Towards Cost-Effective Real-Time High-Throughput End Station Design for Time-Sensitive Networking (TSN)

Chuanyu Xue
University of Connecticut
chuanyu.xue@uconn.edu

Tianyu Zhang
University of Connecticut
tianyu.zhang@uconn.edu

Song Han
University of Connecticut
song.han@uconn.edu

Abstract

Time-Sensitive Networking (TSN) technology has been increasingly deployed in mission- and safety-critical industrial applications to achieve high throughput and deterministic communications. To provide stringent timing guarantee, TSN requires that network devices follow a predefined communication schedule for real-time end-to-end packet processing, involving both TSN bridges and end stations. Extensive efforts have been devoted on the TSN bridge design in the literature. Achieving TSN compatibility on the end stations (especially on COTS hardware), however is challenging due to their constrained resources. To fill this gap, this work presents a software-based open-source TSN end station design that i) enables ultra-low latency and nanosecond-level transmission accuracy based on DPDK, and ii) employs a novel multi-core scheduling framework to boost the throughput of real-time TSN traffic. Our proposed solution leverages existing COTS hardware and thus is more generic and cost-effective compared to existing hardware-centric solutions. We validate our design by developing a prototype end station and incorporating it in a real-world TSN testbed. Our extensive experiments demonstrate the efficiency and effectiveness of our design compared with other state-of-the-art solutions.

1 Introduction

Time-Sensitive Networking (TSN) has been increasingly adopted in industrial Internet-of-Things (IIoT) systems to provide real-time communications for industrial applications [1]. As a set of open standards extended from Ethernet, TSN aims to provide both deterministic communications and much higher throughput compared with traditional fieldbus [2]. To enable stringent real-time communications, TSN standard requires that all the network devices, including both TSN bridges and end stations (ES) [3], are able to support precise packet forwarding according to the pre-defined schedule.

Many commercial TSN bridge products (e.g., TTTech Evaluation Board [4] and Cisco Industrial Ethernet 4000 Switch [5]) can support real-time and high-throughput (e.g., 1 Gbps) traffic with microseconds-level precision. However, due to the inefficiency of general CPU and unpredictable bus contention [6], the design of real-time TSN-compatible ES is much more challenging and remains an open problem. In the literature, real-time TSN ES designs can be classified into *hardware-based* design and *software-based* design. The former approaches (e.g., [6–9]) implement TSN ES on a dedicated hardware platform, e.g., Field Programmable Gate Arrays (FPGA) and system-on-chip (SoC), which can generally support higher throughput and faster scheduling actions. However, hardware-based solutions are expensive and inflexible. They require long development period and introduce operational complexity in production environment. The software-based solutions (e.g., [10–12]), which implement the TSN ES based on general Linux kernel, are more cost-effective and flexible. However, they suffer from several limitations, including high latency and low throughput, based on our measurement results from a real-world TSN testbed.

In this work, we address these limitations by presenting a software-based open-source approach based on Data Plane Development Kit (DPDK) [13] to design real-time and high-throughput TSN ES on COTS hardware. Our solution enables nanosecond-level packet transmission accuracy on DPDK by offloading the scheduling duty from CPU to network interface card (NIC), and adapting high precision event timer (HPET) to reduce CPU resource contention. To improve the throughput, we propose a novel multi-core stream allocation framework which guarantees that the traffic flows allocated on the same CPU core do not conflict with others.

We validate our design by developing a prototype TSN ES and incorporating it into a real-world TSN testbed. The evaluation on the testbed shows that our proposed solution can support periodic traffic up to 50 kHz with negligible jitter (16 ns) and 0% packet loss ratio. The transmission delay is at least 2 times lower (32 μs in the worst case) and 8.7 μs on average compared to existing software-based solutions. Through extensive simulation-based evaluation, we further demonstrate that the proposed multi-core stream allocation framework can achieve 2.4 times higher feasibility ratio compared with that on a single-core platform. Our contributions of this work are summarized as follows:

1) We propose a software-based approach based on COTS hardware and open-source DPDK library to design and implement TSN ES with nanosecond-level packet transmission accuracy and high throughput.

2) We propose a novel multi-core stream allocation framework with conflict-free guarantees to improve the throughput.

