Network Precision Time with Data-plane Timestamping and Hardware-backed Scheduling

David Zage[1], Srinivasan S Iyengar[1], Hector Blanco Alcaine[2]
Christopher S Hall[3], Sreedevi Joshi[4], Priyalee Kushwaha[3]
Intel Corporation*
[1]Santa Clara, CA, USA; [2]Munich, Germany; [3]Hillsboro, OR, USA; [4]Austin, TX, USA
{david.zage, srinivasan.s.iyengar, hector.blanco.alcaine, christopher.s.hall, sreedevi.joshi, priyalee.kushwaha}@intel.com

Abstract

Precision timing in Linux networking has largely been limited to control-plane timestamping, where timing is observed but not directly controlled, typically at rates below modern line speeds. Bringing timestamping into the data plane at line rate is challenging because driver behavior introduces variability that can undermine timestamp fidelity even with hardware assistance. We examine a hardware-centric approach that relocates clock synchronization functions from the control plane to the data plane within a SmartNIC architecture. The design applies real-time clock corrections using packet-derived timestamps and a lightweight two-bit synchronization protocol.

However, timestamp accuracy is not sufficient on its own. Linux can measure time, but it does not yet reliably enforce timing in the transmit path. We explore how `SO_TXTIME` shifts timing from observation to control by allowing applications to explicitly schedule packet transmission. While powerful, software-based scheduling struggles to scale to modern link rates and tight jitter bounds. To complete the model, we propose hardware-timed pause semantics that can be used for low-jitter transmit control. Together, data-plane timestamping, explicit transmit scheduling, and hardware-backed timing control move Linux networking toward deterministic, time-aware data planes.

Keywords

Precision Time Protocol (PTP), data-plane timestamping, SmartNIC, launchtime, `SO_TXTIME`, `TPAUSE`

Introduction

Modern networking environments increasingly require deterministic timing behavior. Applications such as 5G Radio Access Networks, autonomous systems, and distributed cloud computing rely on highly accurate clocks to coordinate operations across geographically distributed devices [8, 6, 13, 3, 10, 14]. Even small synchronization errors can result in degraded quality of service, inaccurate measurements, scheduling failures, and regulatory compliance issues. Consequently, maintaining precise synchronization within the network has become a fundamental requirement rather than an optional enhancement.

SmartNICs have emerged as powerful networking platforms capable of offloading packet-processing workloads from host CPUs. These devices often contain multiple accelerators, packet-processing engines, and network interfaces that share a common timing infrastructure. Traditional synchronization approaches utilize software-based control loops running on host processors. While flexible, these implementations introduce latency and jitter because every timing correction requires software intervention. Additionally, the multi-IP and multi-host nature of SmartNIC creates challenges in distributing time across multiple components and systems. Prior work has shown that careful time synchronization and traffic-control mechanisms can improve datacenter timing and latency behavior [3, 10, 14, 4, 15]. To overcome these limitations, we discuss how to move synchronization intelligence directly into the packet-processing data plane.

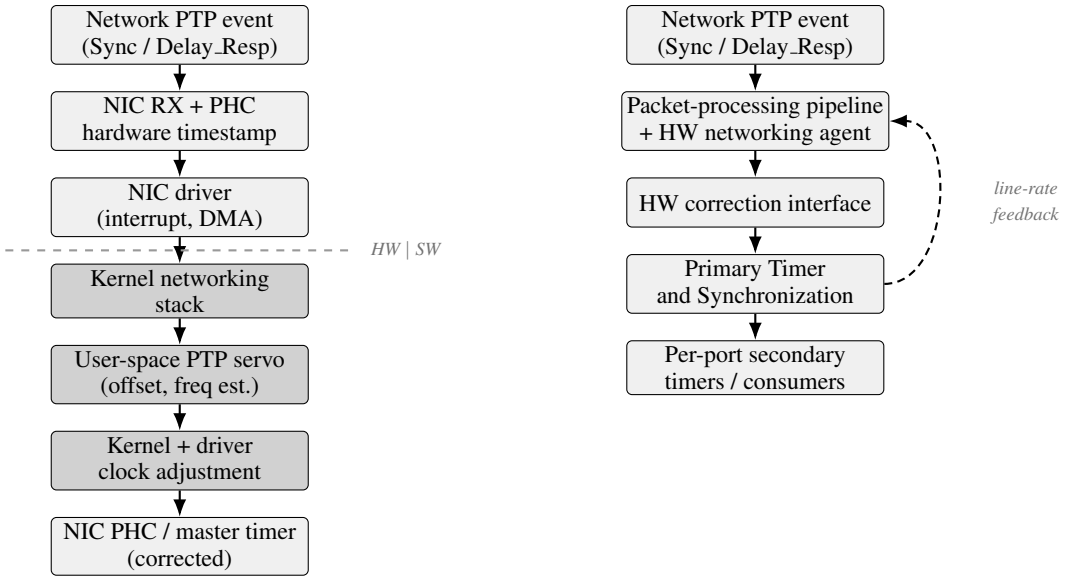
The paper then connects this synchronized data plane to packet transmission. We examine how `SO_TXTIME` exposes per-packet departure times in Linux and how `TPAUSE` can reduce transmit-path jitter when software scheduling alone is too variable. Finally, we summarize experimental results that quantify the effect of hardware-assisted timing on scheduling error and packet transmission jitter.

