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

David Zage · Srinivasan S. Iyengar · Hector Blanco Alcaine · Christopher S. Hall · Sreedevi Joshi · Priyalee Kushwaha

MOTIVATION

Why time synchronization is hard on a multi-host NIC

- **A SmartNIC keeps a common master timer, distributed to many accelerator blocks across the SoC**
- **That timer must stay frequency- and phase-aligned with the network at every distribution point**
  - Timestamping, traffic shaping, latency measurement all depend on it
  - Enable nanosecond accuracy at the PHY (e.g., 5G RAN)
- **Sync sources are diverse: 1588, GPS, SyncE, …**
- **Now multiply by N hosts sharing one NIC, who owns the clock?**

What's at stake

- **Tight timing error budgets can impact domain compliance**
- **Phase drift across ports → inconsistent measurements**
- **Host-driven correction → latency + jitter in the loop**

TODAY

The conventional path: control-plane time synchronization

- The primary timer is steered by host software

The limits

- Every correction takes a trip to the host → added latency and jitter in the control loop
- Hard to hit low ns phase alignment at the PHY when the decision lives off-datapath
- Multi-host arbitration of a single shared clock adds complexity

THE KEY IDEA

Move clock steering into the data plane.

A hardware **network IP block** inside the packet pipeline corrects the timer in real time packet by packet at line rate.

