SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2210

Resource partitioning for Integrated Modular Avionics: comparative study of implementation alternatives

Sanghyun Han and Hyun-Wook Jin*,†

Department of Computer Science and Engineering, Konkuk University, 1, Hwayang-dong, Kwangjin-ku, Seoul 143-701, Korea

ABSTRACT

Most current generation avionics systems are based on a federated architecture, where an electronic device runs a single software module or application that collaborates with other devices through a network. This architecture makes the software development process very simple, but the hardware system becomes very complicated and it is difficult to resolve issues of size, weight, and power efficiently. An integrated architecture can address the size, weight, and power issues and provide better software reusability, testability, and reliability by means of partitioning. Partitioning provides a framework that can transparently integrate several real-time applications on the same computing device, allowing the isolation of the execution environment in terms of resources and faults. Several studies on partitioning software platforms have been reported; however, to the best of our knowledge, extensive comparison and analysis of design and implementation alternatives have not been conducted owing to the extreme complexity of their implementation and measurement. In this paper, we present three design alternatives for partitioning at the user, kernel, and virtual machine monitor levels, which are compared quantitatively. In particular, we target the worldwide standard software platform for avionics systems, that is, Aeronautical Radio, Incorporated Specification 653 (ARINC 653). Overall, our study provides valuable design references and demonstrates the characteristics of design alternatives. Copyright © 2013 John Wiley & Sons, Ltd.

Received 24 August 2012 Revised 10 May 2013 Accepted 15 May 2013

KEY WORDS: ARINC 653; Integrated Modular Avionics; Linux; partitioning; virtualization

1. INTRODUCTION

Most current generation avionics systems are based on a *federated* architecture, where an electronic device runs a single software module or application that collaborates with other devices through a network. This architecture makes the software development process very simple with respect to synthesis and reconfiguration; however, the hardware system becomes very complicated as the number of electronic devices in avionics systems continues to increase. Thus, the federated architecture makes it difficult to efficiently resolve issues of size, weight, and power.

To simplify the physical system architecture and address size, weight, and power issues, there is a growing demand for combining multiple avionics applications within a single computing device [1, 2]. In contrast to the traditional federated architecture, we refer to this as an *integrated* architecture. Providing an efficient way of integrating avionics applications on a computing device is highly critical for scaling to the large number of applications required by future generation avionics systems. However, the application software is usually developed by different organizations without consideration of resource sharing, which means seamless integration on a single device is not easily provided.

*Correspondence to: Hyun-Wook Jin, Department of Computer Science and Engineering, Konkuk University, 1, Hwayang-dong, Kwangjin-ku, Seoul 143-701, Korea.
†E-mail: jinh@konkuk.ac.kr

To support an integrated modular architecture, the software platform should allow the isolation of the execution environment in terms of resources and faults. The resources required by an application to meet its real-time requirements must be guaranteed once they are approved, regardless of how many applications run together. In addition, a fault induced by an application should not affect another. Thus, a software platform needs to provide a level of abstraction, where each application occupies an entire system thereby preventing any potential side effects from other applications running on the same device. The concept of partition is introduced to facilitate the isolation of real-time applications, as well as better software reusability, testability, and reliability. Partitioning provides a framework that reserves system resources, such as processor and memory, for each application, whereas the applications can communicate with each other if necessary without any requirement for knowing whether other applications are running on the same node.

Several studies on software platforms that provide partitions have been reported. For example, Asberg *et al*. [3] studied the design of Automotive Open System Architecture (AUTOSAR) for automobiles. Dubey *et al*. [4], Masmano *et al*. [5], and Han and Jin [6] presented implementations of Aeronautical Radio, Incorporated Specification 653 (ARINC 653) for avionics systems. Partitioning is an essential feature for realizing an integrated modular architecture, but the method of implementing partitions has a major effect on the overall performance. However, alternative methods for the design and the implementation of partitioning have not been studied comparatively owing to the extreme complexity involved in their implementation and measurement.

In this paper, we present several design alternatives for partitioning and compare them quantitatively. In particular, we target the worldwide standard software platform for avionics systems, that is, ARINC 653 [7]. This standard defines the features for partitioning, communication, and health monitoring, as well as their APIs. We explore the design space for partitioning at the user, kernel, and virtual machine monitor (VMM) levels. We implement these on the same operating system base to ensure a fair comparison. To the best of our knowledge, this is the first systematic and extensive comparison of design and implementation alternatives for partitioning. We run a real flight control program in a hardware-in-the-loop simulation (HILS) environment to measure the performance of different implementations. We summarize our contributions as follows:

- *Realistic designs for partitioning at different levels*: We suggest realistic design alternatives to implement partitioning, which provide very valuable reference sources for researchers who plan to develop a partitioning feature by either extending an existing operating system or implementing one from scratch.
- *Extensive comparison between implementation methodologies*: Our implementations on the same operating system provide fair comparisons. Our comprehensive quantitative measurement reveals the characteristics of each design alternative and provides a performance metric to aid the selection of the best-fit implementation methodology for a target system.
- *Insights into software platform for avionics systems*: This study can provide insights into the Integrated Modular Avionics (IMA) architecture from the viewpoint of software for next generation avionics systems. Furthermore, the discussions in this paper can be extended to apply to other systems that require resource partitioning.

The rest of the paper is organized as follows. Section 2 briefly summarizes ARINC 653 and related work. Section 3 describes three different design and implementation alternatives for partitioning. Our performance measurement results and comparisons between different implementations are presented in Section 4. Finally, we conclude this paper in Section 5.

2. BACKGROUND

In this section, we describe the ARINC 653 standard focusing on the partitioning feature for IMA. In addition, we discuss related work on implementation and theoretical studies of resource partitioning.

2.1. ARINC 653

The IMA architecture [8] is an integrated architecture for avionics, which provides an abstraction layer as a common execution environment for various hardware platforms. This allows applications to run safely without affecting other applications in terms of resource utilization and fault, which facilitates the modularity, portability, and reusability of avionics software.

Aeronautical Radio, Incorporated Specifications cover a vast range of air transport avionics equipment and systems. Among these, the standard number 653, that is, ARINC 653, provides standardized guidelines for implementing the IMA architecture, which includes general purpose Application/Executive interfaces between the operating system of an avionics computer and the application software [7].

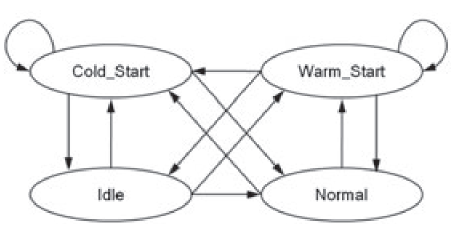
ARINC 653 defines a partition that allows one or more avionics applications to be executed independently in terms of processor and memory resources. This partitioning concept is a key aspect of the IMA architecture because it provides isolation between applications. The partition code executes in the user mode only. This prevents a fault caused by an application from propagating to others. Partitions are created during the system initialization phase, and they cannot be removed or added dynamically. Attributes of partitions, such as period and execution time, are predetermined. A partition is in one of the Cold_Start, Warm_Start, Idle, and Normal states, as shown in Figure 1(a). At the initialization phase, a partition is in the Cold_Start state. The Warm_Start state is similar, but the partition does not need to be loaded in the memory in this mode, provided that the partition image still remains in the memory after a power interruption. In the Idle state, the partition does not execute any of its processes. For example, a transition to this state may occur when an error is detected in this partition. In the Normal state, a partition runs its processes according to the scheduling policy. Therefore, partitions are usually in the Normal state if they are initialized successfully and do not experience any fatal errors.

A partition comprises one or more processes that share the resources of the partition, but those are not visible outside the partition. A process can be either periodic or aperiodic, depending on the attributes configured when each process is created. A process is in a Dormant, Ready, Waiting, or Running state, as shown in Figure 1(b). A process will be in the Dormant state before it is started and after it is terminated. A process in the Ready state is eligible for scheduling, whereas a process in the Waiting state is blocked until a particular event occurs (e.g., I/O (input/output) completion). A process that is currently being executed on a processor is in the Running state.

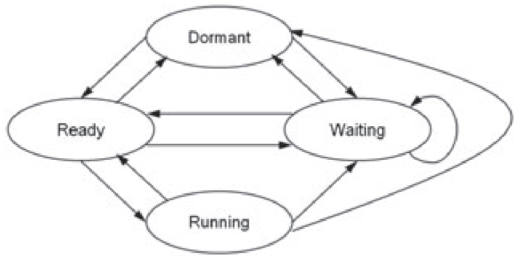
In this paper, we focus on the partitioning feature of ARINC 653. It is beyond the scope of this paper to investigate other features, such as communication and health monitoring, but our study also can provide a basis for analyzing them.

2.2. Related work

Existing ARINC 653 implementations can be classified into user-level, kernel-level, and VMM-level implementations. ARINC 653 simulator for modular based space applications (AMOBA) [9] and ARINC 653 simulator simulated IMA (SIMA) [10] are user-level implementations based on



(a) Partition



(b) Process

Figure 1. State transition diagrams of Aeronautical Radio, Incorporated Specification 653 (ARINC 653) partition and process.

Portable Operating System Interface (POSIX). In addition, CORBA Component Model with ARINC 653 [4] defines a component-based model for ARINC 653 that is also implemented on POSIX.

Several commercial ARINC 653 implementations, such as LynuxWorks LynxOS-178 [11] and GreenHills Integrity-178B [12], are implemented at the kernel level. Han and Jin [6] presented an implementation of ARINC 653 in the Linux kernel. In addition to ARINC 653, there have been studies on AUTOSAR [13], a software platform for automobiles. Initially, AUTOSAR lacked the concept of partition, but it was incorporated in Release 4.0 to provide better software integration. Asberg *et al.* [3] studied the design of kernel-level hierarchical scheduling for partitioning of AUTOSAR.

Recent researchers have tried to exploit virtualization technology for implementing ARINC 653. Virtualization technology provides multiple virtual machines on a single physical node. The software layer that provides the virtual machines is called VMM. Some VMMs (e.g., Xen [14]) can run on bare hardware (Type 1 [15]). In this case, the VMM can solely implement the partitioning feature. Other VMMs (e.g., VMware Workstation [16] and VirtualBox [17]) run on top of an operating system, known as a host-operating system (Type 2 [15]), where the VMM needs help from the host-operating system to support partitioning. We can also classify the virtualization technology into full virtualization and paravirtualization [14]. The full virtualization allows the legacy software either guest-operating systems or applications to run in a virtual machine without any modifications. On the other hand, the paravirtualization requires modifications of guest-operating systems to minimize the virtualization overhead. There are still many issues in terms of performance, but the virtualization technology has very high potential for providing an ideal implementation of the IMA concept. It also can leverage efficiency of certification for a system that consolidates several avionics software module. VanderLeest [18] implemented a prototype of ARINC 653 on Xen. Masmano *et al.* [5] presented a design of ARINC 653 on XtratuM. The AIR project aimed to support both real-time and general purpose operating systems over partition management kernel [19]. These studies were based on paravirtualization, whereas Han and Jin [20] suggested ARINC 653 partitioning with full virtualization. There also have been studies on AUTOSAR partitioning with virtualization. For example, Navet *et al.* [21] discussed issues related to virtualization-based partitioning. Although Hergenhan and Heiser [22] suggested providing partitions by using OKL4-based virtualization, the details of their designs are not described. Chung and Jin [23] also applied virtualization technology in the context of automobiles to increase safety by leveraging fault isolation, but they did not consider resource partitioning.

