6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)

in Conjunction with the
22nd Euromicro Conference on Real-Time Systems
(ECRTS 2010)

July 6, 2010, Brussels, Belgium

POLITÉCNICO DO PORTO

Edições POLITEMA

Stefan M. Petters and Peter Zijlstra (Editors)

International Workshop on Operating Systems Platforms for Embedded Real-Time Applications

Workshop Proceedings

Table of Contents

Editors:
Stefan M. Petters
Peter Zijlstra

Message from the Workshop Chairs

It has been a pleasure to serve again in the preparation of this year's Workshop on Operating Systems Platforms for Embedded Real-Time Applications. Again we aimed for an interactive format in the workshop providing a discussion forum for novel ideas as well as the interaction between academics and practitioners. To enable this we have looked at providing ample of discussion time both in between paper presentations as well as in the dedicated discussion slots.

Obviously such endeavours are not the result of one or two individuals working, but are the product of many helping hands. The first thanks goes to Gerhard Fohler for trusting us in our second go at this Workshop.

At the start of the day and after lunch between the discussion sessions we have scheduled three paper presentations sessions. The 10 papers presented were selected out of a total of 16 submissions. We thank all the authors for their hard work and submitting it to the workshop for selection, the PC members and reviewers for their effort in selecting an interesting program, as well as the presenters for ensuring that this will be an entertaining and informative day.

Last, but not least, we would like to thank you, the audience, for your attendance. A workshop lives and breathes because of the people asking questions and contributing opinions throughout the day.

We hope you will find this day interesting and enjoyable.

The Workshop Chairs

Stefan M. Petters
Peter Zijlstra

Program Committee

David Andrews, University of Arkansas Fayettville, USA

Neil Audsley, University of York, UK

Peter Chubb, NICTA, Australia

Steve Goddard, University of Nebraska Lincoln, USA

Hermann Härtig, TU Dresden, Germany

Johannes Helander, Microsoft, Germany

Robert Kaiser, University of Applied Sciences Wiesbaden, Germany

Tei-Wei Kuo, National Taiwan University, Taiwan

Stefan M. Petters, IPP-Hurray, Portugal

Peter Zijlstra, Red Hat, Netherlands

Program:

09:00-10:30 Session 1: Virtualisation

Dual Operating System Architecture for Real-Time Embedded Systems
Daniel Sangorrin, Shinya Honda, and Hiroaki Takada; *Nagoya University, Japan*

Timeslice Donation in Component-Based Systems
Udo Steinberg, Alexander Böttcher, and Bernhard Kaue; *TU Dresden, Germany*

Full Virtualization of Real-Time Systems by Temporal Partitioning
Timo Kerstan, Daniel Baldin, and Stefan Groesbrink ; *Univerisy of Paderborn, Germany*

10:30-11:00 Coffee Break

11:00-12:30 Session 2: Panel Discussion Linux Scheduler Meet Real-Time
12:30 -13:30 Lunch

13:30-15:30 Session 3: Implementation and Analysis

Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability
Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson; *University of North Carolina at Chapel Hill, USA*

Implementation of Overrun and Skipping in VxWorks
Mikael Åsberg, Moris Behnam, Thomas Nolte, and Reinder Bril; *Maelardalen RTC, Sweden and TU Eindhofen, Netherlands*

Schedulable Device Drivers: Implementation and Experimental Results
Nicola Manica, Luca Abeni, Luigi Palopoli, Dario Faggioli, and Claudio Scordino; *University of Trento, Scuola Superiore S. Anna, and Evidence Srl, Italy*

Evaluating Android OS for Embedded Real-Time Systems
Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho; *Polytechnic Institute of Porto, Portugal*

15:30-16:00 Coffee Break 16:00-17:30 Session 4: Resource Sharing and Communication

Extending an HSF-enabled Open Source Real-Time Operating System with Resource Sharing
Martijn M.H.P. van den Heuvel, Reinder J. Bril, Johan J. Lukkien, and Moris Behnam; *TU Eindhofen, Netherlands and Maelardalen RTC, Sweden*

Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux
Andreu Carminati, Rômulo Silva de Oliveira, Luís Fernando Friedrich, and Rodrigo Lange; *Federal University of Santa Catarina, Brazil*

The Case for Thread Migration: Predictable IPC in a Customizable and Reliable OS
Gabriel Parmer; *George Washington University, USA*

17:30-18:00 Session 5: Group Discussion Wrap up and lessons learned

Dual Operating System Architecture for Real-Time Embedded Systems

Daniel Sangorrin, Shinya Honda and Hiroaki Takada

Graduate School of Information Science, Nagoya University, Japan
{dsl,honda,hiro}@ertl.jp

Abstract

Virtualization architectures for the combination of real-time and high-level application tasks, on the same embedded platform, pose special reliability and integration requirements compared to solutions for the enterprise domain. This paper presents a software architecture to execute concurrently, on a single processor, a real-time operating system (RTOS) and a general-purpose operating system (GPOS). The proposed approach, based on common embedded security hardware (ARM TrustZone®), enables integrated scheduling of both operating systems to enhance the responsiveness of the GPOS soft real-time tasks and interrupts while preserving the determinism of the RTOS and without modifications to the GPOS core. The presented architecture has been implemented and evaluated on real hardware. Its low overhead and reliability makes it suitable for embedded applications such as car navigation systems, mobile phones or machine tools.

1. Introduction

In recent years, methods for integrating real-time control systems and high-level information systems on a single platform to reduce product costs are gaining considerable interest from different embedded domains [14]. For example, the market for high function in-vehicle technology has experienced a rapid growth. New car functionality may include satellite navigation, road information, entertainment systems or Internet connectivity. In addition, parking and driving aid systems use the information provided by those applications to cooperate with control systems for the steering gear or the engine [16].

In order to develop high-level applications (e.g., a web browser or media player) efficiently, a general-purpose operating system (GPOS) with a high level of functionality is usually essential. However, most GPOS are not able to satisfy the strict requirements of real-time control systems in terms of security, reliability and determinism [15]. For instance, security holes are discovered continuously in GPOS such as Windows or GNU/Linux [17]. For that reason, different solutions that execute a real-time operating system (RTOS) in parallel with a GPOS have been proposed.

In *hybrid kernel* methods [22, 21] both operating systems execute in privileged level with direct access to the hardware as shown in Fig 1. Although they have the advantage of achieving very low overhead, there is no hardware protection between them, and the RTOS can therefore be threatened by a malicious attack or misbehavior of the GPOS. For example, if the GPOS runs out of control with interrupts disabled, the RTOS would not be able to recover control. Furthermore, the GPOS may access memory assigned to the RTOS, causing its failure or stealing sensitive information.

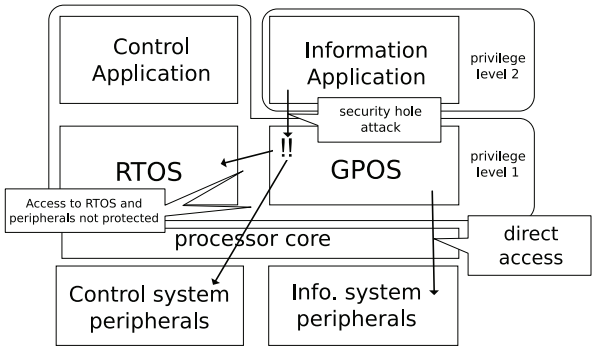


Figure 1: Hybrid kernel method

Virtual Machine Monitor (VMM/hypervisor) methods [15, 8, 9, 24], on the other hand, provide strong isolation among multiple guest operating systems by executing them under a lower privilege level. The disadvantages of these methods are typically the additional execution overhead caused by privileged instruction emulation; the modifications needed on the guest operating systems [20]; and the loss of isolation caused by DMA capable devices [19]. Fortunately, some embedded processor architectures have recently introduced hardware extensions that facilitate the process of virtualization [23].

A virtualization architecture designed for real-time embedded systems must use a deterministic scheduling algorithm. In some approaches both operating systems are scheduled as black boxes using fixed cyclic scheduling [9]. The main problem with this method is that the worst case response time of the RTOS's activities depends on the size of the slot assigned to the GPOS. This is especially a problem for RTOS interrupts that require short response times. In many other approaches, the GPOS is only executed when the RTOS becomes idle [13, 24]. This method allows the RTOS to take precedence over the GPOS and thus maintain its determinism and low response times. However not all RTOS tasks require the same degree of responsiveness [22],

and some GPOS applications and interrupt handlers, such as multimedia on mobile wireless devices, require a certain quality of service [18].

This paper presents a dual operating system virtualization architecture that supports integrated scheduling to enhance the responsiveness of the GPOS while preserving the determinism of the RTOS. The proposed approach takes advantage of common embedded security hardware (ARM TrustZone®) to improve the reliability and isolation of the RTOS with low overhead and no modifications to the GPOS core.

The paper is organized as follows. Sec. 2 introduces, briefly, the ARM TrustZone capabilities. Sec. 3 outlines the set of requirements that must be satisfied by the presented architecture, whose design, implementation and evaluation are explained in Sec. 4, 5 and 6. Finally, Sec. 7 draws some conclusions and discusses future work.

2. ARM TrustZone

This section briefly introduces the ARM TrustZone® security extensions that will be used for the presented virtualization architecture. For more information, refer to [4, 5].

2.1. Trust vs. Non-Trust concept

ARM processors define two privilege levels. In privileged mode, all the system resources can be accessed. On the other hand, in user mode, access to resources is restricted. In a GPOS, the kernel usually runs in privileged mode while applications run in user mode. TrustZone is orthogonal to privilege levels, adding the so-called *Trust* and *Non-Trust* states.

Trust state provides similar behavior to existing privileged and user mode levels. On the other hand, code running under Non-Trust state, even in privileged mode, cannot access memory space (including devices) that was allocated for Trust state usage, nor can it execute certain instructions that are considered critical.

In order to control the TrustZone state, a new mode, called Secure Monitor mode, has been added to the processor. Switching between Trust and Non-Trust state is performed under Security Monitor mode. As a general rule, code in Security Monitor mode runs with interrupts disabled to avoid registers being overwritten when an interrupt arrives.

MMU registers and part of the control registers are banked. When switching the TrustZone state, only general purpose registers that are not banked need to be saved. TrustZone is also supported by the caches to avoid flushing the cache when switching between both states. These features are important for reducing the overhead of switching between Trust and Non-Trust states.

2.2. Address space partitioning

In TrustZone, the address space is divided between regions only accessible in Trust state (Trust area) and regions accessible from both states (Non-Trust area).

When the processor core accesses the bus, a signal indicates the current state (Trust/Non-Trust). Bus controllers and devices can use that signal to determine the state from which the access was performed. The TrustZone protection controller [3] can be used to configure different regions of memory as Trust or Non-Trust space.

2.3. Interrupts

In the ARM processor, there are two types of interrupt signals: FIQ and IRQ. When a FIQ or IRQ is generated, the execution is suspended and the program counter is loaded with the address of the corresponding interrupt vector. In TrustZone, there are independent interrupt vectors for the Trust and Non-Trust state. If an interrupt occurs while executing in Trust state, the Trust state vector will be executed, and vice versa. In addition, Secure Monitor mode has its own vector table. It is possible to configure whether FIQ and IRQ are handled by the Trust/Non-Trust vectors, or by the Secure Monitor vectors.

In ARM processors, FIQ and IRQ interrupts can be disabled separately. With TrustZone, it is possible to prevent the Non-Trust side from disabling FIQ interrupts (IRQs can be disabled). For this reason, it is recommended to use IRQs for the Non-Trust state, and FIQs for the Trust state. Distribution of interrupts between both states can be done through the TrustZone Interrupt Controller [2].

3. VMM requirements

The following set of requirements were specified for the presented VMM architecture, taking into account the needs of different embedded domains, such as car navigation systems, mobile phones and machine tools.

(a) Support concurrent execution of a GPOS and an RTOS on an ARM TrustZone single processor.

(b) Spatial isolation of the RTOS. GPOS failures cannot spread to the RTOS.

(c) Time isolation of the RTOS. The real-time deterministic behavior of the RTOS must not be affected by the GPOS.

(d) Support integrated scheduling of the GPOS soft real-time tasks and interrupts.

(e) Basic mechanisms for device sharing. No overhead must be introduced for devices that are not shared.

(f) Mechanisms to implement a health monitoring system at the RTOS to monitor the GPOS.

(g) No modifications to the GPOS core (i.e., dispatcher or interrupt handling code) are required. On the other hand, changes to the RTOS are allowed due to its lower scale.

(h) The TrustZone monitor implementation must have an execution time smaller than the RTOS interrupt latency.

(i) The code of the TrustZone monitor implementation must be small and easy to verify.

As shown by requirement (a), the goal of this research is to build a dual operating system architecture, based on security hardware extensions, on a single low-cost processor. Support for multi-processor architectures is planned for future research.

Requirement (b) is specified due to the difficulty of increasing the reliability level of a GPOS to the one of a lower size RTOS.

Requirement (c) is specified because systems controlled by the RTOS have stricter real-time requirements than the GPOS processing, while time-consuming tasks such as route searching are better handled by the GPOS. The worst case interrupt response time of the RTOS must be independent of the length of the maximum critical section of the GPOS.

Requirement (d) is specified because some GPOS applications (e.g., multimedia) require a certain quality of service. Integrated scheduling methods to improve the responsiveness of the GPOS without affecting the determinism of the RTOS are desired.

Requirement (e) is related to device sharing. For example, typical devices in a car navigation system include a display, storage disk, in-vehicle network (CAN), timers, to name a few. Not all devices need to be shared. For example, the display is usually able to allocate a frame buffer for each OS, where the RTOS frame buffer has preference over the GPOS buffer. Also, CAN is only used by the control system and each OS has an independent timer device. It is preferable that the use of virtualization does not introduce additional overhead to the operation of devices which do not need to be shared. The hard disk is an example of a device that may require sharing, because the RTOS may need to store critical data. In this study only the basic mechanisms to share a device between Trust and Non-Trust are investigated, leaving for future research a more detailed architecture with a more sophisticated application interface.

An example where requirement (f) is important is a car navigation rear guide application which usually runs on the GPOS. If the image displayed becomes frozen due to some problem, there is a high risk of collision. For that reason, the process of updating the image may need to be monitored from the RTOS. If the GPOS application suffers a failure, a system at the RTOS could be instructed to take the appropriate measurements. The infrastructure for developing a health monitor application that has been investigated in this study includes low-level mechanisms for monitoring the status of the GPOS and its interrupts and an interface to stop, reset and resume the operation of the GPOS.

Regarding requirement (g), in order to run two operating systems simultaneously on a single processor, changes on each of those operating systems might be inevitable. When changes are performed on a large scale, software verification becomes very difficult. In addition, the maintenance of such modifications on different versions of the GPOS usually requires a considerable engineering effort. Therefore changes in the GPOS must be minimized. On the other hand, the RTOS has a lower scale, and so changes are allowed.

Requirement (h) is a performance requirement. The introduction of a TrustZone monitor causes overhead in the interrupt handling due to the necessity to switch between both operating systems. This overhead needs to be as small as possible.

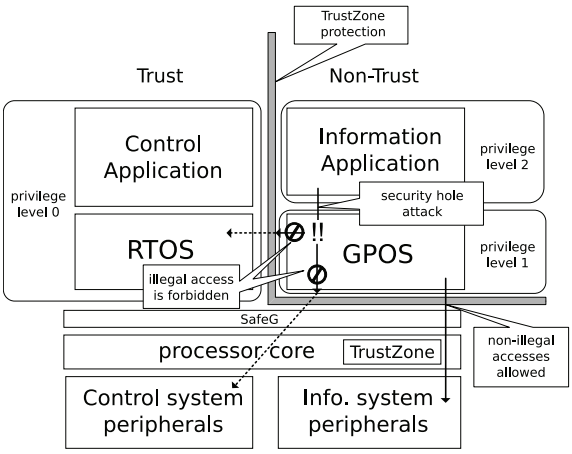


Figure 2: VMM based on TrustZone

Finally, requirement (i) is specified because the TrustZone monitor is the cornerstone of the presented architecture. If its reliability is deteriorated, the whole system reliability will be affected. Some important factors for its verifiability are the number of tests required to cover all its possible execution paths, and the size of its code for review.

4. VMM architecture

This section describes the virtualization architecture proposed in this paper, which has been designed with the previous requirements in mind.

4.1. TrustZone configuration

The overall organization of the system is depicted in Fig. 2. To satisfy requirement (a), a TrustZone Monitor called SafeG (Safety Gate) has been designed to execute in Secure Monitor mode and handle the switching between the GPOS, executed in Non-Trust state, and the RTOS, executed in Trust state. The implementation of SafeG is described in Sec. 5.1.

Spatial isolation (requirement (b)) is supported by configuring resources (memory and devices) used by the RTOS to be accessible only from Trust state. The remaining resources are configured to be accessible both from Trust and Non-Trust state. This configuration is performed at initialization time after SafeG is loaded. If the GPOS tries to access some resource configured as Trust space, an exception occurs and SafeG is called.

Time isolation of the RTOS (requirement (c)) is supported by carefully using the two types of interrupt. FIQ interrupts are forwarded to the RTOS, while IRQ interrupts are forwarded to the GPOS. In Trust state, IRQs are disabled so that the GPOS cannot interrupt the execution of the RTOS. For that reason, the GPOS can only execute once the RTOS makes an explicit request, through a Secure Monitor Call (SMC), to SafeG. On the other hand, during the GPOS execution, FIQs are enabled so that the RTOS can recover the control of the processor. TrustZone is configured to prevent the Non-Trust side from disabling FIQ interrupts.

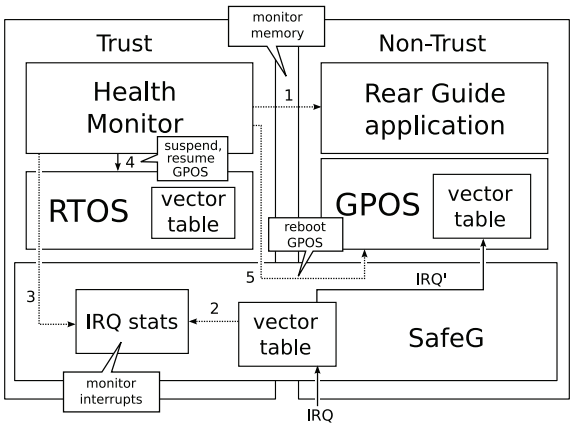


Figure 3: Health monitoring mechanisms

4.2. Support for health monitoring

Some useful mechanisms for requirement (f) include the ability to monitor, suspend, resume and restart the operation of the GPOS. Fig. 3 shows the health monitoring mechanisms in the presented architecture. Monitoring the GPOS status from the RTOS (access 1) is possible because the GPOS resides in Non-Trust space memory, which is accessible from Trust state. To support GPOS interrupt monitoring, IRQs are first processed by SafeG, which implements a Secure Monitor mode vector table, before being forwarded to the GPOS. The frequency and inter-arrival time of the GPOS interrupts can be tracked (access 2) and used by a particular health monitor application in the RTOS (access 3). The operation of the GPOS can be suspended or resumed (access 4) using the RTOS application interface, as described in Sec. 4.4.1. In addition, SafeG offers an SMC to reboot (access 5) the GPOS from the RTOS.

4.3. Device sharing

Devices that do not need to be shared are configured to Trust or Non-Trust space and are accessed directly, without any additional overhead. A basic mechanism for sharing the remaining devices has been designed to achieve requirement (e). More refined methods, including a standard interface, are left for future research. The mechanism is based on using an SMC to make a request to the other OS to handle a certain device. For example, the RTOS may need to handle a shared disk to store sensible data while the GPOS needs to make a request which will be verified. In order to transmit the request, the GPOS executes an SMC together with some parameters, for example the address of the buffer with the data to be stored. The SMC call is handled by SafeG which switches to the Trust state. During the switch, SafeG copies the request type and parameters to the RTOS task, see details in Table 1, that made a switch request previously. In addition, SafeG can be configured to force a special exception in the RTOS when immediate processing is desired. The buffers to store or read the associated data must be in Non-Trust memory so that they can be accessed from both operating systems.

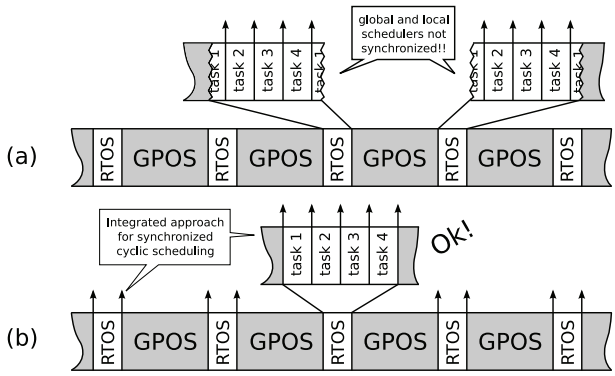


Figure 4: Black box vs Integrated cyclic scheduling

4.4. Integrated Scheduling

4.4.1 The GPOS as an RTOS task

In most VMM architectures, virtual machines are scheduled as black boxes. However, using such a hierarchical scheduling approach, it is difficult to support the integrated scheduling indicated in requirement (d). Furthermore, implementing a new scheduler inside SafeG would complicate its verification, making requirement (h) difficult to satisfy. It would also increase the interrupt latency, since SafeG is executed with interrupts disabled. For that reason, in the presented architecture, the RTOS is used to schedule the GPOS, which is represented as a normal RTOS task. Representing the execution of the GPOS as a fully featured RTOS task gives the user the possibility to use the native RTOS application interface to suspend or resume the operation of the GPOS. This functionality can be used to produce an integrated approach, both for cyclic and priority-based scheduling, and this will be shown in Sec. 4.4.2 and 4.4.3.

4.4.2 Cyclic scheduling

Cyclic scheduling offers very good determinism properties. Proof of that is the fact that it is used for scheduling virtual machines (partitions) in VMMs aimed at safety critical systems such as spatial systems [9], or in the avionics ARINC 653 standard [1].

One of the problems of cyclic scheduling when applied to dual virtual machines is depicted in Fig. 4 (a). If the RTOS is also based on a cyclic scheduler, it becomes difficult to maintain both the global and internal schedulers synchronized.

In the presented architecture, an integrated cyclic scheduling approach has been implemented. A periodic handler is used, in the RTOS, to suspend and resume the execution of the task associated with the GPOS. With this approach it is easy to produce the synchronized schedule shown in Fig. 4 (b).

Another problem of scheduling virtual machines as black boxes is that the worst case latency of every RTOS operation becomes dependent on the length of the GPOS time slice. This is especially a problem for RTOS interrupts that require a very short response time.

In the presented architecture FIQ interrupts—which are

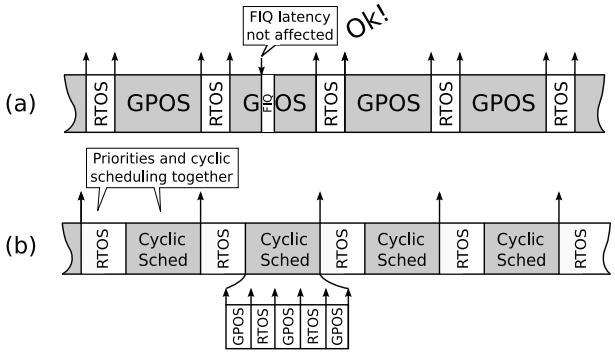


Figure 5: Latency in integrated cyclic scheduling

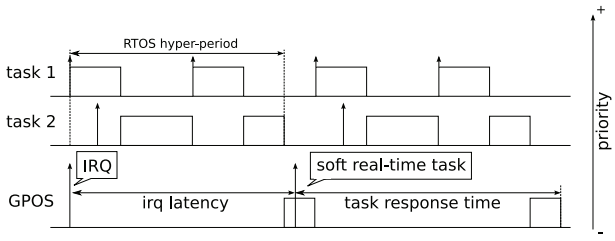


Figure 6: GPOS as idle task

assigned to the RTOS and can not be disabled by the GPOS—are able to preempt the execution of the GPOS at any instant. Once the processing of the FIQ handler finishes, the RTOS scheduler resumes the task that was executing (in this example the task representing the GPOS) as shown in Fig. 5 (a). Furthermore, RTOS tasks that require short latency can be scheduled at a priority higher than the whole cyclic schedule as shown in Fig. 5 (b). Another advantage of the presented architecture is that the period and length of the time slice associated with the GPOS can be easily modified at run-time with standard application-level function calls. Furthermore, idle times inside the time slots associated to the RTOS can be easily used to execute the GPOS by creating a new task representing the GPOS at a lower priority.

4.4.3 Priority based scheduling

In many other VMMs oriented to real-time systems, the GPOS is only executed when the RTOS becomes idle [13] [24]. This method allows the RTOS to take precedence over the GPOS and thus maintain its determinism. In the presented architecture, this is easy to achieve by configuring the task representing the GPOS processing as the lowest priority task of the RTOS. However, Fig. 6 highlights the problem of using the lowest priority thread to schedule the GPOS. In the figure, an IRQ request in the GPOS is delayed until all the processing at the RTOS is finished. In the worst case, the IRQ request will be attended only after a hyper-period in the RTOS schedule. The latency that can be achieved with this method may not be enough for certain devices. The same situation can happen for the GPOS soft real-time tasks which require a certain quality of service to work correctly.

In order to solve this problem without modifying the

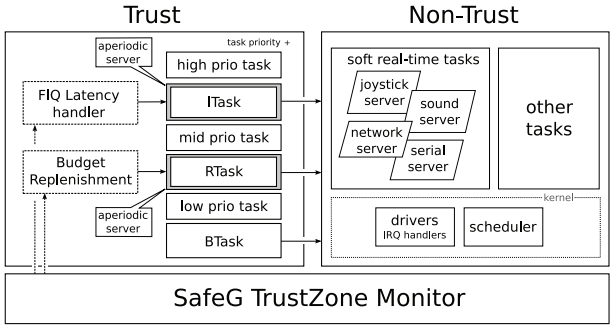


Figure 7: Integrated scheduling architecture

GPOS core (to satisfy requirement (g)) the architecture depicted in Fig. 7 has been implemented. The execution of the GPOS is now handled through several RTOS tasks. In addition to the lowest priority task *BTASK* (Background Task), the *ITASK* (Interrupt Task) is used to enhance the latency of GPOS interrupts, and the *RTASK* (Real-Time Task) is used to improve the quality of service of GPOS soft real-time tasks. The body of the three tasks is the same. The main difference between them is the priority level at which they execute and the way in which they are activated.

The RTASK task is executed at a middle configurable priority, between the BTASK background priority and the priority of the ITASK task. It is necessary that RTOS tasks with lower priority do not suffer starvation or deadline misses. In order to achieve this time isolation (as specified in requirement (c)) the RTASK task runs under the control of an aperiodic server with a configurable period and budget. When the RTASK consumes all the capacity allocated for it, it is suspended. Once it receives new capacity through a budget replenishment it is resumed again. The period and budget of the aperiodic server can be configured to provide a certain quality of service to the GPOS soft real-time tasks. For example, it makes it possible to guarantee that the GPOS will receive an amount of processing time which is equal to its budget for every server period.

The ITASK task also runs under the control of an aperiodic server but at a higher priority and with a different activation scheme. GPOS interrupts that require short latency are configured temporarily as FIQ interrupt sources and processed by a special handler, called FIQ Latency handler, at the RTOS. Fig. 8 depicts the timeline of an ITASK task activation. Here SafeG processing overhead is included for completeness. When a GPOS interrupt occurs the Latency handler activates the ITASK task and forwards further interrupt requests to the Non-Trust side, as IRQs, by configuring the TrustZone Interrupt Controller. When the ITASK task is scheduled (i.e., when it is the active task with the highest priority) a switch to the Non-Trust state is performed through SafeG, and the GPOS interrupt handler executes. This method makes it possible to represent certain GPOS interrupts through a task with high priority, and so their latency can be enhanced. In the presented architecture, modifications to the GPOS core are avoided (requirement (g)), and therefore the budget of the ITASK task is completely consumed each time a Non-Trust interrupt is

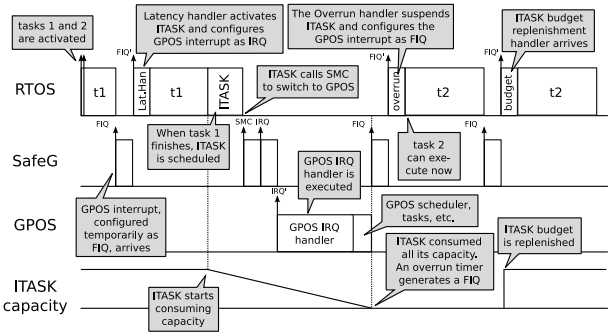


Figure 8: ITASK activation timeline (priority: t1 > ITASK > t2)

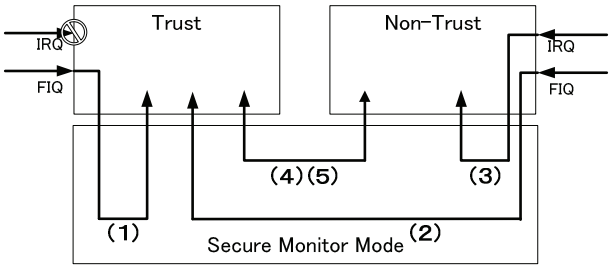


Figure 9: SafeG execution paths

processed. More refined methods that preserve the budget between interrupt instances can be implemented. However, they may require small modifications to the GPOS core in order to return the control back to the RTOS, once the interrupt is served, and therefore they are left for future research.

5. VMM implementation

The implementation of the presented architecture includes the SafeG TrustZone Monitor, the RTOS and the GPOS. TOPPERS/ASP was chosen as the RTOS and Linux as the GPOS.

5.1. SafeG TrustZone Monitor

SafeG is the implementation of the TrustZone Monitor for the presented architecture. It runs in Secure Monitor mode with interrupts disabled. Fig 9 shows the five possible execution paths in SafeG that are necessary to provide the means to switch between both operating systems and handle interrupts. As described in Sec. 4.2, interrupts are captured by SafeG before being forwarded to the appropriate operating system. In order to satisfy requirement (g), it is necessary that the GPOS interrupt entry is not modified. For that reason, SafeG adjusts the value of the registers to appear as if the interrupt had occurred directly, before loading the program counter with the GPOS interrupt vector address. The same procedure is used for RTOS interrupts.

The execution path (1) in Fig 9 shows the path when a FIQ occurs while the RTOS is executing in Trust state. The execution flow is suspended and the SafeG FIQ handler is executed. SafeG FIQ handler performs the necessary processing and jumps to the RTOS interrupt handler.

The execution path (2) shows the path when an FIQ occurs while the GPOS is executing in Non-Trust state. The execution flow is suspended and SafeG FIQ handler performs a context switch. More in detail, 38 GPOS general purpose 32-bit registers are saved in Trust memory and replaced by the corresponding 38 RTOS general purpose registers, the context. After that, the RTOS interrupt handler is called in a similar way as in path (1).

The execution path (3) shows the path when an IRQ occurs while the GPOS is executing in Non-Trust state. The execution flow is suspended and SafeG IRQ handler is invoked. SafeG IRQ handler performs the necessary processing (e.g., interrupt frequency monitoring) and jumps to the GPOS IRQ vector handler.

The execution paths (4) and (5) show the paths when a Secure Monitor Call (SMC) is executed by one of the operating systems. Registers are used to indicate the request type (i.e., switch, message or reboot) and pass arguments when required. When a *switch* request is performed, SafeG saves the context of the calling operating system and loads the context of the other operating system. *Message* requests can be used for device sharing as mentioned in Sec. 4.3, or for other types of intercommunications using Non-Trust memory, to place the associated data. In addition, SafeG can be configured to force an artificial exception in the RTOS, when a message is received, for synchronization purposes.

5.2. RTOS porting and extensions

The TOPPERS project [11] follows the ITRON standard for real-time operating systems to produce high quality open-source software for embedded systems. ASP (Advanced Standard Profile kernel) is one of TOPPERS real-time kernels and is based on the μITRON4.0 [7] specification with several extensions.

For the purpose of this study, ASP has been ported to an ARM TrustZone-enabled platform (PB1176JZF) to run concurrently with a GPOS (Linux). The porting allows execution in Trust and Non-Trust states as two different target platforms. The porting includes support for:

- Trustzone Interrupt Controller [2].
- Non-Trust Generic Interrupt Controller.
- TrustZone-enabled Memory Management Units.
- FIQ and IRQ interrupt routines with nesting.
- Exception handlers (i.e., data abort, SWI, undefined).
- Dispatcher and context switch code.

In order to schedule the GPOS mentioned in Sec. 4.4.1, an RTOS task whose body is a loop executing the function shown in Table 1 is used. The function consists of an SMC instruction plus the address of a buffer. A `while` instruction is used because when the GPOS execution is preempted by an RTOS FIQ and then the task is resumed, the SMC instruction must be executed again. An approach [13] such as modifying the link register value to point to the SMC instruction could be an alternative. However, using the mechanism shown in Table 1, the GPOS is also able to pass requests to the RTOS which can be used as an intercommunication mechanism or to share devices as mentioned in Sec. 4.3. To implement the cyclic scheduling approach pro-

Table 1: ITASK, RTASK and BTASK function body

```
ret_args->arg0 = NULL;
while (1) {
    Asm("mov r0, %0\n\t"
        "smc %1\n\t"
        :: "r" (ret_args),
           "I" (NT_SMC_SWITCH)
        : "r0", "memory");
    if (ret_args->arg0 != NULL)
        return;
}
```

posed in Sec. 4.4.2, μITRON4.0 cyclic handlers are used to suspend (`sus_tsk`), resume (`rsm_tsk`) or change the priority (`chg_pri`) of the GPOS processing. This is done at application level, and thus the internals of the RTOS do not need to be modified.

To implement a priority-based integrated scheduling approach like that described in Sec. 4.4.3, the ASP kernel has been extended to support execution-time overrun handlers. This mechanism makes it possible to limit the processor time of a specific task, and it has been used to implement deferrable servers [12] at application level without modifying the ASP scheduler. Cyclic handlers are used for the budget replenishments.

5.3. GPOS modifications

In the presented VMM architecture the following items may require modifications to the GPOS. No modifications to the core (i.e., scheduler or interrupt handlers) of the GPOS kernel were required.

5.3.1 Vector table

In the ARM processor, it is possible to select a normal vector table which starts at 0x0 or a high vector table starting at 0xFFFF0000. SafeG uses a normal vector table. If the GPOS uses the normal vector table, the GPOS vector table must be stored at a position different to 0x0, and the base address must be written to the vector base address register. In the evaluation environment, Linux was configured to use the high vector table to prevent conflicts with SafeG vector table, and thus no changes are required.

5.3.2 Memory and devices allocation

Memory and devices must be allocated either to Trust or Non-Trust space. To achieve this, it is necessary to modify the GPOS hardware configuration file for a specific platform. If a single device wants to be shared by the RTOS and the GPOS, the application libraries need to be modified to make a request to the RTOS through SafeG.

6. Evaluation

This section shows the evaluation of the presented architecture on a real machine. The evaluation platform is a PB1176JZF-S board, equipped with a TrustZone enabled ARM1176jzf processor. The core clock frequency is 210Mhz and it has 32KB data and instruction caches. TOPPERS/ASP version 1.3.1 was used as the RTOS. For the GPOS, an ARM Linux 2.6.24 kernel was used together with a CRAMFS file-system placed into FLASH memory. Since the GPOS can affect the execution time of the RTOS or SafeG through the cache, the whole cache is flushed before taking each measurement.

6.1. SafeG overhead

In this section, requirement (h) is evaluated. Although the presented architecture does not introduce overhead in memory accesses or normal device operation, SafeG introduces a bounded overhead in interrupt processing and TrustZone state switching.

Table 2: SafeG execution time

Path	WCET
(1) While RTOS runs FIQ occurs	0.7μs
(2) While GPOS runs FIQ occurs	1.6μs
(3) While GPOS runs IRQ occurs	1.2μs
(4) Switch from RTOS to GPOS	1.5μs
(5) Switch from GPOS to RTOS	1.7μs
From ASP IRQ vector until IRQs enabled	5.1μs

Measurements for each of the execution paths of SafeG are shown in Table 2. Since execution paths (2), (4) and (5) require TrustZone state switching (save and restore context), their execution time is a little higher than paths (1) and (3). As SafeG makes no assumptions about the RTOS that is running, all the processor registers (38 in this case) need to be saved and restored. It is possible to reduce the overhead further by saving only the necessary registers for a specific configuration. For example, in the presented architecture ASP makes switch requests only under task context, and therefore only registers used by the task need to be saved. This optimization reduces the execution time of paths (2), (4) and (5) by 240ns.

The last row in Table 2 indicates the interrupt latency —which is measured as the time between the ASP interrupt vector handler starts until interrupts are enabled again, just before the interrupt service routine is executed—. This processing takes place inside the core of ASP, is written in assembly with no loops, and consists of several steps: save processor context; nested interrupts handling code; judge the interrupt cause; set the interrupt priority; obtain the address of the interrupt handler; and enable interrupts. After that, the user interrupt service routine is called. The execution time of each path is always smaller than ASP interrupt latency, thus complying with requirement (h).

6.2. SafeG code verifiability

To satisfy requirement (i), SafeG code must be small and have as few forks as possible so that source code review becomes easier. The size of the code and data sections of SafeG, ASP and Linux are shown in Table 3. SafeG occupies a total of 1968 bytes, around 1/60 of the size of ASP, which is small enough for verification purposes. A total of 304 bytes in the `.bss` section of SafeG corresponds to the area where the context of each operating system is stored.

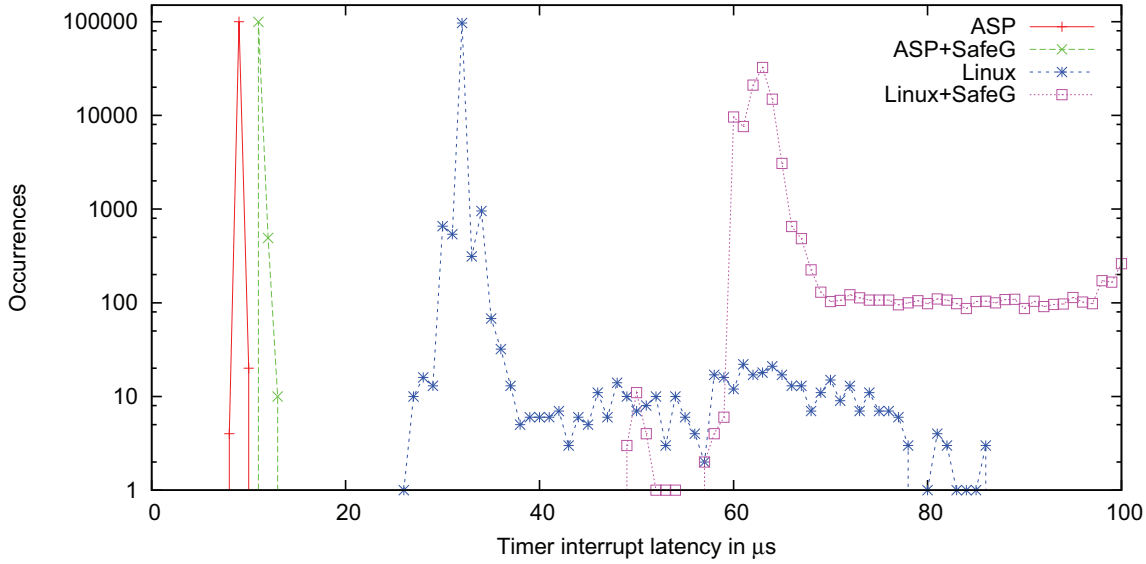


Figure 10: Timer interrupt latency

Table 3: Code and data size (in bytes)

	text	data	bss	total
SafeG	1520	0	448	1968
ASP	34796	0	83140	117936
Linux	1092652	148336	89308	1330296

Inside SafeG there are four forks in total. Three of them occur when SafeG captures an interrupt (IRQ/FIQ) or receives an SMC request. They are required to execute different processing depending on the current TrustZone state. The other fork is required to support instances when a high vector is used, as explained in Sec. 5.3.1. Because there are four forks inside SafeG, only eight types of tests are necessary in order to cover all possible execution paths.

Finally, another factor that helps to satisfy requirement (i) is the fact that SafeG runs with interrupts disabled. If interrupts were enabled, it would be necessary to verify accurately at which point interrupts enter while executing SafeG.

