International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 21 (2017) pp. 11179-11185

Real-time Servo Control using EtherCAT Master on Real-time Embedded Linux Extensions

Raimarius Delgado
Ph.D. Student, Department of Electrical and Information Engineering,
Seoul National University of Science and Technology, 232 Gongneung-ro, Nowon-gu,Seoul, South Korea.
Orcid Id: 0000-0002-6759-4240

Byoung Wook Choi
Professor, Department of Electrical and Information Engineering,
Seoul National University of Science and Technology, 232 Gongneung-ro, Nowon-gu, Seoul, South Korea.
Orcid Id: 0000-0002-2404-7415

Abstract

In this paper, we present a real-time servo control application based on an open-source EtherCAT Master under two different real-time embedded Linux approaches, the fully pre-emptible kernel and Xenomai. The fully pre-emptible approach utilizes the native Linux kernel patched to support pre-emption of high-priority tasks at any time and protects critical sections with spin locks. On the contrary, Xenomai is an example of dual-kernel approach, which provides real-time extensions to tasks piped alongside the standard Linux kernel through a hardware abstraction layer. The EtherCAT master is stacked on top of each real-time operating system to provide real-time connection with servo drives that are equipped with EtherCAT slave controllers that supports CANopen protocol. This paper aims to analyze the real-time performance of the EtherCAT Master depending on the real-time Linux extension it is running on in terms of cyclic task periodicity, jitter, and in-controller execution time. The in-controller execution is defined as the time interval from the master sending information to slaves and vice-versa. The experiment is conducted on an actual system where the master is required to control a set of servo motors. The results acquired from each master are analyzed and compared to serve as a guideline in in designing industrial and automation systems using different real-time Linux extensions.

Keywords: EtherCAT, Xenomai, RT_PREEMPT, Real-time, Embedded Linux.

INTRODUCTION

In practical applications, precise control period is critical for accurate manipulation. For example, in a robot system where servo motors symbolize each joint, velocity commands given to each of the motors should be kept in strict time sampling to smoothly follow the desired path. Moreover, data from the environment and other disturbances are acquired via several types of sensors which are updated periodically, if significant data loss occurs, the system would not be able to react accordingly and result to a failure. Defining the importance of conserving real-time requirements [1-2].

To meet these constraints, real-time Ethernet protocols have been developed to ensure determinism over standard Ethernet since it is not optimized to send subsequent short messages and could not realize standard automation real-time requirements in comparison to other fieldbuses, which are vital in controlling intelligent and dynamic systems in the industrial field. EtherCAT (Ethernet for Control Automation) is a real-time Ethernet protocol that is gaining popularity in rigorous automation. It offers various appealing features such as optimal usage of the Ethernet bandwidth for data transfers, short cycle times, and full compatibility with the standard Ethernet protocol [3-5].

Commonly, EtherCAT systems require a master connected to several slaves that are configured either into daisy chain, star, or tree topology. Slave connection can also be a combination of any of the three. The EtherCAT master is required to be running on top of a real-time operating system (RTOS) that provides mechanisms for synchronized data-handling and real-time scheduling. Commercial and open source EtherCAT master distributions such as KPA EtherCAT Master, Simple Open EtherCAT Master (SOEM), and IgH EtherCAT Master are designed originally for standard Linux [6-8]. However, standard Linux does not provide a real-time scheduler since it values fairness over priority [9]. To achieve real-time capability, there are two popular real-time Linux methods that are largely used in general: pre-emptible kernel approach and dual-kernel approach, represented respectively by RT_PREEMPT and Xenomai [10-11].

Sung et.al [12] developed an EtherCAT system using older versions of both Xenomai (2.6.0) and IgH EtherCAT (1.5.0) and constructed a holistic timing analysis model of the system in terms of end-to-end delay between the synchronized EtherCAT processes. However, real-time performance of the

control task that handles EtherCAT frames running on the master was not included in their timing model.

