International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 8 (2017) pp. 1675-1682
© Research India Publications. http://www.ripublication.com

Comparative Analysis of a User Space EtherCAT Master Application for Hard Real-time Control

Raimarius Delgado
Ph.D. Student,
Department of Electrical and Information Engineering,
Seoul National University of Science and Technology,
Seoul, South Korea.

Byoung Wook Choi*
Professor,
Department of Electrical and Information Engineering,
Seoul National University of Science and Technology,
Seoul, South Korea.

Abstract
In this paper, we evaluate the real-time performance of an open source EtherCAT master developed by IgH Etherlab on Xenomai, running as a real-time co-kernel of standard Linux. The environment is implemented on top of two embedded platforms with different architectures, ARM and x86. First, Xenomai is ported on the target architecture in order to achieve real-time characteristics and to provide real-time mechanisms for inter-process communication used by the EtherCAT master. Although applications that run in the kernel space give better performance, programming is very difficult due to limited user-friendly functions. It is also fragile that a single mistake can easily crash the whole system. On the other hand, user space applications are uncomplicated and debugging is also simplified, with lesser risk of damaging hardware or reference memory. Thus, we developed a real-time application in the user space of Xenomai to actuate multi-axis servo drives focusing on the periodicity and execution time as performance measurements. The execution time of the EtherCAT master is evaluated and compared for each platform by calculating the difference between the timestamps during send and receipt of the EtherCAT frames. The experiment is performed both in short-term and long-term to test the consistency of each platform in carrying out hard real-time control applications.

Keywords: EtherCAT, Xenomai, Embedded Linux, Real-time, Performance analysis.

Introduction

The popularity of EtherCAT has been steadily increasing due to the growing demands of real-time controllers that do not need customized interface cards [1]. In comparison to other real-time Ethernet network protocols, EtherCAT has shown the best performance in terms of response time and communication jitter in accordance to the standard speed. This trend led to various interesting researches related to both the implementation and evaluation of low-cost solutions using an open source EtherCAT master [2-4].

Such solution requires to be implemented in a software framework that provides real-time mechanisms for data handling and communication. IgH EtherCAT master is modelled after the standard Linux, thus simplifying programming for developers with experience working in the similar environment. However, the standard Linux scheduler determines the most worthwhile process to run valuing fairness over priority. This behavior limits Linux to be used directly for real-time applications that needs strict timing determinism for multiple tasks that offers dynamic priorities and task preemption. In order to solve the problem above, the scheduler of the standard Linux is modified to meet real-time requirements. The most popular real-time Linux approaches available are RT_PREEMPT, RTAI, and Xenomai [5-7].

According to a study conducted by Barbalace et al. [8], RTAI possesses the best performance among the three using interrupt latency and rescheduling jitter of periodic tasks as the test metrics. But in this paper, we decided to build a working environment based on Xenomai for it supports more processor architectures and embedded platforms as opposed to RTAI and RT_PREEMPT. Better performance in data communication and synchronization between tasks using real-time mechanisms is also expected as shown by the results of evaluation performed by Koh et al. [9] using the jitter of periodic tasks and response time of various real-time mechanisms as performance indicators.

The IgH EtherCAT master was designed to support all of the aforementioned real-time Linux approaches and can be developed either in kernel or user space. Control applications in the kernel space optimizes performance for it avoids the application from frequently switching between kernel space and user space and allows direct communication with the physical layer of the network interface. However, high mathematical complex algorithms are not solved through a straightforward manner because calculations using floating points and other user space libraries are inaccessible [10]. Kernel space is also very sensitive because it normally has full access to all memory and all the components within the machine [11-12]. It is advisable that only the most trusted and well tested code is run in the kernel space to avoid crashing the whole system. Thus, the performance of the EtherCAT master in user space is required to be tested using the selected real-time Linux extension.

The viability of such solution on an embedded platform has been presented in our previous research [13], where we have conducted a performance analysis of the IgH EtherCAT master on an ARM-based platform running on Xenomai. We have also provided detailed guidelines to build the working environment on which most developers have had a hard time due to lack of technical documents. However, the results

* Corresponding author

were only displayed for an application with a very short runtime. Most of the time industrial controllers are left unattended for long hours and are still expected to perform the time-critical tasks. Practical feasibility of the real-time controller for such applications is highly dependent on the performance results during long-term operation [14].

