Kernel-Level ARINC 653 Partitioning for Linux

Sanghyun Han
Dept. of Computer Science & Engineering
Konkuk University
Seoul, Korea
whacker@konkuk.ac.kr

Hyun-Wook Jin
Dept. of Computer Science & Engineering
Konkuk University
Seoul, Korea
jinh@konkuk.ac.kr

ABSTRACT

The Integrated Modular Avionics (IMA) architecture has been suggested for the next-generation avionics systems. ARINC 653 is the standard for application programming interfaces (APIs) of avionics software for IMA architecture. There are several researches on design and implementation of ARINC 653 but legacy operating systems have not been considered much for a base operating system of ARINC 653. Though the legacy operating systems may not be initially developed for avionics systems, some of them including Linux recently show high potential of providing software platform for avionics systems. In this paper, we suggest a kernel-level design to support partitioning and hierarchical real-time scheduling of ARINC 653 for Linux. We believe that our suggestion can provide a very valuable reference for extending an existing operating system for ARINC 653 especially due to the complexity of the Linux kernel. We show that the overhead and jitter of the proposed design is significantly low compared with a user-level design.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design – *real-time systems and embedded systems*

General Terms

Performance, Design, Experimentation

Keywords

ARINC 653, Partitioning, Hierarchical Scheduling, Linux

1. INTRODUCTION

The most of current-generation avionics systems are based on the federated architecture [20], where an electronic device runs a single software module or application and collaborates with other devices through network. As the number of electronic devices in the avionics systems keeps increasing, the internal system architecture becomes very complicate and hard to handle with respect to integration, test, and maintenance. Thus, it is desirable to run several avionics applications or modules on a single computing device so that we can simplify the network cabling and reduce weight. The avionics applications are, however, usually developed by different organizations independently and not provided with an environment for such seamless integration on a single device. Therefore, providing an efficient way to integrate separate avionics applications on a computing device is a very critical issue for the next-generation avionics systems as the Integrated Modular Avionics (IMA) concept addresses [27].

The IMA architecture aims to transparently integrate several avionics applications or modules developed by separate organizations on the same computing device while the applications communicate each other without knowing whether others are running on the same node or not. This enables the applications to be reused on different avionics systems together with other sets of applications. ARINC 653 is the standard for application programming interfaces (APIs) of avionics software for IMA architecture [1]. The standard includes the APIs for scheduling control, communication, and manipulation of status information.

Recently, there are several researches on design and implementation of ARINC 653 but legacy operating systems have not been considered much for a base operating system of ARINC 653. Though these operating systems are not initially developed for avionics systems, some of them show high potential of providing software platform for avionics systems in many examples. In this paper, we deal with the design and implementation issues of ARINC 653 at the Linux kernel-level. Especially, we focus on the partitioning and hierarchical real-time scheduling defined by ARINC 653. In order to reduce the overhead and jitter, we suggest a kernel-level design. We believe that our suggestion can provide a very concrete reference for extending an existing operating system for ARINC 653. We evaluate our implementation with respect to scheduling overhead and jitter. We also carry out a case study that runs real flight control software over Hardware-In-the-Loop Simulation (HILS) environment.

The rest of the paper is organized as follows: Section 2 briefs ARINC 653 as background. Section 3 details the suggested Linux kernel-level design of ARINC 653. In this section, we describe the initialization and scheduling of partitions. Performance measurement results are presented in Section 4. In this section, we also present a case study on an unmanned aerial vehicle. The related work is discussed in Section 5. Finally we conclude this paper in Section 6.

2. OVERVIEW OF ARINC 653

The ARINC standards include the vast scope of air transport avionics equipment and systems. Among them, the standard number 653, called ARINC 653, gives the standardized guideline to implement the IMA architecture, which includes the general-purpose APEX (APplication/EXecutive) interfaces between operating system of an avionics computer and application software [1]. The interfaces allow the application software to control the scheduling, communication, and status information of its internal processing elements.