On the other hand, Cereia et.al [13] performed a similar study to evaluate the performance of IgH EtherCAT Master using RT_PREEMPT. Although the real-time performance of the EtherCAT control task was evaluated, their procedure runs the EtherCAT master more like a network adapter than a main controller. Thus, performance with actual workload that is useful in practical application was not established.

Therefore, the contributions of this paper are divided into three parts: First, we develop two EtherCAT master using the latest version of IgH EtherCAT, v1.5.2 [14] on each real-time Linux extension. For consistency of the results, we selected a Linux kernel version that can support both real-time extensions in their latest version. Defining the kernel version used is essential especially for RT_PREEMPT since the kernel code, including software dependent part, is drastically modified to guarantee real-time performance. Secondly, both EtherCAT masters are connected to six servo drives using CANopen-over-EtherCAT (CoE) protocol to exhibit actual workload by actuating the corresponding AC motors. Finally, we performed timing analysis for each EtherCAT master to guarantee practicality in servo control application. The performance was evaluated in terms of the periodicity and corresponding jitters of the real-time control task, and the in-controller execution time. The execution time is defined as the overall time that it takes for the master to handle data frames while in connection with the slaves.

The second section begins with the introduction of the two real-time embedded Linux approaches. Section 3 is about the CoE protocol. The fourth section describes the environment on which the experiments of this paper were performed. This section also discusses the difference of each real-time Linux extension in writing a user space application for a real-time EtherCAT control task. The fifth section shows the experiment results and some remarks regarding the acquired data. The last section closes the paper with the conclusion.

REAL-TIME LINUX EXTENSIONS

Pre-emptible Kernel Approach

The key point of the pre-emptible kernel approach is to minimize the amount of kernel code that is non-pre-emptible, while also minimizing the amount of code that must be changed to provide this added pre-emptibility. Critical sections, interrupt handlers, and interrupt-disable code sequences are normally pre-emptible.

In the Linux kernel with the patch RT_PREEMPT, spinlocks and mutexes are converted to real-time spinlocks and mutexes, respectively. These mechanisms are used to implement mutual exclusions and leverage the SMP capabilities of the Linux kernel to add this extra pre-emptibility without requiring a complete kernel rewrite.

In a sense, one can loosely think of preemption as the addition of a new CPU to the system, and then use the normal locking primitives to synchronize with any action taken by the preempting task. This preemption cannot be done safely at arbitrary places in the kernel code. One section of code where this may not be safe is within a critical section. A critical section is a code sequence that must not be executed by more than one process at the same time.