Limitations of Conventional Time Synchronization

In a conventional Precision Time Protocol (PTP) implementation, synchronization occurs through a control-plane feedback loop [5]. The host CPU runs a software servo algorithm that receives timing information from the network, calculates clock offsets, and determines the appropriate corrections. These corrections are then transmitted through NIC firmware or microcontrollers to adjust the device's master timer.

Although this methodology has been successfully deployed in many environments, several performance limitations become apparent in high-precision applications. Each correction must traverse multiple software layers, introducing latency into the control loop. Software execution is inherently nondeterministic because of interrupt handling, cache misses, and workload scheduling. Consequently, timing corrections themselves exhibit variable latency that degrades synchronization accuracy.

*The views, analyses, and conclusions presented in this paper are those of the authors and do not necessarily reflect the official positions, policies, or opinions of Intel Corporation.



(a) Control-plane timing path **(b) Data-plane timing path**

Figure 1: Timing correction paths compared: (a) the conventional control-plane loop through the host, and (b) the proposed data-plane loop kept entirely in hardware.

Another challenge arises in multi-host SmartNIC deployments where multiple systems share a common timing infrastructure. Arbitration mechanisms increase complexity and can degrade time synchronization quality. Figure 1 contrasts this conventional control-plane loop with the data-plane path proposed in the following section.

Data-Plane Time Synchronization Architecture Overview

We propose to relocate the clock steering logic from software into a hardware networking agent embedded directly within the packet-processing pipeline. Instead of relying on software-generated corrections, synchronization decisions are performed as timestamp packets traverse the network data path. This enables real-time clock adjustments at line rate while eliminating software round-trip delays.

At the heart of the SmartNIC timing capabilities is a Primary Timer and Synchronization (PTS) module, shown in Figure 2. Internally the PTS contains two-bit hardware synchronization registers (`i_sync`), a multiplexer that selects between the synchronization sources, and the primary timer for the SmartNIC. The multiplexer has two inputs. The first is a firmware-driven `i_sync` value that is programmed in response to the host software interface, and the second is a hardware-driven `i_sync` value produced by the SmartNIC networking agent in the packet data path. A firmware-controlled selection signal chooses which source drives the timers, making the transition from software steering (*i.e.*, the control plane) to hardware steering (*i.e.*, the data plane) explicit.

While not pictured in Figure 2, the PTS also distributes time values to all of the network subsystem agents across the device (at a different rate than the primary timer) as well as emitting a one-pulse-per-second signal for external alignment and validation. Each Ethernet port carries its own secondary IEEE 1588 timer used for timestamping. Every timer, primary and secondary alike, is clocked from the shared SmartNIC oscillator, so all distribution points share a common reference frequency. Because the same hardware `i_sync` signal drives the primary timer and every secondary timer, a single correction can be applied everywhere in parallel.

This architecture offers several benefits. Eliminating host participation reduces control loop latency and removes software-induced jitter. Utilizing hardware logic ensures deterministic behavior since synchronization decisions occur within the same predictable processing environment as packet forwarding operations. Corrections also can be applied simultaneously to both primary and secondary timers.