6.3. RTOS time isolation

To confirm that RTOS time isolation from the GPOS is satisfied (requirement (c)), the latency of the RTOS timer interrupt is measured before and after the introduction of SafeG. In the RTOS only a periodic operation to renew the system tick is executed every 1ms. The ASP system tick timer is replaced by the ARM performance monitor, whose value is read at the beginning of the timer interrupt service routine, and which contains the cycles corresponding to the interrupt handling code inside ASP. At the GPOS side, a terminal, Xeyes, Xclock and the top command are executed on top of X windows. For completeness, the latency of Linux timer interrupts were also measured.

All measurements were repeated for 10,000 times. The results are shown in Fig. 10 where the vertical axis shows the frequency, in logarithmic scale, and the horizontal axis shows the timer interrupt latencies. As shown in the graph, the introduction of SafeG increased the latency of ASP interrupts by around 2us. This increase is consistent with the values measured for SafeG overhead in Table 2. The worst case occurs when an FIQ interrupt arrives just after SafeG starts switching to the GPOS (with interrupts disabled). Despite the interrupt latency of the RTOS is being increased, it remains bounded and isolated from the GPOS operation, and therefore requirement (c) is satisfied.

The most common measured value of the GPOS timer interrupt latency was incremented by around 31μs due to SafeG overhead plus the preemptions caused by the periodic processing of ASP system tick. The worst measured value, 7637μs, does not appear in the graph and depends on long critical sections executed with interrupts disabled inside the Linux kernel. This problem is not caused by SafeG and can be addressed with different patches available for the Linux kernel.

6.4. Integrated scheduling evaluation

This section evaluates whether requirement (d) is satisfied or not with the approach proposed in Sec. 4.4.3.

6.4.1 Experiment no. 1: ITASK

In this experiment, the ITASK task shown in Fig. 7 has been assigned to the Non-Trust serial driver interrupt number. The input has been generated randomly by a user who typed characters on the serial console of the Non-Trust operating system at human speed. The *serial driver latency* is measured from the beginning of the FIQ Latency handler until the character is received by the user of the serial driver. Therefore it includes the overhead caused by the architecture, plus the possible blocking times of the RTOS tasks. The RTOS executes two tasks with the parameters shown in Table 4.

Table 5 shows the measured values for three situations.

Table 4: RTOS task parameters

task	priority	period	duration	utilization
1	high	50ms	10ms	20%
2	low	300ms	100ms	33%

The first row shows the results when the RTOS tasks from Table 4 are not present. The second row shows the results when using the idle approach. The maximum measured value, 113.9ms, includes the blocking caused by tasks 1 and 2. The last row shows the results when an ITASK task is used at a priority between the two RTOS tasks and with a server period of 30ms and budget of 2ms. The use of the ITASK mechanism reduced the measured latency of the serial port driver to 30.3ms.

Table 5: Serial driver latency (in μs)

approach	min	avg	max
alone	15.7	15.81	19.47
idle	14.6	22681	113833
itask	15.45	2292	30275

Configuring the server with 10ms period gave a maximum latency of 10.3ms caused by the blocking from task 1. However, it is worth mentioning that the shorter the period, the greater the overhead on the RTOS tasks. Non-Trust interrupts assigned to the ITASK task are temporarily handled as FIQs in order to notify the existence of a pending interrupt in the Non-Trust side. A notification takes similar time as a normal interrupt and it is bounded by the server associated to the ITASK task.

6.4.2 Experiment no. 2: RTASK

In order to evaluate the effectiveness of the RTASK task for improving the quality of service of soft real-time tasks in the GPOS, the *cyclictest* program [10], has been used to measure the response time of the Linux kernel, which was configured with high resolution timers. This test contains a periodic thread that measures the difference between the instant when a wake up call is scheduled, and then that wake up actually occurs. This measurement includes the interrupt latency, the interrupt service routine duration and the Linux scheduler latency. The program was cross-compiled using Buildroot [6], which was also used to generate a minimal root file-system based on the uClibc C library. It was run for 60 seconds at priority 80, with a single 10ms periodic thread using `clock_nanosleep`. In the background, Linux was loaded to use 100% of the CPU.

The test was repeated in three different configurations. Fig. 11 shows the results obtained when Linux is executed alone. Fig. 12 shows the results obtained when the GPOS is executed as the idle task of the RTOS, where they are also compared to those obtained when the GPOS is executed using the architecture described in Sec. 4.4.3. The RTASK task is scheduled through a deferrable server, with period 30ms, budget 5ms and a priority level between the two RTOS tasks. Vertical lines in Fig. 11 indicate the minimum, average and maximum measured values.

As shown in Fig. 11, Linux is capable of obtaining short

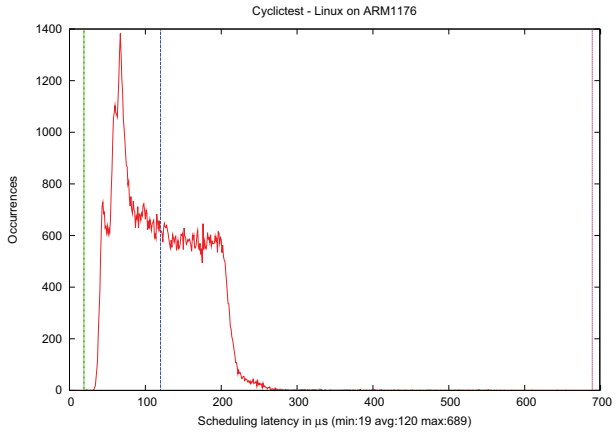


Figure 11: Cyclictest - Linux alone

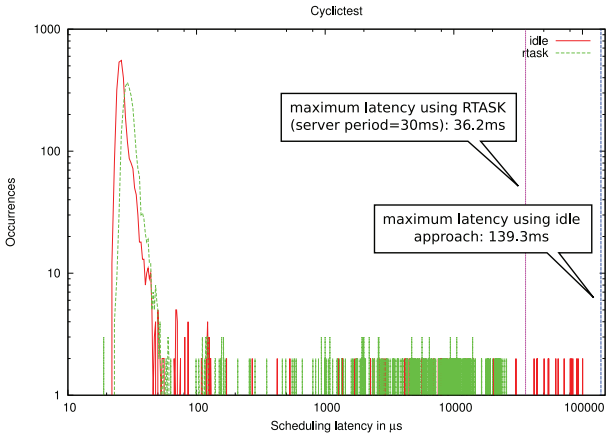


Figure 12: Idle vs. RTASK approach

latencies, which is in part thanks to the support for high resolution timers. The worst case measured value in that case was 689μs. However, when Linux is executed concurrently with the RTOS and scheduled as the idle thread (Fig. 12) the cyclictest results in a measured maximum latency of 139,3ms. This latency can be explained by the blocking caused by the RTOS task 2 (100ms), plus two or three instances of the RTOS task 1 (10ms each). As Table 6 shows, the maximum measured latency in experiments, which occurs when using an RTASK task with a deferrable server of period 30ms, was reduced to 36,2ms. As the budget and period of the deferrable servers can be configured, the latency of the GPOS soft real-time tasks and interrupts can be adjusted in an integrated scheduling fashion that satisfies requirement (d).

7. Conclusions

In this paper a dual operating system virtualization architecture based on ARM TrustZone® was explained and evaluated. Although the natural goal of TrustZone is security, its functionality has been applied to build a VMM architecture that can satisfy the initial requirements shown in Sec. 3. A similar configuration, with some slight differences, was used by [13] but it does not address require-

Table 6: Linux kernel latency (in μs)

approach	min	avg	max
alone	19	120	689
idle	22	30095	139265
rtask	19	5470	36108

ments (d), (e) and (f). In addition, no RTOS was ported to the TrustZone platform in that work, and so requirements (a) and (h) were not fully satisfied. The presented approach satisfies all requirements (a)-(i) and has been proven to be appropriate for enhancing the responsiveness of the GPOS, while preserving the determinism of the RTOS, without requiring modifications to the GPOS core.

One possible improvement to increase the usefulness of TrustZone, as a virtualization hardware, could be a mechanism to separate the control of the Trust/Non-Trust caches. Currently, caches are not separated and therefore the Non-Trust side can affect the performance of the Trust side by flushing the cache. Also, when switching from Non-Trust to Trust state there are 38 registers which are not banked, and which need to be saved and restored continuously. The introduction of an instruction for storing and restoring these registers could improve the performance.

As future research, a more refined method for integrating scheduling with small modifications to the GPOS kernel core will be investigated. Device sharing and intercommunications mechanisms using standard interfaces will also be explored. Finally, a porting to new multi-core TrustZone-enabled embedded processors is planned for the immediate future.

References

[1] Airlines Electronic Engineering Committee, 2551 Riva Road, Annapolis, Maryland 21401-7435. Avionics Application Software Standard Interface (ARINC-653), March 1996.

[2] AMBA3 TrustZone Interrupt Controller (SP890) Technical Overview, DTO 0013B, ARM Ltd., 2008.

[3] AMBA3 TrustZone Protection Controller (BP147) Technical Overview, DTO 0015A, ARM Ltd., 2004.

[4] ARM Security Technology. Building a Secure System using TrustZone Technology, PRD29-GENC-009492C, ARM Ltd., 2009.

[5] ARM1176JZF-S. Technical Reference Manual, DDI 0301G, ARM Ltd., 2008.

[6] Buildroot. `http://buildroot.uclibc.org/`.

[7] H. Takada and K. Sakamura, "μITRON for small-scale embedded systems", *IEEE Micro*, vol. 15, pp. 46-54, Dec. 1995.

[8] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park and C. Kim, "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones", In *Proceedings of the 5th Annual IEEE Consumer Communications & Networking Conference*, USA, January 2008.

[9] M. Masmano, I. Ripoll, A. Crespo, and J.J. Metge. "XtratuM: a Hypervisor for Safety Critical Embedded Systems", 11th Real-Time Linux Workshop. Dresden. Germany 2009.

[10] T. Gleixner cyclictest. `https://rt.wiki.kernel.org/index.php/Cyclictest`.

[11] TOPPERS: Toyohashi OPen Platform for Embedded Real-Time Systems. `http://www.toppers.jp`.

[12] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *20th IEEE Real-Time Systems Symposium*, Phoenix. USA, Dec 1999.

[13] I. Cereia, M. Bertolotti. Asymmetric virtualisation for real-time systems. In *ISIE 2008*, pages 1680 – 1685, Cambridge, 2008.

[14] G. Heiser. The role of virtualization in embedded systems. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16, Glasgow, UK, Apr 2008. ACM SIGOPS.

[15] G. Heiser. Hypervisors for consumer electronics. In *CCNC'09: Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, pages 614–618, Piscataway, NJ, USA, 2009. IEEE Press.

[16] A. Hergenhan and G. Heiser. Operating systems technology for converged ECUs. In *6th Embedded Security in Cars Conference (escar)*, Hamburg, Germany, Nov 2008. ISITS.

[17] Y. Kinebuchi, H. Koshimae, S. Oikawa, and T. Nakajima. Virtualization techniques for embedded systems. In *Proceedings of the Work-in-Progress Session: the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, 2006.

[18] Y. Kinebuchi, M. Sugaya, S. Oikawa, and T. Nakajima. Task grain scheduling for hypervisor-based embedded system. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 190–197, Washington, DC, USA, 2008. IEEE Computer Society.

[19] B. Leslie, N. FitzRoy-Dale, and G. Heiser. Encapsulated user-level device drivers in the Mungi operating system. In *Proceedings of the Workshop on Object Systems and Software Architectures 2004*, Victor Harbor, South Australia, Australia, Jan 2004. `http://www.cs.adelaide.edu.au/~wossa2004/HTML/`.

[20] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[21] M. Masmano, I. Ripoll, and A. Crespo. An overview of the XtratuM nanokernel. In *1st Intl. Workshop on Operating Systems Platforms for Embedded Real-Time applications. OSPERT 2005*, Palma de Mallorca, Spain, Jul 2005.

[22] H. Takada, S. Iiyama, T. Kindaichi, and S. Hachiya. Linux on ITRON: A Hybrid Operating System Architecture for Embedded Systems. In *SAINT-W '02: Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops*, pages 4–7, Washington, DC, USA, 2002. IEEE Computer Society.

[23] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing Embedded Security on Dual-Virtual-CPU Systems. *IEEE Des. Test*, 24(6):582–591, 2007.

[24] S. Yoo, Y. Liu, C.-H. Hong, C. Yoo, and Y. Zhang. Mobivmm: a virtual machine monitor for mobile phones. In *MobiVirt '08: Proceedings of the First Workshop on Virtualization in Mobile Computing*, pages 1–5, New York, NY, USA, 2008. ACM.

Timeslice Donation in Component-Based Systems

Udo Steinberg
Technische Universität Dresden
udo@hypervisor.org

Alexander Böttcher
Technische Universität Dresden
boettcher@tudos.org

Bernhard Kauer
Technische Universität Dresden
bk@vmmon.org

***Abstract*—An operating system that uses a priority-based scheduling algorithm must deal with the priority inversion problem, which may manifest itself when different components access shared resources. One solution that avoids priority inversion is to inherit the priority across component interactions. In this paper we present our implementation of a timeslice donation mechanism that implements priority and bandwidth inheritance in the NOVA microhypervisor. We describe an algorithm for tracking dependencies between threads with minimal runtime overhead. Our algorithm does not limit the preemptibility of the kernel, supports blocked resource holders, and facilitates the abortion of inheritance relationships from remote processors.**

I. Introduction

Priority inversion [1] occurs when a high-priority thread *H* is blocked by a lower-priority thread *L* holding a shared resource *R* as illustrated in Figure 1. Priority inversion can be unbounded if a medium-priority thread *M* prevents the low-priority thread from running and thus from releasing the resource. A lock that protects a critical section from concurrent access is a typical example for a shared resource that can cause priority inversion. In component-based systems the shared resource may also be a server thread that is contacted by multiple clients.

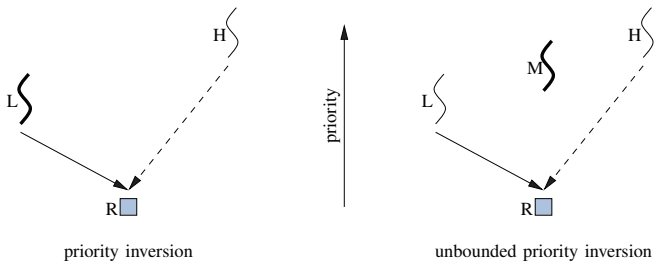


Figure 1. Example of priority inversion: The currently active thread is marked bold.

Resource Access Protocols

Several solutions for circumventing the priority inversion problem have been proposed. They range from disabling preemption to using complex protocols to control resource access. Disabling preemption while holding a shared resource is prohibitive in systems with real-time or low-latency requirements. Protocols such as the priority ceiling protocol (PCP) and the priority inheritance protocol (PIP) [2] avoid priority inversion by defining rules for resource allocation and priority adjustment that guarantee forward progress for threads holding shared resources.

The priority ceiling protocol prevents deadlocks that arise from contention on shared resources. However, PCP requires a priori knowledge about all threads in the system. At construction time every shared resource is assigned a static ceiling priority, which is computed as the maximum of the priorities of all threads that will ever acquire the resource. Priority ceiling is therefore unsuitable for open systems [3] where threads are created and destroyed dynamically or where the resource access pattern of threads is not known in advance. Because priority ceiling relies on static priorities it is not applicable to scheduling algorithms with dynamic priorities, such as earliest deadline first (EDF) [4].

When using the priority inheritance protocol, the priority of a thread that holds a shared resource is temporarily boosted to the maximum of the priorities of all threads that are currently trying to acquire the resource. Priority inheritance works in systems with dynamic priorities and does not require any prior knowledge about the interaction between threads and resources.

Bandwidth inheritance (BWI) [5] can be considered an extension of the priority inheritance protocol to resource reservations. Instead of inheriting just the priority, the holder of a shared resource inherits the entire reservation of each thread that attempts to acquire the resource. Bandwidth inheritance reduces the blocking time for other threads when the resource holder's own reservation is depleted.

Resource reservations in our system are called timeslices and consist of a time quantum coupled with a priority. Timeslices with a higher priority have precedence over those with a lower priority. The time quantum facilitates round-robin scheduling among timeslices with the same priority.

In this paper, we describe and evaluate the timeslice donation mechanism of the NOVA microhypervisor [6]. This mechanism allows for an efficient implementation of priority and bandwidth inheritance in an open system with many threads. We discuss issues that arise when threads block or unblock while holding shared resources and explore how blocking dependencies can be tracked with minimal overhead.

II. Background

Component-based operating systems achieve additional fault isolation by running device drivers and system services in different address spaces. Communication between these components must use inter-process communication (IPC) instead of direct function calls in order to cross address-space boundaries. When multiple clients contact the same server, threads in the server are a shared resource and therefore prone to cause priority inversion.

Lazy scheduling was originally introduced as a performance optimization in the L4 microkernel family to bypass the scheduler during inter-process communication [7]. Figure 2 illustrates the communication between a client and a server thread. Because threads and timeslices are separate kernel objects, the kernel can switch them independently. During IPC, the kernel changes the current thread from the client C to the server S and back, without changing the current timeslice. The effect is that the client donates its timeslice to the server.

Figure 2. Synchronous communication between a client and a server in a component-based system. Left side: During the request the client thread C donates its timeslice c to the server thread S. Right side: When the server responds, the kernel returns the previously donated timeslice c back to the client.

Timeslice donation can be used to implement priority inheritance, but only if the kernel correctly resumes the donation after a preemption. For this purpose the kernel must track dependencies between threads so that it can determine the thread to which a timeslice has been most recently donated. Most versions of L4 do not implement dependency tracking. Therefore, priority inversion may occur when a server thread is preempted and afterwards uses its own, potentially low-priority, timeslice. This problem is described in more detail in [8].

Timeslice Donation and Helping

The NOVA microhypervisor implements priority and bandwidth inheritance using the following two closely related mechanisms:

Donation: In the left example of Figure 3, a high-priority client thread C sends an IPC to a low-priority server thread S. By donating the client's timeslice c to the server, the priority of S is boosted to that of C and the kernel can directly switch from the client to the server without having to check for the existence of ready threads with priorities between the client and the server, such as the medium-priority thread T. Without timeslice donation, S would use its own low-priority timeslice s and T would be able to preempt S, thereby causing priority inversion for C. During IPC, the kernel establishes an explicit donation dependency from C to S, which we denote by a solid arrow. When the scheduler selects the client's timeslice c, it follows the donation dependency and activates S instead of C, thereby resuming the donation.

Helping: Donation boosts the priority of a server to that of its current client and ensures that, for as long as the server works on behalf of the client, it can only be preempted by threads with a higher priority than the client. Helping augments donation by boosting the priority of the server even further when higher-priority clients try to rendezvous with the server while it is busy handling a request. In the right example of Figure 3, the server thread S is handling the request of a client thread C and initially uses the client's timeslice c. Another thread H with a higher priority than C can preempt the server and attempt to rendezvous with S. Because the rendezvous fails, H switches directly to S in order to help S finish its current request, thereby elevating the priority of S to that of H. Unlike donation, the kernel does not establish an explicit dependency from H to S. Upon selecting the timeslice h, the scheduler activates H, which simply retries its operation. We denote such an implicit helping dependency by a dashed arrow.

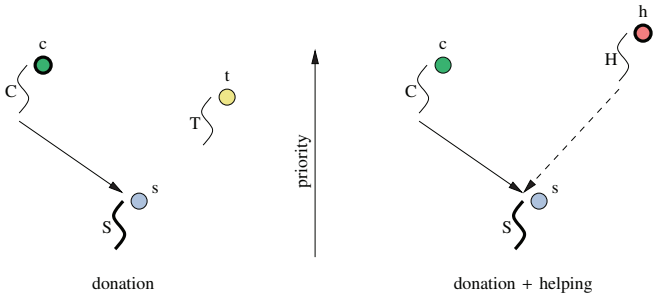


Figure 3. Example of timeslice donation and helping during client-server communication. The currently active thread and timeslice are marked bold.

Threads in a realtime system typically obtain only a limited time quantum in each period of execution. If a server exhausts the time quantum of its current client during the handling of a request, the server becomes stuck until the client's time quantum has been replenished. In such cases other clients cannot rendezvous with the server and therefore make use of the bandwidth inheritance property of the helping mechanism to allow the server to run the request to completion.

Similar issues arise when a client aborts its request before the server can reply, when the client is deleted, or when the communication channel between the client and the server is destroyed. Such cases leave the server in an inconsistent state that is similar to the state when the server is preempted, except that the old client will no longer provide the time quantum for the server to complete the request. Instead, subsequent clients use the helping mechanism to bring the server back into a consistent state where it can accept the

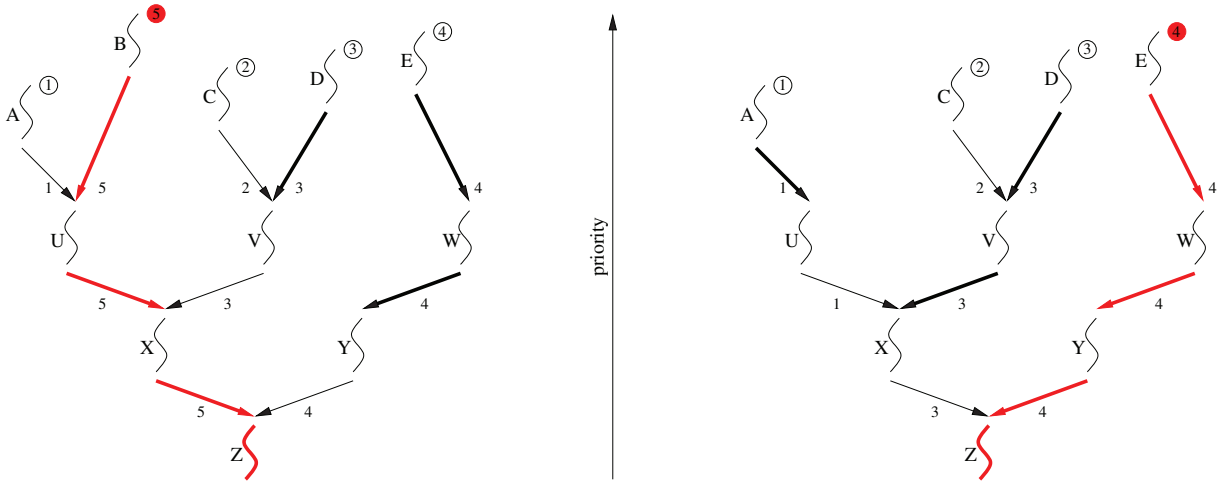


Figure 4. Dependency tracking: The highest-priority incoming edge of each node and the currently active thread and timeslice are marked bold. Changes to nodes in the priority inheritance tree may require updates along the path from the changed node to the root node. In this example the incoming edges of U, X, and Z must be updated when B leaves the priority inheritance tree.

next request. Because synchronous communication between threads on the same CPU always uses timeslice donation and helping, server threads that can only be contacted on their local CPU do not need a timeslice of their own.

The donation and helping mechanisms are transitive. If a server needs to contact another server to handle a client request, it further donates the current timeslice to the other server for the duration of the nested request. Therefore, the kernel must be able to handle large dependency tracking trees.

Multiprocessor Considerations

Helping and donation cannot be easily extended to multiprocessor systems and we are currently aware of only one proposal that describes a multiprocessor priority inheritance protocol [9].

One observation is that priorities of threads on different CPUs are not directly comparable. Additionally, the result of any comparison would quickly become outdated when other processors reschedule. Another observation is that a client cannot donate time from its CPU to help a server on another CPU. Such an operation would cause time to disappear on one processor and to reappear on another. Donating additional time to an already fully loaded CPU causes overload and can potentially break real-time guarantees.

The overload situation can be avoided if the client pulls a preempted server thread over to its CPU to help it locally. However, such an approach requires the address space of the server to be visible and identically configured on all processors on which clients for this server exist. In cases where client threads from different CPUs attempt to help the same server thread simultaneously, the kernel would need to employ a complex arbitration protocol among all helping client threads to ensure that each server thread executes on one processor only at a time. Furthermore, migrating the working set of the server thread to the CPU of the client and then back to the original CPU can result in a significant amount of coherence traffic on the interconnect.

Due to these drawbacks our algorithm does not include cross-processor helping. However, it supports IPC aborts from remote CPUs.

III. Related Work

In our previous work on capacity-reserve donation [10], we described an algorithm for computing the effective priority of a server as the maximum of the effective priorities of its current and all pending clients. The algorithm performs the tracking of dependencies and priorities by storing priority information inside the nodes and along the edges of a priority inheritance tree. For each node in the tree, the outgoing edge is marked with the maximum of the priority along all incoming edges of that node as shown in Figure 4.

Unfortunately, changes to nodes of the inheritance tree may require numerous updates to the edges of the tree as shown on the right side. When thread *B* leaves the priority inheritance tree (because it experiences an IPC timeout or is deleted), the kernel must recompute the priorities along the edges from the changed node down to the root node. In this example, the kernel must update the incoming edges of threads *U*, *X*, and *Z* to determine that the timeslice of thread *E* has become the highest-priority timeslice donated to *Z*. Depending on the nesting level of IPC, the number of updates to the priority inheritance tree can become very large, resulting in long-running kernel operations that must be executed atomically. Protecting the whole tree with a global lock for the duration of the update is undesirable because it disables preemption and limits the scalability of the algorithm in multiprocessor environments.

A more efficient version of the bandwidth inheritance protocol [11] has been implemented in the Linux kernel. It also uses a tree structure to track the dependencies between tasks and resources. When a task blocks on a shared resource, all tasks that previously inherited their bandwidth to that task must be updated to inherit their bandwidth to the holder of the shared resource instead.

OKL4 is a commercially deployed L4 microkernel, which is derived from L4Ka::Pistachio. OKL4 tracks IPC dependencies across preemptions and implements a priority inheritance algorithm. The kernel grabs a spinlock during updates to the inheritance tree in order to guarantee atomic updates.

With the realtime patch [12] series, support for priority inheritance was introduced to the Linux kernel. Because the realtime patch made the kernel more preemptible, the need arose to avoid unbounded priority inversion when threads are preempted while holding kernel locks [13]. Further research based on the Linux realtime patches, especially in the context of priority inheritance, is conducted by the KUSP [14] group. Their research focus is on supporting arbitrary scheduling semantics using group scheduling [15] in combination with priority inheritance.

IV. Implementation

Dependency tracking algorithms that store priority information along the edges of the priority inheritance tree share the problem that updates to a node in the tree require a branch of the tree to be updated atomically. For example, when a client with a high-priority timeslice joins or leaves an existing priority inheritance tree, it must rewrite the priority information along the edges from the client to the server at the root of the tree as shown in Figure 4. The update of the tree cannot be preempted because the scheduler must not see the tree in an inconsistent state. Therefore, the duration of the update process defines the preemptibility of the kernel. In an open system, a malicious user can create as many threads as his resources permit, arrange them in a long donation or helping chain and then cause an update in the priority inheritance tree that will disable preemption in the kernel for an extended period of time. Therefore, we devised a new dependency tracking algorithm that does not affect the kernel's preemptibility and at the same time keeps the dependency tracking overhead low. Before we describe this algorithm in detail, we present our requirements.

Requirements

To prevent malicious threads from being able to cause long scheduling delays in the kernel, we require updates in the priority inheritance tree to be preemptible. Furthermore, we demand that each operation is accounted to the thread that triggered it. Our goal is to move all time-consuming operations from the performance-critical paths in the kernel into functions that are called infrequently. For example, we strive to move as much dependency tracking as possible out of the IPC path into the scheduler and into functions that handle deletion of threads and communication aborts. The new dependency tracking algorithm works for an arbitrary number of threads and is not limited to small-scale systems or systems where all communication patterns must be known in advance.

Improved Algorithm for Dependency Tracking

Our new algorithm is based on the idea of storing no priority information whatsoever in nodes of the tree, which obviates the need for updating the priorities when threads join or leave the priority inheritance tree. Furthermore, priority information in the tree cannot become stale. However, this approach requires the kernel to restore the missing information during scheduling decisions, which works as follows:

When invoked, the scheduler selects the highest-priority timeslice from the runqueue and then follows the donation links to determine the path that the timeslice has taken prior to a previous preemption. When the scheduler finds a thread that has no outgoing donation link, it switches to that thread. In the left example of Figure 4, the scheduler selects the timeslice with priority 5, which belongs to thread B, and then follows the donation links from B via U and X to Z. Because Z has no outgoing edge, Z is dispatched. When thread B leaves the tree as shown in the right example of Figure 4, the scheduler selects the timeslice with priority 4, which belongs to thread E and then follows the donation links from E via W and Y to the server Z.

Traversing the donation links from a client's timeslice to the server at the root of the priority inheritance tree is a preemptible operation. If a higher-priority timeslice is added to the runqueue while the scheduler is traversing the tree, the kernel restarts the traversal, beginning with the higher-priority timeslice instead. The benefit of this algorithm is that whenever nodes in the inheritance tree are added or removed, no priority information must be updated. Algorithms that store priorities in all nodes of the tree can quickly determine the highest-priority timeslice donated to a thread by checking the highest-priority incoming edge of that thread. In contrast, our algorithm must compute this information by traversing the priority inheritance tree after a preemption. We quantify the cost for this operation in Section V.

Blocking

An interesting scenario occurs when the server thread at the root of a priority inheritance tree blocks. This can happen when the server waits for an interrupt that signals completion of I/O or when it waits for the reply from a cross-processor request for which timeslice donation cannot be used.

In the left example of Figure 5, a server thread Z blocks while using timeslice b. In that case the kernel removes b

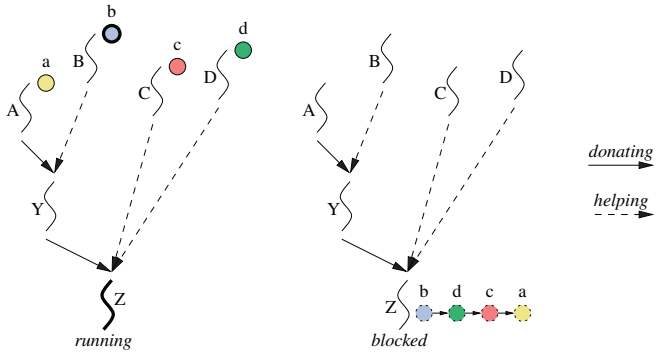


Figure 5. Blocking of threads when the holder of a shared resource is blocked.

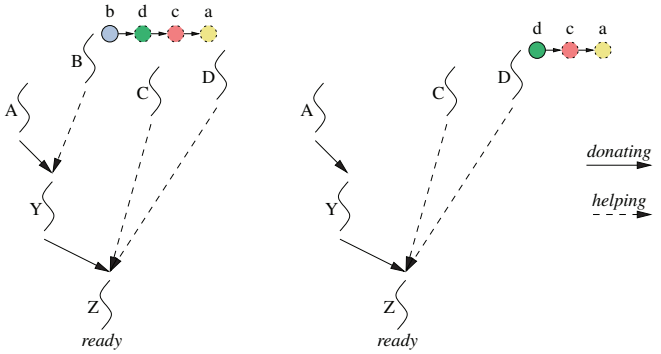


Figure 6. Staggered wakeup of threads when the holder of a shared resource unblocks.

from the runqueue and enqueues it in a priority-sorted queue of timeslices that are blocked on Z. During the subsequent reschedule operation, the scheduler selects timeslice *d* and traverses the priority inheritance tree down to *Z*. When it finds that *Z* is still blocked, *d* is also added to the queue of blocked timeslices. The right side of Figure 5 illustrates that all other timeslices that have been donated to *Z* are gradually removed from the runqueue and become blocked on *Z* when they are selected by the scheduler.

Staggered Wakeup

When *Z* eventually becomes unblocked, all timeslices that have previously been blocked on *Z* must be added back to the runqueue, effectively reversing the operation of blocking from Figure 5. Because an arbitrary number of timeslices can potentially be blocked at the root of a priority inheritance tree, releasing all of them at once contradicts our requirement of avoiding long scheduling delays. Based on the observation that only the highest-priority timeslice from the blocked queue will actually be selected by the scheduler, releasing the other timeslices can be deferred. The left side of Figure 6 illustrates that, when *Z* becomes unblocked, the kernel adds *b*, the highest-priority timeslice blocked on *Z*, back to the runqueue. The other timeslices that were blocked on *Z* remain linked to *b* and are not added to the runqueue yet. When *b* lowers its priority or is removed from the runqueue, the kernel adds *d*, the first timeslice linked to *b*, back to the runqueue and leaves the remaining timeslices linked to *d* as shown in the right of Figure 6. The benefit of this approach is that when Z unblocks, only a single timeslice needs to be added to the runqueue. The other blocked timeslices will be released in a staggered fashion.

Direct Switching

Recall from Figure 2 that the kernel implements timeslice donation by directly switching from one thread to another while leaving the current timeslice unchanged. When a server responds to its client, the kernel must check whether it can undo the timeslice donation by directly switching back to the client. Switching back to the client is wrong in cases where the server is currently using the timeslice of a high-priority helper and the client and the helper do not share the same incoming edge in the priority inheritance tree of the server. For example, when *Z* responds to *Y* in the left example of Figure 5, the kernel can only switch from *Z* to *Y* if *Z* is running on timeslice *a* or *b*. If *Z* is running on timeslice *c* or *d*, the kernel cannot return the timeslice to *Y*, because the timeslice was not donated to *Z* via *Y*. The kernel must instead switch to thread *C* or *D* so that they can retry their rendezvous with *Z*. Because our algorithm does not store any information along the edges of the priority inheritance tree, the kernel uses the following trick: When the scheduler selects a new timeslice and starts traversing the tree, the kernel counts the number of consecutive donation links along the path in a CPU-local *donation counter*. At the beginning of a new traversal and every time the kernel encounters a helping link, the donation counter is reset to zero. The donation counter indicates how often the kernel can directly switch from a server back to its client. When *Z* replies to *Y* in the left example of Figure 5, the current timeslice is *b* and the donation counter is 1, indicating that the kernel can directly switch from *Z* to *Y*, but not from *Y* to *A*. When a client donates the current timeslice to a server, the kernel increments the donation counter. When a server responds to its client, the kernel decrements the donation counter. The update of the donation counter is the *only* overhead added to the performance-critical IPC path by our dependency tracking algorithm.

Livelock Detection

Communication in component-based systems can lead to deadlock when multiple threads contact each other in a circular manner. In Figure 7, a client thread *C* contacts a server *Y*, which in turn contacts another server *Z*. Deadlock occurs when *Z* tries to contact *Y*. In our implementation, *Y* and *Z* would permanently try to help each other, thereby turning the deadlock into a livelock.

In NOVA, the kernel can easily detect such livelocks during the traversal of the priority inheritance tree by counting the number of consecutive helping links in a *helping counter*. When the value of the helping counter exceeds the number of threads in the system, the kernel can conclude that the current timeslice is involved in a livelock scenario. It can then remove the timeslice from the runqueue and print a diagnostic message.

Figure 7. Development of a Livelock

V. Evaluation

We evaluated the performance of our priority-inheritance implementation using several microbenchmarks, which we conducted on an Intel Core2 Duo CPU with 2.67 GHz clock frequency.

In contrast to dependency tracking algorithms that store priority information in each node of the inheritance tree, our algorithm keeps the priority information only in the timeslices bound to the client threads that form the leaves of the inheritance tree. Whenever the current timeslice changes, the scheduler must follow the dependency chain to find the server thread at the root of the inheritance tree to which the timeslice has been donated. Fortunately such tree traversals are neither very frequent nor very expensive.

Frequency of Dependency Tracking

The kernel invokes the scheduler to select a new current timeslice when the current thread suspends itself and thereby removes the current timeslice from the runqueue. The scheduler is also invoked when the kernel releases a previously blocked thread and that thread adds a timeslice with a higher priority than the current timeslice to the runqueue. It should be noted that the scheduler need not be invoked during client-server communication (see Figure 2) because the current timeslice remains the same and the runqueue need not be updated.

The frequency of scheduler invocations depends on the number of preemptions in the system, which in turn depends on the length of timeslices and the frequency of higher-priority threads being released.

Cost of Dependency Tracking

The costs of each traversal depends on the depth of the priority inheritance tree and on the type of dependency encountered during the traversal. Donation dependencies are explicitly tracked by the kernel, which stores the IPC partner in the thread control block. Therefore, following a donation dependency is a pointer chasing operation, which can lead to cache and TLB misses. In NOVA, all timeslices and threads are allocated from slab allocators and thus likely to be in close proximity. Furthermore, the kernel uses superpages for its memory region to reduce the number of TLB misses. The traversal of a donation dependency typically only causes a cache miss.

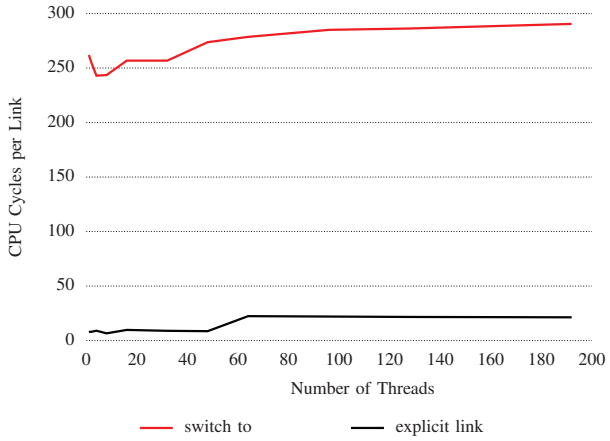


Figure 8. Overhead for traversing a helping link: The average cost is independent of the number of threads along the path. The graph compares an implementation in which the kernel simply switches to the destination thread with an implementation that reduces the overhead by tracking helping links explicitly.

A helping dependency indicates that a client thread did not manage to rendezvous with the server because the server was busy. In that case the client thread retries its rendezvous and thereby switches to the server thread. The thread switch is all that is required to traverse a helping dependency. The cost for the switch typically includes the cost for switching address spaces unless both threads happen to be in the same address space. The overhead can be reduced by tracking helping dependencies explicitly. There is a tradeoff between faster traversal of dependencies and having to store more information in the priority inheritance tree. We implemented and measured both variants. The CPU cycles required to traverse the priority inheritance tree are accounted to the newly selected current timeslice where the traversal started. Determining the thread at the root of the tree that will use the timeslice is part of the actual helping process.

Figure 8 shows that simply switching to the thread in order to help is much more expensive than following an explicit link. Furthermore, the costs of traversing a single helping link is nearly constant, irrespective of chain length. The step in the lower curve and the slight increase of 10–30 cycles in the upper curve can be attributed to additional cache usage when touching more threads.

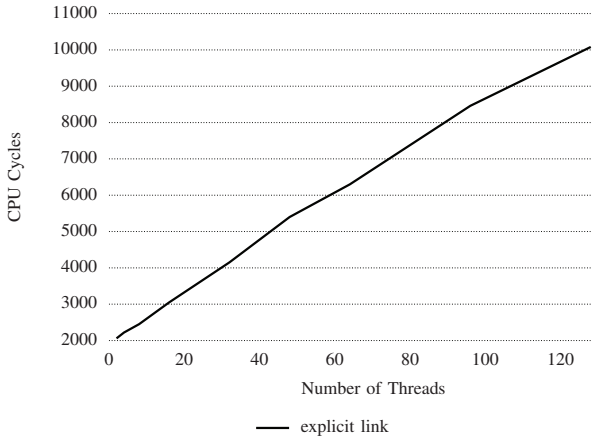


Figure 9. Cost for canceling an IPC: The cost of aborting an IPC operation scales linearly with the length of the path to the root node.

Cost of Modifying the Inheritance Tree

Updates to the inheritance tree are required when the link between two threads in the tree is broken. Possible reasons include thread deletion, abortion of an ongoing IPC between two threads, or revocation of the communication channel between the client and its server.

Breaking a link in the inheritance tree requires the deletion of the IPC connection between the affected threads and a traversal of the inheritance tree down to the leaf to check for blocked timeslices. If blocked timeslices are found, the kernel performs a staggered wakeup for them.

To avoid taking any locks in the IPC path, the IPC connection is deleted on the CPU on which the client, server, and the priority inheritance tree are located. In case the deletion was initiated by a thread on a different processor, the remote CPU must send an inter-processor interrupt to break the link. However, the costly part of the tree traversal down to the root is performed by the initiating thread on the remote CPU.

Figure 9 shows the cost for an inheritance tree update. We measured the implementation where helping and donation links are explicitly tracked in the kernel and the update was triggered from a remote processor. A thread running on one CPU is aborted by a remote thread running on another CPU, so that an additional cross-processor synchronization is included in the overhead. The overhead depends on the length of the path from the aborted thread to the root node. The number of cycles required for breaking a link increases linearly with the length of path in the inheritance tree. The absolute duration to update the inheritance tree is less than 5μs for a path length of up to 64 threads and less than 8μs for up to 512 threads. To date we have not observed calling depths of more than 16 threads in real-world scenarios.

