OSCAR: *O*(1)-Step Convergence And Readily-deployable Congestion Control

Zhaochen Zhang, Feiyang Xue, and Rui Ning, *Nanjing University;* Keqiang He, *Shanghai Jiao Tong University;* Gianni Antichi, *Politecnico di Milano & Queen Mary University of London;* Jiaqi Gao, *unaffiliated;* Zhimeng Yin, *City University of Hong Kong;* Kexin Liu, Rui Li, Zhengqi Cui, Zhehao Lin, Peirui Cao, Guihai Chen, and Chen Tian, *Nanjing University*

https://www.usenix.org/conference/nsdi26/presentation/zhang-zhaochen

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

OSCAR: $O(1)$-Step Convergence And Readily-deployable Congestion Control

Zhaochen Zhang[†], *Feiyang Xue*[†], *Rui Ning*[†], *Keqiang He*[△], *Gianni Antichi*[□], *Jiaqi Gao*[+], *Zhimeng Yin*[◇], *Kexin Liu*[†], *Rui Li*[†], *Zhengqi Cui*[†], *Zhehao Lin*[†], *Peirui Cao*[†*], *Guihai Chen*[†], *Chen Tian*[†]
[†]*State Key Laboratory for Novel Software Technology, Nanjing University*
[△]*Shanghai Jiao Tong University* [+]*Unaffiliated*
[□]*Politecnico di Milano & Queen Mary University of London* [◇]*City University of Hong Kong*

Abstract

Datacenter CCs typically target full bandwidth utilization and minimal queueing delay and strive to converge to these targets as quickly as possible. State-of-the-art CCs exhibit different convergence speeds, with the fastest ones converging in $O(1)$ steps, which means reaching the target in constant time regardless of network conditions. However, their reliance on network features makes them not readily deployable. For instance, precise-INT-based CCs, such as HPCC and PowerTCP, achieve $O(1)$-step convergence through MIMD operations based on precise congestion information from the lengthy INT header, which is challenging to support for high-speed commodity hardware. Our key insight is that delay and delay gradient can exhibit precision comparable to INT, enabling $O(1)$-step convergence without specialized network features. Based on this insight, we propose OSCAR, the first $O(1)$-Step Convergence And Readily-deployable CC. OSCAR introduces novel techniques to accurately estimate the delay gradient with minimal overhead, eliminate overreaction in MIMD updates, and coordinate independent control loops to converge to one target. Testbed evaluations demonstrate OSCAR can rapidly converge to the fair share under real-world noise. In large-scale simulations with realistic workloads, OSCAR consistently outperforms precise-INT-based CCs by 12%-48% on average FCT and 40%-74% on tail FCT.

1 Introduction

Datacenter congestion control (CC) algorithms play a vital role in providing high throughput and low latency data transmission for critical services such as distributed AI model training [33, 54, 75], high-performance distributed storage [16,36], resource disaggregation [29,34] and beyond. To meet these goals, CCs [7, 11, 14, 55, 57, 59, 61, 72, 85, 88] typically target full bandwidth utilization and minimal queueing delay, and strive to converge to these targets as quickly as possible. With the rapid evolution of network bandwidth [33,54,75] and the prevalent on-off traffic patterns [75, 77], the convergence speed of CCs has become increasingly critical [14, 88].

[*] Peirui Cao is the corresponding author.

In particular, it is worth noting that state-of-the-art CCs exhibit different convergence speeds. We employ the big O notation, typically used in algorithmic time complexity, to characterize how the CC's convergence time scales with the gap between its current and target states. This gap is analogous to the input size in time complexity analysis (§ 2.1). The ideal CCs should converge in $O(1)$ steps, *i.e.*, the convergence time is independent of the gap between current and target states, yielding predictable performance under diverse network dynamics [83]. Existing $O(1)$-step convergence CCs fall into three categories. (i) Precise-INT-based CCs such as HPCC [61] and PowerTCP [7] achieve $O(1)$-step convergence through MIMD (multiplicative increase/multiplicative decrease) operations based on precise congestion information obtained from INT headers. (ii) Receiver-driven CCs [20, 35, 43, 48, 72] enable $O(1)$-step convergence for last-hop congestion by adapting transmissions based on credits sent from receivers. (iii) Switch-driven CCs [8, 14, 55] utilize switches to manage network states and proactively instruct senders to adjust their rates for $O(1)$-step convergence.

Despite their effectiveness, $O(1)$-step convergence CCs are *not readily deployable* due to their assumption on network features (§ 2.2). Precise-INT-based CCs require network hardware to support lengthy INT header parsing, which is shown to be challenging without compromising throughput [12, 13, 64, 85]. Receiver-driven CCs [20, 25, 35, 43, 48, 72] typically require a well-provisioned and symmetric datacenter topology, which is hard to satisfy due to frequent link failures [39, 62, 69, 94, 95]. Moreover, some receiver-driven CCs rely on features such as packet trimming [43] or symmetric routing [25], which can lack widespread hardware support. Similarly, switch-driven CCs [8, 14, 55] require computational capabilities that are unavailable in commodity switches. Given these deployment challenges, we pose the question: *Can a readily deployable CC achieve $O(1)$-step convergence?*

The answer is yes. The key insight is that delay and delay gradient can exhibit precision comparable to INT, enabling $O(1)$-step convergence through MIMD operations without

reliance on specialized hardware or network topologies. While prior studies have separately identified weaker forms of our insight (*e.g.*, MIMD converges faster [55, 61], delay and its gradient are effective congestion signals [22, 57, 70]), to the best of our knowledge, we are the first to explicitly establish this precision equivalence and delineate the specific constraints under which it holds. Based on this insight, we propose **OSCAR, the first $\underline{O}(1)$-Step Convergence And Readily-deployable CC.**

Our contributions are summarized as follows:

• **We present two design principles for CC aiming for $O(1)$-step convergence and deployability.** (i) *MIMD reaction to precise congestion signals is essential.* Our analysis in § 3.1 reveals that MIMD reaction to precise congestion signals can achieve $O(1)$-step convergence while AIMD reaction fails even with precise signals. (ii) *Delay and delay gradient can serve as precise congestion signals to achieve $O(1)$-step convergence, provided their intrinsic constraints are addressed.* As demonstrated in § 3.2, under constraints that the bottleneck queue is neither empty nor full, the delay can precisely reflect the aggregate window of flows if flows are all window-bounded, and the delay gradient can precisely reflect the total rate if flows are all rate-bounded. As delay and delay gradient are precise under mutually exclusive conditions, *i.e.*, whether flows are window-bounded or rate-bounded, a CC should employ them complementarily to consistently achieve $O(1)$-step convergence.

• **We propose several techniques to overcome the challenges of applying the above-mentioned principles in the design of OSCAR.** (i) Precise and low-overhead calculation of delay gradient in high-speed networks is challenging. We propose the Batched Least Squares (BLS) algorithm to compute delay gradients with constant time-space overhead (§ 4.1.1). (ii) Achieving $O(1)$-step convergence necessitates large update steps, which risk overreaction, a problem we observed in existing precise-INT-based CCs when congestion lasts over an RTT. OSCAR elegantly eliminates overreaction by updating based on the sending state synchronized with congestion signals, rather than the current sending state (§ 4.1.2). (iii) OSCAR utilizes delay and delay gradient complementarily. As delay can reflect the aggregate window, OSCAR employs it to control the window, targeting to stabilize the queueing delay. Similarly, OSCAR employs the delay gradient to control the sending rate, targeting to equalize the aggregate sending rate with the line rate. Since these two control loops have distinct control variables and targets, directly employing them can lead to conflicts. Our analysis (Theorem D.2) derives the control law that can simultaneously achieve the targets of two control loops, enabling their collaboration (§ 4.2).

Through careful design, OSCAR outperforms Precise-INT-based CCs in three aspects. First, OSCAR eliminates the bandwidth overhead of INT headers. Second, OSCAR prevents oscillations caused by overreactions in MIMD control. Third, OSCAR's absence of overreactions allows it to perform MI more frequently, improving responsiveness to underutilization.

• **We implement OSCAR and conduct comprehensive evaluations in both testbed and simulation.** We implement OSCAR in DPDK. Overhead analysis indicates that OSCAR incurs less computation overhead than most of the datacenter CCs (§ 5). Testbed evaluations confirm that OSCAR delivers superior performance in noisy real-world environments (§ 6.2). Evaluation of BLS proves its delay gradient calculation is robust to noise, achieving 13.4× accuracy improvement over methods used by TIMELY, the state-of-the-art delay-gradient-based CC. In all-to-all communication [46, 73], OSCAR outperforms Swift and TIMELY by up to 93.9% on average and 51.4% at the tail.

In large-scale simulations with realistic workloads, OSCAR's average Flow Completion Time (FCT) for small flows (< 1 BDP) is 7.6% longer than HPCC's due to HPCC's zero-queue target state. In contrast, OSCAR reduces large flow FCT by up to 62.2% compared to HPCC. If HPCC is configured with the same target state as OSCAR, OSCAR outperforms it by 10%-42% across all flow sizes. Compared to precise-INT-based CCs, OSCAR reduces average FCT by 12%-48%, tail FCT by 40%-74% considering all flows, and reduces the completion time of collective communications used in model training by up to 17%. Moreover, unlike precise-INT-based CCs that rely on the strict assumption of a single forwarding path [61], OSCAR is compatible with per-packet LB (§ 4.4) and outperforms all tested non-INT-based CCs by 17% to 76% in such environments.

As a contribution to the community and to ensure reproducibility, our artifacts are made publicly available [90]. *This work does not raise any ethical issues.*

2 Motivation

2.1 CCs Have Different Convergence Speeds

We employ the big O notation, typically used in algorithmic time complexity, to characterize how the CC's convergence time[1] scales with the gap between its current and target states. This gap (denoted as Δ) is analogous to the input size in time complexity analysis.

The convergence speed of CCs can be further decomposed into *acceleration* and *deceleration* speeds. Acceleration speed is crucial for achieving high bandwidth utilization, thereby benefiting throughput-sensitive applications [33, 54, 75] with large flows. Meanwhile, deceleration speed is vital for maintaining short queues and low latency, which is critical

[1] Consistent with recent literature [7, 42, 56, 57, 61, 68, 88], unless explicitly stated, "convergence" refers to achieving target state (typically near-full bandwidth utilization or a shallow queue) rather than converging to fairness. Convergence to the target state means achieving high throughput and low latency. In this paper, we quantify convergence time by the number of CC update steps (AI, MI, or MD operations), since the time used by each step is on the order of one RTT for most CCs [8, 11, 55, 57, 61, 85, 88, 91].

Category	Representative CCs	Acceleration Speed	Deceleration Speed	Readily Deployable
AIMD-based CC	DCQCN [91], TIMELY [70], Swift [57]	$O(\Delta)$	$O(\log\Delta)$	✓
Short-INT-based CC	Poseidon [85]	$O(\log\Delta)$	$O(\log\Delta)$	✓✗
Precise-INT-based CC	HPCC [61], PowerTCP [7]	$O(1)$	$O(1)$	✗
Receiver-driven CC	NDP [43], Homa [72]	$O(1)$	$O(1)$	✓✗
Switch-driven CC	XCP [55], Bolt [14], Harmony [8]	$O(1)$	$O(1)$	✗
OSCAR, the first $O(1)$-step convergence CC via readily accessible signals		$O(1)$	$O(1)$	✓

Δ denotes the gap between the initial state and the target state.

Table 1: Comparison of OSCAR and state-of-the-art datacenter CCs.

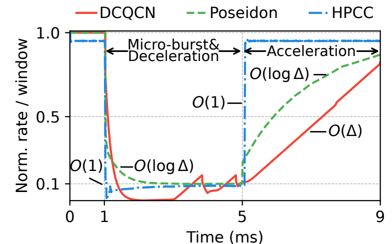


Figure 1: Convergence process of representative CCs in a typical microburst.

for latency-sensitive applications [29, 34, 76] with small flows. The ideal CCs should converge in $O(1)$ steps, *i.e.*, the convergence time is independent of the state gap Δ during both acceleration and deceleration. This characteristic ensures predictable performance across diverse network dynamics.

Table 1 classifies existing CCs by their convergence speed. To illustrate this, Figure 1 shows the convergence of a long-lived flow under three representative CCs during a typical microburst scenario induced by nine short-lived flows active from 1 ms to 5 ms (experiment setup is detailed in § 6.3). The y-axis plots the long-lived flow's sending rate (for DCQCN) or window (for Poseidon and HPCC), normalized by the link capacity (line rate or base BDP, *i.e.*, line rate × base RTT). Therefore, the long-lived flow's target state corresponds to 0.1 during the microburst and 1.0 otherwise.

AIMD-based CCs, such as DCQCN, exhibit an $O(\Delta)$ acceleration speed and an $O(\log\Delta)$ deceleration speed. During acceleration, they advance by a fixed step, making the number of steps to close the gap Δ proportional to Δ itself. During deceleration, the multiplicative deceleration step closes only a fraction of the gap per step, leading to logarithmic convergence (formally proven in Theorem D.1).

Short-INT-based CC, *i.e.*, Poseidon [85], employs a compact INT header (ideally 2 bytes in size) to carry per-hop delay and uses a control law that accelerates faster at a lower rate, resulting in an $O(\log\Delta)$ convergence speed during both acceleration and deceleration as shown in Figure 1.

Precise-INT-based CCs, such as HPCC and PowerTCP, can achieve $O(1)$-step convergence by MIMD reaction to precise queue length and output rate embedded by switches into a 42-byte INT header, which will be detailed in § 3.1.

Additionally, the following two categories of CCs can also achieve $O(1)$-step convergence. **Receiver-driven CCs** [20, 25, 35, 43, 48, 72] enable $O(1)$-step convergence for last-hop congestion, where the congestion information can be directly seen at the receiver. **Switch-driven CCs** [8, 14, 30, 55] utilize switches to manage network states and proactively instruct senders to adjust their rates for rapid, even sub-RTT convergence [14].

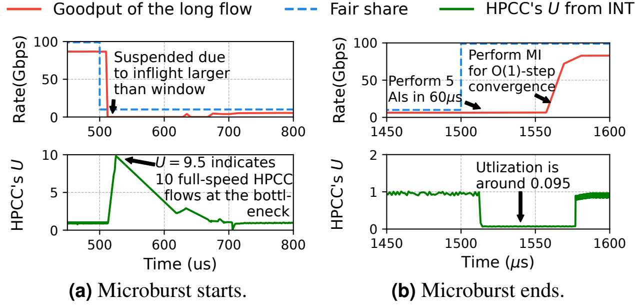
2.2 Current $O(1)$-step Convergence CCs Are Not Deployment-Friendly

While $O(1)$-step convergence CCs offer superior performance, they are not readily deployable due to their assumption on specialized hardware or network topologies.

The main challenge in deploying precise INT lies in adopting high-speed network hardware to handle extensive INT headers without compromising throughput [85]. There are two methods for integrating INT headers [2, 58]. (i) *INT padding* involves inserting empty INT headers for typical hop counts (*e.g.*, 5-hops [61]) at the sender. This approach necessitates a longer parser for the deeper header, impacting the forwarding rate of switches [64]. Moreover, during link failures, this approach struggles as packets navigate additional hops to reach their destination. (ii) *INT prepending* involves each switch prepending the INT header at a fixed offset, causing prior headers to shift backward in sequence. This method requires NICs to adopt a variable-length parser, impacting their throughput [12, 13]. During link failures, this method may cause the packet length to exceed the MTU [2, 85]. Besides, excessively lengthy INT headers will compromise the network goodput by reducing the proportion of payload in packets [17]. In contrast, the short INT used by Poseidon is easier to implement by avoiding these complications [79]. However, using short INT headers still requires upgrading switches, and cannot support $O(1)$-step convergence as shown by Poseidon [85].

Receiver-driven CCs typically rely on packet spraying to avoid congestion at the network core, necessitating well-provisioned and symmetric datacenter topologies. However, many modern data centers adopt oversubscription to scale the network topology [31, 75], while frequent link failures [39, 62, 69, 94, 95] make maintaining symmetric topologies challenging. Besides, packet spraying can cause out-of-order delivery. When applied to the RDMA network, this feature is *fully supported* only by the latest commodity RDMA NIC (ConnectX-8 [1]). In contrast, the support in previous-generation NICs is limited to a subset of verbs [83], and packet reordering can disrupt their retransmission algorithms [65, 83]. Additionally, some receiver-driven CCs require packet trimming [43], priority queues [72], or symmetric routing [25], which is not supported by all switches. Switch-driven CCs require computational capabilities on switches, which are not supported by commodity switches.