ARINC 653 defines the partition that enables one or more avionics applications to execute independently from each other in

SAC'12, March 26-30, 2012, Riva del Garda, Italy.

terms of memory and processor resources. This partitioning concept is a key for IMA architecture as it provides isolation between applications. The partitioned code executes in user mode only. This model prevents a fault caused by an application propagating to others. The partitions are created at the system initialization phase and cannot be removed or added dynamically. The scheduling algorithm of partitions is predetermined, repetitive with a fixed periodicity. A partition is in one of Cold Start, Warm Start, Idle, and Normal states. Since the partitioning is the most critical feature provided by ARINC 653, we focus on this in this paper.

A partition comprises one or more processes that share the resources of the partition but are not visible outside of the partition. Each process has a priority level and can be preempted by a higher priority process. The process scheduling can be either periodic or aperiodic based on the policy configured when each process is created. A process is in one of Dormant, Ready, Waiting, and Running states.

3. KERNEL-LEVEL ARINC 653 PARTITIONING

In this section, we detail the design and implementation of kernel-level ARINC 653 partitioning for Linux.

3.1 Why Linux?

Linux is initially not targeting real-time systems. Though these days Linux is deployed by many real-time embedded systems, still Linux has a limitation on certification for avionics systems such as DO-178B [21]. Since Linux is an open project where anonymous developers can contribute without formatted development process and design documentation, it is very hard to get through such certification process. Linux, however, has provided a design reference for many POSIX-compliant real-time operating systems due to its plentiful features and stable operation. In this perspective, we believe that our study assuming Linux can cover wide range of design issues and can be applied to many existing real-time operating systems as well.

On the other hand, Linux recently shows high potential of providing software platform for avionics systems [11]. For example, several of unmanned aerial vehicles are already using Linux. In addition, there is work on trimming Linux kernel source code to minimize its size and trying to get the certification evaluation. This is possible because DO-178B does not require following the development process defined by DO-178B from very early stage of development. It rather allows developers to provide the required documents after the implementation. Thus we can also expect that a minimized version of Linux kernel can be applied for some parts of avionics systems in the future.

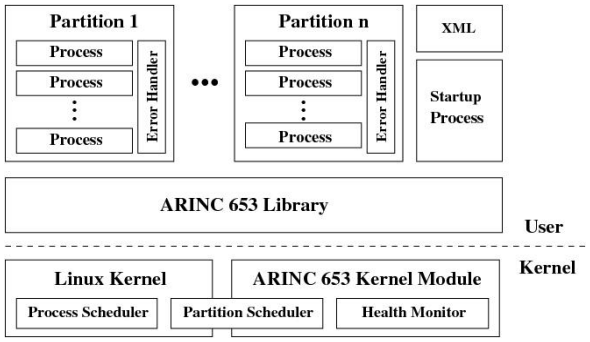

Figure 1. Overall design of ARINC 653 over Linux

3.2 Overall Design

An ARINC 653 partition is guaranteed to use processor and memory resources in an isolated manner while the processes in the same partition share the resources. Though Linux does not have the concept of partition, it is very similar with the process of Linux in the sense that a Linux process can occupy the processor resource for a given time and can be protected from others by virtual memory system. Thus, we map an ARINC 653 partition to a Linux process. In addition, the ARINC 653 process is very similar with the Linux lightweight process; thus, we map an ARINC 653 process onto a Linux lightweight process.

The overall design suggested is shown in Figure 1. The startup process performs the system initialization, which includes partition creation and scheduling table creation. Each partition comprises several processes and error handler. We have process and partition schedulers at the kernel level. The health monitor invokes the error handler of the corresponding partition when an error occurs. The ARINC 653 library provides the interfaces defined by the ARINC 653 standard for user-level processes. The ARINC 653 kernel module provides the kernel-level functions to support ARINC 653 library and partition scheduler.

3.3 System Initialization

Due to the nature of avionics systems, all the system configurations and initializations are performed statically. For efficient integration of partitions and their initialization, the ARINC 653 standard specifies XML-based configuration file format. In our system, we allow following items:

- PartitionIdentifier: a unique identifier number assigned to the partition
- PartitionName: partition name
- Criticality: degree of impact by loss or malfunction of the partition
- EntryPoint: executable file name for the partition
- PeriodSeconds: period of the partition
- PeriodDurationSeconds: execution time given to the partition every period

The attributes specified in the XML file is interpreted by the startup process that performs actual creation of partitions and processes. At the initialization phase, the information for each partition is also stored in the ARINC 653 kernel module and provided to the user land via ARINC 653 APIs.