VI. Conclusion

We have designed a novel mechanism that implements priority and bandwidth inheritance in a component-based system. Our algorithm does not limit the preemptibility of the kernel, and keeps the runtime cost on the performance-critical IPC path minimal. The algorithm supports threads that block while holding shared resources, and can detect livelocks. Our evaluation shows that the performance overhead of the dependency tracking scales linearly with the number of threads in a call chain.

Acknowledgements

We thank Jean Wolter and the anonymous reviewers for their comments on an earlier version of this paper.

References

[1] B. W. Lampson and D. D. Redell, “Experience with Processes and Monitors in Mesa,” *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, 1980.

[2] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[3] Z. Deng and J. W.-S. Liu, “Scheduling Real-time Applications in an Open Environment,” in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 1997, pp. 308–319.

[4] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[5] G. Lipari, G. Lamastra, and L. Abeni, “Task Synchronization in Reservation-Based Real-Time Systems,” *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1591–1601, 2004.

[6] U. Steinberg and B. Kauer, “NOVA: A Microhypervisor-Based Secure Virtualization Architecture,” in *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2010, pp. 209–222.

[7] J. Liedtke, “Improving IPC by Kernel Design,” in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1993, pp. 175–188.

[8] S. Ruocco, “Real-Time Programming and L4 Microkernels,” in *In Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006.

[9] M. Hohmuth, “Pragmatic Nonblocking Synchronization for Real-Time Systems,” Ph.D. dissertation, TU Dresden, Germany, 2002.

[10] U. Steinberg, J. Wolter, and H. Härtig, “Fast Component Interaction for Real-Time Systems,” in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, 2005, pp. 89–97.

[11] D. Faggioli, G. Lipari, and T. Cucinotta, "An Efficient Implementation of the Bandwidth Inheritance Protocol for Handling Hard and Soft Real-Time Applications in the Linux Kernel," in *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2008, pp. 1–10.

[12] Linux Community, "Linux Realtime Patches," 2010, April 2010. [Online]. Available: http://rt.wiki.kernel.org

[13] S. Rostedt, "RT-mutex Implementation Design," 2010, Document shipping with the Linux 2.6 kernel sources, file: [Documentation/rt-mutex-design.txt].

[14] D. Niehaus and group, "Proxy Execution in Group Scheduling," 2010, April 2010. [Online]. Available: http://www.ittc.ku.edu/kusp/kusp_docs/gs_internals_manual/index.html

[15] M. Friesbie, D. Niehaus, V. Subramonian, and C. Gill, "Group Scheduling in Systems Software," in *In Workshop on Parallel and Distributed Real-Time Systems*, 2004.

Full virtualization of real-time systems by temporal partitioning

Timo Kerstan
University of Paderborn
Fuerstenalle 11
33102 Paderborn
Germany
timo.kerstan@uni-paderborn.de

Daniel Baldin
University of Paderborn
Fuerstenalle 11
33102 Paderborn
Germany
dbaldin@uni-paderborn.de

Stefan Groesbrink
University of Paderborn
Fuerstenalle 11
33102 Paderborn
Germany
morenga@uni-paderborn.de

***Abstract*—Virtualization is a promising and challenging technology in the design of embedded systems. It is feasible for hard real-time tasks if the guest operating systems can be modified to enable the virtual machine monitor to access to required information of the guests. Since licensing restrictions may prohibit this, full virtualization may remain as the only solution. Thus communication between the guest OS scheduler and the scheduler of the virtual machine monitor is not possible, as the operating system cannot be modified. We will describe a scheduling model based on single time slot periodic partitions. This model is based on the idea of temporal partitioning. Each temporal partition executes a virtual machine, has a fixed length and is activated periodically. We show how to determine the global period length enclosing one instance of each temporal partition. Based on the global period we show how to determine the length of the temporal partitions inside this global period, to ensure the real-time constraints of each virtual machine.**

***Index Terms*—Real time systems**

I. Introduction

Virtualization has been a key technology in the desktop and server market for a fairly long time. Numerous products offer hardware virtualization at the bare metal level or at the host level. They enable system administrators to consolidate whole server farms and end users to use different operating systems concurrently. In the case of server consolidation, virtualization helps to improve the utilization or load balance which facilitates a reduction of costs and energy consumption. Distributed embedded systems used in automotive and aeronautical systems consist of multitudinous microcontrollers, each executing a dedicated task to guarantee isolation and to prevent a fault from spreading over the whole network. In addition, the utilization of a single microcontroller may be very low. Applying virtualization to distributed embedded systems can help to increase the scalabilty while preserving the required isolation, safety, and reliability. It is not possible to apply the virtualization solutions for server and desktop systems one-to-one to embedded systems. The inherent timing constraints of embedded systems preclude this. These timing constraints add temporal isolation as a requirement to virtualized embedded real-time systems. Especially the classical approach of the schedulability analysis[1] is no longer applicable to virtualized environments[2], such as the open system environments[3],

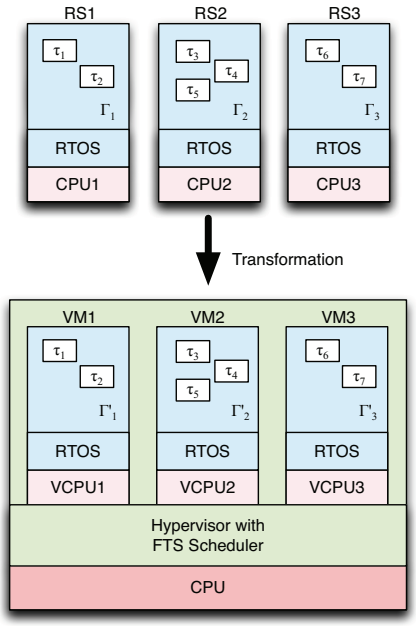


Fig. 1. Virtual real-time system using FTS as a global scheduling policy

[4]. An open system environment requires a modification of the OS level scheduler to communicate with the virtual machine monitor (VMM, Hypervisor), which may not be possible because of licensing restrictions. Instead, it is necessary to be able to schedule virtual machines and their real-time tasks using full virtualization at the VMM level. Figure 1 shows our approach in an abstract manner. We are using given tasksets Γ_i from a given real-time execution environment called RS_i and transform these tasksets to tasksets Γ'_i being schedulable in a virtual real-time system hosting the tasksets Γ'_i as virtual machines VM_i.

To be able to execute more than one virtual machine on a shared CPU, a global scheduling approach or a time partitioning of the CPU is necessary. As in this paper we will focus on full virtualization, we prefer the approach of time partitioning, because in that case it is not necessary to modify the local schedulers. Mok, Feng and Chen introduced

the resource partition model for real-time systems[2] to handle the schedulability analysis of a periodic taskset executed on a resource partition.

Definition A resource partition Π is a tuple (γ, P), where γ is an array of N time pairs $((S_1, E_1), ..., (S_N, E_N))$ that satisfies $0 \leq S_1 < E_1 < ... < S_N < E_N$ for some $N \geq 1$, and P is the partition period. The physical resource is available to the set of tasks executed on this partition only during intervals $(S_i + j \cdot P, E_i + j \cdot P), 1 \leq i \leq N, j \geq 0$.[2]

In a nutshell, a (temporal) partition is simply a collection of time intervals during which the physical resource is made available to a set of tasks which is scheduled on this partition. Mok, Feng and Chen introduced two types of periodic resource partitions:

- Single Time Slot Periodic Partitions (STSPP)
- Multiple Time Slot Periodic Partitions (MTSPP)

In case of a STSPP, there exists only one time interval ($N = 1$), while in case of a MTSPP, more than one time interval is specified ($N > 1$). We will call the time pairs of a resource partition activation slots and in this paper, we will focus on the derivation of STSPPs for each virtual machine to obtain feasible schedules.

II. Contribution of this paper

To eliminate the need of paravirtualization because of licensing reasons, we will introduce a methodology to derive a virtual real-time system from given periodic tasksets Γ_j. The local schedulers of the VMs will be left completely untouched and do not communicate with the scheduler of the hypervisor. The periodic tasksets will hence run in a full virtualized environment in contrast to existing solutions like the open system environment [3], [4], PikeOS[5], and OKL4 [6] which require a paravirtualized guest operating system. To make the given tasksets Γ_i executable on the virtual real-time system we will determine the needed processor speed of the virtual real-time system and transform the tasksets Γ_i according to the determined speed into tasksets Γ'_i to be executed on our virtual real-time system defined in Section III. Based on that model we will illustrate the problem of determining the activation slots of the STSPPs in Section IV. In Section V we will show that it is not possible to choose the partition period P arbitrarily followed by Section VI which will give a solution to this problem. To show the applicability of our model we evaluated our model in a case study being presented in Section VIII and show a solution in section IX on how to face the problem of the overhead introduced by our model.

III. Modeling the virtualized system

We define the virtual real-time system using STSPPs as:

Definition A virtual real-time system is composed of $i = 1, ..., n$ periodic tasksets with $\Gamma_i = \{\tau_k(T_k, C_k) | k = 1, ..., m\}$ according to the periodic task model of Liu et. al [1]. These tasksets are executed in virtual machines using their own local EDF or RM scheduler. The virtual machines themselves are scheduled by the VMM using a STSPP $\Pi_i = \{\{(S_i, E_i)\}, P | S_1 = 0, E_i = S_i + \alpha_i \cdot U(\Gamma_i) \cdot P, S_i = E_{i-1}\}$ to activate each virtual machine.

This definition essentially describes a virtual real-time system hosting n different virtual machines being scheduled by a simple fixed time slice scheduler (FTS), where the virtual machines are activated once within the period P. The advantage of our approach is that it is very easy to implement and it enables the usage of full virtualization eliminating the need of paravirtualization as this might be restricted due to software licenses.

Beside the problem of determining the parameters of the STSPPs P_i we may have the problem that the virtual real-time system is overloaded when executing the tasksets Γ_i directly as virtual machines. Therefore it is necessary to transform the tasksets Γ_i into tasksets Γ'_i that do not overload the virtual real-time system and still guarantee a feasible schedule for their local scheduler. Therefore we assume that the real-time systems RS_i are executed on different hardware platforms. Our goal is to give the designer a hint on how fast the virtualized system has to be able to host the virtual machines. We will first normalize the tasks' execution times based on the slowest hardware platform used. Then we will consider the utilization of this hardware platform when executing all real-time system as virtual machines as the speedup factor which is needed to dimension the host system relatively to the slowest hardware platform. To simplify matters, we assume that all real-time systems use a common processor architecture (pipelines, caches, etc.). This leads to a simple speedup factor relatively to the clock rate of the slowest system which we will define in this section.

First we start with the definition of the normalization factor based on the given real-time system introduced in section I which will allow the normalization of the overall task set:

Definition Let $c_1, ..., c_n$ be the clock rates of the systems $RS_1, ..., RS_n$. Assuming $c_s = min(c_1, ..., c_n)$, then the factors $s_1 = \frac{c_1}{c_s}, ..., s_n = \frac{c_n}{c_s} \in \mathbb{R}^+$ are the normalization factors of each system based on the slowest system.

In case of executing all virtual machines on the slowest hardware platform, the execution times of the tasks have to be recalculated to be in line with the execution time on this platform.

$$\Gamma_{i_s} = \{\tau_k(T_k, C_k \cdot s_i) | \tau_k \in \Gamma_i\} \quad (1)$$

$$U(\Gamma_{i_s}) = \sum_{\tau_k \in \Gamma_i} \frac{C_k \cdot s_i}{T_k} = s_i \cdot U(\Gamma_i) \quad (2)$$

As the slowest system may be overloaded, we now define a speedup factor S relatively to this system to determine the required speed of the virtualization host system which executes all virtual machines. So the execution times of the tasks have to be recalculated to be in line with the execution time on

the virtualization host. However, one may not find a system in practice which offers exactly the needed speedup S, but one can choose the best one fitting the demand.

$$\Gamma'_i = \{\tau_k(T_k, \frac{C_k \cdot s_i}{S}) | \tau_k \in \Gamma_{i_s}\} \quad (3)$$

The utilization of a VM_i executing Γ'_i considering the speedup factor S is

$$U(\Gamma'_i) = \sum_{\tau_k \in \Gamma_i} \frac{C_k \cdot s_i}{S \cdot T_k} = \frac{s_i}{S} \cdot U(\Gamma_i) \quad (4)$$

The identification of S depends on the scheduling algorithm used within each VM. We will cover EDF and RM in the following two subsections.

A. Scaling based on EDF

Assuming all real-time systems use EDF with its utilization bound of 1, the speedup factor S can be calculated as the overall utilization of all virtual machines executing on the slowest system.

The speedup factor S for EDF based virtual machines is the overall utilization of the original tasksets given as:

$$S = U(\bigcup_{i=1}^{n} \Gamma_{i_s}) = \sum_{i=1}^{n} U(\Gamma_{i_s}) \quad (5)$$

The utilization of the virtualized host in that case can thus be calculated by

$$U(\bigcup_{i=1}^{n} \Gamma'_i) = \sum_{i=1}^{n} \frac{s_i}{S} \cdot U(\Gamma_i) = 1 \quad (6)$$

and shows a full utilization of the virtualized host system.

B. Scaling based on RM

To guarantee the schedulability using RM, it is not possible to use the scaling of the previous section as this is based on the assumption that the system can be fully utilized through the fact that the utilization bound of EDF is 1. Since the utilization bound of RM depends on the task set, we define the speedup factor S to be relative to the least upper bound of the virtual machines.

The speedup S for a virtual real-time system consisting of VMs using RM only is given as:

$$S = U(\bigcup_{i=1}^{n} \Gamma_{i_s}) = \sum_{i=1}^{n} \frac{1}{U_{lub}(\Gamma_i)} \cdot U(\Gamma_{i_s}) \quad (7)$$

with $U_{lub}(\Gamma_i) = m \cdot (2^{\frac{1}{m}} - 1)$[1] being the least upper bound of the taskset Γ'_i.

The resulting utilization of the virtualized host can be calculated by

$$U(\bigcup_{i=1}^{n} \Gamma'_i) = \frac{1}{S} \cdot \sum_{i=1}^{n} s_i \cdot U(\Gamma_i) \quad (8)$$

We have shown in this section how to determine the speedup factor S based on the used scheduling algorithm and how to transform the given tasksets Γ_i into tasksets Γ'_i being executed on the virtual real-time system being S times faster than the slowest given real-time system while not overloading the new system. However to execute these tasksets without violating the deadlines we have to determine the appropriate parameters of the corresponding STSPP Π_i.

IV. Determining the activation slots

To answer the question on how to determine the activation slots of a STSPP Π_i we assume that the first VM starts at time $t = 0$ thus $S_1 = 0$. The parameter E_i has to be scaled relatively to the utilization bound using α_i introduced in the definition of the virtual real-time system. α_i thus depends on the applied scheduling algorithm as scheduling algorithms may differ in their utilization bound. In case of EDF $\alpha_i = 1$ has to be used to fully utilize the virtual system. The activation slot of Π_i for VM_i can then be calculated by:

$$S_i = E_{i-1} \quad (9)$$

$$E_i = S_i + U(\Gamma'_i) \cdot P \quad (10)$$

In case of RM we need to scale relatively to the utilization bound $U_{lub}(\Gamma_i)$ resulting in $\alpha_i = \frac{1}{U_{lub}(\Gamma_i)}$. The slots can then be calculated by:

$$S_i = E_{i-1} \quad (11)$$

$$E_i = S_i + \frac{1}{U_{lub}(\Gamma_i)} U(\Gamma'_i) \cdot P \quad (12)$$

Theorem 1. *Let the period P of an STSPP Π_i be arbitrarily chosen and $\alpha_i = 1$ in case of EDF scheduling respectively $\alpha_i = \frac{1}{U_{lub}(\Gamma_i)}$ in case of RM scheduling. Then the ratio of the required utilization $U(\Gamma'_i)$ to the allocated utilization $\frac{E_i - S_i}{P}$ is equal to the utilization bound of the VM executing the associated taskset Γ'_i.*

Proof:

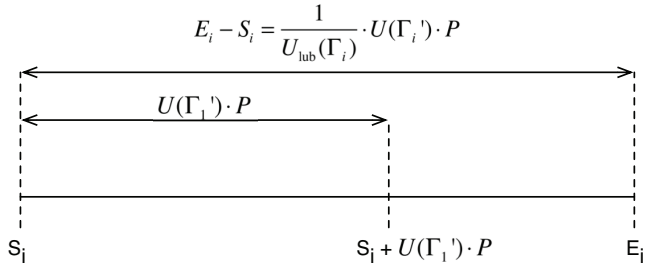


Fig. 2. Graphical illustration of the required computation time within the allocated interval $E_i - S_i$ of Π_i

Figure 2 shows how the absolute times for the required computation time and the allocated computation time within the period P are calculated. By dividing the required computation time $R = U(\Gamma'_i) \cdot P$ by the allocated computation time $A = E_i - S_i = \alpha_i \cdot U(\Gamma'_i) \cdot P$ we get

$$\frac{R}{A} = \frac{U(\Gamma'_i) \cdot P}{\alpha_i \cdot U(\Gamma'_i) \cdot P} = \frac{1}{\alpha_i}$$

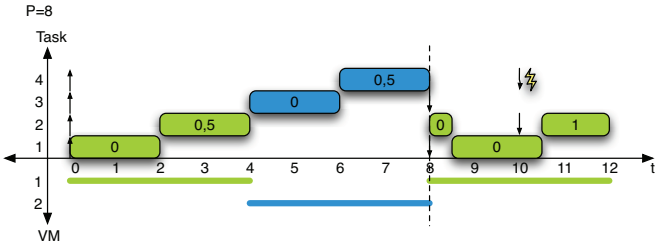


Fig. 3. Missed deadline due to the wrong selection of the STSPP period. The numbers within the boxes denote the remaining computation time of that task.

In case of EDF with $\alpha_i = 1$ this results in the fact that within the interval $E_i - S_i$ the allocated computation time is exactly the required computation time so that $\frac{R}{A} = 1$ being the utilization bound of EDF.

In case of RM with $\alpha_i = \frac{1}{U_{lub}(\Gamma_i)}$ this results in the fact that within the interval $E_i - S_i$ the ratio of the required computation time to the allocated computation time is $\frac{R}{A} = U_{lub}(\Gamma_i)$ ■

V. Problems choosing the period of STSPPs

Although STSPPs are a technically simple possibility to schedule a set of virtual machines it is not possible to choose the period arbitrarily. Figure 3 shows an example schedule with two virtual machines using EDF. The first virtual machine executes a task set of two tasks $\Gamma'_1 = \{\tau_1(8,2), \tau_2(10,2.5)\}$ and the second virtual machine contains a task set with the same computation times and deadlines $\Gamma'_2 = \{\tau_3(8,2), \tau_4(10,2.5)\}$. We decide to choose the period of the STSPPs to be $P = 8$. Both virtual machines have a utilization factor of $U(\Gamma'_1) = U(\Gamma'_2) = \frac{1}{2}$ and do not overload the system. This results in $S_1 = 0, E_1 = S_1 + U(\Gamma_1) \cdot P_1 = 4 \Rightarrow \Pi_1 = ((0,4),8)$ and $S_2 = E_1 = 4, E_2 = S_2 + U_{\Gamma_1} \cdot P_2 = 8 \Rightarrow \Pi_2 = ((4,8),8)$. The ends of the STSPPs are highlighted by a vertical line at time $t = 8$ and the activation time slots within the STSPPs are shown by the bar below the schedule. The execution order is chosen arbitrarily. By the bad choice of $P = 8$, task 4, which is executed in VM_2, will miss its deadline at time $t = 10$, because the required utilization $U(\Gamma_2) = \frac{1}{2}$ has not been completely assigned by Π_2 at $t = 10$.

The timing restrictions of the original real-time systems have to be kept in mind when determining the properties of a virtualized system hosting multiple real-time systems as virtual machines. Therefore, information about the performance of the original real-time systems is needed. Imagine the case that two real-time systems, both fully utilizing the same microprocessor, shall be virtualized to a new system. To prevent overloading, the virtualized system obviously needs at least a microprocessor of twice the speed of the original ones. Additionally, the worst case execution times (WCET) and the deadlines of all real-time tasks have to be available to consider the real-time aspects. With this information, it is possible to determine the length of the period P for a STSPP

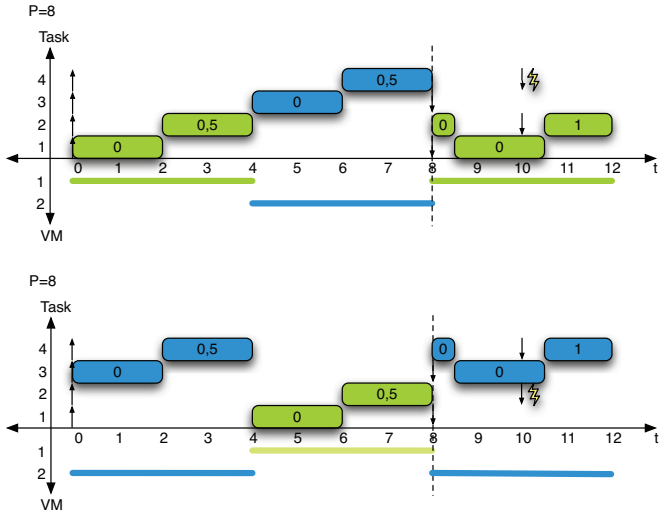


Fig. 4. Missed deadline due to the wrong selection of the STSPP period. Exchange of the activation order has no effect.

of a virtual machine in such a manner that no deadline in the virtual machines will be missed.

As we have now seen there is a serious problem of missing deadlines concerning an arbitrary choice of the period length P of the STSPPs. We will show how to derive the STSPPs to guarantee that no deadline miss occurs based on the tasksets Γ'_i which will be derived in the following section.

VI. Calculating the period of the STSPPs

If we consider only STSPPs as a possible partitioning scheme, the period lengths have to be equal. Otherwise, the activation slots of the partitions may overlap, which is not possible for a uniprocessor system. The remaining question to be answered in this section is how to calculate the period P of the STSPPs for each virtual machine VM_i.

As stated in section V, the choice of P is very important, since the real-time tasks scheduled within the virtual machines must not miss their deadlines. When P is chosen arbitrarily, a deadline may occur anywhere within the interval P. The activation of the virtual machine within its STSPP has to be timed in such a manner that the reserved utilization of the VM has been completely assigned at the time of its deadline. We will now reconsider the example in figure 4. This example shows that it is not possible to fix the missed deadline at time $t = 10$ by simply exchanging the order in which the virtual machines are activated within their partitions. This is due to the fact that the example consists of two identical virtual machines. Figure 5 illustrates the assigned computation time $Comp(\Pi, t)$ to a VM by a STSPP Π for the given example.

The supply function U_A represents the assigned utilization by STSPP Π at time t within the whole system and is given as:

$$U_A(\Pi_i, t) = \frac{Comp(\Pi, t)}{t} \tag{13}$$

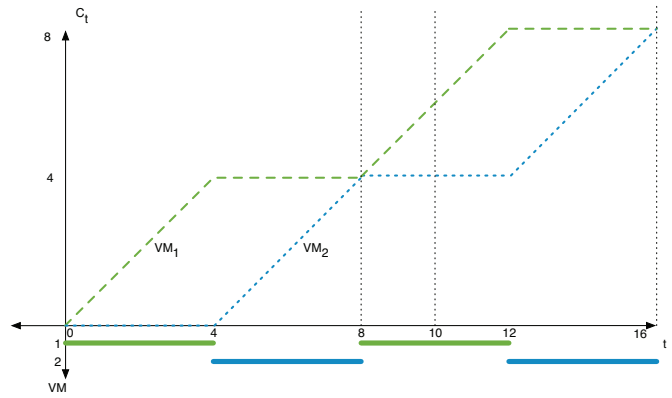


Fig. 5. Allocated computation time $Comp(\Pi, t)$ for VM_1 and VM_2 assigned by $\Pi_1 = \{(0, 4), 8\}$ and $\Pi_2 = \{(4, 8), 8\}$

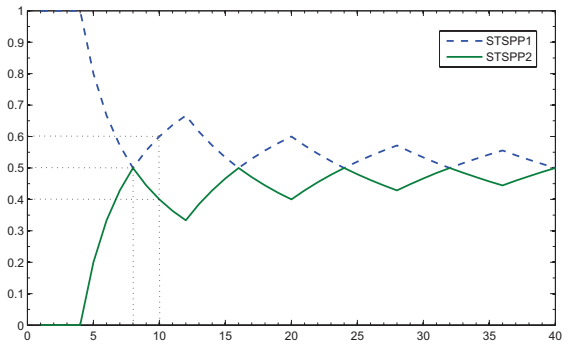


Fig. 6. $U_A(\Pi_1, t)$ and $U_A(\Pi_2, t)$ of the given example.

In this example at time $t = 10$, the assigned utilization of VM_2 is not equal to the requested utilization of $U(\Gamma'_2) = \frac{1}{2}$ which can be seen in figure 5 and 6. To ensure this important property we will now state our main theorem of this paper.

Theorem 2. *When* $S_0 = 0$, $S_i = E_{i-1}$ *and* $E_i = S_i + \frac{1}{U_{lub}(\Gamma_i)} U(\Gamma'_i) \cdot P$ *for every* Π_i *and* P *is chosen as* $P = \gcd(\{T_k | \tau_k \in \bigcup_{i=1}^{n} \Gamma'_i\})$, *then no task* $\tau_k \in \bigcup_{i=1}^{n} \Gamma'_i$ *will miss its deadline, if the tasksets* Γ_i *are transformed by* $\Gamma'_i = \{\tau_k(T_k, \frac{C_k \cdot s_i}{S}) | \tau_k \in \Gamma_{i_s}\}$ *with* $S = \sum_{i=1}^{n} \frac{1}{U_{lub}(\Gamma_i)} \cdot U(\Gamma_{i_s})$.

Proof:

To show that the assigned allocation is guaranteed at every single deadline of a virtual machine, we claim that there exists a task $\tau_k \in \Gamma'_i$ with deadline T_k where $U_A(\Pi_i, t)$ at time $t = T_k$ is smaller than the required utilization $U(\Gamma'_i)$.

$$\exists \tau_k \in \Gamma'_i : U_A(\Pi_i, T_k) < U(\Gamma'_i)$$

Due to Theorem 1 we know that $U(\Gamma_i)$ is equal to the required utilization R depending on the least upper bound while U_A is equal to the allocated computation time A assigned by the STSPP Π_i. Thus we can follow

$$R \cdot \alpha_i < R,$$

what is a contradiction, because of $\alpha_i \geq 1$.

■

VII. Handling aperiodic loads

The occurrence of aperiodic loads within the virtual machines can be handled by using static priority servers in case of RM oder dynamic priority servers in case of EDF. Therefore the periods of the servers need to be considered when determining the period P of the STSPPs. This ensures the correct timing behavior of all periodic real-time tasks. The value of P has a big impact on interrupt latency. The bigger P gets the worse the interrupt latency gets, as the activation slots grow with P. While another virtual machine is active, interrupts have to be buffered for the non-active virtual machines for as long as the other virtual machines are active. Thus there is always a trade-off between virtualization overhead induced by context-switching and interrupt latency as virtualization overhead decreases with P while interrupt latency increases with P.

VIII. Case Study

To evaluate our approach, we decided to use our real-time operating system ORCOS together with our virtualization platform Proteus[7].

Proteus is the first hybrid configurable virtualization platform which supports both full virtualization and paravirtualization, even a mixture of both if needed. The high configurability allows the system designer to adopt the platform for its dedicated field of application. By consequence, very small and efficient systems can be designed. The maximum memory overhead induced by the virtual machine monitor is below 11Kb which lets Proteus compete with current state-of-the-art virtualization platforms like Trango[8] or VirtualLogix[9].

ORCOS is a reimplementation of DREAMS[10], [11] which was developed at the University of Paderborn within the last decade. ORCOS is designed to be highly portable and may be run on a broad number of target platforms, from tiny microcontrollers with restricted computing resources to embedded platforms with megabytes of RAM.

After this very short introduction of Proteus and ORCOS we will now describe our example scenario and the resulting theoretical schedules in section VIII-A. Afterwards in section VIII-B the execution of the scenario on real hardware is compared to the theoretical results.

A. Scenario

We use two different real-time systems, executing two tasks each. The first real-time system is executed on a PowerPC405 running at 150MHz, while the second real-time system is executed on a PowerPC405 running at 450MHz. The operating system of these systems is the smallest possible ORCOS

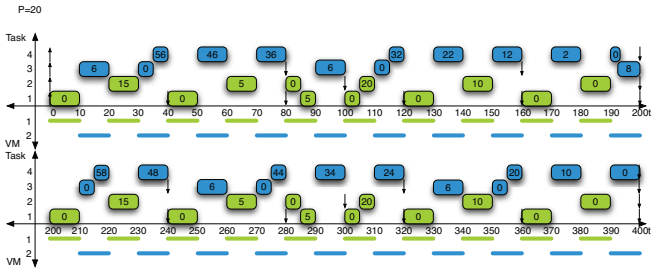


Fig. 7. Example schedule of a virtualized real-time system hosting two real-time virtual machines using EDF.

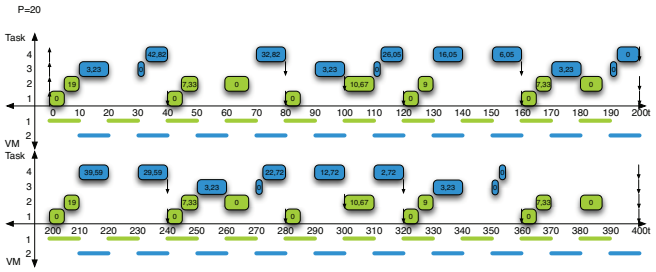


Fig. 8. Example schedule of a virtualized real-time system hosting two real-time virtual machines using RM.

configuration to avoid unnecessary OS overhead in our measurements. We will transform these real-time systems into virtual machines and derive the STSPPs with their activation time slots according to our presented methodology in section III and VI. Both virtual machines are executed on top of our hypervisor Proteus. Caching is completely disabled for this case study, on the real-time systems as well as on the virtualization host.

First, we define the task sets and the clock speeds of the real-time systems. The WCETs and the periods are denoted in milliseconds. We can calculate the slow down factors s_1 and s_2:

$$\Gamma_1 = \{\tau_1(40, 20), \tau_2(100, 50)\}$$
$$\Gamma_2 = \{\tau_3(80, \frac{32}{3}), \tau_4(200, \frac{120}{3})\}$$
$$c_1 = 150 \Rightarrow s_1 = 1, c_2 = 450 \Rightarrow s_2 = 3$$

We scale both virtual machines as if they were executed on the slowest system:

$$\Rightarrow \Gamma_{1_s} = \{\tau_1(40, 20), \tau_2(100, 50)\}$$
$$\Rightarrow \Gamma_{2_s} = \{\tau_3(80, 32), \tau_4(200, 120)\}$$

We have to distinguish between the use of RM and EDF.

1) EDF: The use of EDF in both virtual machines results in a speedup factor $S = 2$. By consequence, the virtualization host needs to be twice as fast as RS_1. We need at least a PowerPC405 running at 300 MHz. The execution of the virtual machines on this system results in the following task sets:

$$\Rightarrow \Gamma'_1 = \{(40, 10), (100, 25)\}$$
$$\Rightarrow \Gamma'_2 = \{(80, 16), (200, 60)\}$$

The overall utilization of the virtualized host is 100%.

$$P = \text{gcd}(\{40, 80, 100, 200\} = 20$$

Now, we can derive the STSPPs:

$$\Pi_1 = (\{(0, 5)\}, 10) \quad (14)$$
$$\Pi_2 = (\{(5, 10)\}, 10) \quad (15)$$

The schedule for the case that both virtual machines use EDF is depicted in figure 7. No task misses its deadline till the hyperperiod of all tasks.

2) RM: The speedup factor in the case of RM is $S = 2 \cdot \frac{1}{2 \cdot (2^{\frac{1}{2}} - 1)} \approx 2,42$:

$$\Rightarrow \Gamma'_1 = \{(40, 8.27), (100, 20.67)\}$$
$$\Rightarrow \Gamma'_2 = \{(80, 13.23), (200, 24.80)\}$$

The resulting overall utilization of the virtualized host is approximately 83% which is equal to the least upper bound for two tasks.

$$P = \text{gcd}(\{40, 80, 100, 200\} = 20$$

The resulting STSPPs are:

$$\Pi_1 = (\{(0, 5)\}, 10)$$
$$\Pi_2 = (\{(5, 10)\}, 10)$$

B. Real Execution

We executed the two virtual machines, which used EDF as the local scheduling scheme, on our development platform, a PowerPC405 running at 300 MHZ, using Proteus using the schedule based on the STSPPs 14 and 15. The tasks read a hardware register, calculated some mathematical equation and stored the value into another hardware register within each instance. With a logic analyzer we measured the points of time at which a task started or stopped execution. With these time points, we were able to monitor the real schedule of the system as seen in figure 9. Only the first 400ms are shown, since the schedule repeats afterwards.

In order to get the real system running we had to face multiple problems. We assume the worst case execution time of the original tasks to be calculated by some means. We also assume the execution time needed to execute the interrupt handlers and the kernel code, as well as the time needed for switching the context, to be included in the task's execution times. This can be observed at the beginning of the third time slot of the virtual machine one. At that time point, the kernel is handling a timer interrupt and is starting the second instance of task one which has a higher priority. The execution time for this operation belongs to the execution time of task one,

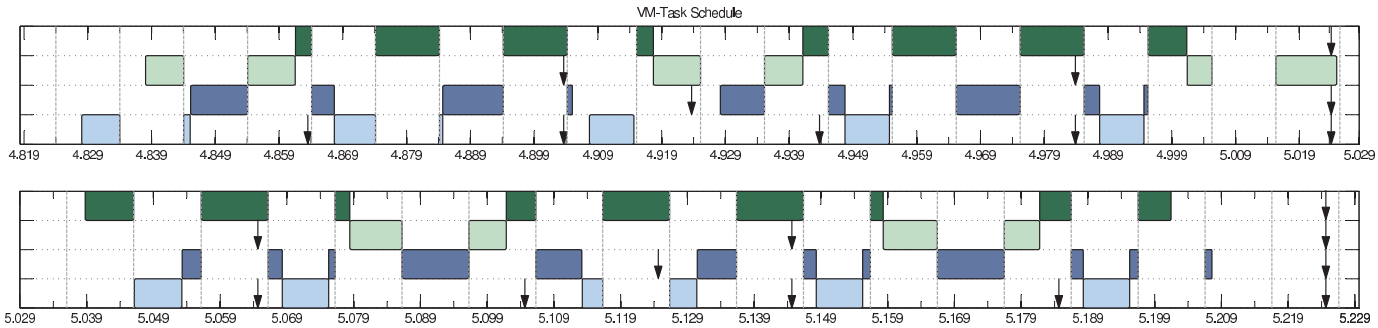


Fig. 9. Monitored schedule on a PowerPC405@300 MHz with two virtual machines using ORCOS running on the Proteus hypervisor.

but due to the event monitoring approach, it is drawn as a part of task two. The same holds for the boot up process at the beginning of each virtual machine execution which is not drawn at all, but can be recognized as the delayed start time of the tasks.

Whenever there is no task executing and the kernel is operating (e.g. interrupt handling), the execution time spent is not drawn. This can be observed at the 11th time slot of virtual machine one.

By assuming the above definition of the worst case execution time, we may also ignore the problem of task synchronization at system start, since our algorithm expects all tasks to be released at the same time so that the switching time is an integer fraction of the deadlines. This is feasible for our example since the boot up process of ORCOS is very short and thus the overhead induced by this assumption stays small compared to the computation time of the tasks. Accordingly, the execution time of later instances of the tasks are fairly lower than of those at the beginning, which leads to idle times inside the VMs. Anyhow, it is basically impractical to get an utilization of exactly one inside the real system.

Regarding later synchronization issues, we assume that the specific time slices of the virtual machines can be generated cycle-accurately by the virtual machine monitor.

As the figure shows, the deadlines of all tasks were met and we were able to place two formerly physically spread systems onto one platform while guaranteeing the real time constraints of all tasks.

IX. Reducing the Virtual Machine Switching Overhead

In some cases the time slice period P and the resulting activation slots (E_i, S_i) might become too small to place the virtual machines together on one system since the switching overhead would become too big. In these cases however, it is still possible to run the virtual machines with slightly increased period P. This will probably increase the needed speedup factor for a virtual machine in order to meet its realtime constraints since the period might no longer be a fraction of the deadlines. Let us reconsider the example from above with $\Gamma'_1 = \{\tau_1(40, 10), \tau_2(100, 25)\}$ and $\Gamma'_2 = \{\tau_3(80, 16), \tau_4(200, 60)\}$. The computed period $P = \gcd(\{40, 80, 100, 200\} = 20$ resulted in time intervals of length $E_1 - S_1 = E_2 - S_2 = 10$. In order to determine the speedup needed for a higher period, it is possible to examine the quotient of the needed amount of computation time $N(t)_i$ and the provided amount $Z(t)_i$ of computation time up to any deadline of a task inside a VM_i. Since we get different quotients for each deadline depending on the period P and on the time slices $Ts_i = E_1 - S_1 | E_1, S_1 \in STSPP_i$, we need to take the greatest of those quotients to fulfill the real-time

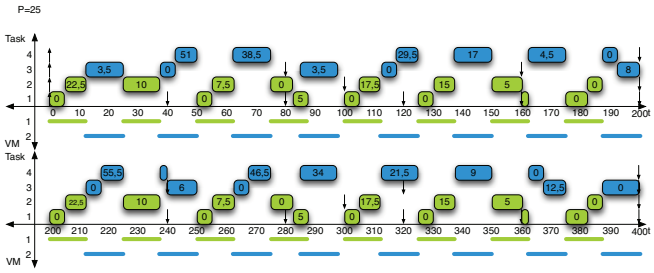


Fig. 10. Task Schedule of the example virtual machines using the period $P = 25$.

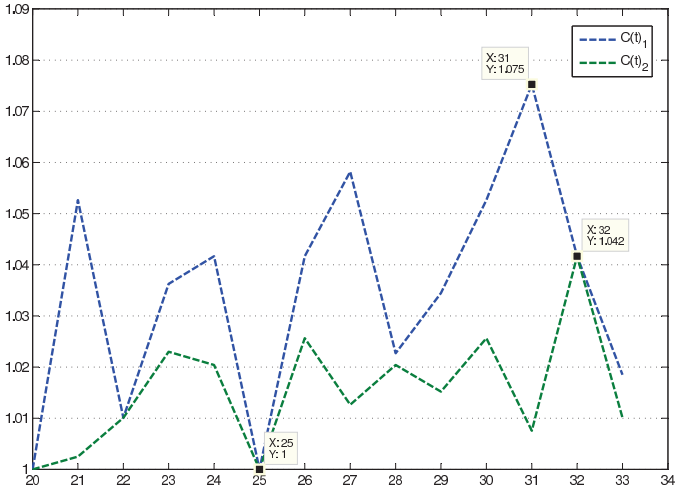


Fig. 11. Computation of the speedup factor $C(t)_i$ which is needed to generate a feasible schedule for different time slice periods P and virtual machines VM_i.

constrains for all deadlines. We call this value $C(t)_i$ for VM_i:

$$C(t)_i = max_{t \in T_{D_i}} \frac{N(t)_i}{Z(t)_i} \quad (16)$$

In order to compute the provided time for VM_i up to a point of time t, it is possible to use the following modified response time analysis formula:

$$Z(t)_i = \left\lfloor \frac{t}{P} \right\rfloor \cdot Ts_i + (t - \left\lfloor \frac{t}{P} \right\rfloor \cdot P - \delta_i) \quad (17)$$

δ_i represents the relative starting time of the time slice inside the period P. It is possible to assume the worst case here for all virtual machines, which assumes that VM_i is executed last inside the period P. The amount of needed computation time up to a point of time t can be calculated by the following formula:

$$N(t)_i = \sum_{\tau_j \in \Gamma'_i} \left\lfloor \frac{t}{T_j} \right\rfloor \cdot C_j \quad (18)$$

The calculation needs to be done only for all possible deadlines T_{D_i} of all tasks $\tau_j \in VM_i$ up to the least common multiple of the periods of the tasks and P, since afterwards the schedule repeats:

$$T_{D_i} = \bigcup_{\tau_j \in \Gamma'_i, k \in \mathbb{N}} \{k \cdot T_j\} \mid k \cdot T_j \leq lcm(\cup\{T_j\} \cup \{P\}) \quad (19)$$

The calculation of the quotient $C(t)_i$ has been done for a series of values P for the virtual machines described above. The results can be seen in figure 11. In order to generate a feasible schedule, the maximum speedup over all virtual machines needs to be used for a chosen value of P. Thus, in order to generate a feasible schedule, a value of $P = 31$, resulting in time slices $T_{s_1} = T_{s_2} = 15.5$, would need a 1.075 times faster system. There might exist values of P bigger than the greatest common divisor which will still generate a feasible schedule without any further speed up. This can be seen in figure 11 at $P = 25$.