3) We perform both hardware-based validation and simulation-based evaluation to demonstrate superior performance compared to other state-of-the-art solutions.

To the best of our knowledge, this is the first work on the design and implementation of DPDK-based TSN ES to support deterministic real-time communications in nanoseconds-level accuracy. Notably, the hardware components of the prototype, including an embedded computation module, IO board, and a commercial NIC have a total cost of less than $500, underlining the affordability of our approach.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658260

```
- 82.14% ___sys_sendmsg
| - 55.85% ____sys_sendmsg
|   - 50.40% sock_sendmsg
|     - 49.93% inet_sendmsg
|       - 49.04% udp_sendmsg
|         - 23.67% udp_send_skb
|           - 22.92% ip_send_skb
|         - 19.11% ip_make_skb
|    - 2.39% uaccess_ttbr0_disable
|    - 2.33% uaccess_ttbr0_enable
| - 25.82% sendmsg_copy_msghdr
|    - 17.66% __copy_msghdr_from_user
|      - 9.53% move_addr_to_kernel.part.0
|      - 4.09% uaccess_ttbr0_disable
|      - 3.65% uaccess_ttbr0_enable
```

Figure 1. The profiling results of Linux API-based TSN ES by Perf.

2 Background and Motivation

While Linux API-based solutions are widely used in TSN ES software implementations, their efficiency of managing real-time TSN traffic is increasingly challenged. For example, [10] implements a software-based solution based on Linux socket with `SO_TXTIME` extension. The authors claim that the proposed solution can support time-triggered (TT) traffic with a period of 31.25 μs and worst-case delay of 4.17 μs. [11] follows a similar design approach to support 100 μs periodic traffic with 42 μs delay. Both approaches fall short of meeting the industry's requirements from emerging applications in terms of both transmission frequency and latency, for example, the automation servo control application in TSN profile [14] requires to communicate every 31.65 μs in control loop.

In order to gain a quantitative understanding on the limitations of the existing software-based approaches, we conducted an empirical analysis using a Linux API-based ES, drawing on methodologies outlined in [11]. The performance was profiled using *Perf* [15], and the results are presented in Fig. 1. From the results, we can observe that the performance bottleneck is at transferring the packets between user space and kernel space, accounting for 25.82% of runtime. In addition, the processing overheads associated with the Linux network stack, such as IP header preparation, also contribute substantially (19.11%). Beyond these delays, the Linux API-based solution also exhibits the lack of flexibility in handling multi-core parallelism. The scheduler, which is typically integrated with the Linux traffic control module and embedded within the kernel, further complicates CPU pinning and the effects of OS scheduling policy.

To reduce the processing delay in the kernel and increase the flexibility of handling multi-core parallelism, our approach designs the TSN ES using DPDK [13]. DPDK is a set of open-source libraries and drivers developed by Intel for fast packet processing in data plane applications, which is widely used in high-speed network interfaces in recent years. The key principle of DPDK lies in its user space operations, eliminating the need for memory copying as bypassing the kernel's networking stack, thereby achieving low-latency packet processing. Moreover, DPDK allows the user to directly access the packet descriptor and hardware registers, which is essential for us to implement the high-accuracy and high-throughput TSN ES without relying on kernel functions.

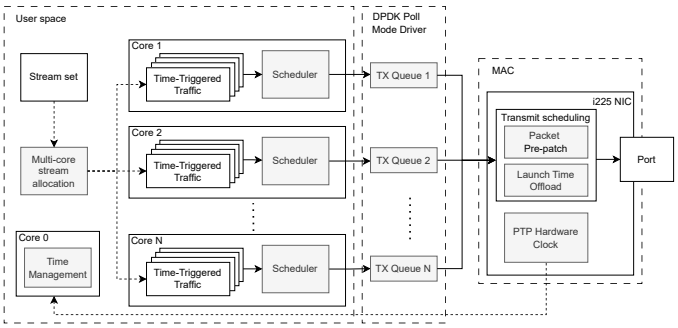

Figure 2. The overall architecture of our TSN ES design based on DPDK.

3 TSN End-Station Design

In this section, we first describe the overall design of our DPDK-enabled TSN ES. We then introduce the key features of each component and the optimization we made to improve the scheduling accuracy and device throughput.

