Virtualizing Industrial Networks with EtherCAT Protocol Implementation

Shuhei Otaki
Graduate School of Science and Engineering,
Chitose Institute of Science and Technology
Chitose, Japan
m2240120@photon.chitose.ac.jp

Takashi Yamada
Silicon Research Center,
Chitose Institute of Science and Technology
Chitose, Japan
t-yamada@photon.chitose.ac.jp

***Abstract*—In factory networks, various network protocols are used to meet strict requirements for accuracy and latency. Because the controllers that use these protocols are built with dedicated equipment, it is difficult to mix them with other protocols. Moreover, these controllers often rely on specialized software, complicating the integration of innovative technologies like AI. Therefore, virtualization of factory networks, which involves software-based factory network protocols and using general-purpose servers as controllers, is a very attractive approach. However, one of the challenges of software-based factory network protocols is real-time responsiveness. The time fluctuation of communication packet intervals caused by using a general-purpose server reduces the accuracy and speed of control. In this paper, we implemented the EtherCAT protocol in software and measured the jitter. Through experimentation, we confirmed that our system achieves stable 8 ms cyclic communication, enabling consistent control of multiple daisy-chained slave devices.**

Keywords— Software Defined Automation, vPLC, Protocols

I. Introduction

In recent years, under the paradigm of Industry 4.0, the realization of smart factories—where all machines are interconnected through networks and autonomously optimize production processes—has gained significant attention. At the core of this concept lies the Industrial Internet of Things (IIoT), which enables seamless connectivity among industrial devices, sensors, and actuators, facilitating real-time data acquisition, analysis, and control.

To support these functionalities, communication networks in industrial automation must meet extremely stringent requirements, such as ultra-low latency, deterministic behavior, precise synchronization, high availability, and secure data transmission. Production downtimes due to network delays or failures can result in immediate and substantial economic losses. Consequently, real-time Ethernet technologies such as EtherCAT, PROFINET, and EtherNet/IP have been widely deployed in industrial environments, achieving communication latencies in the sub-millisecond range and ensuring reliable cyclic exchange between controllers and field devices [1].

Despite their capabilities, these industrial communication protocols rely heavily on proprietary technologies, which limits their interoperability across vendors and systems. Moreover, they are tightly coupled to specialized hardware platforms, particularly Programmable Logic Controllers (PLCs), making system integration, migration, or reconfiguration costly and inflexible [2]. This rigidity imposes constraints on hardware reusability and modular upgrades, increasing the total cost of ownership.

A wide-ranging survey by Danielis et al. [3] has classified and compared real-time communication technologies via Ethernet in industrial automation environments, highlighting challenges such as non-uniform protocol stacks, timing constraints, and implementation overheads. These foundational insights underline the need for more unified and flexible network architectures to support future IIoT applications.

To address these challenges, researchers have explored the application of Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) in industrial networks. SDN decouples the control plane from the data plane, enabling centralized, programmable management of traffic flows. This opens possibilities for dynamic reconfiguration, flow prioritization, and fault-tolerant routing in deterministic environments. A comprehensive survey by Granelli et al. [4] further categorizes SDN solutions and controller architectures for IIoT scenarios, identifying gaps in scalability, reliability, and real-time guarantees.

For example, the VirtuWind project proposed an SDN- and NFV-based architecture tailored for mission-critical environments such as wind farms. This architecture introduces network slicing, multi-tenancy, and on-demand service provisioning while maintaining the stringent timing and reliability constraints of industrial control networks [5].

In parallel, the combination of SDN and Time-Sensitive Networking (TSN) has been proposed to support mixed-criticality applications, allowing coexistence of time-critical and best-effort traffic in the same physical infrastructure [6], [7]. These approaches suggest that SDN can enhance the manageability and extensibility of industrial networks without compromising real-time performance.

