PCM: Programmable Congestion Management for Terabit RDMA NICs

Mikhail Khalilov[1], Timo Schneider[1], Tommaso Bonato[1],
Siyuan Shen[1], Marcin Chrapek[1], Yehor Kozyr[1], Alberto Merino[2],
Rip Sohan[3], Surendra Anubolu[3], Mohan Kalkunte[3],
Ahmad Ghalayini[4], Abdul Kabbani[4], and Torsten Hoefler[1,4]

[1]ETH Zürich
[2]Universidad de Castilla-La Mancha
[3]Broadcom Inc.
[4]Microsoft Corporation

Abstract

Datacenter congestion control (CC) and load balancing (LB) evolve rapidly as workload demands change, but deployment depends on RDMA hardware and transport interfaces. Emerging programmable RDMA engines tie policy code to vendor APIs. Operators must therefore port each policy for every platform they deploy. We address this with PCM, a programming model that separates portable congestion management from platform integration on datacenter NICs. Developers write event-driven handlers that they can debug in simulation and reuse on real hardware. PCM exposes flow telemetry as signals and uses triggers to support per-event reactions and coarser responses to accumulated feedback. Vendors provide optimized backends that map signals and output controls to native NIC mechanisms with defined feedback and control semantics. We demonstrate PCM's expressivity by implementing eleven well-known datacenter CC and LB algorithms. We deploy PCM across four distinct datapaths: an Ultra Ethernet simulator, CPU-based UCCL, and on-NIC offloads on NVIDIA ConnectX-8 and Microsoft 1P. On CX-8, PCM-Swift achieves processing capacity equivalent to one reaction per three 4 KiB packets at 800 Gbit/s, using one-fifth of the NIC's hardware threads. In UCCL, it matches native Swift's 1.56 Tbit/s goodput with 64 KiB chunk-level control. PCM's design informed ongoing Ultra Ethernet work toward a portable congestion management interface.

1 Introduction

Datacenter operators seek workload-specific congestion control and load balancing [1, 7, 19, 25, 26, 31, 42, 46] to meet rapidly evolving workload requirements [14, 15, 17, 29, 30, 32, 34, 40, 44]. Hardwiring these policies into the RDMA transport ASIC forces a multi-year cycle to tape out, test, and deploy new silicon [43]. This cycle cannot keep pace with changing AI models, topologies, or congestion signals.

Emerging programmable RDMA engines shorten the cycle by accepting new algorithms after deployment, but their execution models and APIs tie policy code to a particular architecture, transport, or vendor [4, 23, 35, 39]. Software control planes offer a third option. CCP executes policy in a portable user-space control plane [33], but it doesn't support offloading, and its microsecond-scale control cannot serve nanosecond-scale per-packet path decisions.

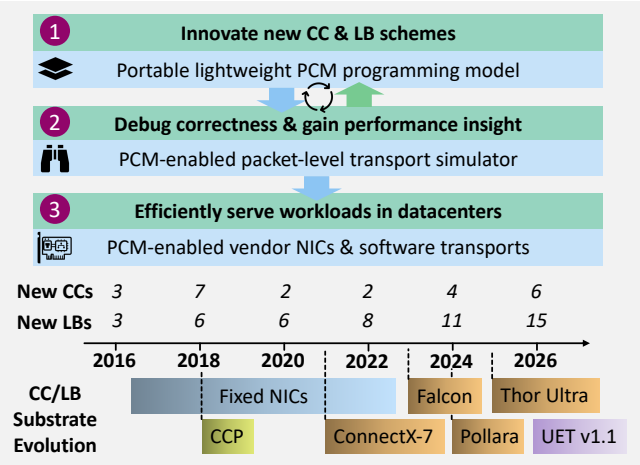


Figure 1: PCM generalizes decades of NIC congestion management evolution into a reusable workflow: tune policies in simulation and deploy them unchanged through vendor-optimized platform backends.

No common interface combines policy source portability with efficient execution across vendors and transports with different semantics. Operators need to develop a policy once and reuse it wherever the required feedback and controls exist. Vendors should implement each platform mapping once and optimize it independently of policy code. PCM provides this separation between policy development and platform integration (Figure 1).

Programmable Congestion Management (PCM) is an open-source programming model for datacenter NIC policies. It separates policy development from platform integration through portable control objects and performance-optimized accessors. Developers express algorithms as event-driven *handlers* over signals, triggers, controls, and private state. Platform backends generate *shells* that map these objects to

native state and CC/LB hooks, preserving handler source while allowing platform-specific optimization. This enables *source compatibility*: developers reuse unchanged handler code across supported platforms by generating a shell for each target. Deployment parameters and behavioral validation remain target-specific.

Handler performance is as important as portability. At 1.5 GHz and 0.5 instructions per cycle, a core can execute only about 15 instructions per 4 KiB packet at 1.6 Tbit/s. Setting up a single function call can exhaust this budget. PCM closes this gap at three levels. First, shell generators compile NIC state accessors into native loads and stores to minimize abstraction overhead. Second, handlers for independent flows map transparently to parallel, energy-efficient NIC cores, increasing aggregate capacity. Third, hardware-aware signal accumulation preserves counts and feedback summaries while decoupling the execution of instruction-heavy or resource-limited handlers from event arrival rates. Together, these mechanisms let portable handlers approach bare-metal performance, react at nearly per-packet granularity when capacity permits, and consume accumulated feedback when policies tolerate coarser reactions.

To demonstrate PCM's expressivity, we implement eleven state-of-the-art datacenter CC and LB algorithms, most in fewer than 100 lines of C. We implement shell generators for four platforms. UET-htsim enables correctness checks and parameter tuning before deployment [12]. Integrating PCM into UCCL tests portability to a CPU-based software RDMA transport [45]. We support PCM offload on NVIDIA CX-8 and Microsoft 1P, two NIC architectures with different hardware RDMA transports.

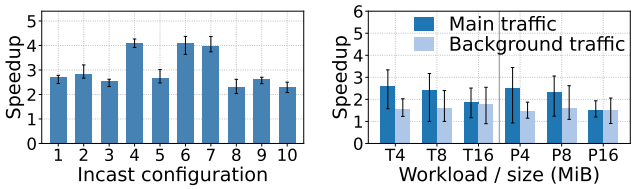
Our evaluation answers four questions: What does the abstraction cost? What does policy reuse require? How does execution scale? What reaction rates can each platform sustain? Matched native baselines and targeted experiments reveal how arithmetic, feedback granularity, worker placement, and flow-state footprint shape the performance and behavior of portable PCM policies.

We make the following contributions:

1. **Design requirements:** We derive expressivity, portability, and performance requirements from workload needs and existing mechanisms.
2. **A unified, efficient programming model:** We define hardware-friendly abstractions for portable NIC policies with per-event and coarser reactions.
3. **Multi-platform validation:** We demonstrate eleven policies, four platform integrations, and full CC offload.
4. **Ultra Ethernet standardization impact:** Our work informed ongoing standardization of programmable congestion management.

2 Motivation

Three use cases motivate reusable policies across NICs.



(a) Incast-aware CWND tuning. (b) Mixed LB policies.

Figure 2: Maximum flow completion time (FCT) speedup. T/P: tornado/permutation. Bars show means over ten endpoint mappings; whiskers show min–max.

Incast-aware CC tuning: PCM allows operators to explicitly tune policies for workloads and NICs. Even moderate incast can collapse throughput and raise tail latency in collectives, storage, analytics, and caching [14, 15, 34, 40, 46]. BDP-sized initial windows suit long, well-balanced flows such as all-to-all collectives, but cause packet loss at shallow-buffered ports under incast before CC converges. Figure 2a compares BDP-scale and incast-tuned smaller initial windows for the MP-RDMA policy in the packet-level Ultra Ethernet transport simulator (UET-htsim). Ten configurations (19–60 senders, 150–500 KB messages) use a 1024-node, two-tier 400 Gbit/s fat tree and ten shared random endpoint mappings each. We observe a 3× speedup of maximum FCT. The tuned policy avoids 98.6% of baseline packet drops.

Mixed LB policies: PCM benefits scenarios where applications impose conflicting requirements on packet delivery. Adaptive packet spraying lets AI collectives use all fabric paths' bandwidth, while co-deployed legacy MPI and some RDMA applications need in-order delivery [17, 21, 37]. PCM lets operators combine ECMP [22] for legacy applications with adaptive schemes such as REPS [7] for AI traffic. Figure 2b compares per-flow coexistence of REPS and native ECMP against cluster-wide ECMP [7, 22] in a simulated 128-node, two-tier 400 Gbit/s fat tree. Tornado and permutation workloads induce core congestion with 4, 8, and 16 MiB messages and ten random mappings per configuration [7, 36]. REPS controls 115 of 128 flows. ECMP serves the 13 background flows, avoiding multipath-induced reordering. Compared to cluster-wide ECMP, REPS+ECMP improves maximum FCT by 1.51–2.61× for main traffic and 1.48–1.80× for background.

Domain-specific CCs: PCM enables endpoints to deploy custom CC policies per flow. This is particularly useful in heterogeneous multi-datacenter training [41]. Operators typically deploy different policies depending on the traffic's destination: within a datacenter, low-latency schemes like Swift [26]; across datacenters, delay-tolerant schemes like BBR [5, 9].