3.1 Overall End Station (ES) Design

The overall architecture of our TSN ES design based on DPDK library is shown in Fig. 2. Our ES design is fundamentally structured two components: the high-precision traffic scheduler and the multi-core stream allocation algorithm. The stream allocation algorithm first determines the specific core responsible for handling each traffic. Subsequently, individual schedulers operating on each core take charge, executing transmissions according to the predefined schedule. To achieve high precision transmission on TT traffic, we first design the scheduler on single core by i) incorporating launch time offloading feature of NIC, and ii) adapting the high precision event timer in the scheduler. To further improve the throughput, we design a multi-core parallel framework for packet processing by leveraging multiple CPU cores on the ES, and utilizing the data pre-fetch hardware function to guarantee the correct traffic order among cores. A parallel contention-free stream allocation algorithm is also designed to optimally assign subsets of streams to specific CPU cores, ensuring that the real-time performance of each traffic type is reliably maintained. We now present the design details below.

3.2 High-Precision Scheduler

The high-precision scheduler aims to send TT traffic following the pre-defined schedule. That is, each packet must be dispatched from the NIC at the exact time specified by the schedule. To achieve this, we first order the streams into an array of packets according to their scheduled transmission time within the hyper-period (the least common multiple of all the stream periods). Then, the scheduler sends the packets in the array following a wake/sleep pattern. Specifically, by comparing the current time with the scheduled time of the packet, the scheduler determines whether to sleep or wake up and send the packet.

To implement the high-precision scheduler, we need to tackle the following two challenges: i) How to achieve deterministic and precise transmission time with minimum jitter?

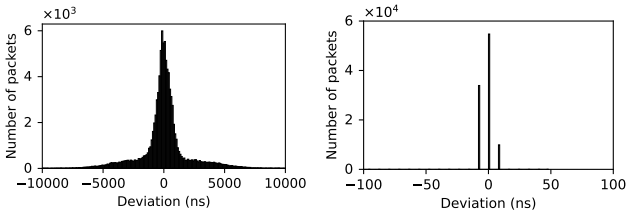

Figure 3. The distribution of time deviation between actual and scheduled sending times under CPU scheduling (left without launch time offload) and NIC scheduling (right with launch time offload) measured in our testbed.

ii) How to avoid transmission congestion and packet loss with high transmission frequency? We discuss our solutions below.

Launch time offload. Traditional TSN ES schedulers rely on CPU scheduling, which suffers from non-deterministic delays caused by command execution time on CPU and direct memory access (DMA) delay. To address this issue, we propose using the hardware offload function to implement the scheduling function on NIC. This is supported by the recent trend of *scheduling time offload* feature development in some NICs (e.g., Intel i210/i225 and Nivida Mellanox CX-5), allowing the NIC to control packet release at specific times. We implement this function in our scheduler based on the latest DPDK (23.07 update), which exposes this offload feature by adding the `SEND_ON_TIMESTAMP` flag in device configure. Once this feature is enabled, we can set the launch time of each packet in the packet advanced descriptor when we send it from the user space.

The main benefit of this design is to isolate the jitter caused by CPU and the random DMA delay between CPU and NIC. Furthermore, as the launch time is referred on the hardware clock of NIC, it also reduces the synchronization error by referring to a more accurate clock. To demonstrate its effectiveness, we compare the jitter distribution of packet transmissions with and without[1] launch time offload in Fig. 3. In the comparison, we send 100,000 packets every 1 ms from the talker and measure the actual time when the packets are received at the listener. We can observe that the delay of sending a packet with launch time offload is much more stable (within 16 ns) than that without launch time offload.

High precision event timer. The second challenge in the implementation of the high-precision scheduler is how to ensure the scheduler runs steadily in high frequency to avoid congestion and packet loss. During our implementation, we identified that the bottleneck is usually caused by the resource contention due to the low timer accuracy. Specifically, if the timer sleeps longer than expected, the time when the CPU issues the packet sending command is already later than the scheduled time, causing the packet miss the sending time. On the other hand, if the timer sleeps shorter than expected, the sending speed will be faster than scheduled and eventually cause the overflow of TX queue, causing the packet loss.