Meanwhile, virtualization of industrial control systems is gaining attention as an alternative to traditional PLCs. Approaches such as cloud-based soft-PLCs [8] and container-based virtual PLCs [9] demonstrate the feasibility of implementing control logic on general-purpose computing hardware. These architectures enable dynamic software

979-8-3315-6946-4/25/$31.00 ©2025 IEEE

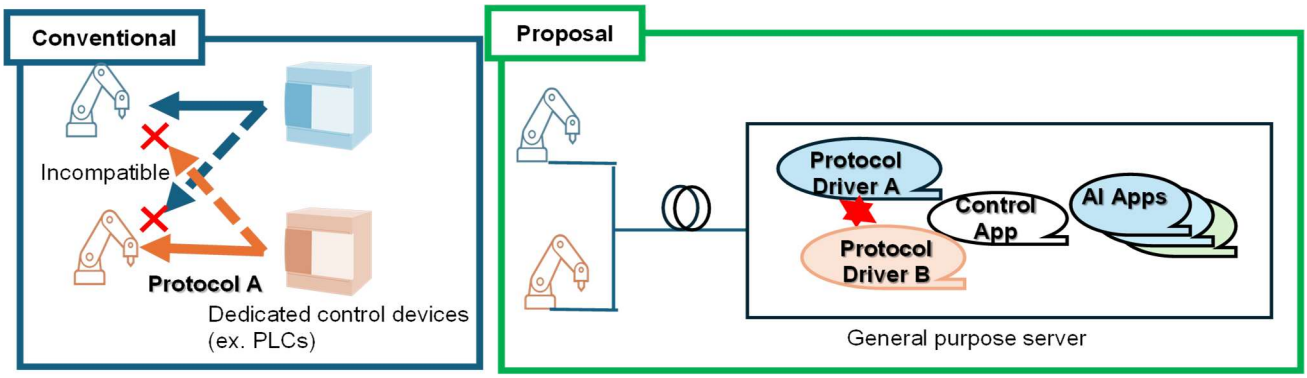


Fig. 1 Proposed Architecture

updates, integration with AI-based decision-making, and remote or distributed control scenarios previously impractical with hardware-bound PLCs. However, as noted by Gundall et al. [10], virtualization technologies must be carefully evaluated in terms of their impact on timing guarantees, reliability, and orchestration complexity. Their study raises the important question of whether virtualization will ultimately serve as a catalyst or a show-stopper for industrial automation, especially in time-critical scenarios.

At a higher architectural level, frameworks like the "Controller of Controllers" have been proposed to manage heterogeneous SDN domains. This hierarchical control design allows for protocol-agnostic management of industrial traffic across multiple technologies and vendors, contributing to unified industrial networking [11].

In response to these trends and challenges, we previously proposed a novel architecture that transforms industrial Ethernet protocols into modular software drivers executable on general-purpose servers [12]. Our system enables real-time protocol generation and motion control—e.g., for EtherCAT—without requiring specialized hardware or modifications to the physical network. Moreover, control logic can be written in conventional programming languages and integrated with advanced functionalities such as delay compensation and AI-based collaborative control. Since the system operates on standard server platforms, updates and extensions can be performed with minimal overhead, offering scalability, portability, and compatibility for modern industrial automation.

This paper is organized as follows. In Section II, we describe the EtherCAT, which is the protocol we implemented in this study. In Section III, the overview of the proposed system is described. In Section IV, the experimental results of the implemented EtherCAT protocol driver are presented. In Section V, the evaluation of software-induced timing jitter is discussed, along with strategies for its improvement. In Section VI, the paper is concluded.

II. Proposed System

This section describes EtherCAT, a protocol developed to address issues of communication speed and synchronization control that were limitations in conventional industrial networks. EtherCAT utilizes standard Ethernet as its physical layer while adopting a uniquely optimized communication method for control applications, offering exceptional real-time performance and synchronization accuracy. It was introduced by Beckhoff Automation in 2003 and has since become widely adopted as an open standard.