As figure 10 shows, the schedule with the newly computed period $P = 25$ is still feasible without any further speedup, although the period is not a fraction of all task periods. Using the modified response time analysis, it is possible to create a modified system using a higher period P and thus to reduce the relative cost for switching between the virtual machines.

X. Related Work

There already exist a few commercial virtualization platforms or hypervisors for a range of embedded processors, nearly all of them being proprietary. Trango[8] and VirtualLogix[9] allow virtualization for a range of ARM and MIPS processors. Green Hills Integrity[12] and LynxSecure[13] use virtualization to implement high security systems targeted for the military market and have been certified to fulfill the EAL7 standard. All these products use paravirtualization to provide reasonable performance and support real-time applications only by the use of dedicated resources. Naturally, this limits the applicability of these products to a subset of all possible scenarios.

The scheduling of real-time virtual machines has been realized in different ways. Deng, Liu and Sun proposed a system called Hierarchical Scheduling for Open System Environments[3], [4] which is based on a server approach. This method uses a global EDF scheduler to schedule a set of servers which contain a local scheduler for real-time applications. Servers can leave and enter the system dynamically. An analysis on how to determine appropriate servers is given by Lipari and Bini[14]. The global EDF scheduler needs however access to information of the local schedulers. This is only possibly in case of paravirtualization.

Mok, Feng and Chen introduced in [2] a formalism for the schedulability analysis of a task group executed on a single time slot or multiple time slot periodic partition. Their approach allows to execute these task groups in a virtual machine through the use of such a partition, without being necessarily paravirtualized. This is of great advantage if the licensing restrictions prohibit the adaption of an RTOS to the underlying hypervisor. Shin and Lee extended this work in [15], [16], [17]. They developed a compositional real- time scheduling framework where global (system-level) timing properties are established by composing together independently (specified and) analyzed local (component- level) timing properties. This work goes quite in the same direction as our work does, but they assume a given period length for composition, while we are focusing on deriving the period length and the activation

Another real-time approach is a partitioning with proportional share scheduling[18], [19]. A fixed fraction of the CPU is assigned to each application. This can be realized by a simple fixed time slice scheduler. A deterministic, predictable activation of the hosted applications is ensured. The approach can be applied to schedule different virtual machines concurrently, based on their computational weight. The problem arising, when using proportional share scheduling, is the decision on how to choose the granularity being defined by the time quantum q. This is similar to the decision of determining the period length of STSPPs being the main focus of this paper.

In [20] and [5], Kaiser introduces a method to schedule hard real-time virtual machines according to proportional share scheduling which is used in PikeOS. It requires paravirtualization of the guest operating systems and allows the concurrent execution of time-triggered and event-triggered hard real-time virtual machines. Each virtual machine obtains a fixed priority and is assigned to a time domain. The time domains are activated by round robin scheduling while the virtual machines within a time domain are activated by a priority based fixed time slice scheduling. The length of the time slice is calculated based on proportional share scheduling. To guarantee short latencies, event-triggered virtual machines are placed in a background time domain which is always active together with the other active time domain. If such an event-triggered virtual machine becomes active, it may preempt the active time-triggered virtual machine in case of a higher priority. The configuration of the time domains is done offline and faces the

same problem of choosing the correct parameters for the time slots and the period length as just described for proportional share scheduling.

XI. Conclusion and Future Work

The approach proposed in this paper enables the design of a virtualized system hosting different real-time systems without modifications of the operating systems. The major features of our methodology are:

- Full virtualization support
- EDF and RM are supported as guest OS scheduler
- Simple determination of the required speedup for the virtualization host
- Simple determination of the STSPP period length and their activation time slots
- Improvement of the period length by response time analysis at the cost of additional speedup

The proposed methodology was evaluated using our hybrid hypervisor Proteus on a PowerPC405 prototyping platform with two virtual machines executing our real-time operating system ORCOS. This experiment proofs the applicability to real systems.

In the future, we would like to investigate whether it is possible to enlarge the period lengths without performing a response time analysis to further reduce the switching overhead induced by the virtual machine monitor. Another interesting aspect is the synchronization of the virtual machines to ensure that the switching times of the virtual machine monitor is synchronized with the deadlines of the tasks. We want to achieve this without losing the support of full virtualization.

References

[1] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for computing Machinery*, Jan 1973.

[2] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," *Proc. of IEEE Real-Time Technology and Applications ...*, Jan 2001.

[3] Z. Deng, J. Liu, and J. Sun, "A scheme for scheduling hard real-time applications in open system environment," *Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on*, pp. 191 – 199, May 1997.

[4] Z. Deng and J. Liu, "Scheduling real-time applications in an open environment," *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pp. 308 – 319, Nov 1997.

[5] R. Kaiser, "Alternatives for scheduling virtual machines in real-time embedded systems," *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, Apr 2008.

[6] "Virtualization for embedded systems," Open Kernel Labs, Tech. Rep., Apr 17 2007.

[7] D. Baldin and T. Kerstan, "Proteus, a hybrid virtualization platform for embedded systems," *Proceedings of the International Embedded Systems Symposium, 14 - 16 September 2009 IFIP WG 10.5*, 2009.

[8] VmWare, "TRANGO Virtual Prozessors: Scalable security for embedded devices." Website, February 2009, http://www.trango-vp.com/.

[9] VirtualLogix, "VirtualLogix - Real-time Virtualization for Connected Devices :: Products - VLX for Embedded Systems:," Website, February 2009, http://www.virtuallogix.com/.

[10] C. Ditze, "A customizable library to support software synthesis for embedded applications and micro-kernel systems," *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, Sep 1998. [Online]. Available: http://portal.acm.org/citation.cfm?id=319195.319209

[11] ——, "A step towards operating system synthesis," in *In Proc. of the 5th Annual Australasian Conf. on Parallel And Real-Time Systems (PART). IFIP, IEEE*, 1998.

[12] G. Hills, "Real-Time Operating Systems (RTOS), Embedded Development Tools, Optimizing Compilers, IDE tools, Debuggers - Green Hills Software," Website, February 2009, http://www.ghs.com/.

[13] LynuxWorks, "Embedded Hypervisor and Separation Kernel for Operating-system Virtualization: LynxSecure," Website, February 2009, http://www.lynuxworks.com/virtualization/hypervisor.php.

[14] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing*, Jan 2005.

[15] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," *Proceedings of the 24th IEEE International Real-Time ...*, Jan 2003.

[16] ——, "Compositional real-time scheduling framework," *Proc. of IEEE Real-Time Systems Symposium*, Jan 2004.

[17] ——, "Compositional real-time scheduling framework with periodic model," *ACM Transactions on Embedded Computing Systems ...*, Jan 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1347383

[18] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," *Real-Time Systems Symposium, 1996., 17th IEEE*, pp. 288 – 299, Nov 1996.

[19] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson, "Proportional share scheduling of operating system services for real-time applications," *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pp. 480 – 491, Nov 1998.

[20] R. Kaiser, "Applying virtualization to real-time embedded systems," *1. GI/ITG KuVS Fachgespr"ach Virtualisierung*, 2008.

[21] J. E. Smith and R. Nair, *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[22] G. C. Buttazzo, *Hard Real-Time Computing Systems*. Springer, 2004.

[23] R. Greene and G. Lownes, "Embedded cpu target migration, doing more with less," 1994, 197743 429-436.

Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability*

Andrea Bastoni† Björn B. Brandenburg‡ James H. Anderson‡

Abstract

A job that is impeded by a preemption or migration incurs additional cache misses when it resumes execution due to a loss of cache affinity. While often regarded negligible in scheduling-theoretic work, such cache-related delays must be accounted for when comparing scheduling algorithms in real systems. Two empirical methods to approximate cache-related preemption and migration delays on actual hardware are proposed, and a case study reporting measured average- and worst-case overheads on a 24-core Intel system with a hierarchy of shared caches is presented. The widespread belief that migrations are always more costly than preemptions is refuted by the observed results. Additionally, an experiment design for schedulability studies that allows algorithms to be compared objectively under consideration of cache-related delays is presented.

1 Introduction

A controversial topic with regard to the choice of scheduler in multiprocessor real-time systems—both within academia and among practitioners—is the relative impact of *cache-related preemption and migration delay* (CPMD), *i.e.*, the delay that a preempted job incurs due to a loss of cache affinity after resuming execution.

Traditionally, migrations are considered to be a source of unacceptable overhead, and thus implementors tend to favor *partitioning* to avoid migrations altogether. This view, however, conflicts with scheduling-theoretic advances from the last decade that show that there exist *global* schedulers, which allow jobs to migrate freely, that are provably superior to partitioning if overheads (including CPMD) are negligible [5, 17, 19, 24, 33]. In contrast, if CPMD is deemed to be significant (which, of course, it is in practice), then one can easily *construct* scenarios in which global scheduling is not a viable alternative by assuming prohibitively high migration costs. However, how realistic are such scenarios?

Clearly, a meaningful comparison of schedulers requires CPMD to be taken into account, yet arbitrarily choosing delays is of only little benefit. The problem is compounded by the difficulty of measuring CPMD [9], which can only be observed indirectly and is heavily dependent on the *working set size* (WSS) of each task, in contrast to other OS delays such as scheduling overhead [8, 9].

This raises three central questions:

1. How can CPMD be estimated empirically when evaluating algorithms for a specific platform?
2. What are reasonable values to assume for CPMD when evaluating (newly-proposed) algorithms in scheduling-theoretic work?
3. How can schedulers be evaluated without accidentally introducing a bias towards a particular WSS?

In particular, we are interested in *simple*, yet effective, methods that can be realistically performed as part of scheduling research and by practitioners during early design phases (*i.e.*, when selecting platforms and algorithms).

Contributions. In this paper, we propose two approaches to measure CPMD: a "schedule-sensitive method" that can measure scheduler-dependent cache effects (Sec. 3.1), and a "synthetic method" that can be used to quickly record a large number of samples (Sec. 3.2). (A preliminary version of the "schedule-sensitive method" was previously used—but not described in detail—in [9].)

To demonstrate the efficacy of our approaches, we report average and maximum CPMD for various WSSs on a current 24-core Intel platform with hierarchical caches (Sec. 4). Perhaps surprisingly, in a system under load, *migrations were found to not cause significantly more delay than preemptions* (Sec. 4.2). In particular, our results show that CPMD is **(i)** in excess of one millisecond for working set sizes exceeding 256 KB (Fig. 2(a)), **(ii)** ill-defined if there is heavy contention for shared caches (Fig. 2(c)), **(iii)** strongly dependent on the preemption length (Fig. 3), and **(iv)** not dependent on task set size (Fig. 4).

Finally, we discuss how CPMD can be integrated into large-scale schedulability studies without rendering the results dependent on particular WSS assumptions (Sec. 5).

Related work. Accurately assessing cache-related delays is a classical component of *worst-case execution time* (WCET) analysis [42], in which an upper bound on the

*Work supported by AT&T, IBM, and Sun Corps.; NSF grants CNS 0834270, CNS 0834132, and CNS 0615197; ARO grant W911NF-09-1-0535; and AFOSR grant FA 9550-09-1-0549.

†SPRG, University of Rome "Tor Vergata"

‡Dept. of Computer Science, U. of North Carolina at Chapel Hill

maximum resource requirements of a real-time task is derived *a priori* based on control- and data-flow analysis. Unfortunately, predicting cache contents and hit rates is notoriously difficult: even though there has been some initial success in bounding *cache-related preemption delays* (CPDs) caused by simple data [31] and instruction caches [37], analytically determining preemption costs *on uniprocessors with private caches* is still generally considered to be an open problem [42]. Thus, on multicore platforms with a *complex hierarchy of shared caches*, we must—at least for now—resort to empirical approximation. However, given recent advances in bounding migration delays [21] and analyzing interference due to shared caches [16, 43], we expect multicore WCET analysis to be developed eventually.

Trace-driven memory simulation [41], in which memory reference traces collected from actual program executions are interpreted with a cache simulator, has been applied to count cache misses after context switches [29, 36]. Using traces from throughput-oriented workloads, Mogul and Borg [29] estimated CPDs to lie within $10\mu s$ to $400\mu s$ on an early '90s RISC-like uniprocessor with caches ranging in size from 64 to 2048 kilobytes. In work on real-time systems, Stärner and Asplund [36] used trace-driven memory simulation to study CPDs in benchmark tasks on a MIPS-like uniprocessor with small caches. As the simulation environment is fully controlled, this method allows cache effects to be studied in great detail, but it is also limited by its reliance on accurate architectural models (which may not always be available) and representative memory traces (which are difficult to collect due to complex instrumentation requirements).

Several probabilistic models have been proposed to predict expected cache misses on uniprocessors [1, 27, 39]. In the context of evaluating (hard) real-time schedulers, such models apply only to a limited extent because it is difficult to extract bounds on the worst-case number of cache misses. Further, they rely on task parameters that are difficult to obtain or predict (e.g., cache access profiles [27]), and do not predict cache misses after migrations.

Closely related to our approach are several recent CPD microbenchmarks [18, 25, 40]. Li *et al.* [25] measured the cost of switching between two processes that alternate between accessing a data array and communicating via a pipe on an Intel Xeon processor with a 512kB L2 cache, and found that average case CPDs can range from around $100\mu s$ to $1500\mu s$, depending on array size and access pattern. In the context of real-time systems, Li *et al.*'s experimental setup is limited because it can only estimate average-case, but not worst-case, delays. David *et al.* [18] measured preemption delays in Linux kernel threads on an embedded ARM processor with comparably small caches and observed CPDs in the range of $60\mu s$ to $120\mu s$. Tsafrir [40] investigated the special case in which the scheduled job is not preempted, but cache contents are perturbed by periodic clock interrupts, and found that slowdowns vary heavily among workloads. None of the cited empirical studies considered job migrations.

Once bounds on cache-related delays are known for a given task set, they must be accounted for during schedulability analysis. This is typically accomplished by inflating task parameters to reflect the time lost to reloading cache contents. Straightforward methods are known for common uniprocessor schedulers [6, 11, 22] and have also been derived for global schedulers [19, 34]; we use this approach in Sec 5. A limitation of these methods is that each preemption between any two tasks is assumed to cause maximal delay, an assumption that is likely unnecessarily pessimistic. More advanced methods that yield tighter bounds by analyzing per-task cache use and the instant at which each preemption occurs have been developed for static-priority uniprocessor schedulers [23, 30, 37]. However, similar to WCET analysis, these methods have not yet been generalized to multiprocessors since they require useful cache contents to be predicted accurately. Stamatescu *et al.* [35] propose including average memory access costs in specific analysis, but do not report measured costs.

Several research directions orthogonal to this paper are concerned with avoiding, or at least reducing, cache-related delays in multiprocessor real-time systems. On an architectural level, Sarkar *et al.* [32] have proposed a scheduler-controlled cache management scheme that enables cache contents to be transferred in bulk instead of relying on normal cache-consistency updates. This can be employed to lessen migration costs by transferring useful cache contents before a migrated job resumes [32]. Likewise, Suhendra and Mitra [38] have considered cache locking and partitioning policies to isolate real-time tasks from timing interference due to shared caches. While promising, neither technique is supported in current multicore architectures.

In work on real-time scheduling, numerous recently-proposed schedulers aim to balance the advantages of partitioning and global scheduling by reducing the number of migrations. Such hybrid approaches can be classified into two families: in *semi-partitioned* schedulers [2], most jobs are fixed to processors and only few migrate, whereas in *clustered* schedulers [4, 13], all jobs may migrate, but only among processors that share a cache. Going a step further, *cache-aware* schedulers [12, 20], which make shared caches an explicitly-managed resource, have been proposed to both prevent interference in hard real-time systems [20] and to encourage data reuse in soft real-time systems [12].

Work on hybrid schedulers makes strong assumptions on the relative costs of migrations and preemptions—to fairly evaluate the merits of said approaches thus requires sound estimates of cache-related delays. We detail our methods for obtaining such estimates in Sec. 3, after briefly summarizing required background next.

2 Background

CPMD is a characteristic of modern processors and independent of any particular task model. Given our interest in real-time systems, our focus is the classic *sporadic task model* [15, 26, 28] in which a workload is specified as a collection of *tasks*. Each task T_i is characterized by a *WCET* e_i and a *minimum inter-arrival time* or *period* p_i, and releases a *job* for execution at most once every p_i time units.

Such a job J is *preempted* if its execution is temporarily paused before it is completed, *e.g.*, in favor of another job with higher priority. Suppose J is preempted at time t_p on processor P and resumes execution at time t_r on processor R. J is said to have incurred a *preemption* if $P = R$, and a *migration* otherwise. In either case, we call $t_r - t_p$ the *preemption length*. A job may be preempted multiple times.

Scheduling. Let m denote the number of processors. There are two fundamental approaches to scheduling sporadic tasks on multiprocessors [15]: with *global* scheduling, processors are scheduled by selecting jobs from a single, shared queue, whereas with *partitioned* scheduling, each processor has a private queue and is scheduled independently using a uniprocessor scheduling policy. *Clustered* scheduling [4, 13] is a generalization of both approaches: tasks are partitioned onto m/c clusters of c processors each, which are then scheduled globally (with respect to the processors in each cluster). We use the *earliest-deadline-first* (EDF) policy in each category, *i.e.*, in this paper, we consider *partitioned EDF* (P-EDF, $c = 1$), *clustered EDF* (C-EDF, $1 < c < m$), and *global EDF* (G-EDF, $c = m$).

Caches. Modern processors employ a hierarchy of fast *cache memories* that contain recently-accessed instructions and operands to alleviate high off-chip memory latencies. Caches are organized in layers (or levels), where the fastest (and usually smallest) caches are denoted *level-1* (L1) caches, with deeper caches (L2, L3, *etc.*) being successively larger and slower. A cache contains either instructions or data, and may contain both if it is *unified*. In multiprocessors, *shared* caches serve multiple processors, in contrast to *private* caches, which serve only one.

Caches operate on blocks of consecutive addresses called *cache lines* with common sizes ranging from 8 to 128 bytes. In *direct mapped* caches, each cache line may only reside in one specific location in the cache. In *fully associative* caches, each cache line may reside at any location in the cache. In practice, most caches are *set associative*, wherein each line may reside at a fixed number of locations.

The set of cache lines accessed by a job is called the *working set* (WS) of the job; workloads are often characterized by their *working set sizes* (WSSs). A cache line present in a cache is *useful* if it is going to be accessed again. If a job references a cache line that cannot be found in a level-X cache, then it suffers a *level-X cache miss*. This can occur for several reasons. *Compulsory misses* are triggered the first time a cache line is referenced. *Capacity misses* result if the WSS of the job exceeds the size of the cache. Further, in direct mapped and set associative caches, *conflict misses* arise if useful cache lines were evicted to accommodate mapping constraints of other cache lines. A shared cache must exceed the combined WS of all jobs accessing it, otherwise, frequent capacity and conflict misses may arise due to *cache interference*. Jobs that incur frequent level-X capacity and conflict misses even if executing in isolation are said to be *thrashing* the level-X cache.

Cache affinity describes the effect that a job's overall cache miss rate tends to decrease with increasing execution time (unless it thrashes all cache levels)—after an initial burst of compulsory misses, most useful cache lines have been brought into a cache and do not cause further misses. This explains CPD: when a job resumes execution after a preemption, it is likely to suffer additional capacity and conflict misses as the cache was perturbed [27]. Migrations may further cause affinity for some levels to be lost completely (depending on cache sharing), thus adding compulsory misses to the penalty.

A job's memory references are *cache-warm* after cache affinity has been established; conversely, *cache-cold* references imply a lack of cache affinity.

In this paper, we restrict our focus to *cache-consistent* shared-memory machines: when updating a cache line that is present in multiple caches, inconsistencies are avoided by a *cache consistency protocol*, which either invalidates outdated copies or propagates the new value.

Schedulability. In a *hard real-time system*, each job must complete by its specified deadline, whereas bounded deadline tardiness is permissible in a *soft real-time system* [19]. In the design of a real-time system, a validation procedure—or *schedulability test*—must be used to determine *a priori* whether all timing constraints will be met.

As discussed in Sec. 1, current WCET analysis is limited to yield bounds assuming non-preemptive execution [42, 43]. Hence, schedulability tests must be augmented to reflect system overheads such as CPMD [6, 11, 19, 22, 28, 34]. In particular, under each of the aforementioned EDF variants, it is sufficient to inflate each task's execution cost by the maximum delay caused by one preemption or migration [19, 28]. Formally, let D_c denote a bound on the maximum CPMD incurred by any job, and let, for each task T_i, $e'_i = e_i + D_c$ denote the inflated execution cost: all timing constraints will be met if the task system passes a schedulability test assuming an execution cost of e'_i for each T_i [19, 28]. Generally speaking, CPMD can be factored into execution costs using similar scheduler-specific formulas as long as the maximum number of preemptions incurred or caused by a job can be bounded.

In practice, additional delay sources such as scheduling overheads [8, 9, 14] and interrupt interference [8, 10] must also be taken into account using similar methods, but such

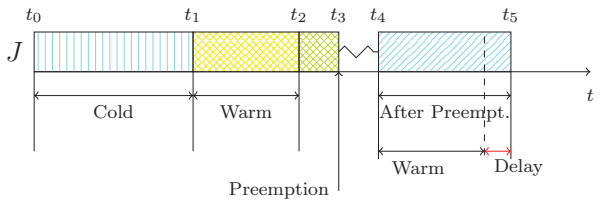


Figure 1: Cache-delay measurement.

considerations are beyond the scope of this paper—our focus is to empirically approximate D_c in real systems.

3 Measuring Cache-Related Delays

Recall that a job is delayed after a preemption or a migration due to a (partial) loss of cache affinity. To measure such delays, we consider jobs that access their WS as illustrated in Fig. 1: a job J starts executing cache-cold at time t_0 and experiences compulsory misses until time t_1, when its WS is completely loaded into cache. After t_1, each subsequent memory reference by J is cache-warm. At time t_2, the job has successfully referenced its *entire* WS in a cache-warm context. From t_2 onward, the job repeatedly accesses single words of its WS (to maintain cache affinity) and checks after each access if a preemption or migration has occurred. Suppose that the job is preempted at time t_3 and not scheduled until time t_4. As J lost cache affinity during the interval $[t_3, t_4]$, the length of the interval $[t_4, t_5]$ (*i.e.*, the time needed to reference again its *entire* WS) reflects the time lost to additional cache misses.

Let d_c denote the cache-related delay suffered by J. After the WS has been fully accessed for the third time (at time t_5), d_c is given by the difference $d_c = (t_5 - t_4) - (t_2 - t_1)$.[1] After collecting a trace $d_{c,0}, d_{c,1}, \ldots, d_{c,k}$ from a sufficiently large number of jobs k, $\max_l\{d_{c,l}\}$ can be used to approximate D_c (recall that D_c is a bound on the maximum CPMD incurred by any job). Similarly, average delay and standard deviation can be readily computed during off-line analysis.

On multiprocessors with a hierarchy of shared caches, migrations are categorized according to the level of cache affinity that is preserved (*e.g.*, a job migration between two processors sharing an L2 cache is an *L2-migration*). A *memory migration* does not preserve any level of cache affinity. Migrations can be identified by recording at time t_3 the processor P on which J was executing and at time t_4 the processor R on which J resumes execution.

Each sample $d_{c,l}$ can be obtained either directly or indirectly. A low-overhead clock device can be used to directly measure the WS access times $[t_1, t_2]$ and $[t_4, t_5]$, which immediately yield d_c. Alternatively, some platforms include *hardware performance counters* that can be used to indirectly measure d_c by recording the number of cache misses. The number of cache misses experienced in each interval is then multiplied by the time needed to service a single cache miss. In this paper, we focus on the direct measure of WS access times, as reliable and precise clock devices are present on virtually all (embedded) platforms. In contrast, the availability of suitable performance counters varies greatly among platforms.

Cache-related preemption and migration delays clearly depend on the WSS of a job and possibly on the scheduling policy [9] and on the task set size (TSS). Hence, to detect such dependencies (if any), each trace $d_{c,0}, d_{c,1}, \ldots, d_{c,k}$ should ideally be collected on-line, *i.e.*, as part of a task set that is executing under the scheduler that is being evaluated *without altering the implemented policy*. We next describe a method that realizes this idea.

3.1 Schedule-Sensitive Method

With this method, d_c samples are recorded on-line while scheduling a proper task set under the algorithm of interest. Performing these measurements without changing the regular scheduling of a task set poses the question of how to efficiently distinguish between a cold, warm, and post-preemption (or migration — *post-pm*) WS access. In particular, detecting a post-pm WS access is subtle, as jobs running under OSs with address space separation (*e.g.*, Linux) are generally not aware of being preempted or migrated. Solving this issue requires a low-overhead mechanism that allows the kernel to inform a job of every preemption and migration. Note that the schedule-sensitive method crucially depends on the presence of such a mechanism (a suitable implementation is presented in Sec. 3.3 below).

Delays should be recorded by executing test cases with a wide range of TSSs and WSSs. This likely results in traces with a variable number of valid samples. To obtain an unbiased estimator for the maximum delay, the same number of samples should be used in the analysis of each trace. In practice, this implies that only (the first) k_{min} from each trace can be used, where k_{min} is the minimum number of valid samples among all traces.

Since samples are collected from a valid schedule, the advantage of this method is that it can identify dependencies (if any) of CPMD on scheduling decisions and on the number of tasks. However, this implies that it is not possible to control *when* a preemption or a migration will happen, since these decisions depend exclusively on the scheduling algorithm (which is not altered). Therefore, the vast majority of the collected samples are likely invalid, *e.g.*, a job may not be preempted at all or may be preempted prematurely, and only samples from jobs that execute exactly as shown in Fig. 1 can be used in the analysis. Thus, large traces are required to obtain few samples. Worse, for a given scheduling algorithm, not all combinations of WSS and TSS may be

[1] The interval $[t_2, t_3]$ is not reflected in d_c since jobs are simply waiting to be preempted while maintaining cache affinity during this interval.

able to produce the execution pattern needed in the analysis (*e.g.*, this is the case with G-EDF, as discussed in Sec. 4).

Hence, we developed a second method that achieves finer control over the measurement process by artificially triggering preemptions and migrations of a single task.

3.2 Synthetic Method

In this approach, CPMD measures are collected by a single task that repeatedly accesses working sets of different sizes. The task is assigned the highest priority and therefore it cannot be preempted by other tasks.

In contrast to the schedule-sensitive method, preemptions and migrations are explicitly triggered in the synthetic method. In particular, the destination core and the preemption length are chosen randomly (preemptions arise if the same core is chosen twice in a row). In order to trigger preemptions, L2-migrations, L3-migrations, *etc.* with the same frequency (and thus to obtain an equal number of samples), proper probabilities must be assigned to each core. Furthermore, as the task execution is tightly controlled, post-pm WS accesses do not need to be detected, and no kernel interaction is needed.

The synthetic method avoids the major drawback of the previous approach, as it generates only valid post-pm data samples. This allows a statistically meaningful number of samples to be obtained rapidly. However, as preemption and migration scheduling decision are externally imposed, this methodology cannot determine possible dependencies of CPMD on scheduling decisions or on the TSS.

3.3 Implementation Concerns

Both methods were implemented using LITMUS$^{\mathrm{RT}}$, a real-time Linux extension developed at UNC [14]. The current version of LITMUS$^{\mathrm{RT}}$ is based on Linux 2.6.32.

Precise time measures of WS access times were obtained on the x86 Intel platform used in our experiments by means of the *time-stamp counter* (TSC), a per-core counter that can be used as high-resolution clock device. The direct measure of CPMD on a multiprocessor platform should take into account the imperfect alignment of per-processor clock devices (*clock skew*). Clock skew errors can be avoided if WS access times are evaluated only based on samples obtained on the same processor (*e.g.*, in Fig. 1, t_1 and t_2 should be measured on the same processor, which may differ from the processor where t_4 and t_5 are measured). In most OSs, time interval measurements can be further perturbed by interrupt handling. These disturbances can be avoided by disabling interrupts while measuring WS access times. Although this does not prevent *non-maskable interrupts* (NMIs) from being serviced, NMIs are infrequent events that likely only have a minor impact on CPMD approximations. We note, however, that our methodology currently cannot detect interference from NMIs.

Disabling interrupts under the schedule-sensitive method is a tradeoff between accuracy and the rate at which samples are collected. On the one hand, disabling interrupts increases the number of valid samples, but on the other hand, it implicitly alters the scheduling policy by introducing non-preemptive sections. We chose to disable interrupts to reduce the length of the experiments.

Within LITMUS$^{\mathrm{RT}}$, we implemented the low-overhead kernelspace–userspace communication mechanism required by the schedule-sensitive method by sharing a single per-task memory page (the *control page*) between the kernel and each task. A task can infer whether it has been preempted or migrated based on the control page: when it is selected for execution, the kernel updates the task's control page by increasing a preemption counter and the job sequence number, storing the preemption length, and recording on which core the task will start its execution.

4 Case Study

To verify and compare results of the two presented methods, we measured cache-related preemption and migration delays using both methodologies on an Intel Xeon L7455. The L7455 is a 24-core 64-bit uniform memory access (UMA) machine with four physical sockets. Each socket contains six cores running at 2.13 GHz. All cores in a socket share a unified 12-way set associative 12 MB L3 cache, while groups of two cores each share a unified 12-way set associative 3 MB L2 cache. Every core also includes an 8-way set associative 32 KB L1 data cache and an identical L1 instruction cache. All caches have a line size of 64 bytes.

4.1 Experimental Setup

We used the G-EDF algorithm to measure CPMD with the schedule-sensitive method, but we emphasize that the method can be applied to other algorithms as well. For this method, we measured the system behavior of periodic task sets consisting of 25 to 250 tasks in steps of variable sizes (from 20 to 30, with smaller steps where we desired a higher resolution). Task WSSs were varied over $\{4, 32, 64, \ldots, 2048\}$ KB. Per-WSS write ratios of 1/2 and 1/4 were assessed.[2] For each WSS and TSS, we measured ten randomly-generated task sets using parameter ranges from [8, 9]. Each task set was traced for 60 seconds and each experiment was carried out once in an otherwise idle system and once in a system loaded with best-effort cache-polluter tasks. Each of these tasks was statically assigned to a core and continuously thrashed the L1, L2, and L3 caches

[2] In preliminary tests with different write ratios, 1/2 and 1/4 showed the highest worst-case overheads, with 1/4 performing slightly worse. All write ratios are given with respect to individual words, not cache lines. There are eight words in each cache line, thus each task updated every cache line in its WS multiple times. Tests with write ratios lower than 1/8, under which some cache lines are only read, exhibited reduced overheads.

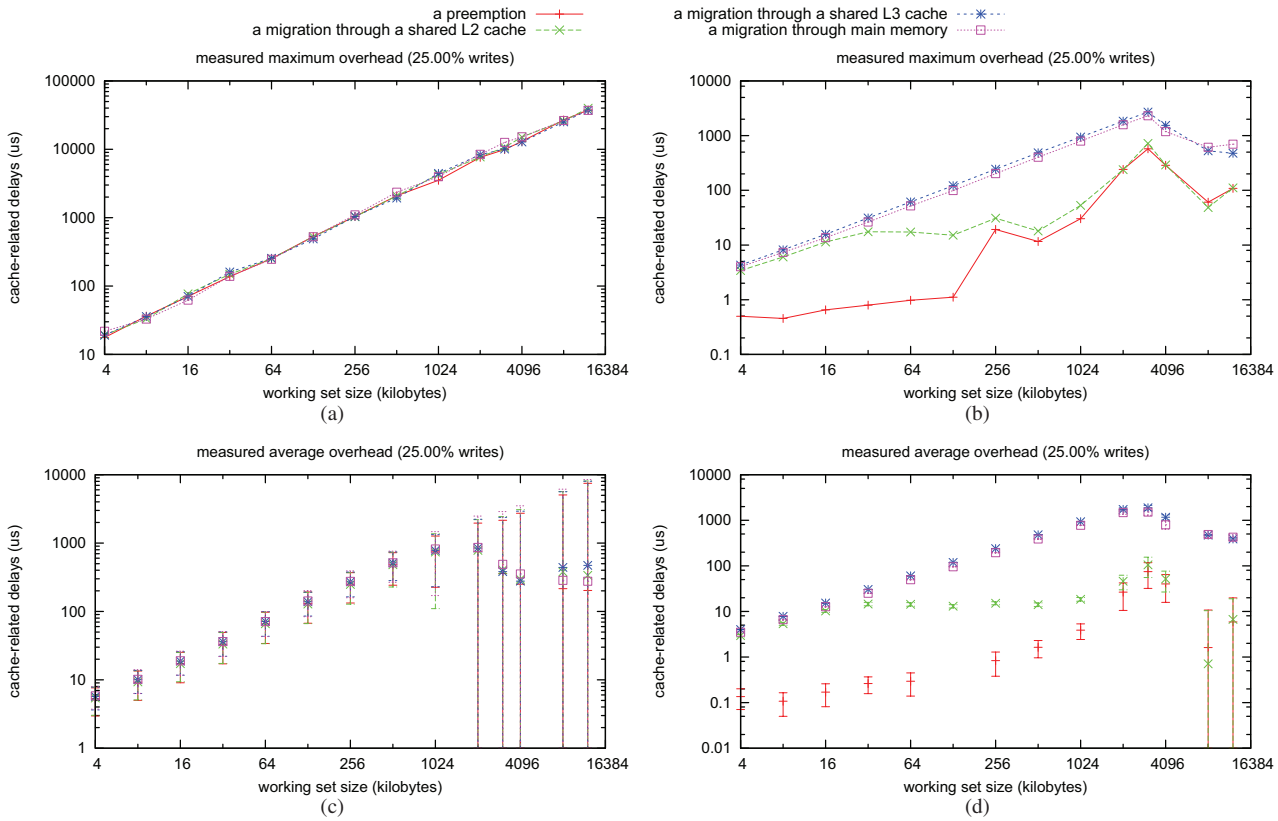


Figure 2: CPMD approximations obtained with the synthetic method. The graphs show maximum and average CPMD (in μs) for preemptions and different types of migrations as a function of WSS (in KB). **(a)** Worst-case delay under load. **(b)** Worst-case delay in an idle system. **(c)** Average-case delay under load. **(d)** Average-case delay in an idle system. The error bars indicate one standard deviation.

by accessing large arrays. In total, more than 50 GB of trace data with 600 million overhead samples were obtained during more than 24 hours of tracing.

We used a single `SCHED_FIFO` task running at the highest priority to measure CPMD with the synthetic method. The WSS was chosen from $\{4, 8, 16, \ldots, 8192\}$ KB. We further tested WSSs of 3 and 12 MB, as they correspond to the sizes of the L2 and L3 cache respectively. In these experiments, several per-WSS write ratios were used. In particular, we considered write ratios ranging over $\{0, 1/128, 1/64, 1/16, 1/4, 1/2, 1\}$. For each WSS we ran the test program until 5,000 valid after-pm samples were collected (for each preemption/migration category). Preemption lengths were uniformly distributed in $[0ms, 50ms]$. As with the schedule-sensitive method, experiments were repeated in an idle system and in a system loaded with best-effort cache-polluter tasks. More than 3.5 million *valid* samples were obtained during more than 50 hours of tracing.

4.2 Results

Fig. 2 shows preemption and migration delays that were measured using the synthetic method (the data is given numerically in Appendix A). Each inset indicates CPMD values for preemptions and all different kinds of migrations (L2, L3, memory) as a function of WSS, assuming a write ratio of 1/4. The first column of the figure (insets (a,c)) gives delays obtained when the system was loaded with cache-polluter tasks, while the second column (insets (b,d)) gives results that were recorded in an otherwise idle system. The first row of the figure presents worst-case overheads, and the second row shows average overheads; the error bars depict one standard deviation. Both axes are in logarithmic scale. Note that these graphs display the difference between a post-pm and and a cache-warm WS access. Declining trends with increasing WSSs (insets (b,c,d)) thus indicate that the cache-warm WS access cost is increasing more rapidly than the post-pm WS access.

Observation 1. The predictability of overhead measures is heavily influenced by the size of L1 and L2 caches. This can be seen in inset (c): as the WSS approaches the size of the L2 cache (3072 KB, shared among 2 cores), the standard deviation of average delays becomes very large (the same magnitude of the measure itself) and therefore overhead estimates are very imprecise. This unpredictability arises because jobs with large WSSs suffer frequent L2- and L3-cache misses in a system under load due to thrashing and cache interference, and thus become exposed to memory bus contention. Due to the thrashing cache-polluter tasks, bus access times are highly unpredictable and L3 cache interference is very pronounced. In fact, our traces show that

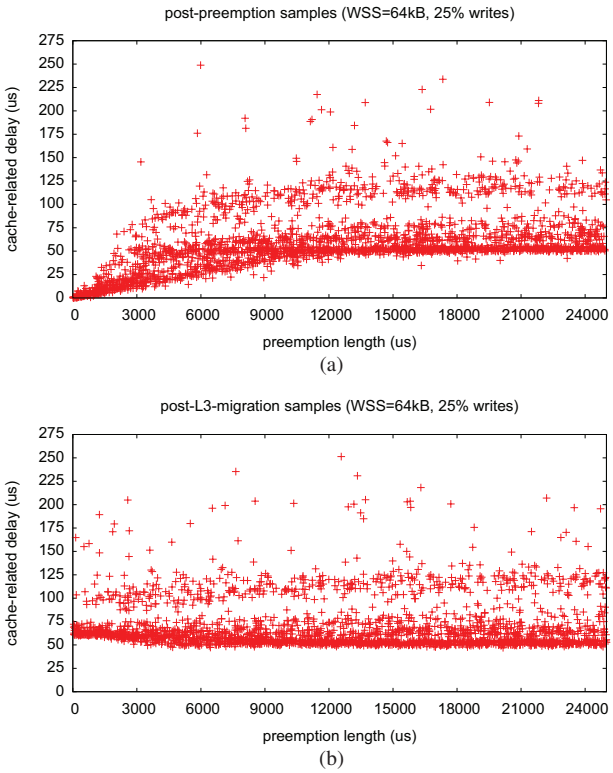


Figure 3: Scatter plot of observed d_c samples vs. preemption length in a system under load. **(a)** Samples recorded after a preemption. **(b)** Samples recorded after an L3-migration. The plots have been truncated at $25ms$; there are no trends apparent in the range from $25ms$ to $50ms$.

jobs frequently incur "negative CPMD" in such cases because the "cache-warm" access itself is strongly interfered with. This implies that, from the point of view of schedulability analysis, CPMD is not well-defined for such WSSs, since a true WCET must account for worst-case cache interference and thus is already more pessimistic than CPMD, *i.e.*, actual CPMD effects are likely negligible compared to the required bounds on worst-case interference.

Observation 2. In a system under load, there are *no substantial differences* between preemption and migration costs, both in the case of worst-case (inset (a)) and average-case (inset (c)) delays. When a job is preempted or migrated in the presence of heavy background activity, its cache lines are likely evicted quickly from all caches and thus virtually every post-pm access reflects the high overhead of refetching the entire WS from memory. Inset (a) shows that, in a system under load, the worst-case delay for a 256 KB WSS exceeds $1ms$, while the cost for a 1024 KB WSS is around $5ms$. Average-case delays (inset (c)) are much lower, but still around $1ms$ for a 1024 KB WSS.

Observation 3. In an idle system, preemptions always cause less delay than migrations, whereas L3- and memory migrations have comparable costs. This behavior can be observed in insets (b,d). In particular, if the WS fits into the L1 cache (32 KB), then preemptions are negligible (around $1\mu s$), while they have a cost that is comparable with that of an L2 migration when the WSS approaches the size of the L2 cache (still, they remain less than $1ms$). Inset (d) clearly shows that L2-migrations cause less delay than L3-migrations for WSSs that exceed the L1-cache size (about $10\mu s$ for WSSs between 32 and 1024 KB). In contrast, L3- and memory migrations have comparable costs, with a maximum around $3ms$ with 3072 KB WSS (inset (b)). Interestingly, memory migrations cause slightly less delay than L3 cache migrations. As detailed below, this is most likely related to the cache consistency protocol.