[1]For scheduler without launch time offload, the transmission time is controlled by CPU, i.e., a thread sleeps to the scheduled transmission time and call DPDK rte_eth_tx_burst function immediately.

Table 1. The CPU cyclic profiling results of nanosleep and HPETsleep.

	nanosleep			**HPETsleep**		
Cycle	**min**	**mean**	**max**	**min**	**mean**	**max**
1 μs	2.4 μs	52.5 μs	86.7 μs	94 ns	170 ns	3.5 μs
10 μs	2.4 μs	52.8 μs	141.3 μs	91 ns	160 ns	5.5 μs
100 μs	2.7 μs	54.5 μs	348.5 μs	92 ns	246 ns	10.3 μs
1 ms	4.4 μs	98.7 μs	820.8 μs	101 ns	520 ns	10.9 μs

One straightforward way to design the timer is using nanosleep system call to sleep the thread until the next sending time. However, the nanosleep function is not deterministic and its actual sleeping duration is usually longer than expected according to our experiments. The main reason is that as a system call, the nanosleep function is subject to the OS scheduling policy and it only guarantees the sleeping time longer than the designated time. To solve this issue, we propose to use high precision event timer (HPET) to replace the nanosleep function. HPET is a hardware timer which commonly available in modern computer architecture. It produces periodic interrupts at a much higher resolution than the system RTC clock. In our implementation, the timer resolution is measured as 2.6 GHz on the testbed.

To demonstrate the effectiveness of using HPET timer, we compare it with the precision of the nanosleep function in Table 1. The measurement process is inspired by the well-known *Cyclictest*, which measures the difference between a thread's intended wake-up time and the time when it actually wakes up in order to provide statistics about the system's latency. We conduct the measurement by using *nanosleep* and HPET timers to sleep the thread in 1 μs to 1 ms frequency for 5 minutes and measure the actual sleeping time. Both measurements are referred to the physical clock resident in NIC for fair comparison. From the results, we can observe that the HPET timer can achieve much better worst-case precision (up to 10.9 μs) than the *nanosleep* function. In addition, we find that the HPET timer can achieve sub-microsecond precision on average.

3.3 Multi-Core Parallel Transmission

Due to the non-deterministic delays introduced by the CPU and DMA access, we have to insert a safe inter-space between any two consecutive sending commands on each CPU core to compensate for the worst-case combined delay. Otherwise, the accumulated jitter may cause the packets miss their sending times violating the deterministic transmission requirement. Such inter-space guide time significantly limits the network throughput.

To improve the throughput, we propose to rely on the multi-core architecture by leveraging DPDK's multi-process feature. The key idea is to divide the stream set into multiple subsets and allocate each stream subset on a specific CPU core. DPDK supports the lock-free operation on TX queue connected to each CPU core and this allows the scheduler on different CPUs to send packets concurrently without any conflict. As shown in Fig. 2, we use the master core (core 0) to manage the synchronization and allocate memory buffer

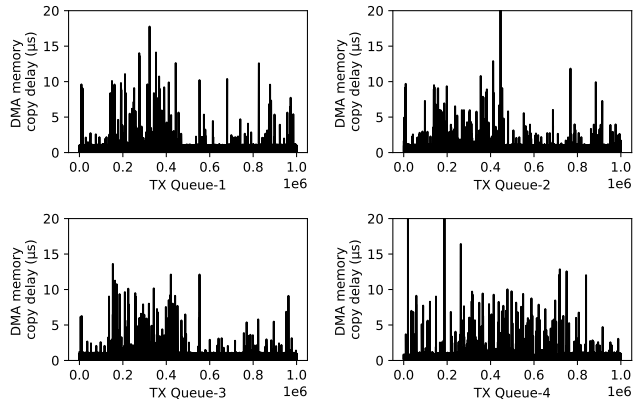

Figure 4. The DMA delay measurement of different TX queues. The X-axis represents the packet index during measurement.

pool and TX queue for each slave core. Each TX queue is associated with one scheduler for the lock-free operation.

However, a major challenge under the multicore parallel transmission design is how to guarantee the packet sending order across different CPUs and the corresponding TX queues given NIC's default priority-based fetch algorithm. Specifically, the packets from the TX queue with higher priority will be fetched first, regardless of their scheduled transmission times. To tackle this issue, we manage to use the data pre-fetch offload function in NIC to guarantee that the packet always be fetched at specific time in NIC.