Since avionics applications are developed by different organizations, the developers may not have overall picture of whole systems. To cope with this issue, we use a mechanism that can decide both period and execution time of each partition automatically based on those of processes and generates a scheduling table [14]. The table stores period and execution time information of partitions. In our system, the scheduling table is generated by the startup process performing schedulability test. This mechanism defines a partition model that allows variable execution time for each micro-period (i.e., minimum process period across all partitions). The intention of having variable execution time is to prevent priority inconsistency while achieving high system throughput. The priority inconsistency is the situation where a task in a low-priority partition preempts a task in a high-priority partition. We refer further details about the scheduling algorithm to [14].

The variable execution time is somewhat does not follow the ARINC 653 partition model that has a single constant execution time (i.e., PeriodDurationSeconds) for every period. To overcome

this mismatch, we set the macro-period (i.e., maximum process period across all partitions) and summation of execution time of micro-periods into partition's period and execution time, respectively, while the partition scheduler still internally refers the scheduling table. In addition, since the mechanism we use targets fixed-priority preemptive scheduling, we consider the criticality of partition as priority.

3.4 Hierarchical Scheduling

In order to efficiently provide process and partition scheduling, we design and implement a hierarchical schedulers. A partition implements an avionics application, where the processes interoperate to carry out the duties assigned to the partition. In general, all the processes should run correctly with respect to deadline and functionality for performing given tasks successfully. We use Earliest Deadline First (EDF) algorithm for process scheduling because EDF dynamically changes the priority of processes based on deadlines of processes and maximizes the system utilization. The ARINC 653 defines two different priorities, BASE_PRIORITY and CURRENT_PRIORITY, as attributes of processes. We use BASE_PRIORITY as an authority to control (e.g., suspending and resuming) other processes while CURRENT_PRIORITY is dynamically changed by the EDF scheduler.

The Linux kernels higher than 2.6.23 version use Completely Fair Scheduling (CFS) as default. Unlike O(1) scheduler implemented in early versions of Linux kernel 2.6, CFS uses a red-black tree as the process run queue. In the red-black tree, the processes are sorted by the processor time used; thus, the leftmost node represents the process that has used the least amount of time. For the sake of fairness, the scheduler picks the leftmost process for the next process to run. The EDF scheduler [10] we are using also employs the red-black tree but the processes are sorted by their deadline. Thus the leftmost process is the urgent one that the scheduler has to select as the next process to run.

We use fixed-priority scheduling for partition scheduling though this may not maximize the system utilization. The intention is to consider processes in high priority partition more importantly avoiding priority inconsistency as discussed in Section 3.3.

Since Linux does not support hierarchical scheduling, we implement a partition scheduler in the Linux kernel. Figure 2 shows the data structures to manage the partition scheduling. As we can see in the figure, each partition has a separate red-black tree, which consists of the processes belong to the corresponding partition. The partition scheduler is implemented on top of the high resolution timer, which is triggered based on the scheduling table mentioned in Section 3.2. As shown in Figure 2, CURRENT_PARTITION and NEXT_PARTITION point the partition that is running currently and the partition that is going to run in the next micro-period, respectively. Since the process scheduler only can see the red-black tree of CURRENT_PARTITION, the EDF algorithm is applied within the partition during the given time window.

A process may block or unblock regardless of which partition is the current one. Therefore, we need to carefully move a process woken up into the red-block tree of the corresponding partition even if it is not the current partition. We take care of this by storing the partition identifier in the process control block and moving unblocked processes accordingly.

Linux has several default daemon processes (e.g., swap daemon) that are initialized when the boot up phase and belong to the red-black tree of CFS scheduler. It is to be noted that though the CFS scheduler has lower priority than the EDF scheduler such daemon processes should not suffer from starvation to operate the system correctly. In order to tackle this issue, we reserve time window for the CFS scheduler so that Linux can run the default daemons even in the heavily loaded situation. This time reservation is also reflected during the schedulability test.