Although there have been studies for ARINC 653 implementations, an exhaustive comparison and analysis of design and implementation alternatives are lacking from the literature. Leiner *et al.* [24] compared several partitioning operating systems, including example implementations of ARINC 653 and AUTOSAR. However, they did not investigate the impact of different methods of implementing partitions. We also studied some of the design alternatives for partitioning and published preliminary results [6, 20]. However, performance comparisons between user-level and kernel-level designs were performed in a very limited scope without discussions about VMM-level design in [6]. In addition, our early designs did not rigorously consider minimizing partition scheduling overheads and jitters for user-level and VMM-level designs. The VMM-level partitioning also did not include ARINC 653 APIs. Overall, this paper analyzes performance comparatively in an integrated manner compared with previous publications.

There are only limited studies on spatial partitioning. Eswaran and Rajkumar [25] suggested a memory page reservation scheme. Although they did not specifically consider partitioned systems, the memory pages reserved can be isolated by means of hierarchical reservation. Yun *et al.* [26] suggested the MemGuard framework that provides an efficient memory bandwidth reservation on multicore systems.

As described in Subsection 3.1, our study is carried out on the basis of Linux. There have been several studies of real-time enhancements in Linux. Linux provides POSIX real-time process schedulers such as SCHED_RR and SCHED_FIFO, but they do not support periodic tasks by the specifications of POSIX standard. Thus, several researchers tried implementing a periodic scheduler in the Linux kernel [27, 28]. Calandrino *et al.* [29] provided an evaluation of realistic implementations of real-time scheduling in the Linux kernel for multicore processors, although hierarchical scheduling for partitioning was not considered. Yodaiken and Barabanov [30] and Hartig *et al*. [31] introduced a real-time domain in Linux by inserting a parasite operating system into the Linux kernel, but it did not provide partitioning features.

There has also been a significant amount of research on hierarchical scheduling algorithms, which can provide partitioning efficiently. However, previous researchers have focused mainly on the schedulability test for a given period and the execution time of partitions [32–35]. Thus, the integrator has to perform a cumbersome tuning process to arrive at a reasonable period and execution time for all partitions. Few studies have proposed an enhanced method for determining a partition’s execution time [36, 37]. In this paper, we utilize the algorithm suggested in [37], but the basic discussions in this paper do not depend on a specific scheduling algorithm. There have also been implementation studies of hierarchical scheduling [38–44]. We use similar designs with those for some components in our implementations. However, none of the existing studies focus on ARINC 653 and compare different implementation alternatives on the same base system.

3. DESIGN ALTERNATIVES FOR AVIONICS APPLICATION STANDARD SOFTWARE INTERFACE 653 PARTITIONING

3.1. Overview

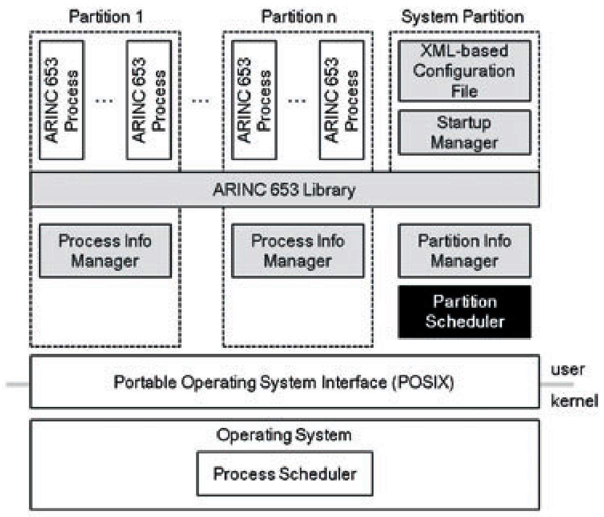
In this subsection, we briefly describe three design alternatives and some basic issues, such as the base-operating system and fundamental-partitioning mechanisms.

3.1.1 Design alternatives: In this paper, we deal with three different design and implementation alternatives for partitioning, that is, user-level, kernel-level, and VMM-level partitioning. User-level partitioning is built on top of POSIX, and thus the implementation can be portable across POSIX-compliant operating systems without (significant) modifications. In contrast, kernel-level partitioning is highly dependent on the internal implementation of target-operating system, although it may minimize overheads. VMM-level partitioning can support different implementations of ARINC 653 over different operating systems at the same time by exploiting virtualization technology.

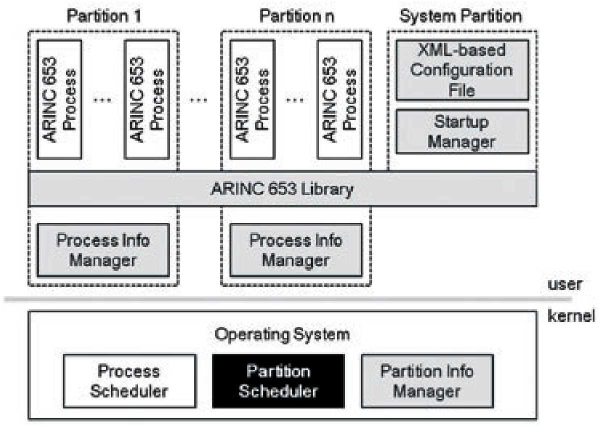
The three design alternatives are shown in Figure 2. We will provide more details in each design in the following subsections (i.e., 3.2, 3.3, and 3.4). In this subsection, we describe the basic common components, including the process and partition schedulers, the ARINC 653 library, the system partition, and the process/partition info managers. The figures show that each partition is comprised of several processes. Processes within a partition are scheduled by the process scheduler, whereas the partition scheduler manages scheduling between partitions. As shown in the figures, we classify the design alternatives based on the location of the partition scheduler. The ARINC 653 library provides programming interfaces that generate and manipulate ARINC 653 processes and partitions. The ARINC 653 standard describes that the system partition performs functions, such as managing communication and fault tolerance. The system partition is optional, and its detail design is implementation-dependant. As shown in Figure 2, our system partition includes the startup manager that initializes partitions and their processes specified in the XML-based configuration file. The process/partition info managers store and update current information of processes and partitions. Moreover, these managers implement actual services invoked via ARINC 653 APIs to manage processes and partitions.

3.1.2 Operating system: We study three design alternatives based on the Linux operating system. Linux has provided a design reference for many POSIX-compliant real-time operating systems due to its abundant features and stable operation. From these perspectives, our study covers a wide range of design issues that can be applied to many other real-time operating systems.

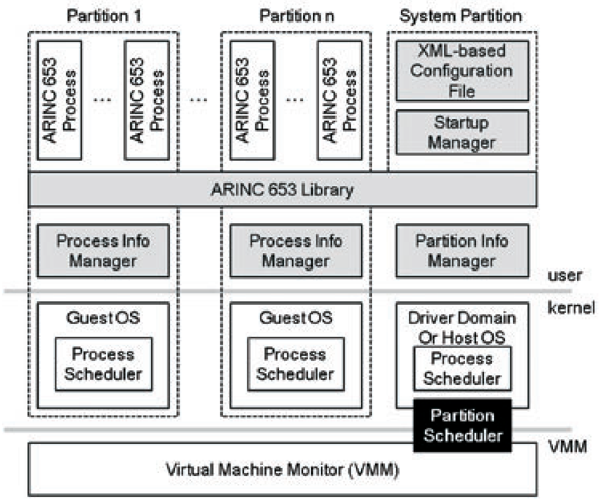
Linux has been employed in avionics systems. Unmanned aerial vehicles are already using Linux actively [45–48]. It should also be noted that the US Navy chose Linux very recently for its unmanned vertical take-off and landing aircraft control system. Goiffon and Gaufillet [49] showed that Linux has high potential for civil avionics systems. In general, communication, image processing, and runtime monitoring/testing domains are the main avionics areas where Linux is currently



(a) User-level partitioning



(b) Kernel-level partitioning



(c) VMM-level partitioning

Figure 2. Design alternatives for Avionics Application Standard Software Interface 653 partitioning.

utilized, although their internal details are not discussed in literature. Moreover, there has also been research on trimming Linux kernel source code to minimize its size, with the aim of certification evaluation. DO-178B defines software development process and required documents for software used in airborne systems [50]. Existing open software development projects, such as Linux, are hard to acquire through a certification because these usually do not follow a strict development process. However, in this paper, we do not consider certification issues intensively while focusing on implementation approaches of resource partitioning.

3.1.3 Temporal partitioning: To provide temporal partitioning, processor resources should be scheduled in a hierarchical manner. At the first level, the *partition scheduler* assigns processor resources across partitions according to their period and execution time. The partition scheduler is also known as global scheduler. In the next level, the *process scheduler* runs processes in the current (i.e., active) partition during the time window given by the partition scheduler. The process scheduler is also known as local scheduler.

The ARINC 653 standard does not define detail scheduling algorithms. We use a mixed-criticality offline scheduling algorithm [37] for partition scheduling. The basic idea of this algorithm is to generate a partition scheduling table preventing a partition of a low criticality from preempting a partition of a high criticality (i.e., criticality inversion avoidance) while facilitating high throughput. The startup manager in the system partition generates a scheduling table with this algorithm during the system initialization phase, and it performs a schedulability test for given partitions. ARINC 653 does not allow the system to add partitions dynamically during the runtime, so this offline scheduler can be used effectively. However, the study in this paper is not limited to a specific partition scheduling algorithm.The system partition is also required to be periodically scheduled as regular partitions. As we have mentioned earlier, our system partition in Figure 2 includes the startup manager. Thus, a short execution time of system partition results in a longer system initialization time, whereas a long execution time limits time windows for regular partitions after initialization. To resolve this issue, we employ multiple module schedules defined in ARINC 653 Part 2 – Extended Services [51], which enables the partition scheduler to manage multiple scheduling tables and to choose one of them according to the current mode. We create two scheduling tables for initialization mode and running mode, respectively. In the initialization mode, only the system partition runs with a long execution time.There have been many studies on improving the real-time properties of Linux. One avenue is to provide a real-time scheduler such as the earliest deadline first (EDF) scheduler in the Linux kernel. In this paper, we utilize an implementation of the EDF scheduler developed by Evidence S.r.l. [52] as the periodic process scheduler inside a partition. This EDF scheduler is implemented on top of a high-resolution timer (`hrtimer`) to ensure very accurate process scheduling.With the EDF patch, three process schedulers coexist in the Linux kernel, namely POSIX real-time scheduler, EDF scheduler, and completely fair scheduler (CFS). We assigned the highest priority to the POSIX real-time scheduler to timely schedule system processes, such as partition info manager and partition scheduler in the user-level design, which run as system daemons and do not belong to a partition. These processes need to be scheduled immediately when an API call or timer event is triggered. It is important that system processes operate in blocking mode (i.e., run only if a corresponding event occurs) to guarantee these processes not to consume processor resources uselessly. For other aperiodic processes including default system daemons of Linux, we use CFS, which is assigned the lowest priority among schedulers.In this paper, we consider polling-driven I/O devices that do not generate interrupts for I/O operations. Instead, processes take care of checking I/O completion and I/O processing. Thus, we do not consider I/O interrupt handling in temporal partitioning.