Data pre-fetch is a hardware-based offloading function that allows the NIC to fetch packets from the memory before launch time. CPU will inform NIC at what time to retrieve packet when it writes TX queue. By enabling this function, we can make sure the packet always pre-fetched at NIC before the launch time comes. Since the data pre-fetch function is not supported by the latest DPDK version yet, we modify the DPDK igc driver to enable this function by writing `DATA_FETCH_TIM` and `FETCH_TIM_DELTA` register fields. Furthermore, since the data pre-fetch function fetches the packet from descriptor to NIC's internal buffer prior to launch time, we need to decide this time distance. We measured the DMA delay of different TX queues in our testbed using the *dpdk-test-dma-perf* tool. According to the measurement results in Fig. 4, the DMA access delay is less than 5 μs on average but reaches 20 μs in the worst case. Thus, we set the data pre-fetch time to 50 μs in our implementation.

Fig. 5 summarizes the operation sequence enabled with both data pre-fetch and launch time offload for packet transmission.

4 Multi-Core Stream Allocation

In our multi-core parallel transmission design, we need to guarantee that the traffic streams allocated to the same CPU core do not conflict. To achieve this, in this section, we first present the system model and formulate the multi-core contention-free stream allocation problem. Then, we map the problem to a classical graph k-coloring problem, and leverage

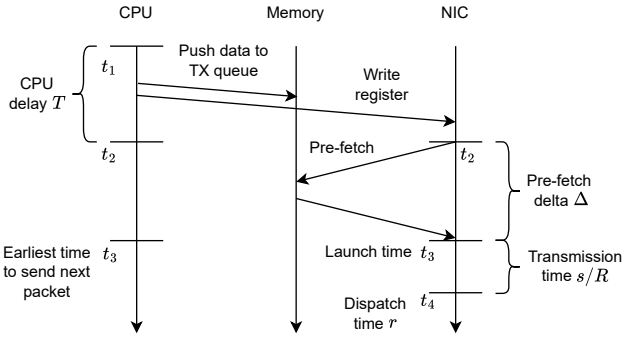

Figure 5. The operation sequence of transmitting a packet in our scheduler design: the CPU issues the sending command at t_1 and the NIC fully dispatches the packet on wire at t_4.

the existing graph coloring algorithms to determine the core allocation for the deterministic TT traffic.

4.1 System Model

Our TSN ES design can be formalized as $E = \{R, Q, \Delta, T\}$ with the line rate R of NIC, the number of available TX queues Q, the pre-fetch time Δ, and the worst-case CPU delay T. Each TX queue is associated with a dedicated CPU core $c_i \in C$. On each CPU core, we assume that the timing overhead caused by synchronization error and command execution time is bounded by a constant T. Each core is responsible for handling a set of TT traffic $f \in F$ following the network-wide TSN schedule. We formalize the schedule of each TT traffic as $f_i = \{p_i, r_i, s_i\}$ with period p_i, predefined releasing offset from NIC r_i, and frame size s_i. T_{hyper} represents the hyper-period of stream set.

4.2 Problem Formulation

The objective of the multi-core stream allocation problem is to find a stream-to-CPU mapping $g : F \to C$ that guarantees all streams scheduled on the same CPU core do not conflict. Specifically, the transmission conflict occurs if: 1) the CPU time occupied by two packets on the same core are overlapped, 2) the number of packets in the same queue exceeds the queue size at any time. In this paper, we simplify the problem by imposing a stricter constraint, limiting the TX queue size to one. This approach results in a sub-optimal yet safe allocation. The problem is formalized with following constraint:

$$\begin{aligned} &\forall f_i, f_j \in F, i < j, g(f_i) = g(f_j) \\ &\forall k_i \in T_{hyper}/p_i, \forall k_j \in T_{hyper}/p_j \Rightarrow \\ &(r_i - (s_i/R + \Delta + T))k_i p_i \geq (r_j - s_j/R)k_j p_j \\ &(r_j - (s_j/R + \Delta + T))k_j p_j \geq (r_i - s_i/R)k_i p_i \end{aligned} \quad \text{Or} \quad (1)$$

Fig. 5 shows an example. Given the transmission time of each stream at t_4, we aim to allocate this stream to CPU cores and make sure there is no other stream overlapping in the preparing time (t_1 to t_3) on the same CPU core.

