FULL VIRTUALIZATION BASED ARINC 653 PARTITIONING

Sanghyun Han       Hyun-Wook Jin
Department of Computer Science and Engineering, Konkuk University, Seoul, Korea

Abstract

As the number of electronic components of avionics systems is significantly increasing, it is desirable to run several avionics software on a single computing device. In such system, providing a seamless way to integrate separate applications on a computing device is a very critical issue as the Integrated Modular Avionics (IMA) concept addresses. In this context, the ARINC 653 standard defines resource partitioning of avionics application software. The virtualization technology has very high potential of providing an optimal implementation of the partition concept. In this paper, we study supports for full virtualization based ARINC 653 partitioning. The supports include extension of XML-based configuration file format and hierarchical scheduler for temporal partitioning. We show that our implementation can support well-known VMMs, such as VirtualBox and VMware and present basic performance numbers.

Introduction

As the aerial vehicles comprise many electronic components and carry out extremely diverse and critical missions, the avionics software running on such electronic components becomes more important. The current avionics systems are based on the federated architecture, where electronic devices are connected through networks and collaborate each other. Since the number of electronic devices required in the state-of-the-art systems shoots up, the internal system architecture is now very complicate and hard to handle them with respect to integration and test. Thus, it is desirable to run several avionics applications on a single computing device so that we can simplify the network cabling and reduce the weight. The applications are, however, usually developed by different organizations independently. In addition, some of them can be reused on a different avionics system together with other set of avionics applications. Therefore, providing a seamless way to integrate separate applications on a computing device is a very critical issue as the Integrated Modular Avionics (IMA) [1] concept addresses.

In this context, the ARINC 653 standard defines resource partitioning of avionics application software [2]. The partitions can provide temporal and spatial isolation between applications on a single physical device by guaranteeing exclusive resource allocation. The virtualization technology has very high potential of providing an optimal implementation of the partition concept. The virtualization technology provides multiple virtual machines, which can run operating system and application processes over emulated hardware such as processor, memory, and I/O controller, on a single device. Thus each virtual machine can be considered as a partition. We can classify the virtualization technology into two: full virtualization and paravirtualization. The full virtualization allows the legacy software either operating systems or applications to run in virtual machine (i.e., guest domain) without any modifications. On the other hand, the paravirtualization requires modifications of guest operating systems in order to minimize the virtualization overhead. Though the paravirtualization presents better performance than full virtualization, we believe that the full virtualization is more acceptable for avionics systems because it does not require modifications of existing software that has been verified rigorously in the fields.

However, there are critical design issues when we exploit the available full virtualization technology to implement the ARINC 653 partitions. For instance, since the guest domains generally run as user-level processes in the full virtualization environment, the partition scheduling should be done at the process scheduler level of host operating system rather than Virtual Machine Monitor (VMM). On other hand, we have more chances to support various VMMs because many of existing VMMs for full virtualization are implemented as separate software packages with less dependency on a specific operating system.
To cope with these issues, in this paper, we study supports for full virtualization based ARINC 653 partitioning. The supports include extension of configuration file format and hierarchical scheduler for partition and process scheduling. It is to be noted that we carefully design our support package so that it can work with various existing VMMs. We show that our implementation can support well-known VMMs, such as VirtualBox and VMware. In addition we present basic performance numbers of our preliminary implementation.

Though there are several implementations of ARINC 653, their internal details are not described in the literature because most of them are commercial versions. Recently, several companies try to utilize the virtualization technology for ARINC 653 partitioning. There are few researches on virtualization based partitioning using open source software but they use paravirtualization [3][4]. Our discussions about the design issues of ARINC 653 partitioning and suggested full virtualization based partitioning will provide valuable references to the developers and researchers who are working on the open architectures of avionics software.

**Background**

In this section, we give an overview of the ARINC 653 standard and virtualization technology.

**ARINC 653**

The IMA architecture provides an abstraction layer to provide a common execution environment over various hardware platforms and allows the applications to run safely without affecting from other applications in terms of fault and resource utilization. Consequently, it is expected that modularity, portability, and reusability of avionics software can be achieved with the IMA architecture.

The ARINC standards include the vast scope of the 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 the operating system of an avionics computer and the application software [2]. 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 terms of memory and processor resources. This partitioning concept is a key for IMA architecture as it provides isolation between applications. The partition 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. 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. Since the partitioning is the most critical feature provided by ARINC 653, we focus on this in this paper.