Although current $O(1)$-step convergence CCs demonstrate superior performance and may see broader adoption in the future, their reliance on advanced hardware features poses significant deployment challenges for data centers that cannot



(a) Microburst starts. **(b)** Microburst ends.

Figure 2: HPCC multiplicatively reacts to congestion information derived from INT to achieve $O(1)$-step convergence.

afford or upgrade to the latest hardware. Therefore, we pose the question: *Can readily deployable CC achieve $O(1)$-step convergence?*

3 Design Rationale

Given the prevalent hardware reliance of receiver-driven CCs (non-blocking topology) and switch-driven CCs (computational capabilities on switches), we choose to use a sender-driven approach to implement a readily deployable $O(1)$-step convergence CC. To realize this vision, we conduct in-depth analysis and experiments to identify the crux of the $O(1)$-step convergence of Precise-INT-based CCs (§ 3.1). We then explore readily accessible congestion signals that can deliver precise congestion information (§ 3.2). Based on the observations, we propose the idealized control law for achieving this objective (§ 3.3).

3.1 Multiplicative Reaction to Precise Signals is Critical for $O(1)$-step Convergence

By definition, $O(1)$-step convergence guarantees that a CC can reach its target state within a constant number of steps, regardless of the initial and target state gap Δ. This necessitates that at least one of these steps can eliminate the entire remaining state gap (*i.e.*, one-step convergence). Conversely, if a CC can eliminate only a portion of its current state gap each step, the convergence will be an iterative process and can be proven to require at least $O(\log \Delta)$ steps (Theorem D.1). Therefore, our discussion of the control law will primarily focus on how to achieve *one-step convergence*. Analysis of existing CCs reveals two key observations:

Observation 1.1: *Precise-INT-based CCs achieve one-step convergence via* ***multiplicatively*** *reacting (MIMD) to the* ***precise congestion signal***.

To analyze with fine-grained network information, we conduct micro-benchmark experiments on HPCC using ns-3 [3]. Our experiment setting is detailed in § 6.3. Within this setup, 1 of the 10 senders continuously transmits a long flow, while the other 9 senders transmit short flows between 500 µs and 1500 µs to mimic a typical microburst scenario. In Figure 2, the upper subfigures show the goodput of the long flow and the network's fair share, and the lower subfigures show the congestion information (bottleneck utilization U) HPCC obtains from INT. HPCC's target utilization U_{target} is set to 0.95[2], following the original paper [61].

Figure 2a shows the start of the microburst, where HPCC quickly reduces its window to the fair share after congestion occurs. After a brief suspension caused by the inflight exceeding the window, it starts sending at its fair share. This rapid convergence stems from two key mechanisms:
(i) *The congestion signal (U) precisely reflects the network state.* Upon congestion, U rises to 9.5, which is 10× of the U_{target}, precisely reflecting 10 full-speed competing flows.
(ii) *HPCC multiplicatively reacts to the signals.* Based on the congestion signal, each flow multiplicatively decreases the window to the $\frac{1}{10}$ of its current size to converge.

An analogous process occurs for acceleration. As shown in Figure 2b, when congestion ceases (at 1500 µs), HPCC's U drops to around 0.095, indicating that the bottleneck is operating at only 10% of the U_{target}. After 5 RTTs (HPCC performs MI only after observing underutilization for 5 consecutive RTTs), HPCC multiplicatively increases its window size by 10×, achieving the target utilization.

Observation 1.2: *AI-based CC can not achieve $O(1)$-step convergence*, **even with precise congestion signals**. HPCC's effectiveness stems from its MIMD response to precise congestion signals. This raises a natural question: can AIMD-based CCs also achieve O(1)-step convergence? We argue that this is fundamentally impossible. The limitation is inherent to the AI mechanism, where the aggregate rate increase scales with the number of flows n, a parameter that the CC can neither control nor perceive.

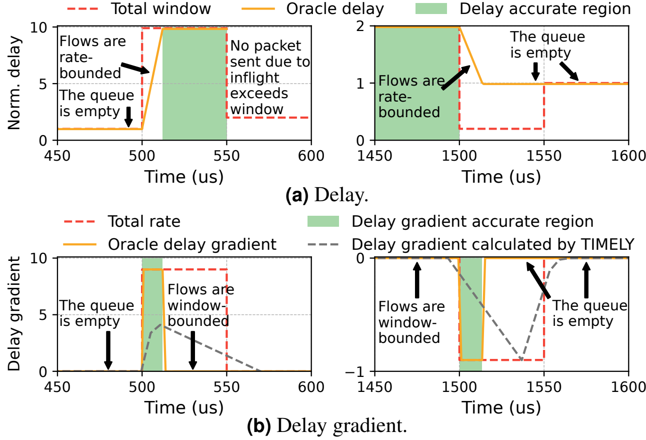
The AI of a flow f can be formulated as $r_f(t) = r_f(t-1) + \alpha$, where t and $t-1$ denote current and previous times, r_f denotes f's rate, and α denotes the AI step. For n flows sharing a bottleneck, the aggregate rate after an AI step is $\sum r_f(t) = \sum r_f(t-1) + n \cdot \alpha$. To eliminate the gap Δ in a single step, CC must set $\alpha = \frac{\Delta}{n}$, which is impractical as n is challenging to measure from congestion signals.

Conversely, an MI step can be formulated as $r_f(t) = r_f(t-1) \cdot \beta$, where β is the multiplicative factor. Thus, the aggregate arrival rate at the bottleneck is $\sum r_f(t) = \sum r_f(t-1) \cdot \beta$. To eliminate the gap $\Delta = R_{target} - \sum r_f(t-1)$ where R_{target} denotes the target aggregate rate, CC needs to set $\beta = \frac{R_{target}}{R_{target} - \Delta}$. This is feasible because R_{target} is already known and Δ can be measured through precise congestion signals.

Based on Observation 1.1 and 1.2, we conclude:

Principle 1: CC should multiplicatively react to precise congestion signals to achieve $O(1)$-step convergence.

[2] Due to the overhead from lengthy INT headers, HPCC's goodput is only 86% of the fair share, not the target 95%. Using 4 KB jumbo frames increases this to 92% but does not fully eliminate the bandwidth waste.



(a) Delay.

(b) Delay gradient.

Figure 3: Readily accessible congestion signals.

3.2 Delay and Delay Gradient can Provide Precise Congestion Information

$O(1)$-step convergence necessitates precise congestion information. Next, we explore whether readily accessible multi-bit congestion signals (*i.e.*, delay and delay gradient) can deliver such precision[3]. Through derivation and experimental verification, we obtain the following observations.

Observation 2.1: Delay precisely reflects the total window size *of flows passing through the bottleneck*[4] *under the constraint that the queue is neither empty nor full, and flows are bounded by window.*

Delay can be modeled as follows, where $d(t)$ denotes delay at time t, $d_q(t)$ denotes queueing delay at t, $q(t)$ denotes queue length at t, $i(t)$ denotes the inflight size of a flow pass through the queue, μ denotes packet departure rate of the queue, $w(t)$ and $r(t)$ denote flow's window and rate[5].

$$d(t) = RTT_{base} + d_q(t) = RTT_{base} + \frac{q(t)}{\mu} \quad (1)$$

$$= RTT_{base} + \frac{\sum i(t) - BDP_{base}}{\mu} = \frac{\sum i(t)}{\mu} \quad (2)$$

(When queue is neither empty nor full)

$$\leq \frac{\sum w(t)}{\mu} \quad \textit{(Equality holding when window-bounded)} \quad (3)$$

Equation 1 shows the direct relationship between delay and queue length. When the queue is neither empty nor full, the relationship between the queue length and the total inflight through the queue is described by Equation 2. When flows' transmissions are window-bounded, inflight packets' size matches the window size. Therefore, the delay can reflect the total window size of the flows passing through the bottleneck, *i.e.*, $d(t) = \sum w(t)/\mu$ (Equation 3), when the queue is not empty or full, and flows are window-bounded.

[3]While single-bit signals such as ECN can provide multi-bit congestion estimates through aggregation [11, 40], achieving the requisite precision remains a non-trivial challenge, which is discussed in Appendix A.2.

[4]Analysis in this section is based on the single-bottleneck scenario.

[5]We refer the reader to Table 3 for all the notations being used.

To verify the accuracy and the constraint of delay without the influence of CC's behavior, we implement *oracle CC* in ns-3, which controls both window and rate to achieve the target of one base BDP queue at bottleneck. It always converges to the target in one step with a reaction delay of 50 µs. The setting of the experiments is aligned with Figure 2.

Figure 3a shows the delay normalized by base RTT and total window normalized by base BDP at the start and end of the microburst. Upon the start of the microburst at 500 µs, the delay increases to $10 \times RTT_{base}$ in around 10 µs. During this period, the new flow has not yet fully utilized the window, *i.e.*, rate-bounded. Thus, the delay does not reflect the window. When stabilizing at $10 \times RTT_{base}$ (510 µs to 550 µs), the delay accurately reflects the total window of the flows as $10 \times BDP_{base}$. After the oracle CC decides to reduce the window, no packets are sent until the inflight is below the window. After the termination of short flows (1500 µs-1510 µs), the long flow converts from window-bound to rate-bounded due to delay reduced. Therefore, delay does not reflect the total window.

Observation 2.2: Delay gradient precisely reflects the total rate *of flows passing through the bottleneck under the constraint that the queue is neither empty nor full, and flows are rate-bounded.*

Denote delay gradient as $g(t)$ and packet arrival rate of the queue at t as $\lambda(t)$, the physical meaning of $g(t)$ can be derived as follows:

$$g(t) = \frac{\mathrm{d}d(t)}{\mathrm{d}t} = \frac{\mathrm{d}(RTT_{base} + d_q(t))}{\mathrm{d}t} = \frac{\mathrm{d}q(t)}{\mu \cdot \mathrm{d}t} \quad (4)$$

$$= \frac{\mathrm{d}\left(\int_{t-\delta t}^{t} (\lambda(t) - \mu)\,\mathrm{d}t + q(t - \delta t)\right)}{\mu \cdot \mathrm{d}t} = \frac{\lambda(t) - \mu}{\mu} \quad (5)$$

(When queue is neither empty nor full)

$$\leq \frac{\sum r(t)}{\mu} - 1 \quad \textit{(Equality holding when rate-bounded)} \quad (6)$$

Equation 4 is based on the definition of gradient and Equation 1. In Equation 5, δt denotes an infinitely small time period used in the analysis. This equation gives the relationship between arrival rate $\lambda(t)$ and $g(t)$ and is held when the queue is neither empty nor full from $t - \Delta t$ to t. When flows' transmissions are rate-bounded, the arrival rate equals the total send rate. In summary, the delay gradient can reflect the total send rate of the flows passing through the bottleneck, *i.e.*, $g(t) = \frac{\sum r(t)}{\mu} - 1$ (Equation 6), when the queue is not empty or full and flows are rate-bounded.

Figure 3b depicts the delay gradient and normalized total rate at the start and end of the microburst, where the oracle delay gradient is calculated by Sliding Window Least Squares (SWLS) [52]. At the start of congestion (500 µs-510 µs), the oracle delay gradient rises to 9, accurately reflecting the bottleneck arrival rate at 10μ. When the microburst ends (1500 µs-1510 µs), the oracle delay gradient decreases to -0.9, indicating that the bottleneck's arrival rate reduces to 0.1μ. However, once the network is window-bounded (510

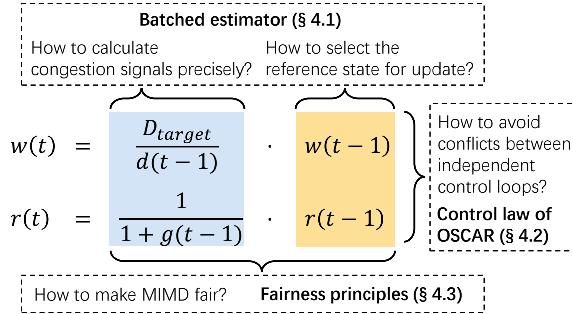


Figure 4: Core control mechanism of OSCAR.

µs-550 µs), the window's self-clocking equalizes the arrival and departure rate at the bottleneck. The delay gradient is 0 and no longer informative.

Observation 2.3: *Precisely calculating the delay gradient is challenging.*

Unlike delay, which can be directly measured, delay gradient must be derived through calculation. Figure 3b presents the delay gradient calculated by TIMELY, a state-of-the-art delay-gradient-based CC. Its gradient calculation is less accurate during the microburst and exhibits a lag in response to the microburst end. The limitations of TIMELY's gradient calculation will be further explored in § 4.1.1.

Based on Observation 2.1, 2.2 and 2.3, we conclude:

Principle 2: CC can leverage delay and delay gradient as precise congestion signals, while also needing to overcome their intrinsic limitations that they only serve as precise signals under specific constraints and are challenging to measure.

3.3 One-step Convergence Based on Delay and Delay Gradient

This subsection presents the idealized method for converging the total window and rate to the target state in a single step by multiplicatively reacting to delay and delay gradient. The mechanism for converging to fairness is discussed in § 4.3.

Assuming the target queue length is Q_{target}, which corresponds to target delay $D_{target} = RTT_{base} + Q_{target}/\mu$. Given that the delay can reflect the total window size as $d = \sum w/\mu$, each flow can *update its window* multiplicatively by D_{target}/d to make the delay converge to D_{target} in one step when delay is precise. The target total rate of flows passing through a bottleneck is typically set to the bottleneck's departure rate, *i.e.*, $\sum r = \mu$, to ensure full bandwidth utilization. The delay gradient reflects the total rate as $\sum r = \mu(1+g)$. Therefore, when delay gradient is precise, by *updating each flow's rate* multiplicatively by $1/(1+g)$, the total rate converges to μ.

However, as shown in § 3.2, delay and delay gradient are precise under mutually exclusive conditions, *i.e.*, whether flows are window-bounded or rate-bounded. Therefore, a CC should employ them complementarily to consistently achieve $O(1)$-step convergence.

4 Design

Based on the above principles, we propose **OSCAR, the first $\underline{O}(1)$-$\underline{S}$tep $\underline{C}$onvergence $\underline{A}$nd $\underline{R}$eadily-deployable CC.** The core of OSCAR is multiplicative reaction to delay and delay gradient, as depicted in Figure 4, which ideally enables convergence to the target state in one step, *i.e.*, achieving $O(1)$-step convergence. To achieve this vision, OSCAR addresses the following challenges:

• $O(1)$-step convergence needs precise congestion signals. However, as shown in § 3.2, accurately calculating delay gradient in high-speed networks is challenging. (§ 4.1.1).

• Unlike AIMD-based CCs, which update based on the current state by relatively small steps, MIMD-based CCs often apply a substantial multiplicative factor on the reference state, *i.e.*, $w(t-1)$ and $r(t-1)$ in Figure 4, for rapid convergence. Thus, minor deviations in reference state can lead to fluctuations or overreactions, making it essential for OSCAR to select correct reference states to update (§ 4.1.2).

• As shown in Figure 4, OSCAR employs two independent control loops to establish the control law: the <delay, window> loop aims at stabilizing the queue at the target length, and the <delay gradient, rate> loop aims at equalizing the bottleneck ingress rate with the line rate. Without proper coordination, these two loops may often conflict as they work for distinct targets (§ 4.2).

• MIMD is well-known for its inherent lack of fairness. OSCAR needs to ensure fairness while retaining MIMD as its core control mechanism(§ 4.3).

4.1 Batched Estimator for Network State

OSCAR employs a batched estimator to assess the network state, which includes congestion signals estimation (§ 4.1.1) and reference state calculation (§ 4.1.2). Algorithm 1 presents the pseudocode for OSCAR's batched estimator. Line numbers referenced in this subsection are specified to Algorithm 1.