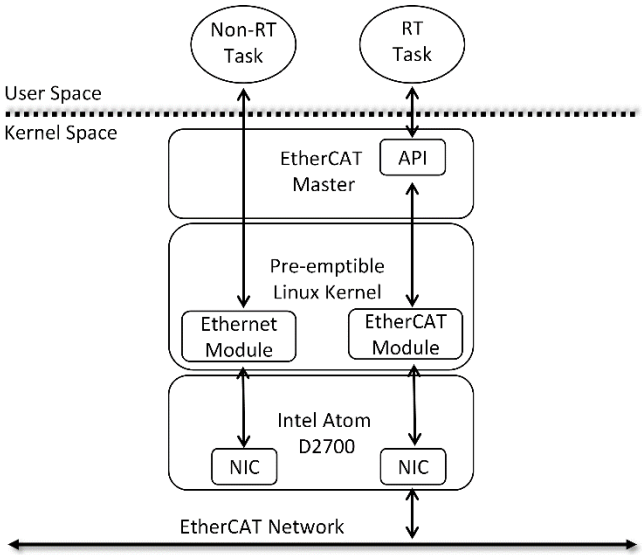

Figure 1: Pre-emptible kernel approach using RT_PREEMPT

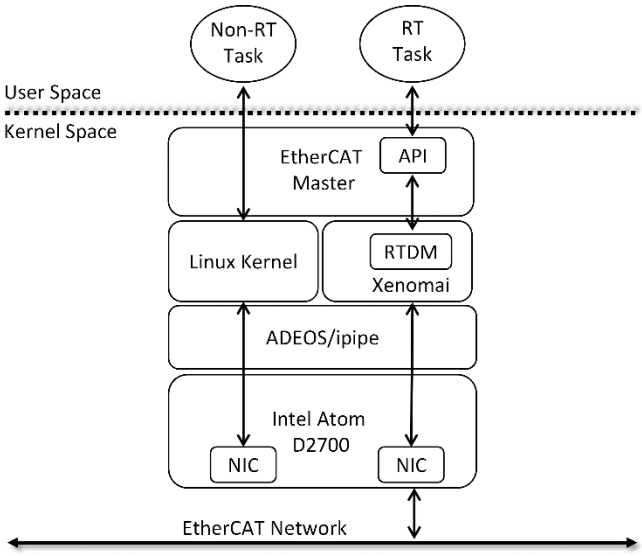

Figure 2: Dual Kernel Approach using Xenomai

In the Linux kernel these sections are protected by spin locks [10]. In addition, with a pre-emptible kernel, breaking locks to allow rescheduling is simpler than with the preemption patches. If the kernel releases a lock and then re-acquires it, when the lock is released preemption will be checked for. Massive changes to the kernel code is required to ensure real-time performance. Meaning that each kernel version requires a specific routine or patch to handle these changes.

As shown in Fig. 1. After patching the standard Linux kernel, CONFIG_PREEMPT_RT has been enabled to reduce the latency of the kernel by allowing all kernel code that is not in a critical section. The synthetic real-time task obeys the FIFO scheduling policy. The main source of timing is a periodic real-time signal originated from POSIX timers.

Dual Kernel Approach

In contrast with the pre-emptible kernel approach, the dual-kernel approach enables multiple entities called domains to exist simultaneously on the same machine. This also means that to guarantee real-time performance, massive change of the kernel code is not required.

These days, the most popular distribution of the dual-kernel approach is Xenomai. Xenomai is a real-time development framework cooperating with the Linux kernel, to provide a pervasive, interface-agnostic, hard real-time support to user space applications, seamlessly integrated into the Linux environment [11]. The basic structure of a dual-kernel system using Xenomai is shown in Fig. 2.

In 2003 it was merged with the Real-Time Application Interface (RTAI) project to produce a production-grade real-time free software platform. Eventually, this fusion effort became independent from RTAI in 2005 as the known as the Xenomai project. Hence, it also uses the ADEOS that acts as a resource virtualization layer to allow sharing of hardware resources among multiple kernel components, with Xenomai as the highest priority in the ADEOS domain.

In comparison to RTAI, the main goal of the Xenomai project is to focus more on the clean extensibility, portability, and maintainability rather than fast dispatching. It is also better structured and is available for a larger number of platforms. One of the features of interest in Xenomai is the real-time shadow services that enable the real-time thread to be migrated between the Xenomai and Linux domain. When a real-time task executes into the Linux domain, the Linux kernel as a whole inherits the real-time priority of such task, and thus competes for the CPU resourced by priority with other real-time tasks regardless of the domain they happen to belong to.

CANopen-over-EtherCAT

CANopen is a communication protocol and device profile specification for embedded systems, used in automation. In terms of the open systems interconnection model, CANopen implements the layers above and including the network layer. The standard consists of an addressing scheme and an application layer defined by a device profile. Fig. 3 shows the sequence diagram in developing a control application for a CANopen-based slave using the CoE protocol with IgH EtherCAT Master. The SDO protocol is used directly, so that existing CANopen stacks can be used practically unchanged. Optional extensions are defined that lift the 8-byte limit and enable complete readability of the object list. The process data are organized in process data objects (PDO), which are transferred using the efficient means of EtherCAT, naturally without the 8-byte limit [15]. All CANopen profiles, including the drive profile, DS 402, are fully usable, and devices based on it can be transferred to EtherCAT very easily. Slaves offer their inputs and outputs by presenting PDOs to the master. The available PDOs can be either determined by reading out the TxPDO and RxPDO SII categories from the EEPROM of the slaves or by reading out the appropriate CoE objects, if available.