Mechanism	Policy scope	Execution	Event handling	Integration requirements	Open source
Fixed NICs [13]	CC/LB	Sync	Per-event	ASIC	No
Tonic [4]	CC	Sync/Async	Per-event/periodic	FPGA, Verilog	Yes
AMD Pensando [2]	CC/LB	Not reported	Not reported	Pensando pipeline, P4-like language	No
Broadcom Thor [8]	CC/LB	Not reported	Not reported	Thor pipeline, C-like NPL language	Spec only
SwCC [23]	CC	Sync	Per-event	RoCEv2, SwCC RDMA engine, C	Yes
DOCA PCC [35]	CC	Async	Event-specific HW coalescing	RoCEv2, NVIDIA DPA, C	Samples
Falcon [39]	CC/LB	Async	Event-specific HW coalescing	Falcon PDL/RUE	Spec only
Linux eBPF CC [20]	CC	Sync	Per-event	Linux networking stack	Yes
UCCL [45]	CC/LB	Sync	Per-event	Verbs, C++	Yes
Presto [38]	CC	Async	Periodic	P4, C++	Yes
CCP [33]	CC	Async	Arbitrary SW runtime filters	CC hooks, CCP DSL, Rust	Yes
PCM	CC/LB	Sync/Async	min/max/sum/last event accumulation	CC/LB hooks, C	Yes

Table 1: Sync/Async: policies run within/independently of the datapath. Integration requirements identify the datapath interfaces and execution support required by each mechanism.

3 Related Work

Supporting workload-specific congestion management requires both programmable policies and reusable platform integration. Table 1 compares existing mechanisms for NIC congestion management. They bind policy code to hardware, interfaces, or host runtimes. PCM addresses this gap through CC/LB source portability across substrates, defined signal and control semantics, and native execution on embedded NIC cores. This combination extends policy reuse from simulation to NIC deployment.

Hardware and transport dependencies: HotCocoa provides programmable hardware abstractions for CC logic in Verilog [3]. Tonic extends this approach to a 100 Gbit/s FPGA design, with Verilog templates for incoming-packet logic and periodic CC updates [4]. It processes relevant events every cycle, but policy changes require synthesis, timing closure, and target-specific verification. AMD Pensando and Broadcom Thor Ultra similarly tie P4-like pipeline code to vendor compilers, stages, metadata, and actions.

NVIDIA DOCA PCC compiles C callbacks for the ConnectX DPA, with ABI-specific flow contexts, events, algorithm attributes, and results [35, 43]. It can coalesce events into aggregate counters. The DOCA PCC interface exposes RoCEv2 and RDMA engine control events and RTT probes, and provides nanosecond RTTs and fixed-point arithmetic helpers.

Google's Falcon transport specifies an asynchronous engine that consumes ACK-, NACK-, and retransmission-triggered events and returns CC and path-selection controls [39]. The Falcon datapath filters control events according to queue occupancy, elapsed time, and acknowledged volume, while accumulating ACK counts between engine updates. These NICs expose native transport event formats, state layouts, and controls to policy code.

Programming language and runtime dependencies: Linux TCP loads telemetry-driven CC handlers as eBPF bytecode without rebuilding the kernel. Their C code depends on kernel callbacks (tcp_congestion_ops), helpers, and socket state, preventing offload to RDMA NICs [20]. Presto's Tofino 2 RMT datapath implements TCP, aggregates acknowledged/ECN-marked bytes and duplicate ACKs, and enforces rates [38]. Its off-path C++ control plane runs DCTCP or Timely and updates rate tables, with configured minimum intervals of 50 μs between polls and 80 μs between datapath-generated credit messages. These policies retain dependencies on the P4 datapath and C++ control plane. UCCL implements GPU transport pacing, loss recovery, CC, and path selection over multi-packet chunks striped across RDMA queue pairs using Verbs UC Write [45]. Portability across Verbs providers still leaves this logic tied to UCCL's classes and path APIs. All three systems require interface changes for deployment on other substrates. For example, Presto's and UCCL's C++ policies cannot compile with the DOCA PCC C11 compiler [35].

Host control plane delay: CCP preserves CC source across different host datapaths and protocols, including Linux TCP, QUIC, and mTCP [33]. It extends event coalescing with programmable filters and reporting rules. Policies can request per-event reports when the control-plane agent keeps pace or aggregate measurements before reporting. CCP interprets these rules as DSL programs on each ACK or timeout to accumulate measurements, select reports, and update controls. Portus, CCP's Rust runtime, runs the CC algorithm in a separate user-space process. It delivers reports to that algorithm and returns its control updates through POSIX IPC. On our Intel Xeon Gold 6140, CCP's NewReno cycle takes 10 μs. A single Linux kernel IPC call takes about 3 μs on this system, the runtime-interpreted DSL takes 350 ns, and NewReno calculation takes 300 ns. At 1.6 Tbit/s, a 4 KiB packet arrives every 20 ns. CCP's filters and control plane round trip exceed this deadline by orders of magnitude. While some sender-side CC algorithms tolerate coarse control, others, such as MP-RDMA, target nearly per-packet reactions [31]. Receiver-side CC likewise requires fine-grained credit allocation and

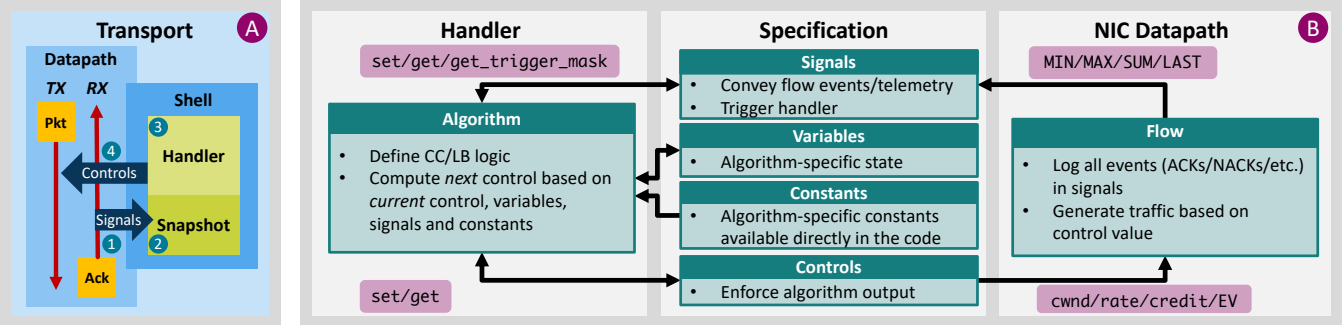


Figure 3: PCM separates a portable handler from the transport datapath. (A) A shell accumulates events, evaluates triggers, constructs a snapshot, executes the handler, and commits its outputs. (B) The specification names control objects, while each shell maps them to native mechanisms.

pacing [36], while per-packet LB requires a path decision for each transmission [7]. Host round trips for fresh per-packet decisions stall transmissions or leave the datapath using stale state during transient bursts and path congestion.

4 Design Requirements

The use cases and NIC mechanisms discussed in Sections 2 and 3 motivate three requirements for PCM.

Expressivity: A common programming model needs to express sender-side CC, receiver-side credit control, and stateful LB to accommodate diverse workload requirements.

Portability: Maintaining separate simulator and NIC implementations requires repeated integration work and creates opportunities for bugs. Policy reuse therefore requires separating handler logic from native APIs, state layouts, measurement units, and arithmetic implementations. We target *source compatibility*: unchanged policy code across supported backends, with deployment-specific parameters and numerical precision. Feedback granularity, arithmetic, scheduling, and native and approximated controls can still affect behavior, so each deployment requires *behavioral validation*. A backend must reject a handler if the platform cannot provide a required signal or enforce a required control.

Performance: Terabit rates demand low execution overhead and sufficient aggregate processing capacity. Portable handlers should approach native execution performance and scale across independent flows. Platforms also expose feedback at different granularities: DOCA PCC and Falcon can aggregate feedback before invoking policy code, while UCCL delivers ACKs for multi-packet chunks. PCM should exploit native execution and feedback mechanisms to minimize overhead and increase processing capacity. It should support per-event execution when the platform can sustain it and accumulated feedback when deployment-specific validation shows that the policy tolerates coarser reactions.

Section 7 examines these requirements through algorithm coverage, policy reuse across transports, and measurements of execution cost, goodput, and scaling.

5 PCM Programming Model

We introduce a programming model to express congestion management algorithms as portable, event-driven *handlers*. A PCM handler is a stateful function that operates on per-flow state independently of hardware details. Each handler context belongs to one flow. Developers may compose modular logic by associating a flow with multiple handlers whose controls do not overlap. For example, separate handlers can implement load balancing and congestion control. Matching flow events to a handler context is outside PCM's scope. It depends on the NIC transport, traffic-steering engine, and transport addressing.

5.1 Unified Datapath Abstractions

Figure 3 shows how PCM separates congestion management policy from platform-specific event management and control enforcement. A handler uses `get` and `set` accessors to read signals, update controls, and maintain private state. These accessors provide a safe API that hides each object's internal representation.

Controls: A *control* specifies a transport knob, such as a window, rate, path entropy, or cumulative credit. After execution, the handler returns its controls to the transport.

Signals: A *signal* exposes datapath telemetry or live state. Examples include counts of ACKs, NACKs, RTOs, and ECN marks, as well as RTT, elapsed time, queue backlog, and path identity. Each signal selects an accumulator: `SUM` for counts, `MIN`/`MAX` for extrema, or `LAST` for the newest value. Accumulator storage is independent of event count, and several named accumulators can share one telemetry source.