4.1.1 Congestion Signals Estimation

RTT as delay measurement. OSCAR measures delay using the RTT of data packets, defined as the difference between the data packet's sending timestamp *sendTs* and the corresponding ACK's receiving timestamp *recvTs*. Given the limited memory space on high-speed NICs, OSCAR's sender embeds the *sendTs* in the packet header. The receiver then echoes *sendTs* back to the sender in the ACK. The sender calculates the RTT from *sendTs* in the ACK and the time ACK received. OSCAR uses 4 bytes for *sendTs* with the unit of nanoseconds.

Batched least square for gradient estimation. Although using delay gradient as the congestion signal has been introduced in existing studies [44,51,70], the accuracy of their delay gradient calculations does not meet the requirements of OSCAR for high-precision control in high-speed network.

A straightforward method for calculating the delay gradient is to divide the difference in delays by the difference in

Algorithm 1: OSCAR's Batched Estimator

Input: $sendTs, recvTs, pktInfl$
Result: $delay, gradient, inflight, rate$

```
1  Function Estimate(sendTs, recvTs, pktInfl)
2      pktDelay ← recvTs − sendTs
3      sumX += sendTs, sumY += pktDelay
4      sumXX += sendTs · sendTs
5      sumXY += sendTs · pktDelay
6      counts += 1, sumInfl += pktInfl
7      if sendTs − startTs ≥ τ && counts ≥ 3 then
8          delay ← sumY / counts
9          gradient ← (counts·sumXY − sumX·sumY) / (counts·sumXX − sumX·sumX)
10         inflight ← sumInfl / counts, rate ← counts·MTU / (sendTs − startTs)
11         sumX, sumY, sumXX, sumXY, counts, sumInfl ← 0
12         startTs ← sendTs
13         return True
14     return False
```

packet sending timestamps, followed by applying a noise-smoothing algorithm to reduce the error[6]. However, in modern datacenters with transmission rates typically reaching or exceeding 100 Gbps, the sending intervals between packets are extremely small (~10 ns). When performing division, the small denominator can significantly amplify minor errors (~µs) in delays. Moreover, EWMA (Exponentially Weighted Moving Average), the most prevalent noise-smoothing algorithm in CCs [11, 70, 74, 82], can diminish or delay the perception of congestion information, as demonstrated by the delayed gradient calculation of TIMELY in Figure 3b.

Instead, by treating time as the independent variable and delay as the dependent variable, OSCAR estimates delay gradient as the slope between time and delay through least squares. The most classic least squares solutions for estimating system dynamics are the recursive (RLS) [45] and the sliding-window least squares (SWLS) [26, 53]. However, RLS is numerically unstable [45, 63], while SWLS with a window size of n requires $O(n)$ storage space [87].

We propose the *Batched Least Squares (BLS)* for delay gradient estimation with constant time and space complexity, *i.e.*, the overhead is irrelevant to the rate or the number of packets. In deriving the least squares solution, the delay gradient calculation can be succinctly represented as:

$$g = \frac{n \cdot \sum(x_i \cdot y_i) - \sum x_i \cdot \sum y_i}{n \cdot \sum(x_i^2) - (\sum x_i)^2} \quad (7)$$

where $\sum$ is the abbreviation of $\sum_{i=1}^{n}$, with x_i representing the $sendTs$ of a packet and y_i the corresponding delay, and n the number of packet for a batched estimation. From the formula, we find that calculating the delay gradient only needs to store five cumulative values: $\sum x_i, \sum y_i, \sum x_i^2, \sum x_i y_i$ and n. Consequently, BLS does not store statistics upon reception but instead updates the algorithm's state accordingly (line 3-5). Then, it calculates the delay gradient in the period τ (line 7). The average delay is also calculated per τ (line 8).

[6] While there are differences in details, many CCs based on delay gradients such as CARD [51], CDG [44], and TIMELY [70] employ similar techniques.

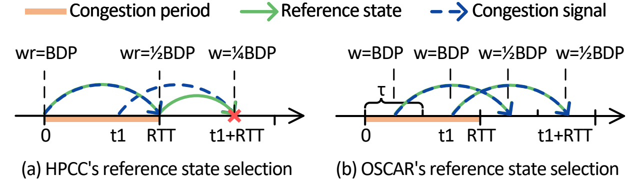


Figure 5: Illustration of OSCAR's reference state selection.

We set τ to half the base RTT to balance the responsiveness and noise robustness. To prevent the aforementioned small denominator issue, the calculation is postponed if fewer than three ACKs are received within τ (line 7).

OSCAR employs 4-byte timestamp in nanoseconds, leading to a wrap-around approximately every 4.3 seconds, which is much longer than typical lifespan of datacenter flows. This issue is resolved with only 1 bit of storage overhead and a maximum of τ estimating delay, as detailed in Appendix A.1.

4.1.2 Reference State Selection

In MIMD updating, the correct reference state selection is essential to avoid fluctuations and overreactions.

HPCC updates the reference window w_r once per RTT, attempting to avoid overreacting. Upon receiving each ACK, w_r is used as the reference state for updating the window. However, as depicted in the left half of Figure 5, this design can lead to overreactions. Consider a typical congestion scenario where two flows with a BDP window compete for a bottleneck. Congestion begins at time 0 and lasts for a RTT. One RTT after congestion occurs, HPCC receives the congestion signal from time 0 and updates w_r using the window size from that moment, reducing the window to $\frac{1}{2}BDP$. Then, at time $t_1 + RTT$, HPCC receives the congestion signal from the packet sent at t_1, which is at the same magnitude as time 0. HPCC then updates based on previous w_r, further reducing the window to $\frac{1}{4}BDP$, resulting in an overreaction. PowerTCP employs a similar reference state selection to that of HPCC, thereby encountering the same issues.

The overreaction of HPCC stems from using the congestion signal from time t_1 while referring to the state at time RTT. Based on this observation, OSCAR *uses the state synchronized with the congestion signal as the reference state to update*. As shown in the right half of Figure 5, at time $t_1 + RTT$, OSCAR state is updated through congestion signal from t_1 combined with state at t_1, thereby avoiding overreaction.

To implement the synchronization mechanism, OSCAR integrates the reference state estimation into the batch estimator. Recall that OSCAR employs both window and rate to control transmission, the actual inflight may deviate from the window size due to rate constraints, and the actual transmission rate may vary from the rate specified by CC due to window limitations. Therefore, OSCAR needs to estimate the actual inflight size and send rate rather than merely

Algorithm 2: OSCAR's Main Algorithm

```
   Input: sendTs, recvTs, pktInfl
   Result: w, r
1  Procedure NewAck(sendTs, recvTs, pktInfl)
2      estimated ← BE.Estimate(sendTs, recvTs, pktInfl)
3      if estimated == False then return
4      if BE.delay == RTT_base then
5          u ← u + U_HAI                          // Hyper increase
6      else
7          u_w = BE.inflight / (BE.delay·μ)       // (Delay, window) loop
8          u_r = BE.rate / ((1+BE.gradient)·μ)    // (Gradient, rate) loop
9          if BE.delay < D_target then u = max(u_w, u_r)
10         else u = min(u_w, u_r)                 // Coordinate two loops
11         u = u + U_AI   // Consistent AI term for fairness
12     w = max(u · D_target · μ, BDP_base)
13     r = u · μ
```

recording the window and rate specified by CC. During transmission, OSCAR's sender includes the current inflight $pktInfl$ in the header, which is then echoed back in the ACK by the receiver. Upon receiving an ACK, the batched estimator accumulates $pktInfl$ into the $sumInfl$ (line 6). The batch estimator calculates actual inflight as $\frac{sumInfl}{counts}$ and the send rate as $\frac{counts \cdot MTU}{sendTs - startTs}$, where MTU denotes Maximum Transmission Unit and $sendTs - startTs$ is the length of the estimation period (line 10). Ablation studies confirm that HPCC's reference state selection leads to overreactions, while OSCAR's eliminates them (§ 6.3). OSCAR can work with ACK coalescing, as detailed in Appendix A.1.

4.2 Control Law of OSCAR

OSCAR employs two independent control loops to establish its control law. To coordinate the control loops, we first identify how to coordinate the window and rate to ensure they can converge to the same target. During the coordination of window and rate, OSCAR determines which control loop to adopt based on the relationship between the current and the target network state. The pseudocode of the control law is presented in Algorithm 2. Upon receiving a new ACK, OSCAR feeds the packet's information to the batched estimator (BE at line 2 in Algorithm 2). Following each estimation, OSCAR executes congestion control actions.

Unified window ratio and rate ratio. When both window and rate are used, the target state is essentially a dual-state: the target total window and the target total rate. Analysis of this dual-state yields the following theorem. Due to space constraints, the proof is placed in Appendix D.2.

Theorem 4.1. *When both window and rate are used simultaneously in a CC, convergence is feasible only when the window and rate are at the same ratio relative to the target window and target rate, respectively.*

Based on this theorem, we define the unified window ratio as the ratio of a flow's window to the target total window as $u_w = \frac{w}{D_{target} \cdot \mu}$ and unified rate ratio similarly as $u_r = \frac{r}{\mu}$. OSCAR employs the unified window ratio and unified rate ratio to coordinate the two control loops. During updating, OSCAR first calculates the ratio u_w and u_r through two control loops (line 7, 8), then coordinates the ratios into the unified ratio u (line 9-10), with the method of coordination to be introduced later. Finally, OSCAR converts the unified ratio to window and rate as $w = u \cdot D_{target} \cdot \mu$ and $r = u \cdot \mu$, thereby ensuring that window and rate are at the same ratio relative to the target, respectively (line 12, 13). Note that OSCAR does not set the window above the base BDP, as doing so would not increase throughput but only increase delay (line 12).

Coordinate the two control loops. We derive the coordination approach based on the following two observations.

(i) It is impossible to always satisfy the target of both loops simultaneously. For example, if the queue stabilizes above the target while the ingress and line rates are equalized, the <delay, window> loop suggests decelerating, whereas the <delay gradient, rate> loop suggests maintaining the current rate. To make proper decisions during conflicts, OSCAR prioritizes the target of <delay, window> loop, *i.e.*, stabilizing the queue length at the target queue length. This is because setting the primary target as equalizing the bottleneck's ingress and line rates can lead to arbitrary (including excessively high) convergence queue length [7,92], which is undesirable. Consequently, OSCAR accelerates when the observed delay falls below the target (line 9), and vice versa (line 10).

(ii) Inaccuracies in delay and delay gradient only reduce the signal magnitude when the bottleneck queue is not empty or full. As shown in § 3.2, when flows are not window-bounded, the magnitude of the delay is less than $\frac{\sum w(t)}{\mu}$. Similarly, when flows are not rate-bounded, the magnitude of the delay gradient is less than $\frac{\sum r(t)}{\mu} - 1$. Thus, control based on imprecise delay and delay gradient will only result in insufficient updates rather than overreactions or adverse reactions. Therefore, OSCAR adopts the larger result of the two loops when accelerating (line 9), and vice versa (line 10).

Control law when facing empty or full queue. An empty queue indicates potential underutilization, and both delay and delay gradient are imprecise in such a scenario. OSCAR employs a hyper increase strategy to boost the window and rate, thereby creating a queue in this case. After each batched estimation, OSCAR verifies whether the estimated delay equals RTT_{base} (in practice we compare the estimated delay with $RTT_{base} + \varepsilon$, where ε accounts for delay measurement inaccuracy). If so, the utilization ratio is increased by U_{HAI}. It is worth noting that hyper increase is seldom triggered during a OSCAR flow's lifecycle, as OSCAR typically converges rapidly to the target state. This will be further demonstrated in Appendix C.3. OSCAR does not need specialized handling for full queue scenarios. Because a full queue will cause a high delay, promoting OSCAR significantly decelerates to reduce the queue length. How OSCAR handles full queue scenarios (packet loss or PFC) gracefully is detailed in Appendix A.1.

OSCAR starts at line rate ($u = 1$), like most datacenter CCs.

4.3 Fairness of OSCAR

Fairness principles of MIMD-based CCs. Rather than merely demonstrating the fairness of OSCAR, we prove two principles, with adhering to, can ensure the fairness of MIMD-based CC: (i) The MIMD operation is linear with respect to its own state, (ii) An AI operation is executed subsequent to each MIMD operation. The intuition is that MI is primarily used for convergence. After converging, subsequent AI slightly overutilizes the bandwidth, triggering MD to re-establish convergence. This process will repeat as CC always performs AI following MD. Thus, after convergence, CC enters the AIMD cycle which ensures fairness. A similar method is first utilized in ABC [40], while we provide the formalization and theoretical proof of its underlying principles.

Due to space constraints, a graphical illustration and a formal algebraic proof for these principles are placed in Appendix D.3. The difference between these principles and existing similar fairness analyses is discussed in Appendix B.

OSCAR's approach to fairness. By adding a constant AI term U_{AI} after each MIMD operation (line 11 in Algorithm 2), OSCAR ensures convergence to fairness. As OSCAR utilizes MIMD to converge to the target, the subsequent AI will cause the actual convergence point to be slightly higher than the target, which is discussed in Appendix A.2.

4.4 Design Discussion

Advantages over Precise-INT-based CCs. OSCAR outperforms current precise-INT-based CCs in three aspects. (i) OSCAR eliminates bandwidth occupation of lengthy INT headers. (ii) OSCAR prevents oscillations caused by overreactions in MIMD control. (iii) HPCC conservatively performs MI once every 5 RTTs. OSCAR's absence of overreactions and assurance of fairness allows it to perform MI more frequently, improving responsiveness to underutilization.

Advantages over θ-PowerTCP. θ-PowerTCP also uses delay and delay gradient, but only achieve $O(\Delta)$-step convergence (Appendix C.5). The core flaw of θ-PowerTCP lies in its design premise that delay signals can not reflect network state precisely during under-utilization, restricting it to using only AI in such scenarios. Contrary to this premise, our analysis shows *the precise reflection is feasible* (§ 3.2). OSCAR fully exploits this precision, achieving $O(1)$-step convergence.

OSCAR is compatible with per-packet load balancing (LB). Per-packet LB [21, 38] is becoming an increasingly prevalent technique in LLM training to prevent flow collisions [62]. However, existing sender-driven CCs struggle with per-packet LB. Conventional CCs like DCQCN are over-sensitive to single-path congestion signals, which could lead to underutilization [62]. Meanwhile, precise-INT-based CCs are fundamentally incompatible, as their *txRate* calculation relies on consecutive INTs from the same port—a prerequisite that per-packet LB violates. OSCAR overcomes these issues with its batched estimator, which aggregates congestion information over a time period. This allows OSCAR to avoid overreacting to congestion on a single path and maintain rapid O(1)-step convergence even during events like link failures (see Appendix A.3 for details).

Parameters. OSCAR has only four parameters, each with a clear purpose, facilitating straightforward parameter tuning. (i) D_{target} is the target delay. A higher D_{target} benefits throughput-sensitive traffic, while a lower D_{target} benefits latency-sensitive traffic. We recommend setting it to $1.5 \times RTT_{base}$ as a balance. (ii) τ serves as both the period for the batch estimator and the update interval for OSCAR. We set τ to half the base RTT to balance the responsiveness and noise robustness. (iii) U_{HAI} is set to 0.01 according to parameter sensitivity experiments. (iv) U_{AI} is used to ensure fairness among flows. We recommend setting it to 0.001. The rationale for D_{target} and U_{AI} setting and an analysis of tuning complexity is detailed in Appendix A.2. The parameter sensitivity experiments are present in Appendix C.3.

Due to space constraints, the impact of reverse congestion and discussion of RTT-fairness are placed in Appendix A.2.

5 Implementation

This section introduces our implementation of OSCAR in DPDK and analyzes its overhead. We also discuss the implementation of OSCAR on RDMA network cards (RNIC). The testbed experiments are detailed in § 6.2.

DPDK implementation. We implement OSCAR and other CCs for comparison in Linux 5.4.0 with DPDK [32] 25.03 and GCC 8.4 at the highest optimization level. Since the NIC does not support hardware timestamps, we utilized CPU's Time Stamp Counter to record the timestamps for RTT calculation.

Algorithm	OSCAR (w/o BE)	OSCAR (w/ BE)	DCQCN	Swift	HPCC	PowerTCP
Overhead	34	52	46	104	92	159

Table 2: Calculation overhead (in CPU cycles) in DPDK.