Observation 4. The magnitude of CPMD is strongly related to preemption length (unless cache affinity is lost completely, *i.e.*, in the case of memory migrations). This trend is apparent from the plots displayed in Fig. 3. Inset (a) shows individual preemption delay measurements arranged by increasing preemption length, inset (b) similarly shows L3-migration delay. The samples were collected using the synthetic method with a 64 KB WSS and a write ratio of 1/4 in a system under load (similar trends were observed with all WSSs $\leq$ 3072 KB). In both insets, CPMD converges to around $50\mu s$ for preemption lengths exceeding $10ms$. This value is the delay experienced by a job when its WSS is reloaded entirely from memory.[3] In contrast, for preemption lengths ranging in $[0ms, 10ms]$, average preemption delay increases with preemption length (inset (a)), while L3-migrations (in the range $[0ms, 5ms]$) progressively decrease in magnitude (inset (b)). The observed L3-migration trend is due to the cache consistency protocol: if a job resumes quickly after being migrated, parts of its WS are still present in previously-used caches and thus need to be evicted. In fact, if the job does not update its WS (*i.e.*, if the write ratio is 0), then the trend is not present.

Observation 5. Preemption and migration delays do not depend significantly on the task set size. This can be observed in Fig. 4, which depicts worst-case delay for the schedule-sensitive method in a system under load as function of the TSS. The plot indicates CPMD for preemptions and all migration types for WSSs of 1024, 512 and 256 KB (from top to bottom).

Note that Fig. 4 is restricted to TSSs from 75 to 250 because, under G-EDF, only few task migrations occur for small TSSs. Thus, the number of collected valid delays for small TSSs is not statistically meaningful.

Furthermore, Fig. 4 shows that worst-case preemption and migrations delays for the same WSS have comparable magnitudes, thus confirming that, in a system under load, preemption and migration costs do not differ substantially (recall Fig. 2(a) and Observation 2).

[3] Due to space limitations, plots for memory and L2-migrations are not shown. L2-migrations reveal a trend that is similar to the preemption case, while memory migrations do not show a trend (samples are clustered around $50\mu s$ delay regardless of preemption length).

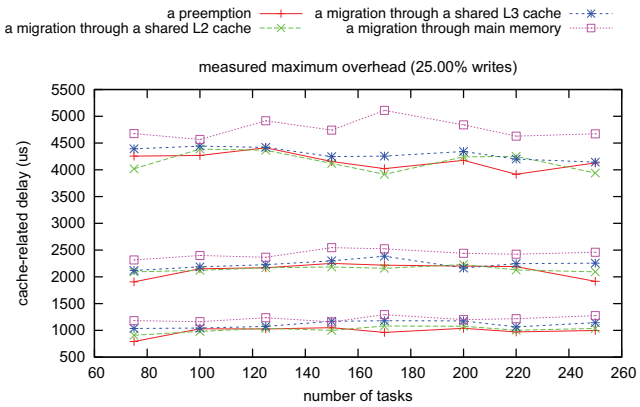


Figure 4: Worst-case CPMD approximations as function of TSS in a system under load (obtained with the schedule-sensitive method). Lines are grouped by WSS: from top: WSS = 1024 KB, WSS = 512 KB, WSS = 256 KB.

Interpretation. The setup used in the experiments depicted in Fig. 2(a,c) simulate *worst-case scenarios* in which a job is preempted by a higher-priority job with a large WSS that (almost) completely evicts the preempted job's WS while activity on other processors generates significant memory bus contention. In contrast, insets (b,d) correspond to situations in which the preempting job does not cause many evictions (which is the case if it has a virtually empty WS or its WS is already cached) and the rest of the system is idle, *i.e.*, insets (b,d) depict *best-case scenarios*. Hence, Fig. 2(a) (resp., Fig. 2(c)) shows the observed worst-case (resp., average) cost of reestablishing cache affinity in a worst-case situation, whereas Fig. 2(b) (resp., Fig. 2(d)) shows the worst-case (resp., average) cost of reestablishing cache affinity in a best-case situation.

Further note that, even though the synthetic method relies on a background workload to generate memory bus contention, the data shown in Fig. 2(a,c) also applies to scenarios in which the background workload is absent if the real-time workload itself generates significant memory bus contention.

This has profound implications for empirical comparisons of schedulers. If it is possible that a job's WS is completely evicted by an "unlucky" preemption, then this (possibly unlikely) event must be reflected in the employed schedulability test(s). Thus, unless it can be shown (or assumed) that *all* tasks have only small WSSs and there is *no* background workload (including background OS activity), then bounds on CPMD should be estimated based on the high-contention scenario depicted in Fig. 2(a,c).

Therefore, based on our data, it is *not warranted* to consider migrations to be more costly than preemptions when making worst-case assumptions (*e.g.*, when applying hard real-time schedulability tests). Further, unless memory bus contention is guaranteed to be absent, this is the case even when using average case overheads (*e.g.*, when applying soft real-time schedulability tests).

5 Impact on Schedulability

The *schedulability* of an algorithm with respect to a given scenario is the fraction of task sets that can be shown to meet their timing constraints. It estimates the probability that a randomly chosen task set can be scheduled and is a commonly employed method to compare scheduling algorithms. For example, Baker [3] and Bertogna *et al.* [7] studied schedulability as a function of system load to assess various global and partitioned schedulers. As argued in Sec. 1, schedulability studies should account for CPMD. However, given that CPMD strongly depends on the WSS, assuming any specific value for D_c introduces a bias on the corresponding specific WSS. In this section, we describe a method to integrate CPMD bounds into schedulability studies that overcomes this limitation. We start by discussing the setup previously used in [8, 9, 10, 14], which in turn is based on an earlier design by Baker [3], and then present the corresponding WSS-agnostic extension.

Task set generation. A schedulability study is based on a parametrized task set generation procedure. Said procedure is used to repeatedly create (and test) task sets while varying the parameters over their respective domains. This enables the schedulability under each of the tested algorithms to be evaluated as a function of the task set generation procedure's parameters.

Recall that a task T_i is defined by its WCET e_i and period p_i. A task's *utilization* $u_i = e_i/p_i$ reflects its required processor share; a task set's *total utilization* is given by $\sum_i u_i$. Our generation procedure depends on three parameters: a probability distribution for choosing u_i, a probability distribution for choosing p_i, and a *utilization cap* U.

Tasks are created by choosing u_i and p_i from their respective distributions and computing e_i. A task set is generated by creating tasks until the total utilization exceeds U and by then discarding the last-added task (unless U is reached exactly). Discarding the last task ensures that all parameters stem from their respective distributions, but the total utilization of the resulting task set may be less than U. Alternatively, the last-added task's utilization can be scaled such that U is reached exactly. This procedure lends itself to studying schedulability as a function of system load by varying U from zero to m while assuming fixed choices for utilization and period distributions (*e.g.*, see [8, 9]).

Accounting for overheads. Task execution costs are commonly inflated to accomodate overheads caused by scheduling decisions, context switches, timer ticks, job releases, and other OS activity during the execution of one job [8, 9, 10, 14, 28]. Such system overheads must be accounted for after a task set has been generated, since most overheads are TSS-dependent [8, 9, 14].

This is in stark contrast to CPMD, which our experiments revealed to be independent of TSS, as discussed in Sec. 4 (Observation 5). Instead, bounding CPMD requires knowledge of a task's WSS. Thus, either a specific WSS must be assumed throughout the study, or a WSS must be chosen randomly during task set generation. Anticipating realistic WSS distributions is a non-trivial challenge, hence prior studies [8, 9, 14] focused on selected WSSs.

Implicit WSS. Instead, CPMD should be an additional parameter of the task set generation procedure, thus removing the need for WSS assumptions. In this *WSS-agnostic setup*, schedulability (*i.e.*, the ratio of task sets deemed schedulable for given parameters) is a function of two variables (U and D_c) and can therefore be studied assuming a wide range of values for D_c (and thus WSS).

While conceptually simple and appealing due to the avoidance of a WSS bias, this setup poses some practical problems. Besides squaring the number of required samples, a "literal" plotting of the results requires a 3D projection, which renders the results virtually impossible to interpret (schedulability plots routinely show four to eight individual curves, *e.g.*, [3, 8, 9]). To overcome this, we propose the following aggregate performance metric instead.

Weighted schedulability. Let $S(U, D_c) \in [0, 1]$ denote the schedulability for a given U and D_c under the WSS-agnostic setup, and let Q denote a set of evenly-spaced utilization caps (*e.g.*, $Q = \{1.0, 1.1, 1.2, \ldots, m\}$). Then *weighted schedulability* $W(D_c)$ is defined as

$$W(D_c) = \frac{\sum_{U \in Q} U \cdot S(U, D_c)}{\sum_{U \in Q} U}.$$

This metric reduces the obtained results to a two-dimensional (and thus easier to interpret) plot without introducing a fixed utilization cap. Weighting individual schedulability results by U reflects the intuition that high-utilization task systems have higher "value" since they are more difficult to schedule. Note that $W(0) = 1$ for an optimal scheduler (if other overheads are negligible).

Weighted schedulability offers the great benefit of clearly exposing the *range of CPMDs* in which a particular scheduler is competitive. Recall from Sec. 1 that global schedulers are provably superior if CPMD is negligible, but not so if migrations are costly. At which point does partitioning become the superior choice? This can be inferred from the weighted schedulability, as is demonstrated next.

Example. Fig. 5 shows $W(D_c)$ assuming soft timing constraints for four schedulers on our 24-core experimental platform: G-EDF, P-EDF, and two C-EDF configurations with clusters of two (resp., six) processors each chosen based on L2 (resp., L3) cache sharing. Task parameters are uniformly distributed, with $u_i \in [0.5, 0.9]$ and $p_i \in [3\text{ms}, 33\text{ms}]$. This workload is of particular interest since smooth video playback and interactive games fall in this range of periods, and high per-task utilizations can easily result from high-definition multimedia processing.

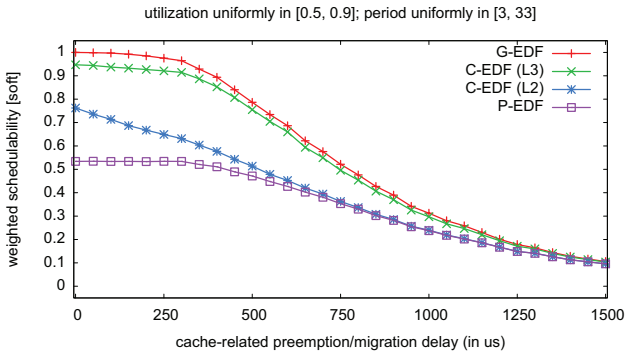


Figure 5: Weighted soft schedulability as a function of CPMD.

Fig. 5 clearly reveals the tradeoff between bin-packing limitations under P-EDF and migration costs under G-EDF. Let D_c^G (resp., D_c^P) denote a bound on CPMD under G-EDF (resp., P-EDF), and let W^G (resp., W^P) denote weighted schedulability under G-EDF (resp., P-EDF). In Fig. 5, the G-EDF curve dominates the P-EDF curve. This implies that G-EDF is a superior choice assuming equal CPMD *i.e.*, if $D_c^G = D_c^P$ then $W^G(D_c^G) \geq W^P(D_c^P)$.

More interesting is the case in which migrations are costly, *i.e.*, $D_c^G > D_c^P$. Suppose preemptions are negligible, *e.g.*, $D_c^P = 0\mu s$: at which point does P-EDF become preferable to G-EDF? The curve for P-EDF reveals that $W^P(0) \approx 0.55$; by tracing G-EDF's curve we find that weighted schedulability under G-EDF drops below 0.55 at $D_c^G \approx 725\mu s$. Thus, G-EDF is preferable to P-EDF, *i.e.*, $W^G(D_c^G) > W^P(D_c^P)$, if $D_c^G < 725\mu s$.

These results should be combined with the actual observed CPMD: under P-EDF, jobs incur only preemptions, whereas they may incur both migrations and preemptions under G-EDF. In a system under load, Fig. 2(a,c) reveals that $D_c^G \approx D_c^P$ in both the average and the worst case, and thus G-EDF is preferable for any WSS. In contrast, in an idle system, $D_c^G > D_c^P$, but, as shown in Fig. 2(b,d), D_c^G does not exceed $725\mu s$ for WSSs smaller than 1024 KB, and thus G-EDF is preferable for such WSSs.

This illustrates that weighted schedulability, in combination with actual CPMD measurements, can reveal interesting tradeoffs between schedulers that cannot be inferred from overhead-oblivious schedulability studies.

6 Conclusion

We have presented two methods for measuring CPMD: the schedule-sensitive method can detect scheduler-dependent cache-related delays since it does not alter the scheduling policy, while the synthetic method rapidly produces large numbers of samples by artificially triggering preemptions and migrations. We have discussed strengths and weaknesses of the two approaches, and have demonstrated their

efficacy by reporting average and maximum CPMD for various WSSs on a 24-core Intel UMA machine with two layers of shared caches. Our findings show that, on our platform, CPMD in a system under load is only predictable for WSSs that do not thrash the L2 cache. We further observed that preemption and migration delays did not differ significantly under load, which calls into question the widespread belief that migrations are necessarily more costly than preemptions. In particular, our data indicates that (on our platform) preemptions and migrations differ only little in terms of both worst-case and average-case CPMD if cache affinity is lost completely in the presence of either a background workload or other real-time tasks with large WSSs. Additionally, our experiments showed that incurred CPMD depends on preemption length, but not on task set size.

We have further proposed a method for incorporating CPMD bounds into large-scale schedulability studies without biasing results towards a particular WSS choice. Based on weighted schedulability, this method allows regions to be identified in which a particular scheduler is competitive.

Limitations. Since our methods are based on empirical measurements, they cannot be used to derive safe bounds on true worst-case delays. Further, our current implementation focuses on data caches (but could be extended to apply to instruction caches) and cannot detect if samples were disturbed by the processing of non-maskable interrupts. Nonetheless, we believe that our approach offers a good tradeoff between experimental complexity and accuracy, and hope that it will enable CPMD to routinely be considered in future evaluations of multiprocessor schedulers.

Future work. We plan to validate our experiments by substituting the TSC with performance counters to directly measure cache misses. Further, we would like to apply our measurement methodology to embedded and NUMA platforms. Repeating these experiments in the presence of frequent DMA transfers by I/O devices and atomic (*i.e.*, bus-locking) instructions could yield further insights. Based on the observed trends, further research into bounds on maximum per-task preemption lengths and non-preemptive global schedulers is warranted.

Acknowledgement. We thank Alex Mills for his valuable and helpful suggestions regarding the data analysis.

A CPMD Data

The CPMD data corresponding to the graphs shown in Fig. 2 is given in Tables 1–4.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208, 2005.

[3] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, Florida State University, 2005.

[4] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*. Chapman Hall/CRC, 2007.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[6] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1994.

[7] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, 2005.

[8] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224, 2009.

[9] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169, 2008.

[10] B. Brandenburg, H. Leontyev, and J. Anderson. Accounting for interrupts in multiprocessor real-time systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 273–283, 2009.

[11] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 204–219, 1996.

[12] J. Calandrino. *On the Design and Implementation of a Cache-Aware Soft Real-Time Scheduler for Multicore Platforms*. PhD thesis, University of North Carolina at Chapel Hill, 2009.

[13] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256, 2007.

[14] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, 2006.

[15] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

[16] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[17] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 101–110, 2006.

[18] F. M. David, J. C. Carlyle, and R. H. Campbell. Context switch overheads for Linux on ARM platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[19] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, North Carolina, 2006.

[20] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the 7th ACM International Conference on Embedded Software*, pages 245–254, 2009.

WSS (KB)	Preemption	Migrat. through L2	Migrat. through L3	Migrat. through Mem.
4	17.52	19.09	18.94	21.58
8	35.98	32.89	35.48	32.55
16	69.76	76.13	69.73	61.71
32	136.16	147.49	159.10	137.55
64	248.86	248.82	252.63	244.07
128	525.08	520.77	484.50	520.55
256	1,027.77	1,020.08	1,031.80	1,088.35
512	2,073.41	2,064.59	1,914.32	2,333.64
1,024	3,485.44	4,241.11	4,408.33	3,935.43
2,048	7,559.04	7,656.31	8,256.06	8,375.53
3,072	9,816.22	10,604.52	9,968.44	12,491.07
4,096	12,936.70	14,948.87	12,635.93	15,078.12
8,192	26,577.31	25,760.44	24,923.14	26,091.24
12,288	37,139.30	39,559.55	36,923.48	36,688.75

Table 1: CPMD data. Worst-case delay (in μs) in a system under load. This table corresponds to Fig. 2(a).

WSS (KB)	Preemption	Migrat. through L2	Migrat. through L3	Migrat. through Mem.
4	0.49	3.38	4.27	3.98
8	0.45	5.99	8.08	7.27
16	0.65	11.22	15.53	13.50
32	0.79	17.28	31.01	26.10
64	0.97	17.15	60.91	51.33
128	1.10	14.95	120.47	98.25
256	19.05	30.60	241.68	199.54
512	11.46	17.88	481.52	397.67
1024	30.03	52.63	935.29	784.89
2048	239.45	235.94	1,819.50	1,567.65
3072	567.54	713.33	2,675.71	2,287.87
4096	283.65	288.22	1,523.32	1,169.90
8192	60.23	47.90	522.20	606.40
12288	107.68	109.94	472.15	690.81

Table 2: CPMD data. Worst-case delay (in μs) in an idle system. This table corresponds to Fig. 2(b).

[21] D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proceedings of the 17th International Conference on Real-Time and Network Systems*, pages 45–54, Paris France, 2009.

[22] L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *Proceedings of the 2007 Conference on Design, Automation and Test in Europe*, pages 1623–1628, 2007.

[23] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, 2001.

[24] H. Leontyev. *Compositional Analysis Techniques For Multiprocessor Soft Real-Time Scheduling*. PhD thesis, University of North Carolina at Chapel Hill, 2010.

[25] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[26] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, January 1973.

[27] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker. Characterizing and modeling the behavior of context switch misses. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–101, 2008.

[28] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[29] J. C. Mogul and A. Borg. The effect of context switches on cache performance. *ACM SIGPLAN Notices*, 26(4):75–84, 1991.

[30] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 201–206, 2003.

[31] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, to appear, 2008.

[32] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 80–89, 2009.

WSS (KB)	Preemption		Migrat. through L2		Migrat. through L3		Migrat. through Mem.	
	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
4	5.24	2.31	5.37	2.36	5.66	2.07	5.77	2.08
8	9.14	4.18	9.24	4.21	9.93	3.69	10.09	3.82
16	17.02	8.04	17.05	7.77	18.58	7.00	18.78	7.18
32	32.88	15.94	32.93	15.73	35.82	13.96	35.99	14.30
64	65.05	31.38	65.33	31.90	70.72	27.87	70.50	28.02
128	128.64	62.08	127.09	61.22	137.35	52.48	141.05	58.20
256	248.81	117.10	246.34	119.89	267.73	106.19	272.56	115.37
512	478.45	239.08	476.95	251.41	507.27	227.87	509.18	245.06
1024	739.20	515.18	733.27	624.26	772.68	544.80	810.37	641.08
2048	740.10	1,200.93	773.22	1,409.55	837.53	1,373.93	853.27	1,605.60
3072	355.76	1,781.11	400.88	2,021.39	377.96	1,974.79	483.20	2,373.09
4096	247.88	2,456.97	291.93	2,756.90	274.51	2,622.08	350.07	3,118.26
8192	212.90	4,793.77	374.45	5,230.35	436.19	5,153.05	282.28	5,797.23
12288	201.20	7,211.18	333.80	7,683.50	467.50	7,485.35	274.23	8,122.10

Table 3: CPMD data. Average-case delay (in μs) in a system under load. This table corresponds to Fig. 2(c).

WSS (KB)	Preemption		Migrat. through L2		Migrat. through L3		Migrat. through Mem.	
	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
4	0.13	0.06	2.91	0.18	3.98	0.06	3.48	0.12
8	0.11	0.06	5.31	0.39	7.75	0.07	6.54	0.15
16	0.17	0.09	10.19	0.72	15.17	0.10	12.62	0.24
32	0.26	0.10	14.41	1.24	30.11	0.14	24.82	0.38
64	0.29	0.15	14.21	1.29	59.79	0.22	49.18	0.72
128	-0.17	0.30	12.89	1.08	118.23	0.47	94.60	1.40
256	0.83	0.45	15.02	1.14	236.94	1.43	192.42	2.88
512	1.62	0.66	13.96	1.30	477.22	1.69	384.35	4.48
1024	3.85	1.45	18.28	1.70	921.61	4.79	769.80	7.93
2048	26.21	15.80	45.61	16.03	1,721.14	130.92	1,459.52	120.58
3072	74.04	42.28	104.26	49.47	1,867.61	246.62	1,501.23	229.21
4096	39.66	23.96	51.29	24.75	1,156.36	153.95	790.41	157.98
8192	1.60	9.06	0.70	9.60	468.26	14.14	482.73	66.05
12288	5.84	13.85	6.62	12.16	385.62	24.14	420.76	57.21

Table 4: CPMD data. Average-case delay (in μs) in an idle system. This table corresponds to Fig. 2(d).

[33] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

[34] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[35] G. Stamatescu, M. Deubzer, J. Mottok, and D. Popescu. Migration overhead in multiprocessor scheduling. In *Proceedings of the 2nd Embedded Software Engineering Conference*, pages 645–654, 2009.

[36] J. Stärner and L. Asplund. Measuring the cache interference cost in preemptive real-time systems. *ACM SIGPLAN Notices*, 39(7):146–154, 2004.

[37] J. Staschulat and R. Ernst. Scalable precision cache analysis for real-time software. *ACM Transactions on Embedded Computing Systems*, 6(4):25, 2007.

[38] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303, 2008.

[39] D. Thiebaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[40] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[41] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *Performance Evaluation: Origins and Directions (LNCS 1769)*, pages 97–139, 2000.

[42] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[43] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.

Implementation of Overrun and Skipping in VxWorks

Mikael Åsberg, Moris Behnam and Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
{mikael.asberg,moris.behnam,thomas.nolte}@mdh.se

Reinder J. Bril
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5612 AZ Eindhoven
The Netherlands
r.j.bril@tue.nl

***Abstract*—In this paper we present our work towards allowing for dependence among partitions in the context of hierarchical scheduling of software systems with real-time requirements, and we present two techniques for cross-partition synchronization. We have earlier developed a Hierarchical Scheduling Framework (HSF) in VxWorks for independent real-time tasks, and in this paper we extend the HSF implementation with capabilities of synchronization between tasks resident in two different partitions. In particular, we have implemented the overrun and skipping mechanisms in our modular scheduling framework. Our framework has a key characteristic of being implemented on top of the operating system, i.e., no modifications are made to the kernel. Such a requirement enforce some restrictions on what can be made with respect to the implementation. The evaluation performed indicates that, under the restrictions of not modifying the kernel, the skipping mechanism has a much lower implementation overhead compared to the overrun mechanism[1].**

I. Introduction

Advanced operating system mechanisms such as hierarchical scheduling frameworks provide temporal and spatial isolation through virtual platforms, thereby providing mechanisms simplifying development of complex embedded software systems. Such a system can now be divided into several modules, here denoted subsystems, each performing a specific well defined function. Development and verification of subsystems can ideally be performed independently (and concurrently) and their seamless and effortless integration results in a correctly functioning final product, both from a functional as well as extra-functional point of view.

In recent years, support for temporal partitioning has been developed for several operating systems. However, existing implementations typically assume independence among software applications executing in different partitions. We have developed such a modular scheduling framework for VxWorks without modifying any of its kernel source code. Our scheduling framework is implemented as a layer on top of the kernel. Up until now, this scheduling framework required that tasks executing in one subsystem must be independent of tasks executing in other subsystems, i.e., no task-level synchronization was allowed across subsystems. In this paper we present our work on implementing synchronization protocols for our hierarchical scheduling framework, allowing for task-level synchronization across subsystems. We implemented the synchronization protocols in VxWorks, however, they can naturally be extended to other operating systems as well. We are considering, in this paper, a two level hierarchical scheduling framework (as shown in Figure 1) where both the local and global schedulers schedule subsystems/tasks according to the fixed priority preemptive scheduling (FPS) policy.

The contributions of this paper are the descriptions of how the skipping and overrun mechanisms are implemented in the context of hierarchical scheduling without modifying the kernel. The gain in not altering the kernel is that it does not require any re-compilation, there is no need to maintain/apply kernel modifications when the kernel is updated/replaced and kernel stability is maintained. We have evaluated the two approaches and results indicate that, given the restriction of not being allowed to modify the kernel, the overhead of the skipping mechanism is much lower than the overhead of the overrun mechanism.

The outline of this paper is as follows: Section II gives an overview of preliminaries simplifying the understanding of this paper. Section III presents details concerning the implementation of the skipping and overrun mechanisms. Section IV presents an evaluation of the two methods, Section V presents related work, and finally Section VI concludes the paper together with outlining some future work.

II. Preliminaries

This section presents some preliminaries simplifying the presentation of the rest of the paper. Here we give an overview of our hierarchical scheduling framework (HSF) followed by details concerning the stack resource policy (SRP) protocol and the overrun and skipping mechanisms for synchronization, in the context of hierarchical scheduling.

A. HSF

The Hierarchical Scheduling Framework (HSF) enables hierarchical scheduling of tasks with real-time constraints. In [1] we assume that tasks are periodic and independent, and we use periodic servers to implement subsystems. The HSF is implemented as a two layered scheduling framework

[1]The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

as illustrated in Figure 1, where the schedulers support FPS and EDF scheduling.

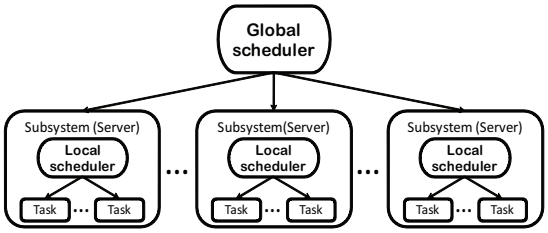


Fig. 1. HSF structure

Both schedulers (local and global) are activated periodically according to task/server parameters and a one-shot timer is used to trigger the schedulers. The next triggering (absolute) time of the tasks/servers are stored in a *Time Event Queue* (TEQ). The TEQ is essentially a priority queue, storing the release times (in absolute values) of tasks/servers. The input to the one-shot timer is a value derived by subtracting the shortest time in the TEQs from the current absolute time (since the timer input should be in relative time). Three TEQs can be active at once, the TEQ holding server release times, the current active servers TEQ for task release times and a TEQ (with one node) holding the current active servers budget expiration time. The current absolute time is updated only at a scheduler invocation, i.e., when the one-shot timer is set, we also set the absolute time equal to the next triggering time. When the next event arrives, the current absolute time will match the $real$ time. It is important to note that if we would like to invoke our scheduler $before$ the event arrives, then the current absolute time will not be correct. This fact needs to be taken into account when implementing synchronization protocols in our framework. The triggering of the global and local schedulers are illustrated in Figure 2. The $Handler$ is responsible for deriving the next triggering event (could be task or server related). Depending on which kind of event, i.e., task activation, server activation or budget expiration, the $Handler$ will either call the $Global\ scheduler$ or the $Local\ scheduler$. The $Global\ scheduler$ will call $Local\ scheduler$ in case of server activation (there might be task activations that have not been handled when the server was inactive). The VxWorks scheduler is responsible for switching tasks in the case when a task has finished its execution. The VxWorks scheduler will be invoked after an interrupt handler has executed (i.e., after $Handler$ has finished), but only if there has been any change to the ready queue that will affect the task scheduling.

All servers, that are ready, are added to a server ready queue and the global scheduler always selects the highest priority server to execute (depends also on the chosen global scheduling algorithm). When a server is selected, all tasks that are ready, and that belong to that subsystem, are added to the VxWorks task ready queue and the highest priority ready task is selected to execute.

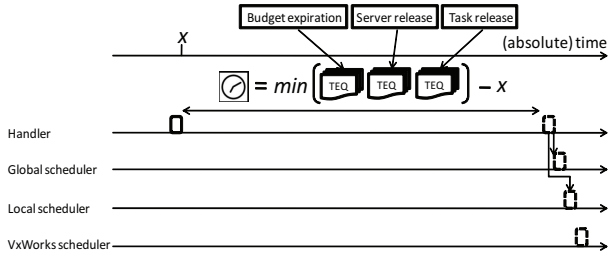


Fig. 2. Scheduler triggering

B. Shared resources in HSF

The presented HSF allows for sharing of logical resources between arbitrary tasks, located in arbitrary subsystems, in a mutually exclusive manner. To access a resource, a task must first lock the resource, and when the task no longer needs the resource, it is unlocked. The time during which a task holds a lock is called a critical section. For each logical resource, at any time, only a single task may hold its lock. A resource that is used by tasks in more than one subsystem is denoted a *global shared resource*. A resource only used within a single subsystem is denoted a *local shared resource*. In this paper, both local and global shared resources are managed by the SRP protocol. This protocol has the strength that it can be used with different scheduling algorithms such as FPS and EDF scheduling, which are supported by HSF at both global and local scheduling level.

1) Stack resource policy (SRP): To be able to use SRP in a HSF for synchronizing global shared resources, its associated terms resource, system and subsystem ceilings are extended as follows:

- **Resource ceiling:** Each global shared resource is associated with two types of resource ceilings; an *internal* resource ceiling for local scheduling and an *external* resource ceiling for global scheduling. They are defined as the priority of the highest priority task/subsystem that access this resource.
- **System/subsystem ceiling:** The system/subsystem ceilings are dynamic parameters that change during execution. The system/subsystem ceiling is equal to the highest external/internal resource ceiling of a currently locked resource in the system/subsystem.

Under SRP, a task τ_k can preempt the currently executing task τ_i (even inside a critical section) within the same subsystem, only if the priority of τ_k is greater than its corresponding subsystem ceiling. The same reasoning can be made for subsystems from a global scheduling point of view. The problem that SRP solves (synchronization of access to shared resources without deadlock) can arise at two completely different levels, due to that subsystems share resources and because tasks (within a subsystem) share resources. That is why SRP is needed at both local and global level, and also the reason why a global resource has a local and global ceiling.

2) Mechanisms to handle budget expiry while executing within a critical section: To bound the waiting time of tasks from different subsystems that want to access the same shared

resource, subsystem budget expiration should be prevented while locking a global shared resource. The following two mechanisms can be used to solve this problem:

- **The overrun mechanism:** The problem of subsystem budget expiry inside a critical section is handled by adding extra resources to the budget of each subsystem to prevent the budget expiration inside a critical section. Hierarchical Stack Resource Policy (HSRP) [2] is based on an overrun mechanism. HSRP stops task preemption within the subsystem whenever a task is accessing a global shared resource. SRP is used at the global level to synchronize the execution of subsystems that have tasks accessing global shared resources. Two versions of overrun mechanisms have been presented; 1) *with payback*; whenever overrun happens in a subsystem S_s, the budget of the subsystem will, in its next execution instant, be decreased by the amount of the overrun time. 2) *without payback*; no further actions will be taken after the event of an overrun.
- **The skipping mechanism:** Skipping is another mechanism that prevent a task from locking a shared resource by skipping (postpone the locking of the resource) its execution if its subsystem does not have enough remaining budget at the time when the task tries to lock the resource. Subsystem Integration and Resource Allocation Policy (SIRAP) [3] is based on the skipping mechanism. SIRAP uses the SRP protocol to synchronize the access to global shared resources in both local and global scheduling. SIRAP checks the remaining budget before granting the access to the globally shared resources; if there is sufficient remaining budget then the task enters the critical section, and if there is insufficient remaining budget, the local scheduler delays the critical section entering of the job until the next subsystem budget replenishment (assuming that the subsystem budget in the next subsystem budget replenishment is enough to access the global shared resource by the task). The delay is done by blocking that task that want to access the resource (self blocking) during the current server period and setting the local ceiling equal to the value of internal resource ceiling of the resource that that task wanted to access.

Scheduling analysis of both of these two mechanisms can be found in [2] respectively [3].

III. Implementation

This section compares and discusses some issues related to the implementation of the skipping and overrun mechanisms. These implementations are based on our previous implementation of the Hierarchical Scheduling Framework (HSF) [1] in the VxWorks operating system. To support synchronization between tasks (or subsystems) when accessing global shared resources, advances in the implementation of VxWorks made since [1] does not include the implementation of the SRP protocol, and SRP is used by both both skipping and overrun mechanisms. Therefore, our implementation of the SRP protocol is outlined below.

A. Local synchronization mechanism

Since both skipping and overrun depend on the synchronization protocol SRP, which is not implemented in VxWorks, we have implemented this protocol ourselves. The implementation of SRP is part of our previous VxWorks implementation (HSF), hence, this SRP implementation is adjusted to fit with hierarchical scheduling. We added two queues to the server TCB, see Figure 3. Whenever a task wants to access a locally shared resource (within a subsystem), it calls a corresponding `SrpLock` function (Figure 4). When the resource access is finished, it must call `SrpUnlock` (Figure 5).

```
1: struct SERVER_TCB {
2:       // Resource queue, sorted by ceiling
3:       queue SRP_RESOURCES;
4:       // Blocked tasks, sorted by priority/preempt. level
5:       queue SRP_TASK_BLOCKED_QUEUE;
6:       /* The rest of the server TCB */
```

Fig. 3. Data-structures used by SRP

```
1: void SrpLock (int local_res_id) {
2:       InterruptDisable( );
3:       LocalResourceStackInsert(local_res_id); // Ceiling is updated
4:       InterruptEnable( );
5: }
```

Fig. 4. Lock function for SRP

```
1: void SrpUnlock (int local_res_id) {
2:       InterruptDisable( );
3:       LocalResourceStackRemove(local_res_id); // Ceiling is updated
4:       if (LocalCeilingHasChanged( ))
5:          MoveTasksFromBlockedToReady(RunningServer);
6:       NewTask = GetHighestPrioReadyTask( );
7:       if (RunningTask.ID ≠ NewTask.ID)
8:          RunningServer.LocalScheduler( );
9:       InterruptEnable( );
10: }
```

Fig. 5. Unlock function for SRP

Lines (3, 5, 8) in Figure 5 are specific to each server, since they have their own task ready-, blocked- and resource-queue (stack), and a local scheduler. The same goes for line (3) in Figure 4. Note that `SrpUnlock` is executed at task-level (user-mode). Hence, we start the local scheduler by generating an interrupt that is connected to it. When our local scheduler (which is part of an interrupt handler) has finished, the VxWorks scheduler will be triggered if a context switch should occur. This is illustrated in Figure 6, where we use the VxWorks system call `sysBusIntGen` to generate an interrupt which will trigger the corresponding connected handler, which in this case is our local scheduler.

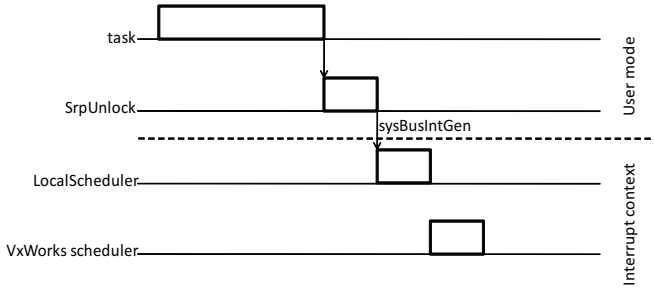


Fig. 6. Local scheduler invocation

The only modification made in our local scheduler is that it compares the local system ceiling against the task priority, before releasing a task (putting it in the task ready queue).

B. Global synchronization mechanisms

It is important to note that from the user perspective, there is no difference when locking a local or global resource, since all global resources are mapped to one corresponding local resource. When calling a `lock` function that implements a global synchronization protocol (i.e., overrun or skipping), the only information needed is the local resource ID. From this, we can derive the global resource ID. Hence, the global synchronization protocol calls the local synchronization protocol (i.e., SRP in this case), and, it also implements the global synchronization strategy, i.e., skipping or overrun in this case. Both of them need to use a local synchronization protocol, other than that, skipping is the only protocol of the two that need direct access to the local system, i.e., the local scheduler. The reason for this is covered in section III-E.

To support the synchronization mechanisms, additional queues are required in the system level (resource queue and blocked queue) to save all global resources that are in use, and to save the blocked servers. Similar queues are required for each subsystem (covered in section III-A) to save the local resources that are in use within the subsystem, and to save the blocked tasks. The resource queues are sorted, by the resource ceilings, hence, the first node represents the system ceiling (there is one (local) system ceiling per server and one (global) system ceiling). The resource queues are mapped as outlined in Figure 7.

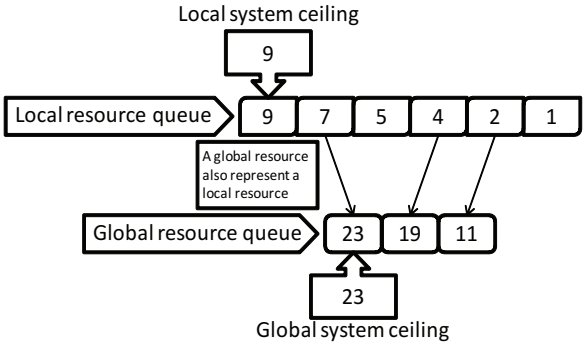


Fig. 7. Resource queue mapping

When a task wants to access a global shared resource, it uses the `lock` function, and when the task wants to release the resource, it calls the function `unlock`. The implementation of `lock` and `unlock` depends on the type of synchronization approach (overrun or skipping). In general, `lock` and `unlock` change some parameters that are used by the scheduler, e.g., system/subsystem ceiling, server/task ready queue, and server/task blocked queue. When a server/task is activated, the local/global schedulers checks weather the server/task has a higher priority than the current system/subsystem ceiling. If yes, then the server/task is added to the ready queue, otherwise the server/task will be added to the blocked queue. When the `unlock` function is called, all tasks and servers that were blocked, by the currently released shared resource, should be moved from the server/task blocked queue to the ready queue, and then the scheduler should be called to reschedule the subsystems and tasks. For this reason, it is very important that the `lock`/`unlock` functions should have mutual exclusion with respect to the scheduler, to protect the shared data-structures. In this implementation, interrupt disable in the `lock`/`unlock` function has been used to protect shared data-structures, noting that the interrupt disable time should be very short.

Since the scheduler can be triggered by the `unlock` functions (unlike the implementation in [1]), the current absolute time for this event should be calculated by subtracting a current timestamp value with the timestamp from the latest scheduler invocation and adding this value to the latest evaluated absolute time. The difference in time between the *real* current absolute time and the calculated one is the drift caused by both the skipping and overrun mechanisms. More of this is discussed in the next section.

C. Time drift

Budgets and time-triggered periodic tasks are implemented using a one-shot timer [1], which may give rise to relative jitter [4] due to inaccuracies caused by time calculations, setting the timer, and activities that temporarily disable interrupts. Relative jitter (or drift) may give rise to severe problems whenever the behavior of the system needs to remain synchronized with its environment. In the implementation used in this paper, such explicit synchronization requirements is not assumed, however. Implementation induced relative jitter can therefore be accommodated in the analysis as long as the jitter can be bound. By assuming a maximum relative jitter for every time the timer is set, and a maximum number of times the timer is set for a given interval, the relative jitter can be bound for periods of both budgets and time-triggered tasks and for capacities of budgets. Now the worst-case analysis can be adapted by making worst-case assumptions, i.e., by using (a) maximal inter-arrival times for periods and minimal capacities for budgets and (b) minimal-inter-arrival times (and worst-case computation times) of tasks. For the two types of synchronization protocols discussed in this paper, i.e., overrun with (or without) payback and skipping, the impact of relative jitter is similar.

D. Overrun mechanism implementation

Besides the data-structures needed for keeping track of global system ceiling, line (1) in Figure 8, and the queue of blocked servers, line (2) in Figure 8, overrun also need data-structures to keep track of when an overrun has occurred, line (5,7) in Figure 8.

```
1: queue GLOBAL_RESOURCES; // Used by Overrun
2: queue SERVER_BLOCKED_QUEUE; // Used by Overrun
3: struct SERVER_TCB {
4:         // Nr of global resources that are locked
5:         char nr_global_resources_locked;
6:         // Flag for keeping track if an overrun has occurred
7:         char overrun;
8:         /* The rest of the server TCB */
```

Fig. 8. Data-structures used by Overrun

Figure 9 shows the `OverrunLock` function for the overrun mechanism. The resource that is accessed is inserted in both the global and local resource queue which are sorted by the node's resource ceilings.

```
1: void OverrunLock (int local_res_id) {
2:         SrpLock(local_res_id);
3:         InterruptDisable( );
4:         GlobalResourceStackInsert(local_res_id); // Ceiling is updated
5:         RunningServer.nr_global_resources_locked++;
6:         InterruptEnable( );
7: }
```

Fig. 9. Lock function for Overrun

In line (5) in Figure 9, the function increment `RunningServer.nr_global_resources_locked` by one, which indicate the number of shared resources that are in use. This is important for the scheduler so it does not terminate the server execution at the budget expiration. When the budget of a server expires, the scheduler checks this value. If it is greater than 0 then it does not remove the server from the server ready queue and it sets the budget expiration event equal to X_s, which means that the server is overrunning its budget (i.e., there will not be a scheduler event until `OverrunUnlock` is called). Also, the scheduler indicates that the server is in overrun state by setting the `overrun` flag, line (7) in Figure 8, to true. Otherwise, the scheduler removes the server from the server ready queue.