ARINC 653 also defines communication interfaces for inter-partition and intra-partition communication. The inter-partition communication interfaces allow communication between two or more partitions executing either on the same node or on different nodes. To specify a connection between two or more partitions, ARINC 653 uses a 2-tuple composed by channel and port. A channel defines a logical link between source and destination partitions. A channel consists of one or more ports that provide the required resources for either sending or receiving messages. It is to be noted that the processes in the same partition can share the same port because the source and destination of messages are not processes but partitions. Both channel and port are defined statically by the configuration file. ARINC 653 defines two different communication modes: queuing mode and sampling mode. In the queuing mode, the messages are stored in the internal buffer interchangeably and passed to the application software in FIFO manner when it becomes ready to receive. In the sampling mode, on the other hand, only the most recent message is saved overwriting the previous one.

ARINC 653 also provides four mechanisms for intra-partition communication: buffers, blackboards, semaphores, and events. Buffers store messages in
the message queue and deliver to the receiver in FIFO order. On the other hand, a blackboard does not queue messages. Any message written to a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. The semaphores are used to synchronize between processes running in the same partition. An event is a communication mechanism that notifies an occurrence of a condition to processes which are waiting for it.

Virtualization

The virtualization technology provides multiple virtual machines where application processes and guest operating systems can run on emulated hardware. Since each virtual machine can be considered as a partition in ARINC 653, the virtualization is very well fit to implement ARINC 653 partitioning [3][4]. The software layer that provides the virtual machines is called Virtual Machine Monitor (VMM) or hypervisor. A VMM can run on bare hardware (Type-1) or on top of an operating system (Type-2). A VMM supplies virtual hardware platforms for guest operating systems and monitors the execution of the guest operating systems.

We can classify the virtualization technology into two: full virtualization and paravirtualization. The full virtualization allows the legacy software either operating systems or applications to run in virtual machine without any modifications. To do this, VMMs of full virtualization usually performs binary translation and emulate every detail of physical hardware platforms. VMware and VirtualBox are examples of VMMs for full virtualization. On the other hand, the paravirtualization requires modifications of guest operating systems in order to minimize the virtualization overhead. VMMs of paravirtualization provides guest operating systems with programming interfaces, which is similar with the interfaces provided by hardware platforms but much simpler and lighter. Thus the the paravirtualization presents better performance than full virtualization. Xen [5] and XtratuM [4] are examples of VMMs for paravirtualization.

Design of Full Virtualization based ARINC 653 Partitioning

We believe that the full virtualization is more acceptable than paravirtualization for avionics systems because it does not require modifications of existing software that has been verified rigorously in the fields. Accordingly, we target full virtualization environment for our research. Though we especially assume Type-2 full virtualization in this paper, Type-1 full virtualization would be the ultimate target environment because this can provide lower overhead and better system reliability.

The host and guest operating systems we are considering in this paper are mainly Linux. Recently, there are several researches on design and implementation of ARINC 653 over real-time operating systems but Linux has not been considered much for a base operating system of ARINC 653. Though Linux is not initially developed for real-time systems and does not follow a standardized development process such DO-178B, it shows high potential of providing software platform for avionics systems. For example, several of unmanned aerial vehicles are already using Linux. On other hand, since Linux suggests a complicate but solid internal architecture, we believe that a design that works for Linux can be easily applicable to other operating systems.

Overall design

The suggested design for full virtualization based ARINC 653 partitioning consists of four components: i) ARINC 653 library, ii) Startup process, iii) Process scheduler, and iv) Partition scheduler. The overall architecture of the suggested design is shown in Figure 1.

The ARINC 653 library provides the programming interfaces to generate and manipulate the ARINC 653 processes and partitions. Since the processes in a virtual machine (i.e., partition) are only recognized by the guest operating system in the full virtualized environment, programming interfaces for processes are implemented on top of guest operating system. On the other hand, programming interfaces for partitions mostly used by the startup process are implemented on top of host operating system because the virtual machines are managed by the host operating system.

The startup process initializes the partitions and their processes according to the XML-based configuration file. We detail the extension of the XML schema specified by the ARINC 653 standard
and steps to initialize the system in “Initialization” section.