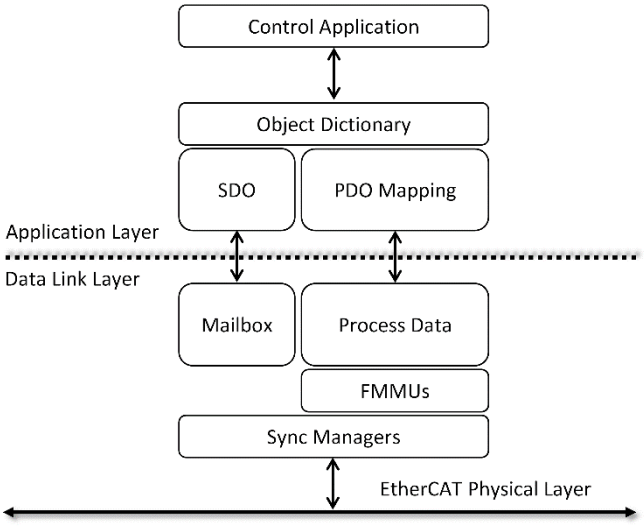

Figure 3: CANopen over EtherCAT device architecture

The application can register the PDO entries for exchange during cyclic operation. The sum of all registered PDO entries defines the PDO Mapping. An application can register PDO entries for exchange. Every PDO entry and its parent PDO is part of a memory area in the slave's physical memory that is protected by a sync manager for synchronized access. In order to make a sync manager react on a datagram accessing its memory, it is necessary to access the last byte covered by the sync manager.

Otherwise the sync manager will not react on the datagram and no data will be exchanged. That is why the whole synchronized memory area should be included into the process data image: For example, if a certain PDO entry of a slave is registered for exchange with a certain domain, one FMMU will be configured to map the complete sync-manager protected memory, the PDO entry resides in.

If a second PDO entry of the same slave is registered for process data exchange within the same domain, and it resides in the same sync-manager-protected memory as the first one, the FMMU configuration is not altered, because the desired

memory is already part of the domain's process data image. If the second PDO entry would belong to another sync-manager-protected area, this complete area would also be included into the domains process data image.

EXPERIMENT PROCEDURE

In this paper, the experiment was conducted in an EtherCAT system using the embedded hardware as shown Fig. 4. Table 1 describes the specification of each component within the system.

Table 1: Hardware specifications of the EtherCAT master

Master	
CPU	Intel Atom D2700 @ 2.13 GHz
Memory	2 GB DDR3 SDRAM
Network Card	Realtek RTL8169 Gigabit
Slaves	
Servo Drive	Sanmotion R Advanced Model
PDOs	24 Bytes for each slave (144 Bytes) (TxPDO 12 Bytes, RxPDO 12 Bytes)