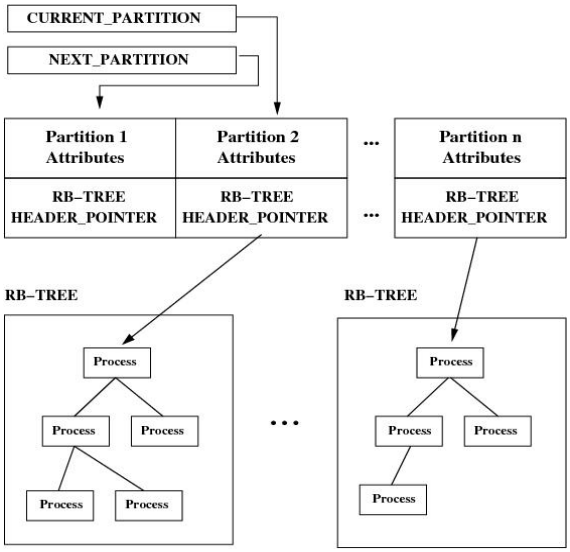

Figure 2. Data structures to manage partitions and processes

3.5 Application Programming Interfaces

Our current implementation supports ARINC 653 APIs listed in Table 1. The APIs are accessed from user-level through the ARINC 653 library, which internally calls existing system calls or the ioctl() interface provided by the ARINC 653 kernel module.

Table 1. ARINC 653 APIs implemented

APIs	Description
GET_PARTITION_STATUS	is used to obtain the status (e.g., identifier, period, duration, and state) of the current partition.
SET_PARTITION_MODE	is used to set the operating mode of the current partition.
GET_PROCESS_ID	is used to obtain a process identifier by specifying the process name.
GET_PROCESS_STATUS	returns the current status (e.g., deadline and process state) of the specified process.
CREATE_PROCESS	creates a process and returns its identifier.
SUSPEND_SELF	suspends the execution of the current process.
SUSPEND	suspends the execution of any other process.
RESUME	resumes another previously suspended process.
STOP_SELF	stops the current process.
STOP	stops the specified process.
START	initializes actual process and allows it to be executed by the scheduler.
GET_MY_ID	returns the identifier of the current process.
PERIODIC_WAIT	suspends execution of the requesting process until the next period.

The partition and process control APIs not implement yet are only four: SET_PRIORITY, DELAYED_START, LOCK_PREEMPTION, and UNLOCK_PREEMPTION. Since we use the EDF scheduler for process scheduling, the process priority is changed continuously based on the deadline. Thus the SET_PRIORITY interface is meaningless. We create actual process in the START interface by using the clone() system call. Therefore, in order to implement DELAYED_START, we need to extend the system call to specify the delay or implement a daemon process that receives the request and delays calling the clone() system call. We plan to investigate these two alternatives as future work. The LOCK_PREEMPTION and UNLOCK_PREEMPTION can be implemented easily by disabling and enabling the timer interrupt. However, we are analyzing the side effects of this idea because disabling timer interrupt can affect on not only other applications but also system performance.

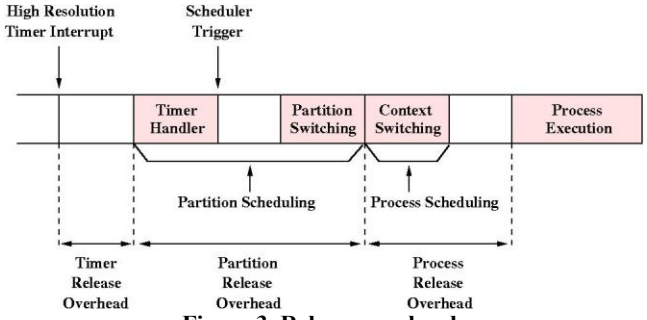

Figure 3. Release overheads

4. EVALUATION

In this section, we evaluate the performance of ARINC 653 over Linux in terms of overhead and jitter. In addition, we also carry out a case study. We have implemented our design on Ubuntu Linux (kernel version 2.6.32) and measured its performance over an industrial embedded board equipped with an Intel Mobile 1.4GHz processor.