Figure 2 shows a typical topology in EtherCAT network. EtherCAT employs a master-slave architecture, in which a single master PLC controls multiple slave devices such as sensors, actuators, and I/O modules. Physically, these slave devices are typically connected in a daisy-chain configuration using Ethernet cables. Each slave is equipped with two Ethernet ports—an IN port and an OUT port—and processes frames by receiving, handling, and forwarding them to the next device. This simple configuration eliminates the need for Ethernet switches or hubs, thereby simplifying wiring and minimizing communication delay.

One of the most distinctive features of EtherCAT is its on-the-fly processing mechanism. In this method, a single frame transmitted by the EtherCAT master sequentially passes through each slave on the network. As it passes through, each slave reads from or writes to its designated byte region in real time, without stopping or storing the frame. The processing is completed within a time frame of several tens to hundreds of nanoseconds. This mechanism enables ultra-low communication latency, reduces overall header overhead by eliminating individual responses, and ensures highly stable communication timing. This on-the-fly mechanism stands in stark contrast to conventional polling-based communication systems, where the master must communicate with each slave individually. It is a core technology that defines the high-performance characteristics of EtherCAT.

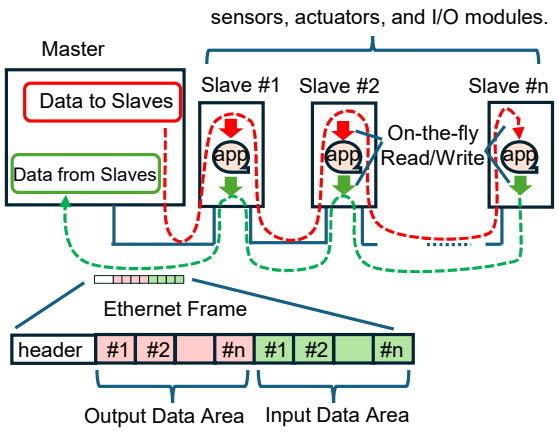


Fig. 2 EtherCAT Network Topology

An EtherCAT frame can contain multiple datagrams within a single Ethernet frame. Each datagram is capable of accessing a specific slave or memory address range. This architecture enables simultaneous updates and distributed access in multi-node environments, contributing to the protocol's efficiency and scalability.

III. EtherCAT Protocol

The proposed system is illustrated in Fig. 3. A general-purpose server, acting as the EtherCAT master, is connected to multiple slaves in a daisy-chain topology. The EtherCAT master consists of two main components: a Control Application, which computes commands to be issued to the slaves (such as robotic arms), and a Protocol Driver, which generates the EtherCAT protocol frames. Although our primary objective is to complete the protocol driver on the master side, we also implemented a protocol driver on the slave side for experimental purposes. Data exchange between the Control Application and the Protocol Driver is carried out via UDP communication within the server.

Figure 4 shows the flowchart of the transmission process in the protocol driver. The protocol driver is composed of two multithreaded components: the Data Receive Handler, which receives UDP packets from the Control Application, and the EtherCAT Frame Generator, which transmits EtherCAT frames outside the server. When the Data Receive Handler receives a UDP packet, it extracts the data from predefined positions in the packet and stores it in memory. The EtherCAT Frame Generator periodically sends EtherCAT frames at fixed intervals defined by the protocol, regardless of whether new data has been received from the Control Application. If data is present in memory, it is copied into the appropriate datagram region of the EtherCAT frame before transmission.

We implemented the EtherCAT protocol driver using the Go programming language and C programming language respectively. In the protocol driver, we implemented four EtherCAT commands: Logical Memory Write (LWR), Logical Memory Read (LRD), Broadcast Read (BRD), and Auto Increment Write (APWR). The LWR command is used to send data from the master to the slaves, while the LRD command retrieves data, such as sensor values from the slaves to the master. The BRD command is employed to notify the master of any status abnormalities in the slaves, and the APWR command is used for issuing initial configuration instructions to the slaves.