Figure 10 shows the `OverrunUnlock` function. In this function, the released resource is removed from both the local and global resource queues and the system and subsystem ceilings are updated, which may decrease them. If the system/subsystem ceiling is decreased, the function checks if there are servers/tasks in the blocked queue that are blocked by this shared resource. It will move them to the server/task ready queues, depending on their preemption levels and the system/subsystem ceilings. In line (9) in Figure 10, the function checks if it should call the global scheduler, and there

```
1: void OverrunUnlock (int local_res_id) {
2:         SrpUnlock(local_res_id);
3:         InterruptDisable( );
4:         GlobalResourceStackRemove(local_res_id); // Ceiling is updated
5:         if (GlobalCeilingHasChanged( ))
6:           MoveServersFromBlockedToReady( );
7:         RunningServer.nr_global_resources_locked–;
8:         NewServer = GetHighestPrioReadyServer( );
9:         if ((RunningServer.overrun == TRUE &&
10:            RunningServer.nr_global_resources_locked == 0) ||
11:            RunningServer.ID ≠ NewServer.ID)
12:                GlobalScheduler( );
13:        InterruptEnable( );
14: }
```

Fig. 10. Unlock function for Overrun

are two cases to do this. The first case is when the server was in overrun state, then it should be removed from the ready queue. The second case is if the server, after releasing the resource, is not the highest priority server, then it will be preempted by another server. The global scheduler will be invoked through the `sysBusIntGen` system call, similar to the local scheduler in the SRP implementation. The reason is that there will be a task switch (so the VxWorks scheduler needs to be invoked), and of course also a server switch, but this can be handled without the help of the VxWorks scheduler. The global/local scheduler (in HSF) must have knowledge about the current absolute time in order to set the next scheduling event, so this time must be derived before calling the scheduler.

At every new subsystem activation, the server checks if there has been an overrun in its previous instance. If so, this overrun time length is subtracted from the servers budget, in the case of using overrun with payback mechanism. The global scheduler measures the overrun time when it is called, in response to budget expiration, and when it is called in response to the unlock function. If the other version of overrun is used (ONP), then the budget of the subsystem does not change.

On all server activations, the preemption level of each server is checked against current system ceiling. If the preemption level is lower than ceiling, then the server is inserted in the blocked queue.

E. Skipping mechanism implementation

The skipping implementation uses the same data-structures as overrun for keeping track of system ceiling and blocked servers. What is further needed, in order to implement skipping, is a simple FIFO (First In First Out) queue for tasks, line (8) in Figure 11. Also, a post in the VxWorks task TCB is needed, line (2) in Figure 11. According to the SIRAP protocol, the time length of the critical section must be known (and therefore also stored) so that it can be compared against the remaining budget, in order to prevent the budget from overrunning. One disadvantage with our current implementation is that we only allow maximum one shared global resource per task. This implementation can easily be extended to support

more than one global resource per task, by adding more data-structures to store the locking times of the resources.

```
1: /* The rest of struct WIND_TCB (VxWorks TCB) */
2:       int spare4; // We keep resource locking time here
3: };
4: queue GLOBAL_RESOURCES; // Used by Skipping
5: queue SERVER_BLOCKED_QUEUE; // Used by Skipping
6: struct SERVER_TCB {
7:       // Used by Skipping to queue tasks during self-blocking
8:       queue TASK_FIFO_QUEUE;
9:       /* The rest of the server TCB */
```

Fig. 11. Data-structures used by Skipping

When calling the `SkippingLock` function (Figure (12), it checks if the remaining budget is enough to lock and release the shared resource before the budget expires (line (4) in Figure (12)). If the remaining budget is sufficient, then the resource will be inserted in both the global and local resource queue, similar to the overrun mechanism mentioned earlier. If the remaining budget is not sufficient, then the resource will be inserted in the local resource queue and the local system ceiling is updated, finally, the task is suspended in line (12) in Figure (12). Note that the rest of the function, lines (13-17), will not be executed until this task is moved to the ready queue. When the task is executed next time, it will continue from line (13) and insert the shared resource (line (14)) in the global resource queue, then update the global system ceiling and finally start executing in the critical section. Whenever a server starts to execute, after it has been released, its local scheduler checks if there are tasks that are suspended (by checking the `TASK_FIFO_QUEUE`), if any, it moves them (in FIFO order) to the ready queue. In this way, skipping affects the local scheduler while overrun does not.

```
1: void SkippingLock (local_res_id) {
2:       InterruptDisable( );
3:       RemainBudget = CalcRemainBudget(RunningServer);
4:       if (RemainBudget ≥ RunningTask.spare4) {
5:          GlobalResourceStackInsert(local_res_id); // Ceiling is updated
6:          SrpLock(local_res_id);
7:       }
8:       else { // Budget is not enough, block the task
9:          SrpLock(local_res_id);
10:         BlockedQueueInsert(RunningTask);
11:         InterruptEnable( );
12:         TaskSuspend(RunningTask); // This call will block...
13:         InterruptDisable( ); // ...cont. here when task is awakened
14:         GlobalResourceStackInsert(local_res_id); // Ceiling is updated
15:      }
16:      InterruptEnable( );
17: }
```

Fig. 12. Lock function for Skipping

The `SkippingUnlock` function is similar to the `OverrunUnlock` function (Figure 10), but with two differences. The first one is that skipping does not need to keep count of the number of locked global resources, and second, skipping will call the scheduler only if there is a server in the ready queue that has higher priority than the currently running server. In case of nested critical sections, the task call `SkippingLock/SkippingUnlock` functions only when it access and release the outermost shared resource, and the ceiling of the outermost shared resource equals to the highest ceiling of the nested shared resources.

```
1: void SkippingUnlock (int local_res_id) {
2:       SrpUnlock(local_res_id);
3:       InterruptDisable( );
4:       GlobalResourceStackRemove(local_res_id); // Ceiling is updated
5:       if (GlobalCeilingHasChanged( ))
6:          MoveServersFromBlockedToReady( );
7:       NewServer = GetHighestPrioReadyServer( );
8:       if (RunningServer.ID != NewServer.ID)
9:          GlobalScheduler( );
10:      InterruptEnable( );
11: }
```

Fig. 13. Unlock function for Skipping

If the global system ceiling has changed then the servers, for which preemption level is higher than global system ceiling, are put in the server ready queue. If the new global system ceiling causes a higher priority server to be inserted in the ready queue, then current running server is removed, and the global scheduler is called.

IV. Evaluation

In order to compare the runtime overhead of both synchronization mechanisms, we generated 8 systems according to the setup illustrated in Figure 14. In this setup, a system S^i contains 5 servers with 8 tasks each, and each system has 2 global resources (2-6 tasks will access the global resources). We monitored both skipping and overrun with payback.

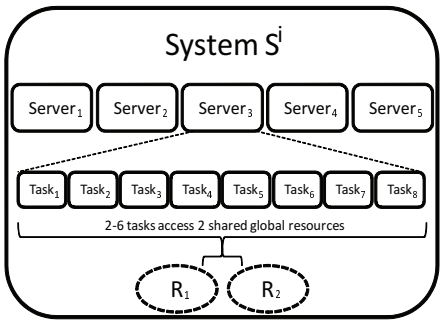


Fig. 14. Experimental setup

The metrics we used are the number of calls to the corresponding lock and unlock functions as well as the number of calls to the scheduler. The measurements were recorded in 600 time units (tu), and the range of tasks periods were scaled from 40 to 100 tu and the range of subsystem periods were 5-20 tu (we scaled the periods of subsystem and tasks in order to remove the effect of scheduling overhead). The task utilization was set to 15% per system.

Protocol	System															
	S^1	S^2	S^3	S^4	S^5	S^6	S^7	S^8	S^1	S^2	S^3	S^4	S^5	S^6	S^7	S^8
	# calls to `lock`/`unlock`								# calls to `Scheduler`							
Skipping	306	335	248	275	181	224	202	236	8	5	7	4	5	5	10	6
Overrun	304	335	247	275	181	225	203	236	47	13	40	16	36	17	30	25

TABLE I
EXPERIMENTAL RESULTS

Table I shows the results of running systems S^1 to S^8. Each of these systems (S^i) had different task/server parameters, different amount of resources and different resource users (depending on the generation of the systems). It is clear that the number of scheduler calls under the skipping mechanism is lower compared to using the overrun mechanism, which makes the runtime overhead for the skipping mechanism lower than the corresponding overhead when using the overrun mechanism. The difference between the corresponding `unlock` functions under skipping and under overrun is also the reason why the number of calls to the scheduler differs. For the overrun mechanism, the unlock function calls to the scheduler when the server unlocks the shared resource after overrun, while there is no such case in skipping, i.e., there is a higher risk that the scheduler is called in overrun, than in skipping (since there is two cases in overrun and one case in skipping). This explains the recorded results with respect to the number of scheduler calls.

V. RELATED WORK

Related work in the area of hierarchical scheduling originated in open systems [5] in the late 1990's, and it has been receiving an increasing research attention [6], [5], [7], [8], [9], [10], [11], [12]. However, the main focus of the research was on the schedulability analysis of independent tasks, and not much work has been conducted on the implementation of the proposed theories.

Among the few implementation work, Kim *et al.* [13] propose the SPIRIT uKernel that is based on a two-level hierarchical scheduling framework simplifying integration of real-time applications. The SPIRIT uKernel provides a separation between real-time applications by using partitions. Each partition executes an application, and uses the Fixed Priority Scheduling (FPS) policy as a local scheduler to schedule the application's tasks. An off-line scheduler (timetable) is used to schedule the partitions (the applications) on a global level. Each partition provides kernel services for its application and the execution is in user mode to provide stronger protection. Parkinson [14] uses the same principle and describes the VxWorks 653 operating system which was designed to support ARINC653. The architecture of VxWorks 653 is based on partitions, where a Module OS provides global resource and scheduling for partitions and a Partition OS implemented using VxWorks microkernel provides scheduling for application tasks.

The work presented in this paper differs from the above last two works in the sense that it implements a hierarchical scheduling framework in a commercial operating system without changing the OS kernel.

The implementation of a HSF in VxWorks without changing the kernel has been presented in [1] assuming the tasks are independent. In this paper, we extend this implementation by enabling sharing of logical resources among tasks located in the same and/or different subsystem(s). More recently, [15] implemented a two-level fixed priority scheduled HSF based on a timed event management system in the commercial real-time operating system μC/OS-II, however, the implementation is based on changing the kernel of the operating system, unlike the implementation in this paper.

In order to allow for dependencies among tasks, many theoretical works on synchronization protocols have been introduced for arbitrating accesses to shared logical resources, addressing the priority inversion problem, e.g., the Stack Resource Policy (SRP) [16]. For usage in a HSF, additional protocols have been proposed, e.g., the Hierarchical Stack Resource Policy (HSRP) [2], the Subsystem Integration and Resource Allocation Policy (SIRAP) [3], and the Bounded-delay Resource Open Environment (BROE) [17] protocols. The work in this paper concerns the former two, targeting systems implementing FPPS schedulers.

VI. CONCLUSION

In this paper we have presented our work on implementing synchronization protocols for hierarchical scheduling of tasks without doing any modification to the operating system kernel. We have presented two techniques for synchronization; overrun and skipping, and we have implemented the two techniques in our hierarchical scheduling framework for VxWorks [1]. The evaluation of these two techniques indicates that, when the synchronization protocol is implemented, skipping requires far less overhead when compared to the overrun mechanism.

Future work includes management of memory and interrupts towards a complete operating system virtualizer implemented as a layer on top of an arbitrary operating system kernel.

REFERENCES

[1] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, July 2008, pp. 63–72.

[2] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS'06)*, December 2006, pp. 257–267.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in *Proceedings of the ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, October 2007, pp. 278–288.

[4] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[5] Z. Deng and J.-S. Liu, "Scheduling real-time applications in an open environment," in 18^{th} *IEEE Int. Real-Time Systems Symposium (RTSS'97)*, Dec. 1997.

[6] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *RTSS'05*, December 2005, pp. 389–398.

[7] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in 23^{th} *IEEE Int. Real-Time Systems Symposium (RTSS'02)*, Dec. 2002.

[8] T.-W. Kuo and C.-H. Li, "A fixed-priority-driven open environment for real-time applications," in 20^{th} *IEEE International Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.

[9] G. Lipari and S. K. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, May-Jun. 2000.

[10] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS'03)*, Jul. 2003.

[11] S. Matic and T. A. Henzinger, "Trading end-to-end latency for composability," in 26^{th} *IEEE International Real-Time Systems Symposium(RTSS'05)*, December 2005, pp. 99–110.

[12] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in 24^{th} *IEEE International Real-Time Systems Symposium (RTSS'03)*, Dec. 2003.

[13] D. Kim, Y. Lee, and M. Younis, "Spirit-ukernel for strongly partitioned real-time systems," in *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, 2000.

[14] L. K. P. Parkinson, "Safety critical software development for integrated modular avionics," in *Wind River white paper. URL http://www.windriver.com/whitepapers/*, 2007.

[15] M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Virtual timers in hierarchical real-time systems," *Proc. WiP session of the RTSS*, pp. 37–40, Dec. 2009.

[16] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, March 1991.

[17] N. Fisher, M. Bertogna, and S. Baruah, "The design of an edf-scheduled resource-sharing open environment," in *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, December 2007, pp. 83–92.

Schedulable Device Drivers: Implementation and Experimental Results

Nicola Manica*, Luca Abeni*, Luigi Palopoli*, Dario Faggioli† and Claudio Scordino‡
*University of Trento
Trento - Italy
Email:{nicola.manica, luca.abeni, luigi.palopoli}@unitn.it
†Scuola Superiore Sant'Anna
Pisa - Italy
Email:d.faggioli@sssup.it
‡Evidence Srl
Pisa - Italy
Email:claudio@evidence.eu.com

***Abstract*—An important issue when designing real-time systems is to control the kernel latencies introduced by device drivers. This result can be achieved by transforming the interrupt handlers into schedulable entities (threads). This paper shows how to schedule such threads (using resource reservations) so that both the performance of real-time tasks and the device throughput can be controlled. In particular, some tools based on a kernel tracer (`Ftrace`) are used to collect timing information about the IRQ threads, and a novel reservation-based scheduler for Linux (`SCHED_DEADLINE`) is used to schedule them. An implementation of the proposed technique is validated through an extensive set of experiments, using different kinds of resources and of realistic applications.**

I. Introduction

One of the prominent issues in the design of modern real-time operating systems is accounting for interference of device drivers on the real-time tasks. Indeed, the interference possibly generated by a device driver (which contributes to the so called "Kernel latencies") introduces some unbounded blocking times that can compromise the schedulability of task sets deemed schedulable by the formal analysis techniques (because the worst case blocking times are unknown, hence it is not possible to account for them in the formal analysis).

A straightforward strategy to address this problem is to give real-time tasks higher priorities than device drivers. However, this is not possible on general purpose systems, where interrupt handlers and device drivers are not schedulable entities (and are executed with a higher priority than all the user-space tasks). For this reason, recent developments in the Linux kernel allow transforming the interrupt handlers (both Interrupt Service Routines — ISRs — and Bottom Halves) into kernel threads, the so called IRQ threads (note that in the past similar solutions have been mainly used in μkernel based systems [1], [2] or in proprietary real-time kernels such as LynxOS). This functionality was originally developed for the Preempt-RT real-time kernel [3], [4], and enables a control on the amount of interference from device drivers suffered by real-time tasks [5], [6], [7].

Once interrupt handlers have been transformed into schedulable entities, the problem remains open of identifying the best scheduling algorithm that can be used to serve the newly introduced threads. For example, Manica et al. [8], [9] have provided a clear evidence that using resource reservations [10] to schedule the interrupt handlers (IRQ threads, in Preempt-RT) allows the designer to find appropriate trade-offs between the response time of real-time tasks and the device throughput (this is important when the device is used by real-time tasks). However, to the best of our knowledge, most experiments and tests with advanced scheduling solutions have been performed only using prototypical schedulers or experimental Operating Systems [11], [12]. Only recently has a Linux scheduler based on resource-reservation has been proposed to the kernel community [13]. Such a scheduler exports an API that can be easily used to schedule kernel threads implementing the device drivers. Additionally, most of the previous work has focused on network devices [14], [15], [16], [5], [9] paying little or no attention to other types of devices (e.g., disks). Finally, another limitation of previous results is that they are mostly collected on artificial task sets.

This paper takes a step forward to show that the results collected with experiments based on prototypical schedulers can be repeated: 1) with a scheduler likely to become main line in the near future, 2) using different kinds of resources, 3) with realistic applications rather than with artificial task sets.

As a last contribution, a set of tools based on the *Ftrace* kernel tracing facility is used to collect the stochastic distribution of the execution time and of the inter-arrival time of device drivers. This way it is possible to apply design techniques that enable an appropriate dimensioning of the scheduling parameters.

The paper is organised as follows: Section II recalls the scheduling algorithm used in this work, briefly describes an implementation of such an algorithm, and explains how to correctly assign the scheduling parameters to IRQ threads; Section III describes the tracing tools used for analysing the scheduler behaviour and to collect information about the IRQ

threads; Section IV presents some experimental results, and Section V concludes the paper.

II. Scheduling the IRQ Threads

This section briefly recalls some basic concepts about resource reservations, and about assigning proper reservation parameters to the interrupt threads. It also introduces the reservation-based scheduler for Linux (named `SCHED_DEADLINE`) that has been used for scheduling the interrupt threads.

A. Reservation-Based Scheduling

The basic idea of reservation-based scheduling is that each task is reserved an amount Q of CPU time (named *maximum budget*) every T time units (T is called *reservation period*). Such a strategy can be implemented by using various scheduling algorithms. The particular reservation algorithm used in this paper is the Constant Bandwidth Server (CBS) [17], which, contrary to different scheduling algorithms of the same kind, is well behaved with both regular and periodic tasks and with aperiodic and dynamically changing tasks.

The CBS algorithm assigns each task a *scheduling deadline*, and schedules processes and threads using an Earliest Deadline First (EDF) policy (i.e., the task with the earliest scheduling deadline is selected first for execution). When a task wakes up, the CBS checks if its current scheduling deadline can be used; otherwise, a new scheduling deadline is generated (as $d = t + T$, where t is the wakeup time). The scheduling deadline is then postponed by T ($d = d + T$) every time that the task executes for Q time units (if having a work conserving algorithm is not important, the task is removed from the runqueue until time $d - T$).

An interesting feature of the reservation-based schedulers is that they provide *temporal isolation* among tasks. This means that the temporal behaviour of a task is not affected by the behaviour of the other tasks in the system: if a task requires a large execution time, it cannot affect the schedulability of the other tasks, or monopolise the processor. This is a basic property needed for scheduling real-time tasks on general-purpose operating systems.

B. The Linux SCHED_DEADLINE Policy

In the recent versions, the official Linux kernel has introduced a new scheduling framework that replaces the old O(1) scheduler. This framework contains an extensible set of *scheduling classes*. Each scheduling class implements a specific algorithm and schedules tasks with a specific policy.

Currently, two scheduling classes are available in the Linux kernel:

- `sched_fair`, which implements the *"Completely Fair Scheduler"* (CFS) algorithm, and schedules tasks having `SCHED_OTHER` or `SCHED_BATCH` policies. Tasks are run at precise weighted speeds, so that each task receives a "fair" amount of processor share.
- `sched_rt`, which implements a POSIX fixed-priority real-time scheduler, and handles tasks having `SCHED_FIFO` or `SCHED_RR` policies.

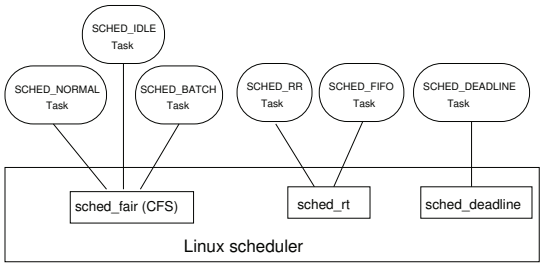


Figure 1. Linux scheduler with SCHED_DEADLINE.

As explained in previous papers [13], using these scheduling policies with tasks characterised by temporal constraints might be problematic, mainly because the standard API used in general purpose kernels like Linux does not allow to associate temporal constraints (e.g., deadlines) to the tasks. In fact, although it allows to assign a share of processor time to a task, there is no way to specify that the task must finish the execution of a job before a given time. Using CFS, moreover, the time elapsed between two consecutive executions of a task is not deterministic and cannot be bound, since it depends on the number of tasks running in the system at that time.

For these reasons, very recently, a new scheduling class based on resource reservations has been implemented and proposed to the kernel community[1]. The project, formerly known as `SCHED_EDF`, changed name to `SCHED_DEADLINE` after the request of the kernel community.

This class adds the possibility of scheduling tasks using the CBS algorithm, without changing the behaviour of tasks scheduled using the existing policies. Figure 1 depicts schematically the Linux scheduler extended with the `SCHED_DEADLINE` scheduling class (note that scheduling classes have increasing priorities from left to right).

The implementation does not make any restrictive assumption on the characteristics of the tasks. Thus, it can handle periodic, sporadic and aperiodic tasks. It is aligned with the current mainstream kernel, and it relies on standard Linux mechanisms to natively support multicore platforms and to provide hierarchical scheduling through a standard API.

a) Main Characteristics of the Implementation: In the implementation, red-black trees are used for ready queues to enable efficient handling of events such as earliest deadline task scheduling, new task activation, task blocking/unblocking, etc. One run-queue per each CPU is used to avoid contention and achieve high scalability even on large systems. Moreover, it is enriched with the following features:

- support for bandwidth reclaiming, to make the scheduler work conserving without affecting guarantees;
- capability of synchronising tasks with the scheduler;
- support for resource sharing similar to priority inheritance (already present in the kernel for fixed priority real-time tasks);

[1] `SCHED_DEADLINE`. The code is open and available at http://gitorious.org/sched_deadline.

- support for standard Linux mechanisms for debugging and tracing the scheduler behaviour and for specifying per-user policies and limitations;
- capability of sending signals to the tasks on budget overruns and scheduling deadline misses;
- support for bandwidth management throughout admission control, both system-wide and for separate groups of tasks.

b) User Level API: An user-level application can exploit the services provided by the `SCHED_DEADLINE` scheduling class by means of some new system calls and a new data structure that accommodates additional scheduling parameters. The new data structure is called `sched_param_ex` and comprises the following fields:

- temporal parameter of the task — i.e., `sched_runtime` and `sched_deadline` which will be Q and T of its reservation, respectively;
- `sched_flag` for controlling some aspects of the scheduler behaviour. More precisely, (i) whether or not a task wants to be notified about budget overruns and/or scheduling deadline misses and (ii) whether or not a task wants to exploit some kind of bandwidth reclaiming;
- some other fields left there for backward compatibility or future extensions (`sched_priority` and `sched_period`).

The most important system calls added are:

- `sched_setscheduler_ex` (and a couple of others), that manipulates `sched_param_ex`;
- `sched_wait_interval` to synchronise the task with the scheduler. This means a task can ask to be put to sleep until either its next deadline or whenever it will be possible to receive its full budget again.

The security model adopted is very similar to the one already in use in the kernel for fixed priority real-time tasks — i.e., it is based on user permissions and capabilities and it can be affected by standard UNIX security mechanism, like `rlimits`. Controls exist for managing the fraction of CPU time usable by the whole EDF scheduler as well as to a group of EDF tasks, but they are not described here for space reasons.

C. Assigning the Reservation Parameters

The reservation parameters (Q, T) can be dimensioned by performing a deterministic or a stochastic analysis of the interrupt behaviour [9]. The deterministic case is simpler to analyse, and allows to dimension the reservation so that no interrupt is lost (at the cost of some overestimation of the reserved CPU time). First of all, if P is the minimum inter-arrival time between two consecutive interrupts and C is the maximum amount of time needed to serve an interrupt, then Q and T must be assigned according to Equation 1

$$\frac{Q}{T} \geq \frac{C}{P} \tag{1}$$

In other words, the fraction of CPU time reserved to the IRQ thread should be greater than or equal to the fraction of CPU time needed by the IRQ thread for executing.

However, some hardware devices have an upper bound N_c on the number of pending interrupts (interrupts that have not been processed yet), and if an interrupt fires when N_c interrupts requests are already pending, then the interrupt is lost even if Equation 1 is respected. As discussed in our previous work [9], this problem ban be addressed by producing the following condition that has to be respected to avoid losing interrupts:

$$\frac{T - Q}{P} < N_c \tag{2}$$

The same paper also presented a stochastic analysis instrumental to the correct dimensioning of the CBS parameters: in this case, instead of considering the worst-case times P and C, the interrupt inter-arrival times and the execution times of the interrupt handlers are modelled as stochastic variables. As a result, the probability to drop an interrupt can be computed.

Both approaches require a precise characterisation of the workload generated by IRQ threads. Hence, the need for the tracing mechanism described in the next section.

III. Inferring the IRQ Parameters

According to Section II-C, if the probability distributions of the inter-arrival and execution times of an IRQ thread are known, then it is possible to schedule such thread with a (Q, T) reservation so that no interrupt is lost (note that if no interrupt is lost then the device can achieve its maximum throughput). Hence, to assign the maximum budget Q and the reservation period T to an IRQ thread it is necessary to know the *IRQ parameters* (that is, the probability distributions of the inter-interrupt times and of the times needed to serve an interrupt).

Such probability distributions can be measured by using the *Ftrace* tracer provided by the Linux kernel and by properly parsing its traces. In the proposed approach, this is done through a set of tools organised in a pipeline, as shown in Figure 2 (more details about the used tracing tools are available in a technical report [18]).

A. The Tracing Pipeline

The kernel traces produced by Ftrace can be used to extract various information about tasks' timings, so that their temporal behaviour can be inferred.

The first stage of the pipeline (the trace parser) transforms the textual traces exported by Ftrace in an internal format, which is used by the other tools in the pipeline. This step is needed because Ftrace exports traces in the form of text files, whose format can change from one kernel version to another, containing redundant and unneeded information (this happens because the Ftrace format has been designed to be easily readable by humans). Hence, the textual traces produced by Ftrace are parsed and transformed in a more compact, kernel-independent, binary format which is used as input by the next stages of the pipeline. Such stages are composed by a second set of tools that can:

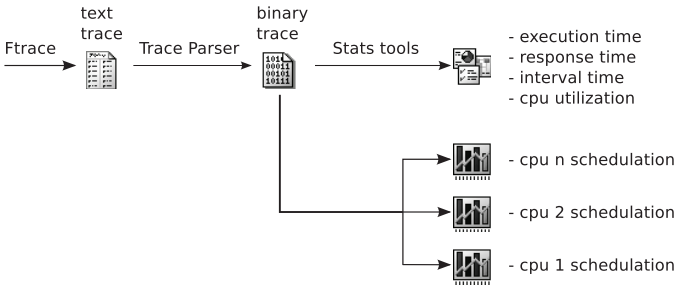


Figure 2. General architecture of the tool

- parse the internal format to gather statistics about execution times, inter-arrival times, response times, and utilisation;
- generate a chart displaying the CPU scheduling;
- infer some of the tasks temporal properties, identifying (for example) periodic tasks.

In this context, the presented tools are used to extract the probability distributions of the execution and inter-arrival times of the IRQ threads.

The various tools composing the pipeline communicate through standard Unix FIFOs (named pipes) and can be combined in different ways, to collect different kinds of information. For example, a tool which periodically displays important statistics for selected tasks (similarly to the standard "`top`" utility) can be inserted into the pipeline. In this work, the collected values are generally saved to files to be processed off-line later, but in other situations they can also be summarised by some statistics that are saved instead of the raw sequence of values, to save some disk space.

Since connecting the different tools in a correctly working pipeline (creating all the needed FIFOs, etc.) can sometimes be difficult, some helper scripts have been developed.

B. Examples

The first possible usage of the proposed tools is to visually analyse the scheduler's behaviour, to check its correctness or to understand the reason for unexpected results. An example about this usage will be presented in Section IV. If, instead, a statistics module is used in the last stage of the pipeline, it is also possible to collect some information for performance evaluation. For example, some statistics for some periodic tasks have been collected and shown in Table I. The Cumulative Distribution Functions (CDFs) of the response times for the three tasks, as measured using a different output module, are displayed in Figure 3. Note that all the results presented up to now can be obtained by just changing the final stage of the processing pipeline.

As explained, in this paper the presented tools are used to collect timing information about IRQ threads. However, before performing such measurements, it is important to test the reliability of this information. For this purpose, some experiments have been performed by considering the network IRQ threads: a stream of periodic UDP packets has been sent between two computers, measuring the inter-packet times in

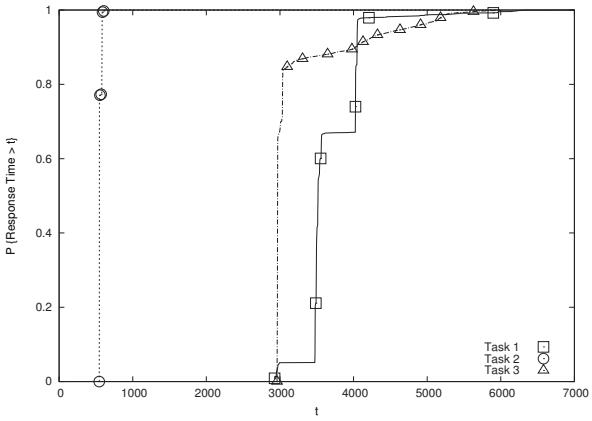


Figure 3. CDFs of the response times for 3 periodic tasks.

Table II
INTER-PACKET TIMES AS MEASURED IN THE SENDER. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	1190	29	1569	1040
T5	5198	22	5278	5058
T10	10195	22	10277	10062
T50	50207	27	50298	50081
T100	100207	25	100290	100093

the sender (Table II) and in the receiver (Table III). Note that, as expected, the values in Table III almost match the values in Table II: the only noticeable difference is test T1, in which the inter-packet times on the receiver have a large maximum value and 0 as a minimum value. This is probably due to some delayed scheduling of the receiver task. After these initial measurements, the proposed tools have been used to extract the inter-arrival times of the network IRQ thread, summarised in Table IV. By comparing Table II and Table IV, it is possible to verify that the average inter-activation times of the network IRQ thread in the receiver are consistent with the average inter-packet times in the sender. The maximum times also match, while the minimum times present some differences. In particular, in tests T1, T5 and T100 the minimum inter-arrival time for the network IRQ thread in receiver is much smaller than the minimum inter-packet time in the sender. A more detailed analysis revealed that this is probably due to some non UDP packets (ICMP or ARP) which are not directly generated by the test program in the sender machine (hence, they are not periodic and they are not listed in Table II). In any case, the comparison between Table II and Table IV seems to confirm the correctness of the collected data.

Some information about the IRQ thread execution times (needed to perform some kind of performance analysis of the system) are shown in Table V, and some examples of distribution functions obtained using these tools will be presented in Section IV.

IV. EXPERIMENTAL RESULTS

The effectiveness of the proposed approach has been tested by an extensive set of experiments. In particular, the per-

Table I
STATISTICS COLLECTED FOR SOME PERIODIC TASKS. TIMES ARE IN μs.

Task	Execution Time				Inter-Arrival Time				Response Time			
	Avg	Std Dev	Max	Min	Avg	Std Dev	Max	Min	Avg	Std Dev	Max	Min
Task 1	2991	273	8953	2956	5993	303	10720	11	3182	555	5986	2960
Task 2	553	66	6025	544	2997	10	3002	2991	556	229	6027	546
Task 3	2941	51	5859	2919	7993	24	9049	6938	3683	397	7285	2927

Table III
INTER-PACKET TIMES AS MEASURED IN THE RECEIVER. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	1207	1011	14336	0
T5	5212	1019	6144	4096
T10	10210	271	12288	8192
T50	50229	1023	51200	49152
T100	100204	530	100352	98304

Table IV
INTER-ARRIVAL TIMES FOR THE NETWORK IRQ THREAD. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	1210	32	1424	59
T5	5222	117	5385	63
T10	10264	60	10353	10093
T50	50832	627	50353	50082
T100	100424	9342	100313	76

formance of the Linux `SCHED_DEADLINE` policy and the influence of the scheduling parameters have been evaluated when the new scheduling class is used to:

- schedule real-time tasks sets
- schedule IRQ threads
- schedule hybrid task sets composed of both real-time tasks and IRQ threads.

The next subsections will report the results obtained for each of these cases.

A. Using SCHED_DEADLINE

To see the new `SCHED_DEADLINE` policy in action, consider a periodic task (with period $5ms$) and two greedy tasks (task which never block, and try to consume all the CPU time) scheduled by two CBSs $(1ms, 10ms)$ and $(1ms, 4ms)$. Figure 4 shows a segment of the schedule produced by the tools presented in Section III.

On the other hand, Figure 5 shows how the CBS scheduler implemented by `SCHED_DEADLINE` is more effective in handling multiple time sensitive applications than the fixed priority policy `SCHED_FIFO` provided by the standard Linux kernel. A simple video player based on GTK[2] is used as a real-time task, and two player's instances reproduce the "Big Buck Bunny"[3] trailer either (i) with two different `SCHED_FIFO` priority or (ii) within two CBSs, $(12.5ms, 40ms)$ and $(25ms, 40ms)$.

Since the frame rate of the video is 25 frames per second (fps), the expected time interval between two consecutive

[2]http://www.gtk.org

[3]http://www.bigbuckbunny.org

Table V
STATISTICS ABOUT THE EXECUTION TIMES OF THE IRQ THREAD. TIMES ARE IN μs.

Test	Average	Std Dev	Max	Min
T1	15	5	63	9
T5	19	1	68	18
T10	14	1	29	13
T50	16	2	28	15
T100	21	3	23	12

frames (named *Inter-Frame Time - IFT* - from now on) is supposed to be $1000/25 = 40ms$. In this experiment, two instances of the video player are executed in parallel, using different scheduling algorithms and priorities. As it clearly emerges from the figure, when `SCHED_FIFO` is used, the player execute with higher priority correctly reproduces the stream, and the IFTs are constant around $40ms$ (average $39573.0\mu s$, standard deviation 4725.1); however, the low priority instance has poor performance, to the point where playback stops completely at frame 310 (this is the meaning of the peak in the graph) and starts again only when the other instance finished.

When the reservation-based approach enabled by `SCHED_DEADLINE` is used, instead, both the instances are able to proceed and the performance they achieve are proportional to the fraction of CPU time they can use. This is shown in the right side of the figure and by the fact that IFT average and standard deviation are, respectively, $39082.0\mu s$, 5735.3 for $(25, 40)$ and $39517.0\mu s$, 19455.0 for $(12.5, 40)$.

B. Controlling the Device Throughput

The next set of experiments has been performed to check the effects of scheduling the disk IRQ thread with a (Q, T) reservation, for different values of Q and T.

First of all, the disk throughput has been measured by using the `hdparm` command and disabling the disk caches. The results of this experiment showed that the disk throughput only depends on the fraction of CPU time Q/T reserved to the disk IRQ thread, and is not affected by the specific values of Q and T. This result seems to contradict the condition expressed by Equation 2, and is probably due to the fact that the disk controller has a large buffer (i.e., N_c is very large).

Figure 6 shows the disk throughput as a function of Q/T (confirming that the throughput is proportional to the fraction of CPU time reserved to the IRQ thread), while Figures 7 and 8 show the probability distributions of the interrupt inter-arrival and execution times. According to such probability distributions, the maximum utilisation of the disk

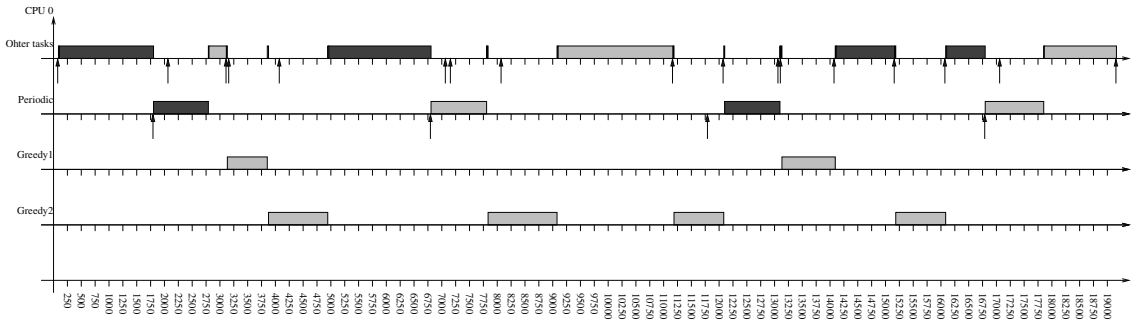


Figure 4. SCHED_DEADLINE serving a periodic task and two CPU hungry (greedy) tasks.

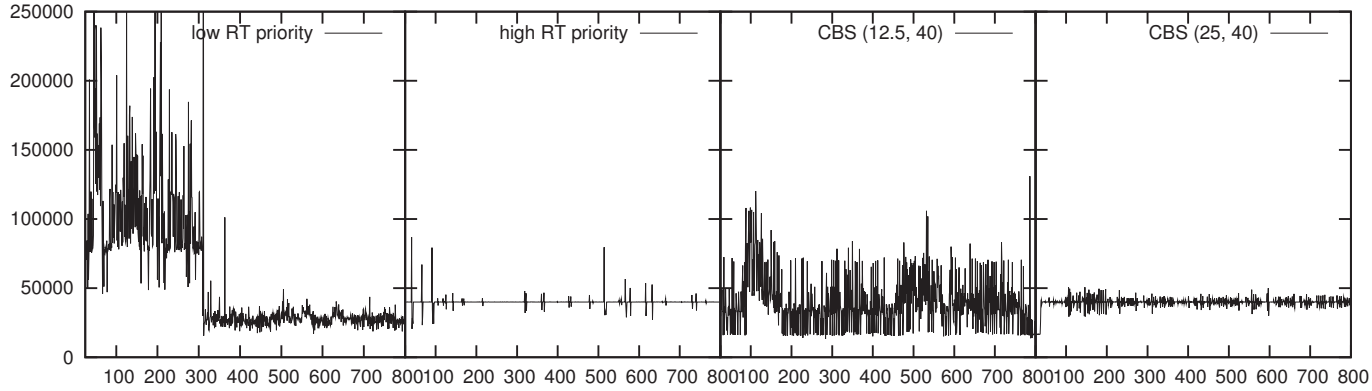


Figure 5. Inter-frame times for two instances of the video player when executing under SCHED_FIFO (with different priorities) or SCHED_DEADLINE. (within different reservations)

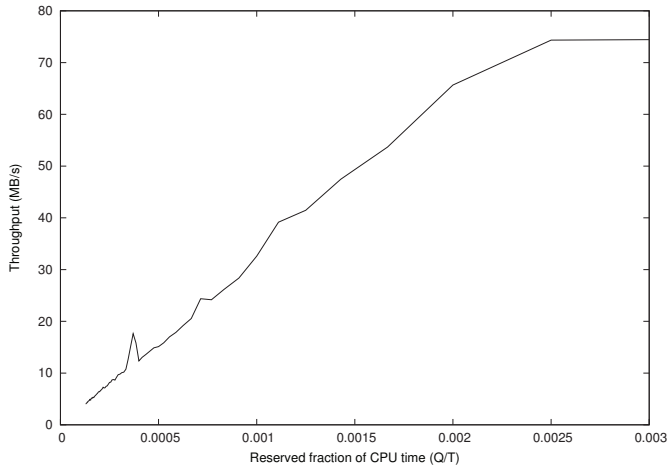


Figure 6. Disk throughput (as measured by hdparm) when the disk IRQ thread is scheduled by a CBS, as a function of the reserved fraction of CPU time.