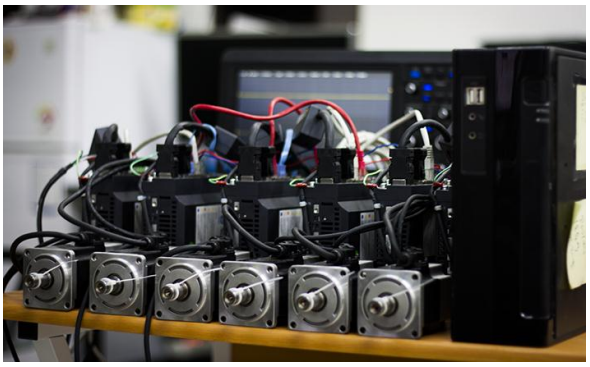

Figure 4: Experimental EtherCAT system

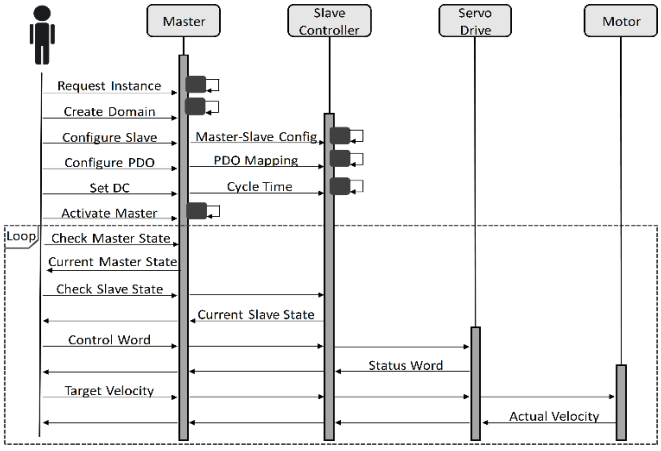

Figure 5: EtherCAT control application in UML

User Space Application

Fig. 5 shows a user space application in UML to connect the EtheCAT master to the slaves via CANopen-overEtherCAT protocol. The application is designed to request a master instance, map the process data, communicate with the slaves, and configure or activate the bus. Before a user space application can access the EtherCAT Master, an instance should be reserved for exclusive use.

Next, a process data domain is created which is used for registering PDOs and exchanging them in cyclic operation for data transaction. The application should also address the connected slaves with their proper alias, position, and identification/product code. If the data is not matched, the EtherCAT operation is halted and the slaves would not be configured. These PDOs are registered to the created process data domain. Before runtime, the user could also choose whether to enable the distributed clock (DC), if it is available on the slave device. DC is a clock synchronization mechanism making the first connected slave as the reference clock for the entire network. Then, the master will be signaled that the configuration phase is finish and that the real-time operation will be started. PDO configurations are not allowed beyond this point.

IgH EtherCAT Master is equipped with a user space library in order to manipulate EtherCAT slaves inside a real-time control application. The important functions that are required in starting the EtherCAT Master instance is organized in the order of function call before the real-time cyclic task:

ecrt_request_master: This is the first function that an application has to call to use EtherCAT that requests an EtherCAT master instance for real-time user space operation.

ecrt_master_create_domain: At least one process data domain is required for process data exchange. This object is used for registering PDOs and exchanging them in cyclic operation.

ecrt_master_slave_config: Creates a slave configuration object for the given alias and position. If the slave with the given address is found during bus configuration, its vendor ID and product code are matched against the given value. On mismatch, the slave is not configured and an error occurs.

ecrt_slave_config_pdos: This function specifies a complete PDO mapping configuration for the master to reserve the complete process data although the slave is not present at configuration time.

ecrt_domain_reg_pdo_entry_list: Registers a bunch of PDO entries for the created domain.

ecrt_slave_config_dc: Enables and configures the usage of distributed clocks.

ecrt_master_activate: This function tells the master that the configuration phase is finished and that the real-time operation will begin. PDO configurations are not allowed beyond this point.

ecrt_domain_data: This acquires the domain process data and should be called after activating the master.

From here on, the real-time task starts after the initial master and slave configuration. Both the master and the slave should be in the operational state to start data transfer. Inside the real-time task, the states of the both the master and the slaves are checked using *ecrt_master_state* and *ecrt_slave_config_state*, which reads the current master and slave states, respectively.

Table 2: Real-time Linux API and task parameters

Functions	**RT_PREEMPT**	**Xenomai**
Task Creation	pthread_create()	rt_task_create()
Set Desired Period	clock_nanosleep()	rt_task_set_periodic()
Read Timer	clock_gettime()	rt_timer_read()
Parameters	**RT_PREEMPT**	**Xenomai**
Highest Priority	80	99
Clock	CLOCK_REALTIME	
Scheduler	FIFO	

Another important sequence to follow during the cyclic task is fetching the stored datagram from the Ethernet device buffer and determining the state or the working counters of the EtherCAT frame. These are done as soon as the state of the master and slaves are checked in the order of *ecrt_master_receive* and *ecrt_domain_process*. The contents of the datagrams are copied to local variables using EC_READ_{*type*}_{*bit*}, which reads the values from an EtherCAT frame where the data {*type*} could either be S for signed and U for an unsigned integer. In addition, {*bit*} depends on the size of the data and could be either 8, 16, 32, or 64 bits. On the other hand, EC_WRITE writes the processed values to the EtherCAT frame where the used data type and size are the same as in EC_READ.