Calculation overhead. We evaluate the calculation overhead of each CC by executing it one million times and averaging the results. As we compile at the highest optimization level, all available CPU instruction optimizations are enabled. Table 2 displays the overhead in CPU cycles. OSCAR requires only addition, product and comparison operations upon each ACK. The batched estimator and rate update are triggered every τ, which includes six division operations. Without batch estimating (OSCAR w/o BE), it incurs a time overhead of 34 cycles, whereas with batch estimating activated (OSCAR w/ BE), it requires 52 cycles. The overhead of DCQCN averages 46 cycles per CNP or timer trigger, which is comparable with OSCAR w/ BE. In contrast, HPCC needs $3+3p$ division operations per ACK, where p is the number of hops. PowerTCP needs $3+3p$ division operations and $3+3p$ product operations per ACK. Setting p at 5, the typical data center hop count, the overhead is 92 and 159 cycles for HPCC and PowerTCP, respectively. In summary, *the calculation overhead of* OSCAR *is lower than that of most data center*

CCs, particularly precise-INT-based CC.

Discussion on RNIC implementation. The main challenge of implementing CC on RNIC lies in the memory space and timers [61]. OSCAR incurs 7 extra variables. By using 8B to store $\sum x_i^2$ and $\sum x_i y_i$ and 4B for other variables, OSCAR needs an extra 36B of memory space. This is acceptable compared to the overhead introduced by the existing RNIC implementation of DCQCN (~60B) [66]. Additionally, OSCAR does not require extra timers, as the batched estimator is triggered by timestamp comparison.

6 Evaluations

We comprehensively evaluate OSCAR through the testbed, micro-benchmark simulations, and large-scale simulations. Simulations are based on the ns-3 [3]. In the testbed, we aim to evaluate OSCAR's performance under actual hardware and software fluctuations and OSCAR's enhancements under distributed model training (*training* for short) workload. Microbenchmark simulations involve ablation studies to validate OSCAR's design choices. In the large-scale simulations, we examine OSCAR's benefits across diverse challenging scenarios involving microbursts, and per-packet LB.

6.1 Setup

Comparisons and parameters. Our primary comparisons are made against HPCC [61] and PowerTCP [7], two precise-INT-based CCs. Additionally, we compare various representative CCs, including θ-PowerTCP (the delay version of PowerTCP), short-INT-based Poseidon [85], ECN-based DCQCN [91], delay-based Swift [57], delay-gradient-based TIMELY [70], and NDP [43], the representative receiver-driven CC. The parameter settings for all these CCs are consistent with the recommendations in the original papers. OSCAR's parameters align with § 4.4. To fairly compare the performance differences caused by the convergence speeds of OSCAR and HPCC, we also evaluate HPCC with the same target as OSCAR, denoted as HPCC*.

Testbed setup. Our testbed consists of seven servers, each of which is equipped with an Intel Xeon E5-2650 CPU @2.20GHz, Mellanox ConnectX-5 or ConnectX-6 NICs, and 256GB RAM. Servers are interconnected via a Mellanox SN2700 switch. The links are configured at 40 Gbps, with the 13.5 µs base RTT. As our switch does not support INT, we are *unable* to compare HPCC and PowerTCP in the testbed.

Simulation workloads. In large-scale simulations, we focus on three scenarios: long-tail, microburst, and training. (i) In the long-tail scenario, we generate traffic using Cache follower workloads [80]. (ii) In the microburst scenario, we generate incast flows consisting of 32 randomly chosen senders, each sending 450KB, alongside background flows according to the Hadoop workload [80], which mainly consists of small flows. (iii) In the training scenario, we generate all-reduce traffic based on the ring algorithm, which is used in data parallelism model training, and all-to-all traffic for Mixture of Experts (MOE) training. The all-reduce traffic is generated according to ResNet [47] and VGG [5] by Astrasim [78]. The all-to-all traffic is generated based on the expert selection distribution and flow size as reported by [46].

Simulation topology. In the large-scale simulations of long-tail and microburst workload, a fat-tree topology [10] with 320 hosts is adopted. The fat-tree topology consists of five pods interconnected by 16 core switches, each pod equipped with 4 aggregation switches, 4 edge switches, and 16 servers. We configure host-to-edge switch links at 100 Gbps and inter-switch links at 400 Gbps to establish a non-blocking topology. In training scenarios, we employ a two-tier topology consisting of 2 spine switches and 16 leaf switches, each connected to 4 hosts, with all links being 200 Gbps, following the prevalent practice of managing most training traffic within two tiers [33, 75, 84]. Switch buffers are set at 32MB, consistent with typical settings [4], and utilize shared configurations with Priority Flow Control (PFC) and dynamic threshold algorithms [27] enabled. Unless specified, we use per-flow ECMP as the default. We also employ packet-spraying as a typical per-packet LB.

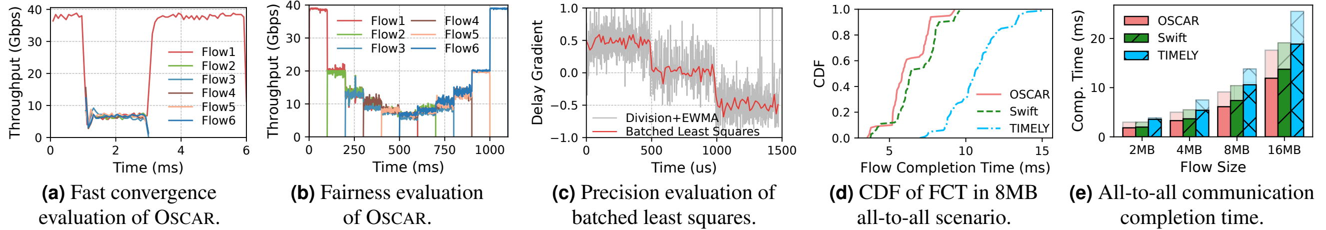
6.2 Testbed Evaluation

OSCAR can converge to the target state rapidly. We replicate the experiment shown in Figure 2 within the testbed with one receiver and six senders. One sender transmits a long flow, while the other five issue short flows starting at 1 ms and lasting 2 ms. As depicted in Figure 6a, the results demonstrate that OSCAR traffic quickly converged to a fair share upon the start and end of short flows.

OSCAR can converge to a fair allocation. As shown in Figure 6b where six senders initiate flows at 100 ms intervals and lasting 600 ms, OSCAR flows rapidly converge to fairness upon flow arrival and departure.

OSCAR's batched estimator precisely and robustly estimates delay gradient. To assess the accuracy of the batched estimator (§ 4.1), we conduct experiments with two senders sending two flows. One flow consistently sends at 20 Gbps, while another sends at 40 Gbps and 20 Gbps each for around 500 µs. Ideally, the first flow's observed delay gradient should be 0.5, 0, and -0.5, each lasting 500 µs. Figure 6c shows that the delay gradients calculated by OSCAR accurately reflect the network, and the Mean Squared Error (MSE) is only 0.0058. In contrast, delay gradients calculated via division and smoothed using an EWMA (weight = 0.2) show considerable jitter due to the small denominator issue (§ 4.1.1), with an MSE of 0.079, 13.4× worse than OSCAR.

OSCAR improves all-to-all operations. We generate all-to-all traffic among six hosts, with flow sizes varying from 2 MB to 16 MB, and each is tested 100 times. The destination is randomly selected using the expert selection distribution in [46]. Figure 6d shows the FCT CDF for 8 MB all-to-all. Benefiting from the quick and fair convergence, OSCAR improves both average and tail FCT compared to



(a) Fast convergence evaluation of OSCAR. **(b)** Fairness evaluation of OSCAR. **(c)** Precision evaluation of batched least squares. **(d)** CDF of FCT in 8MB all-to-all scenario. **(e)** All-to-all communication completion time.

Figure 6: Testbed evaluations.

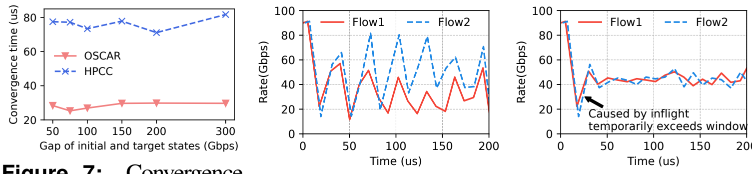
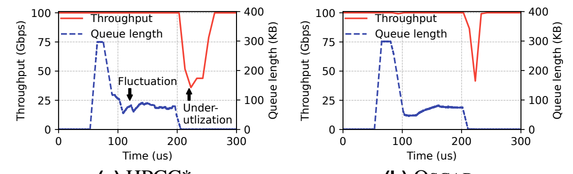


Figure 7: Convergence time of OSCAR and HPCC.

(a) w/o reference selection. **(b)** w/ reference selection.

Figure 8: Ablation study of reference selection.



(a) HPCC*. **(b)** OSCAR.

Figure 9: Microscopic view on congestion reation.

Swift and TIMELY. Figure 6e shows the average and 99th percentile (99th tail) completion times for 2MB to 16MB all-to-all operations. OSCAR outperforms Swift and TIMELY by up to 19.6% and 93.9% in average completion times, and by 14.8% and 51.4% in tail completion times, respectively.

Due to space constraints, additional testbed evaluations are placed in Appendix C.1. These evaluations reveal that Swift and TIMELY can not converge rapidly and fairly in the same environment, while a µs-level microscopic view evaluation of OSCAR confirms it converges in $O(1)$ steps.

6.3 Micro Benchmarks

Our micro-benchmark simulates microburst scenarios in a Clos topology with multiple senders, 1 receiver, and 1 bottleneck port. The link bandwidth is set to 100 Gbps, and the network's round-trip time (RTT) is 12 µs to align with typical data center networks.

OSCAR can achieve $O(1)$-step convergence. Figure 7 shows the convergence time of the long flow after the microburst ends. We vary the gap Δ by changing the link bandwidth and the number of short flows. The convergence time of both OSCAR and HPCC is independent of Δ ($O(1)$-step convergence). However, OSCAR converges within around 25 µs , while HPCC takes around 80 µs as discussed in § 4.4.

OSCAR's reference state selection can avoid fluctuations. We validate the effectiveness of OSCAR's reference state selection on HPCC without its congestion signal smoothing mechanism[7] and observing the convergence of two HPCC flows competing on a 100 Gbps port. As shown in Figure 7a, with HPCC's origin reference state selection, the two flows fail to converge and fluctuate continuously. By enhancing HPCC with OSCAR's reference state selection , HPCC flows converge to fair share stably, as depicted in Figure 7b.

OSCAR achieves stable convergence and minimal underutilization. Figure 9 shows the queue length and bottleneck bandwidth utilization. In the experiment, a long flow runs on the bottleneck, and at 50 µs, two short flows, each sized at 4 base BDP, emerge. As shown in Figure 8a, HPCC* can converge the queue length to around 75 KB ($0.5\times$ base BDP) but fluctuations in the queue length do occur. After the microburst ends, HPCC* takes about 60 microseconds to converge, leading to underutilization. In contrast, OSCAR accurately and smoothly converges the queue to 75 KB and minimizes underutilization.

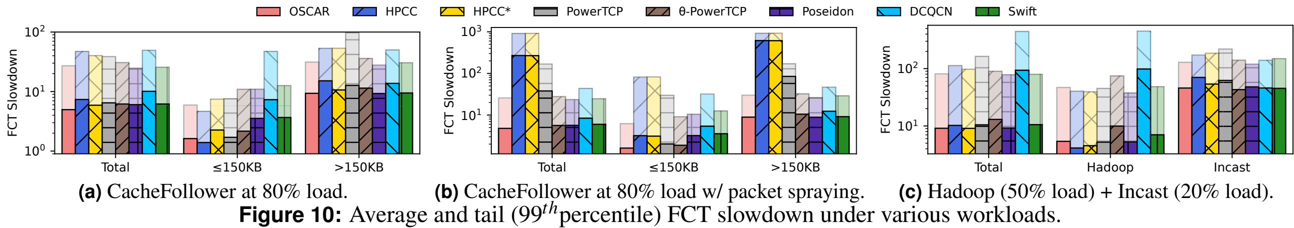
[7] HPCC's congestion signal smoothing moderates the intensity of congestion signals, masking fluctuations caused by overreactions.

6.4 Large-scale Simulations

In long-tail workloads (CacheFollower), we categorize flows into small ($\leq$150KB, *i.e.*, 1 base BDP), and larger (>150KB) groups. In microburst scenarios, Hadoop traffic and incast traffic are analyzed separately. *Total* in the figure represents the *average FCT slowdown for all flows*.

OSCAR delivers comparable performance for small flows and superior performance for larger flows compared to precis-INT-based CCs. Overall, OSCAR outperform precis-INT-based CCs. Figure 10a shows the FCT slowdown under the CacheFollower workload at 80% load. For small flows, OSCAR's slowdown is 7.6% worse than HPCC, while 32.9% and 4.9% better than HPCC* and PowerTCP, respectively. For larger flows, OSCAR outperforms HPCC, HPCC*, and PowerTCP by 62.2%, 13.9%, and 35.7%, respectively. Overall, OSCAR outperforms HPCC, HPCC*, and PowerTCP by 48.1%, 18.6%, 30.0% on average, and 74.1%, 48.4%, 43.0% at tail. OSCAR also outperforms θ-PowerTCP and Swift by 21.3% and 24.4% on average. Figure 11 provides a detailed breakdown of FCT slowdown. OSCAR's small flow performance is comparable to HPCC, and its larger flow performance is better than HPCC*. Across all flow sizes, OSCAR outperforms HPCC* by 10%-42% on average.

OSCAR is compatible with per-packet LB. Figure 10b shows the results with packet spraying. As detailed in Appendix A.3, precise-INT-based CCs are not effective because of erroneous congestion information calculations. Among all other CCs, OSCAR shows a distinct advantage, outperforming PowerTCP,



(a) CacheFollower at 80% load. **(b)** CacheFollower at 80% load w/ packet spraying. **(c)** Hadoop (50% load) + Incast (20% load).

Figure 10: Average and tail (99^{th}percentile) FCT slowdown under various workloads.

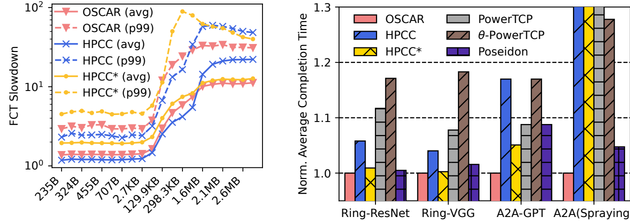


Figure 11: Breakdown of FCT slowdown in CacheFollower at 80% load.

Figure 12: Normalized average completion time of collective communications.

Poseidon, DCQCN, and Swift by 17.5%, 18.4%, 76.0%, and 25.2% on average, respectively.

OSCAR effectively alleviates congestion from incast without penalizing incast traffic. Figure 10c shows the results under a mix of Hadoop traffic at 50% load and incast at 20% load. For incast traffic, OSCAR outperforms HPCC by 53.0%, demonstrating its ability to handle severe congestion without overreaction. For Hadoop traffic, OSCAR is inferior to HPCC, HPCC* and PowerTCP by 23.3%, 15.9% and 3.1% yet outperforms all other algorithms, demonstrating that OSCAR effectively mitigates incast. Considering all flows, OSCAR outperforms HPCC by 12.1% on average, and by 39.9% at tail. DCQCN fails to effectively control incast traffic due to the lack of a window mechanism. With a fixed window, it can effectively control incast as shown in Appendix C.6.

OSCAR effectively enhances model training performance over state-of-the-art CCs. We evaluate model training scenarios in a spine-leaf topology with 64 hosts, divided into four groups performing identical collective operations. Figure 12 shows the collective operations' average completion times normalized by OSCAR's completion times. For all-reduce (ring) operations (ResNet/VGG), OSCAR is faster than HPCC by 5.8%/4.0% and PowerTCP by 11.7%/7.8%. For all-to-all operations, OSCAR surpasses HPCC and PowerTCP by 17.0% and 8.8%. Furthermore, unlike precise-INT-based CCs that fail under packet spraying, OSCAR gains 11.2% improvement from it for all-to-all operations. Across all collective operations, OSCAR achieves a speedup of up to 8.8% and 27.8% over Poseidon and θ-PowerTCP.

