IPSJ Transactions on Advanced Computing Systems Vol.5 No.2 99–110 (Mar. 2012)

Regular Paper

Integrated Scheduling for a Reliable Dual-OS Monitor

Daniel Sangorrin[1,a)] Shinya Honda[1] Hiroaki Takada[1]

Received: July 19, 2011, Accepted: December 26, 2011

Abstract: Virtualization solutions aimed at the consolidation of a real-time operating system (RTOS) and a general-purpose operating system (GPOS) onto the same platform are gaining momentum as high-end embedded systems increase their computation power. Among them, the most extended approach for scheduling both operating systems consists of executing the GPOS only when the RTOS becomes idle. Although this approach can guarantee the real-time performance of the RTOS tasks and interrupt handlers, the responsiveness of GPOS time-sensitive activities is negatively affected when the RTOS contains compute-bound activities executing with low priority. In this paper, we modify a reliable hardware-assisted dual-OS virtualization technique to implement an integrated scheduling architecture where the execution priority level of the GPOS and RTOS activities can be mixed with high granularity. The evaluation results show that the proposed approach is suitable for enhancing the responsiveness of the GPOS time-sensitive activities without compromising the reliability and real-time performance of the RTOS.

Keywords: virtualization, scheduling, embedded, trustzone, safeg

1. Introduction

Current high-end embedded systems are no longer confined to standalone, resource-constrained devices. On the contrary, they frequently consist of highly connected devices with increasing sophistication and complexity. In order to address such complexity within a realistic time span, it is a current trend to leverage existing open source software [1] that was originally born to service the needs of personal computer users. Despite the fact that the need for a general-purpose operating system (GPOS) and the rich functionality of its libraries is beyond question, most embedded systems are still expected to support certain requirements—such as security, reliability or timeliness—which only a small-scale real-time operating system (RTOS) can satisfy [2], [3], [4].

There are several alternatives for consolidating a RTOS and a GPOS onto the same embedded system platform. The traditional approach is to allocate separate hardware resources for each operating system. However, since that approach increases the hardware cost, new dual-OS virtualization techniques have been invented to help both operating systems to share the same hardware platform. A dual-OS virtualization technique [3], [5], [6], [7] must guarantee the security and reliability of RTOS activities (i.e., interrupt handlers and tasks) even in the presence of faulty or malicious GPOS software. For example, dual-OS virtualization techniques must make sure that memory and devices assigned to the RTOS cannot be accessed by the GPOS. In addition, the real-time performance (e.g., no deadline misses) of RTOS activities must be guaranteed. For that reason, the most extended approach for scheduling both operating systems consists of executing the GPOS only when the RTOS becomes idle [5], [6], [7], [8].

The aforementioned *idle scheduling* approach is suitable for situations where the RTOS is restricted to running time-sensitive I/O-bound activities that have a short execution time. However, modern high-end embedded systems often include additional compute-bound activities which exhibit a longer execution time and still require the security and reliability provided by the RTOS [9]. A few examples are cryptographic services—such as digital rights management (DRM) services—and secure-store facilities. Since these activities usually require access to devices that must not be accessible by the GPOS (e.g., secure memory containing cryptographic keys or passwords), they need to execute on the RTOS. The idle scheduling approach is not suitable for this situation because compute-bound activities affect negatively the responsiveness of time-sensitive GPOS activities. For example, the hardware buffer of a network card managed by the GPOS may get overwritten by the arrival of new packets if the execution of the corresponding interrupt handler is delayed for an excessive amount of time.

The goal of this work is to fix the shortcomings of the idle scheduling approach in the scope of a reliable dual-OS virtualization solution called SafeG [10]. The most important contribution of this paper is an *integrated scheduling* (IS) architecture whose main features are:

- It supports mixing the execution priority level of the GPOS and RTOS activities. This allows, for example, configuring the priority of a GPOS time-sensitive activity to be higher than the one of a RTOS compute-bound task.
- The real-time performance and reliability of the RTOS is guaranteed even if the GPOS is faulty or misbehaves. In particular, time-sensitive GPOS activities are controlled through CPU time resource reservations. This mechanism guarantees that RTOS activities executing with a lower priority will not suffer unbounded blocking nor starvation.

[1] Graduate School of Information Science, Nagoya University, Nagoya, Aichi 464–8603, Japan
[a)] dsl@ertl.jp

© 2012 Information Processing Society of Japan

- The architecture does not require modifications to the source code of the dual-OS virtualization monitor nor to the RTOS kernel. This feature makes integrated scheduling easier to maintain and verify.

We built the IS architecture on a physical platform, and evaluated it through several experiments and a realistic example application. The evaluation results show that the architecture is suitable for enhancing the responsiveness of GPOS time-sensitive activities (e.g., by scheduling RTOS compute-bound tasks at a lower priority) and the overhead introduced is small enough for practical usage.

The paper is organized as follows. Section 2 reviews the main concepts of SafeG, the dual-OS virtualization technique on top of which the presented research is built. Section 3 constitutes the core of this paper and explains the IS architecture proposed in this paper. Section 4 evaluates the IS architecture, and includes a use case example to show the effectiveness of the IS architecture in a real situation. Section 5 compares this research with previous work. Finally, the paper is concluded in Section 6.

2. Review of SafeG

SafeG [10] (*Safety Gate*) is a reliable dual-OS virtualization technique designed to support the concurrent execution of a RTOS and a GPOS on top of an ARM TrustZone-enabled [11] single processor. SafeG guarantees that RTOS data and instructions are protected from the GPOS, which is considered by default unreliable. Moreover, SafeG provides time isolation for the RTOS to guarantee that hard real-time tasks always meet their deadlines. Both memory and time isolation are backed by the ARM TrustZone hardware extensions, which allows for a low-overhead implementation and minimal modifications to the GPOS.

2.1 ARM TrustZone

ARM TrustZone is a set of hardware extensions present in high-end ARM embedded processors such as ARM 1176 [12] or the Cortex-A series. This section briefly introduces some concepts of ARM TrustZone that are necessary for understanding the rest of the paper. For more information, refer to Refs. [9], [11], [12], [13].

- *Virtual CPUs*: an ARM TrustZone-enabled single processor provides two Virtual CPUs (VCPUs), the Secure VCPU and the Non-Secure VCPU. Each VCPU is equipped with its own memory management unit and exception vector table; and supports all ARM operation modes (i.e., User, FIQ, IRQ, Supervisor, Abort, System and Undefined modes). It is important to understand that these two VCPUs do not run in parallel but in turns. In other words, only one of the VCPUs can be active at any given time.
- *The Monitor*: the Secure VCPU has an additional mode—called the monitor mode—which is used to context switch between both VCPUs. The software executing in monitor mode is commonly known as the secure Monitor. The entry to monitor mode is tightly controlled and can only be triggered by software executing the Secure Monitor Call (SMC) instruction or the occurrence of a hardware exception (i.e., IRQ, FIQ, Data abort and Prefetch abort exceptions) through an exception vector table in monitor mode. A VCPU context switch involves saving and restoring all ARM general purpose registers plus coprocessor registers that are shared by both VCPUs.
- *Address space partitioning*: when a bus master accesses memory or devices, the NS bit (Non-Secure bit) is propagated through the system bus indicating the privilege of that access (i.e., secure or non-secure). This allows the partitioning of the address space into two virtual worlds: the Secure and the Non-Secure world. This partitioning can be done statically by the hardware vendor or at run time through the TZPC [14]. The Secure VCPU is allowed to access memory and devices from both worlds. However, hardware logic makes sure that Secure world memory and devices cannot be accessed by the Non-Secure VCPU nor other Non-Secure bus masters, such as Non-Secure DMA devices.
- *Device interrupts partitioning*: ARM processors have two types of interrupt known as FIQ and IRQ. The main difference between them resides in the fact that FIQ interrupts have higher priority and more banked registers than IRQ interrupts. In a TrustZone-enabled processor, ARM recommends that Secure devices are configured to generate FIQ interrupts and Non-Secure devices are configured to generate IRQ interrupts. This configuration is carried out through the TZIC [15] which is only accessible from the Secure VCPU.