No host round trip

Correction happens on-datapath, in hardware

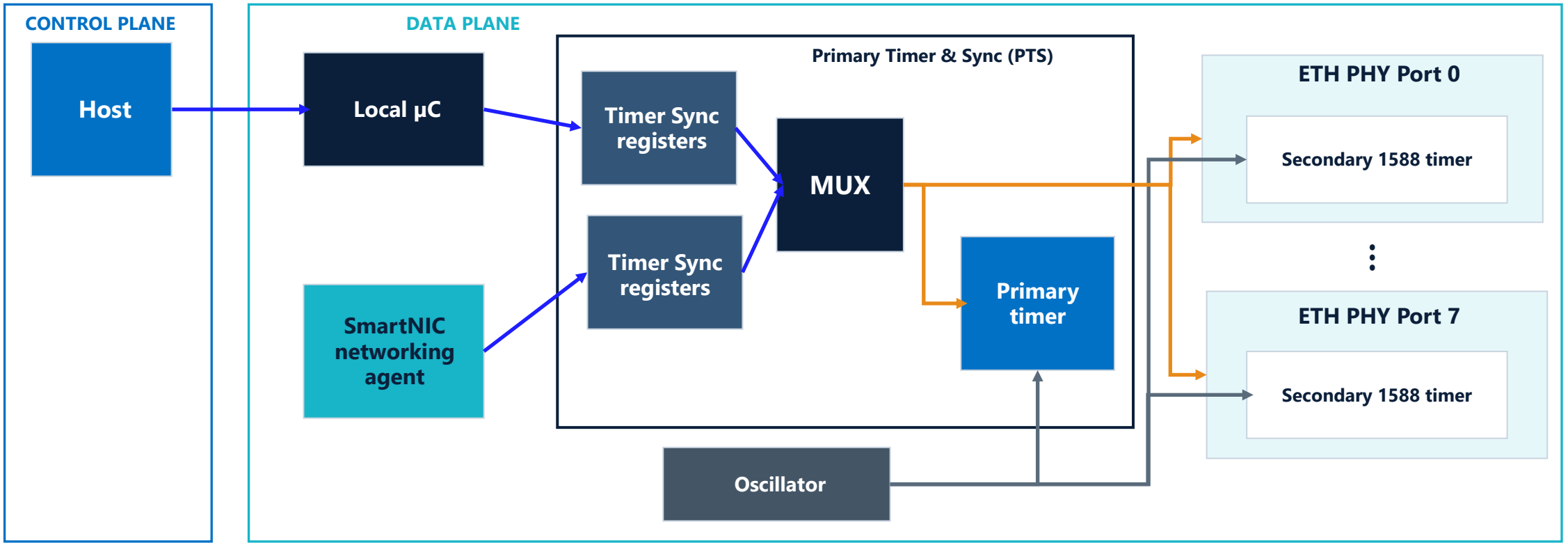
Nanosecond granularity

Fine/coarse steps via a programmable register

Parallel updates

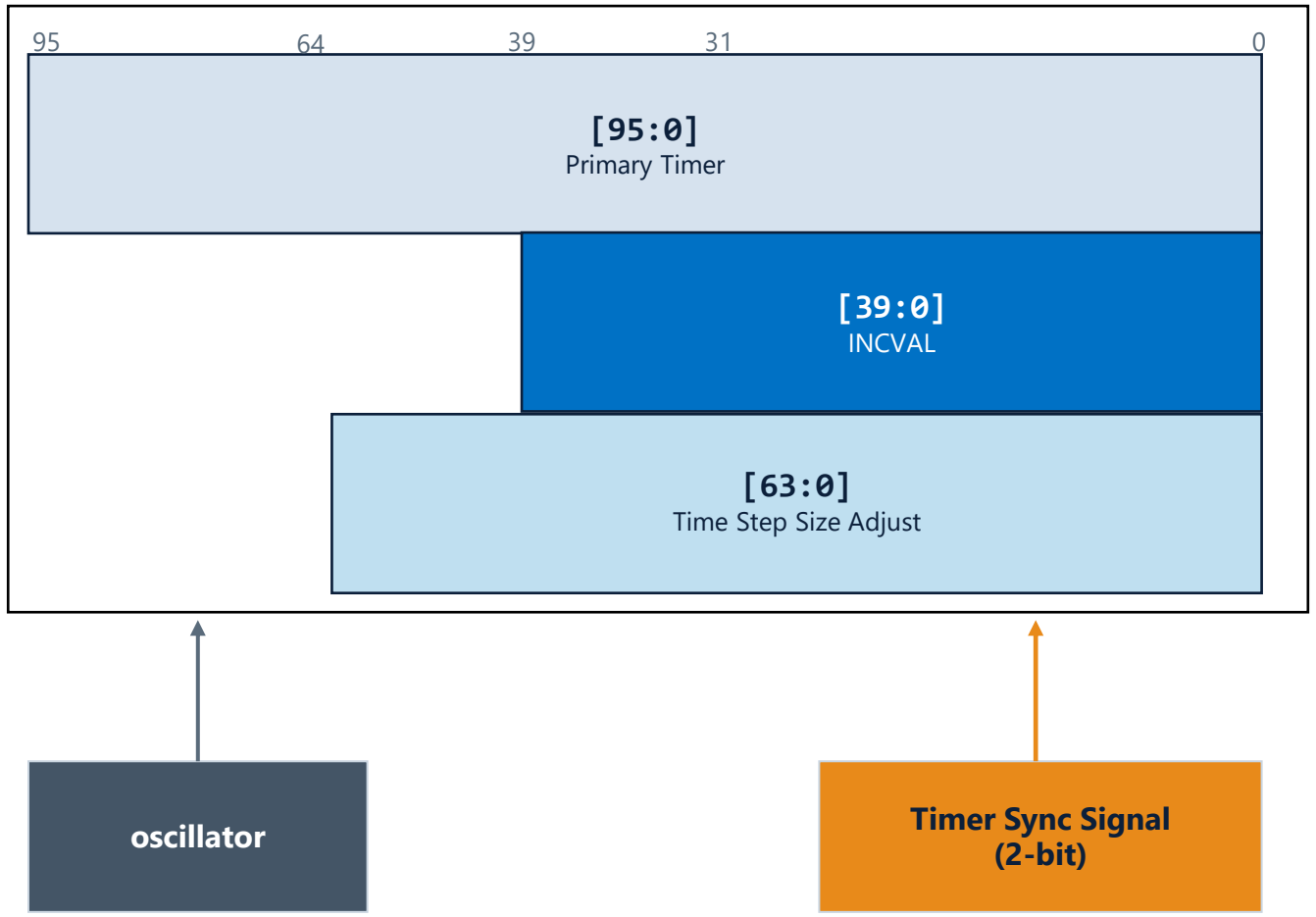
Primary + per-port secondary timers together

Primary timer and synchronization



Inside the primary timer (96-bit)

- **96-bit free-running counter, clocked by an oscillator**
  - 800 MHz → toggling bit 32 = 1 ns
- **40-bit increment value field (INCVAL)**
  - Used for both increment and decrement
- **Time-step-size register sets fine vs coarse**
- **Emits 1 PPS & distributes 64-bit time across the network subsystem IPs**



↓ 96-bit time value → 1 PPS + 64-bit NSS distribution

MECHANISM

The 2-bit timer sync protocol

- All timer corrections travel on a 2-bit interface
- Tiny, cheap, and routable to every distribution point on the SoC
- Same signal steers the primary and per-port secondary timers
- Driven by software (init) or by the network IP (runtime)