OSCAR maintains its performance advantages in various network environments, including lossy, oversubscribed, large-scale incast, and non-symmetric topology. Due to space limitations, other evaluations are detailed in Appendix C. Here we highlight key findings: (i) In *lossy networks* with microbursts, OSCAR's performance is comparable with lossless networks and surpasses HPCC by 4%. (ii) In *oversubscribed networks*, we evaluate NDP [43], the state-of-the-art receiver-driven CC. With a 2:1 oversubscription, OSCAR with packet spraying outperforms NDP by 15% for small flows and underperforms by 11% for large flows. (iii) We also evaluate BFC, the state-of-the-art flow control algorithm (FC), demonstrating that OSCAR and FCs can cooperate to achieve better performance.

7 Discussion

Limitations. OSCAR cannot achieve max-min fairness in multi-hop congestion like INT-based CCs [7,61,85] due to its adoption of delay and delay gradient. These signals cannot distinguish whether increases in signals arise from single or multiple congestion points. This is detailed in Appendix A.2.

Related work. Most datacenter CCs are discussed in § 2. OSCAR's advantage over θ-PowerTCP is discussed in § 4.4. A comprehensive discussion of other related works [15, 23, 24, 55, 68, 85, 88] is provided in Appendix B.

Future Work. The key insight and the techniques developed to realize OSCAR can be extended to other domains. For instance, by leveraging our finding that delay and delay gradient can reflect network state at the precision comparable to INT, the batched estimator (§ 4.1) can serve as a low-overhead, readily deployable network monitoring tool alternative to INT-based ones [17, 50, 93] for data centers lacking INT support.

8 Conclusion

We propose OSCAR, the first readily deployable CC achieving $O(1)$-step convergence. OSCAR relies solely on end-to-end congestion signals without reliance on specific network features. Overhead analysis shows that it incurs lower overhead than most of datacenter CCs. Testbed experiments and large-scale simulations show that OSCAR can precisely and rapidly converge to a fair target state under real-world disturbances, and outperform state-of-the-art INT-based CCs.

Acknowledgments

We sincerely thank our shepherd Ahmed Saeed and the anonymous reviewers for their valuable feedback on this paper. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, 62372296, 62441236 and U25B2035, the Key Program of Natural Science Foundation of Jiangsu under grant No. BK20243053, in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant JYB2025XDXM103.

References

[1] Connectx-8 supernic. https://nvdam.widen.net/s/pxsjzhgw6j/connectx-datasheet-connectx-8-supernic-3231505.

[2] In-band network telemetry (int) dataplane specification. https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[3] Ns3 network simulator. https://www.nsnam.org/.

[4] Tomahawk3 / BCM56980 Series.

[5] Vgg16 architecture. https://iq.opengenus.org/vgg16.

[6] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, 2008.

[7] Vamsi Addanki, Oliver Michel, and Stefan Schmid. {PowerTCP}: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51–70, 2022.

[8] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. Harmony: A congestion-free datacenter architecture. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 329–343, 2024.

[9] Anurag Agrawal and Changhoon Kim. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32. IEEE Computer Society, 2020.

[10] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, August 2008.

[11] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[12] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in {High-Speed}{NICs}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, 2020.

[13] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. Nanotransport: A low-latency, programmable transport layer for nics. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 13–26, 2021.

[14] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt:{Sub-RTT} congestion control for {Ultra-Low} latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 219–236, 2023.

[15] Venkat Arun and Hari Balakrishnan. Copa: Practical {Delay-Based} congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, 2018.

[16] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[17] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. 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*, pages 662–680, 2020.

[18] Tommaso Bonato, Sepehr Abdous, Abdul Kabbani, Ahmad Ghalayini, Nadeen Gebara, Terry Lam, Anup Agarwal, Tiancheng Chen, Zhuolong Yu, Konstantin Taranov, Mahmoud Elhaddad, Daniele De Sensi, Soudeh Ghorbani, and Torsten Hoefler. Uno: A one-stop solution for inter- and intra-datacenter congestion control and reliable connectivity, 2025.

[19] Tommaso Bonato, Abdul Kabbani, Daniele De Sensi, Rong Pan, Yanfang Le, Costin Raiciu, Mark Handley, Timo Schneider, Nils Blach, Ahmad Ghalayini, et al. Smartt-reps: Sender-based marked rapidly-adapting trimmed & timed transport with recycled entropies. *arXiv preprint arXiv:2404.01630*, 2024.

[20] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcpim: Near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 53–65, 2022.

[21] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 49–60, 2013.

[22] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

[23] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Priyaranjan Jha, Yousuk Seung, Kevin Yang, Ian Swett, Victor Vasiliev, Bin Wu, Luke Hsiao, et al. Bbrv2: A model-based congestion control performance optimization. In *Proc. IETF 106th Meeting*, pages 1–32, 2019.

[24] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[25] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 239–252, 2017.

[26] B.-Y. Choi and Z. Bien. Sliding-windowed weighted recursive least-squares method for parameter estimation. *Electronics Letters*, 25(20):1381–1382, September 1989.

[27] A.K. Choudhury and E.L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking*, 6(2):130–140, April 1998. Conference Name: IEEE/ACM Transactions on Networking.

[28] Cisco. Cisco nexus 9400 series switches data sheet. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/nexus9400-series-switches-ds.html, 2024.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 401–414, USA, 2014. USENIX Association.

[30] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[31] Enfabrica. Scaling to 100k+ gpu ai clusters using flat 2-tier network designs. Enfabrica Technical Blog, August 2024. Published on August 7, 2024.

[32] Linux Foundation. Data plane development kit (DPDK), 2015.

[33] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 57–70, 2024.

[34] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 249–264, 2016.

[35] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015.

[36] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {rdma}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[37] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, April 2018. USENIX Association.

[38] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 225–238, 2017.

[39] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361, 2011.

[40] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. {ABC}: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, 2020.

[41] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. Backpressure flow control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, 2022.

[42] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[43] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 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*, pages 29–42, 2017.

[44] David A Hayes and Grenville Armitage. Revisiting tcp congestion control using delay gradients. In *International Conference on Research in Networking*, pages 328–341. Springer, 2011.

[45] Simon S. Haykin. *Adaptive Filter Theory*. Pearson, 2014.

[46] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 120–134, 2022.

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[48] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. 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*, pages 422–434, 2020.

[49] Huawei. Cloudengine 16800 data center switch datasheet. https://e.huawei.com/eu/material/networking/dcswitch/54056f0999bd4200b7bf56d2b1bfa5a5, 2024.

[50] Jonghwan Hyun, Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Real-time and fine-grained network monitoring using in-band network telemetry. *International Journal of Network Management*, 29(6):e2080, 2019.

[51] Raj Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM SIGCOMM Computer Communication Review*, 19(5):56–71, 1989.

[52] Jin Jiang and Youmin Zhang. A novel variable-length sliding window blockwise least-squares algorithm for on-line estimation of time-varying parameters. *International journal of adaptive control and signal processing*, 18(6):505–521, 2004.

[53] Jin Jiang and Youmin Zhang. A revisit to block and recursive least squares for parameter estimation. *Computers & Electrical Engineering*, 30(5):403–416, 2004.

[54] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[55] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[56] Vishnu Konda and Jasleen Kaur. Rapid: Shrinking the congestion-control timescale. In *IEEE INFOCOM 2009*, pages 1–9. IEEE, 2009.

[57] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, 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*, pages 514–528, 2020.

[58] Jai Kumar, Surendra Anubolu, John Lemon, Rajeev Manur, Hugh Holbrook, Anoop Ghanwani, Dezhong Cai, Heidi Ou, Yizhou Li, and Xiaojun Wang. Inband Flow Analyzer. Internet-Draft draft-kumar-ippm-ifa-08, Internet Engineering Task Force, April 2024. Work in Progress.

[59] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Dx: Latency-based congestion control for datacenters. *IEEE/ACM Transactions on Networking*, 25(1):335–348, 2016.

[60] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186. USENIX Association, November 2020.

[61] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[62] Yuxuan Li, Zhenghang Ren, Wenxue Li, Xiangzhou Liu, and Kai Chen. Congestion control for ai workloads with message-level signaling. In *Proceedings of the 9th Asia-Pacific Workshop on Networking*, APNET '25, page 59–65, New York, NY, USA, 2025. Association for Computing Machinery.

[63] A.P. Liavas and P.A. Regalia. On the numerical stability and accuracy of the conventional recursive least squares algorithm. *IEEE Transactions on Signal Processing*, 47(1):88–96, January 1999. Conference Name: IEEE Transactions on Signal Processing.

[64] Huan Liu, Zhiliang Qiu, Weitao Pan, Jiajun Li, and Jinjian Huang. Hyperparser: A high-performance parser architecture for next generation programmable switch and smartnic. In *Proceedings of the 5th Asia-Pacific Workshop on Networking*, pages 50–56, 2021.

[65] Xiangzhou Liu, Wenxue Li, and Kai Chen. Enabling packet spraying over commodity rnics with in-network support. In *Proceedings of the 9th Asia-Pacific Workshop on Networking*, APNET '25, page 51–58, New York, NY, USA, 2025. Association for Computing Machinery.

[66] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. {Multi-Path} transport for {RDMA} in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 357–371, 2018.

[67] Percy A MacMahon. *Combinatory analysis, volumes I and II*, volume 137. American Mathematical Society, 2001.

[68] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. 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*, pages 615–631, 2020.

[69] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018.

[70] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[71] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 313–326, 2018.

[72] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[73] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement. *Proceedings of the ACM on Management of Data*, 1(1):1–19, 2023.

[74] Dr. Vern Paxson and Mark Allman. Computing TCP's Retransmission Timer. RFC 2988, November 2000.

[75] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[76] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[77] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. Congestion control in machine learning clusters. In *Proceedings of the 21st*

ACM Workshop on Hot Topics in Networks, pages 235–242, 2022.

[78] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 81–92. IEEE, 2020.

[79] Abhiram Ravi, Nandita Dukkipati, Naoshad Mehta, and Jai Kumar. Congestion Signaling (CSIG). Internet-Draft draft-ravi-ippm-csig-01, Internet Engineering Task Force, February 2024. Work in Progress.

[80] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[81] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates. 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*, SIGCOMM '20, pages 735–749, New York, NY, USA, July 2020. Association for Computing Machinery.

[82] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

[83] 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. Falcon: A reliable, low latency hardware transport. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 248–263, New York, NY, USA, 2025. Association for Computing Machinery.

[84] W Wang, M Ghobadi, K SHAKERI, et al. How to build low-cost networks for large language models (without sacrificing performance)? *Available: How to Build Low-cost Networks for Large Language Models (without Sacrificing Performance)*, 2023.

[85] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, robust, and practical datacenter {CC} via deployable {INT}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 255–274. USENIX Association, 2023.

[86] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. Congestion control for cross-datacenter networks. *IEEE/ACM Transactions on Networking*, 30(5):2074–2089, 2022.

[87] Qinghua Zhang. Some Implementation Aspects of Sliding Window Least Squares Algorithms. *IFAC Proceedings Volumes*, 33(15):763–768, June 2000.

[88] Yiran Zhang, Qingkai Meng, Chaolei Hu, and Fengyuan Ren. Revisiting congestion control for lossless ethernet. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 131–148, 2024.

[89] Yiran Zhang, Qingkai Meng, Yifan Liu, and Fengyuan Ren. Revisiting congestion detection in lossless networks. *IEEE/ACM Transactions on Networking*, 31(5):2361–2375, 2023.

[90] Zhaochen Zhang. Online resource of OSCAR. https://github.com/NASA-NJU/OSCAR, 2026.

[91] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

[92] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 313–327, 2016.

[93] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 45(4):479–491, August 2015.

[94] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

[95] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, et al. Resiliency at scale: Managing {Google's}{TPUv4} machine learning supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774, 2024.

Appendix

Notation	Description
Δ	The gap between the current state and the target state.
$d(t)$	Delay at time t.
$g(t)$	Delay gradient at time t.
$w(t)$	Window size at time t.
$r(t)$	Rate at time t.
$d_q(t)$	Queuing delay at time t.
$q(t)$	Queue length at time t.
$i(t)$	Inflight size of time t.
$\lambda(t)$	Arrival rate of a queue at time t.
μ	Departure rate of a queue, which typically equals the line rate.
RTT_{base}	Base RTT, *i.e.*, RTT when there is no queue.
BDP_{base}	Base BDP. *i.e.*, line rate times base RTT.

Table 3: Notations used in OSCAR.

Notation	Description	Recommend
D_{target}	Target delay.	$1.5 \times RTT_{base}$
τ	The period of the batched estimator.	$0.5 \times RTT_{base}$
U_{AI}	Increase step to improve fairness.	0.001
U_{HAI}	Increase step to avoid underutlization.	0.01

Table 4: Parameters used in OSCAR.

A Supplemental Design Discussion

A.1 Additional Design Details

Solution of timestamp wrap-around in batched estimator. OSCAR employs a 4-byte timestamp in nanoseconds, leading to a wrap-around approximately every 4.3 seconds. To address the issue, we employ a low-overhead approach that records the highest bit of the latest received *sendTs*. A transition of this bit from 1 to 0 indicates a wrap-around, prompting the estimator to reset and start a new estimation cycle. This technique adds only 1 bit of storage overhead and introduces a maximum delay of τ in the estimation following a wrap-around.

How OSCAR works with ACK coalescing. As mentioned in § 4.1, OSCAR relies on ACKs to convey *sendTs* and *pktInfl*. In addition, OSCAR estimates the sending rate based on the number of ACKs echoed. OSCAR can work with ACK coalescing as these can be *partially omitted* or *aggregated* at the receiver end. (i) *sendTs* is used for delay gradient calculation. With ACK coalescing, OSCAR does not require any special handling on *sendTs*, *i.e.*, the receiver only needs to piggyback the *sendTs* of the data packet triggering the ACK. (ii) With ACK coalescing, the receiver needs to aggregate *pktInfl* and packet counts between two ACKs and include them in the ACK. This way, ACK coalescing will not affect the estimation of inflight and the sending rate. We modify OSCAR in ns-3 to use only three pairs of timestamps for each BLS computation (denoted as OSCAR-ACKC), simulating the impact of ACK coalescing on OSCAR. According to the experiments (Appendix C.5 and C.6), ACK coalescing has a minimal impact on performance.

Handling full queue scenarios. When the queue is empty or full, delay and delay gradient do not accurately reflect congestion information. The handling of empty queue is discussed in § 4.2. OSCAR does not specifically tackle full queue scenarios, as the delay becomes very large under such conditions, prompting OSCAR to rapidly reduce the window size to alleviate the severe congestion. As shown in § 6.4, OSCAR is effective in both lossless and lossy networks during severe congestion. OSCAR is orthogonal to existing solutions for severe congestion issues, such as head-of-line blocking in Priority Flow Control (PFC) [89] and packet loss in lossy networks [71].

A.2 Miscellaneous Detailed Discussion

Reverse congestion. RTT can be influenced by reverse congestion, *i.e.*, congestion on the ACK path. Our experiments show that reverse congestion minimally impacts OSCAR (Appendix C.6). OWD (one way delay) provides a more accurate delay measure than RTT, as it eliminates the impact of congestion on the ACK path. Precise OWD necessitates sub-microsecond clock synchronization between sender and receiver, which is achievable through the widely supported [9,28,49] Precision Time Protocol [6] (PTP). In the absence of hardware support for PTP, state-of-the-art software-based clock synchronization algorithms [37, 60] can be used to achieve nanosecond-level synchronization. Additionally, prioritizing ACK packets can also prevent the impact of reverse congestion on RTT [19, 43]. While OSCAR benefits from existing methods that mitigate reverse congestion, it also performs well independently, leveraging readily accessible RTT as the congestion signal.