In this paper, first we implemented IgH EtherCAT and Xenomai on two different embedded platforms with processors based on x86 (Intel) and ARM (i.MX6Q), respectively. The same versions of the software and Linux kernel were applied for the sake of comparison minimizing the probable causes of varying results. Since both platforms use different network interface card, we have also added an external NIC with the same chipset as in Intel to i.MX6Q platform to further minimize the point of comparison. This method has been chosen since it is easier to find off-the-shelf NICs with the identical chipsets to those that are installed in Intel platforms than the opposite. Then we developed a real-time application running on the user space of Xenomai to actuate six servo drives connected in daisy chain topology using velocity commands that are stored in a buffer for lower calculation inside the control task with the highest priority. The performance of the embedded platforms is evaluated and compared using periodicity of the real-time task and EtherCAT execution time as demonstrated in our previous research in [13]. The time interval introduced between the send and receive routine of EtherCAT frames from the buffer of the NIC determines the overall real-time performance of the system. The experiment is conducted for the target platforms in both short-term and long-term test with a runtime of 1 minute and 12 hours, respectively.

The second section fully discusses the architecture of IgH EtherCAT master. Section 3 describes the development of the real-time application in Xenomai user space. The fourth section explains how the experiment is conducted. This section is divided into two parts: describing the developed testbed for both platforms and the experiment results from both short-term and long-term operations. The last section closes the paper with the conclusion and discussion of future work.

IgH EtherCAT Master Architecture

IgH EtherCAT Master is designed to be integrated into the Linux kernel for performance purposes. Process in the kernel space has better real-time performance and lesser latency than processes executed in the user space. If the cyclic task which is triggered by timer interrupts inside the kernel is in the kernel space, the execution delay of processing these interrupts is less because there is no need of time consuming context switches. Moreover, the master module can directly communicate with the physical layer of the Ethernet hardware through device drivers, which are already in the kernel space. Fig. 1 gives an overview of the components inside the master architecture.

In the figure, we can easily determine that the EtherCAT master is divided into three main parts: EtherCAT master module, device drivers, and real-time tasks. The master module is the core of all EtherCAT operations. It handles master instances which can run simultaneously, opening doors to redundant EtherCAT master applications. The master module has three operating states: orphaned state, idle state, and operation state. In the orphaned state, the master is still waiting for an Ethernet device to be connected through its device interface. The master is in the idle state when the master module is connected to at least one Ethernet device but is still waiting for a request from an application.

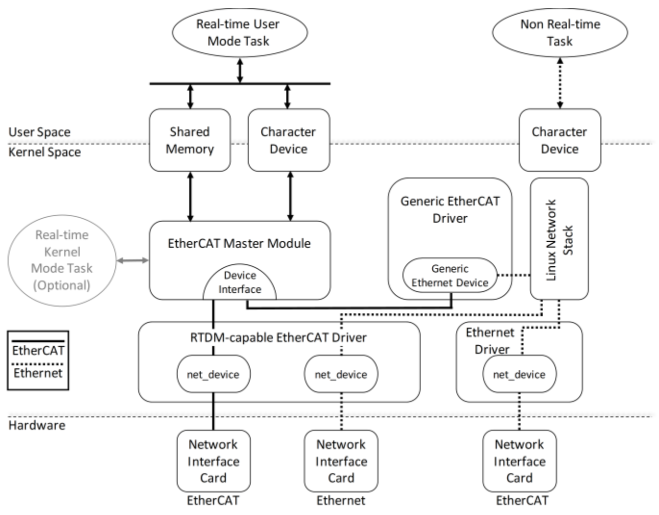

Figure 1. IgH EtherCAT Master Architecture

Finally, the master is in operation state when at least a single master instance is requested by an application. Process domain object (PDO) exchange and bus configuration is only possible when the master module is in the state of operation.

Device drivers connects the physical layer of the NIC to the master module via the device interface. Drivers that are accepted by the master module and can handle EtherCAT operation are classified into two: RTDM-capable EtherCAT driver and generic EtherCAT driver. RTDM, or real-time driver model, are modified network device driver for dedicated interaction with the master module. During EtherCAT operation, these drivers are not required to pass the standard Linux network stack resulting to highly deterministic communication. On the other hand, generic drivers have to transverse the Linux network stack resulting to worse performance. These drivers also could not guarantee support for real-time extensions like Xenomai, because the Linux network stack is addressed and could cause mode switching in real-time context. The upside is that through the generic driver, any Ethernet drivers that are supported by Linux can be used for EtherCAT operation.