Timer Sync Signal	Action
00	no change
01	increment
10	decrement
11	no change

MECHANISM

Step size and slew rate

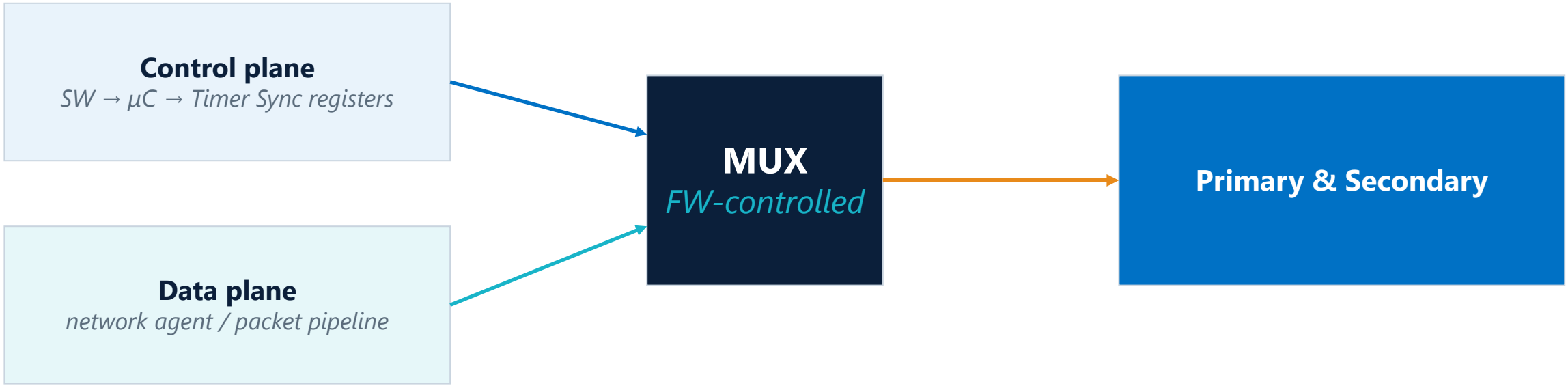
- A programmable time-step-size register sets fine vs coarse granularity (when needed)
- Implemented in the primary timer and at each distribution point

Worked example: slew bound

- 100 MHz adjust clock → up to 100M ±1 ns ops / second
- → maximum slew of ±0.1 second per second
- A 1 s offset corrects in ≈ 10 seconds
- Coarse steps converge fast; fine steps hold ns-level lock

MECHANISM

The handoff: a mux selects who owns the clock



- Firmware flips the mux: software steers during bring-up, hardware takes over for runtime
- The selection is the explicit control-plane → data-plane handoff point

OPERATION

Lifecycle: initialization → handoff → runtime

1 Initialization

- Secure local host brings PTS out of reset
- Reads time-of-day (network)
- Writes initial timer value + step size via µC
- Steps the timer using software-driven Timer Sync Signal