Selection of target. The target state for HPCC is 95% utilization, *i.e.*, an empty queue with the port egress rate at 95% of the bandwidth. For PowerTCP, the target state is a 100% utilization, *i.e.*, an empty queue with the bandwith fully utilized. These algorithms aim to deliver low latency and high throughput, with minimal throughput loss (either 5% or none). However, their zero-queue target state can lead to severe under-utilization under dynamic network conditions. Considering two flows converging to the target state, when one flow finishes, CC requires at least an RTT control loop delay to reconverge to the target state. Without a queue, half the bandwidth will be wasted in the RTT. If a queue is maintained,

CC Alogrithm	Number of Parameters	Parameters Controlling Convergence State	Parameters Controlling Fairness Convergence Speed	Paremeters for Avoiding Underutlization
OSCAR	4	1: D_{target}	1: U_{AI}	1: U_{HAI}
DCQCN	9	3: $K_{min}, K_{max}, P_{max}$	5: B, T, τ', R_{AI}, g	6: $B, T, \tau', R_{AI}, g, F$
Swift	8	5: $base_target, \bar{h}, fs_max_cwnd, fs_min_cwnd, fs_range$	3: ai, β, max_mdf	1: ai
Poseidon	6	3: max_cwnd, min_cwnd, k	3: m, min_md, max_md	4: max_cwnd, min_cwnd, k, m
HPCC	3	1: η	1: W_{AI}	2: $W_{AI}, maxStage$
PowerTCP	2	*PowerTCP can not tune the target state.*	2: β, γ	1: γ

Table 5: Comparison of the parameter counts of CCs.

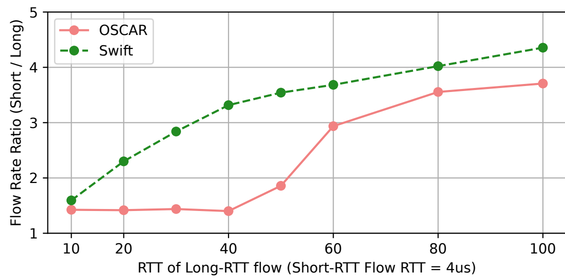


Figure 13: RTT-fairness evaluation.

the control delay will only reduce the queue length by half the BDP, rather than underutilization. Therefore, OSCAR sets the target state to half the BDP queue length (*i.e.*, $1.5 \times RTT_{base}$ target delay) to avoid underutilization of bandwidth in the aforementioned scenarios.

The impact of AI term in large-scale incast. OSCAR utilizes a consistent AI operation followed by each MIMD operation to ensure fairness. As the target of MIMD operations is converging to the target, the subsequent AI will cause the actual convergence point to be slightly higher than the target. Our recommended setting for U_{AI} is 0.001. Our experiments have demonstrated that in large-scale incast scenarios, this parameter only results in the convergence queue length increase by less than 1 KB per flow (Appendix C.4).

RTT-fairness. OSCAR is designed for high-throughput and low-latency data center networks. Consequently, akin to state-of-the-art delay-based data center CCs [57], OSCAR is not specifically designed for RTT fairness, *i.e.*, OSCAR can not ensure fairness when flows' RTT has a significant difference.

We evaluate the RTT fairness of OSCAR through simulation experiments. The experiment involves two flows with disparate RTTs competing for a single bottleneck. The short-RTT flow's RTT is configured to 4 µs, whereas the long-RTT flow's RTT is varied from typical intra-datacenter RTTs (~10 µs) to those observed between data centers in different buildings (~100 µs, corresponding to ~10 km distance between buildings). Figure 13 shows the converged throughput ratio of the two flows. Our results indicate that OSCAR achieves better RTT fairness compared to Swift. A more comprehensive investigation into the RTT fairness of OSCAR is reserved for future work.

Performance under multi-hop congestion. INT-based CCs [7,61,85] can achieve max–min fairness under multi-hop congestion. In contrast, OSCAR cannot as delay and delay-gradient signals do not reveal whether an increase originates from a single bottleneck or multiple bottlenecks. Nonetheless, OSCAR does not cause starvation. We demonstrate this in a 100 Gbps line-rate parking-lot topology with three long-lived flows, where Flow 3 shares one bottleneck with Flow 1 and another with Flow 2, respectively. With OSCAR as the CC, the achieved throughputs are 73.5 Gbps, 73.5 Gbps, and 25 Gbps for Flows 1-3, respectively. Flow 3 is not starved, and both bottlenecks remain fully utilized.

Cross-datacenter traffic. Inter-datacenter (inter-DC) communication is gaining increasing attention, driven by the growing demand for model training and inference across datacenters. Although OSCAR is not specifically designed for inter-DC communication, it can be integrated with existing inter-DC CC algorithms. Prior work on inter-DC CC [18, 81, 86] typically leverages the combination of multiple congestion signals, such as ECN and QCN [81], or combinations of delay and ECN [18,86]. OSCAR can be combined with QCN or ECN, provided that a extended control-loop coordination mechanism (e.g., Annulus-style coordination or extended OSCAR's own coordination mechanism) is used to ensure that the delay and delay-gradient estimates remain accurate when additional congestion signals take effect.

Parameter Count and Tuning Complexity. We argue that for a CC algorithm, *having fewer parameters is not necessarily better*. An ideal CC should provide an orthogonal parameter for each of its key aspects, such as the convergence target state, fairness convergence speed, and avoidance of link underutilization. If an aspect lacks a parameter for tuning, the algorithm may perform well only in its specific design scenario but poorly in diverse network environments, thus lacking flexibility. Conversely, if multiple parameters control a single aspect, the coupling among them can make the tuning process exceptionally complex.

OSCAR's design adheres to this principle. As shown in Table 5, OSCAR uses D_{target} to control its convergence state, U_{AI} to control the fairness convergence speed, and U_{HAI} to avoid underutilization. *This orthogonality greatly simplifies*

tuning. Compared to HPCC, OSCAR's only additional parameter, *i.e.*, the BLS calculation period τ, has a well-tested default value (half the base RTT) and low performance sensitivity (Appendix C.3), making it not an issue in practice.

With only four parameters, OSCAR is tuning-friendly than many industry-deployed CCs, such as DCQCN (9) [91] and Swift (8) [57]. Thus, OSCAR's design strikes a deliberate balance, providing sufficient orthogonal "tuning knobs" for flexibility and low complexity.

Challenges in extracting precise congestion information from standard single-bit signal. By aggregating single-bit feedback over time, an end host can effectively recover multi-bit information about network conditions. This is evidenced by ABC [40], where the sender aggregates *ECN-based signals precisely calculated and marked by switches* to achieve rapid rate adjustments.

However, with standard *probabilistic* ECN marking, the achievable precision is fundamentally limited. Consider a switch that applies linear-probability ECN marking over a queue range of 100 KB–2 MB in a 200 Gbps network with 10 μs RTT and 10 flows sharing a bottleneck. With a 1500-byte MTU and no ACK coalescing, a sender observes only about 16 ACKs per RTT. Because marking is stochastic and the sample size per RTT is small, estimating the instantaneous queue length from the ECN-mark ratio within one RTT incurs an expected error of about 200 KB—approximately one BDP in this setting.

How often queues are non-empty. Delay and delay-gradient signals are accurate only when the queue at the congestion point is neither empty nor full. As detailed in Appendix A.1, OSCAR handles full queue scenarios well. Below we investigate how often queues are non-empty when using OSCAR.

Because OSCAR drives the queue length to a target (a shallow queue) in $O(1)$ time, a queue is present for most of time when bandwidth contention exists under OSCAR's operation. In large-scale experiments, it is difficult to directly determine whether a port experiences bandwidth contention and whether a queue forms under contention. We therefore use a representative micro-benchmark to provide quantitative evidence. We setup ten hosts as senders and one host as the receiver under a ToR switch. The senders generate traffic at 80% load according to a Poisson process and using WebSearch trace. The ToR-to-receiver port is the bottleneck and should see contention for 80% of the time. Benifiting to OSCAR's $O(1)$-step convergence, this port has a queue for 72.2% of the time with OSCAR as the CC, which is closed to 80% theoretical bound. In contrast, with DCQCN, the port has a queue for only 47.6% of the time. This is because DCQCN increases its sending rate at only $O(\Delta)$. After rate backoff, the port remains underutilized for a long time.

A.3 Compatibility with Per-packet LB

Per-packet LB is widely adopted in AI training to prevent flow collisions caused by per-flow ECMP. However, existing sender-driven CCs typically struggle in this environment.

Classical AIMD-based CCs may suffer from under-utilization. CCs like TIMELY and DCQCN are sensitive to congestion signals on a single path [62]. Therefore, congestion on a single path can trigger a significant rate reduction. Moreover, they suffer from slow acceleration after rate reduction, resulting in bandwidth underutilization.

To illustrate this issue, we follow the experimental setup in [62]. The setup uses a spine-leaf topology with four spine switches. We initiate four flows from four hosts under one leaf switch to four hosts under another. All links have a bandwidth of 100 Gbps, and packet spraying is employed as per-packet LB. Ideally, the aggregate rate of the flows should be 400 Gbps. Between 300 µs and 500 µs, a downward port on one of the spine switches experiences a failure, halving its bandwidth to 50 Gbps. The results are presented in Figure 14. All tested CCs' parameters are aligned with the original paper.

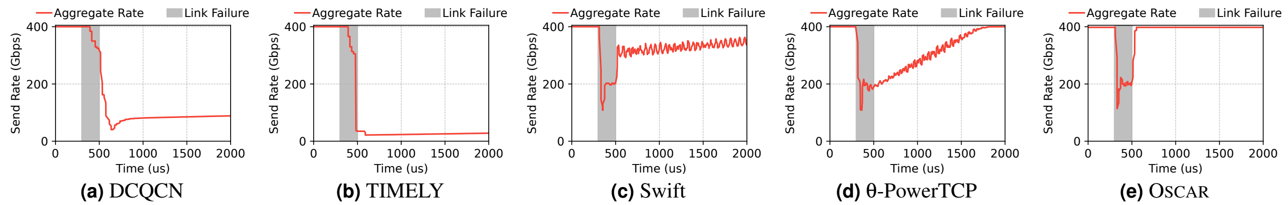
As shown in Figure 14a and 14b, DCQCN and TIMELY exhibit an excessive rate reduction immediately following the link failure. After the failure is resolved, their rates recover at a speed of $O(\Delta)$, leading to severe under-throughput. Figures 14c and 14d present the results for Swift and theta-PowerTCP. They quickly converge the rate to half of the aggregate bandwidth after the link failure. (Note that without a dynamic bandwidth allocation scheme like adaptive routing [38], half the bandwidth is the target state for an end-to-end CC algorithm). When the failure ends, these algorithms are also limited by an $O(\Delta)$ step convergence speed, which results in under-throughput.

Existing precise-INT-based CCs are fundamentally incompatible with per-packet LB. The core of this issue lies in their method for calculating port transmission rates *txRate*, which relies on differencing the "transmitted byte" (*txByte*) field from two consecutive INT headers. In a per-packet LB network, packets are distributed across multiple paths, causing sequential INT headers to record *txByte* of different physical ports. As a result, the calculation of *txRate* becomes entirely meaningless.

A potential solution to this incompatibility is to maintain more state both in the INT header and the sender. In one such scheme, the INT header needs to be augmented with a Port ID to mark the source of the congestion information explicitly. The sender could then store state for each port individually and use sequential INT data from a port to calculate congestion signals.

However, this approach is likely *impractical due to its prohibitive overhead.*

(i) It significantly increases packet header size. Given that modern large data centers can have more than 100,000 ports [75], the Port ID field would require at least 3 bytes. For a



(a) DCQCN **(b)** TIMELY **(c)** Swift **(d)** θ-PowerTCP **(e)** OSCAR

Figure 14: Microscopic view of Non-INT-based CCs under Per-packet LB with a Link Failure (300 µs-500 µs).

typical 5-hop path, this results in 15 bytes of additional packet header overhead. Note that the current *pathID* field in the INT header is the XOR of all switch IDs along the path. It is only used to detect path change, but can not identify the ports the packet passes through in per-packet LB, where the packet's path is randomly selected at each hop [62] or selected according to network state at each hop [38].
(ii) The sender must maintain a substantial amount of state for each flow. Assuming a flow could be distributed across 40 ports (the potential passing port count for a flow in $k = 8$ fat-tree), the required storage for just the telemetry state would be on the order of 40 hops $\times$ (3-byte ID + 6-byte state) = 360 bytes per flow (in this case, both the TS and the *txByte* fields must be stored). This is significantly larger than the state required by existing CCs (around 60 bytes [66]).

OSCAR is compatible with per-packet load balancing (LB). OSCAR's batched estimator overcomes the limitations mentioned above by aggregating congestion information over a period of time. This design coincides with the core principles of the state-of-the-art CC designed for per-packet LB that aggregates congestion signals over a period of time, preventing congestion signals from a single path from impacting the CC's decision. Figure 3e exhibits OSCAR's behavior during the link failure event. Upon both the beginning and ending of the failure, OSCAR converges to the target state in $O(1)$ steps, which surpasses the performance of other CC algorithms. This illustrates that OSCAR is compatible with per-packet LB and maintains $O(1)$-step convergence in such environments.

B Discussion of Related Works

Difference compared with XCP. XCP's control logic is performed on switches. Each switch first multiplicatively calculates the total window adjustments, and then distributes adjustments to all flows according to the AIMD principle. This method achieves $O(1)$-step convergence and can ensure fairness. However, XCP's fairness mechanism is different from OSCAR's. In XCP, each flow receives an equal increase share (AI) when accelerating and a decrease share proportional to its window (MD) when decelerating. From the perspective of flows, this is a variable-step-size AIMD rather than MIMD, because although the increase step size varies across different acceleration events, it is equal for all flows within a single event. It is proven that variable-step-size AIMD cannot achieve $O(1)$-step convergence through end-to-end congestion signals (Observation 1.2 in § 3.1).

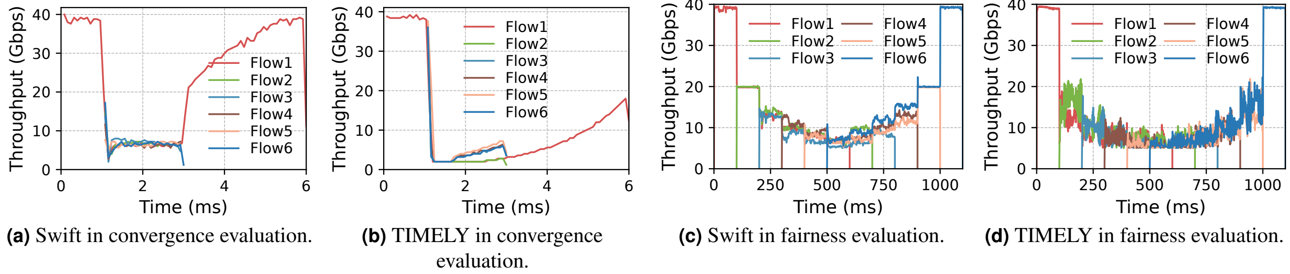
Difference of fairness principles (§ 4.3) compared with fairness analysis of HPCC and PowerTCP. HPCC [61] and PowerTCP [7] both employ additional AI operations to achieve fairness. However, neither paper directly proves that additional AI operations can guarantee fairness for multiple flows at a single bottleneck. Instead, they first assume fair convergence and then prove the relationship between AI operations and bandwidth allocation at multiple bottlenecks (namely, max-min fairness or proportional fairness). To the best of our knowledge, we are the first to *explicitly propose and directly prove* that additional AI operations can provide fairness guarantees for sender-driven MIMD-based CCs.

Difference of fairness principles (§ 4.3) compared with fairness analysis of Poseidon. Poseidon explores a fairness principle beyond AIMD, which allows low-rate flows to accelerate more significantly, while high-rate flows experience smaller increases or even deceleration. The fairness analysis of Poseidon is not covered by the fairness principles proposed in this paper because its MIMD operations are not *linear* relative to its own state.

Novelty compared with θ-PowerTCP. θ-PowerTCP has the same control law as PowerTCP but uses delay and delay gradient instead of INT. However, it overlooks the relationship between the delay gradient and the bottleneck arrival rates, assuming the bottleneck is always fully utilized. Therefore, it dose not perform MI when the bandwidth is underutilized, leading to a convergence rate of $O(\Delta)$. Our micro-bench simulation shows this limitation (Appendix C.5). Moreover, its delay gradient calculation is inevitably vulnerable to errors in real-world environments as it directly uses division for calculation.