The model extends naturally to multi-host platforms. The first networking agent in the pipeline multiplexes traffic arriving over PCIe from all attached hosts and routes control-plane messages to the local microcontroller. Because the shared clock is steered from the data plane (via the SmartNIC networking agent) rather than by any single host, the design avoids the arbitration and ownership contention that arises when multiple hosts attempt to steer one timer through software.

Timer Design and Synchronization Mechanisms

The synchronization architecture uses a primary timer built on a 96-bit free-running counter. While the timer could run at a variety of speeds, we focus on an 800 MHz configuration, whose 1.25 ns clock period sets the effective timing

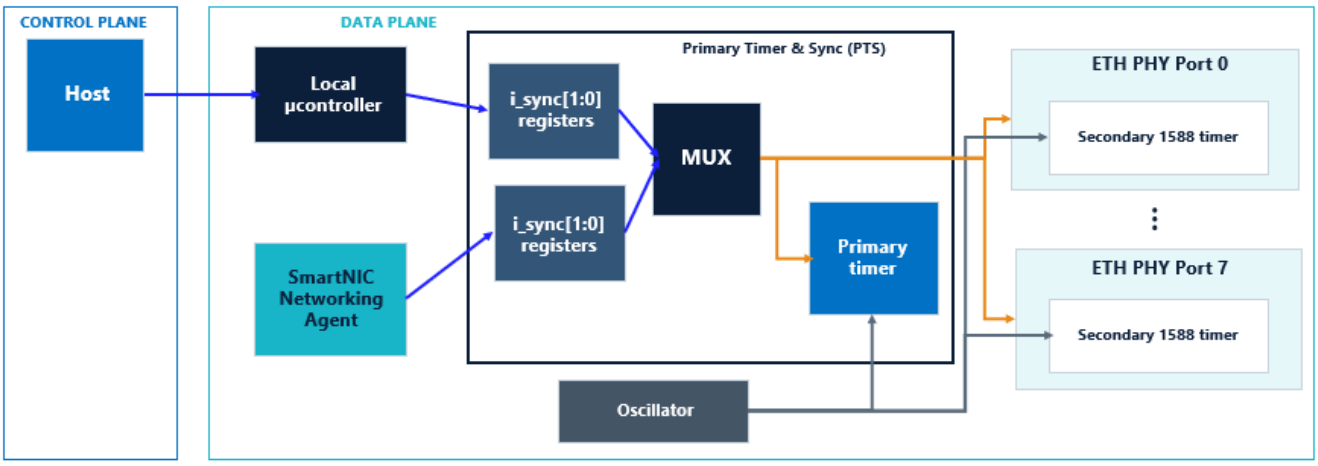


Figure 2: Data-plane time synchronization architecture.

resolution. The counter is a binary fixed-point value where the lower 32 bits hold the sub-nanosecond fraction, so bit 32 corresponds to one nanosecond. A separate 40-bit increment-value field (INCVAL) is added to the counter on each clock tick and serves both increment and decrement operations, with bits 32 through 39 forming the nanosecond adjustment field where incremental synchronization corrections are applied. Together these fields support high-resolution timing while keeping the per-step adjustment logic small.

A programmable time-step-size register controls the magnitude of each adjustment. The register is instantiated in the primary timer and at every distribution point so that coarse and fine behavior stay consistent across the device. Coarse step sizes allow rapid convergence when a large offset exists, while fine steps hold nanosecond-level lock in steady state. During normal operation the step size is set to a single nanosecond. The step size also bounds the achievable slew rate. Assuming the PTS distributes corrections to the subsystems at 100 MHz, single-nanosecond steps give a slew of $1\,\text{ns} \times 100\,\text{MHz} = 0.1$ seconds per second, sufficient to converge a one-second offset in about ten seconds.

Communication between synchronization components occurs through a simple two-wire interface known as `i_sync`. The protocol uses 00 and 11 to indicate no change, 01 to increment time, and 10 to decrement time. The simplicity of this protocol minimizes hardware cost and facilitates distribution throughout the SmartNIC.