  FIQ and IRQ interrupts can be disabled within a privileged mode by setting the F and I flags of the Current Program Status Register (CPSR) respectively. To prevent Non-Secure software masking Secure device interrupts, TrustZone provides the FW (F flag Writable) bit which is only accessible by the Secure VCPU. When the FW bit is set to zero the F flag becomes non maskable for the Non-Secure world.

2.2 TrustZone Configuration in SafeG

The SafeG architecture takes advantage of ARM TrustZone hardware extensions to concurrently execute a RTOS and a GPOS on top of the same processor. The SafeG monitor—which is the main component of the SafeG architecture—is a specific implementation of the ARM TrustZone monitor focused on guaranteeing the real-time performance requirements and memory isolation of the RTOS. **Figure 1** depicts the overall organization of the SafeG architecture[*1]. The SafeG architecture uses ARM TrustZone under the following configuration:

- *Virtual CPUs*: in the SafeG architecture the GPOS is assigned to the Non-Secure VCPU and the RTOS is assigned to the Secure VCPU.
- *The Monitor*: the SafeG monitor executes under monitor mode and handles the switching between the GPOS and the RTOS. The entry to SafeG monitor can only be triggered by software executing the SMC instruction or the occurrence of a FIQ interrupt while the Non-Secure VCPU is active. The SafeG monitor is small—around 2 KB [10]—and executes with all interrupts disabled, which simplifies its verification.

[*1] All figures have Secure and Non-Secure components displayed in gray and white respectively.

© 2012 Information Processing Society of Japan

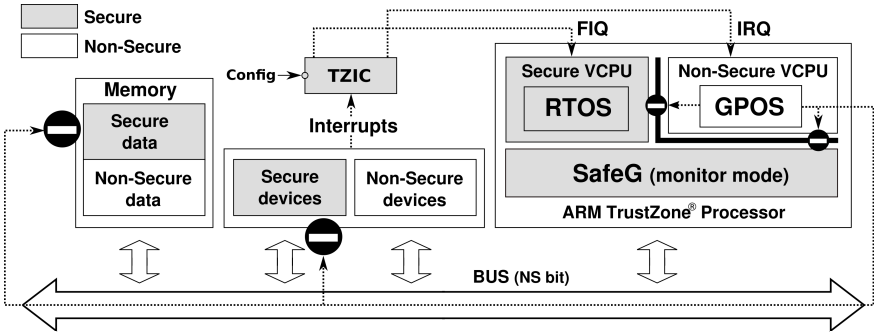


Fig. 1 SafeG architecture.

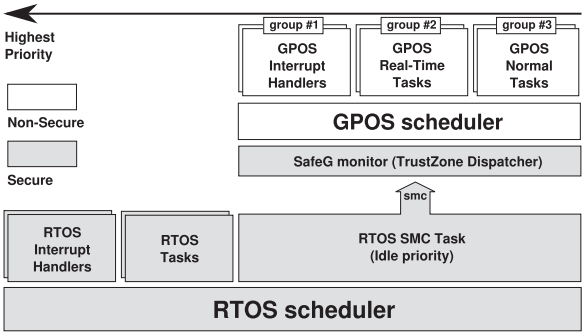


Fig. 2 SafeG idle scheduling.

A VCPU context switch on an ARM1176 [12] processor requires around 200 cycles [10].

- *Address space partitioning*: during initialization SafeG configures RTOS memory and devices as Secure world resources; and GPOS memory and devices as Non-Secure world resources. For that reason, the RTOS address space is protected from potentially malicious accesses by the GPOS.
- *Device interrupts partitioning*: SafeG architecture configures RTOS devices (i.e., Secure devices) to generate FIQ interrupts; and GPOS devices (i.e., Non-Secure devices) to generate IRQ interrupts. This is done through the TZIC [15], a Secure interrupt controller.

2.3 Execution Flow Model of SafeG

The execution flow within the SafeG architecture is controlled by two fundamental principles that allow the RTOS to guarantee real-time performance requirements to its tasks and interrupt handlers.

- *Idle scheduling*: this principle is illustrated by **Fig. 2** and implies that the GPOS is only allowed to execute during the RTOS idle time. The SafeG architecture can be seen as a two-level hierarchical scheduler where the RTOS scheduler plays the role of the global scheduler; and the GPOS scheduler acts as a local scheduler. The whole GPOS is a black box represented in the RTOS scheduler by a task executed at idle priority. We call this task *RTOS SMC Task* because its body consists of a loop executing the `smc` instruction to invoke the SafeG monitor. The SafeG monitor plays the role of a dispatcher that context switches to the GPOS whenever the RTOS becomes idle.
- *Processor control recovery*: this principle refers to the ability of the RTOS to recover control of the processor at any time through the use of a FIQ interrupt. When the RTOS is executing (i.e., Secure VCPU is active) IRQ interrupts are disabled (i.e., the I flag is one). This prevents GPOS devices interrupting the execution of RTOS tasks. On the other hand, when the GPOS is executing (i.e., Non-Secure VCPU is active) FIQ interrupts are always enabled (i.e., the F flag is zero). For that reason, the RTOS can recover the control of the processor at any time (e.g., by using a Secure timer device). The TrustZone FW configuration bit is set to zero to prevent the GPOS disabling FIQ interrupts (i.e., the GPOS cannot set the F flag to one).

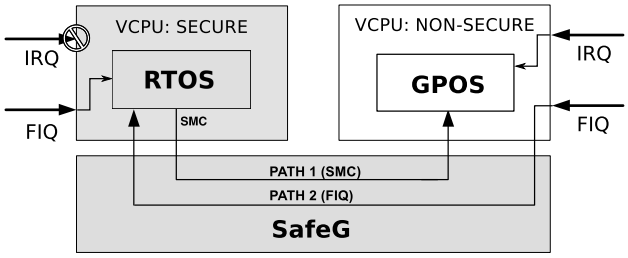


Fig. 3 Execution paths of the SafeG monitor.

Figure 3 illustrates the two main execution paths of the SafeG monitor. *PATH 1* (*SMC*) is used by the RTOS SMC Task to context switch to the GPOS whenever the RTOS becomes idle. *PATH 2* (*FIQ*) occurs when a FIQ interrupt arrives to the processor while the Non-Secure VCPU is active. The arrival of the FIQ interrupt forces the processor to enter Monitor mode, where SafeG FIQ vector handler switches back to the RTOS.

3. Integrated Scheduling

3.1 The Idle Scheduling Problem

Modern GPOSs usually count with support for activities—such as GPOS device interrupt handlers or multimedia applications—that require (soft) real-time performance [16], [17]. Unfortunately, although the idle scheduling principle—illustrated by **Fig. 4** (a)—helps the RTOS to preserve the real-time performance of the RTOS activities, as a side effect the latency of the GPOS interrupt handlers and GPOS (soft) real-time tasks is negatively affected. This negative effect is especially important when the RTOS contains compute-bound tasks executing in background (i.e., RTOS background tasks).