2 Handoff

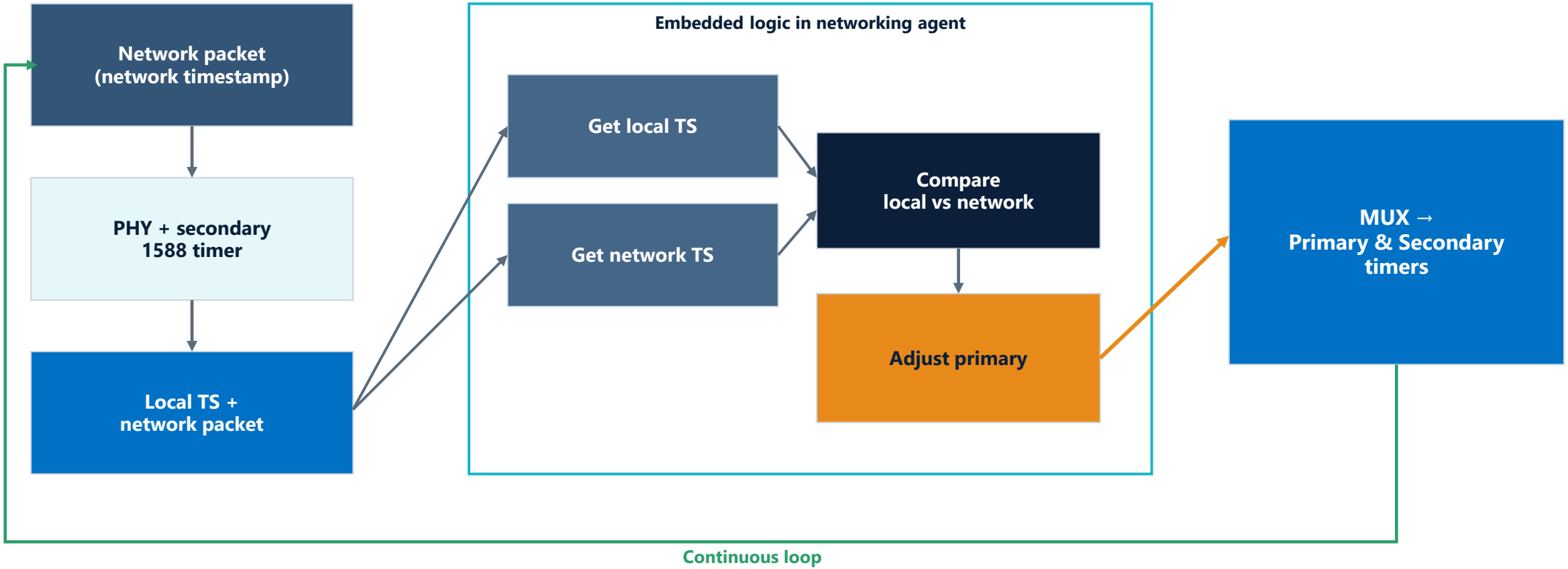
- Control of the primary timer transfers to the data-plane synchronization logic
- Mux flips to the data-plane source
- Software drops out of the steering loop

3 Runtime

- Network IP corrects the timer per timestamp packet
- Increment / decrement decided on-datapath
- Primary + secondaries updated in parallel
- Loop runs continuously at line rate

OPERATION

Runtime correction loop on the datapath



GENERALIZATION

Programmable SmartNIC implementations

- **Programmable pipeline logic**
  - P4 stage, eBPF/XDP offload, or fixed-function hardware
- **Ingress hardware timestamping**
  - Capture arrival time within the packet metadata
- **Low-latency timer steering path**
  - Data plane must be able to adjust the primary timer without a slow firmware/mailbox round trip
- **Shared time distribution**
  - Apply the same correction to all secondary timing endpoints that share a common reference clock
- **Configurable steering logic**
  - Software steers during initialization; data-plane logic steers at runtime

PAYOFF

Why data-plane steering

Lower loop latency

No software round trip; correction lives on the datapath

Nanosecond accuracy

~1 ns resolution at the PHY for RAN-grade timestamping

Deterministic

Hardware decisions remove host-induced jitter

Scalable

Inexpensive 2-bit timer sync protocol fans out across the SoC

Flexible convergence

Coarse steps lock fast, fine steps hold

Clean ownership model

SW inits, HW runs

From observation to control

CONTROL

SO_TXTIME: scheduling the transmit path

- **SO_TXTIME sets a per-packet launch time (Earliest Departure Time) in skb->tstamp**
  - Paired with a clockid; the app says when a packet should hit the wire
- **Shifts timing from observing arrivals to controlling departures**
- **This is the "enforce" half — Linux can express intent to transmit at time T**

Observation → Control

- **Before: hardware timestamps tell you WHEN a packet left**
- **After: the app tells the stack WHEN to send it**
- **But intent ≠ precision — see the next slide**

PROBLEM

Software scheduling can't hold the deadline

- **ETF leans on hrtimer wakeups + the normal TX softirq / driver path**
- **Every layer injects variability before the packet reaches the wire:**
  - OS scheduling & IRQ delivery latency
  - Driver service tasks
  - MMIO write to the PHY; host↔NIC clock correlation may be limited without PTM
- **Software can express the deadline but not reliably meet it**

Periodic Tx jitter, unmodified ixgbe

~150 µs

of cycle-to-cycle jitter at line rate

- **Deadlines could be missed by orders of magnitude**
- **Tail dominated by OS + driver scheduling**
- **Target: sub-µs, deterministic**

IDEA

Hardware-timed pause: the TPAUSE instruction

- **TPAUSE: suspend the core until the TSC reaches a target value**
- **Deterministic wakeup with minimal jitter**
- **Idea: align the Tx kick by waiting in hardware, not in software**
  - Busy-poll app stays hot; the core idle-waits to the exact tick
- **Standalone TPAUSE testing achieved scheduling error of only a few hundred nanoseconds**
- **Validated nonstop for one month across pause durations**

~10s of µs
OS sleep / hrtimer wakeup jitter

~100s of ns
TPAUSE wakeup to TSC target

Why it works: the wakeup deadline lives in the CPU which has been synchronized to the network time, gated by the TSC.
- no softirq, no scheduler, no time rounding between intent and action.