The synchronization process follows initialization, handoff, and runtime phases, as illustrated in Figure 4. During initialization a secure local host brings the PTS out of reset, sets the initial time base, and programs the step size, driving the timer through the software `i_sync` source. After the data plane is brought up, firmware transfers control to a networking agent. During runtime, a network timestamp packet arrives periodically (*e.g.*, every hundredth packet). The receiving port records a local timestamp using its secondary IEEE 1588 timer and carries it alongside the network timestamp in the packet header. The networking agent extracts both values, compares them, and issues an increment or decrement, together with a coarse or fine step selection, over the `i_sync` interface to the primary and secondary timers in parallel. This runtime phase forms a closed correction loop on the data path, shown in Figure 3, that runs continuously in hardware without host involvement.

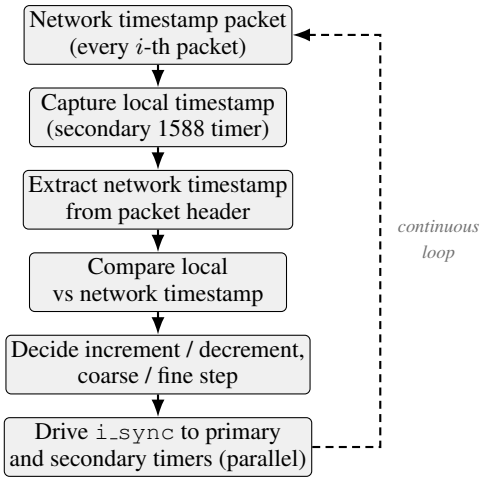


Figure 3: Runtime clock-correction loop on the data path.

Generalizing to Programmable SmartNICs

While the architecture described above is expressed in terms of a specific timer construction and two-bit hardware synchronization interface, the approach generalizes to any SmartNIC that provides the following set of capabilities:

1. A programmable pipeline agent (*e.g.*, a P4 pipeline stage, an eBPF or XDP offload, or fixed-function logic) that compares a network timestamp against a locally captured one at line rate.
2. Ingress hardware timestamping that captures the local time of arrival and carries it alongside the network timestamp.

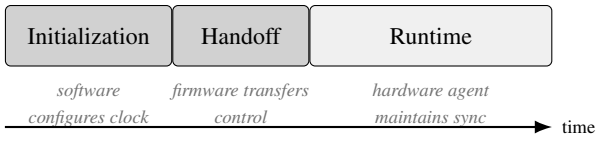


Figure 4: Synchronization phases: initialization, handoff, and runtime.

3. A primary timer that accepts corrections over a low-latency hardware path reachable from the pipeline.
4. A means to distribute the same correction to secondary timers that share a common reference frequency so all timing endpoints stay phase aligned.
5. An ownership handoff that lets software steer the timer during bring-up and the data-plane agent steer it at runtime.

The specific timer width, the correction encoding, and the sampling policy are implementation choices. The essential requirement is that the steering path be reachable from the data plane with bounded latency, since a device whose timer can only be adjusted through a slow or jittery path (*e.g.*, a firmware or mailbox path) would reintroduce the very host round trip this design removes.

Precision Transmission Scheduling Using `SO_TXTIME` and `TPAUSE`

Once time is synchronized on the network controller, it becomes a resource the host can use to place packets on the wire at a chosen instant, rather than merely to observe when they departed. This requires the host clock and the device clock to share a common time base. We assume the host is synchronized to network time, optionally using PCIe Precision Time Measurement (PTM) to bound the host-to-device offset [16].

Linux already exposes per-packet departure times, commonly referred to as "launchtime", through the `SO_TXTIME` socket option [17, 11, 12]. An application tags each packet with a desired transmission timestamp and a scheduler such as the Earliest TxTime First (ETF) qdisc holds the packet until its departure time before releasing it to the driver. If a NIC does not provide launchtime offload, the release of packets is driven by a software timer and is therefore exposed to the full range of host-induced variability including interrupt latency, driver execution time, and memory-management stalls. On an unmodified `ixgbe` driver on an Intel® Ethernet Converged Network Adapter X550 we measured approximately 150 microseconds of transmission jitter for cyclic traffic[1].