The real-time tasks are either executed inside the kernel space, or in the user space through the provided user space library. Real-time task in the user space can choose to connect through a character device or shared memory. In case of the former, standard Linux system calls are used, thus, mode switching will occur with the same effects as using the generic driver. Using shared memory can guarantee better performance, but this method only works for device drivers that are RTDM-capable.

Real-time Control Application in User Space

The latest version of IgH EtherCAT Master is equipped with a user space library that provides users to develop control applications in the user space of real-time Linux, which is Xenomai for this study. Because functions that are mainly required to configure the communication between the master and the slave used standard Linux system calls, it is advised that these functions are called before starting the real-time task. The EtherCAT master module is also required to be activated before the Xenomai task is started.

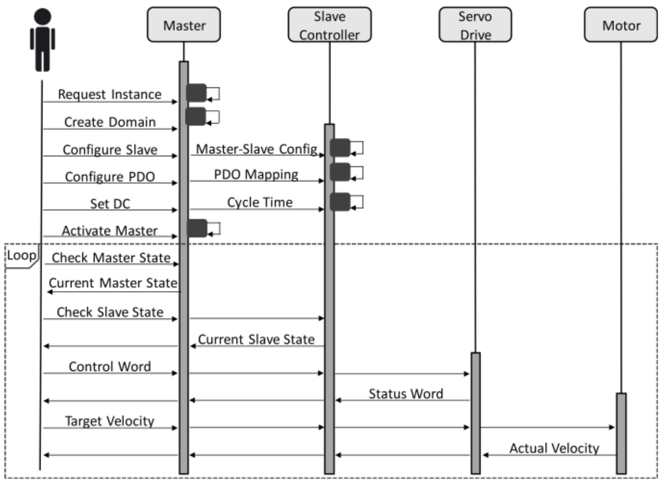

Figure 2. Real-time User Space Application in UML

Fig. 2 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 [4]. 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.

The master is signaled that the real-time is starting, the domain process data should be acquired before starting the cyclic task. The real-time task starts after the initial master and slave configuration.

The important functions offered by the IgH EtherCAT master library that are required in starting a user space application is organized with respect to the figure above:

ecrt_request_master: Requests an EtherCAT master instance for real-time user space operation.

ecrt_master_create_domain: Creates a domain 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. When the given values do not comply with the information stored in the slave's E^{2}PROM, an error will occur and the application will be halted.

ecrt_slave_config_pdos: Specifies a complete PDO mapping configuration for the master to reserve. The data that will be exchanged during cyclic task is according to the PDO map.

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: Signals the master that the real-time operation will begin.

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

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.

Another important sequence to follow during the cyclic task is acquiring 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. The datagrams 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 after processing the datagrams and writing the next set of commands.

In this paper, we will use Xenomai to initiate a real-time control task. Xenomai offers various kinds of interfaces and emulators to resemble traditional real-time operating systems (RTOS) such as VxWorks, uItRON, or VRTX. Although migration and compatibility is not an issue in application development for those emulators, it is better to make full use of the high integration level with the Linux environment. In such, Xenomai provides two programming interfaces or skins for this method: real-time extension of POSIX API and the Native API. The latter is equipped with the same set of real-time services in a seamless manner to applications both in the user space and kernel space making it context independent. Xenomai objects and mechanisms are always reachable even when the location of the object is in another process or space than were the task using it is running. The Native API was designed to resemble after a traditional RTOS with different services for managing real-time tasks, timing, synchronization, device handling, and inter-task communication mechanisms [15].

To create a real-time task using the Native API, a descriptor referring to a task is created using a structure called RT_TASK. This structure describes all the essential information about the task such as its priority, CPU to be used, stack size, and the function to be executed. The task is created by a call to the function *rt_task_create* with arguments including the pointer to the descriptor created earlier using RT_TASK, symbolic name in ASCII string in order for the task to be called in other processes or space, memory stack size, priority of the task with 99 as the highest and 0 is the lowest, and flags which configures: whether the task will use the floating point unit (FPU), which CPU to use, or will the task be immediately started upon creation.

After the task is created, it can be started by calling the function *rt_task_start* specifying the task to be started using the descriptor, the function that the start will execute when started, and the arguments given to the function. Unless the task is created in the suspended mode, the function will be executed immediately after started. When the task is started, any call to a Linux API is avoided to avoid switching modes. In the case of a mode switch the Linux scheduler will be the one to handle the task degrading its performance, thus, trading off determinism. Initially, a task is started in one shot mode, meaning that the periodicity of the task is not guaranteed. For the control task to be scheduled in deterministic manner, a call to *rt_task_set_periodic* is required. The function accepts arguments pertaining to the task descriptor, starting time of the task in absolute time which is expressed in clock ticks. When the task is executed immediately without any delay, a macro TM_NOW is passed as the argument. The last configuration to be set is the period of the task, which accepts values in nanoseconds. Inside the function of the real-time task, a call to *rt_task_wait_period* is required to release the processor when the task has performed all its instruction and wait for the next scheduled starting point.