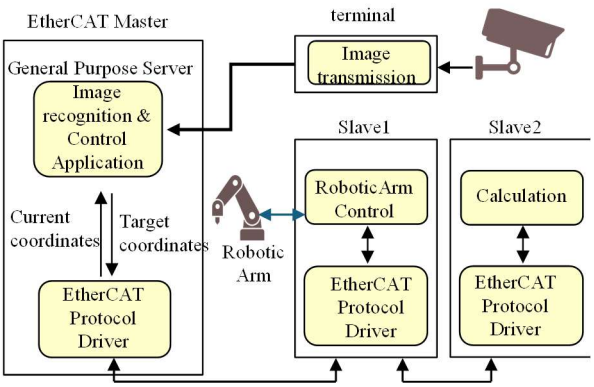


Fig. 3 Proposed System

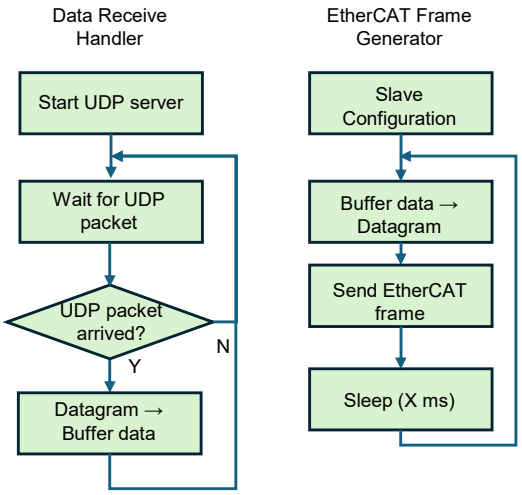


Fig. 4 Protocol Driver Flowchart

In the EtherCAT protocol, since all slaves receive the same Ethernet frame from the master through a daisy chain connection, it is necessary to specify where each slave's input and output are located within the Ethernet frame. Therefore, we propose incorporating this initial configuration into the EtherCAT protocol driver.

command is used to inform each slave of the specific byte offsets in the Ethernet frame where it should read input data from and write output data to. After the initial configuration, the EtherCAT protocol driver will periodically send LRD, LWR, and BRD commands to enable communication between the master and multiple slaves.

IV. EtherCAT Protocol Implementation

Figure 5 shows a visual representation of the experimental setup, including the server for EtherCAT master, Raspberry Pi 4 for slaves and image transmission terminal, and the robotic arm. The robotic arm utilized in this study is the Lite 6 from uFactory, while the EtherCAT module employed is the EasyCAT developed by AB&T. The server is running the Ubuntu 22.04 LTS operating system. The network standard uses 100BASE-TX, which is aligned with the EtherCAT specifications.