EXAMPLE IMPLEMENTATION

Testing TPAUSE in ixgbe

- **Modified ixgbe_xdp_ring_update_tail (AF_XDP / xdpsock busy-poll)**
- **Cross-timestamp snapshot (TSC + realtime) → next_ns, next_ticks**
- **Enter protected section: preempt_disable + local_irq_disable**
  - Guard: deadline in the past → bump now
  - Guard: more than 15 µs away → bump now (safety margin)
  - Else tpause(next_ticks), then wmb + writel(tail) to kick DMA

```
void ixgbe_xdp_ring_update_tail(struct ixgbe_ring *ring)
{
    /* deadline alignment via cross-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);            /* HW idle-wait */
tail_bump:
    wmb();
    writel(ring->next_to_use, ring->tail);  /* kick DMA */

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

RESULTS

Hardware-timed transmit results

ixgbe Tx jitter

150 µs → <1 µs

TPAUSE-gated tail bump

Standalone TPAUSE

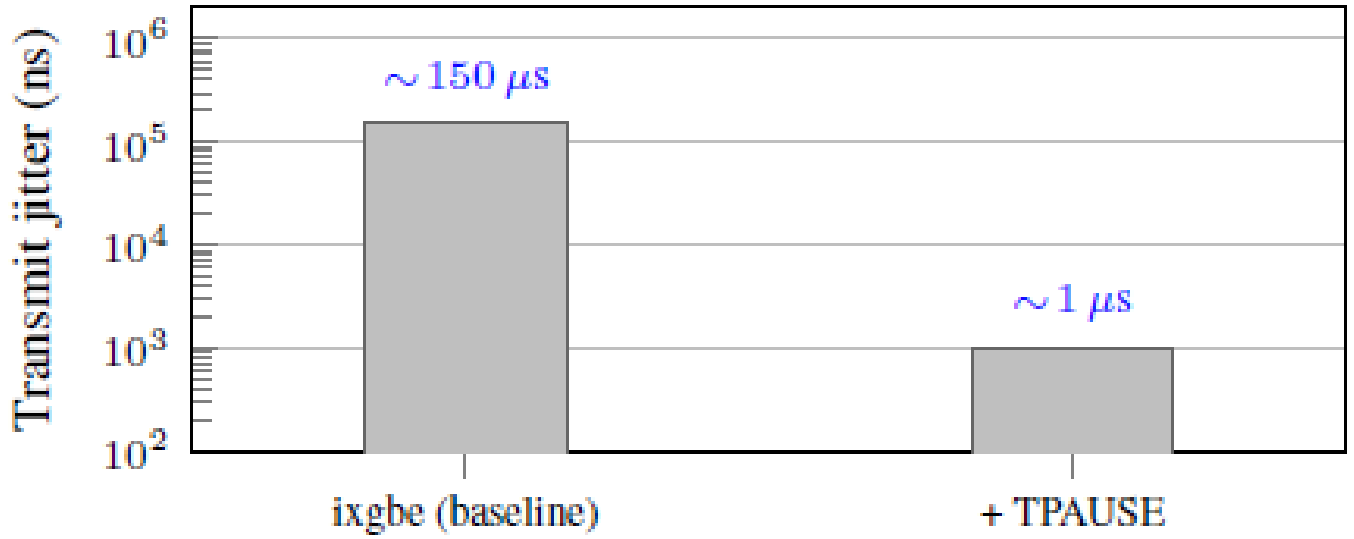
~200 ns scheduling error

kernel scheduling benchmark

Stability

1 month nonstop

no drift across pause durations



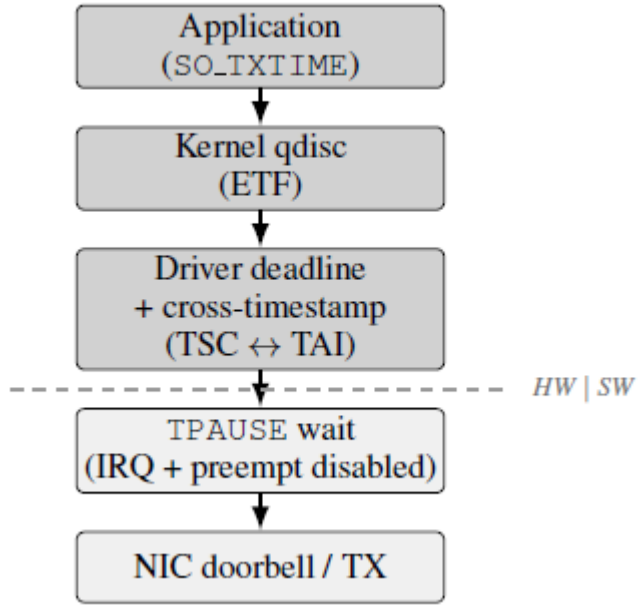
WHAT'S NEXT

Generalize TPAUSE-timed transmit

Make the TPAUSE pattern driver-agnostic

- Portable pattern: cross-timestamp → TSC deadline → gate the MMIO tail write
- Works on any driver with a Tx doorbell — tested in ixgbe

Q: Factor it into a reusable kernel helper, or fold it into the existing SO_TXTIME / ETF offload path?



OPEN QUESTIONS

Multi-host SmartNICs

Host TPAUSE gating vs. device-side scheduling

- IDPF-class devices already expose a HW cross-timestamp (some add a device-side EDT scheduler)
- Host-driven TPAUSE works today, but consumes a busy-polled core