The process scheduler is implemented inside of the guest operating system. The real-time scheduling algorithm we use is Earliest Deadline First (EDF) [6]. In general, all processes for an application should run correctly with respect to deadline and functionality for performing the critical tasks successfully. For example, autopilot software may consist of several processes that read data from sensors, such as AHRS and GPS, and a process that controls the control surfaces. In this example, if one of these processes misses its deadline, the autopilot cannot control the flight properly. Thus, it is highly desirable to utilize available system resources as much as possible to run all the processes satisfying the real-time requirements. We use Earliest Deadline First (EDF) algorithm for process scheduling because EDF changes priority of processes based on deadlines of processes and maximizes the system utilization. The ARINC 653 defines 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. The CURRENT PRIORITY attribute is considered as priority used by the process scheduler, which is dynamically changed by the EDF scheduler in our design.

The partition scheduler is implemented inside of the host operating system. We utilize a static priority real-time scheduling algorithm suggested by Jin and Han [7]. We detail the partition scheduling in “Partition scheduling” section.

**Initialization**

Due to the nature of avionics systems, all the system configurations and initializations are performed statically. For efficient integration of partitions and 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

In addition to the above attributes, we add one more item in the XML schema to specify VMM that runs the corresponding virtual machine (i.e., partition). This extension is shown in Table 1. It is to be noted that this can allow different VMMs to run simultaneously. A requirement is to specify the name of guest image as partition name so that the startup process can run a VMM passing the name of that guest image. An example of XML-based configuration file is shown in Table 2, where there are two partitions virtualized by VMware and VirtualBox, respectively.

The attributes specified in the XML file is interpreted by the startup process that performs actual creation of partitions and processes. During the initialization phase, the information for each partition is stored in the ARINC 653 kernel module and provided to the virtual machines. The startup process invokes VMMs with the guest image. Once a virtual
machine initialized successfully, a daemon process that runs on each partition gathers the information about ARINC 653 processes for that partition and initializes them.

Table 1. XML Schema for ARINC 653 Configuration File

```xml
<xs:element name="Partition" maxOccurs="32">
    <xs:complexType>
        <xs:attribute name="PartitionIdentifier" type="NameType" use="required"/>
        <xs:attribute name="PartitionName" type="xs:string" use="required"/>
        <xs:attribute name="Criticality" type="xs:string" use="required"/>
        <xs:attribute name="SystemPartition" type="xs:boolean" use="optional"/>
        <xs:attribute name="Period" type="xs:integer" use="required"/>
        <xs:attribute name="Duration" type="xs:integer" use="required"/>
        <xs:attribute name="VMM_Select" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:element name="Process" maxOccurs="128">
        <xs:complexType>
            <xs:attribute name="ID" type="NameType" use="required"/>
            <xs:attribute name="PERIOD" type="xs:integer" use="required"/>
            <xs:attribute name="TIMECAPA" type="NameType" use="required"/>
            <xs:attribute name="STACKSIZE" type="xs:integer" use="optional"/>
            <xs:attribute name="BASE_PRIORITY" type="xs:integer" use="required"/>
            <xs:attribute name="NAME" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:complexType>
</xs:element>
```

Table 2. Example of Configuration

```xml
<ARINC_653>
    <Partition VMM_Select="VMWARE" Duration="400000" Period="1000000" SystemPartition="true"
        Criticality="LEVEL_A" PartitionName="Ubuntu2" PartitionIdentifier="1">
        <Process NAME="task1" DEADLINE="HARD" BASE_PRIORITY="1" STACKSIZE="92345"
            TIMECAPA="2000" PERIOD="20000" ID="1"/>
        <Process NAME="task2" DEADLINE="HARD" BASE_PRIORITY="2" STACKSIZE="92345"
            TIMECAPA="10000" PERIOD="20000" ID="2"/>
        <Process NAME="task3" DEADLINE="HARD" BASE_PRIORITY="3" STACKSIZE="92345"
            TIMECAPA="40000" PERIOD="80000" ID="3"/>
        <Process NAME="task4" DEADLINE="HARD" BASE_PRIORITY="4" STACKSIZE="92345"
            TIMECAPA="4000" PERIOD="80000" ID="4"/>
    </Partition>
    <Partition VMM_Select="VBOX" Duration="200000" Period="1000000" SystemPartition="false"
        Criticality="LEVEL_B" PartitionName="other" PartitionIdentifier="2">
        <Process NAME="task5" DEADLINE="HARD" BASE_PRIORITY="1" STACKSIZE="92345"
            TIMECAPA="2000" PERIOD="40000" ID="5"/>
        <Process NAME="task6" DEADLINE="HARD" BASE_PRIORITY="2" STACKSIZE="92345"
            TIMECAPA="2000" PERIOD="40000" ID="6"/>
        <Process NAME="task7" DEADLINE="HARD" BASE_PRIORITY="3" STACKSIZE="92345"
            TIMECAPA="4000" PERIOD="80000" ID="7"/>
    </Partition>
</ARINC_653>
```
Table 3. Algorithm for Scheduling Table Creation

**DecidePartitionDuration**

**Input:**

$N$ is the number of partitions. 