For the measurement of the periodicity and to guarantee that the task can meet the configured deadline, timer management service provided by the Native API is used. The timer related services allow to control the Xenomai system time which is used in all timed operations. This serves as a probe that returns the system time at the calling point of *rt_timer_read*. This is used to measure the execution time, starting time, and end time of the task for the performance analysis which is conducted during experiment for this study.

Experiment

A. Testbed

For comparison of real-time performance, experimental testbeds were set up using two embedded platforms based on x86 and ARM architecture. IgH EtherCAT master and Xenomai were implemented on each target architecture and both masters are connected to six servo drives manufactured by Sanyo Denki as shown in Fig. 3. The latest IgH EtherCAT master available is v1.5.2 and v2.6.5 is for Xenomai 2.x series. Both environments were stacked on the same version of Linux, v3.14.15, which are available respectively in Linux kernel archives [16] and armv7-multiplatform for Intel and i.MX6Q [13].

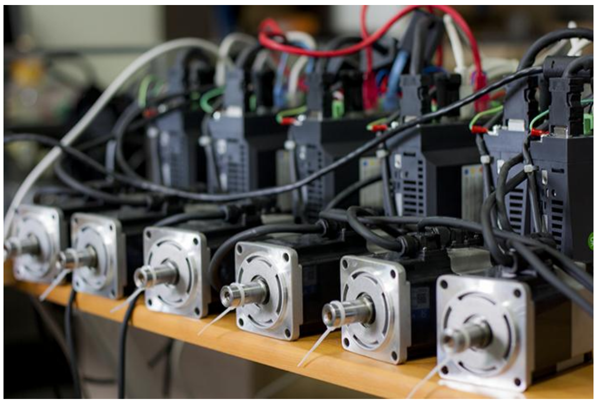

Figure 3. Sanyo Denki EtherCAT Slaves

Table 1. Specifications of Embedded Platform

Specifications	Intel	i.MX6Q
CPU	Intel Core i7 6700 @4.00 GHz	ARM Cortex A9 @1.00 GHz
Memory	8 GB DDR4	1 GB DDR3
NIC	RTL8169 Gigabit	FEC Gigabit

Detailed specifications which mainly focuses on the central processing unit (CPU) model and speed, memory capacity, and NIC is compared and shown in Table 1. With this information, we could expect that the performance of Intel is better than i.MX6Q with faster CPU frequency at 4 GHz compared to 1 GHz although both possess 4 CPU cores. Multitasking will also be smoother with 8 GB of available memory in comparison to 1 GB which is not extendable. But real-time performance is more on the ability to guarantee response time within a given time constraint rather than raw speed. In other words, real-time is the ability to meet specific deadlines in yielding desired output instead of the number of instructions processed in a given amount of time. Another component that could be affect the real-time performance of EtherCAT is the NIC. Intel uses RTL8169 (R8169) on the main board in contrast to the Fast Ethernet Controller (FEC) Gigabit module based on Motorola MPC8xx installed on the i.MX6Q. Both modules are capable of transmitting Ethernet frames at the maximum rate of 1 Gigabit per second.

To furtherly minimize the points of comparison for both platforms, an external R8169 is attached to i.MX6Q which has an external port for mini PCI Express (mPCIe) devices. It is easier to find off-the-shelf NICs with R8169 chipset than those with FEC. In order to connect the external NIC to the embedded platform, a mPCIe daughterboard is required to be connected. The daughterboard uses an on-board clock to implement the latency timer that limits the time that device can hold the bus. Commonly available R8169 NICs are for standard personal computers that has PCIe connectors, thus, another daughterboard that converts mPCIe-to-PCIe is also required. Finally, the external R8169 is connected to the mPCIe-to-PCIe daughterboard which is again connected to PCIe daughterboard. The i.MX6Q with an external R8169 is shown in Fig. 4. Although this approach has minimized the probable causes of performance differentiators between the two embedded platforms, it resulted to a bulkier system which could be unstable due to different levels of connections and data conversions. Hardware delay could also have an impact to the real-time performance of i.MX6Q.