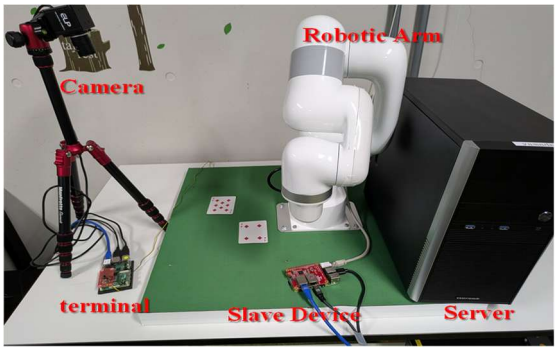


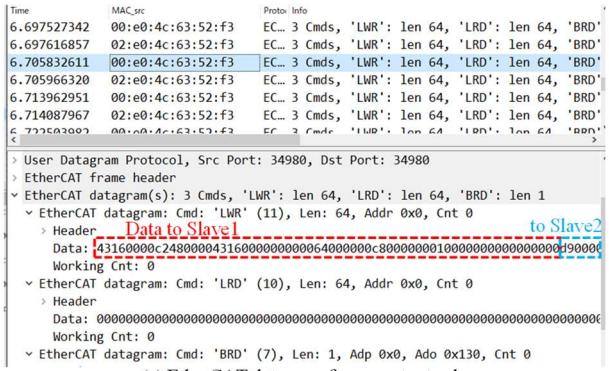
Fig. 5 Experimental System Setup

To demonstrate that communication with multiple slaves is functioning correctly, each slave is running a different program. One is Robotic Arm Control, and the other is Calculation. The Image Recognition and Control Application running on the master system receives image data from an image transmission terminal and performs object detection using YOLOv3 (You Only Look Once version 3). Upon recognizing a playing card, the application extracts its coordinate information and transmits it to the slave device via the EtherCAT protocol. On the slave side, the Robotic Arm Control Program interprets the received coordinates, actuates the robotic arm to pick up the identified card, and transports it to a predefined location. The Calculation program within Slave 2 takes an input X provided by the master and returns an output Y according to the formula Y=255 − X. The master increments X every 8 ms frame.

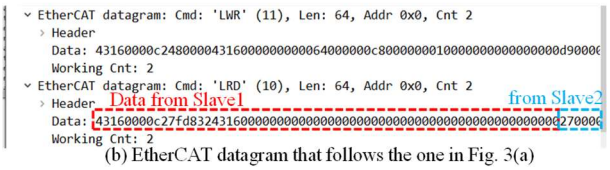
The experimental results are shown in Figs. 6 and 7. Figure 6(a) presents the packet capture results for transmission and reception on the master server. It was confirmed that packets from the master's MAC address 00:e0:4c:63:52 were sent at approximately 8 millisecond intervals. It is also confirmed that packets sent from the master are immediately followed by packets received from the slave (MAC address 02:e0:4c:63:52).

In this experiment, we set the data area within the LRW and LRD commands to 32 bytes per slave. In the LWR command, the first 12 bytes of the data represent the target coordinate values (x, y, z) in single-precision floating-point format. The value 0xd9 at the 33rd byte in this figure represents the data X sent to Slave 2. The datagram of the frame from the slave to the master following the EtherCAT frame is shown in Fig. 6(b). The data in the LRD command has been overwritten, with the first 12 bytes representing the current xyz coordinates of the robotic arm. The value at the 33rd byte is the return value from Slave 2, which, due to the internal processing time of Slave 2, is Y=0x27 in response to the previous input value X=0xd8.

Figure 7(a) illustrates the time-series changes in the coordinates of the robotic arm when two playing cards are

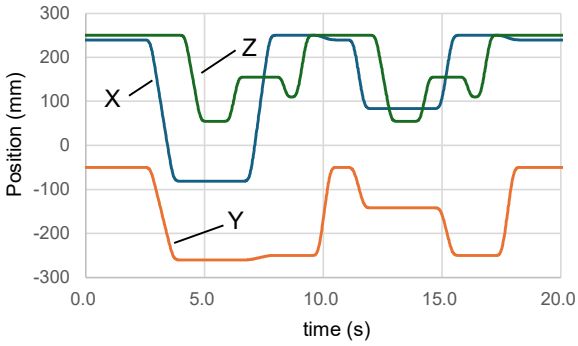


(a) EtherCAT datagram from master to slave

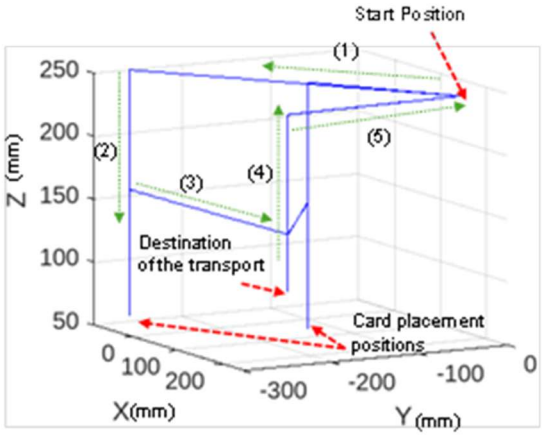


(b) EtherCAT datagram that follows the one in Fig. 3(a)

Fig. 6 Packet Capture Result



(a)



(b)

Fig. 7 Robotic Arm Results (a) coordinate, (b) trajectory

placed. It was confirmed that the robotic arm operates based on the data transmitted via the EtherCAT protocol.