After processing the datagrams and writing the next set of commands, these are copied back to the buffer of the device and sent back to the slaves using the function, *ecrt_domain_queue*, which queues the datagrams for exchanging at the next call of *ecrt_master_send*, which sends all datagrams that are in the queue.

The control application is constructed using the provided application program interface (API) of each real-time Linux approaches. In case of RT_PREEMPT, standard Linux and POSIX APIs, such as threads and timers, are used in order to create a real-time user space task. Xenomai provides a native library that offers task and timer management with their own set of functions in accordance with typical commercial RTOS for easier migration.

Table 2 shows a summary of the common functions used to create a cyclic control task using RT_PREEMPT and Xenomai, respectively.

PERFORMANCE EVALUATION

Sung et al. (2013) [12] presented an end-to-end delay model of synchronized control processes using an EtherCAT network. Based on the analysis model, the characteristics of EtherCAT synchronized processes for varying number of slaves and process cycle time was evaluated using an open-source automation controller. Focusing on the real-time performance of an EtherCAT Master, Cereia et al. [13] presented simulation results in terms of cyclic task accuracy and varying CPU load using RTAI and RT_PREEMPT.

Although the previous studies analyzed important timing criterions for an EtherCAT network, performance differences that occur during Linux kernel update was not addressed.

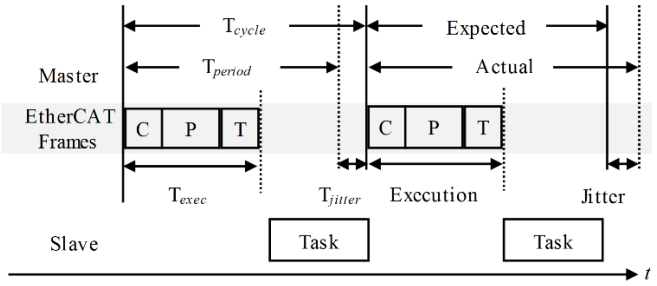

Figure 6: Timing analysis diagram

```
procedure Control_Task
set period and make task periodic
        while (1)
          read start time;
          collect process image from slave;
          process and compute next command;
          transmit process image to slave;
          read end time;
        end
wait period;
end
```

Figure 7: Pseudo code for the EtherCAT control task

Moreover, practical implementation on an actual workload was not conducted and only dealt with simulated tasks. In this paper, performance analysis is discussed in terms of the ability of the EtherCAT Master to handle cyclic control task that is bounded by a certain period, the jitter, and the in-controller execution time of an EtherCAT Master for each real-time approach graphically shown in a timing diagram in Fig. 6 using CoE protocol that is discussed in the previous section.

Here, the time duration to acquire the stored buffer from the network interface card (NIC) is represented by block C. Block P denotes in-controller processing time for the next command, and T is the time duration for sending out the frame through the NIC. The execution time, T_{exec}=C+P+T, depends on the number of slaves and the size of the PDOs.

The actual cycle time, T_{period}, is calculated by subtracting the acquired value of the current instance from the previous iteration. The task release jitter, T_{jitter}, is calculated as the difference between the constant cycle time, T_{cycle} and T_{period}.

Fig. 7 shows the pseudo code for the EtherCAT control task using CoE protocol. Task creation, parameters, and timing APIs varies depending on the real-time Linux extension used, as described in Table 2. For the dual-kernel approach, Xenomai provides its own API, while RT_PREEMPT timing is managed using native Linux and POSIX APIs.