Suppose a system where the RTOS contains a background task that is compute-bound (e.g., a cryptographic service or secure-store application). Despite this task having no real-time require-

© 2012 Information Processing Society of Japan

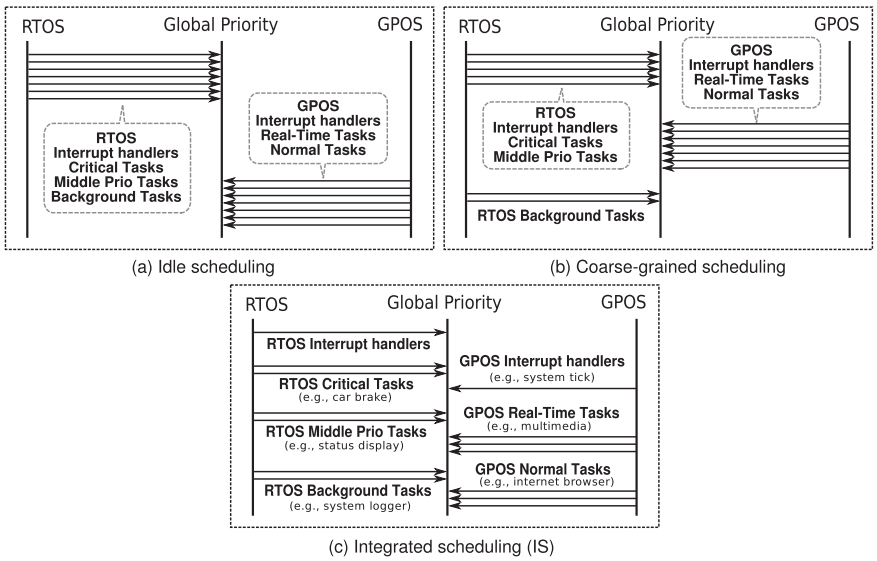


Fig. 4 Dual-OS scheduling alternatives.

ments, the execution of all GPOS interrupt handlers—including those which require a short response time to prevent some device buffers being overwritten—will be delayed until all processing by the RTOS is completed. In Section 4.8 we show an example where the use of idle scheduling causes the frame rate of a GPOS video player drop from 24 to 20 frames per second; and dramatically worsens the video watching experience.

We may try to increase the priority of the RTOS SMC Task—as illustrated by Fig. 4 (b)—to improve the latency of the GPOS interrupt handlers. However, that is a rather coarse-grained solution since the GPOS is still scheduled as a whole, including GPOS normal tasks with no latency requirements. Furthermore, a mechanism to limit the execution time of the GPOS would be needed to ensure that RTOS Background tasks do not starve.

Figure 4 (c) illustrates the concept of our proposal—the *Integrated Scheduling* (IS) architecture—which supports mixing the priority levels of the RTOS and GPOS activities with higher granularity. This allows configuring the execution priority level of GPOS interrupt handlers and (soft) real-time tasks to be higher than the one of RTOS Background tasks or RTOS tasks having long deadlines.

3.2 Assumptions and Requirements

We define the following set of initial assumptions. Relaxing these assumptions for wider usage is left for future work:

- The RTOS must support fixed priority pre-emptive scheduling. Most available RTOS kernels support this scheduling algorithm.
- The GPOS scheduler must allow tasks with (soft) real-time requirements to take precedence over normal tasks by allocating a range of higher execution priority levels. Most popular GPOSs provide this feature.
- We need access to the source code of the GPOS scheduler. For that reason we will use an open source GPOS kernel.

Next, we define a list of requirements that the IS architecture must satisfy. Many of these requirements are common to other virtualization architectures [18]:

(1) GPOS interrupt handlers and real-time tasks can be configured to take precedence over certain RTOS tasks with lower priority.
(2) The worst-case response time of RTOS interrupt handlers and tasks must remain upper-bound in all cases.
(3) The execution-time overhead introduced by the IS architecture must be small enough for practical usage.
(4) Modifications to the GPOS must be minimal and easy to maintain.
(5) The RTOS kernel must not be modified.
(6) The SafeG monitor must not be modified.

Requirement (1) refers to the ability to mix the priority levels of RTOS and GPOS activities from a global point of view. Notice that RTOS interrupt handlers still take precedence over any GPOS activity.

Requirement (2) is necessary to preserve the real-time performance requirements of the RTOS activities even when malicious or defective software is executing in the Non-Secure VCPU. In particular, RTOS tasks with low priority must be protected from execution overruns by GPOS activities executing at a higher global priority.

Requirement (3) means that the execution overhead introduced by the IS architecture must not affect the overall performance of the system to such an extent that it is no longer usable.

Requirement (4) is necessary since a GPOS is large and usually evolves very rapidly, thus increasing the maintenance costs.

Requirements (5) and (6) are defined because both the RTOS and SafeG belong to the trusted computing base. Leaving them unmodified simplifies its verification and smooths its maintainability.

3.3 Integrated Scheduling Architecture

3.3.1 Overview and Merits

The goal of the integrated scheduling (IS) architecture—illustrated by Fig. 4 (c)—is to support mixing the execution priority levels of RTOS and GPOS activities without compromising the reliability and real-time performance of the RTOS. The

© 2012 Information Processing Society of Japan

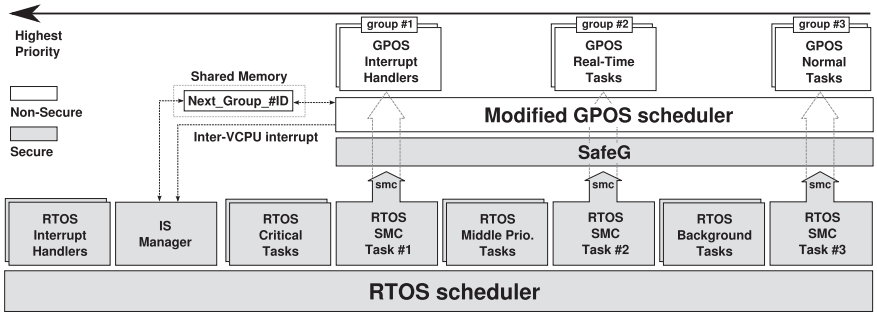


Fig. 5 Integrated scheduler architecture.

IS architecture is based on the collaboration between the RTOS user-space and the GPOS kernel scheduler. We discarded extending the SafeG monitor—basically a dispatcher that executes with all interrupts disabled—with new functionality (e.g., a scheduler or support for execution overrun timers) for several reasons: to keep the certification process of SafeG simple [10]; to minimize the latency of RTOS interrupts; to avoid increasing the complexity of trusted code; and to provide flexibility in the control of the GPOS execution (e.g., a RTOS task can easily suspend or resume the execution of the GPOS using the existing RTOS application interface).

Figure 5 depicts the integrated scheduling (IS) architecture. The main idea is to subdivide GPOS activities into a user-defined number of groups that will be scheduled at different global priority levels by the RTOS scheduler. In order to accurately map each group of GPOS activities into a different RTOS priority, the IS manager needs to track changes in the GPOS scheduled group. For that reason, the GPOS scheduler is modified to notify the IS manager about such GPOS scheduling events. This collaboration is accomplished by means of an inter-VCPU interrupt—provided by the interrupt controller—and shared memory.

Finally, since the GPOS activities execute in a non-trusted open environment we need to make sure that any execution overrun will not affect the real-time performance of the RTOS tasks. To achieve that, the IS architecture runs each group of GPOS activities under the control of a CPU time resource reservation.

3.3.2 Groups of GPOS Activities

We subdivide GPOS activities into several groups according to their latency requirements, and then we map each group to a different global priority level. Unlike the idle scheduling approach, which only has a single RTOS SMC Task at idle priority, the IS architecture assigns each group of GPOS activities to a different RTOS SMC Task, executed with a user-defined priority. In Fig. 5 we divide GPOS activities into the following 3 groups:

- Activities in group #1 (i.e., GPOS interrupt handlers) usually require a very short response time. Therefore they are represented by the RTOS SMC Task #1 which executes at a high priority.
- Activities in group #2 (i.e., GPOS real-time tasks) are usually I/O bound. They spend most of the time waiting for events to arrive, and require good responsiveness to attend to them. For that reason, they are represented by the RTOS SMC Task #2 which executes at a middle priority.
- Activities in group #3 (i.e., GPOS normal tasks) do not have special real-time requirements, and therefore they are represented by the RTOS SMC Task #3 at idle priority. In the case that only this group existed, the IS architecture would be equivalent to the idle scheduling approach.

```
1  Next_Group_ID : Positive range 1..Max_Groups;
2  Pragma Volatile(Next_Group_ID);
3
4  procedure GPOS_Scheduler_Hook (Next_Task : in TCB_Type) is
5    Prev_Group_ID : Positive range 1..Max_Groups;
6  begin
7    Prev_Group_ID := Next_Group_ID;
8    Next_Group_ID := Get_Group (Next_Task);
9    if (Next_Group_ID /= Prev_Group_ID) then
10     Send_InterVCPU_Interrupt;
11   end if;
12 end GPOS_Scheduler_Hook;
```

Fig. 6 Pseudo code of the GPOS scheduler hook function.