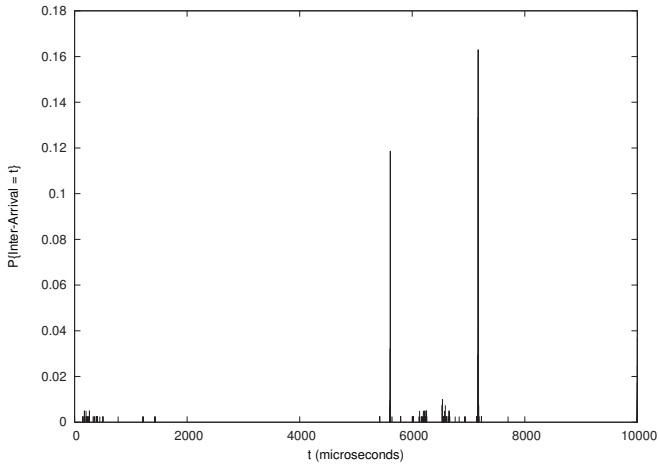


Figure 7. PMF of the inter-arrival times for the disk IRQ thread.

IRQ thread is about 0.41096, while the average utilisation is about 0.0021833. By comparing these data with results shown in Figure 6, it is possible to see that deterministic analysis is too pessimistic and highly overestimates the needed amount of time: in fact, hdparm measures the maximum throughput[4] when $Q/T = 0.003$, which is only a little bit more than the average utilisation. By looking at Figures 7 and 8 again, it is clear that the worst case conditions (leading to the 0.41096 utilisation) are due to a long tail in the execution times probability distribution and to a small amount of small

[4]When running hdparm with the disk IRQ thread scheduled with the maximum fixed priority, the throughput resulted to be about $75MB/s$.

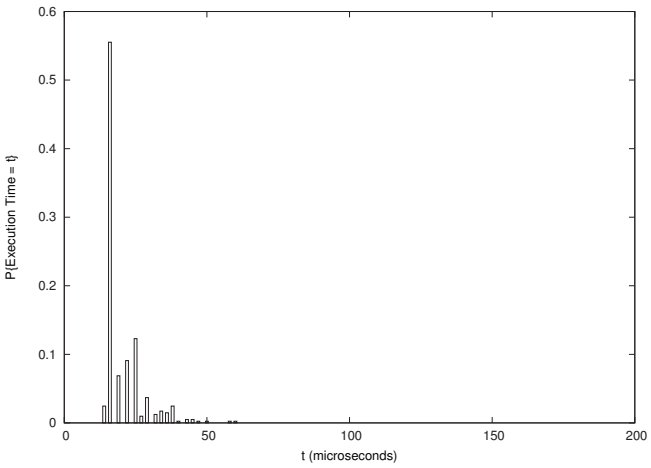


Figure 8. PMF of the execution times for the disk IRQ thread.

inter-arrival times with low probability, hence they are very unlikely. This explains why a fraction of reserved CPU time which is very close to the average utilisation is sufficient for achieving full utilisation. This consideration motivates future investigations on the application of stochastic analysis technique [9]. This research activity is currently under way.

Note that in this experiment the utilisation of disk IRQ thread is quite low, and only a small fraction of the CPU time had to be reserved to it to control the hard-disk performance. Such a low CPU utilisation caused by the disk IRQ thread is due to the usage of DMA when performing disk accesses. Such a mechanism (the DMA) allows to reduce the amount of CPU time needed by the IRQ thread, but can cause some other kind of interference (that cannot be modelled as a task in schedulability analysis) on real-time tasks due to bus contention. By disabling the DMA, all the interference is due to the IRQ thread, and can be properly accounted for in the schedulability analysis. Hence, the experiments have been repeated with DMA disabled (these experiments also allows to better understand what happens when the IRQ thread consumes more time); the results are reported in Table VI. Each line in the table is the average of the results of 20 repeated tests on a UP machine when DMA is disabled; the average utilisation for the disk IRQ thread is about 0.66, with a minimum utilisation of 0.57 and a maximum of 0.93. As expected, the throughput without DMA is much lower than the throughput achieved when using DMA, and it proportionally grows with the fraction of CPU time reserved to the interrupt thread. The maximum throughput (100% of the throughput measured when the disk IRQ thread is scheduled with a fixed priority) is achieved when $Q/T = 0.95$. Again, the throughput seems to only depend on the Q/T ratio, and not on the reservation period T: in other words, the average throughput achieved when using a $(2ms, 100ms)$ reservation is the same achieved when using a $(20ms, 1000ms)$ reservation.

After evaluating the "raw" disk performance through `hdparm`, the next experiments have been run to evaluate the performance of more complex read operations, involving

Table VI
DISK IO-THROUGHPUT WHEN IRQ THREAD IS SCHEDULED WITH DEADLINE SCHEDULER.

Test	Average
$(2ms, 100ms)$	1.9858%
$(4ms, 100ms)$	4.23484%
$(6ms, 100ms)$	6.38374%
$(20ms, 1000ms)$	2.16843%
$(40ms, 1000ms)$	4.46488%
$(60ms, 1000ms)$	6.49811%
$(10ms, 100ms)$	10.6726%
$(20ms, 100ms)$	21.5825%
$(40ms, 100ms)$	41.8182%
$(60ms, 100ms)$	62.4476%
$(80ms, 100ms)$	82.5952%
$(90ms, 100ms)$	92.9371%
$(95ms, 100ms)$	100%
$(100ms, 1000ms)$	11.778%
$(200ms, 1000ms)$	23.261%
$(400ms, 1000ms)$	44.4056%
$(600ms, 1000ms)$	65%
$(800ms, 1000ms)$	84.5455%
$(900ms, 1000ms)$	93.007%
$(950ms, 1000ms)$	100%

Table VII
TIME NEEDED TO PERFORM A FILE COPY, WHEN THE DISK IRQ THREAD IS SCHEDULED WITH DIFFERENT PARAMETERS.

Test	Average	Std Dev	Max	Min
No Reservations	$16.89s$	$0.12s$	$17.05s$	$16.67s$
$(30ms, 100ms)$	$52.85s$	$0.87s$	$55.22s$	$52.36s$
$(40ms, 100ms)$	$39.52s$	$0.61s$	$41.25s$	$39.27s$
$(50ms, 100ms)$	$31.49s$	$0.12s$	$31.74s$	$31.40s$
$(60ms, 100ms)$	$26.23s$	$0.03s$	$26.30s$	$26.19s$
$(70ms, 100ms)$	$22.58s$	$0.20s$	$23.14s$	$22.47s$
$(80ms, 100ms)$	$19.77s$	$0.19s$	$20.31s$	$19.69s$
$(90ms, 100ms)$	$17.59s$	$0.04s$	$17.66s$	$17.55s$

multiple system calls and file system access. The operation involved is a simple `cat` of a large file (about $44MB$) redirecting output to $/dev/null$. The total time for the operation has been measured, disabling disk caches and DMA. Several runs have been repeated, and the results are reported in Table VII. The experiments were performed in a pretty old machine, and the operation lead to a very big interrupt workload, loading the CPU up to about 100% of the CPU time. The first line of the table reports the time needed for the operation when the default scheduler is used (that is, `SCHED_RT` is used for IRQ threads) in a machine with no other load. In the following lines `SCHED_DEADLINE` is used and the time falls down as the reserved fraction of CPU grows.

Note that by modifying the amount of reserved CPU time it is possible to control the amount of time needed for executing the `cat` command. In particular, the throughput (computed as the ratio between the file size and the time needed to cat it) is proportional to Q/T, as shown in Figure 9.

Finally, the time needed to read a large file has been analysed by measuring the system time, the user time, and the total time used by the task performing the read operation (disabling the disk caches so that the experiment is more deterministic and repeatable). The size of the file involved in this experiment was about $80MB$. As expected, the total

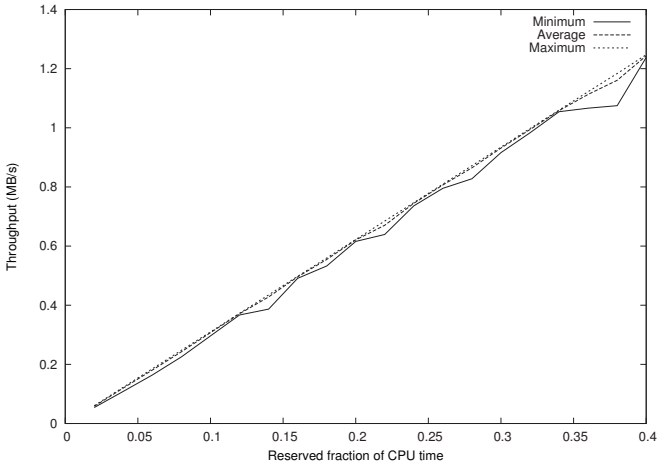


Figure 9. Throughput when reading a large file, as a function of the reserved CPU time.

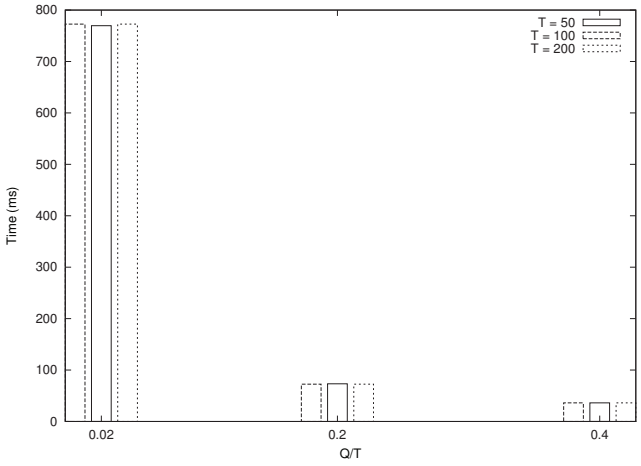


Figure 10. Total time needed to read a large file.

time needed to read the file resulted to be much larger than the sum of the system time and the user time, because the task is often blocked waiting data from the disk (so, the task performing the read operation spends most of the time in the wait state. Most of the CPU time is consumed by the disk IRQ thread, and is not visible in the statistics of the user task performing the read operation). Moreover, the amount of user time and system time used by the task resulted to be very small, and did not depend on the reservation parameters (the user time was around $2.5ms$, and the system time was around $100ms$). On the other hand, the total time (shown in Figure 10) resulted to be proportional to T/Q (because the disk throughput is proportional to Q/T), and (again), disk interrupts did not suffer by Equation 2.

C. Latency/Throughput Trade-Offs

Finally, some experiments have been performed to show how the proposed approach allows one to control both the device throughput and the real-time performance of user-space tasks. The video player presented above has been used as a real-time task, and the network device has been considered (using `netperf` to generate network load and to measure the network throughput [9]).

Table VIII
NETWORK THROUGHPUT ACHIEVED BY USING DIFFERENT RESERVATION PARAMETERS FOR THE VIDEO PLAYER AND FOR THE NETWORK HARD IRQ.

Test	Player CBS	net IRQ CBS	Throughput
Test1	$(29ms, 40ms)$	$(9ms, 100ms)$	$59.75Mbps$
Test2	$(28ms, 40ms)$	$(12ms, 100ms)$	$65.43Mbps$
Test3	$(26ms, 40ms)$	$(13ms, 100ms)$	$70.83Mbps$
Test4	$(25ms, 40ms)$	$(14ms, 100ms)$	$76.14Mbps$
Test5	$(20ms, 40ms)$	$(18ms, 100ms)$	$88.55Mbps$

An instance of `netperf` has then been activated concurrently with the video player, and the experiment has been repeated scheduling the video player as `SCHED_OTHER` and as `SCHED_FIFO` with a priority higher than the network IRQ threads. The results, displayed in Figure 11, show that when the player executes alone it is able to correctly reproduce the video (the inter-frame times are constant around $40ms$), while when some concurrent network load is created, the inter-frame times increases by a large amount (and video playback is not continuous). Finally, if the player is scheduled with a priority higher than the priority of the network IRQ threads, then it is again able to work correctly, but in this case **the network throughput measured by netperf drops from** $88Mbps$ **to about** $58Mbps$.

A trade-off between real-time performance for the player and high network throughput can be found by using reservation-based scheduling. To this purpose, the tool presented in Section III can be used to collect the probability distributions of the execution and inter-arrival times of the network IRQ threads, and these data can be used as shown in Section II-C.

Based on such analysis, the reservation parameters shown in Table VIII have been used, obtaining the network throughput shown in the last column of the table. The evolution of the inter-frame times in the player for the most interesting cases is shown in Figure 12. As it is possible to perceive by looking at the table and at the figure, resource reservations really allow to find latency/throughput trade-offs, as previously claimed:

- if the player is reserved enough CPU time ($29ms$ every period of $40ms$), then the inter-frame times are stable and near to $40ms$ (see the right side of Figure 12). However, in this case it is possible to reserve only a small fraction of the CPU time to the network IRQ thread, and the network throughput is low (about $60Mbps$);
- if enough CPU time is reserved to the network IRQ thread ($18ms$ every $100ms$), then the maximum network throughput can be achieved). But in this case it is possible to reserve only $20ms$ of CPU time every $40ms$ to the player, and the inter-frame times increase (see the left side of Figure 12). Note, however, that the maximum inter-frame times are still under $80ms$ (compare this situation with the middle of Figure 11);

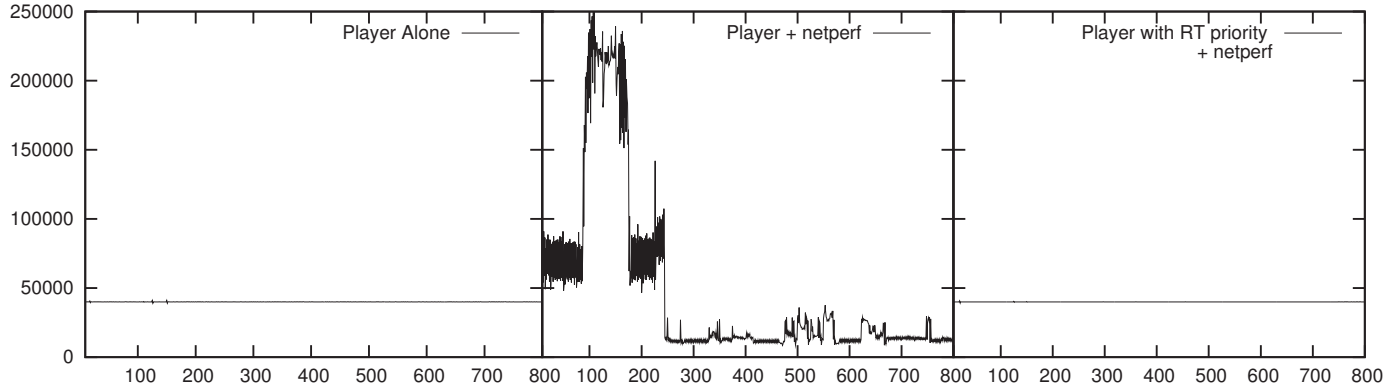


Figure 11. Inter-frame times for the video player when executed alone and concurrently with `netperf`, with different priorities.

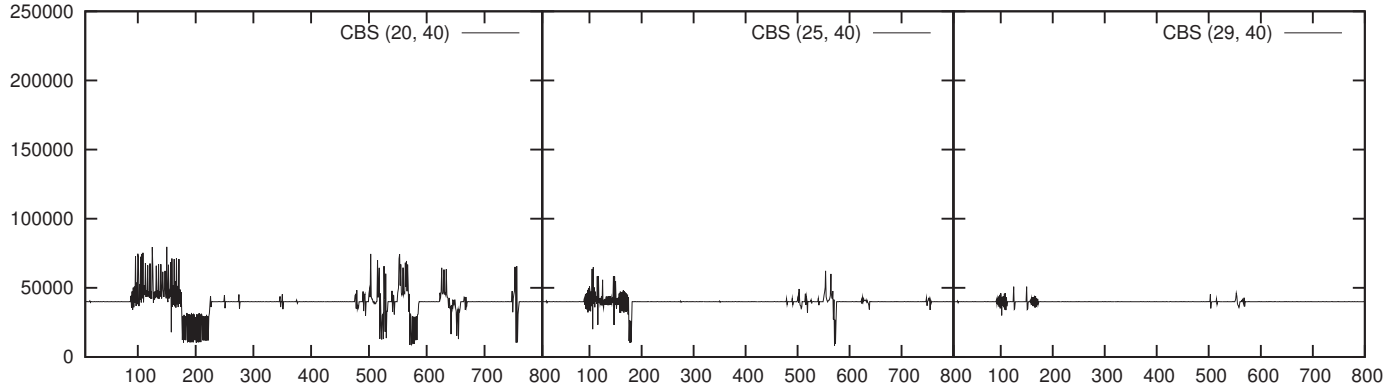


Figure 12. Inter-frame times for a video player when executed alone and concurrently with `netperf`, using different kinds of reservations.

- some compromises can be found: for example, scheduling the player with a $(25ms, 40ms)$ reservation and the network IRQ thread with a $(14ms, 100ms)$ reservation it is possible to have reasonable inter-frame times with a good network throughput (about 86% of the maximum).

V. Conclusions and Future Work

This paper reported the results of some experiences with device drivers scheduling in real-time systems. In particular, some of the presented experiments show how recent developments in the Linux kernel can be exploited to schedule the IRQ threads so that both their interference on real-time tasks and the device throughput can be controlled.

The proposed solution is based on the Linux Preempt-RT kernel (which transforms interrupt handlers into schedulable entities), a new reservation-based scheduler, and some tools based on Ftrace that can be used to infer the timing information needed to correctly assign the scheduling parameters.

As a future work, the effectiveness and usability of the stochastic analysis for IRQ threads will be investigated by considering different workloads and resources. This will probably require to simplify the Markov model used in the stochastic analysis, so that it can be applied more easily.

Acknowledgements

This work has been partially supported by the European Commission under the ACTORS project (FP7-ICT-216586).

This project has also been supported by the "Provincia Autonoma di Trento"[5] by means of the PAT/CRS Project RoSE (http://imedia.disi.unitn.it/RoSE).

References

[1] H. Haertig, R. Baumgartl, M. Borriss, C. joachim Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schnberg, and J. Wolter, "DROPS - OS support for distributed multimedia applications," in *In Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[2] H. Haertig and M. Roitzsch, "Ten years of research on L4-based real-time systems," in *Proceedings of the Eighth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[3] S. Rostedt, "Internals of the rt patch," in *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2007.

[4] L. Henriques, "Threaded IRQs on Linux PREEMPT-RT," in *Proceedings of Fifth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, Dublin, Ireland, June 2009.

[5]Translation of the original statement: "Lavoro eseguito con il contributo della Provincia autonoma di Trento".

[5] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and Wang, "Modeling device driver effects in real-time schedulability analysis: Study of a network driver," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, Bellevue, WA, 2007.

[6] T. Baker, A. Wang, and M. J. Stanovich, "Fitting linux device drivers into an analyzable scheduling framework," in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-time Applications*, Pisa, Italy, July 2007.

[7] G. Modena, L. Abeni, and L. Palopoli, "Providing qos by scheduling interrupt threads," in *Work in Progress of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS 2008)*, St. Louis, MO, April 2008.

[8] L. Abeni, N. Manica, and L. Palopoli, "Reservation-based scheduling for irq threads," in *Proceedings of the 11th Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[9] N. Manica, L. Abeni, and L. Palopoli, "Reservation-based interrupt scheduling," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS 2010)*, Stockholm, Sweden, April 2010.

[10] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[11] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1297, 1996.

[12] R. Black, P. Barham, A. Donnelly, and N. Stratford, "Protocol implementation in a vertically structured operating system," in *Proceedings of the 22nd Annual Conference on Local Computer Networks*, 1997.

[13] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[14] K. Jeffay and G. Lamastra, "A comparative study of the realization of rate-based computing services in general purpose operating systems," in *Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Cheju Island, South Korea, 2000, pp. 81–90.

[15] S. Ghosh and R. Rajkumar, "Resource management of the OS network subsystem," in *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002)*, 2002, pp. 271–279.

[16] Y.Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, Rio de Janeiro, Brazil, December 2006.

[17] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[18] P. Rallo, N. Manica, and L. Abeni, "Inferring temporal behaviours through kernel tracing," DISI - University of Trento, Tech. Rep. DISI-10-021, April 2010.

Evaluating Android OS for Embedded Real-Time Systems

Cláudio Maia, Luís Nogueira, Luís Miguel Pinho
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
Email:{crrm,lmn,lmp}@isep.ipp.pt

***Abstract*—Since its official public release, Android has captured the interest from companies, developers and the general audience. From that time up to now, this software platform has been constantly improved either in terms of features or supported hardware and, at the same time, extended to new types of devices different from the originally intended mobile ones. However, there is a feature that has not been explored yet - its real-time capabilities.**

This paper intends to explore this gap and provide a basis for discussion on the suitability of Android in order to be used in Open Real-Time environments. By analysing the software platform, with the main focus on the virtual machine and its underlying operating system environments, we are able to point out its current limitations and, therefore, provide a hint on different perspectives of directions in order to make Android suitable for these environments.

It is our position that Android may provide a suitable architecture for real-time embedded systems, but the real-time community should address its limitations in a joint effort at all of the platform layers.

***Keywords*-Android, Open Real-Time Systems, Embedded Systems**

I. Introduction

Android [1] was made publicly available during the fall of 2008. Being considered a fairly new technology, due to the fact that it is still being substantially improved and upgraded either in terms of features or firmware, Android is gaining strength both in the mobile industry and in other industries with different hardware architectures (such as the ones presented in [2] and [3]). The increasing interest from the industry arises from two core aspects: its open-source nature and its architectural model.

Being an open-source project, allows Android to be fully analysed and understood, which enables feature comprehension, bug fixing, further improvements regarding new functionalities and, finally, porting to new hardware. On the other hand, its Linux kernel-based architecture model also adds the use of Linux to the mobile industry, allowing to take advantage of the knowledge and features offered by Linux. Both of these aspects make Android an appealing target to be used in other type of environments.

Another aspect that is important to consider when using Android is its own Virtual Machine (VM) environment. Android applications are Java-based and this factor entails the use of a VM environment, with both its advantages and known problems.

Nevertheless, there are features which have not been explored yet, as for instance the suitability of the platform to be used in Open Real-Time environments. Taking into consideration works made in the past such as [4], [5], either concerning the Linux kernel or VM environments, there is the possibility of introducing temporal guarantees allied with Quality of Service (QoS) guarantees in each of the aforementioned layers, or even in both, in a way that a possible integration may be achieved, fulfilling the temporal constraints imposed by the applications. This integration may be useful for multimedia applications or even other types of applications requiring specific machine resources that need to be guaranteed in an advanced and timely manner. Thus, taking advantage of the real-time capabilities and resource optimisation provided by the platform.

Currently, the Linux kernel provides mechanisms that allow a programmer to take advantage of a basic preemptive fixed priority scheduling policy. However, when using this type of scheduling policy it is not possible to achieve real-time behaviour. Efforts have been made in the implementation of dynamic scheduling schemes which, instead of using fixed priorities for scheduling, use the concept of dynamic deadlines. These dynamic scheduling schemes have the advantage of achieving full CPU utilisation bound, but at the same time, they present an unpredictable behaviour when facing system overloads.

Since version 2.6.23, the standard Linux kernel uses the Completely Fair Scheduler (CFS), which applies fairness in the way that CPU time is assigned to tasks. This balance guarantees that all the tasks will have the same CPU share and that, each time that unfairness is verified, the algorithm assures that task re-balancing is performed. Although fairness is guaranteed, this algorithm does not provide any temporal guarantees to tasks, and therefore, neither Android does it, as its scheduling operations are delegated to the Linux kernel.

Android uses its own VM named Dalvik, which was specifically developed for mobile devices and considers memory optimisation, battery power saving and low frequency CPU. It relies on the Linux kernel for the core operating system features such as memory management and

scheduling and, thus, also presents the drawback of not taking any temporal guarantees into consideration.

The work presented in this paper is part of the CooperatES (Cooperative Embedded Systems) project [6], which aims at the specification and implementation of a QoS-aware framework, defined in [7], to be used in open and dynamic cooperative environments. Due to the nature of the environments, the framework should support resource reservation in advance and guarantee that the real-time execution constraints imposed by the applications are satisfied.

In the scope of the project, there was the need of evaluating Android as one of the possible target solutions to be used for the framework's implementation. As a result of this evaluation, this paper discusses the potential of Android and the implementation directions that can be adopted in order to make it possible to be used in Open Real-Time environments. However, our focus is targeted to soft real-time applications and therefore, hard-real time applications were not considered in our evaluation.

The remainder of this paper is organised as follows: Section II briefly describes the Android's architecture. Section III presents a detailed evaluation along with some of the Android internals and its limitations when considering real-time environments. The different perspectives of extension are detailed in Section IV. Finally, Section V concludes this paper.

II. Android's Architecture

Android is an open-source software architecture provided by the Open Handset Alliance [8], a group of 71 technology and mobile companies whose objective is to provide a mobile software platform.

The Android platform includes an operating system, middleware and applications. As for the features, Android incorporates the common features found nowadays in any mobile device platform, such as: application framework reusing, integrated browser, optimised graphics, media support, network technologies, etc.

The Android architecture, depicted in Figure 1, is composed by five layers: Applications, Application Framework, Libraries, Android Runtime and finally the Linux kernel.

The uppermost layer, the Applications layer, provides the core set of applications that are commonly offered out of the box with any mobile device.

The Application Framework layer provides the framework Application Programming Interfaces (APIs) used by the applications running on the uppermost layer. Besides the APIs, there is a set of services that enable the access to the Android's core features such as graphical components, information exchange managers, event managers and activity managers, as examples.

Below the Application Framework layer, there is another layer containing two important parts: Libraries and the Android Runtime. The libraries provide core features to

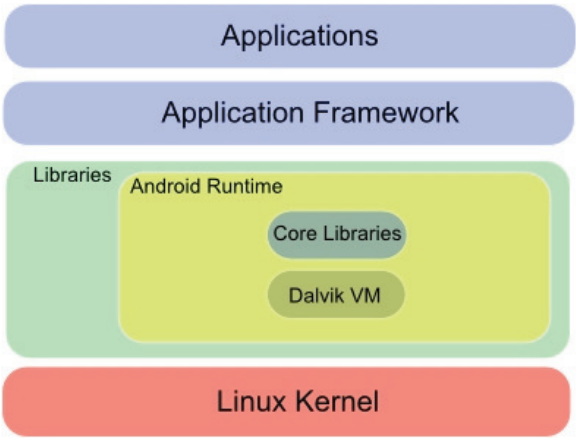


Figure 1. Android Architecture

the applications. Among all the libraries provided, the most important are *libc*, the standard C system library tuned for embedded Linux-based devices; the Media Libraries, which support playback and recording of several audio and video formats; Graphics Engines, Fonts, a lightweight relational database engine and 3D libraries based on OpenGL ES.

Regarding the Android Runtime, besides the internal core libraries, Android provides its own VM, as previously stated, named Dalvik. Dalvik [9] was designed from scratch and it is specifically targeted for memory-constrained and CPU-constrained devices. It runs Java applications on top of it and unlike the standard Java VMs, which are stack-based, Dalvik is an infinite register-based machine. Being a register-machine, it presents two advantages when compared to stack-based machines. Namely, it requires 30% less instructions to perform the same computation as a typical stack machine, causing the reduction of instruction dispatch and memory access; and less computation time, which is also derived from the elimination of common expressions from the instructions. Nevertheless, Dalvik presents 35% more bytes in the instruction stream than a typical stack-machine. This drawback is compensated by the consumption of two bytes at a time when consuming the instructions.

Dalvik uses its own byte-code format name Dalvik Executable (*.dex*), with the ability to include multiple classes in a single file. It is also able to perform several optimisations during *dex* generation when concerning the internal storage of types and constants by using principles such as minimal repetition; per-type pools; and implicit labelling. By applying these principles, it is possible to have *dex* files smaller than a typical Java archive (*jar*) file. During install time, each *dex* file is verified and optimisations such as byte-swapping and padding, static-linking and method in-lining are performed in order to minimise the runtime evaluations and at the same time to avoid code security violations.

The Linux kernel, version 2.6, is the bottommost layer and is also a hardware abstraction layer that enables the

interaction of the upper layers with the hardware layer via device drivers. Furthermore, it also provides the most fundamental system services such as security, memory management, process management and network stack.

III. Suitability of Android for Open Real-Time Systems

This section discusses the suitability of Android for open embedded real-time systems, analyses its architecture internals and points out its current limitations. Android was evaluated considering the following topics: its VM environment, the underlying Linux kernel, and its resource management capabilities.

Dalvik VM is capable of running multiple independent processes, each one with a separate address space and memory. Therefore, each Android application is mapped to a Linux process and able to use an inter-process communication mechanism, based on Open-Binder [10], to communicate with other processes in the system. The ability of separating each process is provided by Android's architectural model. During the device's boot time, there is a process responsible for starting up the Android's runtime, which implies the startup of the VM itself. Inherent to this step, there is a VM process, the Zygote, responsible for the pre-initialisation and pre-loading of the common Android's classes that will be used by most of the applications. Afterwards, the Zygote opens a socket that accepts commands from the application framework whenever a new Android application is started. This will cause the Zygote to be forked and create a child process which will then become the target application. Zygote has its own heap and a set of libraries that are shared among all processes, whereas each process has its own set of libraries and classes that are independent from the other processes. This model is presented in Figure 2. The approach is beneficial for the system as, with it, it is possible to save RAM and to speed up each application startup process.

Android applications provide the common synchronisation mechanisms known to the Java community. Technically speaking, each VM instance has at least one main thread and may have several other threads running concurrently. The threads belonging to the same VM instance may interact and synchronise with each other by the means of shared objects and monitors. The API also allows the use of synchronised methods and the creation of thread groups in order to ease the manipulation of several thread operations. It is also possible to assign priorities to each thread. When a programmer modifies the priority of a thread, with only 10 priority levels being allowed, the VM maps each of the values to Linux *nice* values, where lower values indicate a higher priority. Dalvik follows the *pthread* model where all the threads are treated as native *pthreads*. Internal VM threads belong to one thread group and all other application threads belong to another group. According to source code analysis, Android does

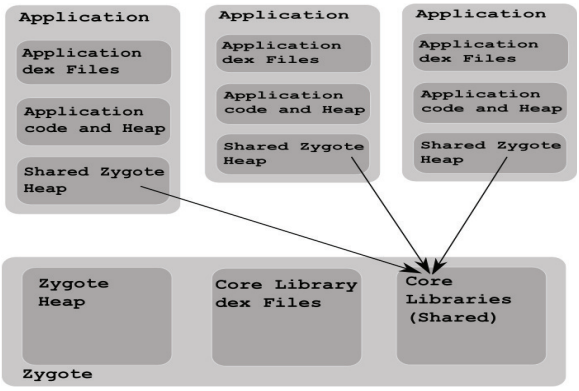


Figure 2. Zygote Heap

not provide any mechanisms to prevent priority inversion neither allows threads to use Linux's real-time priorities within Dalvik.

Threads may suspend themselves or be suspended either by the Garbage Collector (GC), debugger or the signal monitor thread. The VM controls all the threads through the use of a internal structure where all the created threads are mapped. The GC will only run when all the threads referring to a single process are suspended, in order to avoid inconsistent states.

The GCs have the difficult task of handling dynamic memory management, as they are responsible for deallocating the memory allocated by objects that are no longer needed by the applications. Concerning Android's garbage collection process, as the processes run separately from other processes and each process has its own heap and a shared heap - the Zygote's heap - Android runs separate instances of GCs in order to collect memory that is not being used anymore. Thus, each process heap is garbage collected independently, through the use of parallel mark bits that sign which objects shall be removed by the GC. This mechanism is particularly useful in Android due to the Zygote's shared heap, which in this case is kept untouched by the GC and allows a better use of the memory.

Android uses the mark-sweep algorithm to perform garbage collection. The main advantage provided by the platform is that there will be a GC running per process, which wipes all the objects from the application heap of a specific process. This way, GCs belonging to other processes will not impact the GC running for a specific process. The main disadvantage arises from the algorithm used. As this algorithm implies the suspension of all the threads belonging to an application, this means that no predictability can be achieved as that specific process will be freezed while being garbage collected.

Android's VM relies on the Linux kernel to perform all the scheduling operations. This means that all the threads running on top of the VM will be, by default, scheduled with

SCHED_OTHER, and as such will be translated into the fair scheme provided by the kernel. Therefore, it is not possible to indicate that a particular task needs to be scheduled using a different scheduling scheme.

Interrupt/event handling plays another important role when concerning real-time systems, as it may lead to inconsistent states if not handled properly. Currently, Android relies on the Linux kernel to dispatch the interrupt/event via device drivers. After an interrupt, the Java code responsible for the event handling will be notified in order to perform the respective operation. The communication path respects the architecture layers and inter-process communication may be used to notify the upper event handlers.

Currently, Dalvik does not support Just-in-Time (JIT) compilation, although a prototype has already been made available in the official repositories, which indicates that this feature will be part of one of the next versions. Other features that are also being considered as improvements are: a compact and more precise garbage collector and the use of ahead-of-time compilation for specific pieces of code.

As previously stated, Android relies on the Linux kernel for features such as memory management, process management and security. As such, all the scheduling activities are delegated by the VM to the kernel.

Android uses the same scheduler as Linux, known as Completely Fair Scheduler (CFS). CFS has the objective of providing balance between tasks assigned to a processor. For that, it uses a red-black binary tree, as presented in Figure 3, with self-balancing capabilities, meaning that the longest path in the tree is no more than twice as long as the shortest path. Other important aspect is the efficiency of these types of trees, which present a complexity of $O(logn)$, where n represents the number of elements in the tree. As the tree is being used for scheduling purposes, the balance factor is the amount of time provided to a given task. This factor has been named virtual runtime. The higher the task's virtual runtime value, the lower is the need for the processor.

In terms of execution, the algorithm works as follows: the tasks with lower virtual runtime are placed on the left side of the tree, and the tasks with the higher virtual runtime are placed on the right. This means that the tasks with the highest need for the processor will always be stored on the left side of the tree. Then, the scheduler picks the left-most node of the tree to be scheduled. Each task is responsible for accounting the CPU time taken during execution and adding this value to the previous virtual runtime value. Then, it is inserted back into the tree, if it has not finished yet. With this pattern of execution, it is guaranteed that the tasks contend the CPU time in a fair manner.

Another aspect of the fairness of the algorithm is the adjustments that it performs when the tasks are waiting for an I/O device. In this case, the tasks are compensated with the amount of time taken to receive the information they needed to complete its objective.

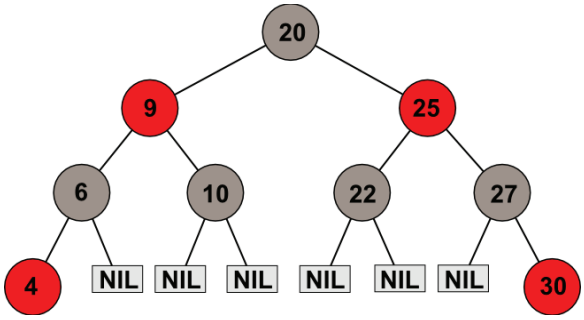


Figure 3. Red-Black Tree example

Since the introduction of the CFS, the concept of scheduling classes was also introduced. Basically, these classes provide the connection between the main generic scheduler functionalities and the specific scheduler classes that implement the scheduling algorithms. This concept allows several tasks to be scheduled differently by using different algorithms for this purpose. Regarding the main scheduler, it is periodic and preemptive. Its periodicity is activated by the frequency of the CPU clock. It allows preemption either when a high priority task needs CPU time or when an interrupt exists. As for task priorities, these can be dynamically modified with the *nice* command and currently the kernel supports 140 priorities, where the values ranging from 0 to 99 are reserved for real-time processes and the values ranging from 100 to 139 are reserved for normal processes.

Currently, the Linux kernel supports two scheduling real-time classes, as part of the compliance with the POSIX standard [11], SCHED_RR and SCHED_FIFO. SCHED_RR may be used for a round robin scheduling policy and SCHED_FIFO for a first-in, first-out policy. Both policies have a high impact on the system's performance if bad programming applies. However, most of the tasks are scheduled with SCHED_OTHER class, which is a non real-time policy.

The task scheduling plays one of the most important roles concerning the real-time features presented by a particular system. Currently, Linux's real-time implementation is limited to two scheduling real-time classes, both based on priority scheduling. Another important aspect to be considered in the evaluation is that most of the tasks are scheduled by CFS. Although CFS tries to optimise the time a task is waiting for CPU time, this effort is not enough as it is not capable of providing guaranteed response times.

One important aspect that should be remarked is that although the Linux kernel supports the real-time classes aforementioned, these classes are only available for native[1] Android applications. Normal Android applications can only take advantage of the synchronisation mechanisms described

[1]A native application in Android is an application that can run on top of the Linux kernel without the need of the VM.

earlier in this paper.

Regarding synchronisation, Android uses its own implementation of *libc* - named *bionic*. *bionic* has its own implementation of the *pthread* library and it does not support process-shared mutexes and condition variables. However, thread mutexing and thread condition variables are supported in a limited manner. Currently, inter-process communication is handled by Open-Binder. In terms of real-time limitations, the mechanisms provided by the architecture do not solve the old problems related with priority inversion. Therefore, synchronisation protocols such as priority ceiling and inheritance are not implemented.

In terms of interrupt/event handling, these are performed by the kernel via device drivers. Afterwards, the kernel is notified and then is responsible for notifying the application waiting for that specific interrupt/event. None of the parts involved in the handling has a notion of the time restrictions available to perform its operations. This behaviour becomes more serious when considering interrupts. In Linux the interrupts are the highest priority tasks, and therefore, this means that a high priority task can be interrupted by the arrival of an interrupt. This is considered a big drawback, as it is not possible to make the system totally predictable.

Resource management implies its accounting, reclamation, allocation, and negotiation [12]. Concerning resource management conducted at the VM level, CPU time is controlled by the scheduling algorithms, whereas memory can be controlled either by the VM, if we consider the heaps and its memory management, or by the operating system kernel. Regarding memory, operations such as accounting, allocation and reallocation can be performed. All these operations suffer from an unbounded and non-deterministic behaviour, which means that it is not possible to define and measure the time allowed for these operations. The network is out of scope of our analysis and thus was not evaluated.

At the kernel level, with the exception of the CPU and memory, all the remaining system's hardware is accessed via device drivers, in order to perform its operations and control the resources' status.

Nevertheless, a global manager that has a complete knowledge of the applications' needs and system's status is missing. The arbitration of resources among applications requires proper control mechanisms if real-time guarantees are going to be provided. Each application has a resource demand associated to each quality level it can provide. However, under limited resources not all applications will be able to deliver their maximum quality level. As such, a global resource manager is able to allocate resources to competing applications so that a global optimisation goal of the system is achieved [7].

IV. Possible Directions

This section discusses four possible directions to incorporate the desired real-time behaviour into the Android architecture. The first approach considers the replacement of the Linux operating system by one that provides real-time features and, at the same time, it considers the inclusion of a real-time VM. The second approach respects the Android standard architecture by proposing the extension of Dalvik as well as the substitution of the standard operating system by a real-time Linux-based operating system. The third approach simply replaces the Linux operating system for a Linux real-time version and real-time applications use the kernel directly. Finally, the fourth approach proposes the addition of a real-time hypervisor that supports the parallel execution of the Android platform in one partition while the other partition is dedicated to the real-time applications.

Regarding the first approach, depicted in Figure 4, this approach replaces the standard Linux kernel with a real-time operating system. This modification introduces predictability and determinism in the Android architecture. Therefore, it is possible to introduce new dynamic real-time scheduling policies through the use of scheduling classes; predict priority inversion and to have better resource management strategies. However, this modification entails that all the device drivers supported natively need to be implemented in the operating system with predictability in mind. This task can be painful, specially during the integration phase. Nevertheless, this approach also leaves space for the implementation of the required real-time features in the Linux kernel. Implementing the features in the standard Linux kernel requires time, but it has the advantage of providing a more seamless integration with the remaining components belonging to the architectures involved.

The second modification proposed, within the first approach, is the inclusion of a real-time Java VM. This modification is considered advantageous as, with it, it is possible to have bounded memory management; real-time scheduling within the VM, depending on the adopted solution; better synchronisation mechanisms and finally to avoid priority inversion. These improvements are considered the most influential in achieving the intended deterministic behaviour at the VM level. It is important to note that the real-time VM interacts directly with the operating system's kernel for features such as task scheduling or bounded memory management. As an example, if one considers task scheduling, the real-time VM is capable of mapping each task natively on the operating system where it will be scheduled. If the operating system supports other types of scheduling policies besides the fixed priority-based scheduler, the VM may use them to schedule its tasks. This means that most of the operations provided by real-time Java VMs are limited to the integration between the VM's supported features and the supported operating system's features.