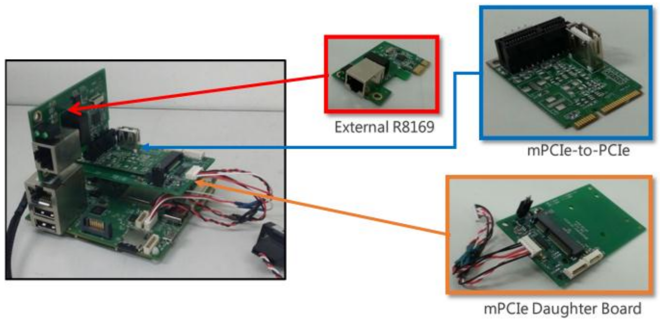

Figure 4. i.MX6Q with External R8169

B. Procedure and Results

The experiment is conducted in the user space of Xenomai by creating a real-time task with a sampling period of 1 ms and the highest priority of 99. We also used the hardware FPU for each embedded platform, and a single core for task execution since Xenomai supports symmetric multiprocessing (SMP), all the resources of a multi-core system are guaranteed to be exploited even using a single core.

The total size of PDO transmitted for each iteration is configured to a total of 96 bytes, classified into 10 bytes of input and 6 bytes of output for each slave. The input includes 2 bytes for the status word which governs the current status of the servo drive according to the CiA 402 profile of the CANopen protocol state machine [17-18]. 4 bytes are allocated for the current position which is the feedback from the absolute encoder attached on the motors and another 4 bytes for the current velocity, on which the calculation is executed by the servo drive with interpolation. The output PDOs are divided into 2 bytes for the control word or the target state to operate the servo controller and 4 bytes for the target velocity. We have configured the servo drives to operate in velocity mode by changing the value stored inside the register that corresponds to the operation mode.

Before the Xenomai user space task is started, the PDOs are configured and one IgH EtherCAT master instance is initiated. Also, velocity profile for each of the servo drives, which is generated with a sampling time of 1 ms is stored arbitrarily inside a buffer to minimize the computation load inside the real-time task. The velocity profile considers the maximum angular velocity of 2 rad/s as shown in Fig. 5. The whole profile has a duration of 40 s, running on different direction every 10 seconds. First, the motor is actuated in the clockwise direction for 10 s, rest for 10 s, run on the opposite direction, and rest again for another 10 s. This profile is executed continuously until the target runtime of the experiment is met.

The same application is enacted on each embedded platform, with i.MX6Q running using the on-board FEC module and another with an external R8169. The short-term experiment is carried out for 1 minute with a runtime of 1 minute and the long-term experiment running for 12 hours.

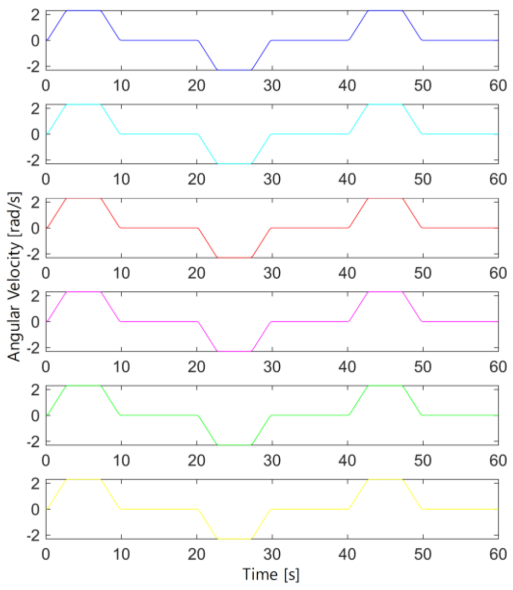

Figure 5. Velocity Commands for Each EtherCAT Slave

The actual period is calculated by subtracting the time stamp of the current instance from the previous iteration. Then the jitter is calculated by subtracting the expected time cycle to the calculated actual period. The timestamps between the sending and receiving routine of IgH EtherCAT master is subtracted to acquire the execution time. A detailed explanation of this process can be found in [13].