3.1.4 Spatial partitioning: Spatial partitioning protects the memory area of a partition from others with respect to both read and write operations. We provide spatial partitioning by means of memory management unit, which supplies virtual memory spaces protected between processes. As described in the following subsections, each partition is mapped to a process or a set of processes in our implementations and thus can own an isolated memory space.

In Linux, we can set memory resource limits of a process by the `setrlimit()` system call. Most VMMs also allow a virtual machine to define its physical memory size going to be used. Because the ARINC 653 configuration file can specify memory limit of a partition, we use this system call and VMM's interface when the process info manager initializes its partition.

3.1.5 Implementation and experimental setup: We implemented three design alternatives on Linux kernel version 2.6.32 and applied the EDF scheduler patch (`SCHED_DEADLINE` version 2 [52]). Table I shows added/changed/deleted source lines of code in the Linux kernel to implement three alternatives. We did not count lines for the EDF scheduler patch as we consider the patched kernel as base. We tried to minimize kernel modifications by implementing most of the features in the kernel module. The reason the kernel-level implementation has more lines added than the VMM-level implementation is because the former case handles more complicated data structures (e.g., red–black tree) and implements partition info manager in the kernel. In the case of VMM-level partitioning, we used VMware Workstation version 7.1.4 and ran Linux as a guest-operating system that supported a part of ARINC 653 APIs.

We measured the performance by using an industrial-embedded board equipped with an Intel Pentium 2.9 GHz processor. The processor included two cores, but we enabled only a single core unless we specified that two cores were enabled. This was to test the impact of multicore processor.

3.2. User-level partitioning

In this design, all the features that support partitioning are implemented at the user-level, as shown in Figure 2(a). The services provided by operating system are requested via standardized common interfaces, such as POSIX. Thus, the implementation can be portable across many POSIX-compliant operating systems, although the performance may not be optimal.

3.2.1 Processes and partitions: Linux does not have a concept of partitions, but it is very similar to that of processes on Linux in the sense that a Linux process can occupy the processor resources for a given time and it can be protected from others by the virtual memory system. Thus, we map an ARINC 653 partition to a Linux process. In addition, the ARINC 653 process is very similar to a thread in that a thread shares resources with other threads belonging to the same process. Thus, we implement an ARINC 653 process on top of a thread.

During the initialization phase, the startup manager in the system partition interprets the XML-based configuration file, generates process info managers as many as partitions, and passes them the process information. Next, the process info manager initializes its partition by launching ARINC 653 processes belonging to the partition. The partition info manager also obtains process information during the initialization phase and stores it into multiple linked lists, each of which contains information on processes in a partition.As mentioned in Subsection 3.1, the system processes have a higher priority than other processes so that the partition info manager and partition scheduler can respond to requests from process info managers and POSIX timer events as fast as possible. In addition, we can also prevent the system processes being preempted by another process.

3.2.2 Partition scheduling: The partition scheduler controls ARINC 653 partitions by means of signaling as shown in Figure 3. To perform a partition switch, the partition scheduler sends `SIGSTOP` signals to processes in the current partition (i.e., *Partition j* in Figure 3) and `SIGCONT` signals to processes in the next partition (i.e., *Partition k* in Figure 3). The partition scheduler is implemented as a user-level daemon process that wakes up periodically by using a POSIX timer. The POSIX timer

Table I. Source lines of code in the Linux kernel to implement three alternatives.

Implementation alternatives	Added (kernel)	Added (kernel module)	Changed	Deleted
User-level	—	—	—	—
Kernel-level	64	1699	4	—
VMM-level	31	1418	1	—

VMM, virtual machine monitor

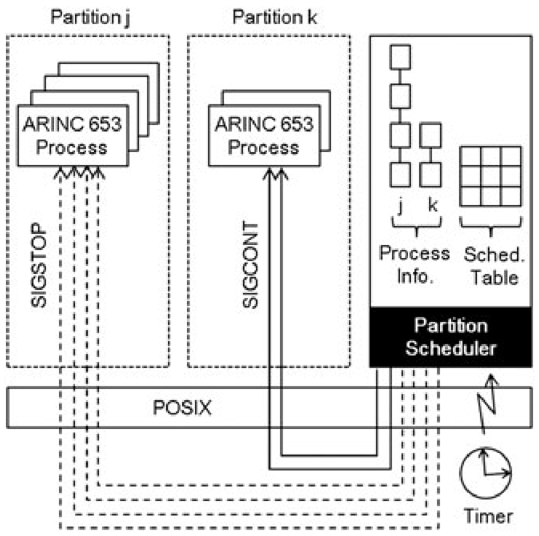


Figure 3. Partition switching in user-level design.

implemented in the current generation Linux internally uses a high-resolution timer. We will discuss about the underlying timer in Subsection 3.3. The partition scheduler sets the POSIX timer according to the scheduling table. Once a partition switching is completed, the scheduling of processes in the activated partition is performed transparently by the process scheduler.

Because the partition scheduler runs as a user-level process, its immediacy can be affected by other processes, which results in a high partition scheduling overhead and high jitter. To minimize such side effects, we assign a higher priority to the partition scheduler than other processes. We observed that the partition scheduler with a higher priority can reduce the average partition switching overhead by 44% and jitter up to 56% compared with the case having the same priority with other processes.A process can be moved to the Waiting state by a `SUSPEND` call of another process in the same partition. The partition scheduler should not send a `SIGCONT` signal to such a process when the corresponding partition becomes active by a partition switching, because this process should be blocked until another process in the same partition resumes it. To deal with this issue, the partition info manager stores the state information of all processes (i.e., *Process Info*. in Figure 3) so the partition scheduler can refer to the state information and selectively sends a signal. In the user-level design, we assume that ARINC 653 processes use `SUSPEND` and `RESUME` interfaces instead of using the `kill` system call directly with `SIGSTOP` and `SIGCONT`.

3.2.3 APIs: ARINC 653 APIs return the state information about a specific process or partition, or change a process's state with assistance from process/partition info managers. All the state information for both process and partition is stored basically by the partition info manager. At the same time, each partition maintains its static information locally, such as process IDs.

Some of API calls can provide requested information by simply referencing the local memory space (i.e., arrow number 1 in Figure 4). In contrast, other API calls that require dynamic information must ask the operating system or the partition info manager (i.e., arrow numbers 2 and 3 in Figure 4), which may cause overheads. The partition info manager not only provides dynamic state information of partitions and processes but also implements `SUSPEND` and `RESUME` APIs that change the state of a process by using POSIX signals.

The intention of providing three different paths for APIs is to reduce overheads and jitters of APIs. Table II shows the average overheads and jitters of 100 samples for each measurement of these three different paths. In this experiment, we simply measured overheads of three primitives without a specific API service. We can observe that the function call shows the least overhead and jitter and thus we implement APIs by using this as much as possible.

Moreover, to analyze the impact of priority assignment on the message queue performance, we assigned the partition info manager to CFS in the *Normal* case and to the POSIX real-time scheduler

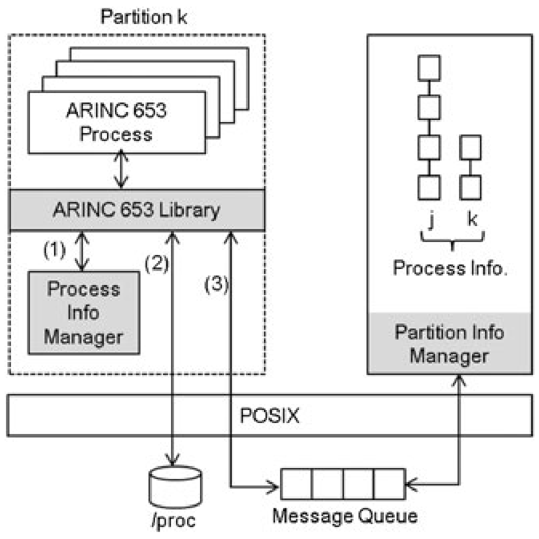


Figure 4. Internal design of APIs in user-level design.

Table II. Comparisons of average overhead and jitter between function call, system call and message queue interprocess communication.

	Function call		System call		Message queue	
Priority of partition info manager	Average overhead (μs)	Jitter (μs)	Average overhead (μs)	Jitter (μs)	Average overhead (μs)	Jitter (μs)
Normal	0.6	0.2	2.5	2.7	38.3	68.0
High	—	—	—	—	28.4	46.0

for the *High* case, respectively. As we described in Subsection 3.1, the POSIX real-time scheduler is assigned the highest priority among schedulers. Table II shows that by assigning a higher priority to the partition info manager, we could reduce both overhead and jitter of message queue interprocess communication.

3.3. Kernel-level partitioning

In this design, we implement both the partition and process schedulers in the kernel, as shown in Figure 2(b). Thus, the process and partition information is also stored in the kernel space. This design does not require signaling and message passing for partition switching and API calls, which results in better performance than user-level partitioning. However, the implementation of this design depends on the architecture and data structures of the target-operating system, which provides less portability.

3.3.1 Processes and partitions: As with user-level partitioning, a partition is mapped into a Linux process, whereas an ARINC 653 process is mapped to a Linux lightweight process, which is created by a `clone` system call. The initialization steps are the same as user-level partitioning, except the process and partition information is passed down to the partition info manager in the kernel. The partition info manager creates separate task queues for each partition and extends the data structure of the process control block (i.e., `task_struct`) to store ARINC 653 specific information.

3.3.2 Partition scheduling: To minimize the partition switching overhead, each partition has a separate task queue that is implemented utilizing a Linux red–black tree. Partition switching is then simply a matter of changing a pointer from one task queue to another as shown in Figure 5.

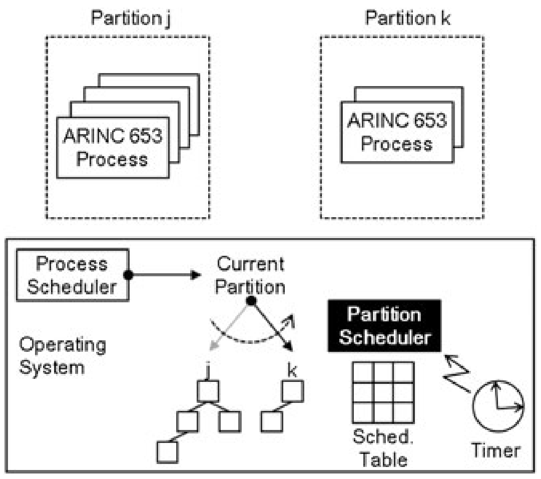


Figure 5. Partition switching in kernel-level design.