RESULTS AND DISCUSSION

Table 3 summarizes the results of the performance analysis of each real-time approach of the EtherCAT Master that is used in our experimental system. EtherCAT Masters using different real-time Linux approaches were connected to a commercial EtherCAT servo drive manufactured by Sanmotion, a brand name of Sanyo Denki servo systems. Each slave was configured to contain 24 bytes of PDOS divided to 12 bytes each for TxPDOs and RxPDOs, respectively. All necessary commands are included especially PDO control word to drive in cyclic synchronous position, velocity, or torque mode. This measurement was performed for 60 seconds with a cyclic time of 1 ms that results to 60,000 samples.

The EtherCAT control task was set to have the highest priority of each EtherCAT Master, with 80 for RT_PREEMPT and 99 for Xenomai. As shown from the results, actual cyclic time for each master shows that although the average cyclic time for both masters are the same, Xenomai is proven to be more stable compared to RT_PREEMPT with lower maximum and minimum values. Moreover, Xenomai results prove higher accuracy as shown by the lower standard deviation, which is also proven by the denser box as shown by the distribution plot in Fig. 8.

Table 3: Summary of the performance evaluation results

T_{period} (ms)	RT_PREEMPT	Xenomai
Average	1.000000	1.000000
Maximum	1.103708	1.024125
Minimum	0.897500	0.984708
St. D.	0.002829	0.001662
T_{jitter} (µs)	**RT_PREEMPT**	**Xenomai**
Average	1.433	1.155
Maximum	103.708	24.125
Minimum	0.000	0.000
St. D.	2.439	1.195
T_{exec} (µs)	**RT_PREEMPT**	**Xenomai**
Average	94.429	96.022
Maximum	214.000	174.875
Minimum	72.625	74.833
St. D.	9.946	9.916

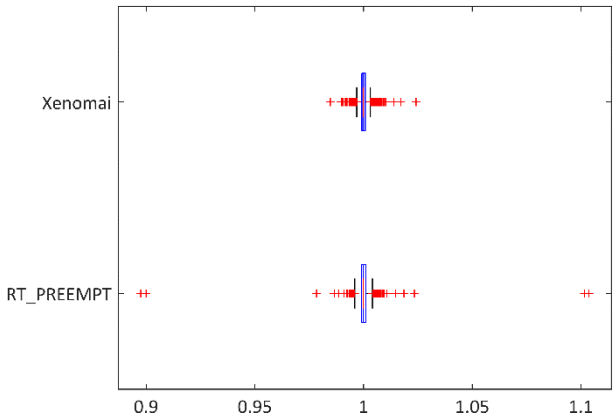

Figure 8: Distribution plot of the actual period of the cyclic EtherCAT control task for each real-time Linux approach in milliseconds

The jitter, T_{jitter}, shows results that are consistent to the actual periodic cycle results with lower values for Xenomai-based EtherCAT Master compared to one on top of RT_PREEMPT.

In case of the in-controller execution time, the given commands to the slaves are pre-generated velocity profile to follow a trajectory. These commands are stored inside a buffer to ensure that the delay would have minimal effects on the EtherCAT control task. The results show that RT_PREEMPT could process and communicate with the slaves faster than Xenomai with an average of 94.429 μs compared to 96.022 μs. Although the maximum and minimum for the system using RT_PREEMPT shows more distance to the mean than Xenomai, the standard deviation is more stable with 9.946 µs to 9.916 μs.

With the results in the table, the jitter of the system and the overall in-controller execution time of each master are statistically under tolerable range in this development environment. We have observed the results using a system with multiple slaves and that the size of the PDOs of the CoE-based EtherCAT Slave is another factor during computation of in-controller execution time.

Another important issue to consider is the reliability of the RT_PREEMPT patch. The patch is still an ongoing project which is still unstable in terms of support and performance resulting to different results for different Linux versions and subversions.

CONCLUSION

In this paper, we have conducted a real-time servo control using the open-source solution, IgH EtherCAT Master. The solution was implemented under two different real-time Linux extensions of RT_PREEMPT and Xenomai for pre-emptible kernel and dual kernel approaches, respectively.