The IS architecture supports each group of activities being further subdivided into smaller groups—up to a single activity per group—to provide a more fine-grained scheduled system. For the sake of clarity and without loss of generality, the following explanations will use the mentioned 3 groups.

3.3.3 GPOS Scheduling Events

We define a *GPOS scheduling event* as the instant when the currently running group of GPOS activities is about to be substituted by a different group.

- Event type 1: GPOS task from group $\alpha \rightarrow$ GPOS task from group β. This scheduling event occurs when the GPOS switches tasks from different groups.
- Event type 2: GPOS interrupt handler $\rightarrow$ GPOS task. This scheduling event occurs when a GPOS interrupt handler ends and a GPOS task that belongs to a different group is resumed.
- Event type 3: GPOS task or RTOS task $\rightarrow$ GPOS interrupt handler. This scheduling event occurs when a GPOS interrupt handler interrupts the execution of a GPOS task that belongs to a different group or a RTOS task.

The IS architecture needs to keep track of all GPOS scheduling events for the IS manager in the RTOS to resume the RTOS SMC Task representing the next scheduled GPOS group. Notifications of GPOS scheduling events are sent to the RTOS through FIQ interrupts as we explain in the next two sections.

3.3.4 Tracking GPOS Scheduling Events of Type 1 and 2

We inserted a hook function into the GPOS scheduler—which is called at every context switch—in order to notify the RTOS about the occurrence of GPOS scheduling events of type 1 and 2. **Figure 6** shows the pseudo code of the GPOS scheduler hook

© 2012 Information Processing Society of Japan

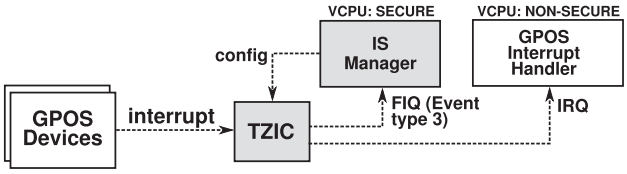


Fig. 7 Tracking GPOS interrupts.

function, which executes in GPOS kernel context and receives the control block of the next scheduled task (Next_Task) as a parameter. At every GPOS context switch, the hook function updates a variable named Next_Group_ID with the #ID of the GPOS group where Next_Task belongs to. Then, the hook function checks whether the next group differs from the previously active GPOS group or not (see line 9). In the case that the next GPOS group is different, the hook function sends an inter-VCPU interrupt to the RTOS (see line 10) as a way to notify the RTOS of the occurrence of a GPOS scheduling event of type 1 or 2. The variable Next_Group_ID is placed in inter-VCPU shared memory. It is used by the IS manager—after checking its range—for the management of the state (i.e., suspended or resumed) of the RTOS SMC Tasks. Even if malicious or faulty GPOS software intentionally set the variable Next_Group_ID to the GPOS group with the highest priority, the RTOS real-time performance is still protected by the corresponding CPU time resource reservations.

3.3.5 Tracking GPOS Scheduling Events of Type 3

GPOS scheduling events of type 3 cannot be tracked from the GPOS scheduler since GPOS device interrupts may be raised asynchronously even while the RTOS is executing. In order to track the occurrence of GPOS device interrupts within the IS architecture, we made a subtle modification to the way that the original SafeG architecture manages device interrupts:

- When a group of GPOS interrupt handlers (i.e., group #1) is not active, the corresponding GPOS device interrupts are configured as FIQ interrupts. Although this configuration violates the original SafeG architecture design (see device interrupts partitioning in Section 2.3), it allows the RTOS to be notified of the occurrence of GPOS interrupts through a FIQ handler.
- When the RTOS is notified of the occurrence of a GPOS interrupt, the corresponding GPOS group is activated and all device interrupts associated to that group are configured back as IRQ interrupts, as in the SafeG architecture.

Figure 7 illustrates the way GPOS device interrupts are managed within the IS architecture. The presented approach allows the RTOS to track the activation of a group of GPOS interrupt handlers even during the RTOS execution, when the I flag is 0. The configuration of the GPOS device interrupts is carried out through the TZIC [15] by a special software agent in the RTOS, called the IS Manager (see the next section). The overhead introduced to the RTOS is bound, but it can disturb the execution of RTOS tasks. Therefore, it must be taken into account when performing the schedulability analysis of the RTOS tasks and interrupt handlers.

3.3.6 IS Manager

The *IS Manager* is a RTOS software agent executed with higher priority (see Fig. 5) than the RTOS SMC Tasks. It is in charge of managing the RTOS SMC Tasks whenever a FIQ interrupt is raised due to the occurrence of a GPOS scheduling event. **Figure 8** shows the internal architecture of the IS Manager, which is composed of the following elements:

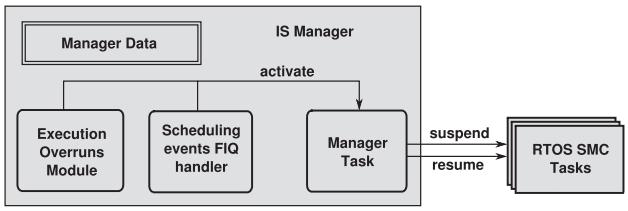


Fig. 8 IS Manager architecture.

```
1  procedure Scheduling_Event_FIQ_Handler (The_Event : in Event) is
2  begin
3    Activate_Task (Manager_Task);
4    if (The_Event.Type = 3) then
5      Next_Group_ID := Get_Group (The_Event.Interrupt)
6      Config_Group_Interrupts_As_IRQ (Next_Group_ID);
7    end if;
8  end Scheduling_Event_FIQ_Handler;
```

Fig. 9 Pseudo code of the scheduling events FIQ handler.

```
1  task body  Manager_Task is
2    Prev_SMC_Task : SMC_Task_Type;
3    Next_SMC_Task : SMC_Task_Type;
4  begin
5    Prev_SMC_Task := Get_Current_SMC_Task(Manager_Data);
6    Next_SMC_Task := Get_Next_SMC_Task(Manager_Data, Next_Group_ID);
7    if (Next_SMC_Task /= Prev_SCM_Task) then
8        Resume_Task (Next_SMC_Task);
9        Suspend_Task (Prev_SMC_Task);
10       Set_Current_SMC_Task (Manager_Data, Next_SMC_Task);
11   end if;
12 end Manager_Task;
```

Fig. 10 Pseudo code of the Manager task.