$M = P_{all_{max}}^{\text{all}}/P_{all_{min}}^{\text{all}}$, where $P_{all_{max}}^{\text{all}}$ and $P_{all_{min}}^{\text{all}}$ are the maximum and minimum process periods across all partitions, respectively.

Period and execution time of processes in all partitions.

**Output:** 

$Sched[N][M]$ is the scheduling table.

**Begin procedure**

1. For each partition $i$, ($i = 0$ to $N - 1$)
2. Set $T_{\text{exec}}$ to 0;
3. For each column $j$ of $Sched[i]$, ($j = 0$ to $M - 1$)
4. For each process in the partition $i$
5. If the execution time of the process is not considered in its period (i.e., this is the first micro-period in the process period)
6. $T_{\text{exec}} = T_{\text{exec}} + \text{execution time}$;
7. End if
9. $Sched[i][j] = \min(P_{\text{all}_{\text{min}}}^{\text{all}} - \sum_{k=0}^{i-1} Sched[k][j], T_{\text{exec}})$;
10. $T_{\text{exec}} = T_{\text{exec}} - Sched[i][j]$;
11. End for
12. End for

**End procedure**

**Figure 2. Partition Scheduling Internals**
Partition scheduling

Though we can specify the period and duration of partitions in the XML-based configuration file, we also provide a way to decide the period and duration automatically by the startup process. Since avionics applications are developed by different organizations, the developers may not have overall picture of whole systems. Thus, in many cases, it is very difficult to decide the period and execution time of each partition that can guarantee the deadline of processes of the corresponding partition. Therefore, in real world, a mechanism that can decide both period and execution time of each partition automatically will be very useful.

As we have mentioned in the “Overall design” section, we implement a periodic fixed-priority partition scheduling. Our partition scheduling scheme uses a periodic, fixed-priority partition model with variable execution time [7]. Traditional partition models with fixed execution time can experience either criticality inconsistency or low system utilization. The criticality inconsistency is the situation where a task in a low-priority partition preempts a task in a high-priority partition. The main idea of our partition model is to assign the duration to the higher priority partition first and then lower priority partition if there are available time windows after some periods. We define the micro-period as the minimum process period across all partitions and decide the duration of partitions for every micro-period. The startup process runs the algorithm shown in Table 3 when the system integrator wants to decide period and duration automatically. This algorithm generates the scheduling table, which represents the duration of each partition for every micro-period.

In the Type-2 full virtualization environment, a virtual machine usually consists of several processes of host operating systems. Among them, a process runs guest operating system and its user-level processes; thus, these guest user-level processes are not recognized by the host operating system. Other processes of a virtual machine take care of hardware emulation and I/O support. In our design, we bind the processes for a virtual machine and manage them as a bundle (ARINC RB-TREE n in Figure 2). If a partition is selected as the current one, we move the processes of the partition into the main scheduling tree (CFS RB-TREE in Figure 2) that really matters to the scheduler of host operating system. The scheduling of ARINC 653 processes (EDF RB-TREE in Figure 2) inside of a partition is done by the guest operating system that is actually executed by a host user-level process as shown in Figure 2.

Performance Measurement

In this section, we present performance numbers of our preliminary implementation. The measurement is carried out over an Intel processor based system that has the processor-level virtualization support.

Initialization overhead

As we have mentioned in the previous section, the startup process parses XML-based configuration file, creates scheduling table, and initiates partitions by booting up virtual machines. We measure the overheads of such initialization steps varying the number of partitions, which are shown in Figures 3, 4, and 5, respectively. These overheads only happen at the system initialization phase and do not during the run time.

![Figure 3. XML Parsing Overhead](image)

![Figure 4. Scheduling Table Creation Overhead](image)
XML parsing and scheduling table creation overheads are unlikely dependent on VMM. As we can see in Figures 3 and 4, these overheads increase as proportional to the number of partitions. In addition, we can observe that the scheduling table creation overhead is very small compared with XML parsing overhead.