Next, we translate the multi-core stream allocation algorithm to classical graph coloring problem. We first show how to construct a conflict graph representation of the original problem, where each vertex corresponds to a TT traffic

stream and edges denote the scheduling conflicts. We then apply graph k-coloring algorithm to find a contention-free allocation from streams to CPU cores. We first give two key definitions of the conflict graph:

Definition 1. *There exists a scheduling conflict between f_i and f_j if and only if it violates constraint (1) when f_i and f_j are allocated on the same CPU core.*

Definition 2. *A conflict graph $G = (V, E)$ is a graph representation of the original multi-core stream allocation problem, where each vertex $v_i \in V$ corresponds to the schedule of TT traffic $f_i \in F$, and each edge $e_{i,j} \in E$ denotes the scheduling conflict between f_i and f_j.*

We now show that the multi-core stream allocation problem can be solved by finding a k-coloring of the conflict graph G.

Theorem 1. *Let $G = (V, E)$ be a conflict graph of the original scheduling problem. There exists a feasible mapping $g : F \to C$ if G is a k-colorable graph that $k \leq |Q|$.*

Proof. Assume G is a k-colorable graph where $k \leq |Q|$. This means that the vertices of G can be colored with at most Q colors such that no two adjacent vertices (representing streams with scheduling conflicts) share the same color. Each color used in this graph coloring can be associated with a distinct CPU core. Since adjacent vertices cannot share the same color, this ensures that no two conflicting streams are allocated to the same core. Therefore, a valid k-coloring of G directly translates to a valid mapping $g : F \to C$ that avoids transmission conflicts. □

Based on the above mapping from multi-core stream allocation problem to the classical graph k-coloring problem, many classic algorithms can be applied for stream allocation. For example, in the case of $k = 2$, the graph coloring problem is equivalent to the graph bipartition problem, which can be solved in polynomial time. For the case of $k > 2$, many effective algorithms exist [16, 17] and we employ Z3 SMT solver to solve the multi-core stream allocation problem [18].

5 Performance Evaluation

We now present the evaluation results of our TSN ES design, which is implemented on an x86 platform consisting of a station powered by a 2.5GHz 11th Gen Intel i7 CPU paired with an Intel i225-T1 NIC. We also conduct extensive simulation-based experiments to validate the performance of the proposed multi-core stream allocation framework.

5.1 Experiment Setup

To evaluate the effectiveness of our ES design, we implement a Linux API-based ES for comparison, and both designs are deployed on the same hardware setup. Our test scenario involves two directly connected ESes. For this, we develop a listener using the Linux stack, incorporating `SO_TIMESTAMPING` for precise measurement of delay and jitter. This setup allows a fair comparison between the two designs by only alternating the talkers. For accurate delay measurement, we utilize the *phc2sys* [19] and DPDK example *ptp-client* programs to achieve sub-microsecond-level synchronization accuracy between the talker and the listener.

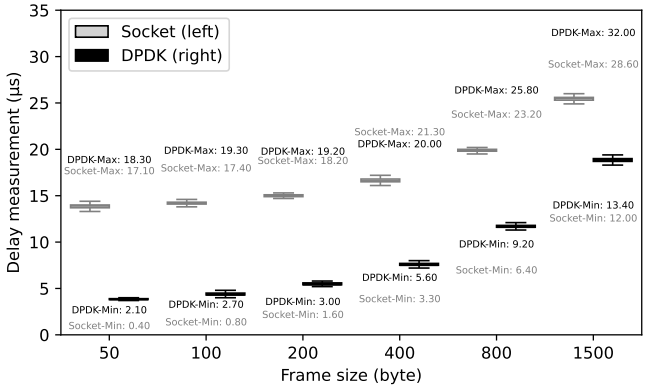

Figure 6. End-to-end delay measurement with various packet sizes.