- *Scheduling events FIQ handler*: a RTOS interrupt handler (i.e., FIQ handler) which is raised whenever a GPOS scheduling event occurs. **Figure 9** shows the pseudo code of the handler. When a GPOS scheduling event occurs, the handler activates the Manager task (which is described below). Then, in the case that the event was of type 3 the handler updates the shared variable Next_Group_ID (see line 5) and configures the GPOS interrupts belonging to that group as IRQ interrupts (see line 6).
- *Manager Task*: a RTOS task aimed at controlling the state of the RTOS SMC Tasks. **Figure 10** shows the pseudo code of the Manager task. When activated by the Scheduling events FIQ handler, the Manager task calculates which RTOS SMC Task should be activated next (see line 6). Then, if that task is different to the previous one—it could be the same due to execution time overruns as we will see later—the Manager task resumes it and suspends the previous one. The RTOS SMC Task running at idle priority is a special case and remains always active since it cannot affect the real-time performance of the RTOS tasks.
- *Execution Overruns Module*: a software module aimed at limiting the execution time of each RTOS SMC Task in order to preserve the timeliness of the RTOS tasks that run with lower priority. Until now, we assumed that GPOS interrupt handlers and GPOS real-time tasks had a fixed worst-case execution time and inter-arrival period. However, a fun-

© 2012 Information Processing Society of Japan

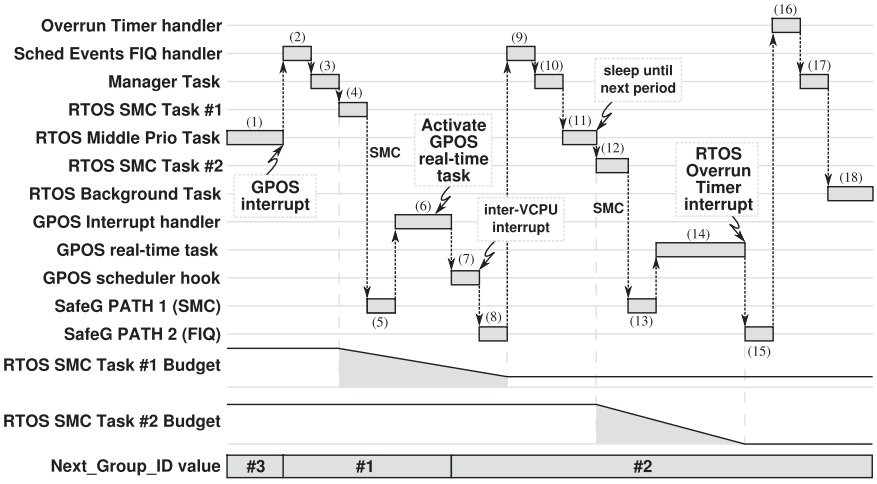


Fig. 11 Timeline: GPOS real-time task activation.

damental guideline in SafeG is to consider the GPOS as a non-trusted component. Therefore, we need to guarantee the real-time performance requirements of the RTOS activities even in the case that the GPOS misbehaves or is attacked by malicious software. For that reason, RTOS SMC Tasks are executed under the control of CPU time resource reservations [19]. A CPU time resource reservation is a mechanism to limit—and at the same time guarantee—a certain amount of execution time (the reserved budget) within a certain period and at a certain execution priority level. Each RTOS SMC Task, except the one that runs with idle priority, is assigned to a CPU time resource reservation which is implemented using the following functionality offered by the RTOS.

– *Overrun timer*: an execution-time timer that is used to keep track of the execution time consumed by the task associated to a CPU time resource reservation. When a RTOS SMC Task executes, the budget of its CPU time resource reservation is consumed. If a RTOS SMC Task exhausts its associated budget, the overrun timer expires. Then, an overrun timer handler activates the Manager task which suspends the associated RTOS SMC Task and activates another lower priority RTOS SMC Task with available budget.
– *Replenishment timer*: a timer used by the Execution Overruns Module to replenish the budget of a CPU time resource reservation. The way to replenish the budget is dependant on the implementation algorithm. In this paper, we used deferrable servers [20] which replenish the budget periodically. When a RTOS SMC task that was previously suspended after exhausting its budget receives a budget replenishment, the Manager task resumes the RTOS SMC task again if appropriate.

Interrupts emitted by the GPOS to the RTOS also consume the budget of the corresponding active RTOS SMC Task. This allows for the protection of the real-time performance of RTOS tasks with lower priority. Also note that the blocking time that an interrupt emitted by the GPOS can cause on RTOS tasks with high priority and RTOS interrupt handlers is bounded to a single occurrence.

- *Manager Data*: contains information about the RTOS SMC Tasks and is shared among the elements inside the RTOS SMC Tasks manager. The information for each RTOS SMC Task includes its execution state (i.e., suspended or resumed), priority, budget, replenishment period and overrun status.

3.3.7 Example

Figure 11 shows a simple example timeline to illustrate the execution flow of the IS architecture. The system is composed of a few tasks and interrupt handlers scheduled with the same priority order as in Fig. 5. A detailed explanation of each step is shown below:

- (1) An RTOS middle priority task executes.
- (2) A GPOS interrupt occurs and it is handled by the RTOS through the Scheduling events FIQ handler. The handler activates the Manager task, updates the Next_Group_ID variable to #1 (i.e., group of GPOS interrupt handlers) and configures GPOS device interrupts as IRQ interrupts.
- (3) The Manager task resumes the execution of the RTOS SMC Task #1.
- (4) Since the RTOS SMC Task #1 has a higher priority than the RTOS middle priority task it resumes its execution. The CPU time resource reservation associated to the RTOS SMC Task #1 starts consuming its budget.
- (5) The RTOS SMC Task #1 calls SafeG through an `smc` instruction. SafeG switches to the GPOS, as explained in Section 2.3, where the GPOS interrupt handler starts executing.
- (6) The GPOS interrupt handler activates a GPOS real-time task and ends.
- (7) After that, the GPOS scheduler executes and the GPOS scheduler hook function is called. The hook function updates the Next_Group_ID to #2 (i.e., GPOS real-time tasks) and sends a GPOS scheduling event to the RTOS through an inter-VCPU FIQ interrupt.
- (8) SafeG traps the interrupt and context switches back to the RTOS as explained in Section 2.3.
- (9) In the RTOS, the Scheduling Events FIQ handler receives the inter-VCPU interrupt and activates the Manager task.

© 2012 Information Processing Society of Japan

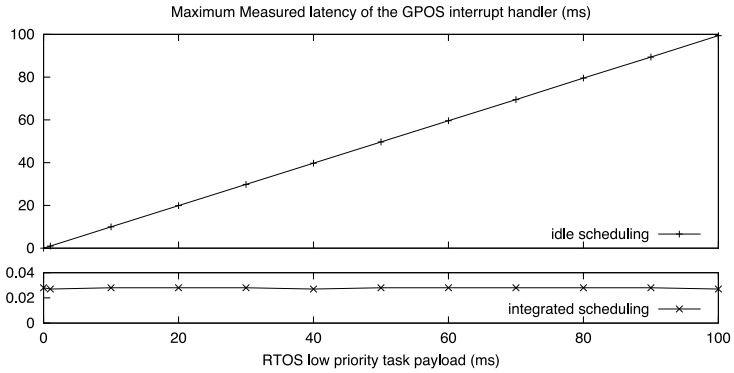


Fig. 12 Evaluation of requirement (1).

- (10) The Manager task suspends the RTOS SMC Task #1 and resumes the RTOS SMC Task #2.
- (11) Since the RTOS middle priority task has precedence over the RTOS SMC Task #2, it resumes its execution.
- (12) When the RTOS middle priority task ends—it goes to sleep until the next period—, the RTOS SMC Task #2 is scheduled and the budget associated to its CPU time resource reservation starts being consumed.
- (13) The RTOS SMC Task #2 calls SafeG through the `smc` instruction. SafeG context switches to the GPOS where the GPOS real-time starts executing.
- (14) The GPOS real-time task executes for an excessive amount of time which causes the associated RTOS overrun timer to expire.
- (15) SafeG traps the RTOS overrun timer interrupt and context switches back to the RTOS.
- (16) The interrupt is then handled by the overrun timer handler, inside the Execution Overruns Module. The handler activates the Manager task.
- (17) The Manager task suspends the GPOS RTOS Task #2. However, the Next_Group_ID variable is not modified. This only means that the group of GPOS real-time tasks will be temporarily represented by a lower priority RTOS SMC Task with available budget (in this case, the one running at idle priority) until the RTOS SMC Task #2 has its budget replenished.
- (18) The RTOS background task resumes its execution since the GPOS real-time task already exhausted its budget.