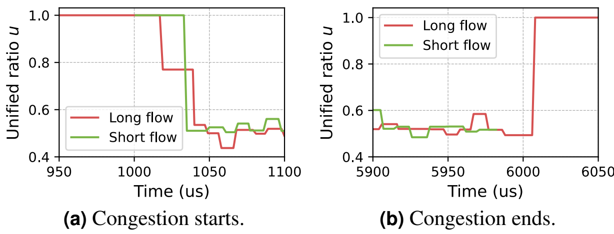
Discussion of ACC [88]. By leveraging the knowledge of ACK time series, ACC could achieve $O(1)$-step convergence when facing congestion in the lossless network. However, compared with OSCAR, it still faces the issue of slow ($O(\Delta)$-step) bandwidth reclaiming, a drawback stemming from its reliance on the AI mechanism. Furthermore, its rapid convergence property could be compromised in lossy networks.

Internet CCs. Recent developments in Internet CC have made significant progress, with many algorithms based on delay or delay gradient [15, 23, 24, 68]. However, these CCs rely on the AIMD and fail to achieve $O(1)$-step convergence.



(a) Swift in convergence evaluation. **(b)** TIMELY in convergence evaluation. **(c)** Swift in fairness evaluation. **(d)** TIMELY in fairness evaluation.

Figure 15: Supplemental testbed evaluations.



(a) Congestion starts. **(b)** Congestion ends.

Figure 16: Microscopic view of OSCAR in testbed.

OSCAR cannot be directly applied to the Internet, as the line rate μ of the bottleneck is typically unknown, which will be the focus of our future work.

C Supplemental Results

C.1 Supplemental Testbed Results

AIMD-based CC cannot quickly converge to the target state. Figures 15a and 15b illustrate the convergence of Swift and TIMELY on the testbed. The experimental setup is the same as in Figure 6a, with three short flows starting at 1 ms and lasting for 2 ms. Swift takes 3 ms to fully utilize the bandwidth after the short flows end. TIMELY overreacts during the appearance of the short flows.

OSCAR achieves more robust fairness than Swift and TIMEY in the testbed. The experimental setups for Figures 15c and 15d are the same as in Figure 6b. It can be observed that Swift cannot quickly converge to fairness. After a thorough investigation, we discover that the issue stemmed from our mixed use of ConnectX-5 and ConnectX-6 NICs, which resulted in different base RTTs between flows. In OSCAR, the delay gradient control loop facilitates fair convergence when base RTTs vary. Meanwhile, TIMELY's flows suffer from overreaction, struggling to converge fairly. The observation on TIMELY is aligned with that in PowerTCP's paper [7].

Microscopic view of in testbed proves OSCAR can achieve $O(1)$-step convergence. Figure 16 illustrates the detailed behaviour of OSCAR in a testbed. In the experiment, a long flow runs continuously while a short flow joins at 1 ms and leaves at 6 ms. The figure shows OSCAR's unified ratio u around 150 µs during congestion starts and ends. As shown in Figure 16a, at the start of congestion, the short flow converges to the target state (u=0.5) in a single update, and the long flow converges to the target state using two steps. Figure 16b shows that, following the end of congestion, the long flow converges to the target state ($u = 1$) in a single update.

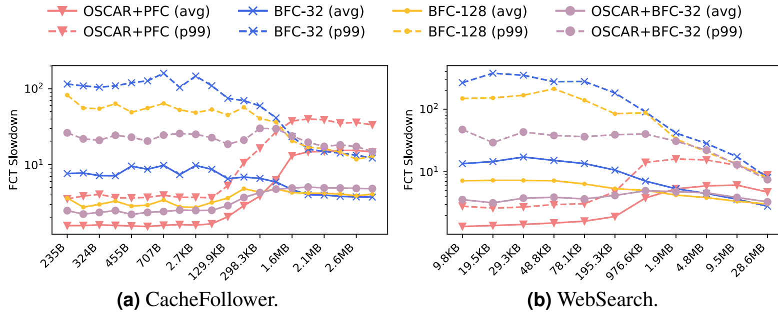
C.2 Comparison with Flow Control

We also evaluate recent in-network flow-control (FC) algorithms, such as BFC [41]. FCs are different from end-to-end CCs, enabling faster congestion management via hop-by-hop switch-based control. However, they are not readily deployable because they require switch modifications. We first briefly compare OSCAR against BFC in terms of standalone performance and then examine whether OSCAR can be integrated with BFC.

We use the same experimental setup as in §6, with traffic generated by CacheFollower and WebSearch at 50% load, as well as a mix of 15% incast and 30% Hadoop traffic. For BFC, we evaluate two configurations: 32 queues (BFC-32) and 128 queues (BFC-128), with parameters following the paper [41]. For PFC, we set the same Xon/Xoff thresholds as BFC to ensure a fair comparison.

Figure 17 shows the breakdown of FCT slowdown under CacheFollower and WebSearch workloads. Compared to OSCAR+PFC, BFC-32 degrades small-flow performance but significantly improves large-flow performance. This is because our workloads contain a large number of flows to simulate the busy period of the network. Once the number of concurrent flows on a port exceeds 32, BFC's queue sharing mechanism introduces head-of-line blocking for small flows. BFC-128 markedly improves small-flow performance over BFC-32, supporting our analysis. OSCAR can be combined with BFC to mitigate head-of-line blocking. As shown in Figure 17, OSCAR+BFC-32 achieves better small-flow performance than BFC-128, while delivering comparable performance for large flows. This is because, once congestion occurs, OSCAR effectively reduces traffic injection at the host, which lowers the frequency of BFC pauses.

Figure 18 shows the average and tail FCT slowdown under three workloads. Here, *Hadoop* reports the FCT slowdown of Hadoop flows under a mixed load of 15% incast and 30% Hadoop flows. Under CacheFollower, OSCAR+PFC achieves



(a) CacheFollower. **(b)** WebSearch.

Figure 17: Breakdown of FCT slowdown in various workload at 50% load.

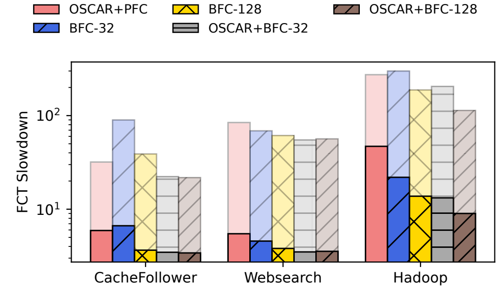
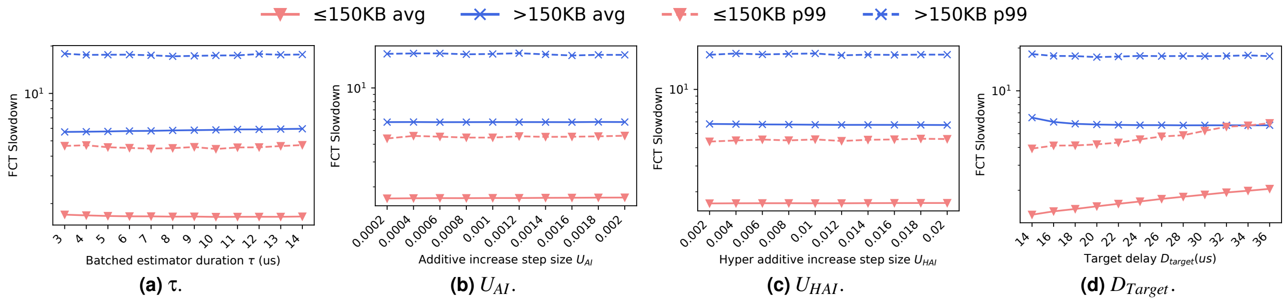


Figure 18: Average and tail (99th percentile) FCT with BFC.



(a) τ. **(b)** U_{AI}. **(c)** U_{HAI}. **(d)** D_{Target}.

Figure 19: FCT slowdown under different parameters.

an 11% lower average FCT slowdown than BFC-32, whereas under WebSearch it is 19% worse than BFC-32. This is because the CacheFollower trace contains flows with smaller size, leading to more concurrent flows and making BFC more prone to head-of-line blocking. Under Hadoop+Incast, OSCAR+PFC is 1.14× worse than BFC-32 because the incast degree exceeds that PFC can sustain, causing severe head-of-line blocking in PFC-enabled network, while BFC avoids excessive blocking. When combining OSCAR with BFC, we observe a significant reduction in FCT across workloads, indicating that OSCAR can integrate with existing FCs to further improve network performance. The largest gain occurs under CacheFollower workload, where OSCAR+BFC-32 reduces FCT slowdown by 41% and 48% compared to OSCAR+PFC and BFC-32, respectively.

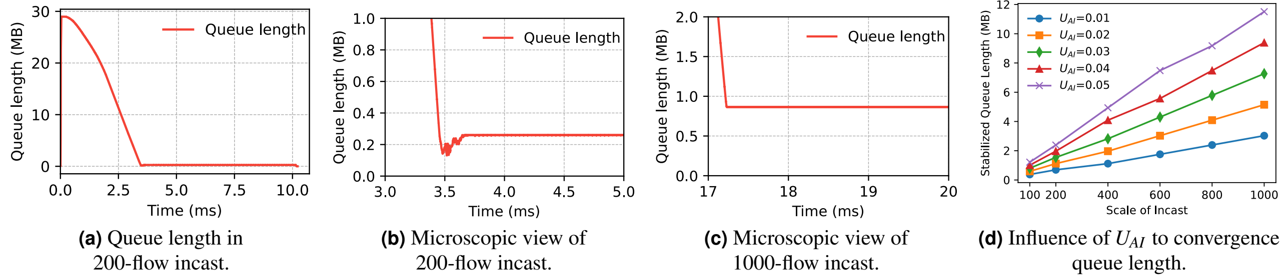
C.3 Parameter Sensitivity Analysis

Figure 19 demonstrates that OSCAR is not sensitive to the selection of the parameters U_{AI}, U_{HAI} and τ. The FCT for both small and larger flows does not exhibit significant changes with different parameter choices. The result indicates that hyper increase is only used as a last resort in OSCAR, which is seldom triggered. This is because OSCAR is capable of maintaining the queue length stably at the target in the vast majority of cases. As the target delay D_{target} increases, the FCT for small flows increases, while the FCT for large flows decreases. We have selected 18 µs, *i.e.*, $1.5 \times RTT_{base}$ as a good balance. This result supports our discussion on the selection of the target in Appendix A.2.

C.4 Large-scale Incast

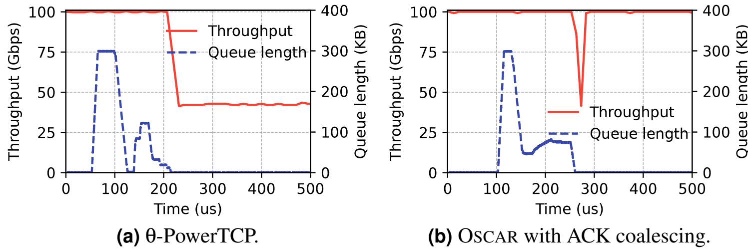
OSCAR can handle large-scale incast gracefully. We evaluate OSCAR's performance in large-scale incast using a single bottleneck topology in ns-3. The link bandwidth is set to 100 Gbps, with the base RTT setting to 12 µs. And the parameters of OSCAR are aligned with § 4.4. All flows are started randomly within one base RTT to transmit 600 KB data. Figure 20a illustrates the queue length during a 200-flow incast. Initially, the queue length rises to around 30 MB (equal to the sum of initial windows of 200 flows) and then rapidly converges. Figure 20b shows the microscopic view of the convergence of OSCAR in 200-flow incast. After drainage of the queue length caused by the initial windows, there is a brief oscillation, and then stable convergence. It is notable that the convergence queue length (around 230 KB) exceeds the target queue length (75 KB). This is due to the influence of the AI operation, which will be discussed below. Figure 20c shows the microscopic view of the convergence of OSCAR in 1000-flow incast, which is similar to 200-flow incast but has a larger convergence queue length (around 900 KB).

The step size of AI operation will influence the convergence queue length in large-scale incast. As discussed in Appendix A.1, the AI operations followed by MIMD



(a) Queue length in 200-flow incast. **(b)** Microscopic view of 200-flow incast. **(c)** Microscopic view of 1000-flow incast. **(d)** Influence of U_{AI} to convergence queue length.

Figure 20: OSCAR in large-scale incast.



(a) θ-PowerTCP. **(b)** OSCAR with ACK coalescing.

Figure 21: Supplemental microscopic view evaluations.

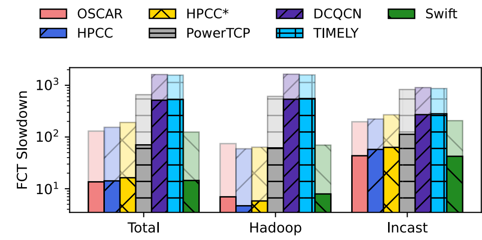


Figure 22: Lossy evaluation.

operations will raise the actual queue convergence point. Figure 20d illustrates the steady-state queue lengths under different incast scale and U_{AI} settings. To make results more significant, we set U_{AI} from 0.01 to 0.05 (our recommended value is 0.001). It is notable that the queue length increases linearly with both the incast scale and the U_{AI}. Therefore, we recommend setting the U_{AI} to 0.001, which ensures fairness while resulting in the steady-state queue length increase of approximately 900 KB per 1000 flows (Figure 20c). As shown in Appendix C.4, in large-scale incast, the increase in queue length is linearly related to U_{AI} and the number of flows. Using the recommended U_{AI} value of 0.001 results in a steady-state queue length increase of 900KB per 1000 flows, which is acceptable. In non-incast scenarios, the increase in queue length due to U_{AI} is negligible.

C.5 Supplemental Microscopic View Evaluations

θ-PowerTCP needs $O(\Delta)$ steps for convergence. Figure 21 replicates the experiment shown in Figure 9. As shown in Figure 21a, θ-PowerTCP exhibits fluctuations during queue convergence. After the micro-burst ends, θ-PowerTCP fails to converge rapidly. θ-PowerTCP accelerates using AI, resulting in $O(\Delta)$-step convergence.

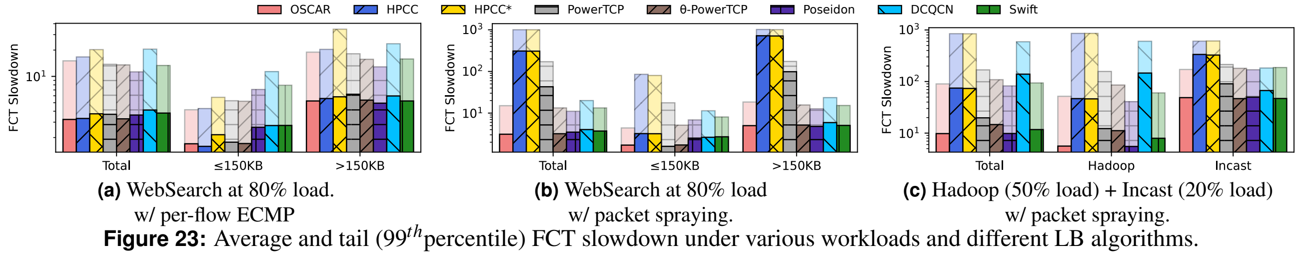
ACK coalescing has minimal impact on OSCAR during micro-burst. Figure 21b shows the reaction of OSCAR-ACKC (OSCAR with ACK coalescing) to micro-burst. The performance of OSCAR-ACKC is basically identical to OSCAR without ACK coalescing (Figure 8b). It quickly converges the queue to the target length following a micro-burst and fully utilizes the bandwidth after the micro-burst ends.

C.6 Supplemental Large-scale Simulations

OSCAR performs well under lossy environment. Figure 22 illustrates the performances of each algorithm under lossy environment. Comparing to lossless environment, OSCAR and HPCC suffer only a little performance decline due to well queue length control, while the performance of DCQCN and TIMELY is severely damaged.

OSCAR performs well under WebSearch workload, both with per-flow ECMP and packet spraying. Figure 23a shows the FCT slowdown under the WebSearch workload at 80% load. For small flows under 150KB, OSCAR's average slowdown is only 2.6% worse than HPCC, and better than all other algorithms, especially 27.0% and 4.4% better than HPCC* and PowerTCP. For larger flows, OSCAR outperforms HPCC and PowerTCP by 6.6% and 19.1% on average. Overall, OSCAR outperforms HPCC, HPCC* by 2.7%, 16.1% on average and 10.8%, 34.2% at tail, respectively. OSCAR is better by 14.6% on average and inferior by 8.7% at tail than PowerTCP. Figure 23b and Figure 23c repeat experiments in Figure 23a and Figure 10c with packet spraying, respectively. The results align with the analysis in Appendix A.1. OSCAR outperforms all other CCs we compare under the packet spraying network.