Triggers: PCM invokes handlers through a *triggering mechanism* driven by accumulated flow events.

A trigger uses a particular signal value to determine when a handler should run. The backend combines all fired triggers into one mask. The handler reads the mask to select its branches. Each handler requires at least one triggering signal.

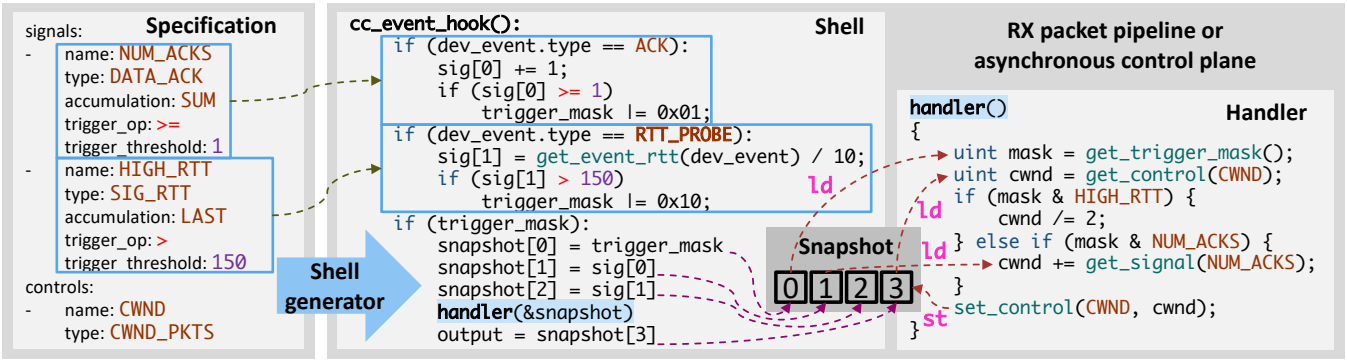


Figure 4: Shell generation from a specification and a handler. The generator emits constant object identifiers, the snapshot layout, and the required accumulation, trigger, snapshot, and commit operations. The compiler folds accessors into native loads and stores, eliminating runtime lookup and mask conversion.

Each trigger evaluates a single-instruction condition against the accumulated value and a static *threshold*. The comparison supports several modes. `RAW` compares the accumulated value with the threshold. `DELTA` compares the change since that trigger's last satisfied check. `MAGNITUDE` compares the absolute change from the same baseline. For example, an ACK counter with `DELTA >= 1` permits per-packet reactions when the shell keeps pace. If the control law permits, a higher threshold allows less frequent reactions.

Variables and constants: *Variables* hold persistent private state, including signed and unsigned scalars and fixed-size arrays. Developers can choose floating-point or fixed-point representations for fractional variables at compile time. *Constants* define state that remains immutable throughout a flow, such as base RTT and MTU. The compiler resolves them at compile time to optimize calculations and eliminate unnecessary loads.

5.2 Handler Compilation

The specification lists all objects used by the handler and their properties. The C source and specification form a handler package. Figure 4 shows how a compiler combines this package with a platform backend API to produce a *shell*.

Object generation: The shell translates datapath events to signals and maps controls to native mechanisms. For each specified object, the compiler emits logic that implements its semantics.

Generated code for `HIGH_RTT` in the example in Figure 4 branches on the native RTT-probe event and checks whether the converted RTT exceeds 150 PCM ticks (15 μs).

Object representation: Shells can normalize or approximate exposed feedback and controls. The example in Figure 4 converts the probe from 10 ns platform ticks to 100 ns PCM ticks. Some handlers evaluated in Section 7 use byte counts. On ConnectX, the backend API delivers only cumulative ACK counts. The shell estimates acknowledged bytes by multiplying that count by the configured MTU. However, shells cannot recover unavailable information or capabilities. The compiler checks interface support before deployment. For example, our ConnectX backend rejects LB handlers because its backend interface lacks path selection.

Source compatibility: Successful shell generation from the handler package establishes *source compatibility*. Developers validate handler behavior in simulation and deployment with the available signals, constants, and numerical precision. Vendors validate event accounting, units, state placement, scheduling, and native control mappings.

Compile-time optimizations: Separating the handler from its specification enables compile-time optimizations (Figure 4). We exploit these opportunities in all our shell compilers (Section 6). The compiler can use the specification to set up a static memory layout for handler objects, fold object accesses into native loads and stores, and map accumulator updates to hardware primitives, such as ConnectX's event coalescing or atomic instructions. On both AArch64 and RISC-V, each signal accumulation takes one (SUM, LAST) or two (MIN, MAX) instructions with operands in registers.

5.3 Handler Execution Modes

PCM defines two execution modes relative to flow events.

In synchronous mode, the shell completes the execution cycle in Figure 3A on the flow's event path. Signal updates and execution do not overlap, so the shell can use non-atomic context storage.

In asynchronous mode, a separate control plane evaluates triggers and runs handlers independently of datapath progress. The shell exchanges signals and controls between the data and control planes. Our asynchronous CPU shell connects the datapath API to accumulators and controls through CPU atomics. On ConnectX, the closed-source PCC runtime exchanges events and control results between the RDMA datapath and the DOCA PCC flow context. This context runs on a statically assigned hardware thread [35].

```
uint cwnd = get_control(CWND);
if (get_trigger_mask() & NACKED_BYTES) {
  // decrease logic
} else if (get_trigger_mask() & ACKED_BYTES) {
  uint cur_aked = get_signal(ACKED_BYTES);
  uint diff = cur_aked - get_var(PREV_ACKED_BYTES);
  cwnd += diff * MTU / cwnd;  // Additive increase.
  cwnd = max(MTU, cwnd);
  set_var(PREV_ACKED_BYTES, cur_aked);
}
set_control(CWND, cwnd);
```

Listing 1: ACK/NACK-based sender CC.

```
if (get_trigger_mask() & (PULL_TARGET | NEEDS_MORE)) {
  uint target = get_signal(PULL_TARGET);
  uint pull = get_control(PULL_NO);
  if (target > pull)
    pull += min(CREDITS_PER_PULL, target - pull);
  set_control(PULL_NO, pull);
  set_signal(NEEDS_MORE, pull < target ? 1 : 0);
}
```

Listing 2: EQDS grant step for unsatisfied sender demand.

Snapshots: *Snapshots* preserve `get`/`set` signal semantics across both execution modes (Figure 4). Datapath updates cannot change the captured values during execution. Handler writes remain local until commit.

Some transports may update multiple signals per event, such as an ACK carrying a cumulative acknowledged count and an RTT sample. Asynchronous shells may need to update individual signal accumulators one by one, as our CPU shell does through atomics. Locking all accumulators while copying a snapshot could stall both planes. To avoid these stalls, PCM permits relaxed snapshot delivery. Each signal preserves accumulation order, but the handler may read feedback from different update epochs. For general bounded skew, we propose letting policies specify a maximum and shells tag updates with sequence numbers to discard snapshots that exceed it. Our evaluated asynchronous CPU shell implements relaxed and zero-skew snapshots.

5.4 Real-World Handler Examples

We show key PCM interface concepts in excerpts from three real handlers in Table 3.

Sender-side CC: Listing 1 illustrates ACK feedback consumption by an MP-RDMA-style sender CC [31]. We provide the corresponding handler's specification in Appendix B, Listing 4. It declares a cumulative ACK-byte counter, a `DELTA >= 1` trigger, and a private baseline `PREV_ACKED_BYTES` variable. On new ACKs, the handler subtracts the baseline to obtain newly acknowledged bytes. It then advances the baseline and writes its window decision through a control.

The shell's `DELTA` baseline determines when to invoke the handler. The private ACK baseline records the consumed feedback. Branch order also matters: NACK represents negative feedback and takes precedence when both triggers fire.

```
uint mask = get_trigger_mask();
uint cur_ecn = get_signal(NUM_ECN);
uint is_ecn = cur_ecn - get_var(PREV_ECN);
if (mask & NUM_ACK) {
  set_var(PREV_ECN, cur_ecn);
  if (!is_ecn) {
    uint head = get_var(EVC_HEAD_IDX);
    uint count = get_var(EVC_NUM_VALID_EVS);
    set_arr(EVC, head, get_signal(ACK_EV));
    if (!get_bitmap_idx(EVC_BITMAP, head)) {
      set_var(EVC_NUM_VALID_EVS, count + 1);
      set_bitmap_idx(EVC_BITMAP, head, 1);
    }
    head = (head + 1) & (EVC_SIZE - 1);
    set_var(EVC_HEAD_IDX, head);
  }
}
```

Listing 3: REPS ACK handling: cache "good" EVs.

Receiver-side CC: Our EQDS receiver [36] illustrates how a handler paces an individual sender. It reads a cumulative credit target and publishes a cumulative pull number to grant credit. `PULL_TARGET` uses a `MAX` accumulator and a `MAGNITUDE >= 1` trigger, retaining the largest requested target. A target increase starts the handler.

Listing 2 grants up to `CREDITS_PER_PULL` to a flow. It caps the new pull number at the target and publishes it through `PULL_NO`. If demand remains, the handler sets `NEEDS_MORE`, a loopback signal internal to the handler and shell. This binary flag requests further service through a `RAW >= 1` trigger. The transport schedules handlers, arbitrates flows, and paces pulls. After serving other flows, it can invoke this handler again on the loopback trigger.