Figure 7(b) shows a 3D plot of the trajectory corresponding to the coordinate data. The robotic arm moves from the start position to the location of the first card (1), (2), then transports the card to a predefined location (3), and finally returns to the start position (4), (5). A similar sequence of operations was also confirmed for the second card.

V. Evaluation of Timing Jitter

One of the critical challenges in implementing the protocol driver in software is timing jitter. In general-purpose operating systems such as Linux, the kernel employs time-sharing to manage various processes. As a result, interrupt handling and task scheduling may cause temporary suspension of the target process, introducing fluctuations in the actual execution timing. This inherent behavior of the OS leads to non-deterministic delays, which become particularly problematic in systems requiring precise timing control.

In the proposed system, the component that demands the highest level of timing accuracy is the Sleep process within the EtherCAT Frame Generator, as illustrated in Figure 4. If the sleep duration deviates from its intended value, the timing of EtherCAT frame transmissions becomes irregular. Such jitter in frame intervals may, in some cases, be interpreted by the network as a protocol violation, potentially causing the entire system to enter a fault state or halt unexpectedly.

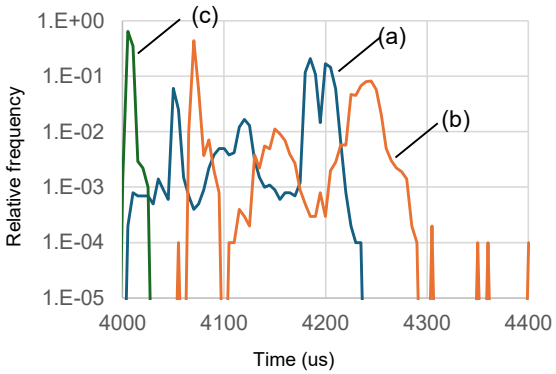


Fig. 8 Evaluation of Timing Jitter

Given the severity of this risk, the first step in our evaluation was to quantitatively assess the accuracy of the sleep duration in the frame transmission logic. By doing so, we aim to identify the extent of timing jitter introduced by the software and determine whether corrective measures are necessary to maintain network reliability and system stability.

This paper evaluates two approaches for reducing timing jitter. The first approach involves real-time enhancement of the Linux operating system. PREEMPT-RT (PREEMPT Real-Time Patch) is a patch set that extends the Linux kernel for real-time processing. By enabling fine-grained control of kernel preemption, it significantly improves both the determinism and responsiveness of task execution. In the PREEMPT-RT kernel, spinlocks used for mutual exclusion are replaced with mutexes, allowing preemption across the entire kernel. Additionally, part of the interrupt handling (ISR) is implemented as kernel threads, enabling more flexible priority control and scheduling between user-space processes and kernel tasks. As a result, PREEMPT-RT makes it possible to achieve sub-millisecond-level jitter control and stable periodic execution, which are difficult to realize in a standard Linux environment.

The second approach is optimization of the real-time execution environment. By applying the SCHED_FIFO scheduling policy and assigning a high priority to the process (priority level 80 in our experiment), we minimized interference from other processes. Furthermore, by setting CPU affinity, the target process was pinned to a specific CPU core. This configuration avoids context switching between CPU cores and cache misses, enabling more stable timing control. These two techniques together are expected to suppress jitter and improve the determinism of time-sensitive operations in software-based periodic processing.

In this experiment, we implemented a C-language program to execute a 4 ms sleep operation and measured the actual sleep duration in order to compare timing accuracy. Measurements were conducted under the following three execution environments; each tested with 10,000 iterations:

a) Baseline: PREEMPT-RT disabled, without SCHED_FIFO or CPU affinity.