4. Evaluation

4.1 Evaluation Environment

The environment for the evaluation of the IS architecture consisted of:

- RealView Platform Baseboard (PB1176JZF-S)
  - Processor: ARM1176JZF-S [12] development chip at 210 MHz.
  - Cache: 32 KB (flushed after each measurement)
  - Mobile DDR RAM: 128 MB (used by the GPOS)
  - PSRAM: 8 MB (used by SafeG and the RTOS)
  - TZIC [15] and TZPC [14]
- RTOS: TOPPERS/ASP 1.6 [ASP] with overrun handlers enabled.
- GPOS: Linux 2.6.33 [Linux] with a minimal buildroot [21] filesystem.

4.2 Requirement (1): GPOS Latency

We evaluate requirement (1) through an experiment in order to confirm that the IS architecture allows enhancing the latency of GPOS activities. The evaluation system is composed of a periodic RTOS task executed at low priority; and a periodic timer interrupt handler on the GPOS executed at high priority (group #1). The experiment consists of measuring the maximum latency of the GPOS timer interrupt handler for several payloads in the RTOS task. The period of the RTOS task is 2 times the payload in all measurements.

The lower part of **Fig. 12** shows the experiment results for the IS architecture. We observe that the maximum latency of the GPOS interrupt handler (28 μs) remains independent of the payload of the low-priority RTOS task. In contrast, when using idle scheduling the maximum latency of the GPOS timer interrupt handler increases proportionally to the payload of the RTOS task. These results are explained by the fact that the IS architecture allows GPOS interrupt handlers to execute with higher priority than low-priority RTOS tasks.

4.3 Requirement (2): RTOS Real-time Performance Guarantees

The goal of the evaluation of this experiment is to confirm that RTOS activities preserve their real-time performance even when the Non-Secure VCPU executes malicious software. The evaluation system consists of a greedy GPOS interrupt handler—represented as part of group #1—that tries to monopolize the processor by executing a continuous loop. We measured the maximum latency of a periodic RTOS interrupt handler and a RTOS middle priority task under different budget scenarios for the GPOS group #1 (i.e., group of GPOS Interrupt handlers).

Figure 13 shows the results of the experiment. We observe that the maximum measured latency of the RTOS interrupt handler remains independent of the budget assigned to the group of GPOS interrupt handlers. However, the latency of the RTOS middle priority task is affected by the execution of the greedy GPOS interrupt handler because it has a globally lower priority.

Nonetheless, Fig. 13 confirms that the latency increase of the RTOS middle priority task is bound by the budget of the CPU

© 2012 Information Processing Society of Japan

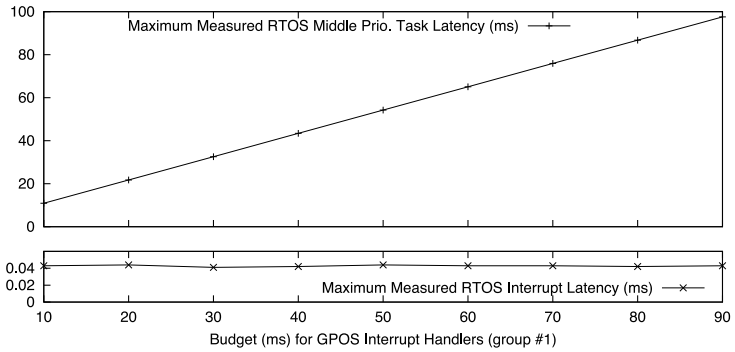


Fig. 13 Evaluation of requirement (2).

Table 1 Overhead of the IS architecture.

type	min (μs)	median (μs)	max (μs)
GPOS scheduling event type 1-2	7	15	28
GPOS scheduling event type 3	6	10	19
Budget replenishments	5	10	18
Budget overruns	5	9	20

SafeG switch time $\simeq 1.7\,\mu$s
Linux system tick = 50 ms (CONFIG_HZ=20)

Table 2 Source code lines and binary size increase.

type	new files	code increase	binary size increase
RTOS user	3	155 lines	2,716 bytes
RTOS kernel	0	0	0
GPOS kernel	5	88 lines	372 bytes
GPOS user	0	0	0
SafeG monitor	0	0	0

time resource reservation associated to the group of GPOS interrupt handlers (i.e., group #1). Therefore, the RTOS real-time performance requirements can be guaranteed. Note that the latency of the RTOS middle priority is also affected by the RTOS interrupts that occur while the GPOS interrupt handler is executing.

4.4 Requirement (3): Execution Overhead

We measured the overhead incurred by the IS architecture. **Table 1** shows the measurement results for the following 4 sources of overhead:

- *GPOS scheduling event type 1-2*: execution time overhead caused by GPOS scheduling events of type 1 and 2. The worst-case measure includes steps (7) to (12) in Fig. 11.
- *GPOS scheduling event type 3*: execution time overhead caused by GPOS scheduling events of type 3. The worst-case measure includes steps (2) to (5) in Fig. 11 plus an additional SafeG context switch (in case the GPOS interrupt was raised while the GPOS was executing).
- *Budget replenishments*: execution time overhead caused by the Replenishment timer from the Execution Overruns Module. The worst-case measure includes 2 SafeG context switches, the execution of the replenishment timer handler and the execution of the Manager task.
- *Budget overruns*: execution time overhead caused by the Overrun timers. The worst-case measure includes steps (15) to (17) in Fig. 11 plus an extra SafeG context switch to the GPOS.

The execution overheads shown in Table 1 must be taken into account during the schedulability analysis of the RTOS. However, they are small enough for practical application, and therefore we can say that the implementation of the IS architecture is able to satisfy the requirement (3).

4.5 Requirement (4): Modifications to the GPOS Kernel

During the implementation of the IS architecture, we extended the Linux kernel with a new driver which includes initialization code and stores information regarding the configuration of the GPOS groups. The module provides a callback function to notify GPOS scheduling events of type 1 and 2, which is inserted as a hook inside the Linux scheduler. The insertion of the hook is facilitated by the `ftrace` [22] kernel tracer hooks infrastructure, which reduces the necessary maintenance efforts. As **Table 2** shows, the implementation of the IS architecture on the Linux kernel required a total amount of 88 lines of C source code. 81 of those lines are independent from the Linux kernel while the remaining 7 lines correspond to the hook inserted in the scheduler.

4.6 Requirement (5): Modifications to the RTOS Kernel

Table 2 shows that the size of the modifications to the RTOS kernel is zero. This is because we managed to implement the IS architecture completely on RTOS user space by using the TOPPERS/ASP [23] application interface which includes overrun handlers, cyclic handlers, semaphores and an interface to suspend and resume tasks. The implementation required a total of 155 C source code lines and the binary size was increased by 2,716 bytes, most of them in the `.bss` section.

4.7 Requirement (6): Modifications to SafeG

In this implementation SafeG was completely unmodified and therefore requirement (6) was satisfied. The main reason for which SafeG may need to be extended in a future implementation is that the chip did not support inter-VCPU interrupts. In that case, a new SMC call should be added for SafeG to emulate inter-VCPU interrupts.

4.8 Use Case Example

In order to prove the practical applicability and effectiveness of the proposed architecture we built the real-world system illustrated by **Fig. 14**. The hardware configuration of the system consists of the following elements:

- PB1176JZF-S: the main board (see details in Section 4.1).

© 2012 Information Processing Society of Japan

- Robot: a Puppy robot [24] by Hokuto Electronics. The robot contains gyroscope and rotary encoder sensors. It is connected to the PB1176JZF-S board through a CAN bus connection which is assigned to the Secure VCPU.
- Secure Disk: a place to store secure data. It is accessed through a serial bus interface which is assigned to the Secure VCPU.
- Display: used by the Non-Secure VCPU to show a video.