Because the process scheduler recognizes only the pointer to the task queue in the current partition, process scheduling can be performed transparently. We can also come up with a simpler switching mechanism that manages two task queues for active and suspended partitions, respectively, and migrates process control blocks between these. This can be easily implemented in a legacy operating system that does not support partitioning but induces a high overhead and jitter. Figures 6 and 7 compare average partition switching overheads and jitters for 100 measurement samples of these two different implementations. As we can see, our design (i.e., pointer switching) shows almost constant overhead and jitter, whereas in the process migration scheme, the overhead and the jitter increase as proportional to the number of processes. We can also notice that the switching overhead of our design is slightly higher for small number of processes due to manipulation of additional data structures but this is compromised as the number of processes increases.

The partition scheduler is implemented on top of the Linux high-resolution timer, which utilizes a one-shot timer and does not depend on ticks provided by the operating system. Thus, a timer event can occur on time whenever the scheduler needs to wake up. A drawback is the need to reset every time the timer expires, but its overhead is negligible, as shown in Subsection 4.2. In contrast, the traditional tick-based timer is examined periodically to determine whether the timer has expired. Therefore, the timer resolution is inaccurate if the tick occurs at relatively large intervals, whereas

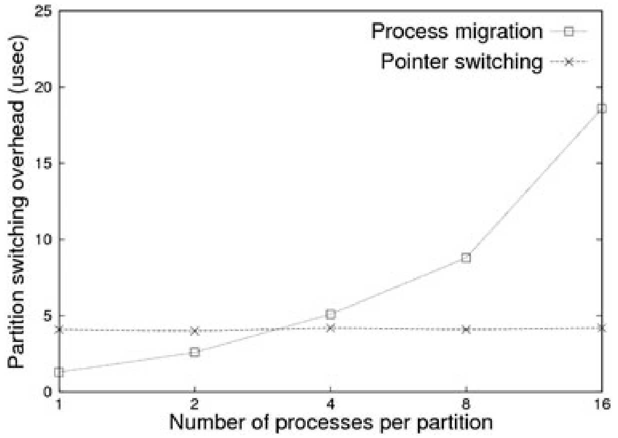


Figure 6. Average overhead of kernel-level partition scheduling with different switching mechanisms.

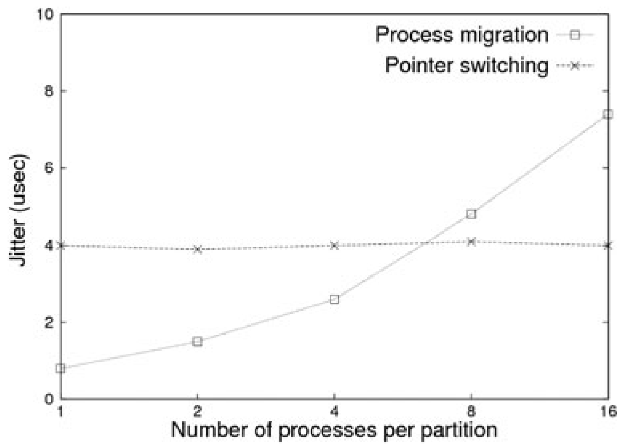


Figure 7. Jitter of kernel-level partition scheduling with different switching mechanisms.

smaller intervals have greater overheads. Figure 8 compares two timers in terms of accuracy of timer expiration. The error in the figure represents the discrepancy between the time point where the timer is expected to be expired and the time when the timer event is actually delivered to the scheduler. Thus, larger and unstable error values of tick-based timer result in a larger jitter of timer overhead. The high-resolution timer is implemented on top of a relative timer. We reset the timer to a new expiration time, which is relative to the previous expiration time, as soon as the timer expires to minimize time drifting caused by runtime system overheads. As described in Subsection 3.1, we consider only polling-based event handling for I/O devices; thus, we do not take into account the interference between interrupts of timer and I/O devices.All processes blocked are queued into the same waiting queue, where a process can be unblocked anytime by an I/O completion, regardless of whether its partition is activated or not. Therefore, we cannot simply move a process that is woken into the task queue of the current partition. We manage this by storing the partition identifier in the process control block and moving unblocked processes as required.

3.3.3 APIs: The partition info manager resides in the kernel space, so APIs internally require a system call to access dynamic state information (i.e., arrow number 2 in Figure 9). In our implementation, we insert a kernel module that exposes the `ioctl` interface to the ARINC 653 library, allowing APIs to access dynamic information. In this manner, we aim to minimize kernel modifications as much as possible. With static information, we follow the same mechanism as user-level partitioning, that is, the API calls can provide the requested information by referencing their memory space, which is represented in Figure 9 by arrow number 1.

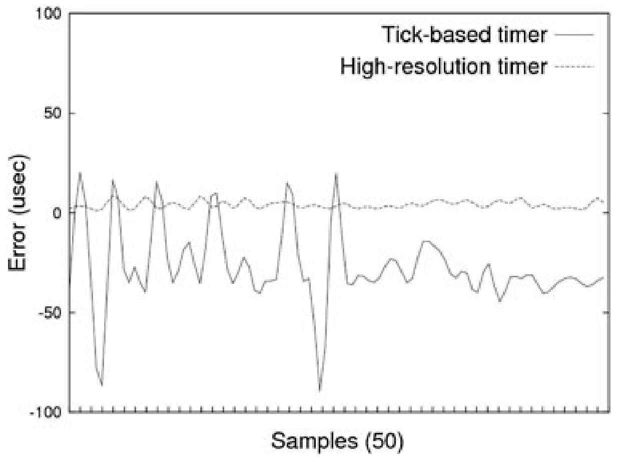


Figure 8. Comparison of errors between tick-based and high-resolution timers.

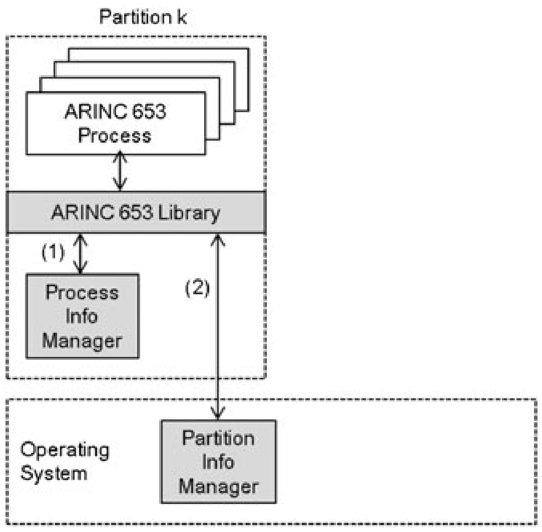


Figure 9. Internal design of APIs in kernel-level design.

3.4. *Virtual machine monitor-level partitioning*

Figure 2(c) shows the overall architecture for VMM-level partitioning. This design can run different implementations of ARINC 653 at the same time, so software developers are allowed to use different versions of ARINC 653 platforms. However, if an ARINC 653 implementation requires to run the system processes, such as process/partition info managers, on the outside of a partition, the communication between a partition and the system processes should be performed through a location-transparent protocol, such as Transmission Control Protocol (TCP)/Internet Protocol (IP), because partitions run as different machines in the VMM-level design. Similarly, if an ARINC 653 implementation requires its own system partition, it also has to use an IP-based protocol to communicate with regular partitions. We can support a number of system partitions for different ARINC 653 implementations as the standard allows multiple system partitions.

We believe that the full virtualization is more attractive than paravirtualization 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. Although 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 scalability [53]. Our current implementation supports VMware Workstation and VirtualBox.

3.4.1 Processes and partitions: Each virtual machine can be considered as a partition in ARINC 653, so the virtualization is highly appropriate for implementing ARINC 653 partitioning that provides transparency from hardware platform and flexibility of choosing an operating system. Thus, in this design, a partition is implemented on top of a virtual machine while an ARINC 653 process is mapped into a process of the guest-operating system.

The XML-based configuration decouples system configuration from actual implementation and provides an easy method for configuring the system. However, the ARINC 653 standard has yet to consider the virtualized environment in its XML schema. To launch virtual machines automatically, the configuration file may have to specify a VMM that runs the corresponding virtual machine (i.e., partition) and the name of the virtual machine image. We expand the configuration file to specify such information in the XML-based configuration file.

3.4.2 Partition scheduling: VMM does not control user-level processes running in a virtual machine; however, it is concerned with resource scheduling across virtual machines. Therefore, VMM implements the partition scheduler, whereas the guest-operating systems implement the process scheduler. Thus, each partition can use its own process scheduling algorithm independently of other partitions.

The implementation of partition scheduler may vary depending on the target VMM. As we mentioned earlier, we assume Type 2 virtualization, where a virtual machine consists of several processes on the host-operating system, which are referred to as *VM processes* in this paper. Among these, the VM main process runs the guest-operating system and its user-level processes (i.e., ARINC 653 processes). Other VM helper processes manage hardware emulation and I/O support. The partition scheduler binds the VM processes of a virtual machine and manages them as a bundle. More precisely, the partition scheduler manages a separate task queue per virtual machine as shown in Figure 10, which is very similar to the kernel-level design although the processes are not ARINC 653 processes but VM processes. During partition switching, the partition scheduler simply moves the pointer of the task queue toward that of the next partition. The POSIX real-time scheduler implemented in Linux takes care of scheduling between VM processes after partition switching is complete. It should be noted that the scheduling of ARINC 653 process is performed by the guest-operating system.Other than the pointer switching mechanism over multiple task queues, we can also consider a process migration mechanism for partition scheduling similarly to discussions in Subsection 3.3. Figures 11 and 12 compare average partition scheduling overhead

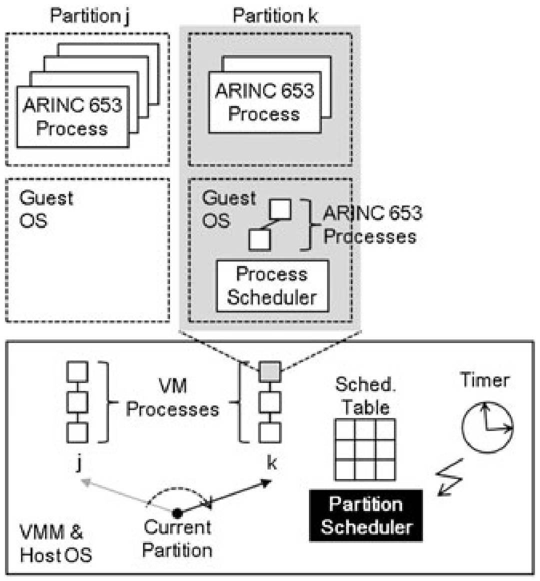


Figure 10. Partition switching in virtual machine monitor-level design.