This variability is dominated by the final interval between deciding to transmit and actually issuing the transmit-tail write. To reduce this interval, we propose to use the `TPAUSE` instruction. `TPAUSE` places the logical processor into an implementation-defined optimized wait state until the processor timestamp counter (TSC) reaches a supplied deadline, after which execution resumes at the following instruction [7].

[1]Due to timing constraints, our investigation focused on the Intel® Ethernet Converged Network Adapter X550 and we subsequently provide generalization to SmartNICs.

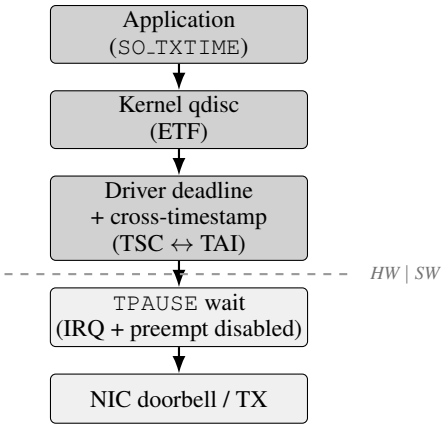


Figure 5: Precision transmit scheduling pipeline for launchtime (`SO_TXTIME`) leveraging TPAUSE.

Unlike an `hrtimer`, it does not rely on an interrupt being delivered and serviced to wake the core, so the wake-up instant is governed by hardware rather than by the scheduler, yielding deterministic release with very low jitter.

Integrating `TPAUSE` into the transmit path replaces the software-timer release with a three-step sequence applied at the driver's transmit-tail update.

1. A hardware cross-timestamp correlates the device clock with the local TSC, allowing the packet's departure time to be converted into a TSC deadline.
2. The driver enters a critical section with preemption and interrupts disabled and bounds the deadline against the current TSC, rejecting values already in the past or unreasonably far in the future.
3. `TPAUSE` idles the core to the deadline, and on wake-up the driver immediately performs the memory-mapped tail write that kicks the NIC.

Since the busy-wait is short and the tail write follows the wake-up with no intervening scheduling decision, the departure time is enforced with higher fidelity. Figure 5 summarizes the resulting pipeline for improving launchtime determinism.

Experimental Results

In order to understand the basic limitations of `TPAUSE`, standalone `TPAUSE` testing on an Intel® Core™ i5-13600HE Processor demonstrated scheduling error of only a few hundred nanoseconds. These numbers are consistent with other characterizations of `TPAUSE` wake-up latency, which report tens to hundreds of nanoseconds depending on the requested idle state and the underlying microarchitecture [9].

We integrated `TPAUSE` into the `ixgbe` driver, as shown in Listing 1. As shown in Figure 6, this integration reduced packet transmission jitter from roughly 150 microseconds to approximately 1 microsecond. While there are additional sources of jitter, such as MMIO delays, driver scheduling, and ring flushes, these are outside the scope of this work.

Listing 1: Example `TPAUSE`-gated transmit-tail update in the `ixgbe` driver.

```
void ixgbe_xdp_ring_update_tail(struct
    ixgbe_ring *ring)
{
    /* cycle alignment via x-timestamp */
    get_snapshot_ns(&xts_tsc1, &xts_real1);
    next_ns    = next_txtime_ns(xts_real1);
    next_ticks = next_txtime_ticks(next_ns);

    preempt_disable();
    local_irq_disable();

    if (next_ns < xts_real1)
        goto tail_bump; /* too late */
    if ((next_ns - xts_real1) > 15000)
        goto tail_bump; /* > 15us away */
    tpause(next_ticks);
tail_bump:
    wmb();
    /* kick DMA */
    writel(ring->next_to_use, ring->tail);

    local_irq_enable();
    preempt_enable();
}
```

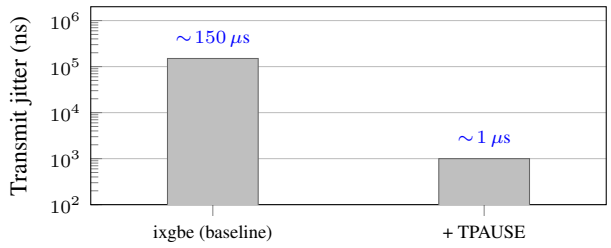


Figure 6: Effect of `TPAUSE` on packet transmission jitter.