LB: REPS uses static arrays to cache healthy, uncongested paths for later transmissions [7]. Listing 3 shows how it populates an EV cache on the ACK RX path. The handler derives the count of new ECN marks from a private baseline and caches the last reported path only when this count is zero. Indexed access through `set_arr` hides the array's storage address. A power-of-two cache size bounds the masked index. `ACK_EV` uses `LAST` to identify the path associated with the feedback. The handler's TX path (not shown) finds an EV in the cache and returns it to the transport. The transport could then add that EV to its table of active EVs.

This example also exposes a semantic limit of accumulation. A `LAST` path signal exposes one identity and cannot preserve all ACK-to-path associations in a coalesced batch. Exact per-packet cache updates therefore require coherent event delivery and sufficient compute resources, as in the synchronous simulator validation.

6 Implementation

Table 2 summarizes PCM mappings and testbeds. Our platform integrations take approximately 600 (htsim, C++),

System	Execution	Shell	Transport	Testbed
UET-htsim	Sync / Async	CPU Reference Shell	Ultra Ethernet	1,024-host, three-tier fat tree, 100 Gbit/s links, 4,150 B packets
CPU microbenchmark	Async	CPU Reference Shell	Synthetic trace	Dedicated Intel Xeon Gold 6140 core as generator and consumer
UCCL	Sync / Async	CPU Reference Shell	UCCL UC Verbs	Two AMD EPYC 9124 servers, each with two 800 Gbit/s ConnectX-8 NICs,
CX-8	Async	DOCA PCC Shell	RoCEv2	sixteen 100 Gbit/s planes per server, Spectrum-5 multi-plane, two-tier fabric
DPA microbenchmark	Async	DOCA PCC Shell	Synthetic trace	AMD EPYC 9124 core as generator and ConnectX-8 DPA threads as consumers
M-1P	Async	1P NIC Shell	Custom RDMA	Two 2×200 Gbit/s Microsoft 1P DPUs and an Arista DCS-7060X6 switch

Table 2: PCM platform shells and testbed systems.

1000 (UCCL, C++), 2500 (Microsoft 1P, C/Python), and 2000 (CX-8 DOCA PCC, C/Python) lines of code.

6.1 CPU Reference Shell

Our reference CPU shell is implemented in C++ and supports integration with UET-htsim and UCCL.

The handler specification is represented as a type tuple that instantiates a variadic template shell class. The instantiated shell class specializes a shell for a given handler. Fold expressions and `if constexpr` omit redundant checks. The shell class can be specialized with signal storage for synchronous or asynchronous execution. Synchronous storage uses ordinary C++ datatypes and operations. Asynchronous storage uses an atomic fetch-add for `SUM`, a store for `LAST`, and compare-and-exchange for `MIN`/`MAX`.

We compile the shell into a separate shared object. Datapaths load algorithms and use the shell factory to create per-flow shell instances. A virtual API flushes a datapath-input structure into signal accumulators. A mask selects feedback, byte counts, RTT, queue state, and path information to selectively apply feedback to the signal accumulators. Separate structure fields return window, rate, and entropy controls. We compile handlers with Clang and LLVM [27] and also provide optional custom passes to check handler safety, including detecting potentially unbounded execution paths.

UET-htsim: Simulator hooks map the simulator clock, ACK, NACK, ECN, LB and credit pull request events to the shell input structure. Synchronous mode applies controls before window enforcement, path selection or credit generation. We also extend the simulator's NIC scheduler to model an asynchronous control engine that executes handlers in a separate event chain.

UCCL: UCCL distributes flows across CPU threads that fragment buffers into multi-MTU chunks and provide reliable delivery over unreliable Verbs RDMA Writes. UCCL defaults to a software CC module that runs synchronously on the chunk-processing path. We connect it to our synchronous CPU shell. Synchronous PCM maps ACK, NACK, and RTT feedback and applies the resulting window inside the transport callback. We also extend UCCL to support an asynchronous CC plane that serves all datapath threads from an independent thread. The two planes communicate through the CPU reference shell's atomic storage. The PCM worker scans contexts and checks triggers. It publishes window updates on separate atomic cache lines for independent datapath reads.

6.2 ConnectX DOCA PCC Shell

The ConnectX-8 shell offloads PCM to Datapath Accelerator (DPA) cores through the DOCA PCC API. The DPA has 16 cores, each with 16 hardware threads. The shell uses DOCA PCC callbacks that run asynchronously relative to the hardware RoCEv2 engine [35]. Each per-flow callback runs the shell and returns a rate and RTT-probe requests. The shell generator uses the handler's specification to emit DOCA PCC glue code that is compiled using the DPA's DPACC compiler. This code wraps native SUM-based coalescing for TX-burst, ACK, NACK, and CNP events and handles RTT probes.

Per-flow state: DOCA PCC provides 48 application-context bytes per flow (RC QP). This space holds a handler context pool index, output rate, outstanding bytes, probe state, RTT, and debugging metadata. The first event atomically reserves a pool slot and initializes the flow's generated handler state.

DPA arithmetic: The DPA lacks floating-point hardware and supports only a few mathematical intrinsics in the fixed-point Q16.16 format. To support Swift and CUBIC, we implement arithmetic, comparison, root, reciprocal, and inverse-square-root macros.

These macros emulate fixed-point arithmetic in software using the open-source fixedptc library. The library supports 32- or 64-bit formats chosen at compile time. Our production implementation will replace these macros with a custom shell compiler pass that uses C arithmetic operators and a subset of the `math.h` API.

Native controls: The only datapath control knob that DOCA PCC supports is QP rate. The shell exposes rate control to PCM handlers in bits per 100 ns tick. It maps this value to ConnectX's native Q20 integer encoding of the link-rate fraction. For functional testing of purely window-based algorithms, the shell adapts Swift's native pacing method to approximate rate as $CWND/EWMA(RTT)$, using an exponentially weighted moving average of RTT [12, 26, 45]. Native window support in future ConnectX generations would let the shell select the window output.

6.3 Microsoft 1P NIC Shell

We implement a shell generator for Microsoft 1P NICs that run a custom hardware RDMA transport. The C shell runs as an asynchronous CC callback on 8 of the 192 hardware

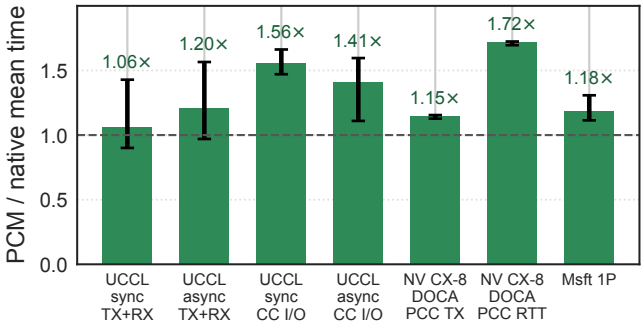


Figure 5: PCM mapping cost across three transports, averaged across 1–4 UCCL workers, 100–2000 ConnectX QPs, and 16 Microsoft 1P QPs. Whiskers show the minimum and maximum. UCCL CC I/O isolates CC cost, while TX+RX covers per-chunk work. In asynchronous UCCL runs, we measure the overhead incurred by UCCL datapath threads. The PCM control-plane worker runs on the dedicated SMT sibling of datapath thread 0, and its cost is excluded.

threads co-located with the RDMA engine. Like the DOCA PCC shell, it dispatches events to per-flow state and controls to native 1P APIs.

7 Evaluation

We answer four research questions:

1. *What does PCM's abstraction cost?*
2. *How does PCM support policy validation and reuse across transports?*
3. *How does PCM scale across datapaths?*
4. *What rates do PCM CC and LB handlers sustain?*

7.1 Experimental Setup

Live experiments scope: We implement the eleven handlers in Table 3 across the platforms in Table 2 and validate all of them in UET-htsim. We test all seven sender-CC handlers under live traffic on UCCL, ConnectX, and Microsoft 1P. The UCCL and ConnectX tests use ACK/RTT-only feedback on a fabric without ECN or trimming.

CPU and ConnectX DPA microbenchmarks: We implement a trace-replay microbenchmark to isolate performance at scales beyond our two-node, four-NIC testbed. A dedicated host CPU thread drives events. For CPU tests, it sends events through the CPU's atomic interconnect to an asynchronous reference shell on another hardware thread. For DPA tests, it sends events to DPA threads that poll host-DPA CQs and demultiplex completions into the DOCA PCC PCM shell. For a given handler, both DPA and CPU execute the PCM shell code that is executed for it when it serves real traffic.

When reporting microbenchmark results for sender-CCs, we use four event mixes and run each mix with three independent seeds. The ACK-only trace contains 100% ECN-free ACKs. The ECN-heavy trace contains 90% ECN-marked ACKs and 10% unmarked ACKs. The loss-heavy trace contains 80% loss events and 20% unmarked ACKs. The mixed trace contains 20% loss events, 40% ECN-marked ACKs, and 40% unmarked ACKs. On the DPA, we expand each synthetic trace event into native DOCA PCC events. Each loss event becomes two events: a TX burst and NACK. Each ACK becomes TX, RTT, and ACK events. For an ECN-marked ACK, we also add a CNP packet event to this sequence.