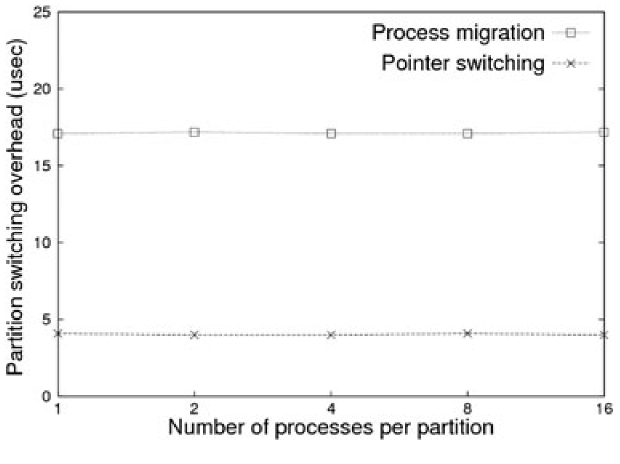


Figure 11. Average overhead of virtual machine monitor-level partition scheduling with different switching mechanisms.

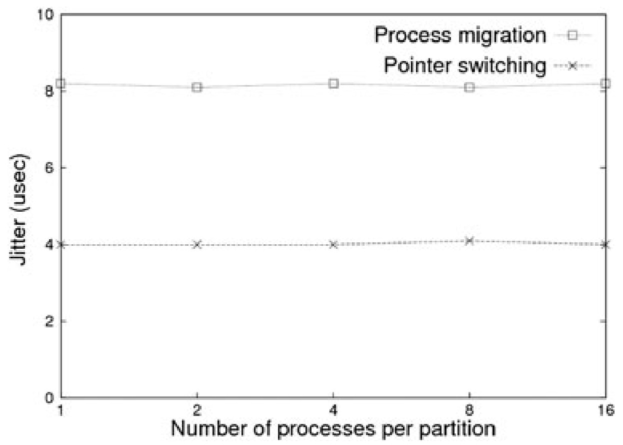


Figure 12. Jitter of kernel-level partition scheduling with different switching mechanisms.

and jitter between these two partition switching mechanisms. We analyzed 100 overhead samples for each case. In the figures, we can observe that the pointer switching mechanism shows less overheads and less jitters. We can also see that the overhead and jitter of both mechanisms are almost constant regardless of number of processes, which is a very different result from what was shown by the kernel-level design. This is because, in the VMM-level design, the processes dealt with in the partition scheduler are VM processes. Thus, the number of user processes does not affect the overhead of partition switching but the number of VM processes.

3.4.3 APIs: ARINC 653 processes are only recognized by the guest-operating system, so the APIs for ARINC 653 processes are implemented on top of the guest-operating system. Arrow numbers 1 and 2 in Figure 13 represent the same cases with the user-level design.

In contrast, partition information is mainly manipulated by the partition scheduler in the host-operating system. Therefore, we implement partition info manager on the host-operating system and provide TCP/IP connections for guest domains. APIs such as `GET_PARTITION_STATUS` need to ask the partition info manager to get partition information. However, each partition is

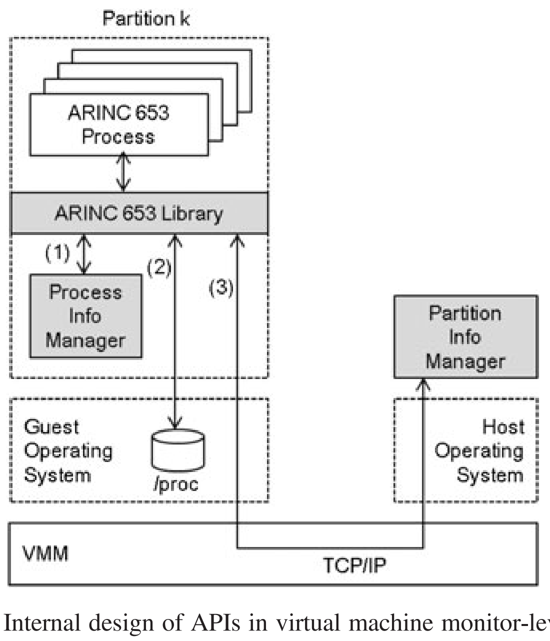


Figure 13. Internal design of APIs in virtual machine monitor-level design.

implemented as a virtual machine so the only way to communicate between partitions and the partition info manager is IP-based communication (i.e., arrow number 3 in Figure 13). We open TCP connections between ARINC 653 library and partition info manager so they can request and respond to each other. We avoid establishment overheads of TCP connections by initializing them at the system startup time. A TCP connection can be disconnected if there is no message exchanged for a period time, so we need to enable keep-alive packets to prevent this. We also analyze the impact of priority of the partition info manager on overhead and jitter of TCP-based APIs. The measurement results showed that assigning a high priority to the partition info manager can reduce both average overhead and jitter up to 19 and 27%, respectively.

4. PERFORMANCE EVALUATION

In this section, we quantitatively compare three design alternatives described in Section 3, providing performance numbers of release overhead (Subsection 4.2), execution time of application task (Subsection 4.3), response time (Subsection 4.4), and startup overhead (Subsection 4.5).

4.1. Measurement methodology

We measured the performance on the same experimental system described in Subsection 3.1; Linux kernel version 2.6.32 and VMware Workstation version 7.1.4 were installed on an Intel Pentium 2.9 GHz processor-based industrial-embedded board. To compare three design alternatives, we ran a real Operational Flight Program (OFP) [54] for an unmanned helicopter on our ARINC 653 implementations, together with a video streaming program in an HILS environment. The HILS environment simulated the real world and airframe so OFP considered that it was controlling a real aircraft. Each program ran as a partition. The OFP was initially implemented for Voyager GSR-260 provided by the Japan Remote Control Co., Ltd. It had a length of 1.4 m and a height of 0.63 m. Its main rotor diameter was 1.77 m.

The period and execution time of tasks are shown in Table III. The *GnC* task controls servomotors to move the control surfaces of the unmanned helicopter in accordance with commands from ground control system (GCS) and sends back the current state information to GCS. The *NavReader* task reads the attitude measurements, angular rates, and body from navigation sensor. Another task called *SwmReader* collects current state information of helicopter flight controls such as cyclic, rudder, throttle, and collective. The *AdtReader* task receives control commands from GCS. The video streaming program included a process (*VdoStrming*) that ran at 12.5 Hz, which sends a video stream to GCS.

Figure 14 shows delays for two partitions as an example. Jobs are released at *r* but started actually at *s* because of system overheads and finished at *f*. The system overheads can be classified into timer overhead (*TmrO*), partition switching overhead (*PtswO*), and context switching overhead (*CtswO*). Thus, the process release overhead *RelO* is represented by summation of *TmrO*, *PtswO*, and *CtswO* for the first process of each partition or summation of *TmrO* and *CtswO* for the rest. The response time *R* of a task is represented by summation of release overhead *RelO* and execution time *e*. To analyze absolute jitter of an overhead, we use $AbsJitter(O) = O_{max} - O_{min}$. Table IV summarizes the notations used throughout the paper. A job can be preempted by a higher priority task either in the same partition or another partition. In our task sets defined in Table III, the *AdtReader* and *VdoStrming* tasks can be preempted by the GnC task.

Table III. Period and execution time of tasks.

	OFP				VSP
	GnC	NavReader	SwmReader	AdtReader	VdoStrming
Period (ms)	20	20	80	80	80
Execution time (ms)	2	6	3	10	35

OFP, Operational Flight Program
VSP, video streaming program

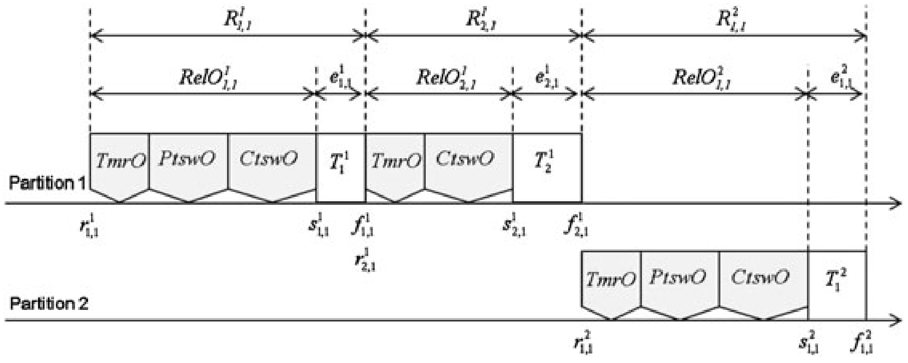


Figure 14. System overheads and task execution time.

Table IV. Notations.

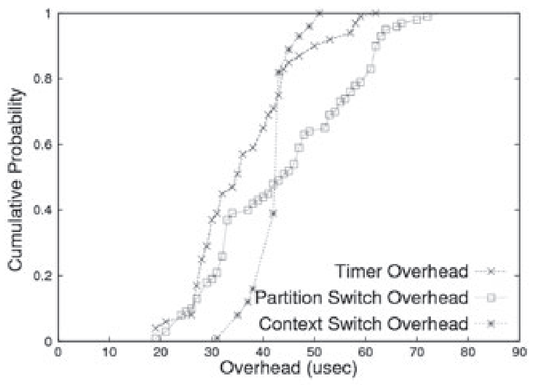
Notation	Description
$r^p_{i,j}$	Release time of *j*th job of task T_i in partition p
$s^p_{i,j}$	Start time of *j*th job of task T_i in partition p
$f^p_{i,j}$	Finish time of *j*th job of task T_i in partition p
TmrO	Timer overhead
PtswO	Partition switching overhead
CtswO	Context switching overhead
$RelO^p_{i,j}$	Release overhead of *j*th job of task T_i in partition p $RelO^p_{i,j} = s^p_{i,j} - r^p_{i,j} = \begin{cases} TmrO + PtswO + CtswO\ (i = 1) \\ TmrO + CtswO\ (i > 1) \end{cases}$
$e^p_{i,j}$	Runtime execution time of *j*th job of task T_i in partition p $e^p_{i,j} = f^p_{i,j} - s^p_{i,j}$
$R^p_{i,j}$	Response time of *j*th job of task T_i in partition p $R^p_{i,j} = RelO^p_{i,j} + e^p_{i,j}$
AbsJitter (O)	Absolute jitter of an arbitrary overhead O $AbsJitter(O) = O_{max} - O_{min}$, where O_{max} and O_{min} are maximum and minimum values, respectively, of a specific overhead O.