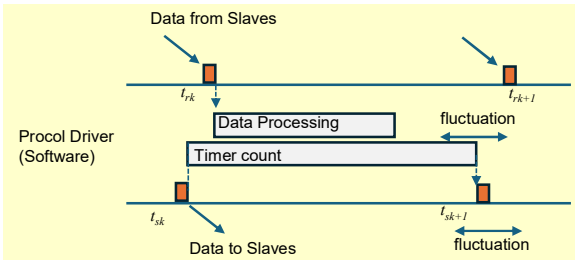
b) With PREEMPT-RT: PREEMPT-RT enabled, without SCHED_FIFO or CPU affinity.

c) With PREEMPT-RT + Execution Optimization: PREEMPT-RT enabled, with SCHED_FIFO and CPU affinity.

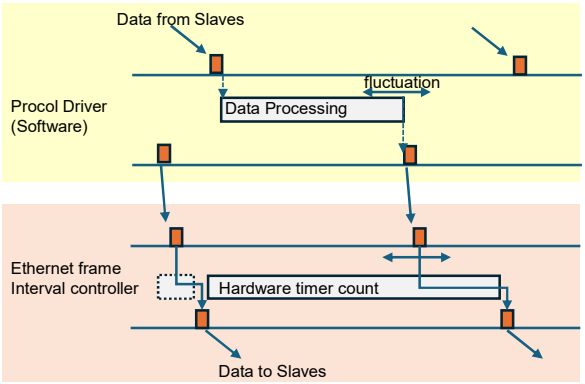
The relative frequency distribution of the actual sleep durations for each case is shown in Figure 8. In case (a), the distribution peaks around 4.2 ms with a spread of approximately 2.5 ms. In case (b), the peak improves to 4.07 ms, but the overall distribution widens to approximately 3.45 ms, indicating an increased jitter. In case (c), the distribution is highly concentrated around 4.005 ms, with a spread of only 20 µs, demonstrating a significant reduction in timing jitter. These results indicate that applying PREEMPT-RT, in combination with real-time scheduling (SCHED_FIFO) and CPU affinity, can greatly improve timing determinism and reduce jitters in software-based periodic processing.

The previous experiment demonstrated that timing jitter can be reduced to approximately 20 µs. However, considering that the minimum communication cycle defined by the EtherCAT specification is 100 µs, further reduction in jitter is required.

To address this issue, we propose a method to suppress jitter by controlling the Ethernet frame transmission interval using hardware, as illustrated in Figure 9. In the conventional approach, shown in Figure 9(a), a software timer is used to



(a) Conventional configuration



(b) Proposed system

Fig 9 Timing Diagram of Frame Transmission

wait for a fixed interval after data transmission. However, this timer-based processing introduces fluctuation, resulting in variability in the subsequent transmission timing. In contrast, the proposed method, shown in Figure 9(b), eliminates the use of software-based timer counting, and enables the system to transmit the data immediately after processing is completed. The accurate timing control of transmission intervals is delegated to hardware, such as a SMART NIC or an FPGA board equipped with multiple Ethernet ports, which performs precise hardware-based timing. This allows the system to absorb timing jitter introduced by the software. In this

configuration, the achievable minimum cycle time is limited by the maximum data processing time and the fixed delay of the Ethernet Frame Interval Controller. Therefore, it is essential to minimize the fluctuation in data processing time, thereby reducing the maximum value. To this end, the application of PREEMPT-RT, real-time scheduling with SCHED_FIFO, and CPU affinity settings, as discussed earlier, is considered effective.

VI. Conclusion

In this paper, we proposed and evaluated a software-based EtherCAT protocol implementation designed for general-purpose computing platforms, with the aim of enabling virtualized industrial networks. Through the development of a protocol driver and control applications, we demonstrated stable cyclic communication and correct control of multiple slave devices, including robotic arms.