The current version of our implementation supports VMware and VirtualBox VMMs. We measure the boot-up time of virtual machines over these VMMs as shown in Figure 5. As we can see in the figure, VirtualBox shows lower boot-up time than VMware as the number of partitions increases. It is to be noted that the boot-up overhead is the dominant overhead during the system initialization. Thus, for the faster system startup, we need to tackle the virtual machine boot-up overhead issue.

Figure 5. Virtual Machine Boot-up Overhead

Scheduling overhead

We also measure partition and process scheduling overheads on VirtualBox-based partitions. In this section, we especially compare user-level partition scheduling and kernel-level partition scheduling. In the user-level design, we do not need any modification to the host operating system while the kernel-level design does. The partitioning overheads are shown in Figure 6. Timer invocation overhead represents the overhead to deliver the timer interrupt when the time given to the current partition is expired. The timer handler overhead is the time to call actual partition switching routine. The partition switching overhead is the time taken to move the processes of the current and next partitions in to and out from the main scheduling tree. The signaling overhead applicable only to the user-level design is the overhead to deliver time expiration event to the user land. As we can see in the figure, there are significant differences in partition switching and signaling overheads between user-level and kernel-level designs. With respect to the process switching overhead, both design cases show the same value because this is done by the guest operating system.

Figure 7 shows the jitter of overall partition scheduling overhead that includes timer invocation, timer handler, partition switching, and signaling overheads. The figure shows that the kernel-level design presents not only low partition scheduling overhead but also smaller jitter. Thus, we can conclude that the kernel-level design is better to meet the real-time requirements though it requires kernel modification.

Figure 6. Partitioning Overheads Breakdown

Figure 7. Jitter of Overall Partition Scheduling Overhead
We also compare the overall partition scheduling overhead between VirtualBox and VMware. As we can see in Figure 8, VMware shows slightly less overhead than VirtualBox.

**Related Work**

There have been several researches on real-time enhancements for Linux. Yodaiken and Barabanov [1] and Hartig et. al. [9] have introduced a real-time domain in the Linux by inserting a parasite operating system into the Linux kernel. Calandrino et. al. [10] have presented comprehensive comparison between real-time algorithms and suggest that global algorithms are a viable alternative to partitioning approaches. In this paper, we have suggested a design of Linux extension to support ARINC 653 on full virtualized environment.

ARINC 653 is implemented in several commercial real-time operating systems, which include WindRiver VxWorks 653, LynuxWorks LynxOS-178, and GreenHills Integrity-178B. There is an implementation of ARINC 653 over POSIX called AMOBA [11], [12] but as we have discussed in “Scheduling overhead” section, the user-level design has limitations on performance and real-time support. In this paper, we suggest a kernel-level design. Recently, researchers try to exploit the virtualization technology to implement the IMA architecture. VanderLeest [3] has implemented a prototype of ARINC 653 over Xen. Masmano et. al. [4] present a design of ARINC 653 over XtratuM. While the previous researches are based on the paravirtualization, we suggest ARINC 653 over full virtualization.

Other than ARINC 653, there have been many researches on other platforms. Asberg et. al. [13] and Diederichs et. al. [14] have studied on the design of software platform such as AUTOSAR for automobiles. Leiner et. al. [13] compare several partitioning operating systems. There are also significant researches on hierarchical scheduling [16], [17], [18], [19], [20], [21]. The previous researches have mainly focused on the schedulability test for given period and execution time of partitions [16], [17], [18], [19], [21]. Thus the integrator has to perform a cumbersome tuning process to come up with reasonable period and execution time for all partitions.

**Conclusions and Future Work**

In this paper, we have extended XML schema defined by ARINC 653 to specify VMM for each partition. In addition, we have suggested the design of fixed-priority partition scheduler that supports Type-2 full virtualized environment. We have shown that our implementation can support well-known VMMs, such as VirtualBox and VMware. In addition we present basic performance numbers of our preliminary implementation. The measurement results reveal that we need to tackle the virtual machine boot-up overhead issue for the faster system startup. In addition, we have also presented that the kernel-level design presents not only low partition scheduling overhead but also smaller jitter.

As future work, we plan to add inter-partition and intra-partition communication features to our implementation. Furthermore, we intend to study on Type-1 VMM for ARINC 653.

**Acknowledgement**

This research was partly supported by Basic Science Research Program (#2011-0013001) and National Space Lab Program (#2011-0020905) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology and also supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (#NIPA-2011-C1090-1131-0003).
References


30th Digital Avionics Systems Conference
October 16-20, 2011