Almost all real-time scheduling algorithms do not consider system overheads, such as *TmrO*, *PtswO*, and *CtswO*, for the sake of simplicity but these cannot be avoided in real systems. In a real implementation, a scheduler may not start the timer until the process/partition scheduling is completed, which continuously prolongs the period of time phase and thus experiences time drifting. On the other hand, if a scheduler keeps the timer working even during the switching operation, the system overheads are piggybacked to the execution time of tasks. That is, system overheads decrease the budget given to the tasks, which results in a high chance of deadline miss. Therefore, it is very important to analyze the system overheads and minimize these. In the following subsections, we analyze system overheads and task's execution time.

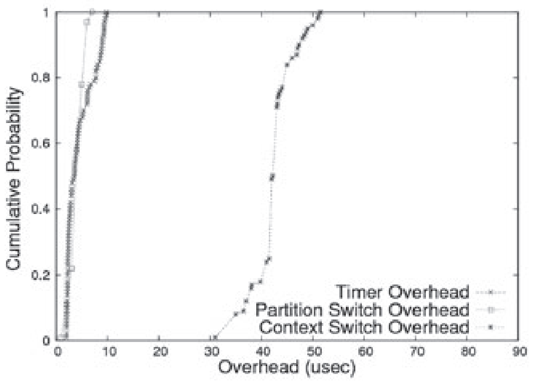
We insert codes that generate time stamps into timer handler, partition scheduler, process scheduler, and user tasks to measure their overheads (i.e., *TmrO*, *PtswO*, *CtswO*, and *e*) and jitters (i.e., *AbsJitter(TmrO)*, *AbsJitter(PtswO)*, *AbsJitter(CtswO)*, and *AbsJitter(e)*). 100 samples of time stamp pairs are stored in memory areas for each measurement and gathered after the tests are completed.

4.2. Release overhead and jitter

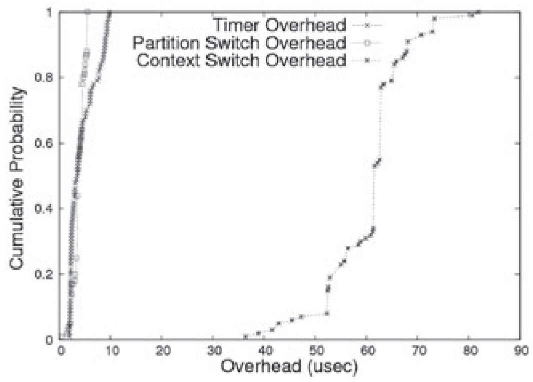
Figure 15 shows the cumulative probability of system overheads with user-level, kernel-level, and VMM-level designs. Table V shows jitter and deviation of *TmrO*, *PtswO*, and *CtswO*. As shown, the overheads of user-level partitioning in Figure 15(a) are spread over a significantly wider range than the other designs. This is mainly because the user-level design includes a user-level timer and signaling. *TmrO* shows unstable values due to the signal that delivers the timer event to user space and due to the time to detect this, even though the POSIX timer is implemented on top of



(a) User-level partitioning



(b) Kernel-level partitioning



(c) VMM-level partitioning

Figure 15. Comparison of system overheads among three design alternatives.

Table V. Jitter and deviation of system overheads.

System overheads (μs)	User-level partitioning		Kernel-level partitioning		VMM-level partitioning	
	Jitter	Deviation	Jitter	Deviation	Jitter	Deviation
TmrO	50	10	7	3	7	3
PtswO	58	15	6	1	6	1
CtswO	20	4	20	4	46	8

VMM, virtual machine monitor

high-resolution timer. *PtswO* in the user-level design vary with the number of processes because the partition scheduler needs to send signals to processes during partition switching, as described in Sub section 3.2. Figure 16 shows that a multicore processor can reduce the overheads of the user-level design. In this experiment, we dedicated a core for partition scheduler and partition info manager. Multiple cores can enhance the concurrency between operating system, partition scheduler, and application partitions, which results in less overhead of detecting the timer and signal events.

The kernel-level and VMM-level designs shown in Figures 15(b) and 15(c) had very small *TmrO* and *PtswO*. Table V also shows that jitters of these overheads are small. This was due to the low-level implementations of these two designs, where the partition scheduler captured a timer event immediately in the privileged mode and these switched partitions by simply changing a pointer for the current task queue. If partitions did not have a separate task queue, the partition switch overheads were affected by the number of processes as with the user-level implementation. However, we can observe that *CtswO* of VMM-level implementation is high. This is because the process scheduler in the guest-operating system runs in a virtualized environment.

4.3. *Execution time and jitter of tasks*

We also measured the overheads of ARINC 653 API calls, as shown in Table VI. We did not include APIs that force the blocking of current process in the experiments because the return point of these APIs are decided by other processes or API arguments. As shown in the table, the kernel-level design had very low overheads and lower jitters than the other methods.

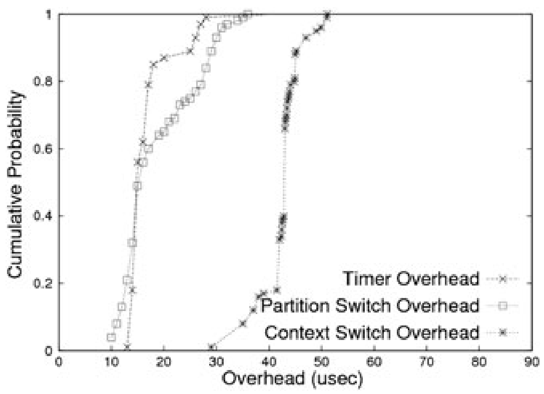


Figure 16. System overheads of user-level design with dual-core processor.

Table VI. Average overheads (in microsecond) of Avionics Application Standard Software Interface 653 API calls and their standard deviations for three design alternatives.

	User-level		Kernel-level		VMM-level	
API calls	Ave.	Dev.	Ave.	Dev.	Ave.	Dev.
`GET_MY_ID`	1.0	0.6	1.0	0.6	1.1	0.8
`GET_PROCESS_ID`	0.8	0.1	0.8	0.1	0.9	0.1
`GET_PROCESS_STATUS`	39.8	11.7	3.0	0.1	52.4	17.4
`CREATE_PROCESS`	6.2	1.2	6.2	1.2	10.4	2.68
`START`	18.2	8.6	18.2	8.6	150.5	105.8
`SUSPEND`	35.4	27.0	3.4	0.1	3.9	0.7
`RESUME`	33.8	25.5	2.8	0.0	3.9	0.6
`GET_PARTITION_STATUS`	38.2	29.9	5.2	0.5	475.4	222.1
`SET_PARTITION_MODE`	37.4	27.9	2.9	0.2	482.2	212.4

VMM, virtual machine monitor

In the case of GET_MY_ID and GET_PROCESS_ID, the API overheads were very low regardless of the design alternative used because, in all designs, a partition held static information on its address space. GET_PROCESS_STATUS acquired the running state of the process from /proc file system in the user-level and VMM-level designs, whereas the kernel-level design directly read this information via the ioctl system call.

Although the standard defines that CREATE_PROCESS creates a process, our implementation only validated its attributes, assigned a process ID, and simply returned because Linux does not have separate calls that create and start a process. We can suspend the process immediately after creation using a signal but the process has a chance of running until the signal is delivered. Thus, we delayed the actual process creation until the START interface was invoked. Thus, START showed higher overheads than CREATE_PROCESS.

Interestingly, we found that the VMM-level design had lower overheads for SUSPEND and RESUME than the user-level design. This was because these calls were served by the guest-operating system inside the partition with the VMM-level design, without help from the partition info manager. In contrast, these APIs needed message passing in the user-level design to request the system partition to send signals.

In terms of the partition management APIs (i.e., GET_PARTITION_STATUS and SET_PARTITION_MODE), the user-level design again had higher overheads than the kernel-level design because these APIs perform message passing between an application partition and the partition info manager. We observed that the POSIX message passing for 1-byte data between these had a 31 μs overhead on average and a standard deviation of 25. The VMM-level design had even higher overheads due to the TCP communication between an application partition and the partition info manager. We observed that the TCP communication between these for 1-byte data had a 398 μs overhead and a standard deviation of 212. If a VMM supports lightweight communication between virtual machines running on the same node [55, 56], this can help reduce API overheads.

If we used multiple cores, the communication performance between a partition and the partition info manager can be improved because of the enhanced parallelization. To analyze the impact of partition parallelization in user-level and VMM-level designs, we measured the API calls with two cores, where one core ran the partition info manager exclusively. As shown in Figure 17, utilizing a multicore processor meant that the API calls could achieve overhead reductions of up to 29 and 23% with the user-level and VMM-level designs, respectively.

To determine the impact of different design alternatives on the application performance, we measured the execution time of application processes during each period. Figure 18 and Table VII show the measurement results for the flight control process (*GnC*) that controlled the control surfaces of the unmanned helicopter based on the sensor values. There was a very stable execution time without virtualization (i.e., user-level and kernel-level designs) but higher jitter with virtualization. In the virtualized case, we tried to measure the impact of VT-x and multicore processors on the process execution time. The Intel VT-x technology provides several features to minimize

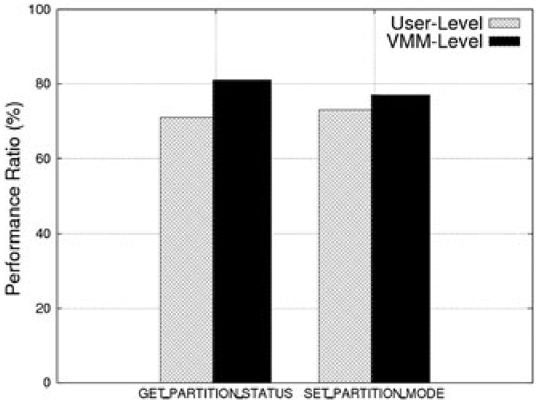


Figure 17. Reduced API overheads with dual-core processor.

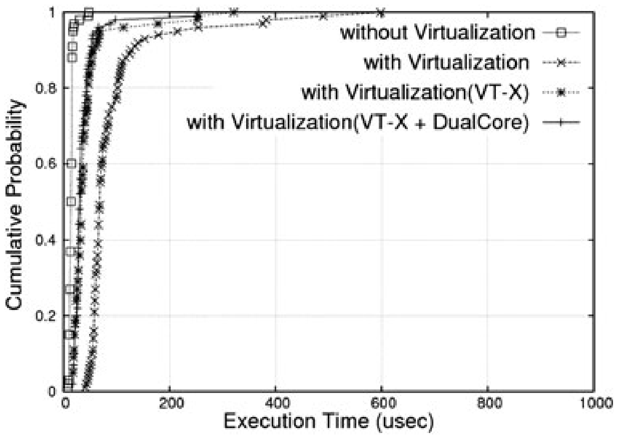


Figure 18. Execution time of flight control process (GnC) in Operational Flight Program.