The results show that the dual-kernel approach using Xenomai has performed to have more accurate cyclic task periodicity compared to an EtherCAT Master based on RT_PREEMPT however, in terms of execution time, the latter has proven to be faster than the former.

Aiming for a standard measurement, for the design and predictability of an EtherCAT Master on different real-time Linux extensions, we have dealt with implementation of two different EtherCAT masters and discussed its performance results.

For our future research, we would extend our analysis to deal with multiple slaves with different data types and sizes. Focused research on the RT_PREEMPT is also planned by comparing results from different versions of the real-time Linux approach that would prove its viability for an industrial automation system.

REFERENCES

[1] E. Rodriguez-Seda, C. Tang, M.W. Spong, and D.M. Stipanovic, "Trajectory tracking with collision avoidance for nonholonomic vehicles with acceleration constraints and limited sensing," International Journal of Robotics Research, vol. 33, no. 12, pp. 1569-1592, 2012.

[2] R. Delgado, C.H. Hong, W.C. Shin, and B.W. Choi, "Development and control of an omnidirectional mobile robot on an EtherCAT network," International Journal of Applied Engineering Research, vol. 11, no. 21, pp. 10586-10592, 2016.

[3] Y. Moon, N.Y. Ko, K. Lee, Y. Bae, and J.K. Park, "Real-time EtherCAT master implementation on Xenomai for a Robot System," International Journal of Fuzzy Logic and Intelligent Systems, vol. 9, no. 3, pp. 244-248, 2009.

[4] Rostan, M. and J.E. Stubbs, 2010. EtherCAT Enabled Advanced Control Architecture. Proceedings of the 2010 IEEE/SEMI on Advanced Semiconductor Manufacturing Conference, July. 11-13, IEEE Xplore Press, San Francisco, CA, pp: 39-44.

[5] ETG, 2014. Introduction to EtherCAT by EtherCAT Technology Group. https://www.ethercat.org/en/downloads/downloads_0F6203AB7DE44BDB93DEC585A581A7D7.htm

[6] Ferrari, P., A. Flammini, D. Marioli and A. Taroni, 2008. A distributed instrument for performance analysis of real-time Ethernet networks. IEEE Transactions on Industrial Informatics, 4: 16-25.

[7] Vitturi, S., L. Peretti, L. Seno, M. Zigliotto and C. Zunino, 2011. Real-time Ethernet networks for motion control. Computer Standards & Interfaces, 33: 465-476.

[8] L. Seno, and C. Zunino, "Real-Time Ethernet networks evaluation using performance indicators," Proc. 14th IEEE Int'l Conf. Emerging Technologies and Factory Automation (ETFA), 2009.

[9] J. Arm, Z. Bradac, and V. Kaczmarczyk, "Real-time capabilities of Linux RTAI," IFAC Conference on Prgrammable Devices and Embedded Systems, 2016.

[10] D.B. Oliveira, and R.S. Oliveira, "Timing analysis of the PREEMPT RT Linux kernel," Software Practice and Experience, vol. 46, no. 6, pp. 789-819, 2015.

[11] B.W. Choi, D.G. Shin, J.H. Park, S.Y. Yi, and S. Gerald, "Real-time control architecture using Xenomai for intelligent service robot in USN environment," Intelligent Service Robotics, vol. 2, no. 2, pp. 139-151, 2009.

[12] M. Sung, I. Kim, and T. Kim, "Toward a holistic delay analysis of EtherCAT synchronized control processes," International Journal of Computers, Communications and Control, vol. 8, no. 4, pp. 608-621, 2013.

[13] M. Cereia, I.C. Bertolotti, and S. Scanxio, "Performance of a real-time EtherCAT Master under Linux," IEEE Transactions on Industrial Informatics, vol. 7, no. 4, pp. 679-687, 2011.

[14] IgH EtherLab, http://www.etherlab.org/en/index.php

[15] F. Pose, IgH EtherCAT Master 1.5.2 Documentation. 2013.