One of the key challenges in software-based industrial protocols is timing jitter. We addressed this by evaluating two jitter mitigation strategies: (1) enabling real-time capabilities in the Linux kernel using PREEMPT-RT, and (2) optimizing the runtime environment through SCHED_FIFO scheduling and CPU affinity. Experimental results showed that the combination of these two techniques significantly reduced timing jitter to approximately 20 μs.

Furthermore, to achieve sub-100 μs cycle times as required by the EtherCAT specification, we proposed a hardware-assisted approach for precise Ethernet frame interval control. By offloading timing-sensitive functions to a Smart NIC or FPGA-based controller, software-induced jitters can be further minimized. These results suggest that a hybrid architecture—combining software-defined control and hardware-level timing stabilization—can provide a viable foundation for real-time, virtualized industrial communication systems.

References

[1] X. Wu, L. Xie, "Performance evaluation of industrial Ethernet protocols for networked control application," Control Engineering Practice, Volume 84, 2019, Pages 208-217

[2] K. Ahmed, J. O. Blech, M. A. Gregory, and H. Schmidt, “Software Defined Networks in Industrial Automation," Journal of Sensor and Actuator Networks, vol. 7, no. 3, Art. no. 33, pp. 1–21, Aug. 2018.

[3] P. Danielis et al., "Survey on real-time communication via ethernet in industrial automation environments," Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), Barcelona, Spain, 2014, pp. 1-8.

[4] J. Granelli, V. I. Munteanu, G. Rizzo, N. Ulrich, D. Siracusa, and M. P. Anastasopoulos, “Software-defined networking solutions, architecture and controllers for the industrial Internet of Things: A review,” IEEE Communications Surveys & Tutorials, vol. 23, no. 3, pp. 1830–1857, 2021.

[5] E. Sakic, V. Kulkarni, V. Theodorou, A. Matsiuk, S. Kuenzer, N. Petroulakis, K. Fysarakis, "VirtuWind – An SDN- and NFV-Based Architecture for Softwarized Industrial Networks," Proceedings of 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB 2018), pp.251-261, 2018.

[6] L. Moutinho, P. Pedreiras, and L. Almeida, “A Real-Time Software Defined Networking Framework for Next-Generation Industrial Networks,” IEEE Access, vol. 7, pp. 164468–164479, 2019.

[7] M. A. Metaal, R. Guillaume, R. Steinmetz, and A. Rizk, “Integrated Industrial Ethernet Networks: Time-Sensitive Networking over SDN Infrastructure for Mixed Applications,” 2020 IFIP Networking Conference (Networking), pp. 803–808, 2020.

[8] T. Goldschmidt, M. K. Murugaiah, C. Sonntag, B. Schlich, S. Biallas, and P. Weber, “Cloud-Based Control: A Multi-tenant, Horizontally Scalable Soft-PLC,” 2015 IEEE 8th International Conference on Cloud Computing, pp. 909–916, 2015.

[9] T. Cruz, P. Simões, and E. Monteiro, “Virtualizing Programmable Logic Controllers: Toward a Convergent Approach,” IEEE Embedded Systems Letters, vol. 8, no. 4, pp. 69–72, 2016.

[10] M. Gundall, D. Reti and H. D. Schotten, "Application of Virtualization Technologies in Novel Industrial Automation: Catalyst or Show-Stopper?," 2020 IEEE 18th International Conference on Industrial Informatics (INDIN), Warwick, United Kingdom, 2020, pp. 283-290.

[11] M. A. Metaal, R. Guillaume, R. Steinmetz and A. Rizk, “Integrated Industrial Ethernet Networks: Time-sensitive Networking over SDN Infrastructure for mixed Applications,” 2020 IFIP Networking Conference (Networking) , pp. 803-808, 2020.

[12] Y. Koyasako, T. Suzuki, T. Hatano, T. Shimada and T. Yoshida, “Demonstration of Industrial Ethernet Protocol Softwarization and Advanced Motion Control for Full Software-Defined Factory Network,” IEEE Access, vol. 12, pp. 104020-104030, 2024.