Table VII. Jitter and standard deviation of execution time.

	without Virtualization	with Virtualization	with Virtualization (VT-x)	with Virtualization (VT-x + DualCore)
Jitter (µs)	33	557	303	237
Dev.	5.4	84	46	34

virtualization overheads, such as domain switching overheads and system call overheads [57]. As shown in the graph, VT-x reduced the execution time and jitter significantly, which shows that embedded processors are also required to implement virtualization support if they want to run VMM-level partitioning. We also found a slight performance improvement with a multicore processor because it can increase the parallelization between VM processes. In this experiment, we split the VM main process and VM helper processes onto different cores.

To further analyze the main overheads of the execution time in Figure 18, we measured the serial I/O overheads, as shown in Figure 19. The flight control process managed servomotors and used an RF modem through serial busses. If we compared Figures 18 and 19, we can see that the serial I/O operations were the main source of jitter, which suggests that I/O performance needs to be optimized to reduce jitter, especially in a virtualized environment.

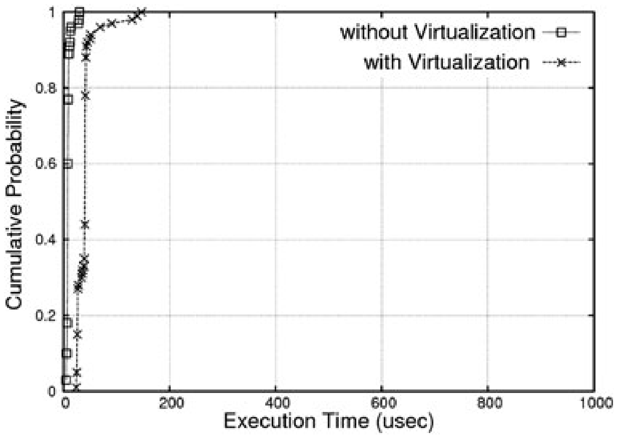


Figure 19. Comparison of serial input/output overheads.

4.4. *Analysis of response time*

In this subsection, we analyze the response time (*R*). Table VIII summarizes the measurement results presented in Subsections 4.2 and 4.3 in an integrated manner. If we take a look at release overhead for the first task ($RelO_1$), the kernel-level design shows much less average overhead and jitter than the others. In addition, we can observe that the VMM-level design achieves a significantly less release overhead than the user-level design especially in terms of jitter. This shows that the VMM-level implementation has a high potential to provide resource partitioning with even less overheads if the process scheduler of the guest-operating system has less overhead. For instance, if an operating system implements a table-based process scheduler for a fixed set of tasks, this can help reducing overheads and jitters in the guest domain.

However, task execution time (*e*) in the VMM-level implementation is very high as shown in Table VIII. Its jitter is particularly high compared with user-level and kernel-level implementations. As we have discussed in Subsection 4.3, this is mainly because the serial I/O overhead in virtualized environment increases and shows a large jitter. This reveals that a critical concern of using virtualization technology in resource partitioning is the I/O virtualization overhead.

In summary, VMM-level partitioning needs to be studied further in threefold, namely: (i) lightweight I/O virtualization; (ii) lightweight guest-operating system; and (iii) processor support for virtualization. As discussed in Subsection 4.3, I/O operations for peripherals induce a very high overhead in virtualized environment. In addition, the communication between partitions on the same node also brings high overhead, which can largely affect the performance of some APIs and interpartition communication. However, lightweight I/O virtualization is not studied thoroughly on embedded systems. Next, we have to target a lightweight guest-operating system for VMM-level partitioning. Because the guest-operating system runs under the supervision of VMM, it experiences slow execution. For instance, we have observed in Subsection 4.2 that the process scheduling overhead becomes larger in the guest domain, which reveals need of targeting lightweight operating systems that have a simple and low-overhead implementation of process scheduler. Finally, embedded processors also need to provide features to support virtualization, such as VT-x. For example, ARM recently introduces TrustZone and Virtualization Extensions to enable hardware acceleration and efficient VMM. Without such hardware support, a VMM may have to sacrifice benefits of strong isolation and software reusability provided by virtualization to elevate performance.

We assigned execution times in millisecond unit to every task as shown in Table III according to legacy requirements [54, 58]. However, we observed that the actual worst-case execution times of tasks are mostly much less than those. For example, the worst-case execution time of the GnC task is less than 50 and 230 μs for nonvirtualized and virtualized cases, respectively. Thus, we can consider adjusting execution times in Table III to actual worst-case values to allow more tasks and partitions that can pass the schedulability test. Then, it becomes more important to reduce system overheads, such as release overhead, because these are comparable with or even larger than actual worst-case execution times as shown in Table VIII and have no room to be piggybacked in execution time. That is, we still have to deal carefully with system overheads in microsecond, whereas we assign execution times in millisecond.

Table VIII. Response time (millisecond).

	Timer overhead (*TmrO*)		Partition switch overhead (*PtswO*)		Context switch overhead (*CtswO*)		Release overhead ($RelO_1$)		Execution time (*e*)		Response time (*R*)	
	Ave.	Jitter	Ave.	Jitter	Ave.	Jitter	Ave.	Jitter	Ave.	Jitter	Ave.	Jitter
User-level	28	50	38	58	40	20	106	128	15	33	121	161
Kernel-level	6	7	5	6	40	20	51	33	15	33	66	66
VMM-level	6	7	5	6	68	46	79	59	38	216	117	275

VMM, virtual machine monitor

4.5. *Partition startup overheads*

We also measured the system initialization overheads with an increasing number of partitions, as shown in Figure 20. The startup overheads are only usually of concern during the initialization phase, but these overheads can affect the performance of restarts and mode changes. As shown in the figure, where the y-axis is a log scale, the VMM-level design had a much higher startup delay than the other designs. This was due to the boot-up time of the guest-operating system. In contrast, the startup overheads of user-level and kernel-level designs were very low and approximately equal because they used almost the same model for process and partition, as described in Section 3.

To further investigate the startup delay in virtualized environment, we also measured the boot-up time of various operating systems on VirtualBox (version 4.1) and ViMo [59] as shown in Figure 21, although they do not implement ARINC 653. Our VMM-level design, shown as *Ubuntu/VirtualBox*, was included as a base performance to be compared. We ran the Gentoo and eCos operating systems on VirtualBox. We can customize a minimal Linux package by using Gentoo. Thus, we can consider Gentoo as a minimized version of Linux. eCos is a lightweight POSIX-compliant real-time operating system. In addition, we ran Ubuntu on ViMo, a full virtualization VMM for ARM-based mobile devices, which allowed us to measure the impact of simple architecture and lightweight VMM on startup overheads. We ran Ubuntu/ViMo on an Odroid embedded board, which is developed by Hardkernel Co., Ltd. in Korea and equipped with

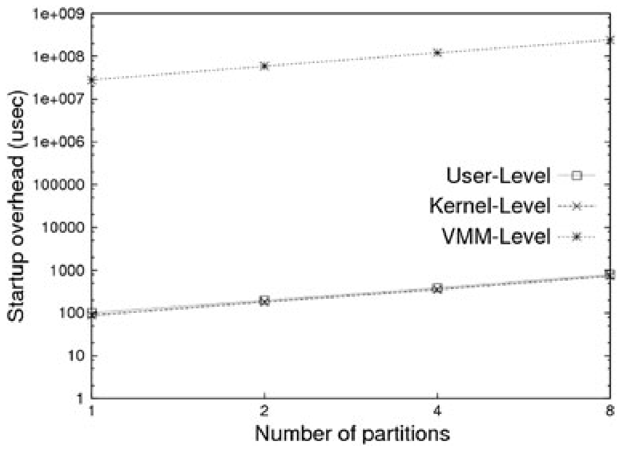


Figure 20. Comparison of startup overheads among three design alternatives.

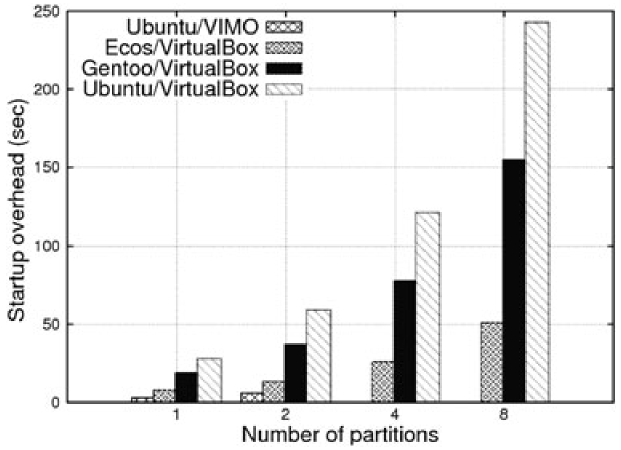


Figure 21. Startup overheads for different platforms in virtualized environment.

a Samsung S5PC110 Cortex-A8 1GHz processor. We increased the number of partitions up to two for this case because the memory size of the embedded board allowed only two guest domains. As shown in Figure 21, Gentoo and eCos reduced the startup time by 37 and 80%, respectively, compared with our VMM-level implementation. ViMo improved the startup delay even further, and it took only few seconds to start Ubuntu in a virtualized environment. This is because the ARM-based system targets a tailored architecture for a specific application. Thus, ViMo does not need to emulate many system components, such as BIOS and chipsets. This reveals that the VMM-level design also has a potential for systems that are concerned with the partition startup delay, although we still found there was high startup overhead compared with nonvirtualized cases.

5. CONCLUSIONS

In this paper, we studied three design alternatives for ARINC 653 partitioning at the user, kernel, and VMM levels with the aim of reducing the overhead and jitter. Our detailed design can provide a very valuable resource for researchers who plan to develop a partitioning feature by either extending an existing operating system or implementing one from scratch. We implemented them on the same Linux operating system to ensure a fair comparison. We especially targeted full virtualization on the basis of VMware and VirtualBox in VMM-level partitioning. We quantitatively measured the performance of three alternatives on an Intel processor-based industrial-embedded board while running a real OFP for an unmanned helicopter. We analyzed the performance of three different implementations in four categories, namely: (i) release overhead; (ii) task execution time; (iii) response time; and (iv) partition startup delay. Overall, the kernel-level implementation had very low overhead and low jitter. The user-level implementation could be portable across different operating systems, but it had high jitter during partition switching because of signaling. This design would be useful for verifying the functionality and compatibility of applications during the development phase. Virtualization technology has very high potential for implementing partitions in an ideal manner; however, our experiments showed that the overheads of process scheduler and the I/O operation need to be minimized. By this reason, VanderLeest [18] also mentioned that the VMM-level partitioning is useful for early application development phase. We believe that a VMM that sacrifices generality while targeting better performance would be acceptable for an avionics system. In addition, targeting lightweight guest-operating systems would be acceptable for VMM-level partitioning. As a result, our in-depth measurements and discussion will provide guidance when selecting the best-fit partitioning methodology for a target avionics system.

ACKNOWLEDGEMENTS

This research was supported by the Basic Science Research Program (#2012-0001671) and the National Space Lab Program (#2012-0006617) through the National Research Foundation funded by the Ministry of Education, Science and Technology.