The software configuration of the system is composed of the following elements:

- Robot software: the robot runs an application on top of TOPPERS/ATK [23]. The application sends the values of the gyroscope and rotary encoder sensors to the PB1176JZF-S board through the CAN bus with a period of 10 ms. Then, it waits for an answer containing an appropriate motor value for the robot to keep balance. If this value arrives to the robot later than the next period (i.e., misses its deadline) the robot will lose balance and fall down.
- Secure VCPU software: the Secure VCPU contains two tasks running on top of ASP. The *robot control task* waits for messages coming from the Puppy robot—containing the sensor values—and sends replies back with the calculated values for the motor. The *logger task* is a compute-bound task whose main function is to encrypt and store the execution log generated by the *robot control task* onto the secure disk.
- Non-Secure VCPU software: the Non-Secure VCPU contains Linux with a minimal filesystem based on buildroot [21]. On top of that, a movie player application—called mplayer [25]—is used to show a 24 fps MPEG4 video on the display through the Linux framebuffer device and executed with the maximum real-time priority in Linux.

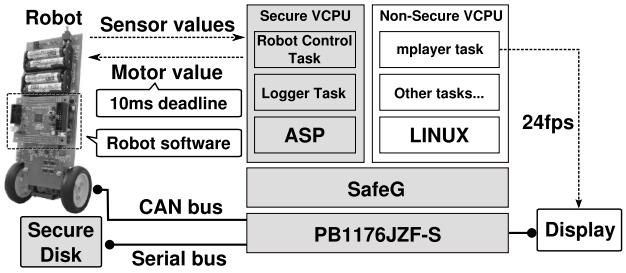


Fig. 14 Use case example.

The period and execution time of each task are described in **Table 3** (deadlines are equal to periods). We compared the performance of the system under the idle scheduling approach and the proposed IS scheduling approach—**Fig. 15** (a) and Fig. 15 (b) respectively—by measuring the frame rate of the video using the *movbench* [17] utilities. On the IS architecture, Linux interrupts and the mplayer task are executed with a global priority higher than the ASP logger task (see Fig. 15 (b)). This configuration provides (soft) real-time support for the GPOS to play the video, without being preempted by the ASP logger task every 4,000 ms. A CPU time resource reservation with 12 ms of budget and 41 ms of replenishment period was used to prevent the ASP logger task from starving. Using real-time response analysis [26], it is easy to confirm that the system is schedulable under the IS architecture, but not under the idle scheduling approach (e.g., the mplayer task has a worst case response time of 1,027 ms under idle scheduling).

Figure 16 shows the dynamic frame rate at which the video is played under both scheduling approaches. We observed that the frame rate under the idle scheduling approach experiments strong drops approximately every 4,000 ms. This is caused by the blocking time imposed by the execution of the ASP logger task. The average frame rate is reduced to 20 frames-per-second, and most importantly the video watching experience gets dramatically worsened. In contrast, we observed that under the IS architecture the video is played smoothly at a constant rate of 24 fps

Table 3 Tasks in the use case example.

OS	task name	period (ms)	execution time (ms)
ASP	Robot Control task	10	5
ASP	Logger task	4,000	500
Linux	mplayer task	41	12

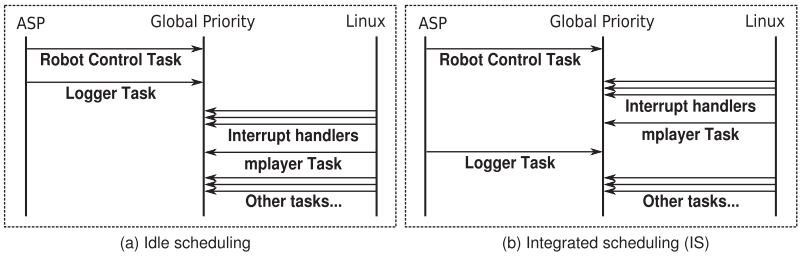


Fig. 15 Execution priority levels for the use case example.

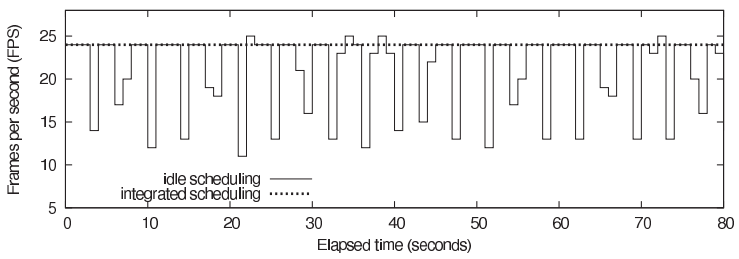


Fig. 16 Frames per second in each scheduling approach.

© 2012 Information Processing Society of Japan

without causing any deadline miss to the ASP robot control task nor to the ASP logger task, and therefore proving the practical applicability of the presented proposal.

5. Related Work

There is a large amount of literature proposing different dual-OS mechanisms to enable the concurrent execution of a GPOS and a RTOS under time and memory isolation conditions. However, research on dual-OS scheduling is rather scarce and most of it can be classified in the following groups:

- *Idle scheduling*: dual-OS monitors that schedule the GPOS as the RTOS idle task belong to this group. A few examples are RTAI [8], Linux on ITRON [5] and more recently, MobiVMM [7]. Reference [27] proposes a combination of a time-driven, priority-based and proportionally shared scheduler. However, in their proposal the RTOS is always assigned a static high priority, and therefore providing responsiveness to the GPOS is rather difficult.
- *Compositional scheduling*: this includes dual-OS monitors that schedule the RTOS and GPOS using time-based compositional scheduling frameworks. An example is the XTRATUM [28] hypervisor whose scheduler is based on the ARINC-653 specification [29]. This approach allows for a great degree of time isolation between the guest OSs. However, it is not suitable for event-driven processing, such as interrupts, that require very short latencies.
- *Fair scheduling*: a typical example is the XEN [30] credit scheduler which enables sharing the CPU proportionally between the guest OS. XEN has several mechanisms [31] to improve the responsiveness of I/O bound guest OSs but it is not able to cope with the hard real-time requirements of a RTOS.

The idle-scheduling problem described in Section 3.1 was also identified by Ref. [32], where a task grain scheduling algorithm for a virtualized embedded system was presented. The architecture proposed in Ref. [32] uses the L4-embedded microkernel as a hypervisor running para-virtualized versions of Linux (Wombat) and TOPPERS/JSP (L4/TOPPERS). In order to implement task grain scheduling, each of the guest operating systems notifies the priority of the running task to a global scheduler. However, their approach does not protect RTOS tasks real-time properties from GPOS misbehavior and is not capable of mixing the priority of GPOS interrupt handlers with RTOS tasks. Also, the architecture proposed in that approach requires an extra global scheduler while in the IS architecture the RTOS scheduler plays that role.

6. Conclusions and Future Work

We presented an integrated scheduling architecture for a reliable dual-OS virtualization technique. The architecture is based on the collaboration of the RTOS and the GPOS scheduler and allows mixing the execution priority levels of RTOS and GPOS activities for enhancing the responsiveness of the GPOS. At the same time, the real-time performance requirements of the RTOS is preserved thanks to the use of overrun control mechanisms. The evaluation results showed that all requirements stated in Section 3.2 were satisfied. We also built a use case example to prove the practical applicability of our proposal in a real situation.

In future work, we plan to extend SafeG with an inter-OS communications system. We will take advantage of the asymmetric privileges of a dual-OS system (i.e., the RTOS can access memory assigned to the GPOS but the inverse is not possible) to design an efficient and reliable communications architecture. The communications architecture is expected to benefit from the IS approach to enhance the latency of messages transmitted between both operating systems. We also plan to investigate the application of SafeG to new multi-core TrustZone processors. In more detail, we will try to find a solution for the lock-holder preemption problem [33] by leveraging the functionality provided by the IS architecture presented in this paper.