Additionally, testing over millions of transmission cycles, we found the `TPAUSE`-based approach was robust to system load, with no drift over pause durations and no measurable increase in jitter when the host was under heavy CPU or I/O load. We also did not find degradation for `TPAUSE` lengths of less than 25 microseconds. The results indicate that `TPAUSE` can provide a reliable mechanism for enforcing precise transmission timing in high-performance networking applications and validate the hypothesis that hardware-assisted synchronization and scheduling can dramatically outperform software-only implementations.

Next Steps and Open Questions

The previous discussion of launchtime and `TPAUSE` was based on a single `ixgbe`-class device. We see several directions in which this work could be generalized, and we present them here to gather feedback on which, if any, are worth pursuing upstream.

Generalizing hardware-timed transmit to SmartNICs

Our `TPAUSE` integration modified the transmit-tail update. The same three-step pattern applies to any driver with an equivalent transmit doorbell, that is, the MMIO tail write that kicks the NIC:

1. use the cross-timestamp path to obtain a correlated device and host time pair,
2. compute a deadline in the processor timebase, and
3. gate the tail write until the deadline is reached.

A natural next step is to factor this into a small, reusable helper rather than per-driver open coding. An open question is whether the community would prefer such logic exposed through an existing abstraction (*e.g.*, the `SO_TXTIME` and ETF offload path) rather than a driver-local mechanism.

Applying the approach to multi-host SmartNICs

Devices built around the IDPF model already expose a hardware cross-timestamp path and, on some platforms, a device-side Earliest Departure Time (EDT) scheduler. This raises a design choice between host-driven `TPAUSE` gating and device-offloaded launchtime. The former needs no new silicon and works today, while the latter scales past a busy-polled core. We are interested in whether the community sees value in supporting both, with a clear policy for when each is selected. There may also be room for hybrid approaches, such as a device-side scheduler that gates the tail write with `TPAUSE` when the deadline is imminent.

Time enforcement for virtualized tenants

On a multi-host SmartNIC, a virtual function does not run PCIe PTM directly. However, a control-plane-mediated cross-timestamp can still deliver a correlated device and host time pair to a tenant. If a tenant has access to an accurate cross-timestamp, then hardware-timed transmit gating becomes possible inside a guest, which is not achievable on today's conventional NICs. The open question is what the kernel must expose to generalize this beyond a single device, namely a common interface through which any NIC can deliver a guest-visible cross-timestamp and accept a time-gated transmit request. A further question is what isolation guarantees such an interface would require.

Standardizing hardware-timed transmit semantics

`TPAUSE` gave deterministic wake-up semantics, but the surrounding contract (who disables preemption and interrupts, how far in advance the wait may be armed, and how to bound the busy-wait) was driver-specific. We would like feedback on whether a common, documented set of semantics for hardware-timed transmit would be useful, and where such a contract should live.

It should be noted that the `TPAUSE` gating pattern is not specific to Intel. Arm provides `WFET` and `WFIT` (introduced in Armv8.7-A and mandatory from Armv9.2-A) which wake the core at an absolute counter deadline similar to how `TPAUSE` waits on the TSC [2]. AMD offers a timed `MWAITX` [1]. Any qdisc and driver logic should be able to

bind to whichever timed-wait primitive a given platform exposes.

Conclusion

This paper examines how to better synchronize and utilize time. First, we show how a SmartNIC architecture can move time synchronization from the control plane into the packet-processing data plane. By employing a dedicated hardware networking agent, a lightweight two-bit synchronization protocol, and a high-resolution primary timer, the architecture achieves nanosecond-class synchronization without continuous host software involvement. Importantly, the design scales naturally to multi-host environments, avoiding ownership contention and arbitration challenges. It is also generalizable to SmartNICs with programmable data planes.

Next we demonstrate how time synchronized across a host and its data plane can be used to enforce, and not just observe, packet timing. Linux launchtime (`SO_TXTIME`) paired with the ETF qdisc already expresses per-packet departure times, but software release leaves the final interval before the transmit-tail write exposed to host jitter. Gating that tail write with the `TPAUSE` instruction reduced periodic transmission jitter on an `ixgbe` device by more than two orders of magnitude, from roughly 150 microseconds to about 1 microsecond. The same gating pattern is not tied to a single vendor, as equivalent timed-wait primitives exist on other platforms.