REFERENCES

1. di Natale M, Sangiovanni-Vincentelli AL. Moving from federated to integrated architectures in automotive. *Proceedings of the IEEE* 2010; **98**(4): 603–620.
2. Watkins C, Walter R. Transitioning from federated avionics architectures to integrated modular avionics. In *Proc. 26th IEEE/AIAA Digital Avionics Systems Conf.*, Oct. 2007; 2.A.1-1–2.A.1-10.
3. Asberg M, Behnam M, Nemati F, Nolte T. Towards hierarchical scheduling in AUTOSAR. In *Proc. 14th IEEE Int. Conf. Emerging Technologies and Factory Automation*, Sep. 2009.
4. Dubey A, Karsai G, Mahadevan N. A component model for hard real-time systems: CCM with ARINC-653. *Software: Practice and Experience* 2011; **41**(12): 1517–1550.
5. Masmano M, Ripoll I, Crespo A, Metge J. XtratuM: a hypervisor for safety critical embedded systems. In *Proc. Real-Time Linux Workshop*, 2009.
6. Han S, Jin H-W. Kernel-level ARINC 653 partitioning for Linux. In *Proc. 27th ACM Int. Symp. Applied Computing*, Mar. 2012; 781–786.
7. Avionics application software standard interface, part 1, required services. ARINC Specification 653P1-2, Dec. 2005.
8. Design guidance for integrated modular avionics. ARINC Report 651, 1991.

9. Edgar P, Rufino J, Schoofs T, Windsor J. AMOBA-ARINC 653 simulator for modular based space applications. In *Proc. Eurospace Data Systems in Aerospace Conf.*, May 2008.
10. Schoofs T, Santos S, Tatibana C, Anjos J. An integrated modular avionics development environment. In *Proc. 28th IEEE/AIAA Digital Avionics Systems Conf.*, Oct. 2009; 1.A.2-1–1.A.2-9.
11. LynuxWorks RTOS for software certification: LynxOS-178. Available from: http://www.lynuxworks.com/rtos/rtos-178.php [last accessed 17 April 2013].
12. GreenHills safety critical products: INTEGRITY-178B RTOS. Available from: http://www.ghs.com/products/safety_critical/integrity-do-178b.html [last accessed 17 April 2013].
13. Automotive open system architecture. AUTOSAR Specification Release 4.0, 2012.
14. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review* 2003; **37**(5): 164–177.
15. King S, Dunlap G, Chen P. Operating system support for virtual machines. In *Proc. Int. Conf. USENIX Annual Technical Conference*, 2003.
16. VMware. Available from: http://www.vmware.com [last accessed 17 April 2013].
17. VirtualBox. Available from: http://www.virtualbox.org [last accessed 17 April 2013].
18. VanderLeest SH. ARINC 653 hypervisor. In *Proc. 29th IEEE/AIAA Digital Avionics Systems Conf.*, Oct. 2010; 5.E.2-1–5.E.2-20.
19. Rufino J, Craveiro J, Schoofs T, Tatibana C, Windsor J. AIR technology: a step towards ARINC 653 in space. In *Proc. Data Systems in Aerospace Conf.*, May 2009.
20. Han S, Jin H-W. Full virtualization based ARINC 653 partitioning. In *Proc. 30th IEEE/AIAA Digital Avionics Systems Conf.*, Oct. 2011; 7E1-1–7E1-11.
21. Navet N, Delord B, Baumeister M. Virtualization in automotive embedded systems: an outlook. Available from: http://nicolas.navet.eu/publi/RTS10_virtualization_bw.pdf, presented at the RTS Embedded Systems 2010 [last accessed 31 March 2010].
22. Hergenhan A, Heiser G. Operating systems technology for converged ECUs. In *Proc. 6th Embedded Security in Cars Conf.*, Nov. 2008.
23. Chung S-M, Jin H-W. Isolating system faults on vehicular network gateways using virtualization. In *Proc. 6th IEEE/IFIP Int. Symp. Trusted Computing and Communications*, Dec. 2010; 791–796.
24. Leiner B, Schlager M, Obermaisser R, Huber B. A comparison of partitioning operating systems for integrated systems. In *Proc. 26th Int. Conf. Computer Safety, Reliability and Security*, Sep. 2007; 342–355.
25. Eswaran A, Rajkumar RR. Energy-aware memory firewalling for QoS-sensitive applications. In *Proc. Euromicro Conf. on Real-Time Systems*, 2005; 11–20.
26. Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L. MemGuard: memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symp.*, Apr. 2013.
27. Oikawa S, Rajkumar R. Portable RK: a portable resource kernel for guaranteed and enforced timing behavior. In *Proc. Real-Time Technology and Applications Symp.*, 1999; 111–120.
28. Faggioli D, Trimarchi M, Checconi F, Scordino C. An EDF scheduling class for the Linux kernel. In *Proc. Real-Time Linux Workshop*, 2009.
29. Calandrino J, Leontyev H, Block A, Devi U, Anderson J. LITMUS^{RT}: a testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. 27th IEEE Real-Time Systems Symp.*, Dec. 2006; 111–123.
30. Yodaiken V, Barabanov M. A real-time Linux. In *Proc. Linux Applications Development and Deployment Conf.*, Jan. 1997.
31. Hartig H, Hohmuth M, Wolter J. Taming Linux. In *Proc. 5th Annu. Australasian Conf. Parallel and Real-Time Systems*, Sep. 1998.
32. Deng Z, Liu JW-S, Sun J. A scheme for scheduling hard real-time applications in open system environment. In *Proc. 9th Euromicro Workshop on Real-Time Systems*, June 1997.
33. Kuo T-W, Li C-H. A fixed-priority-driven open environment for real-time applications. In *Proc. 20th IEEE Real-Time Systems Symp.*, Dec. 1999; 256–267.
34. Saewong S, Raj R, Lehoczky JP, Klein MH. Analysis of hierarchical fixed-priority scheduling. In *Proc. 14th Euromicro Conf., Real-Time System*, June 2002.
35. Davis RI, Burns A. Hierarchical fixed priority pre-emptive scheduling. In *Proc. 26th IEEE Int. Symp. Real-Time Systems*, Dec. 2005; 388–398.
36. Shin I, Lee I. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Int. Symp. Real-Time Systems*, Dec. 2003; 2–13.
37. Jin H-W, Han S. Temporal partitioning for mixed-criticality systems. In *Proc. 16th IEEE Int. Conf. Emerging Technologies and Factory Automation (WiP Session)*, Sep. 2011.
38. Behnam M, Nolte T, Shin I, Asberg M, Bril R. Towards hierarchical scheduling on top of VxWorks. In *Proc. of Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Jul. 2008; 63–72.
39. Heuvel M, Bril R, Lukkien J, Behnam M. Extending an HSF-enabled open source real-time operating system with resource sharing. In *Proc. of Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Jul. 2010; 71–81.
40. Yang J, Kim H, Park S, Hong C, Shin I. Implementation of compositional scheduling framework on virtualization. In *Proc. of Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Nov. 2010.

41. Inam R, Maki-Turja J, Sjodin M, Ashjaei S, Afshar S. Support for hierarchical scheduling in FreeRTOS. In *Proc. of IEEE Int. Conf. on Emerging Technologies and Factory Automation*, Sep. 2011.
42. Asberg M, Forsberg N, Nolte T, Kato S. Towards real-time scheduling of virtual machines without kernel modifications. In *Proc. of IEEE Int. Conf. on Emerging Technology and Factory Automation (WiP Session)*, Sep. 2011.
43. Holenderski M, Bril RJ, Lukkien JJ. An efficient hierarchical scheduling framework for the automotive domain. In: Dr. Seyed Morteza Babamir (Ed.). *Real-Time Systems, Architecture, Scheduling, and Application*, InTech: Rijeka, Croatia April 2012. ISBN: 978-953-51-0510-7, DOI: 10.5772/38266.
44. Palopoli L, Cucinotta T, Marzario L, Lipari G. AQuoSA—adaptive quality of service architecture. *Software: Practice and Experience* 2009; **39**(1): 1–31.
45. Baker J, Cunei A, Flack C, Pizlo F, Prochazka M, Vitek J, Armbruster A, Pla E, Holmes D. A real-time Java virtual machine for avionics. In *Proc. 12th IEEE Real-Time Embedded Technology and Applications Symp.*, Apr. 2006; 384–396.
46. Haomiao H, Hoffmann GM, Waslander SL, Tomlin CJ. Aerodynamics and control of autonomous quadrotor helicopters in aggressive maneuvering. In *Proc. IEEE Int. Conf. Robotics and Automation*, May 2009; 3277–3282.
47. Netter T, Francheschini N. A robotic aircraft that follows terrain using a neuromorphic eye. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2002; **1**: 129–134.
48. Hong W-E, Lee J-S, Rai L, Kang S-J. RT-Linux based hard real-time software architecture for unmanned autonomous helicopters. In *Proc. 11th IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications*, Aug. 2005; 555–558.
49. Goiffon S, Gaufillet P. Linux: a multi-purpose executive support for civil avionics applications? In *Proc. Int. Federation for Information Processing*, 2004; 719–724.
50. Software considerations in airborne systems and equipment certification. DO-178B, RTCA Inc., Dec. 1992.
51. Avionics application software standard interface, part2, extended services. ARINC Specification 653P2-1, Dec. 2008.
52. Evidence srl, SCHED_DEADLINE version 2. Available from: http://gitorious.org/sched_deadline/pages/Home [last accessed February 2010].
53. Camagos F, Girard G, Ligneris B. Virtualization of Linux servers: a comparative study. In *Proc. Ottawa Linux Symp.*, July 2008; 63–76.
54. Kim S-P, Lee JH, Kim B-J, Kwon HJ, Kim ET, Ahn I-K. Automatic landing control law for unmanned helicopter using Lyapunov approach. In *Proc. 25th IEEE/AIAA Digital Avionics Systems Conf.*, Oct. 2006.
55. Zang H, Gu K, Li Y, Sun Y, Meng D. A highly efficient inter-domain communication channel. In Proc. *9th IEEE Int. Conf. Computer and Information Technology*, Oct. 2009; 396–374.
56. Li D, Jin H, Shao Y, Liao X, Han Z, Chen K. A high-performance inter-domain data transferring system for virtual machines. *Journal of Software* 2010; **5**(2): 206–213.
57. Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. In *Proc. 12th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Aug. 2006; 2–13.
58. Locke CD, Lucas L, Goodenough JB. Generic avionics software specification. Technical Report CMU/SEI-90-TR-8, Software Engineering Institute, Carnegie Mellon University, 1990.
59. Oh S-C, Kim KH, Koh KW, Ahn C-W. ViMo (virtualization for mobile): a virtual machine monitor supporting full virtualization for ARM mobile systems. In *Proc. Int. Conf. Cloud Computing, GRIDs, and Virtualization*, 2010.