The results of the performance metrics are analyzed and tabulated in the criteria of the maximum, minimum, average, and standard deviation. The results of the short-term experiment are calculated directly since only a few data samples are generated during the experiment. The jitter shows that the i.MX6Q with external R8169 shows the best performance with only a maximum value of 4.008 µs in comparison to the 17.629 µs of Intel platform. This is unexpected because Intel has better specifications i.MX6Q, this only shows that high-performance is necessarily equal to real-time performance. However, the i.MX6Q using the on-board FEC using the generic EtherCAT driver gives the worst performance with a staggering 205.061 µs of maximum jitter. Although running on the same platform, we could predict that using a different driver affects the real-time performance of the whole system. But we could also see that the standard deviation is better by one-tenth of a microsecond for Intel with 0.250 µs compared to 0.369 µs of i.MX6Q. This means that Intel manages to meet the deadline more frequently than i.MX6Q. Fig. 6 shows the distribution plot for the actual period. The figure proves that the i.MX6Q with external R8169 has the best performance during the short-term experiment.

Table 2. Timing Analysis Results for the Short-term Experiment

Board	**Metric**	**Criterion (μs)**			
		Max.	Min.	Ave.	St. D.
Intel	Jitter	17.629	0	0.069	0.250
	Exec	22.141	5.331	6.214	0.674
i.MX6Q (R8169)	Jitter	4.008	0	0.261	0.369
	Exec	77.667	44.667	52.722	1.739
i.MX6Q (FEC)	Jitter	205.061	0	0.436	1.933
	Exec	1100.245	89.401	106.353	8.665

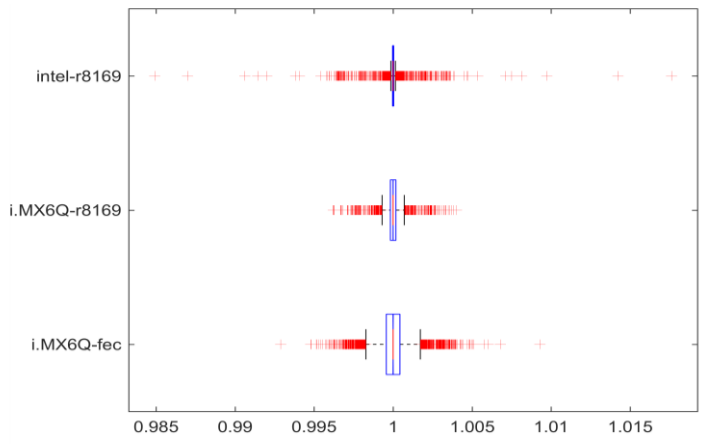

Figure 6. Comparison of Distribution Plot of the Real-time Task in Short-term Experiment

Table 3. Timing Analysis Results for the Long-term Experiment

Board	**Metric**	**Criterion (μs)**			
		Max.	Min.	Ave.	St. D.
Intel	Jitter	24.628	2.098	10.638	4.177
	Exec	35.861	9.382	17.253	4.784
i.MX6Q (R8169)	Jitter	12.667	3.333	6.485	1.339
	Exec	107.735	78.722	82.581	1.443
i.MX6Q (FEC)	Jitter	276.333	2.333	7.124	9.595
	Exec	1101	75.333	113.313	137.076

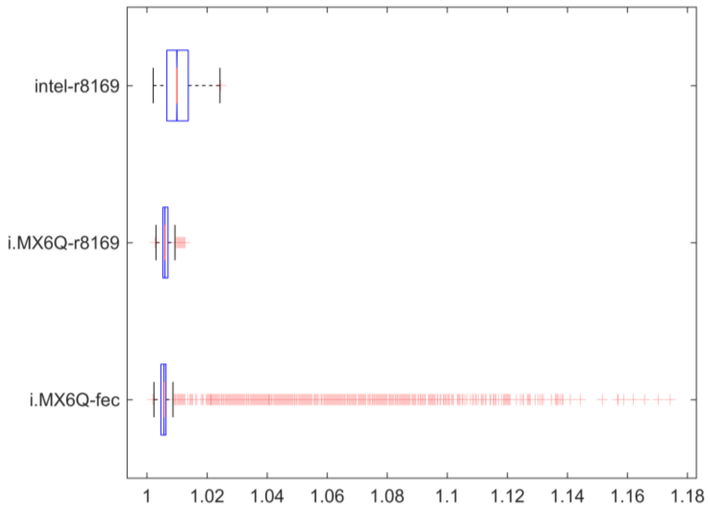

Figure 7. Comparison of Distribution Plot of the Real-time Task in Short-term Experiment

Using the same criteria, the results in the long term experiment is tabulated in Table 3. In comparison to the short-term experiment, only the worst-case values are calculated for the experiment in long-term due to memory limitations of the system. Meaning, only the maximum values for each minute during runtime are stored and analyzed. The results show that i.MX6Q with R8169 still has the best real-time performance with 12.667 μs maximum jitter compared to 24.628 μs of Intel platform. The standard deviation also becomes better at 1.339 μs. As shown in Fig. 7, i.MX6Q with R8169 has the best real-time performance in meeting the deadline of the periodic control task.