Algorithm	Key feature	Size (B)	LoCs
NewReno [24]	Loss-based	176	50
CUBIC [18]	Loss-based	240	95
DCTCP [1]	Loss/ECN-based	216	69
MP-RDMA [31]	ECN-based	128	27
Swift [26]	RTT-based	208	93
SMaRTT [6]	ECN/RTT/Trim hybrid	264	104
NSCC [11]	UET SMaRTT variant	320	172
EQDS [36]	Receiver-based	232	91
OPS [16]	Oblivious per-packet	96	38
REPS [7]	Caches "good" paths	232	70
STrack [28]	Tracks per-path congestion	1240	75

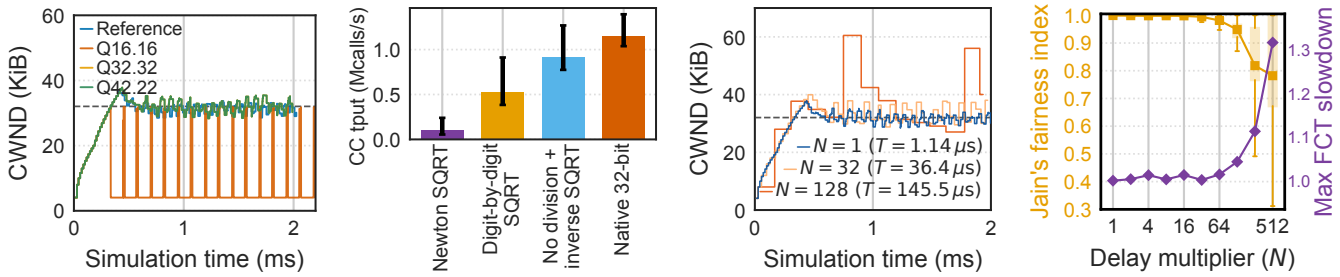
Table 3: CC and LB algorithms implemented in PCM.

We use synthetic traces to measure peak execution capacity of each shell and its handler. Our evaluation of LB and receiver-CC handlers is limited to synthetic traces because neither CX-8 nor M-1P NICs expose public APIs for LB or receiver-CC hooks. For LB, both CPU and DPA microbenchmarks reuse the sender-CC traces on RX path hooks, and supply synthetic TX events that indicate one packet ready for transmission. For EQDS, the CPU microbenchmark uses four receiver-side pull-target traces in which 0%, 25%, 50%, or 75% of events advance the target by 128 credit units (two grant quanta), while the remaining increments are sampled uniformly from 1–127 credit units.

On our Intel Xeon testbed, the CPU generator thread saturates the sibling hardware thread on the same physical core at 35 million events/s (1.15 Tbit/s assuming 4 KiB packets). In our DPA microbenchmark, an AMD EPYC CPU saturates the DPA at 75 million events/s (2.5 Tbit/s).

7.2 Cost of Abstraction

Figure 5 compares built-in CC algorithms with their PCM reimplementations under live traffic on three platforms. The algorithms are Swift in UCCL, rtt_template in DOCA PCC (CX-8), and a custom Microsoft 1P (M-1P) CC algorithm. The CX-8 and M-1P ports are excluded from Table 3. For matched algorithms, we compare control behavior under the tested traffic conditions through CWND-adjustment distributions or rate trajectories. The asynchronous UCCL bars in Figure 5 measure only the added datapath work and exclude the control plane worker. A dedicated worker is a resource cost of asynchronous control plane design, not an intrinsic PCM requirement.



(a) CWND numerical stability. (b) Arithmetic optimizations. (c) CWND fidelity under load. (d) Fairness and slowdown.

Figure 6: PCM-Swift under a 3 MB 8:1 UET-htsim incast with senders 25 μs apart. Panel (a) shows Leaf 1 CWND for the synchronous floating-point reference and asynchronous fixed-point handlers delayed by $T_h \approx 1.1$ μs. T_h is the mean processing time of the optimized Q42.22 variant in panel (b). Panel (b) shows four DPA arithmetic variants: three in Q42.22 and one using CX-8's DPA-native Q16.16 format. Panels (c) and (d) use scheduling delays of NT_h.

CPU-based transport: In UCCL, one to four datapath threads perform 64 KiB chunk fragmentation. PCM increases the isolated cost of CC input and output by $\approx 1.5\times$ in both modes. Native CC occupies 7.4–10.6% of the TX+RX path. This path includes chunk transmissions and ACK handling. Overall TX+RX costs average up to 1.06× and 1.20× native execution for synchronous and asynchronous PCM, respectively. Notably, native UCCL and both PCM extensions reach the same 1.56 Tbit/s practical-goodput rate with four datapath threads.

NIC offloading: Our CX-8 testbed uses the default DOCA PCC allocation of 48 of the DPA's 256 hardware threads, approximately one-fifth of the available threads. For TX-burst callbacks that issue RTT probes, PCM increases the cost from 214 to 246 ns (1.15×). The cost of RTT-probe-arrival callbacks that execute the handler increases from 392 to 676 ns (1.72×). We observe roughly 0.7 RTT-probe arrivals per TX-burst callback. This ratio corresponds to about one RTT callback per 1.4 TX callbacks. The combined overhead is 1.48×. On the M-1P testbed, PCM increases the ACK-reaction cost by 1.18×. The shell runs on 8 of the NIC's 192 hardware threads, one twenty-fourth of the available threads.

7.3 PCM-Swift Case Study

We follow the PCM workflow in Figure 1: validate Swift's numerical representation in UET-htsim, optimize its computation, and test its scheduling tolerance. We then deploy the same C policy source on three live platforms, changing deployment constants such as MTU and BDP in the specification. Platform shells provide target-specific arithmetic and native control mappings. We validate the resulting deployments through goodput, control behavior, and fairness.

We use Swift for end-to-end behavioral validation for two reasons. First, it performs one of the most complex computations among the supported algorithms. Its target-RTT and estimated-RTT computations involve division and square-root operations and together account for $\approx 90\%$ of the per-ACK processing cost. This makes Swift a demanding test of arithmetic performance on our NIC offloading platforms. Second, Swift requires only ACK and RTT feedback, matching the signals available in our Spectrum-5 fabric, and natively supports pacing rate control, the only datapath control available through ConnectX's DOCA PCC.

Step 1: Correctness testing. The CX-8 DPA is one of our offload targets. It lacks floating-point hardware, so we test fixed-point alternatives. The DPA-native 32-bit Q16.16 fixed-point data type has 16 integer and 16 fractional bits. In Figure 6a, we establish a stable fixed-point representation that preserves the tested numerical behavior. Swift's CWND output with Q16.16 is distorted.

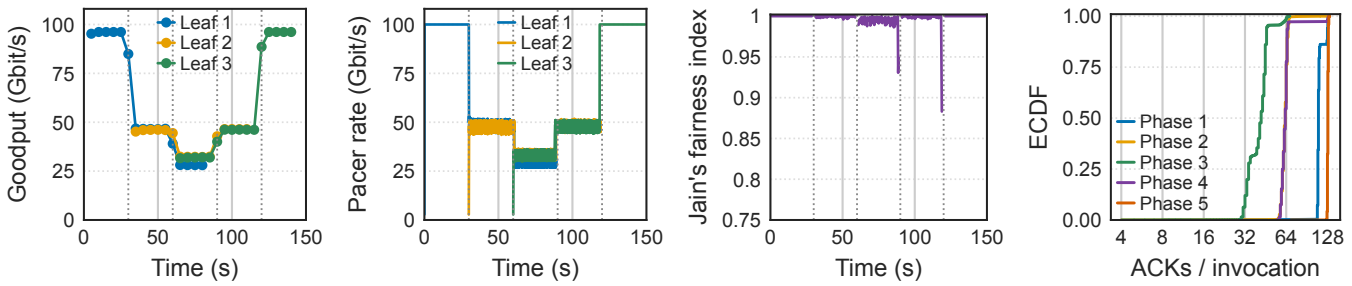
The Q16.16 variant's CWND oscillates between its one-packet initial window and approximately 32 KiB. This variant takes 8.76 ms to complete the flows, compared with 2.17 ms for the reference. It also causes 1030 retransmissions. The 64-bit Q32.32 and Q42.22 variants complete all flows within 0.3% of the reference's maximum FCT. They also achieve mean CWND fairness within 0.001 of the reference.

Step 2: Performance tuning. Our initial reference PCM-Swift computation follows the built-in htsim implementation. The naive Swift computation uses a floating-point square-root (sqrt) calculation to estimate the target delay.

Shell optimization. The fixedptc library's baseline Newton approximation, which implements sqrt in the ConnectX DPA shell, sustains only 0.105 million calls/s (Mcalls/s) per hardware thread (Figure 6b). An optimized digit-by-digit sqrt algorithm yields 0.532 Mcalls/s.

Handler optimization. We further improve performance through algebraic transformations of Swift arithmetic. We replace divisions and the square root in Swift's target-delay computation with multiply-only reciprocal and inverse-square-root operations. The transformed handler exceeds 0.9 Mcalls/s per DPA thread, with a mean processing time of approximately 1.1 μs per input event.

This example demonstrates Swift policy reuse during optimization: after each refinement, we revalidated the policy in the simulator as described in Step 1 and then benchmarked it



(a) 64 KiB RDMA Write. (b) Pacer rate evolution. (c) Pacer rate fairness. (d) Coalescing during logging.

