly 1 ms ahead of the AllReduces, so the two collectives temporally overlap on the bottleneck uplinks. We sweep the link speed across 100, 200, 400, 800, and 1600 Gbps (with per-tier ECN K-min/K-max scaled linearly with link rate). We compare the flow and collective completion times of DCQCN against DCQCN+PICCA, and against DCQCN+Oracle, described below.

The gap between DCQCN alone and DCQCN+PICCA grows with link speed, and the growth is largest in the tail. At 100 Gbps PICCA is roughly neutral on p99 FCT, but the p99 reduction grows to 22% at 200 Gbps, 41% at 400 Gbps, 55% at 800 Gbps, and 94% at 1600 Gbps. Figure 4 shows the FCT CDFs at 400 and 1600 Gbps. Since collectives are tail-sensitive, where a single straggler flow gates iteration progress, the FCT tail reductions translate into large CCT improvements: PICCA's mean CCT improvement grows from 25% at 400 Gbps to 60% at 1600 Gbps, and p99 CCT falls by 70% at 1600 Gbps. As link rates increase, a fixed feedback horizon covers more of each short phase, so scalar DCQCN acts longer on state learned in the previous phase; PICCA instead starts each phase from that phase's state, and live DCQCN feedback continues from there.

To separate the oracle from phase indexing, we add DCQCN+Oracle, which receives the same oracle target rates at the same phase boundaries but keeps one scalar DCQCN state per QP: after seeding, feedback from every phase updates that state, and a recurring phase does not get its own state back. The oracle alone is not enough, especially in the tail and at high link rates: PICCA's p99 FCT is 7.1× lower than DCQCN+Oracle's at 1600 Gbps and 2.3× lower at 400 Gbps (Figure 4). A correct starting rate does not persist once scalar state blends the phases that follow; keeping state per phase is what preserves the gain.

We also test MLTCP+PICCA on a 1 Gbps dumbbell shared by six concurrent flows. Each flow runs five 100 MB training iterations with a 200 ms compute gap and 50 ms cross-flow start jitter. At each iteration boundary, PICCA reseeds `cwnd`/`ssthresh` to the max-min fair-share BDP, (bottleneck bandwidth/active flows)×base RTT. Compared

with MLTCP-Reno, MLTCP+PICCA drops p95 per-iteration FCT from 9.98 s to 5.62 s and p99 from 10.85 s to 6.02 s (Figure 5a). The cost is a higher mean CCT (3.70 s to 4.53 s) because the fair-share seed prevents the fastest starter from over-consuming transient bandwidth.

What does this mean for training time? The impact of boundary error depends on how much communication lies on the iteration critical path. Using SwitchML's 100 Gbps profiles, the exposed (non-overlapped) communication fraction is roughly 63% for DeepLight, 20% for LSTM, 11% for BERT, and 17% for NCF [14]. Thus PICCA's 41–94% p99 FCT reduction corresponds to at most 26–59%, 8–19%, 5–10%, and 7–16% iteration-time reduction, respectively. For the 45% MLTCP+PICCA tail reduction, the corresponding reductions are about 28%, 9%, 5%, and 8%. This is a sensitivity estimate rather than a full training run, but it shows why cross-phase blending is a training-time problem: collective tails gate iteration progress.

Trace-Driven Boundary Analysis. We next isolate the state-abstraction effect on the measured Tx trace from §3. We compare **Scalar**, **PICCA**, **MLTCP**, and **MLTCP+PICCA**. MLTCP+PICCA applies the same progress scaling to phase-indexed state. Here, no oracle seeds slots: they are learned from past visits, with offline phase labels. Figure 5b summarizes the transition-error metrics from §3. PICCA reduces both slow ramp-up and slow-release error. MLTCP reduces some error through progress scaling, but MLTCP+PICCA further reduces the remaining error by changing the state representation. This supports the main architectural claim: progress-aware scheduling and phase-indexed transport state are complementary.

6 But Can We Know Phases?

For each phase, PICCA needs two things: *when* the phase starts, and the *parameters* its slot should hold, such as the starting rate. In §5, an oracle supplied both. In practice, they can come from three sources that know different things, face different challenges, and can be combined. Timing is not equally hard for all phases: a node knows when its own collectives start, since it posts them, but learns that another job started sending on a shared link only a feedback horizon later, and at 800 Gbps a 1 MB phase ends before its first feedback returns (cf. Figure 1).

Planner. The practical counterpart of our oracle is a cluster planner, such as a scheduler, compiler, or collective coordinator, that knows which collectives run, in what order, and with what sharing, so it can compute each phase's parameters. Its weakness is timing: it sees the plan, not the execution, so jitter, stragglers, and tenants it does not manage shift boundaries, and long runs drift from the plan (clock skew is minor: clocks synchronize to sub-microsecond accuracy [4, 8]). Its parameters can also be wrong when the plan misses traffic. A planner can also shape phases: schedulers such as Cassini [12] and MLTCP [13] already decide how jobs interleave and could keep phases regular by design.