4.1 Overhead Measurement

We measure the release overheads of timer, partition, and process in the kernel while running a workload with several random partitions and processes. The release overhead represents the delay from the time point where timer, partition, or process is expected to start until when it really starts as depicted in Figure 3. There are not many things we can optimize to reduce timer release overhead because this is mainly dependent on hardware architecture. The partition release overhead is caused by the partition scheduler. In Section 3.4, we have suggested a kernel-level partition scheduler. In order to investigate the advantage of the kernel-level design, we also implement the partition scheduler at user-level, where a partition receives a signal when the time assigned is exhausted and calls a system call to change its priority by itself and yield the processor. This user-level implementation can be similar with AMOBA [9] and SIMA [24], implementations of ARINC 653 over POSIX interfaces. Since their source codes are not available publicly, we compare between our two different implementations. The process release overhead is generated by the process scheduler. We have inserted codes to generate timestamps in the kernel and gather them after the experiments to evaluate the release overheads.

Table 2 compares the release overheads between user-level and kernel-level partition scheduling. As we can see, the partition release overhead in the kernel-level design is significantly lower than that in the user-level design. This is mainly because it takes time to signal the user-level partition scheduler. On the other hand, since the partition scheduler in the kernel-level design runs as a Softirq, it can response immediately to the scheduling request without intermediate steps such as signaling. The timer and process release overheads are almost identical in both designs because the base implementations of timer and process scheduler are more or less the same and run in kernel mode.

Moreover, we measure the overhead of ARINC 653 API calls as shown in Table 3. We do not include the APIs that force to block the current process in the experiments because the return point of those APIs is decided by other process or arguments. As we can see in the table, most of APIs show very low overhead. However, as SET_PARTITION_MODE(Cold Start) initializes the partitions and processes from the fresh state, its overhead is significantly higher than the others. The START interface also shows relatively high overhead compared with others because this interface actually creates a process. Though the overheads of these two interfaces are high, they are not called in run-time but in the initialization phase only.

Table 2. Measurement results of release overheads (μs)

Release Overheads		User-Level Partition Scheduler				Kernel-Level Partition Scheduler		
		Ave	Min	Max		Ave	Min	Max
Timer		6.5	2.9	10.6		6.48	2.8	10.4
Partition		44.83	22.0	83.0		5.2	2.1	8.8
Process		42.5	35.9	51.4		40.4	37.2	51.1

Table 3. Overhead of ARINC 653 API calls (μs)

Release Overhead	Ave	Min	Max
GET_PARTITION_STATUS	5.2	4.8	8.7
SET_PARTITION_MODE(Normal)	2.9	2.8	4.7
SET_PARTITION_MODE(Cold_Start)	282.1	188.1	458.0
GET_PROCESS_ID	0.8	0.7	1.2
GET_PROCESS_STATUS	3.0	2.7	3.8
CREATE_PROCESS	9.3	8.8	11.7
SUSPEND	3.4	3.1	3.9
RESUME	2.8	2.5	2.9
START	18.2	15.0	32.9
GET_MY_ID	1.2	0.9	3.2
CREATE_ERROR_HANDLER	1.3	1.2	1.3

4.2 Scheduling Jitter

In this section, we analyze the jitter of release overheads. We again compare the kernel-level design with user-level design. Figures 4 and 5 show the frequency and cumulative frequency distribution of release overheads for user-level and kernel-level designs, respectively. The points below the zero value of y-axis represent the overhead values measured at each iteration. In the figures, we can see that the timer and process release overheads show almost the same distribution in both designs. However, the partition release overhead shows significantly different trend. In the user-level design (Figure 4), the overhead is widely spread from 20μs to 80μs. On the other hand, the partition release overhead of kernel-level design is distributed only from 0μs to

10μs as shown in Figure 5. This presents that not only the overhead but also the jitter of our design is quite low.