Figure 7: PCM-Swift on ConnectX-8 DPA controls end-to-end 3:1 incast on one 100 Gbit/s plane. Three leaves issue back-to-back 64 KiB RDMA Writes in five phases, starting 30 seconds apart and stopping in order.

unchanged on the DPA.

Step 3: Scheduling tolerance bound. We characterize Swift's sensitivity to asynchronous scheduling delay under the selected incast workload. After each invocation, the simulator waits $T = NT_h$ before invoking the flow's handler again, where $T_h \approx 1.1$ μs is the mean processing time for optimized PCM-Swift on the CX-8 DPA. The multiplier N models reduced handler availability on a shared DPA thread. We impose this delay directly without adding competing traffic flows. During the delay, the transport continues processing packets, and the shell accumulates the flow's ACK and RTT feedback for the next invocation.

Workload-specific tolerance. Figure 6c shows Flow 1 under delay multipliers $N = 1$, 32, and 128. Larger scheduling delays produce coarser CWND updates. Figure 6d quantifies the resulting changes in fairness and completion time. Median CWND fairness stays at or above 0.99 from $N = 1$ through $N = 32$. It degrades to 0.818 at $N = 256$ and 0.782 at $N = 512$, corresponding to reaction delays of 291 and 582 μs. Maximum FCT increases by 1.11× and 1.32× relative to the $N = 1$ result, respectively. At $N = 32$, the 36.4 μs reaction interval spans the transmission time of 111 packets at 100 Gbit/s with a 4 KiB MTU.

Falcon's FAE study also reports sensitivity to reaction delays: flow completion time increases when the control reaction delay exceeds 32 μs at 200 Gbit/s [39]. Our simulated sensitivity also provides context for Presto's 50 μs control-plane polling period [38].

Step 4: End-to-end NIC offloading deployment. We now evaluate PCM-Swift offloaded to a ConnectX-8. We run a 3:1 incast from three leaf NICs to one root NIC (Figure 7). Each sender runs `ib_write_bw` with contiguous 64 KiB RDMA Writes across a single isolated 100 Gbit/s plane of an 800 Gbit/s port. Traffic from all senders crosses two tiers of switches toward the root NIC, with an RTT of ≈ 15 μs.

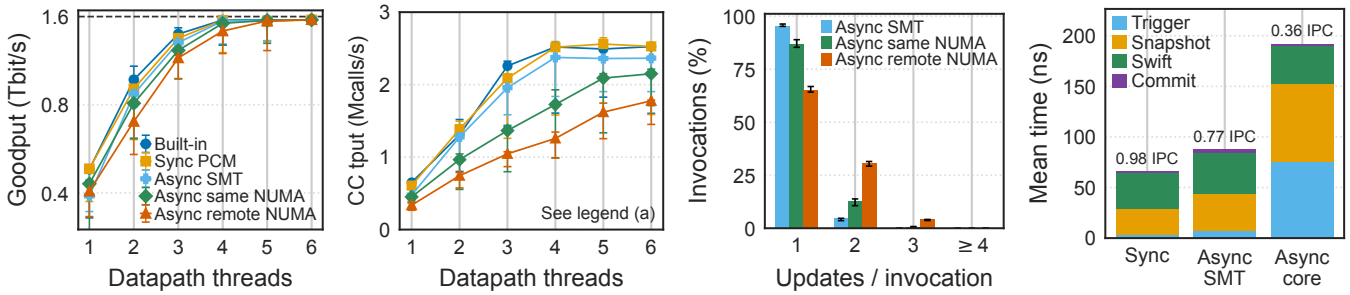
Control instrumentation. To expose every live PCM-Swift control decision, we enable diagnostic mode, which reports handler outputs after every invocation. Emitting and flushing these records on the DPA callback path adds multiple DPA–host PCIe RTTs to the handler time. We exploit this delay to exercise the event-coalescing mechanism and validate the CWND-to-rate adapter used in our Swift implementation (Section 6.2). The next DOCA PCC callback delivers the ACK feedback coalesced during logging to the PCM shell's SUM accumulator.

Fairness under coarse feedback. Figure 7d shows that the median batch size decreases as senders throttle to their fair shares. A lone sender produces median batches of 110–129 ACKs. With two senders, the median batch contains 63 ACKs. With three senders, it contains 42 ACKs. Despite the coarse instrumented schedule, Figure 7c shows that PCM-Swift converges to stable fair-share operation as contenders enter and leave. We exclude the first and last second of each phase when averaging pacer rate fairness. In each of the five phases, mean Jain's fairness indices for NIC pacer rate governed by the policy and Write goodput are at least 0.99.

The production shell build removes per-invocation diagnostics. With one flow per 100 Gbit/s plane, the optimized PCM-Swift handler sustains one invocation per four ACKs. Production reaction intervals also depend on ConnectX's RDMA engine ACK pacing, DOCA PCC's internal callback scheduling and coalescing, and system load. We study the effect of system load in Section 7.4. This live 3:1 incast experiment complements the controlled scheduling-delay sweep in simulation by testing Swift under ConnectX's native event coalescing and rate enforcement.

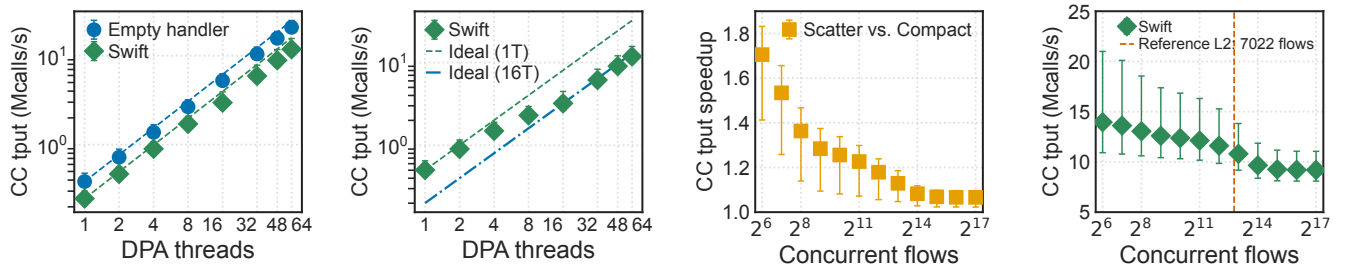
Step 5: Crossing the transport boundary. We tested PCM-Swift on Microsoft 1P under end-to-end and synthetic traffic. We observe that it reacts to synthetically injected high RTT samples by reducing its CWND, matching Swift's control response in UET-htsim. We discuss UCCL in detail as an architectural counterpart to NIC offloading on CX-8 and M-1P. Our UCCL study tests whether co-locating PCM preserves line-rate goodput and fine-grained reactions under CPU saturation. Appendix A evaluates PCM-Swift's CWND response to injected RTT spikes.

Goodput scaling. We deploy the portable PCM handler in UCCL to control the injection of 64 KiB UC Write chunks, as Figure 8a shows. We adjust only Swift parameters in the specification to match UCCL's parameters. The handler



(a) 1 MB RDMA Write. (b) PCM handler-call rate. (c) ACK SUM signal coalescing. (d) Worker shell breakdown.

Figure 8: PCM-Swift in CPU-based UCCL across sixteen ConnectX-8 100 Gbit/s planes (1.6 Tbit/s) and 16 flows on an AMD EPYC 9124 CPU. Panel (a) shows the median and extrema from three runs. Panels (b) and (c) show 99th-percentile top whiskers. Panel (d) profiles the event-trace microbenchmark on an Intel Xeon Gold 6140 and annotates IPC.



(a) Strong scaling, 3,072 flows. (b) Weak scaling, 48 flows/thread. (c) Impact of DPA affinity. (d) Flow scaling, 64 threads.

Figure 9: PCM-Swift scaling under DPA-saturating trace replay. Markers show means of means across four traces and three seeds. Whiskers span individual means. Dashed lines in panels (a) and (b) show ideal scaling.

retains shared policy arithmetic and object interfaces. Built-in Swift and PCM-Swift reach practical RoCEv2 line rate across sixteen planes with four engine workers.

Handler-call scaling. Figure 8b shows how the PCM worker's handler-call rate in response to UCCL ACKs changes with the number of dataplane workers. Synchronous PCM tracks the built-in handler-call baseline through four datapath engines and plateaus when the system saturates all sixteen planes. Figure 8c shows that the median fraction of invocations with multiple ACK updates is below 5% on SMT, 13.3% on a same-NUMA core, and 34.8% on remote NUMA.

Control-execution cost. Figure 8d shows our PCM shell microbenchmark on the Xeon platform. The breakdown shows 66 ns for synchronous execution, 87 ns on SMT, and 192 ns on a separate core. The `std::chrono` operation accounts for a large part of the snapshot cost and takes ≈ 20 ns. Swift remains near 35–40 ns. The synchronous and asynchronous shells execute nearly the same number of instructions. The cache-coherence protocol of the atomic interconnect decreases IPC from 0.98 (Sync) to 0.77 (Async SMT) and 0.36 (Async same NUMA).

Cost of alternative designs. An alternative to PCM's asynchronous control plane is to integrate UCCL with CCP [33], whose datapath DSL accumulates measurements and triggers reports to host-side policies through IPC. A minimal CCP filter for ACK accumulation and triggering alone compiles to four DSL bytecodes, but glue code executes approximately 880 x86-64 instructions to enter and evaluate the filter, before report generation or IPC. Our separate NewReno profile on the same Xeon measures 364 ns for DSL evaluation.