OSCAR performs well in oversubscribed network. For experiments in over-subscription typologies, we adjust inter-switch bandwidth to 200 Gbps and 100 Gbps for 2:1 and 4:1 over-subscription ratios, respectively. The workload is

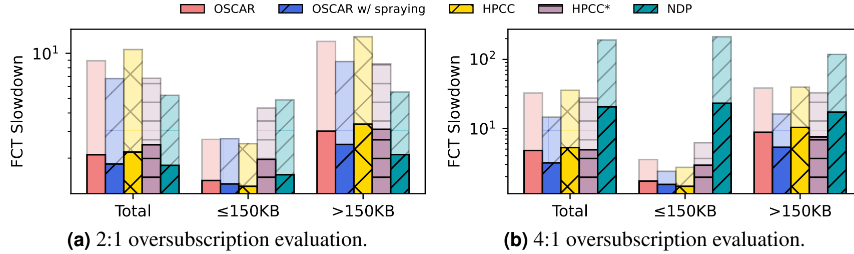


(a) WebSearch at 80% load. w/ per-flow ECMP

(b) WebSearch at 80% load w/ packet spraying.

(c) Hadoop (50% load) + Incast (20% load) w/ packet spraying.

Figure 23: Average and tail (99^{th}percentile) FCT slowdown under various workloads and different LB algorithms.



(a) 2:1 oversubscription evaluation.

(b) 4:1 oversubscription evaluation.

Figure 24: Supplemental large-scale oversubscription evaluations.

WebSearch at 80% load. Under both oversubscription ratios, OSCAR performs better than HPCC by 10% and 8%.

In this topology, we also evaluate the performance of NDP, a receiver-driven CC based on packet spraying. To ensure a fair comparison, we compare it against OSCAR configured with packet spraying (OSCAR w/ spraying in Figure 24). With a 2:1 oversubscription, OSCAR with spraying outperforms NDP by 15% for small flows, though it underperforms by 11% for large flows. At a 4:1 oversubscription, NDP experiences a performance collapse due to its inability to handle in-network congestion. In this case, OSCAR with spraying outperforms NDP by 6.5×.

Reverse congestion has minimal impact on OSCAR. Figure 25 shows the performance of OSCAR and OSCAR-OWD, *i.e.*, OSCAR one-way-delay instead of RTT. Reverse congestion causes OSCAR to slow down, leading to increased FCT for large flows, while small flows that complete within one RTT benefit from reduced queue lengths. Under the CacheFollower workload, OSCAR's FCT for small flows is 7% better than OSCAR-OWD, while the FCT for large flows is 10% worse than OSCAR-OWD, resulting in an overall FCT that is 8% worse. Under the Hadoop with incast workload, the overall FCT, as well as the FCT for Hadoop and incast flows of OSCAR and OSCAR-OWD, differ by less than 5%. Overall, reverse congestion has only a slight impact on OSCAR.

ACK coalescing has minimal impact on OSCAR. Figure 25 shows the performance of OSCAR and OSCAR-ACKC, *i.e.*, OSCAR with ACK coalescing. Under the CacheFollower workload, OSCAR-ACKC exhibits similar performance compared to OSCAR with differences in small and larger, and overall performance not exceeding 5%. Under the Hadoop with incast workload, the network are more bursty. Nevertheless, the FCT of Hadoop traffic in OSCAR-ACKC is only 1.6% worse than that of OSCAR, with an overall performance degradation of just 6.2%.

D Theoretical Analysis

D.1 Convergence Speed Analysis

We employ *MacMahon's master theorem* [67] to demonstrate the following theorem.

Theorem D.1. *A CC that eliminates a ζ portion ($\zeta < 1$) of the current and target state gap Δ in each update converges in $O(\log_{\frac{1}{1-\zeta}} \Delta)$ steps.*

Proof. Let $S(\Delta)$ denote the number of steps required by this CC to converge. $S(\Delta)$ can be expressed as:

$$S(\Delta) = \begin{cases} 1 & \Delta < \varepsilon \\ S((1-\zeta)\cdot\Delta)+1 & \text{otherwise} \end{cases}$$

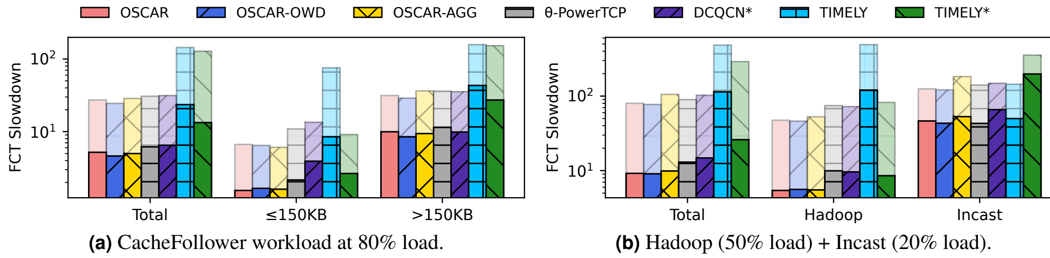
Here, ε represents a small quantity, and convergence is considered achieved when the gap Δ between the current state and the target state is less than ε. By recursively applying the above formula, it can be easily derived that:

$$\begin{aligned} S(\Delta) &= S((1-\zeta)\cdot\Delta)+1 \\ &= S((1-\zeta)^2\cdot\Delta)+2 \\ &= \cdots \\ &= S((1-\zeta)^n\cdot\Delta)+n \end{aligned}$$

And the upper bound of n is given by:

$$\begin{aligned} &(1-\zeta)^n\cdot\Delta < \varepsilon \\ \Rightarrow & n < \log_{1-\zeta}\frac{\varepsilon}{\Delta} = \log_{\frac{1}{1-\zeta}}\frac{\Delta}{\varepsilon} \end{aligned}$$

Therefore, the CC needs $O(\log_{\frac{1}{1-\zeta}} \Delta)$ steps to converge. □



(a) CacheFollower workload at 80% load. **(b)** Hadoop (50% load) + Incast (20% load).

Figure 25: Supplemental comparison between CCs.

D.2 Dual-state Convergence Analysis

We employ dual-state convergence analysis to demonstrate the following theorem.

Theorem D.2. *When both window and rate are used simultaneously in a CC, convergence is feasible only when the window and rate are at the same ratio relative to the target window and target rate, respectively.*

Proof. For n flows passing through the same bottleneck, each with a window w_i and rate r_i and the target total window is W, the target total rate is R, a reasonable assignment of r_i and w_i should satisfy the following properties.

1. If two flows have the same w_i, they should also have the same r_i, and vice versa.
2. The flow with larger w_i should have a larger r_i, and vice versa.
3. If a set of w_i satisfies $\sum_{i=1}^{n} w_i = W$, then the corresponding r_i should satisfy $\sum_{i=1}^{n} r_i = R$, and vice versa.

From Property 1, we deduce that there is a bijection between w_i and r_i, namely, there exists a function f such that $f(w_i) = r_i$. Consider an assignment $w_1, w_2, \ldots, w_n$ and $r_1, r_2, \ldots, r_n$ such that $\sum_{i=1}^{n} w_i = W$ and $\sum_{i=1}^{n} r_i = R$. The CC plans to modify this by decreasing j^{th} flow's window by Δw and increasing k^{th} flows window by Δw, resulting in a new assignment w_i' and r_i'. After the modification, the sum of windows remains constant at $\sum_{i=1}^{n} w_i' = W$, implying that the sum of rates should also be maintained at $\sum_{i=1}^{n} r_i' = R$, *i.e.*, $\sum_{i=1}^{n} f(w_i') = R$. Therefore, we have $f(w_j) + f(w_k) = f(w_j') + f(w_k')$. By rewriting the equation, we have $f(w_j) - f(w_j - \Delta w) = f(w_k + \Delta w) - f(w_k)$. Assuming Δw is infinitesimal and dividing both sides by Δw, we derive $\frac{f(w_j) - f(w_j - \Delta w)}{\Delta w} = \frac{f(w_k + \Delta w) - f(w_k)}{\Delta w}$, *i.e.*, f has the same derivative at w_j and w_k. Since w_j and w_k are arbitrarily chosen, this indicates that the function f has a consistent derivative across any point.

From Property 2, we can derive two boundary values for the assignment. A flow with a window of zero should also have a rate at zero, *i.e.*, $f(0) = 0$. A flow at the target window W corresponds to the target rate R, *i.e.*, $f(W) = R$. Therefore we have $f(w_i) = \frac{R \cdot w_i}{W}$, which is a linear mapping of window and rate. The derivation above is both sufficient and necessary, meaning that $f(w_i) = \frac{R \cdot w_i}{W}$ is the only function that satisfies the three properties. Therefore, we have $\frac{w_i}{W} = \frac{r_i}{R}$, *i.e.*, the window and rate are at the same ratio relative to the target window and target rate, respectively. □

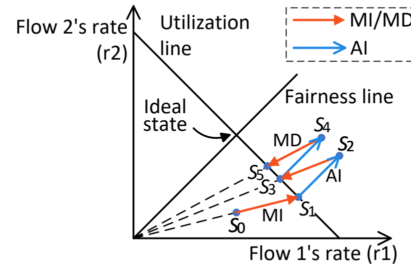


Figure 26: MIMD with consistent AI converges to fairness.

D.3 Fairness of MIMD-based CC

Graphical proof. Figure 26 depicts the convergence of two flows, with the horizontal and vertical axes indicating their respective rates r_1 and r_2. The line defined by $r_1 = r_2$, represents equal rates between the flows, termed the Fairness line. The line $r_1 + r_2 = \mu$ represents the bandwidth is fully utilized, and this line is called the Utilization line.

Initially, r_1 is greater than r_2, and the sum of r_1 and r_2 is less than μ, *i.e.*, the initial state S_0 is neither fair nor fully utilizing the bandwidth. Subsequently, the two flows move to the Utilization line via the MI operation (S_1). In the figure, the linear MIMD operation moves state along the line connecting the origin and the previous state, which is defined by $\frac{r_1}{r_2} = \frac{r_1'}{r_2'}$ where (r_1', r_2') represents the previous state. Linear MIMD operation maintains fairness between the two flows, as the ratio of their rates remains unchanged (Proposition D.1). After the MI operation, both flows employ an AI operation, which advances the state along a 45-degree line to S_2. This operation enhances the fairness (Proposition D.2) with a little overload. Then the MD operation moves back the state to the Utilization Line at S_3. It can be observed that the state S_3 is closer to the Fairness line than S_1. Subsequently, repeated AI and MD operations continuously move the state closer to the Fairness line, ultimately achieving fairness.

Formal algebraic proof. For the sake of brevity, our proof is based on rate and assumes flows are synchronized. The proof is also applicable to CC based on window or both window

and rate. In the proof, we use $\sum$ as the abbreviation of $\sum_{i=0}^{n}$. We define a state sequence beginning at state 0, where each flow has an initial rate of $r_i(0)$, and the initial fairness is $F(0)$. After each operation, the state number increments by one.

Consider n flow with rate at $r_1(t), r_2(t), ..., r_n(t)$. The fairness metric among these flows is defined as:

$$F(t) = \frac{(\sum r_i(t))^2}{n \sum r_i(t)^2}$$

The fairness metric **is always less than or equal to 1**, with **values approaching 1 indicating greater fairness**.

Proposition D.1. *MIMD operation that is linear with respect to its own state does not increase or decrease fairness between flows.*

Proof. A general MIMD operation can be expressed as $r(t) = r(t-1)^{\gamma} \cdot \beta$. For this lemma, we focus on the linear case where $\gamma = 1$. Assuming an MIMD operation occurs at time t, the fairness metric after the operation is

$$\begin{aligned} F(t) &= \frac{(\sum r_i(t))^2}{n \sum r_i(t)^2} = \frac{(\sum r_i(t-1) \cdot \beta)^2}{n \sum (r_i(t-1) \cdot \beta)^2} \\ &= \frac{(\sum r_i(t-1))^2}{n \sum r_i(t-1)^2} = F(t-1) \end{aligned}$$

Hence, the fairness remains unchanged after a linear MIMD operation. □

Proposition D.2. *AI operation increases fairness between flows.*

Proof. An AI operation is expressed as $r(t) = r(t-1) + \alpha$. Assuming an AI operation occurs at time t, the fairness metric after the operation is

$$\begin{aligned} F(t) &= \frac{(\sum r_i(t))^2}{n \sum r_i(t)^2} = \frac{(\sum (r_i(t-1) + \alpha))^2}{n \sum (r_i(t-1) + \alpha)^2} \\ &= \frac{(\sum r_i(t-1))^2 + 2n\alpha \sum r_i(t-1) + n^2\alpha^2}{n(\sum r_i(t-1)^2 + 2\alpha \sum r_i(t-1) + n\alpha^2)} \\ &= \frac{(\sum r_i(t-1))^2 + 2n\alpha \sum r_i(t-1) + n^2\alpha^2}{\underbrace{n \sum r_i(t-1)^2}_{F(t-1)} + \underbrace{2n\alpha \sum r_i(t-1) + n^2\alpha^2}_{\text{last two terms are the same}}} \\ &\geq F(t-1) \end{aligned}$$

The equation holds as an equality if and only if $F(t-1) = 1$.Hence, the fairness are increased after an AI operation. □

Theorem D.3. *MIMD-based CC converges to fairness if the following conditions are met. (1) The MIMD operation is linear with respect to its own state, (2) An AI operation is executed subsequent to each MIMD operation.*

Proof. Record the fairness metric $F(t)$ after each operation to form a sequence $(F(t))_{t=0}^{\infty}$. Demonstrating that a CC converges to fairness is proving that this sequence converges to 1.

We first prove the sequence $(F(t))_{t=0}^{\infty}$ converges. In the sequence where each MIMD operation is followed by an AI operation, the odd-numbered terms in the sequence correspond to the results after the MIMD operations, and even-numbered terms correspond to AI operations. Therefore we have

$$\begin{cases} F(t) = F(t-1) & \text{if } t \text{ is odd} \\ F(t) \geq F(t-1) & \text{if } t \text{ is even} \end{cases}$$

Consequently, the sequence $(F(t))_{t=0}^{\infty}$ is monotonically increasing and bounded above by 1, leading to the conclusion that it converges.

Next we prove the sequence $(F(t))_{t=0}^{\infty}$ converges to 1 by examining its even-indexed subsequence, denoted as $(F(t))_{t=even}$. For this subsequence, the difference $F(t) - 1$ is given by:

$$\begin{aligned} F(t) - 1 &= \frac{(\sum r_i(t-1))^2 - n \sum r_i(t-1)^2}{n \sum r_i(t-1)^2 + 2n\alpha \sum r_i(t-1) + n^2\alpha^2} \\ &= (F(t-1) - 1) \cdot \frac{n \sum r_i(t-1)^2 + 2n\alpha \sum r_i(t-1) + n^2\alpha^2}{n \sum r_i(t-1)^2} \end{aligned}$$

By denoting the last term to $k(t)$ and noting that $F(t-1) = F(t-2)$ if t is even, we have

$$F(t) - 1 = (F(t-2) - 1) \cdot k(t)$$
$$\text{where } 0 < k(t) < 1 \text{ and } t \text{ is an even number}$$

Thus, for even t, the difference $F(t) - 1$ can be expressed as:

$$F(t) - 1 = (F(0) - 1) \cdot \prod_{j=2 \& j \in Even}^{t} k(j)$$

Since $0 < k(t) < 1$, the product of $k(t)$ converges to 0 as $t \to \infty$, leading to $F(t) - 1$ converging to 0. Therefore **the converged sequence $(F(t))_{t=0}^{\infty}$ converges to 1** as its subsequence $(F(t))_{t=even}$ converges to 1. □

This theorem can be interpreted from two perspectives. (i) Linear MIMD does not compromise fairness, and AI can improve fairness. Therefore, fairness is attainable with adequate AI operations across the flow lifecycle, which is guaranteed by Condition 2. (ii) Regardless of the initial states of multiple flows, they first achieve full bandwidth utilization via MI and AI, subsequently entering a cycle of AI and MD. This cycle exhibits convergence properties akin to the traditional AIMD algorithm.