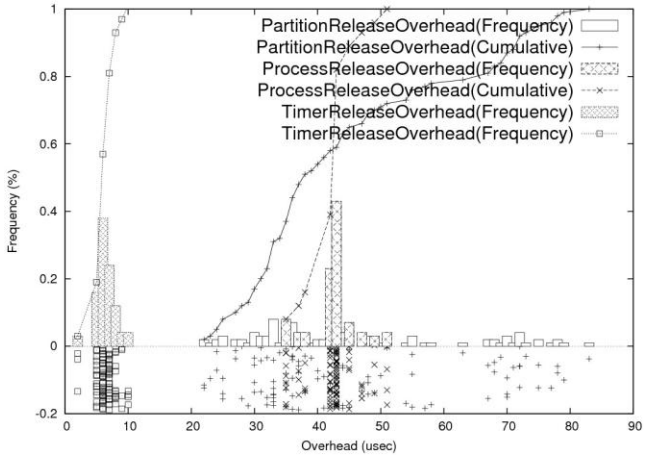

Figure 4. Release overhead distribution of user-level design

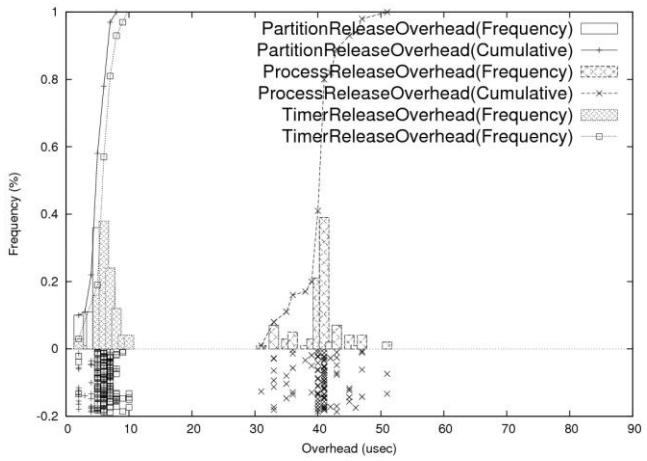

Figure 5. Release overhead distribution of kernel-level design

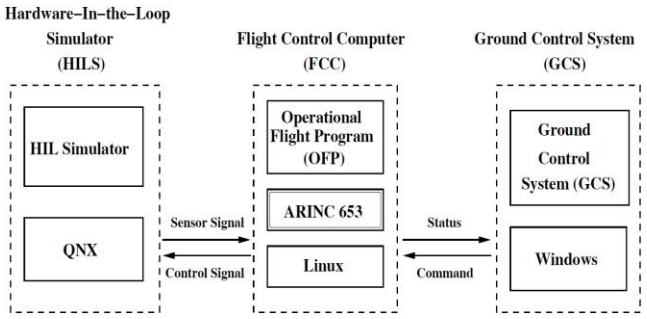

Figure 6. Hardware-in-the-loop simulation for unmanned aerial vehicle

4.3 Case Study

As a case study, we port a real Operational Flight Program (OFP) [15] for unmanned helicopter on ARINC 653 and run this in a Hardware-In-the-Loop Simulation (HILS) environment as shown in Figure 6.

The OFP is initially implemented for Voyager GSR-260 of Japan Remote Control Co., Ltd. It has a length of 1.4m and a height of 0.63m. Its main rotor diameter is 1.77m. The OFP has a control task that controls servo motors in 50Hz to move control surfaces of unmanned helicopter and four I/O tasks that read data from sensors such as Attitude and Heading Reference System (AHRS) and Global Positioning System (GPS) and Radio Frequency (RF) communication channel. The I/O task that reads attitude measurements, angular rates, and body axis also runs in 50Hz. The other I/O tasks run in 12.5Hz.

The HILS node in Figure 6 simulates the real world and airframe so that OFP thinks it controls a real aircraft. The Ground Control System (GCS) allows a person to control the aircraft remotely on the ground without professional knowledge of aeronautics.