In comparison, with operands in registers, PCM-Swift's ACK-count accumulation (SUM) and RTT-sample update (LAST) each compile to one instruction, while its ACK trigger (`DELTA >= 1`) reduces to a comparison and conditional branch. On the Xeon, signal and control exchange uses single-instruction atomic loads, stores, and additions. Swift shell invocations, including policy execution, take 66–192 ns (Figure 8d). Moving CCP's host-side policy execution onto ConnectX would also require replacing Portus's Rust runtime its Rust policies for the DPA's C11 environment [35].

7.4 PCM Offloading Scalability

We analyze PCM offload performance beyond the scale of end-to-end traffic in our four-NIC ConnectX testbed. Figure 9 reports handler performance.

Strong scaling. In Figure 9a, the working-set size remains fixed across threads. The empty handler and Swift reach 75–85% of ideal throughput. With 64 of the DPA's 256 hardware threads and 3072 flows, PCM-Swift sustains 11.83 Mcalls/s.

Weak scaling and thread affinity. Starting at 16 threads,

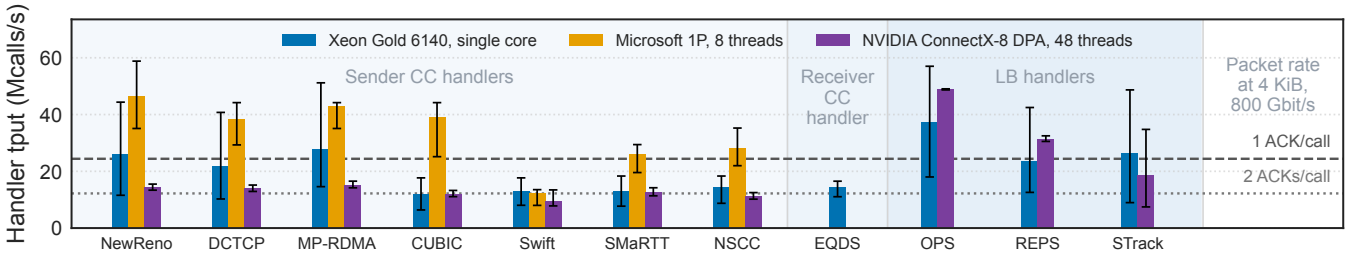


Figure 10: CC and LB handler performance. Whiskers show Microsoft 1P extrema and Intel Xeon 99.9th percentiles. We benchmark Xeon and NVIDIA ConnectX-8 with event traces for 192 flows. 1P runs live RDMA Write traffic across 16 QPs.

PCM-Swift achieves 30–40% of the ideal throughput extrapolated from the single-thread reference (1T) in Figure 9b. Performance then stabilizes. At 64 threads, Swift uses four fully subscribed DPA cores. Swift reaches 90% of a weak-scaling curve derived from the 16-thread (16T) reference. This reference saturates all DPA core threads.

This result signifies the effect of the DPA's SMT microarchitecture. Under the default compact mapping, increasing the thread count from 1 to 16 gradually increases pressure on the same ALU. Figure 9c reports the relative speedup of scattered mappings over compact mappings and shows the impact of shared-ALU pressure. With 4,096 flows, 64 scattered threads, and four threads per ALU, PCM-Swift reaches 13.5 Mcalls/s. Compact mapping reaches 11.6 Mcalls/s. Our Microsoft 1P experiments, which assign four hardware threads to one ALU, exhibit the same trend.

Flow scaling. Figure 9d shows Swift throughput decreasing from 11.61 to 10.81 Mcalls/s as the working set grows from 0.875 to 1.75 MiB. To interpret this trend, we use published BlueField-3 DPA memory-hierarchy measurements as a reference [10]. Its reported L2 capacity corresponds to >7000 Swift flow contexts, marked by the vertical guide. The observed decline is consistent with increased cache pressure near this reference capacity. At 131072 flows, compact and scattered placements both sustain nearly 9 Mcalls/s.

7.5 Performance Across Handlers

We evaluate PCM handlers across the supported shells in Figure 10. This experiment tests shell reuse across policies, policy portability across shells, and overall handler-performance trends across platforms and algorithms. For the asynchronous CPU shell, the trace microbenchmark saturates each worker, as does the DPA benchmark.

One Xeon core sustains 11.8–37.3 million asynchronous controller calls/s, and eight Microsoft 1P threads provide 12.2–46.3 million calls/s. Falcon reports 20 million FAE events/s on one Neoverse-N1 core, sustained across connection counts through state prefetching [39, Fig. 22a]. PCM-Swift in Figure 10 takes on average 77.1 ns/ACK on Xeon, comparable to the 66 ns RUE service interval modeled by Falcon's simulator, which presumably models the performance of Neoverse-N1 cores. This result provides architectural context rather than a matched performance baseline.

ConnectX's 48 DPA threads sustain 9.5–15.1 million CC calls/s. For the simpler, less compute-intensive LB shell, the same 48 DPA threads sustain 18.6–48.8 million LB calls/s. For a lightweight scheme such as MP-RDMA, discussed in Section 5.4, these capacities approach per-packet policy demand on an 800 Gbit/s link. For Swift, the most compute-intensive algorithm, the capacity corresponds to one input event per two to three ACKs.

7.6 Lessons Learned

Optimize policies and backends independently. PCM demonstrates two levels of reuse. Vendors optimize backend arithmetic to accelerate native execution across policies, while developers carry algebraic improvements in portable handlers across supported platforms.

Validate numerical behavior and reaction timing. Swift's fixed-point failure and scheduling sensitivity show why source compatibility needs behavioral validation. Accumulation preserves feedback counts, but delayed reactions can still reduce fairness and increase flow completion times. Use simulation to validate numerical representations and establish a workload-specific reaction budget.

Provision NIC compute for the required reaction rate. Thread placement and flow-state footprint shape processing capacity: hardware threads share execution resources, and growing state increases cache pressure. Provision workers to meet the required reaction rate within these constraints.

8 Conclusion

PCM makes portable congestion management practical for high-performance RDMA. Our programming model spans eleven policies and four datapaths, from simulation and software transports to NIC offload. PCM policies sustain per-chunk control at 1.56 Tbit/s goodput in UCCL and achieve near-per-ACK processing capacity on vendor 400G and 800G NICs. With its design informing ongoing Ultra Ethernet standardization, PCM opens a path for congestion-control and load-balancing innovations to reach heterogeneous datacenters through a shared interface.

Acknowledgments

We thank Shuhao Li, Zhiyi Hu, Inho Cho, Nadeen Gebara, Luca Piccolboni, and Terry Lam for their feedback and assistance with the experiments. This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement PSAP, No. 101002047) and a CAF America grant. We also thank the Swiss National Supercomputing Center (CSCS) for providing the computational resources used in this work.

References

[1] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), SIGCOMM '10, Association for Computing Machinery, p. 63–74.

[2] AMD. Pensando vulcano 800 ai nic. https://www.amd.com/content/dam/amd/en/documents/pensando-business-docs/product-brief/amd-vulcano-ai-nic-networking-product-brief.pdf, 2026.

[3] ARASHLOO, M. T., GHOBADI, M., REXFORD, J., AND WALKER, D. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), pp. 108–114.

[4] ARASHLOO, M. T., LAVROV, A., GHOBADI, M., REXFORD, J., WALKER, D., AND WENTZLAFF, D. Enabling programmable transport protocols in {High-Speed}{NICs}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 93–109.

[5] BONATO, T., ABDOUS, S., KABBANI, A., GHALAYINI, A., GEBARA, N., LAM, T., AGARWAL, A., CHEN, T., YU, Z., TARANOV, K., ELHADDAD, M., DE SENSI, D., GHORBANI, S., AND HOEFLER, T. Uno: A one-stop solution for inter- and intra-data center congestion control and reliable connectivity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2025), SC '25, Association for Computing Machinery, p. 1195–1210.

[6] BONATO, T., KABBANI, A., GHALAYINI, A., AGARWAL, A., DE SENSI, D., PAN, R., RAICIU, C., HANDLEY, M., BRODSCHI, M., SCHNEIDER, T., ET AL. Smartt: Sender-based marked rapidly-adapting trimmed & timed transport. *arXiv preprint arXiv:2404.01630* (2024).

[7] BONATO, T., KABBANI, A., GHALAYINI, A., PAPAMICHAEL, M., DOHADWALA, M., GIANINAZZI, L., KHALILOV, M., ACHERMANN, E., DE SENSI, D., AND HOEFLER, T. Reps: Recycled entropy packet spraying for adaptive load balancing and failure mitigation. In *Proceedings of the 21st European Conference on Computer Systems* (2026), pp. 225–246.

[8] BROADCOM. P1800go - 1 x 800gbe pcie nic. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/p1800go, 2025.

[9] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. Bbr: congestion-based congestion control. *Commun. ACM 60*, 2 (Jan. 2017), 58–66.

[10] CHEN, X., ZHANG, J., FU, T., SHEN, Y., MA, S., QIAN, K., ZHU, L., SHI, C., ZHANG, Y., LIU, M., ET AL. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)* (2024), IEEE, pp. 1–12.