But in this study, we focus more on the performance of IgH EtherCAT master that is working inside the real-time control task of Xenomai user-space. Looking at the execution time of both Tables 2 and 3, Intel platform shows the best performance in both the short-term and long-term experiments, with maximum execution time of 17.629 μs and 24.628 μs, respectively. Intel platform also has tighter squared difference from the mean with a standard deviation of 0.250 μs and 4.177 μs. The next best is the i.MX6Q with external R8169. Although having the better real-time performance in terms of its periodicity and using the same NIC with Intel platform, processing the IgH EtherCAT master shows higher execution time with maximum of 77.667 μs and 107.735 μs respectively for short-term and long-term experiment. From these results, we could assume that the real-time performance is affected by the number of data conversions and connections to different daughter boards that could introduce hardware delay to the whole system. The standard deviation still shows that the i.MX6Q with R8169 can meet the deadline with lower values than the EtherCAT master using Intel platform.

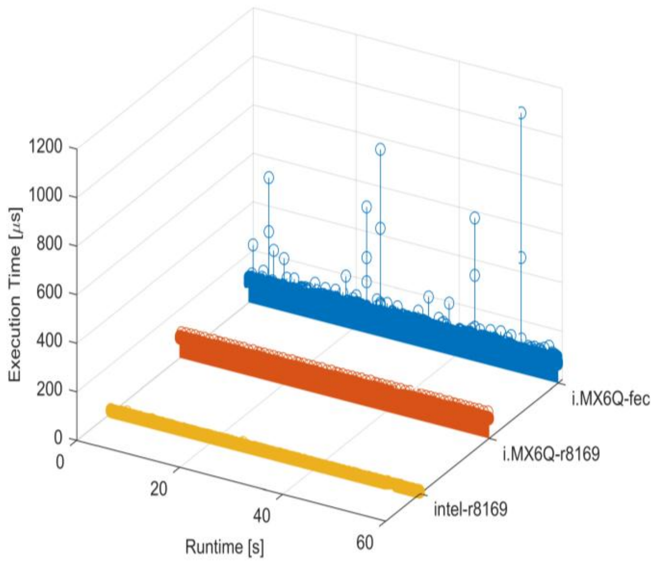

Figure 8. Execution Time in Short-term Experiment

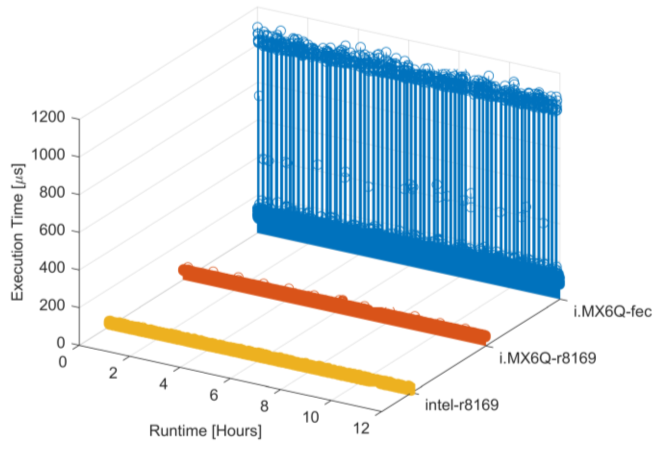

Figure 9. Execution Time in Long-term Experiment

Fig. 8 and Fig. 9 show the execution time distribution of the EtherCAT task in the short-term and long-term experiments. The figures show the expected results from Intel and i.MX6Q with r8169 having the best and second best performance. However, the i.MX6Q with the on-board FEC driver that uses EtherCAT generic driver shows unreliable results with values over than the expected cycle task period of 1ms. This only means that the platform is not applicable for real-time control systems. This is due to the fact that the EtherCAT generic driver uses the standard Linux stack which switches the mode of the Xenomai task from primary to secondary. This means that instead of being handled by the real-time scheduler, the standard Linux scheduler took governance of the task which makes it unpredictable and run in lower priorities.
Therefore, using the Intel platform for real-time applications shows the best performance for real-time EtherCAT applications. The ARM-based i.MX6Q platform is a better choice for real-time cyclic tasks, but to achieve better performance to accommodate the requirements of IgH EtherCAT master it is advisable to use an external device that is RTDM-capable such as R8169 connected to the mPCIe port. That is with a trade-off of a bulkier system which could be unstable due to different levels of connections and data conversions. Hardware delay could also have an impact to the overall latency of the real-time task.