We have observed that our ARINC 653 over Linux satisfies the real-time requirements (i.e., deadlines) of OFP and flies successfully without a crash in the HILS environment. The overheads observed during the HILS test show the same trend with what we have presented in Sections 4.1 and 4.2.

5. RELATED WORK

There have been several researches on real-time enhancements for Linux. Though Linux provides real-time process schedulers such as SCHED_RR and SCHED_FIFO, these do not support periodic tasks. Yodaiken and Barabanov [28] and Hartig et. al. [13] have introduced a real-time domain in the Linux by inserting a parasite operating system into the Linux kernel but cannot provide partitioning features. Goiffon and Gaufillet [11] have presented that Linux has high potential for avionics systems but have not discussed detail design and implementation of support for avionics applications. Calandrino et. al. [3] and real-time group scheduling of Linux can provide partitioning features but do not consider IMA interfaces. In this paper, we have suggested a design of Linux extension to support ARINC 653 partitioning.

ARINC 653 is supported by several commercial real-time operating systems, which include WindRiver VxWorks 653, LynuxWorks LynxOS-178, and GreenHills Integrity-178B. There are also implementations of ARINC 653 over POSIX called AMOBA [9] and SIMA [24]. In addition, ACM [8] defines a component-based model for ARINC 653 and is also implemented on POSIX. As we have discussed in Section 4, the user-level design has limitations on performance and real-time support. In this paper, we suggest a kernel-level design for an open-source general-purpose operating system.

Recently, researchers try to exploit the virtualization technology to implement the IMA architecture. VanderLeest [26] has implemented a prototype of ARINC 653 over Xen. Masmano et. al. [19] present a design of ARINC 653 over XtratuM. The AIR project tries to support real-time operating systems but also general-purpose operating systems over partition management kernel [22]. While the previous researches are based on the para-virtualization, Han and Jin [12] suggest ARINC 653 over full-virtualization. Though the virtualization-based ARINC 653 has very high potential, the overhead and verification issues have to be considered.

Other than ARINC 653, there have been many researches on other software platforms. Asberg et. al. [2] and Diederichs et. al. [7] have studied on the design of software platform such as AUTOSAR for automobiles. Leiner et. al. [18] compare several partitioning operating systems.

There are also significant researches on hierarchical scheduling. The previous researches have mainly focused on the schedulability test for given period and execution time of

partitions [6], [5], [16], [23], [4]. Thus the integrator has to perform a cumbersome tuning process to come up with reasonable period and execution time for all partitions. There are some researches on an enhanced way to decide the partition's execution time [25]. Such schemes can leverage our design for better duration decision algorithm.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have suggested a kernel-level design for ARINC 653 over Linux, in which we use EDF and fixed-priority scheduling for process and partition scheduling, respectively. The experimental results show that the overhead and jitter of the proposed design is significantly low compared with a user-level design. In addition, we also carry out a case study that runs real flight control software over our implementation in a HILS environment. We believe that this study can open more chances to Linux for avionics systems and provide a very valuable reference for extending an existing operating system for ARINC 653.

As future work, we plan to implement the rest of APIs and integrate the proposed design with the results from our previous work on the ARINC 653 inter-partition communication [17]. Regarding the performance measurement and analysis, we plan to compare user-level design, kernel-level design, and virtualization-based design and show their advantages and disadvantages. We also intend to run our implementation on real unmanned vehicles with multiple partitions. Some results of our ongoing work can be found at http://sslab.konkuk.ac.kr/arinc.html.

7. ACKNOWLEDGMENT

This research was supported by Basic Science Research Program (#2011-0013001) and National Space Lab Program (#2011-0020905) through the National Research Foundation of (NRF) funded by the Ministry of Education, Science and Technology.

8. REFERENCES

[1] Aeronautical Radio Inc. Avionics application software standard interface part 1 required services. ARINC Specification 653P1-2, Dec. 2005.

[2] M. Asberg, M. Behnam, F. Nemati, and T. Nolte. Towards hierarchical scheduling in AUTOSAR. *In Proc. of 14th IEEE ETFA*, Sep. 2009.