As networking applications continue to demand greater determinism and lower latency, the integration of data-plane clock synchronization and hardware-assisted scheduling represents a practical path forward. We hope the directions outlined in Section prompt discussion on whether these capabilities belong in expanded SmartNIC platforms and how best to bring them upstream.

AI Assistance Disclosure

The authors used Claude (Anthropic) as a drafting and editing assistant for prose, citation hygiene, and structural review. All technical claims, attributions, and editorial judgements are the authors'.

References

[1] Advanced Micro Devices. 2024. AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions. Advanced Micro Devices, publication 24594. `MONITORX`/`MWAITX`. Available at `https://www.amd.com/en/support/tech-docs`.

[2] Arm Limited. 2024. Arm Architecture Reference Manual for A-profile Architecture. Arm Limited, document ARM DDI 0487. `WFET`/`WFIT` (`FEAT_WFxT`). Available at `https://developer.arm.com/documentation/ddi0487/latest`.

[3] Geng, Y.; Liu, S.; Yin, Z.; Naik, A.; Prabhakar, B.; Rosenblum, M.; and Vahdat, A. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, 81–94. USENIX Association.

[4] Grosvenor, M. P.; Schwarzkopf, M.; Gog, I.; Watson, R. N. M.; Moore, A. W.; Hand, S.; and Crowcroft, J. 2015. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, 1–14. USENIX Association.

[5] IEEE. 2020a. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE Std 1588-2019.

[6] IEEE. 2020b. IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications. IEEE Std 802.1AS-2020.

[7] Intel Corporation. 2026. Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation. Available at `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[8] ITU-T. 2022. Precision Time Protocol Telecom Profile for Phase/Time Synchronization with Full Timing Support from the Network. ITU-T Recommendation G.8275.1/Y.1369.1.

[9] Kuns, M.-C.; Tröpgen, H.; and Schöne, R. 2025. An Analysis of User-space Idle State Instructions on x86 Processors. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, 232–239. ACM.

[10] Li, Y.; Kumar, G.; Hariharan, H.; Wassel, H.; Hochschild, P.; Platt, D.; Sabato, S.; Yu, M.; Dukkipati, N.; Chandra, P.; and Vahdat, A. 2020. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 1171–1186. USENIX Association.

[11] Linux man-pages project. 2026a. socket(7) Linux Manual Page. Linux man-pages project. Available at `https://man7.org/linux/man-pages/man7/socket.7.html`.

[12] Linux man-pages project. 2026b. tc-etf(8) Linux Manual Page. Linux man-pages project. Available at `https://man7.org/linux/man-pages/man8/tc-etf.8.html`.

[13] Mittal, R.; Lam, V. T.; Dukkipati, N.; Blem, E.; Wassel, H.; Ghobadi, M.; Vahdat, A.; Wang, Y.; Wetherall, D.; and Zats, D. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 537–550. ACM.

[14] Namyar, P.; Li, Y.; Wang, W.; Dukkipati, N.; Yap, K.; Gong, J.; Chen, C.; Gao, P.; Ray, D.; Kumar, G.; Ma, Y.; Govindan, R.; and Vahdat, A. 2025. Firefly: Scalable, Ultra-Accurate Clock Synchronization for Datacenters. In *Proceedings of the ACM SIGCOMM 2025 Conference*, 434–452. ACM.

[15] Saeed, A.; Dukkipati, N.; Valancius, V.; Lam, V. T.; Contavalli, C.; and Vahdat, A. 2017. Carousel: Scalable

Traffic Shaping at End-Hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 404–417. ACM.

[16] Stanton, K. B.; Hall, C. S.; Zage, D.; Byagowi, A.; and St. James, J. 2024. Precision Time in the Last Centimeters for Distributed Applications. In *Proceedings of the 2024 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication*, 1–6. IEEE.

[17] The Linux Kernel Developers. 2026. Timestamping. Linux kernel networking documentation. Available at `https://docs.kernel.org/networking/timestamping.html`.