5.2 End-to-End Delay

We first evaluate the end-to-end delay of the proposed ES design. We let the talker send a packet every 1 ms and record the time when CPU issues the sending command with the hardware timer on NIC. We also record the corresponding arrival time of the first bit of that packet on the listener. Since the two ESes are synchronized, we can calculate the end-to-end delay by subtracting the sending time from the arrival time. It is worth noting that since only one CPU core is used in this experiment, we do not enable the pre-fetch function.

Fig. 6 compares the end-to-end delay between our proposed DPDK-based ES and the Linux API-based ES with various packet sizes. The results show that, on average, our ES design can significantly reduce the delay compared to its Linux API-based counterpart. This improvement is largely attributed to DPDK's user space implementation, which bypasses the kernel processing for lower latency, and its efficient memory access leveraging huge pages and NUMA support. However, outliers can be observed in both designs, where the maximum delay of our ES design occasionally exceeded that of the Linux API-based ES. We hypothesize that these anomalies might be due to measurement inaccuracies, potentially incurred from delays in the talker retrieving time from the NIC.

5.3 Throughput

We next present a comparative analysis of the throughput between the two designs. We configure the talker to send a single periodic stream with varying periods ranging from 5 to 50 μs, with a step size of 5 μ and packet size 52 bytes. The packet loss ratio is recorded on the listener side. For testing the multi-core stream allocation framework, workloads are evenly distributed across two CPU cores. For instance, a stream with a 5 μs period is divided into two parallel streams, each with a 10 μs period, transmitted from two separate CPU cores.

Fig. 7(a) compares the throughput under varying traffic periods. Notably, the single-core DPDK-based design can efficiently handle traffic with a period of 35 μs, outperforming the Linux API-based design which can only manage the traffic with a period of 40 μs. More impressively, the multi-core DPDK framework can further handle the traffic with a period

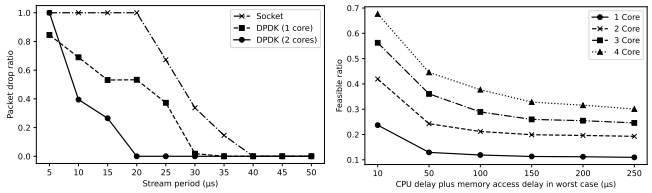

Figure 7. (a) Throughput comparison with varied traffic periods under different numbers of CPU cores. (b) The comparison of feasibility ratio under different numbers of CPU cores. (Core-0 used for synchronization is not counted)

of 20 μs, doubling the capacity of the single-core solution. Besides validating the maximum achievable throughput of our ES design with zero packet loss, our experiments also show that the proposed DPDK-based ES design can achieve even higher throughput when a certain level of packet loss is allowed. These results are omitted here due to the page limit.

5.4 Schedulability

Finally, we conduct simulation-based experiments to evaluate the schedulability of our multi-core stream allocation framework. Our focus is on the feasibility ratio, which represents the proportion of stream sets that can be successfully applied on the ES without contention. We begin the experiments by generating 10,000 random stream sets according to the settings outlined in [20]. These stream sets are then provided as input to a state-of-the-art TSN scheduling algorithm [21], to generate network-wide schedules. Based on the generated schedules, we assess how many of them can be realistically implemented on our ES, taking Constraint (1) into account.

Fig. 7(b) shows the impact of the number of available CPU cores on the feasibility ratio. It can be clearly observed from the results that the increasing number of available CPU cores leads to a higher feasibility ratio in our proposed multi-core parallel framework. Additionally, it is important to note that most existing scheduling algorithms, including the one referenced in [21], do not take these practical constraints into consideration. This highlights the need for developing a high-throughput TSN ES that are compatible with a broader range of existing scheduling algorithms.

6 Conclusion

In this study, we introduce an innovative software-based approach for TSN ES design, utilizing COTS hardware and DPDK open-source libraries to achieve high-precision, ultra-low latency and high throughput real-time packet transmissions for TSN applications. Our method is centered around the scheduler design and can effectively minimize CPU resource contention through the use of HPET and improve the precision by offloading the scheduling duties to the NIC. Additionally, our novel multi-core stream allocation framework ensures efficient management of real-time TSN traffic. Extensive testing in a real-world TSN testbed has demonstrated the superior performance of our approach over existing solutions, marking a significant advancement in TSN ES design. This work not only tackles current challenges in TSN implementations but also provides a cost-effective, adaptable, and high-performance solution, contributing substantially to the field and paving the way for broader IIoT applications.