[11] CONSORTIUM, U. E. Ultra ethernet specification v1.0.1. https://ultraethernet.org/wp-content/uploads/sites/20/2025/10/UE-Specification-1.0.1.pdf.

[12] CONSORTIUM, U. E. uec-transport-simulation-code. https://github.com/ultraethernet/uet-htsim, 2025.

[13] DE SENSI, D., DI GIROLAMO, S., MCMAHON, K. H., ROWETH, D., AND HOEFLER, T. An in-depth analysis of the slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), IEEE, pp. 1–14.

[14] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[15] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[16] DIXIT, A., PRAKASH, P., HU, Y. C., AND KOMPELLA, R. R. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM* (2013), pp. 2130–2138.

[17] GANGIDI, A., MIAO, R., ZHENG, S., BONDU, S. J., GOES, G., MORSY, H., PURI, R., RIFTADI, M., SHETTY, A. J., YANG, J., ET AL. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (2024), pp. 57–70.

[18] HA, S., RHEE, I., AND XU, L. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review 42*, 5 (2008), 64–74.

[19] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, Association for Computing Machinery, p. 29–42.

[20] HINZ, J.-T., ADDANKI, V., GYÖRGYI, C., JEPSEN, T., AND SCHMID, S. Tcp's third eye: Leveraging ebpf for telemetry-powered congestion control. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions* (2023), pp. 1–7.

[21] HOEFLER, T., SCHRAMM, K., SPADA, E., UNDERWOOD, K., ALEXANDER, C., ALVERSON, B., BOTTORFF, P., CAULFIELD, A., HANDLEY, M., HUANG, C., ET AL. Ultra ethernet's design principles and architectural innovations. *arXiv preprint arXiv:2508.08906* (2025).

[22] HOPPS, C. Rfc2992: Analysis of an equal-cost multi-path algorithm, 2000.

[23] HUANG, H., ZHANG, J., CHEN, X., SONG, Z., QIN, J., AND WANG, Z. {SwCC}:{Software-Programmable} and {Per-Packet} congestion control in {RDMA} engine. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)* (2025), pp. 1243–1260.

[24] IETF. Rfc 6582. https://datatracker.ietf.org/doc/rfc6582/.

[25] JIANG, W., HU, Y., WANG, K., HAN, X., WEI, H., ZHANG, Y., YE, J., AND DAI, H. Odin: Rethinking congestion control under all-to-all traffic. In *Proceedings of the ACM SIGCOMM 2026 Conference* (2026), pp. 1246–1263.

[26] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., ET AL. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 514–528.

[27] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[28] LE, Y., PAN, R., NEWMAN, P., BLENDIN, J., KABBANI, A., JAIN, V., SIVARAMU, R., AND MATUS, F. Strack: A reliable multipath transport for ai/ml clusters. *arXiv preprint arXiv:2407.15266* (2024).

[29] LEI, Y., LEE, D., ZHAO, L., KURNIAWAN, D., KIM, C., JEONG, H., KIM, C., CHOI, H., YU, L., KRISHNAMURTHY, A., ET AL. {FAST}: An efficient scheduler for {All-to-All}{GPU} communication. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)* (2026), pp. 2515–2531.

[30] LI, S., ZHAO, Y., VARMA, R., SALPEKAR, O., NOORDHUIS, P., LI, T., PASZKE, A., SMITH, J., VAUGHAN, B., DAMANIA, P., ET AL. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[31] LU, Y., CHEN, G., LI, B., TAN, K., XIONG, Y., CHENG, P., ZHANG, J., CHEN, E., AND MOSCIBRODA, T. Multi-Path transport for RDMA in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 357–371.

[32] MA, X., AND PATTERSON, D. Challenges and research directions for large language model inference hardware. *Computer 59*, 5 (2026), 55–64.

[33] NARAYAN, A., CANGIALOSI, F., RAGHAVAN, D., GOYAL, P., NARAYANA, S., MITTAL, R., ALIZADEH, M., AND BALAKRISHNAN, H. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 30–43.

[34] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 385–398.

[35] NVIDIA. Doca pcc. https://docs.nvidia.com/doca/sdk/doca-pcc/index.html.

[36] OLTEANU, V., ERAN, H., DUMITRESCU, D., POPA, A., BACIU, C., SILBERSTEIN, M., NIKOLAIDIS, G., HANDLEY, M., AND RAICIU, C. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 761–777.

[37] SHALEV, L., AYOUB, H., BSHARA, N., AND SABBAG, E. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro 40*, 6 (2020), 67–73.

[38] SHASHIDHARA, R., KAUFMANN, A., AND PETER, S. Presto: A match-action tcp stack for the terabit era. In *Proceedings of the ACM SIGCOMM 2026 Conference* (2026), pp. 1360–1375.

[39] SINGHVI, A., DUKKIPATI, N., CHANDRA, P., WASSEL, H. M., SHARMA, N. K., REBELLO, A., SCHUH, H., KUMAR, P., MONTAZERI, B., BANSOD, N., ET AL. Falcon: A reliable, low latency hardware transport. In *Proceedings of the ACM SIGCOMM 2025 Conference* (2025), pp. 248–263.

[40] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (USA, 2012), NSDI'12, USENIX Association, p. 2.

[41] ZENG, G., BAI, W., CHEN, G., CHEN, K., HAN, D., ZHU, Y., AND CUI, L. Congestion Control for Cross-Datacenter Networks . *IEEE/ACM Transactions on Networking 30*, 05 (Oct. 2022), 2074–2089.

[42] ZENG, H., SI, M., BALAJI, P., CHEN, Y., CHU, C.-H., GANGIDI, A., KANNAN, P., LIU, B., HASAN, S., HE, D., ET AL. Connecting 100k+ gpus: Building the communication stack for large-scale llm training. In *Proceedings of the ACM SIGCOMM 2026 Conference* (2026), pp. 505–518.

[43] ZHAO, C., MIN, J., LIU, M., AND KRISHNAMURTHY, A. {White-Boxing}{RDMA} with {Packet-Granular} software control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)* (2025), pp. 427–449.

[44] ZHAO, Y., GU, A., VARMA, R., LUO, L., HUANG, C.-C., XU, M., WRIGHT, L., SHOJANAZERI, H., OTT, M., SHLEIFER, S., ET AL. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[45] ZHOU, Y., CHEN, Z., MAO, Z., LAO, C., YANG, S., KANNAN, P. G., ZHANG, X., GAO, J., ZHAO, Y., WU, Y., ET AL. {UCCL-Tran}: An extensible software transport layer for {GPU} networking. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)* (2026), pp. 1143–1166.

[46] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, Association for Computing Machinery, p. 523–536.

A Behavioral Validation of PCM-Swift in UCCL

We compare the CWND behavior of PCM-Swift, deployed through the reference CPU shell, with that of native UCCL Swift. We evaluate both synchronous and asynchronous PCM integrations with UCCL. The asynchronous CPU shell supports relaxed and locked snapshot modes (Section 5.3). In locked snapshot mode, the datapath brackets atomic signal updates with sequence numbers. The control plane worker retries its sequential signal reads until the same even sequence number precedes and follows the copy. Relaxed snapshot mode accepts the first copy.

We model transient network congestion by injecting a synthetic 500 μs RTT every 25th CC update. Figure 11 compares nonzero window adjustments under RTT injection for native UCCL Swift, the synchronous CPU shell, and both asynchronous snapshot modes. In the asynchronous runs, the control plane worker runs on a separate core. Across all modes, the range of PCM-Swift's CWND adjustments closely tracks that of native UCCL Swift.

The median nonzero adjustments in relaxed and locked snapshot modes differ by 1.2%. Their 1st and 99th percentiles differ by less than 1%. The datapath observes an unchanged window in 10.9% of samples in relaxed snapshot mode and 10.7% in locked snapshot mode. When the worker runs on the datapath thread's SMT sibling, median nonzero adjustments in relaxed and locked snapshot modes differ by 0.7%.

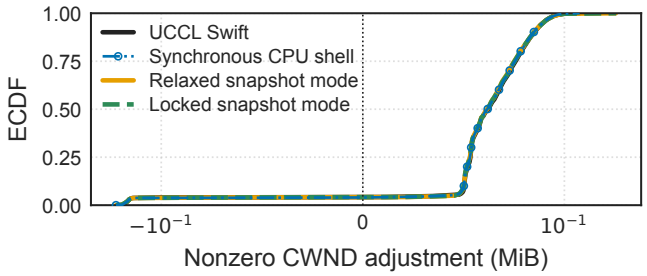


Figure 11: Nonzero CWND adjustments under RTT injection. Each curve uses 400,000 datapath observations from a single run.

B Specification for the MP-RDMA Handler

```yaml
controls:
  - name: CWND
    control: CWND_BYTES
signals:
  - name: ACKED_BYTES
    type: DATA_ACK_BYTES
    accumulation: SUM
    trigger_type: DELTA
    trigger_op: >=
    trigger_threshold: 1
  - name: NACKED_BYTES
    type: DATA_NACK_BYTES
    accumulation: SUM
    trigger_type: DELTA
    trigger_op: >=
    trigger_threshold: 1
variables:
  - name: PREV_ACKED_BYTES
    type: uint
    initial_value: 0
  - name: PREV_NACKED_BYTES
    type: uint
    initial_value: 0
constants:
  - name: MTU
    type: uint
    value: 4096
```

Listing 4: ACK-handling specification for Listing 1.