Conclusion

In this study, we performed a comparative performance analysis of an open source EtherCAT master provided by IgH EtherCAT master on different embedded platforms based on x86 and ARM, respectively referred to as Intel and i.MX6Q. In order to achieve real-time operability, a co-kernel approach of Xenomai is ported to each target architecture to provide real-time mechanisms required by the EtherCAT master.

The performance of each embedded platform is measured in terms of the periodicity of the cyclic task and the execution time of the EtherCAT master in the user space of Xenomai. To reduce the points of comparison, it is ensured that the software on each platform is of the same version. Also, an external NIC is also added to the i.MX6Q to further reduce the probable performance comparison factor. The same application is run for the two platforms, with the i.MX6Q executed twice for the generic EtherCAT driver and external R8169 device driver.

Results show that although having a high-performance specification, real-time performance of i.MX6Q with external R8169 is better than the Intel platform. But due to different levels of data conversion and hardware delay, EtherCAT execution time gives an opposite result. i.MX6Q with the on-board FEC using EtherCAT generic driver gives the worst results in both periodicity and EtherCAT execution time due to the mode switching that triggers the standard Linux scheduler to handle the Xenomai task, making it unable to handle real-time constraints. Thus, i.MX6Q is only viable for real-time control applications using open source EtherCAT master if an external NIC is connected, with the drawback of a bulkier system and requires more extra connectors.

The results of this study will serve as a guide for real-time developers interested in building EtherCAT controllers based on open source software using Linux and the real-time extension, Xenomai, on embedded platforms based on different CPU architectures.

Acknowledgment

This study was supported by the Research Program funded by the SeoulTech(Seoul National University of Science and Technology).

References

[1] S. Vitturi, L. Peretti, L. Senio, M. Zigliotto, and C. Zunino, “Real-time Ethernet networks for motion control,” Computer Standards & Interfaces, vol. 33, no. 5, pp. 465-476, 2011.

[2] 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.

[3] 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.

[4] G. Cena, I.C. Bertolotti, S. Scanxio, A. Valenzano, and C. Zunino, “Evaluation of EtherCAT distributed clock performance,” IEEE Transactions on Industrial Informatics, vol. 8, no. 1, pp. 20-29, 2012.

[5] 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.

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

[7] 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.

[8] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taiercio "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application," IEEE Transactions on Nuclear Science, vol. 55, no. 1, pp. 435-439, 2008.

[9] J.H. Koh, and B.W. Choi, "Real-time performance mechanisms for RTAI and Xenomai in various running conditions," International Journal of Control and Automation, vol. 6, no. 1, pp. 235-246, 2013.

[10] H. Yoon, J. Song, and J. Lee, "Real-time performance analysis in Linux-based robotic systems," In 11th Linux Symposium, 2009.

[11] A. Robbins, *Linux Programming by Example: The fundamentals*. Upper Saddle River, NJ: Prentice Hall, 2004.

[12] D. Abbott, *Linux for Embedded and Real-Time Applications*. Newton, MA: Butterworth-Heinemann, 2003.

[13] R. Delgado, C.H. Hong, W.C. Shin, and B.W. Choi, "Implementation and performance analysis of an EtherCAT Master on the latest real-time embedded Linux," International Journal of Applied Engineering Research, vol. 10, no. 24, pp. 44603-44609, 2015.

[14] S. Yin, H. Luo, and S.X. Ding, "Real-time implementation of fault-tolerant control systems with performance optimization," IEEE Transactions on Industrial Electronics, vol. 61, no. 5, 2014.

[15] Xenomai Native API, https://xenomai.org/api-reference/

[16] Linux Kernel Archives, https://www.kernel.org

[17] K. Erwinski, M. Paprocki, L.M. Grzesiak, K. Karwowski, and A. Wawrzak, "Application of Ethernet Powerlink for communication in a Linux RTAI Open CNC system," IEEE Transactions on Industrial Electronics, vol. 60, no. 2, 2013.

[18] C. Zhou, and F. Luo, "Design of redundant CAN network based on CANopen," Proc. IEEE Int'l Conf. Natural Computation, Fuzzy Systems, and Knowledge Discovery (ICNC-FSKD), 2016.