Application. The collective library knows exactly when each of the node's collectives starts and which peers it involves, and could tell the NIC through a plugin interface. Its weakness is parameters: it knows nothing about other jobs on shared links, so it cannot say what rate a phase should get. The agenda is a narrow, standard interface for phase events between libraries and NICs.

Learn and remember. A node can also learn on its own: store the state each phase settles at and reuse it when the phase recurs. Its weaknesses are recognition and delay. Recognizing a recurrence means judging the conditions a phase meets, not just which collective runs: one iteration of an all-to-all may collide with another job while the next does not, and the two need different slots. A learner can split a slot whose reuse keeps failing and merge slots that agree. It must also predict other jobs' phases, since feedback reveals them too late, much as a CPU predicts its next branch from its branch history [15]. Delay matters because most feedback about a short phase returns after it ends; since a slot is read only when its phase recurs, its update can wait until the boundary is known and credit each signal to the phase that sent the packet: act now, learn later. Learning also depends on phases recurring: when jobs arrive, leave, or are rescheduled, learned slots go stale and must be detected and relearned.

Combining sources. These strengths complement each other: a planner knows parameters but not exact timing, the application knows exact timing for the node's own phases, and learning corrects both from experience. *Late binding* combines them: take each phase's parameters from the plan, switch slots when the application signals that the phase has started, and let learning correct what the plan got wrong. For other jobs' phases, the planner or a switch can relay the end of their previous collective. Which combination suits which workload, and at what cost, is the central open question.

7 Conclusion

AI training traffic recurs in phases whose boundaries drift. A transport that keeps one history per connection blends these phases at every boundary, at a cost that grows with link speed. PICCA keeps state per phase instead; with oracle phase information, it cuts tail completion times in simulation, and §6 lays out how practical sources might capture this headroom.

Datacenter transports have spent a decade learning to react faster. AI fabrics need them to remember.

Acknowledgments. We thank Costin Raiciu and the anonymous reviewers for their valuable feedback. This work was supported by NSF grants 2313164 and 2543556.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM*.

[2] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. 2023. Bolt: Sub-RTT Congestion Control for Ultra-Low Latency. In *NSDI*.

[3] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-based congestion control. *Commun. ACM* (2017).

[4] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *NSDI*.

[5] Soudeh Ghorbani, Yimeng Zhao, Srikanth Sundaresan, Ying Zhang, Yijing Zeng, Abhigyan Sharma, Prashanth Kannan, and Cristian Lumezanu. 2025. Congestion Patterns in a Large-scale RDMA Datacenter. In *IMC*.

[6] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*.

[7] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M G Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *SIGCOMM*.

[8] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. 2020. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *OSDI*.

[9] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High precision congestion control. In *SIGCOMM*.

[10] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*.

[11] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*.

[12] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. In *NSDI*.

[13] Sudarsanan Rajasekaran, Sanjoli Narang, Anton A Zabreyko, and Manya Ghobadi. 2024. MLTCP: A distributed technique to approximate centralized flow scheduling for machine learning. In *HotNets*.

[14] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with in-network aggregation. In *NSDI*.

[15] André Seznec and Pierre Michaud. 2006. A Case for (Partially) TAgged GEometric History Length Branch Prediction. *Journal of Instruction-Level Parallelism* 8 (2006).

[16] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan M. G. Wassel, Naveen Kr. Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, Sarin Thomas, Inho Cho, Hyojeong Lee Seibert, Baijun Wu, Rui Yang, Yuliang Li, Kai Huang, Qianwen Yin, Abhishek Agarwal, Srinivas Vaduvatha, Weihuang Wang, Masoud Moshref, Tao Ji, David Wetherall, and Amin Vahdat. 2025. Falcon: A Reliable, Low Latency Hardware Transport. In *SIGCOMM*.

[17] Rip Sohan, Eric Spada, Eric Davis, Mark Handley, Idan Burstein, Tony Hurson, Jithin Jose, Vivek Kashyap, Rong Pan, and Sayantan Sur. 2026. *Multipath Reliable Connection (MRC) Specification*. Specification Revision 1.0. Open Compute Project. https://www.opencompute.org/documents/ocp-mrc-1-0-pdf

[18] Arvind Srinivasan, Neil Spring, Omar Baldonado, and Rajiv Krishnamurthy. 2026. MetaRoCE: A New RDMA Transport Built for AI-Scale Ethernet. Engineering at Meta. https://engineering.fb.com/2026/08/24/networking-traffic/metaroce-rdma-transport-ai-ethernet/

[19] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *SIGCOMM*.