[3] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: a testbed for empirically comparing real-time multiprocessor schedulers. *In Proc. of the 27th IEEE RTSS*, pages 111–123, Dec. 2006.

[4] R. I. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. *In Proc of 26th IEEE RTSS*, pages 398–408, Dec. 2005.

[5] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. *In Proc. of 18th IEEE RTSS*, pages 308–319, Dec. 1997.

[6] Z. Deng, J. W.-S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. *In Proc. of 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, Jun. 1997.

[7] C. Diederichs, U. Margull, and F. Slomka. An application-based EDF scheduler for OSEK/VDX. *In Proc. of DATE*, 2008.

[8] A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan. A real-time component framework: experience with CCM and ARINC-653. *In Proc. of IEEE ISORC*, 2010.

[9] P. Edgar, J. Rufino, T. Schoofs, and JamesWindsor. AMOBA ARINC 653 simulator for modular based space applications. *In Proc. of DASIA*, Oct. 2008.

[10] Evidence srl. SCHED_DEADLINE, http://www.evidence.eu.com.

[11] S. Goiffon and P. Gaufillet. Linux: a multi-purpose executive support for civil avionics applications? *In Proc. of IFIP International Federation for Information Processing*, 2004.

[12] S. Han and H.-W. Jin. Full virtualization based ARINC 653 partitioning. *In Proc. of the 30th IEEE/AIAA DASC*, Oct. 2011.

[13] H. Hartig, M. Hohmuth, and J. Wolter. Taming Linux. *In Proc. of the 5th Annual Australasian Conference on Parallel and Real- Time Systems*, Sep. 1998.

[14] H.-W. Jin and S. Han. Temporal partitioning for mixed-criticality systems. *In Proc. of 16th IEEE ETFA*, Sep. 2011.

[15] S.-P. Kim, J. H. Lee, B.-J. Kim, H. J. Kwon, E. T. Kim, and I.-K. Ahn. Automatic landing control law for unmanned helicopter using Lyapunov approach. *In Proc. of the 25th IEEE/AIAA DASC*, Oct. 2006.

[16] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. *In Proc. of 20th IEEE RTSS*, pages 256–267, Dec. 1999.

[17] S.-H. Lee and H.-W. Jin. Communication Support for Collaborative Embedded Controllers in Unmanned Aerial Vehicles. *In Proc. of ICIUS*, pages 16-21, Nov. 2010.

[18] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber. A comparison of partitioning operating systems for integrated systems. *In Proc. of 26th International Conference on Computer Safety, Reliability and Security*, pages 342–355, Sep. 2007.

[19] M. Masmano, I. Ripoll, A. Crespo, and J. Metge. XtratuM: a hypervisor for safety critical embedded systems. *In Proc. of Real-Time Linux Workshop*, 2009.

[20] M. D. Natale. Moving from federated to integrated architectures in automotive. *Proceedings of IEEE*, 98(4):603–620, Apr. 2010.

[21] RTCA Inc. Software Considerations in Airborne Systems and Equipment Certification. DO-178B, Dec. 1992.

[22] J. Rufino, J. Craveiro, T. Schoofs, C. Tatibana, J. Windsor. AIR technology: a step towards ARINC 653 in space. *In Proc. of DASIA*, May 2009.

[23] S. Saewong, R. Raj, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. *In Proc. of 14th ECRTS*, pages 173–183, Jun. 2002.

[24] T. Schoofs, S. Santos, C. Tatibana, and J. Anjos. An integrated modular avionics development environment. *In Proc. of the 28th IEEE/AIAA DASC*, Oct. 2009.

[25] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. *In Proc. of 24th IEEE RTSS*, pages 2–13, Dec. 2003.

[26] S. H. VanderLeest. ARINC 653 hypervisor. *In Proc. Of IEEE/AIAA DASC*, Oct. 2010.

[27] C. Watkins and R. Walter. Transitioning from Federated Avionics Architectures to Integrated Modular Avionics. *In Proc. of 26th DASC*, Oct. 2007.

[28] V. Yodaiken and M. Barabanov. A real-time Linux. *In Proc. of USELINUX*, Jan. 1997.