- Device-offloaded launch time scales, but needs the hardware

Q: Should both be supported with a clear policy for when each is selected?

OPEN QUESTIONS

Standardize the semantics

One contract, many timed-wait primitives

- TPAUSE gave deterministic wakeup, but our integration was driver-specific
- Who disables preempt / IRQs? How far ahead may the wait be armed? How to bound the busy-wait?

Q: Do we need a common interface for hardware-timed Tx? If so, where should it live?

Q: Should qdisc/driver logic be bound to whichever timed-wait primitive the platform exposes?

TAKEAWAYS

Three things to remember

1

Discipline the clock in the data plane

A hardware network agent steers primary + secondary timers with a 2-bit timer sync signal providing ns-class time without a host round trip.

2

Move from observing time to enforcing it

SO_TXTIME expresses a launch time, but software scheduling alone can induce undesired artifacts.

3

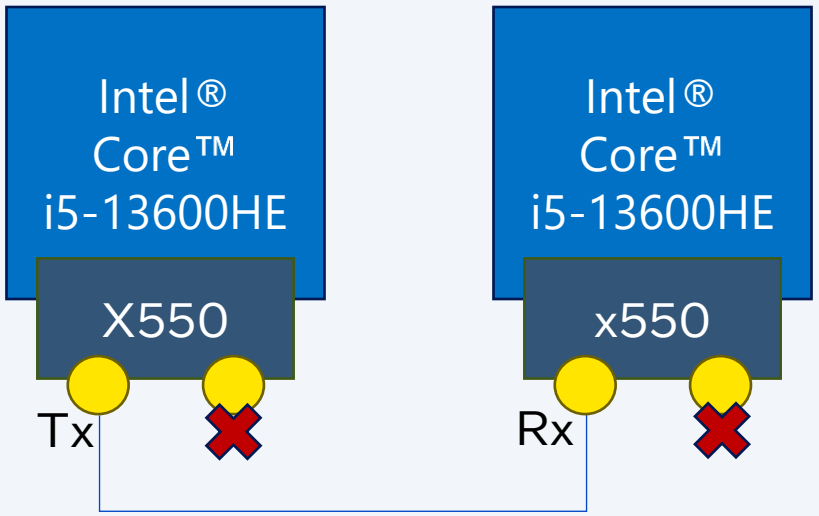
Hardware-timed pause closes a gap

TPAUSE can gate Tx operations based on synchronized time. Time-aware data planes can be built today.

Questions?

HW-timed transmit test

Current measurement setup



Hardware

- Intel® Core™ i5-13600HE Processor
- X550 located on dedicated PEG port
  - PF (SR-IOV disabled)
  - 10G Link Speed

BIOS

- TCC Mode on, ASPM off, root port clock and power gating off

OS

- Debian 12 / upstream 6.8.2-rt11
- Cores 2 and 3 fully isolated (P Cores) via cmdline and also cpusets
- PREEMPT_RT enabled

Benchmark

- RTCTestbench TSNHigh on core 2 (P), logging on core 3 (P)
- Hackbench Kernel Stress Test for Load