7 Acknowledgement

The work is supported in part by the NSF Grant CNS-1932480, CNS-2008463, CCF-2028875, CNS-1925706, and the NASA STRI Resilient Extraterrestrial Habitats Institute (RETHi) under grant number 80NSSC19K1076.

References

[1] Norman Finn. 2018. Introduction to time-sensitive networking. *IEEE Communications Standards Magazine* 2, 2 (2018).
[2] Gang Wang, Tianyu Zhang, Chuanyu Xue, Jiachen Wang, Mark Nixon, and Song Han. 2023. Time-Sensitive Networking for Industrial Automation: Challenges, Opportunities, and Directions. *arXiv preprint arXiv:2306.03691* (2023).
[3] Wolfgang Fischer, Joseph Gelish, and Michael Hegarty. 2021. *Aerospace TSN use cases, traffic types, and requirements.*
[4] TTTech Industrial. 2023. Edge IP solution.
[5] Cisco. 2023. Industrial Ethernet 4000 Series Switches - Data Sheet.
[6] Chenglong Li, Zonghui Li, Tao Li, Cunlu Li, and Baosheng Wang. 2023. A Deterministic Embedded End-system Tightly Coupled with TSN Schedule. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023).
[7] Wei Quan, Wenwen Fu, Jinli Yan, and Zhigang Sun. 2020. OpenTSN: an open-source project for time-sensitive networking system development. *CCF Transactions on Networking* 3 (2020).
[8] Eleftherios Kyriakakis, Maja Lund, Luca Pezzarossa, Jens Sparsø, and Martin Schoeberl. 2020. A time-predictable open-source TTEthernet end-system. *Journal of Systems Architecture* (2020).
[9] Chenglong Li, Tao Li, Junnan Li, Wenwen Fu, and Baosheng Wang. 2023. DRA: Ultra-Low Latency Network I/O for TSN Embedded End-Systems. In *31st International Symposium on Quality of Service (IWQoS)*.
[10] James Coleman, Sara Almalih, Alexander Slota, and Yann-Hang Lee. 2019. Emerging cots architecture support for real-time TSN ethernet. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*.
[11] Marcin Bosk, Filip Rezabek, Kilian Holzinger, Angela Gonzalez Marino, Abdoul Aziz Kane, Francesc Fons, Jörg Ott, and Georg Carle. 2022. Methodology and infrastructure for TSN-based reproducible network experiments. *IEEE Access* 10 (2022).
[12] Christoph Lehr, Patrick Denzler, Thomas Frühwirth, and Wolfgang Kastner. 2023. Buffer Management for TSN-Enabled End Stations. In *19th International Conference on Factory Communication Systems (WFCS)*.
[13] Linux Foundation. 2015. Data Plane Development Kit (DPDK).
[14] Josef Dorr, Karl Weber, and Steven Zuponcic. 2018. *Use Cases IEC/IEEE 60802.*
[15] Arnaldo Carvalho De Melo. 2010. The new Linux Perf tools. In *Slides from Linux Kongress*, Vol. 18.
[16] Philippe Galinier, Alain Hertz, and Nicolas Zufferey. 2008. An adaptive memory algorithm for the k-coloring problem. *Discrete Applied Mathematics* 156, 2 (2008).
[17] Alain Hertz, Matthieu Plumettaz, and Nicolas Zufferey. 2008. Variable space search for graph coloring. *Discrete Applied Mathematics* 156, 13 (2008).
[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.*
[19] Richard Cochran and Cristian Marinescu. 2010. Design and implementation of a PTP clock infrastructure for the Linux kernel. In *2010 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication.*
[20] Chuanyu Xue, Tianyu Zhang, Yuanbin Zhou, Mark Nixon, Andrew Loveless, and Song Han. 2024. Real-Time Scheduling for 802.1Qbv Time-Sensitive Networking (TSN): A Systematic Review and Experimental Study. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium.*
[21] Daniel Bujosa, Mohammad Ashjaei, Alessandro V Papadopoulos, Thomas Nolte, and Julián Proenza. 2022. HERMES: Heuristic multi-queue scheduler for TSN time-triggered traffic with zero reception jitter capabilities. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems.*