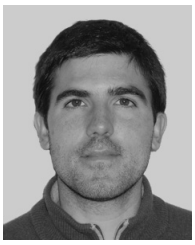
Acknowledgments Part of this work is supported by the KAKENHI (23700035) and the Monbukagakusho scholarship. We would also like to express our gratitude to the reviewers of this paper for their insightful comments.

Reference

[1] Android: Official Website, available from ⟨http://www.android.com/⟩.
[2] Heiser, G.: The Role of Virtualization in Embedded Systems, *Proc. 1st Workshop on Isolation and Integration in Embedded Systems*, Glasgow, UK, pp.11–16, ACM SIGOPS (2008).
[3] Heiser, G.: Hypervisors for consumer electronics, *CCNC'09: Proc. 6th IEEE Conference on Consumer Communications and Networking Conference*, pp.614–618, Piscataway, NJ, USA, IEEE Press (2009).
[4] Hergenhan, A. and Heiser, G.: Operating Systems Technology for Converged ECUs, *6th Embedded Security in Cars Conference* (*ESCAR*), Hamburg, Germany, ISITS (2008).
[5] Takada, H., Iiyama, S., Kindaichi, T. and Hachiya, S.: Linux on ITRON: A Hybrid Operating System Architecture for Embedded Systems, *SAINT-W '02: Proc. 2002 Symposium on Applications and the Internet* (*SAINT*) *Workshops*, pp.4–7, Washington, DC, USA, IEEE Computer Society (2002).
[6] Cereia, M. and Bertolotti, I.: Asymmetric virtualisation for real-time systems, pp.1680–1685, *ISIE 2008*, Cambridge (2008).
[7] Yoo, S., Liu, Y., Hong, C., Yoo, C. and Zhang, Y.: MobiVMM: A virtual machine monitor for mobile phones, *MobiVirt '08: Proc. 1st Workshop on Virtualization in Mobile Computing*, pp.1–5, New York, USA, ACM (2008).
[8] RTAI: Official website, available from ⟨https://www.rtai.org/⟩.
[9] Wilson, P., Frey, A., Mihm, T., Kershaw, D. and Alves, T.: Implementing Embedded Security on Dual-Virtual-CPU Systems, *Design Test of Computers*, Vol.24, No.6, pp.582–591, IEEE (2007).
[10] Nakajima, K., Honda, S., Teshima, S. and Takada, H.: Enhancing Reliability in Hybrid OS System with Security Hardware, *IEICE Trans. Inf. Syst.*, Vol.93, No.2, pp.75–85 (2010).
[11] ARM Ltd.: *ARM Security Technology, Building a Secure System using TrustZone Technology, PRD29-GENC-009492C* (2009).
[12] ARM Ltd.: *ARM1176JZF-S. Technical Reference Manual, DDI 0301G* (2008).
[13] Cereia, M. and Bertolotti, I.C.: Virtual machines for distributed real-time systems, *Computer Standards Interfaces*, Vol.31, pp.30–39 (2009).
[14] ARM Ltd.: AMBA3 TrustZone Protection Controller (BP147) Technical Overview, DTO 0015A (2004).
[15] ARM Ltd.: AMBA3 TrustZone Interrupt Controller (SP890) Technical Overview, DTO 0013B (2008).
[16] Marquet, P., Piel, E., Soula, J. and Dekeyser, J.: Implementation of ARTiS, an asymmetric real-time extension of SMP Linux, 6th Real-Time Linux Workshop, Singapore (Nov. 2004).
[17] Kato, S., Ishikawa, Y. and Rajkumar, R.: CPU Scheduling and Memory Management for Interactive Real-Time Applications, *Real-Time Systems Journal*, Vol.47, pp.454–488 (2011).
[18] Armand, F. and Gien, M.: A practical look at micro-kernels and virtual machine monitors, *Proc. 6th IEEE Conference on Consumer Communications and Networking Conference*, *CCNC'09*, pp.395–401, Piscataway, USA, IEEE Press (2009).
[19] FRESCOR: Official Website, available from ⟨http://www.frescor.org⟩.
[20] Bernat, G. and Burns, A.: New results on fixed priority aperiodic servers, *20th IEEE Real-Time Systems Symposium*, Phoenix, USA (1999).

© 2012 Information Processing Society of Japan

[21] Buildroot: Official website, available from ⟨http://buildroot.uclibc.org/⟩.
[22] Rostedt, S.: Finding origins of latencies using Ftrace, *11th Real-Time Linux Workshop*, Dresden, Germany (Sep. 28–30, 2009).
[23] TOPPERS: Toyohashi OPen Platform for Embedded Real-Time Systems, available from ⟨http://www.toppers.jp⟩.
[24] Electronic, H.: Official website, available from ⟨http://www.hokutodenshi.co.jp/7/PUPPY.htm⟩.
[25] MPlayer: Official website, available from ⟨http://www.mplayerhq.hu/⟩.
[26] Burns, A. and Wellings, A.: *Real-Time Systems and Programming Languages Third Edition*, Addison Wesley (2001).
[27] Kaiser, R.: Alternatives for scheduling virtual machines in real-time embedded systems, *Proc. 1st Workshop on Isolation and Integration in Embedded Systems*, *IIES '08*, pp.5–10, New York, USA, ACM (2008).
[28] Masmano, M., Ripoll, I., Crespo, A. and Metge, J.: XtratuM: A Hypervisor for Safety Critical Embedded Systems, *11th Real-Time Linux Workshop*, Dresden, Germany (2009).
[29] ARINC-653, Avionics Application Software Standard Interface, Airlines Electronic Engineering Committee, Annapolis, Maryland (1996).
[30] Hwang, J., Suh, S., Heo, S., Park, C., Ryu, J. and Kim, C.: Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones, *5th Annual IEEE Consumer Communications & Networking Conference*, USA (2008).
[31] Ongaro, D., Cox, A. and Rixner, S.: Scheduling I/O in virtual machine monitors, *Proc. 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *VEE '08*, pp.1–10, New York, USA, ACM (2008).
[32] Kinebuchi, Y., Sugaya, M., Oikawa, S. and Nakajima, T.: Task Grain Scheduling for Hypervisor-Based Embedded System, *Proc. 10th IEEE International Conference on High Performance Computing and Communications*, *HPCC '08*, pp.190–197, Washington, USA, IEEE Computer Society (2008).
[33] Uhlig, V., LeVasseur, J., Skoglund, E. and Dannowski, U.: Towards scalable multi-processor virtual machines, *VM'04, 3rd Conference on Virtual Machine Research And Technology Symposium*, Berkeley, USA (2004).



Daniel Sangorrin is currently a Ph.D. student at the Department of Information Engineering in the Graduate School of Information Science, Nagoya University. He received his long-cycle degree on Ingenieria de Telecomunicacion at the University of Cantabria (Spain) in 2006. From 2006 to 2009, he was a researcher for the project FRESCOR (FP6/2005/IST/5-034026). His research interests include distributed real-time embedded systems and machine virtualization. He is a member of Ada-Spain.

Shinya Honda received his Ph.D. degree in the Department of Electronic and Information Engineering, Toyohashi University of Technology in 2005. From 2004 to 2006, he was a researcher at Nagoya University Extension Course for Embedded Software Specialists. In 2006, he joined the Center for Embedded Computing Systems, Nagoya University (NCES), as an assistant professor, where he is now an associate professor. His research interests include system-level design automation and real-time operating systems. He received the Best Paper Award from IPSJ in 2003. He is a member of IPSJ, IEICE, and JSSST.

Hiroaki Takada is a professor at the Department of Information Engineering, the Graduate School of Information Science, Nagoya University. He is also the Executive Director of the Center for Embedded Computing Systems (NCES). He received his Ph.D. degree in Information Science from the University of Tokyo in 1996. He was a research associate at the University of Tokyo from 1989 to 1997, and was a lecturer and then an associate professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IPSJ, IEICE, JSSST, and JSAE.

© 2012 Information Processing Society of Japan