Other advantage from this approach is that it is not necessary to keep up with the release cycles of Android, although some integration issues may arise between the VM and the kernel. The impact of introducing a new VM

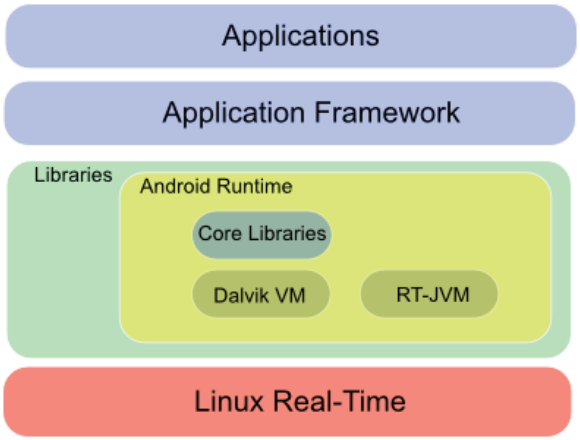


Figure 4. Android full Real-Time

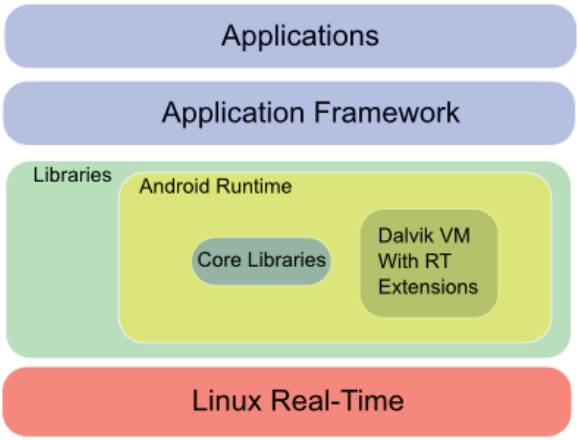


Figure 5. Android Extended

in the system is related to the fact that all the Android specificities must be implemented as well as *dex* support in the interpreter. Besides this disadvantage, other challenges may pose such as the integration between both VMs. This integration possibly entails the formulation of new algorithms to optimize scheduling and memory management in order to be possible to have an optimal integrated system as a whole and also to treat real-time applications in the correct manner.

The second proposed approach, presented in Figure 5, also introduces modifications in the architecture both in the operating system and virtual machine environments. As for the operating system layer, the advantages and disadvantages presented in the first approach are considered equal, as the principle behind it is the same. The major difference lies on the extension of Dalvik with real-time capabilities based on the Real-Time Specification for Java (RTSJ) [13]. By extending Dalvik with RTSJ features we are referring to the addition of the following API classes: *RealTimeThread*, *NoHeapRealTimeThread*, as well as the implementation of generic objects related to real-time scheduling and memory management such as *Scheduler* and *MemoryAreas*. All of these objects will enable the implementation of real-time garbage collection algorithms, synchronization algorithms and finally, asynchronous event handling algorithms. All of these features are specifically related to the RTSJ and must be considered in order to be possible to have determinism and predictability. However, its implementation only depends on the extent one wishes to have, meaning that a full compliant implementation may be achieved if the necessary implementation effort is applied in the VM extensions and the operating system's supported features. This extension is beneficial for the system as with it, it is possible to incorporate a more deterministic behaviour at the VM level without the need of concerning about the particularities of Dalvik. Nevertheless, this approach has the disadvantage of having to keep up with the release cycles of the Android, more specially the VM itself, if one wants to add these extensions to all the available versions of the platform.

Two examples of this direction are [14] and [15]. The work in [14] states that the implementation of a resource management framework is possible in the Android platform with some modifications in the platform. Although the results presented in this work are based on the CFS scheduler, work is being done to update the scheduler to a slightly modified version of EDF [16], that incorporates reservation-based scheduling algorithms as presented in [17].

The work reported in [15] is being conducted in the scope of CooperatES project [6], where a proof of concept of a QoS-aware framework for cooperative embedded real-time systems has already been developed for the Android platform. Other important aspect of this work is the implementation of a new dynamic scheduling strategy named Capacity Sharing and Stealing (CSS) [18] in the Android platform.

Both works show that it is possible to propose new approaches based on the standard Linux and Android architectures and add real-time behaviour to them in order to take advantage of resource reservation and real-time task scheduling. With both of these features, any of these systems is capable of guaranteeing resource bandwidth to applications, within an interval of time, without jeopardising the system.

The third proposed approach, depicted in Figure 6, is also based in Linux real-time. This approach takes advantage of the native environment, where it is possible to deploy real-time applications directly over the operating system. This can be advantageous for applications that do not need the VM environment, which means that a minimal effort will be needed for integration, while having the same intended behaviour. On the other hand, applications that need a VM environment will not benefit from the real-time capabilities of the underlying operating system.

Finally, the fourth approach, depicted in Figure 7, em-

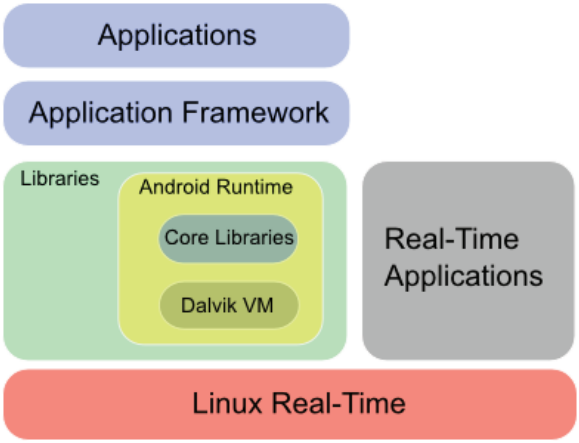


Figure 6. Android partly Real-Time

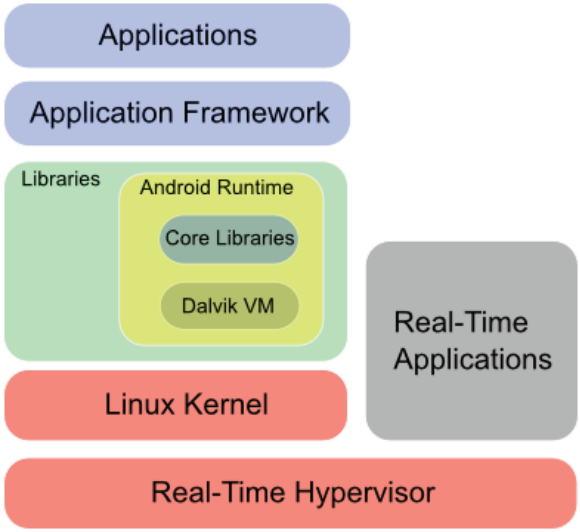


Figure 7. Android with a Real-Time Hypervisor

ploys a real-time hypervisor that is capable of running Android as a guest operating system in one of the partitions and real-time applications in another partition, in a parallel manner. This approach is similar to the approach taken by the majority of the current real-time Linux solutions, such as RTLinux [19] or RTAI [20]. These systems are able to run real-time applications in parallel to the Linux kernel, where the real-time tasks have higher priority than the Linux kernel tasks, which means that hard real-time can be used. On the other hand, the Linux partition tasks are scheduled using the spare time remaining from the CPU allocation. The main drawback from this approach is that real-time applications are limited to the features offered by the real-time hypervisor, meaning that they can not use Dalvik or even most of the Linux services. Other limitation known lies on the fact that if a real-time application hangs, all the system may also hang.

V. Conclusion

At first glance, Android may be seen as a potential target for real-time environments and, as such, there are numerous industry targets that would benefit from an architecture with such capabilities. Taking this into consideration, this paper presented the evaluation of the Android platform to be used as a real-time system. By focusing on the core parts of the system it was possible to expose the limitations and then, to present four possible directions that may be followed to add real-time behaviour to the system.

Android was built to serve the mobile industry purposes and that fact has an impact on the way that the architecture might be used. However, with some effort, as proven by the presented approaches, it is possible to have the desired real-time behaviour on any Android device. This behaviour may suit specific applications or components by providing them the ability of taking advantage of temporal guarantees, and therefore, to behave in a more predictable manner.

However, this effort must be addressed at the different layers of the architecture, in a combined way, in order to allow for potential extensions to be useful for the industry.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was supported by FCT (CISTER Research Unit - FCT UI 608 and CooperatES project - PTDC/ EIA/ 71624/ 2006) and RESCUE - PTDC/EIA/65862/2006, and by the European Commission through the ArtistDesign NoE (IST-FP7-214373).

References

[1] Android, "Home page," Jan. 2010. [Online]. Available: http://www.android.com/

[2] Android-x86, "Android-x86 project," Jan. 2010. [Online]. Available: http://www.android-x86.org/

[3] G. Macario, M. Torchiano, and M. Violante, "An in-vehicle infotainment software architecture based on google android," in *SIES*. Lausanne, Switzerland: IEEE, July 2009, pp. 257–260.

[4] RTMACH, "Linux/rk," Mar. 2010. [Online]. Available: http://www.cs.cmu.edu/~rajkumar/linux-rk.html

[5] A. Corsaro, "jrate home page," Mar. 2010. [Online]. Available: http://jrate.sourceforge.net/

[6] CooperatES, "Home page," Jan. 2010. [Online]. Available: http://www.cister.isep.ipp.pt/projects/cooperates/

[7] L. Nogueira and L. M. Pinho, "Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments," *Journal of Parallel and Distributed Computing*, vol. 69, no. 6, pp. 491–507, June 2009.

[8] O. H. Alliance, “Home page,” Jun. 2010. [Online]. Available: http://www.openhandsetalliance.com/

[9] D. Bornstein, “Dalvik vm internals,” Mar. 2010. [Online]. Available: http://sites.google.com/site/io/dalvik-vm-internals

[10] P. Inc., “Openbinder 1.0,” Mar. 2010. [Online]. Available: http://www.angryredplanet.com/~hackbod/openbinder/

[11] IEEE, “Ieee standard 1003.1,” Mar. 2010. [Online]. Available: http://www.opengroup.org/onlinepubs/009695399/

[12] M. T. Higuera-Toledano and V. Issarny, “Java embedded real-time systems: An overview of existing solutions,” in *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 392–399.

[13] R.-T. S. for Java, “Rtsj 1.0.2,” Jan. 2010. [Online]. Available: http://www.rtsj.org/specjavadoc/book_index.html

[14] R. Guerra, S. Schorr, and G. Fohler, “Adaptive resource management for mobile terminals - the actors approach,” in *Proceedings of 1st Workshop on Adaptive Resource Management (WARM10)*, Stockholm, Sweden, April 2010.

[15] C. Maia, L. Nogueira, and L. M. Pinho, “Experiences on the implementation of a cooperative embedded system framework,” CISTER Research Centre, Porto, Portugal, Tech. Rep., June 2010.

[16] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[17] D. Faggioli, M. Trimarchi, and F. Checconi, “An implementation of the earliest deadline first algorithm in linux,” in *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC09)*. New York, NY, USA: ACM, 2009, pp. 1984–1989.

[18] L. Nogueira and L. M. Pinho, “Capacity sharing and stealing in dynamic server-based real-time systems,” in *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium*, Long Beach,CA,USA, March 2007, p. 153.

[19] W. R. Systems, “Real-time linux,” Jun. 2010. [Online]. Available: http://www.rtlinuxfree.com/

[20] P. d. M. Dipartimento di Ingegneria Aerospaziale, “Realtime application interface for linux,” Jun. 2010. [Online]. Available: https://www.rtai.org/

Extending a HSF-enabled Open-Source Real-Time Operating System with Resource Sharing

Martijn M.H.P. van den Heuvel, Reinder J. Bril and Johan J. Lukkien
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

Moris Behnam
Real-Time Systems Design Group
Mälardalen Real-Time Research Centre
P.O. Box 883, SE-721 23 Västerås, Sweden

***Abstract*—Hierarchical scheduling frameworks (HSFs) provide means for composing complex real-time systems from well-defined, independently analyzed subsystems. To support resource sharing within two-level, fixed priority scheduled HSFs, two synchronization protocols based on the stack resource policy (SRP) have recently been presented, i.e. HSRP [1] and SIRAP [2]. This paper describes an implementation to provide such HSFs with SRP-based synchronization protocols. We base our implementations on the commercially available real-time operating system μC/OS-II, extended with proprietary support for periodic tasks, idling periodic servers and two-level fixed priority preemptive scheduling. Specifically, we show the implementation of SRP as a local synchronization protocol, and present the implementation of both HSRP and SIRAP. Moreover, we investigate the system overhead induced by the synchronization primitives of each protocol. Our aim is that these protocols can be used side-by-side within the same HSF, so that their primitives can be selected based on the protocol's relative strengths[1].**

I. Introduction

The increasing complexity of real-time systems demands a decoupling of (*i*) development and analysis of individual applications and (*ii*) integration of applications on a shared platform, including analysis at the system level. Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating this decoupling [3]. In such *open environments* [4], an application that is validated to meet its timing constraints when executing in isolation will continue meeting its timing constraints after integration (or admission) on a shared platform. Temporal isolation between applications is provided by allocating a *budget* to each subsystem. In this paper we assume a one-to-one relation between applications and subsystems.

To accommodate resource sharing between fixed-priority scheduled subsystems, two synchronization protocols have been proposed based on the *stack resource policy* (SRP) [5], i.e. HSRP [1] and SIRAP [2]. A HSF extended with such a protocol makes it possible to share logical resources between arbitrary tasks, which are located in arbitrary subsystems, in a mutually exclusive manner. A resource that is used in more than one subsystem is denoted as a *global shared resource*. A resource that is only shared within a single subsystem is a *local shared resource*. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of the corresponding subsystem's budget, excessive blocking periods can occur which may hamper the correct timeliness of other subsystems [6]. To prevent the depletion of a subsystem's budget during a global resource access SIRAP uses a skipping mechanism. Contrary, HSRP uses an overrun mechanism (with or without payback), i.e. the overrun mechanism reacts upon a budget depletion during a global resource access by temporarily increasing the budget with a statically determined amount for the duration of that resource access. The relative strengths of HSRP and SIRAP heavily depend on system characteristics [7], which makes it attractive to support both within the same HSF. In this paper we present the implementation of the synchronization primitives for both synchronization protocols to support global (i.e. inter-subsystem) resource sharing by extending a HSF-enabled μC/OS-II operating system. The choice of operating system is in line with our industrial and academic partners.

A. Problem Description

Most off-the-shelf real-time operating systems, including μC/OS-II, do not provide an implementation for SRP nor hierarchical scheduling. We have extended μC/OS-II with support for periodic tasks, idling periodic servers [8] and two-level fixed priority preemptive scheduling (FPPS). Although global resource sharing protocols within HSFs are extensively investigated for ideal system models, implementations are lacking within real-time operating systems. Moreover, the run-time overhead of these protocols is unknown and not included in these models. These overheads become relevant during deployment of such a resource sharing open environment.

B. Contributions

The contribution of this paper is fivefold. First, we present an implementation of SRP within μC/OS-II as a local (i.e. intra-subsystem) resource access protocol. We aim at a modular design, so that one can choose between the original priority inheritance implementation, or our SRP implementation. We show that our SRP implementation improves the existing μC/OS-II implementation for mutual exclusion by lifting several limitations and simplifying the implementation. We restrict this implementation to single unit resources and single processor platforms. Second, we present an implementation

[1]The work in this paper is supported by the Dutch HTAS-VERIFIED project, see http://www.htas.nl/index.php?pid=154

for HSRP [1] and SIRAP [2] to support global (i.e. inter-subsystem) resource sharing within a two-level fixed priority hierarchically scheduled system. We aim at unified interfaces for both protocol implementations to ease the integration of HSRP and SIRAP within the same HSF. Third, we compare the system overhead of the primitives of both synchronization protocols. Fourth, we show how HSRP and SIRAP can be integrated side-by-side within the same HSF. Inline with μC/OS-II we allow to enable or disable each protocol extension during compile time. Finally, we show that our protocol implementations follow a generic design, i.e. the μC/OS-II specific code is limited.

C. Overview

The remainder of this paper is as follows. Section II describes the related work. Section III presents background information on μC/OS-II. Section IV summarizes our basic μC/OS-II extensions for periodic tasks, idling periodic servers and a two-level fixed priority scheduled HSF based on a timed event management system [9, 10]. Section V describes SRP at the level of local resource sharing, including its implementation. Section VI describes common terminology and implementation efforts for SRP at the level of global resource sharing. Section VII and VIII describe the implementation of global SRP-based synchronization protocols, i.e. SIRAP and subsequently HSRP. In Section IX we compare both protocol implementations. Section X discusses the challenges towards a system supporting both protocols side-by-side. Finally, Section XI concludes the paper.

II. Related work

In literature, several implementations are presented to provide temporal isolation between dependent applications. De Niz et al. [11] support resource sharing between reservations based on PCP in their fixed priority scheduled Linux/RK resource kernel. Buttazzo and Gai [12] implemented a reservation-based earliest deadline first (EDF) scheduler for the real-time ERIKA Enterprise kernel, including SRP-based synchronization support. However, both approaches are not applicable in open environments [4], because they lack a distinct support for local and global synchronization. Contrary, Behnam et al. [13] implemented a HSF on top of the real-time operating system VxWorks, but do not support synchronization between applications. Although resource sharing between applications within HSFs has been extensively investigated in literature, e.g. [1, 2, 7, 14, 15, 16, 17], none of the above implementations provide such synchronization primitives. In this paper we describe the implementation of such synchronization primitives by further extending our HSF-enabled μC/OS-II operating system. Looking at existing industrial real-time systems, FPPS is the de-facto standard of task scheduling. Having such support will simplify migration to and integration of existing legacy applications into the HSF, avoiding unbridgeable technology revolutions for engineers. In the remainder of this section we provide a brief overview of both local and global synchronization protocols.

A. Local synchronization

The priority inversion problem [18] can be prevented by the *priority inheritance protocol* (PIP). PIP makes a task inherit the highest priority of any other tasks that are waiting on a resource the task uses. A disadvantage of PIP is that it suffers from the chained blocking and deadlock problem [18]. As a solution, Sha et al. [18] proposed the *priority ceiling protocol* (PCP). An easier to implement alternative to PCP is the *highest locker protocol* (HLP). HLP uses an off-line computed ceiling to raise a task's priority when it accesses a resource. In case of HLP a task is already prevented from starting its execution when a resource is accessed by another task sharing this resource, whereas PCP postpones increasing of priorities until a blocking situation occurs. As an alternative to PCP, Baker [5] presented the *stack resource policy* (SRP). SRP has an easier implementation and induces less context switching overhead compared to PCP, and supports dynamic priority scheduling policies. Because of its wide applicability, ease of implementation and its apparent improvements of the existing synchronization protocol, we have decided to extend μC/OS-II with SRP, as presented in this paper.

B. Global synchronization

Recently, three SRP-based synchronization protocols for inter-subsystem resource sharing between tasks have been presented. Their relative strength depends on various system parameters [15]. BROE [14] considers resource sharing under EDF scheduling. Most commercial operating systems, including μC/OS-II, do not implement an EDF scheduler. Both HSRP [1] and SIRAP [2] assume FPPS. In order to deal with resource access while a subsystem's budget depletes, HSRP uses a run-time overrun mechanism [1]. The original analysis of HSRP [1] does not allow for integration in open environments due to the lacking support for independent analysis of subsystems. Behnam et al. [16] lifted this limitation, enabling the full integration of HSRP within HSFs. Alternatively, SIRAP uses a skipping approach to prevent budget depletion inside a critical section [2]. Recently, both synchronization protocols are analytically compared and their impact on the total system load for various system parameters is analyzed using simulations [7]. In this paper we implement both protocols within μC/OS-II, and compare the efficiency of the corresponding primitives.

III. μC/OS-II: A Brief Overview

The μC/OS-II operating system is maintained and supported by Micrium [19], and is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Micrium provides the full μC/OS-II source code with accompanying documentation [20]. The μC/OS-II kernel provides preemptive multitasking, and the kernel size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled.

A. μC/OS-II Task Scheduling

The number of tasks that can run within μC/OS-II is 64 by default, and can be extended to 256 by altering the configuration. Each group of 8 tasks is assigned a bit in the ready-mask, updated during run-time to indicate whether a task is ready to run within this group. The ready-mask allows for optimized fixed priority scheduling by performing efficient bit comparison operations to determine the highest priority ready task to run, as explained in more detail in [20, Section 3].

B. μC/OS-II Synchronization Protocol

The standard μC/OS-II distribution supports synchronization primitives by means of *mutexes* [20]. However, it is not clearly stated which synchronization protocol they implement. Lee and Kim [21] attempted to identify the protocol by analyzing the source code. The presence of a ceiling in the mutex interface suggested the implementation of HLP. However, we have analyzed the behavior in Figure 1 of two tasks that reserve two resources in opposite order[2]. Since we observe a deadlock, μC/OS-II does not seem to implement HLP or an other deadlock avoidance protocol.

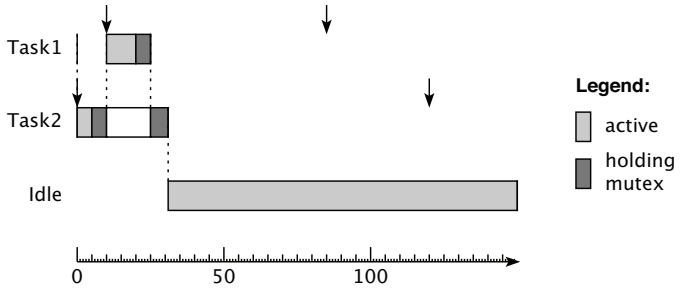


Fig. 1. The mutex implementation in μC/OS-II suffers from the deadlock problem, inherited from the PIP definition. The task-set $\mathcal{T}$ contains tasks τ_1 and τ_2, where τ_1 has the highest priority. τ_1 first locks resource R_1 and subsequently R_2, has a computation time[2] $C_1 = 10 + 5 + 5 + 5 + 5$, and a phasing $\varphi_1 = 10$. τ_2 first locks resource R_2 and subsequently R_1, has a computation time $C_2 = 5 + 10 + 25 + 10 + 5$, and no phasing (i.e. $\varphi_2 = 0$).

μC/OS-II only supports a *single task* on each priority-level, independent of its execution state, because a priority is also used as a task identifier. A priority-level is assigned to each resource on creation of the resource. The priority-level is used to raise the priority of a task when it blocks a higher priority task, which is named *priority calling* in the μC/OS-II terminology. Because of the assignment of a unique priority to each resource, the *transparent* character of PIP is lost. In the original PIP a priority is dynamically raised to the priority of the task that is pending on a resource, which does not require any off-line calculated information. Literature [21] outlines an implementation of PIP within μC/OS-II lifting the limitation on reserving a priority level.

[2]The fragmented computation time C_i denotes the task's consumed time units before/after locking/unlocking a resource, e.g. the scenario $C_{1,1} - Lock(R_1) - C_{1,2} - Lock(R_2) - C_{1,3} - Unlock(R_2) - C_{1,4} - Unlock(R_1) - C_{1,5}$ is denoted as $C_{1,1} + C_{1,2} + \ldots + C_{1,5}$.

IV. Basic μC/OS-II Extensions Recapitulated

In this paper, we consider a HSF with two-level FPPS, where a system $\mathcal{S}$ is composed of a set of *subsystems*, each of which is composed of a set of *tasks*. A *server* is allocated to each subsystem $S_s \in \mathcal{S}$. A global scheduler is used to determine which server should be allocated the processor at any given time. A local scheduler determines which of the chosen server's tasks should actually execute. Given such a HSF mapped on a single processor, we assume that a subsystem is implemented by means of an *idling periodic server* [8]. However, the proposed approach is expected to be easily adaptable to other server models. A server has a replenishment period P_s and a budget Q_s, which together define a *timing interface* $S_s(P_s, Q_s)$ associated with each subsystem S_s. We say that tasks assigned to a server consume processor time *relative to the server's budget* to signify that the consumed processor time is accounted to (and subtracted from) that budget.

Most real-time operating systems, including μC/OS-II, do not include a reservation-based scheduler, nor provide means for hierarchical scheduling. Although some real-time operating systems provide primitives to support periodic tasks, e.g. RTAI/Linux [22], μC/OS-II does not. In the remainder of this section we outline our realization of such extensions for μC/OS-II, which are required basic blocks to enable the integration of global synchronization.

A. Timed Event Management

Real-time systems need to schedule many different timed events (e.g. programmed delays, arrival of periodic tasks and budget replenishment) [9]. On contemporary computer platforms, however, the number of hardware timers is usually limited, meaning that events need to be multiplexed on the available timers. Facing these challenges led to the invention of RELTEQ [9].

The basic idea behind RELTEQ is to store the arrival times of events relative to each other, by expressing the arrival time of an event relative to the arrival time of the previous event. The arrival time of the head event is relative to the current time, as shown in Figure 2. While RELTEQ is not restricted to any specific hardware timer, in this paper we assume a periodic timer. At every tick of the periodic timer the time of the head event in the queue is decremented.

Two operations can be performed on an event queue: new events can be inserted and the head event can be popped. When a new event e_i with absolute time t_i is inserted, the event queue has to be traversed, accumulating the relative times of the events until a later event e_j is found, with $t_i < t_j$, where t_i and t_j are both absolute times. When such an event is found, then *(i)* e_i is inserted before e_j, *(ii)* its time is set relative to the previous event, and *(iii)* the arrival time of e_j is set relative to e_i. If no later event was found, then e_i is appended at the end of the queue, and its time is set relative to the previous event.

In [10] we proposed a technique to extend RELTEQ with the aim to minimize the overhead of handling events belonging

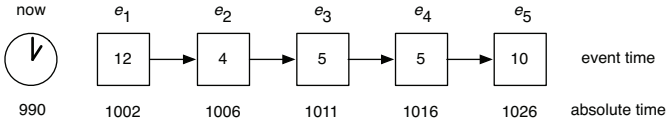


Fig. 2. Example of the RELTEQ event queue.

to inactive servers. To support hierarchical scheduling, we introduced a *server queue* for *each* server to keep track of the events local to the server. At any time at most one server can be active; all other servers are inactive. A *stopwatch queue* keeps track of the time passed since the last server switch, which provides a mechanism to synchronize the server queues with the global time upon a server context switch. Finally, we introduced the notion of a *virtual event*, which are timed events *relative to the consumed budget*, e.g. budget depletion. An additional server event queue that is not synchronized with global time upon a server context-switch implements the infrastructure to support virtual events.

B. Periodic Task Scheduling

The implemented RELTEQ extensions within μC/OS-II easily allow the support for periodic tasks. Because different μC/OS-II services can influence the state of a task, we do not directly alter the task's state. Instead, a periodic task is characterized by an infinite loop which executes its periodic workload and subsequently pends on a semaphore. The event handler corresponding to RELTEQ's activation event releases the pending task and inserts a new event for the next period.

C. Simplified Server Scheduling

Extending the standard μC/OS-II scheduler with basic HSF support requires the identification and realization of the following concepts:

1) Applications: An application can be modeled as a set of tasks. Since μC/OS-II tasks are bundled in groups of eight to accommodate efficient fixed priority scheduling, as explained in Section III-A, a server can naturally be represented by a multiple of eight tasks.

2) Idling Periodic Servers: A realization of the idling periodic server is very similar to the implementation of a periodic task using our RELTEQ extensions [10], with the difference that the server structures do not require additional semaphores. An idling task is contained in all servers at the lowest local priority.

3) Two-level FPPS-based HSF: Similar to the existing μC/OS-II task scheduling approach, we introduce an additional bit-mask to represent whether a server has capacity left. When the scheduler is called it determines the highest priority server with remaining capacity, and hides all tasks from other servers for the local scheduler. Subsequently, the local scheduler determines the highest priority ready task within the server.

V. Stack Resource Policy Implementation

As a supportive step towards global synchronization, first the SRP protocol is summarized, followed by the implementation description of the SRP primitives. Note that in its original formulation SRP introduces the notion of preemption-levels. In this paper we consider FPPS, which allows to unify preemption-levels with task priorities.

A. SRP Recapitulated

The key idea of SRP is that when a task needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. Therefore a preemption test is performed during runtime by the scheduler: A task is not permitted to preempt until its priority is the highest among those of all ready tasks *and* its priority is higher than the *system ceiling*.

1) Resource Ceiling: Each resource is assigned a static, off-line calculated ceiling, which is defined as the maximum priority of any task that shares the resource.

2) System Ceiling: The system ceiling is defined as the maximum of the resource ceilings of all currently locked resources. When no resources are locked the system ceiling is zero, meaning that it does not block any tasks from preempting. When a resource is locked, the system ceiling is adjusted dynamically using the resource ceiling. A run-time mechanism for tracking the system ceiling can be implemented by means of a stack.

B. SRP Data and Interface Description

Each resource accessed using an SRP-based mutex is represented by a `Resource` structure. This structure is defined as follows:

```
typedef struct resource{
    INT8U  ceiling;
    INT8U  lockingTask;
    void*  previous;
} Resource;
```

The `Resource` structure stores properties which are used to track the system ceiling, as explained in the next subsection. The corresponding mutex interfaces are defined as follows:

1) Create a SRP mutex:
```
Resource* SRPMutexCreate(INT8U ceiling,
                         INT8U *err);
```
2) Lock a SRP mutex:
```
void SRPMutexLock(Resource* r, INT8U *err);
```
3) Unlock a SRP mutex:
```
void SRPMutexUnlock(Resource* r);
```

C. SRP Primitive and Data-structure Implementation

Nice properties of the SRP are its simple locking and unlocking operations. Moreover, SRP allows to share a single stack between all tasks within an application. In turn, during run-time we need to keep track of the system ceiling and the scheduler needs to compare the highest ready task priority with the system ceiling.

1) Tracking the System Ceiling: We use the `Resource` data-structure to implement a *system ceiling stack*. `ceiling` stores the resource ceiling and `lockingTask` stores the identifier of the task currently holding the resource. The `previous` pointer is used to maintain the stack structure, i.e. it points to the previous `Resource` structure on the stack. The `ceiling` field of the `Resource` on top of the stack represents the current system ceiling.

2) Resource Locking: When a task tries to lock a resource with a resource ceiling higher than the current system ceiling, the corresponding resource ceiling is pushed on top of the system ceiling stack.

3) Resource Unlocking: When unlocking a resource, the value on top of the system ceiling stack is popped if the corresponding resource holds the current system ceiling. The scheduler is called to allow for scheduling ready tasks that might have arrived during the execution of the critical section.

4) Scheduling: When the μC/OS-II scheduler is called it calls a function which returns the highest priority ready task. Accordingly to SRP we extend this function with the following rule: when the highest ready task has a priority lower than or equal to the current system ceiling, the priority of the task on top of the resource stack is returned. The returned priority serves as a task identifier.

D. Evaluation

To show that our SRP-based implementation improves on the standard mutex implementation we have simulated the same task set as in Figure 1. The resulting trace in Figure 3 shows that our SRP implementation successfully handles nested critical sections, whereas the priority inheritance implementation causes a deadlock of the involved tasks.

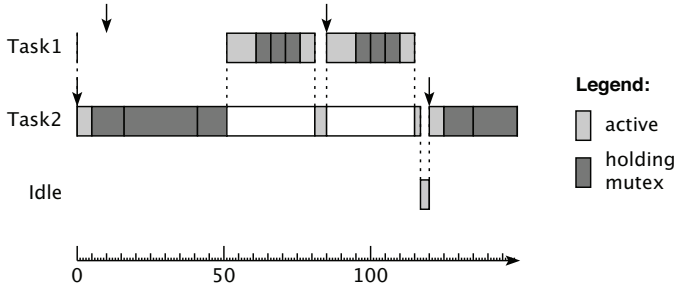


Fig. 3. Using the SRP mutexes the deadlock problem for nested resources is resolved. The task parameters are equal to the example in Section III-B.

Moreover, our implementation reduces the amount of source code: μC/OS-II's PIP implementation consists of 442 lines of code (excluding comments) versus 172 lines of code for our SRP implementation. Additionally, SRP avoids keeping track of the waiting tasks, i.e. it is more processor time and memory space efficient, and lifts the limitation to reserve a priority for each resource.

VI. Global SRP-based Synchronization

Both HSRP and SIRAP can be used for independent development of subsystems and support subsystem integration in the presence of globally shared resources [2, 16]. Besides, both protocols use SRP to synchronize global resource access, and therefore parts of their implementations are common, as described in this section.

A. Definitions

Lifting SRP to a two-level HSF requires to extend our notion of a ceiling compared to the original SRP.

1) Resource ceiling: With every global resource two types of resource ceilings are associated; a *global* resource ceiling for global scheduling and a *local* resource ceiling for local scheduling. These ceilings are defined according the SRP.

2) System/subsystem ceiling: The system/subsystem ceilings are dynamic parameters that change during execution. The system/subsystem ceiling is equal to the highest global/local resource ceiling of a currently locked resource in the system/subsystem. Under SRP, a task can only preempt the currently executing task (even when accessing a global resource) if its priority is higher than its subsystem ceiling. A similar condition for preemption holds for subsystems.

B. Extending the SRP Data Structures

Each global resource accessed using an SRP-based mutex is represented by a `Resource` structure. Additionally, the resource is represented by a `localResource` structure defined as follows:

```
typedef struct {
  struct resource* globalResource;
  INT8U    localCeiling;
  INT8U    localLockingTask;
  void*    previous;
} localResource;
```

The `localResource` structure stores properties which are used to track the subsystem ceiling, as explained in the next subsection.

C. Tracking the Subsystem/System Ceiling

Similar to the SRP implementation we need to maintain a stack for the global and local resource ceilings. The global stack is represented by the stack implementation described in Section V-C. A global mutex creates a normal SRP mutex and passes the system ceiling as a ceiling, i.e.

Pseudo-code 1 $Resource*$ GlobalMutexCreate($INT8U$ globalCeiling);

```
1: InitializeLocalResourceData();
2: return SRPMutexCreate(globalCeiling, 0);
```

To keep track of local subsystem ceilings, we need to maintain a separate *subsystem ceiling stack* for each subsystem. We use the `localResource` data-structure to implement a *subsystem ceiling stack*. The `globalResource` points to the corresponding resource block at the global level. `localCeiling` stores the local resource ceiling and `localLockingTask` stores the identifier of the task currently holding the resource. The `previous` pointer is used to maintain the stack structure, i.e. it points to the previous `localResource` structure on the stack. The `localCeiling` field of the `localResource` on top of the stack represents the current subsystem ceiling.

D. Scheduling

Extending the scheduler with a preemption rule is similar to the SRP implementation. When the scheduler selects the next server to be activated, its associated subsystem priority must exceed the current system ceiling. Similarly, the priority of the selected task must exceed the subsystem ceiling.

VII. SIRAP IMPLEMENTATION

This section presents the SIRAP implementation using the SRP infrastructure described in Section VI. First, we summarize SIRAP, followed by its realization within μC/OS-II.

A. SIRAP Recapitulated

SIRAP uses SRP to synchronize access to globally shared resources [2], and uses a skipping approach to prevent budget depletion inside a critical section. If a task wants to enter a critical section, it enters the critical section at the earliest time instant so that it can complete the critical section before the subsystem budget expires. If the remaining budget is not sufficient to lock and release a resource before expiration, (*i*) the task blocks itself, and (*ii*) the subsystem ceiling is raised to prevent other tasks in the subsystem to execute until the resource is released.

B. SIRAP Data and Interface Description

The SIRAP interfaces for locking and unlocking globally shared resources are defined as follows:

1) Lock SIRAP mutex:
```
void SIRAP_Lock(Resource* r, INT16U holdTime);
```
2) Unlock SIRAP mutex:
```
void SIRAP_Unlock(Resource* r);
```

The lock operation contains a parameter *holdTime*, which is accounted in terms of processor cycles and allocated to the calling task's budget. Efficiently filling in this parameter in terms of system load requires the programmer to correctly obtain the resource holding time [2, 23]. Since this provides an error-prone way of programming, we discuss an alternative approach in Section X.

C. SIRAP Primitive Implementation

SIRAP's locking and unlocking are building on the SRP implementation. Note that kernel primitives are assumed to execute non-preemptively, unless denoted differently (i.e. in SIRAP's lock operation).

1) Resource Locking: The lock operation first updates the subsystem's local ceiling according to SRP to prevent other tasks within the subsystem from interfering during the execution of the critical section. In order to successfully lock a resource there must be sufficient remaining budget within the server's current period. The remaining budget $Q_{remaining}$ is returned by a function that depends on the virtual timers mechanism, see Section IV-A. SIRAP's skipping approach requires the knowledge of the resource holding times ($holdTime$) [23] when accessing a resource. If $Q_{remaining}$ is not sufficient, the task will spinlock until the next replenishment event expires. To avoid a race-condition between a resource unlock and budget depletion, we require that $Q_{remaining}$ is strictly larger than $holdTime$ before granting access to a resource. The lock operation in pseudo-code is shown in Source 2.

When the server's budget is replenished, all tasks spinlocking on a resource are unlocked as soon as the task is rescheduled. Although after budget replenishment a repeated test on the remaining budget is superfluous [2], we claim that spinlocking efficiently implements the skipping mechanism. A disadvantage of this implementation is that it relies on the assumption of a idling periodic server[3]. For any budget-preserving server, e.g. the deferrable server [25], the skipping mechanism by means of a spinlock is unacceptable, because a task consumes server budget during spinlocking.

Pseudo-code 2 void SIRAP_lock($Resource*$ r, $INT16U$ holdTime);

```
1: updateSubsystemCeiling();
2: while holdTime >= Q_remaining do
3:     enableInterrups;
4:     disableInterrups;
5: end while
6: SRPMutexLock(r, 0);
```

An alternative implementation would be to suspend a task when the budget is insufficient and resume a task when the budget is replenished. Firstly, this alternative approach induces additional overhead within the budget replenishment event due to the resumption of the blocked task. Secondly, μC/OS-II requires at any time a schedulable ready task, which is optionally a special idle task at the lowest priority. However, the system/subsystem ceilings prevent the idle task to be switched in. We could choose to make an exception for the idle task, but this breaks the property of SRP allowing to share stack space among tasks [5]. We consider further elaboration on these issues out of the scope of this paper.

2) Resource Unlocking: Unlocking a resource simply means that the system/subsystem ceiling must be updated and the SRP mutex must be released. Note that the latter command will also cause rescheduling.

Pseudo-code 3 void SIRAP_unlock($Resource*$ r);

```
1: updateSubsystemCeiling();
2: SRPMutexUnlock(r);
```

VIII. HSRP IMPLEMENTATION

This section presents the HSRP implementation. First, we summarize HSRP, followed by its realization within μC/OS-II.

A. HSRP Recapitulated

HSRP uses SRP to synchronize access to globally shared resources [1], and uses an overrun mechanism to prevent excessive blocking times due to budget depletion inside a critical section. When the budget of a subsystem expires and the subsystem has a task τ_i that is still locking a globally shared resource, this task τ_i continues its execution until it releases the locked resource. When a task accesses a global resource the local subsystem ceiling is raised to the highest local priority, i.e. for the duration of the critical section the task executes non-preemptively with respect to other tasks within the same subsystem. Two alternatives of the overrun mechanism are presented: (*i*) overrun with payback, and (*ii*) overrun without

[3] A polling server [24] also works under this assumption, but does not adhere to the periodic resource model [3], and therefore increases the complexity to integrate SIRAP and HSRP within a single HSF [7]. We leave the implementation for alternative server models as future work.

payback. The payback mechanism requires that when a overrun happens in a subsystem S_s, the budget of this subsystem is decreased with the consumed amount of overrun in its next execution instant. Without payback no further actions are taken after an overrun has occurred. We do not further investigate the relative strengths of both alternatives, since these heavily depend on the chosen system parameters [7]. In this section we show an implementation supporting both HSRP versions.

B. HSRP Data and Interface Description

The HSRP interfaces for locking and unlocking globally shared resources are defined as follows:

1) Lock HSRP mutex:
```
void HSRP_Lock(Resource* r);
```
2) Unlock HSRP mutex:
```
void HSRP_Unlock(Resource* r);
```

Contrary to SIRAP, the lock operation lacks the $holdTime$ parameter. Instead, HSRP uses a static amount of overrun budget, X_s, assigned to each server within the system.

C. HSRP Primitive Implementation

HSRP's locking and unlocking are building on the SRP implementation. Additionally, we need to adapt the budget depletion event handler to cope with overrun. This requires to keep track of the number of resources locked ($lockedResourceCounter$) within subsystem S_s. The server data-structure is extended with four additional fields for bookkeeping purposes, i.e. $lockedResourceCounter$, $inOverrun$, $consumedOverrun$ and $paybackEnabled$. The consumption of overrun budget ends when the normal budget is replenished [17], which requires an adaption of the budget replenishment event. Optionally, we implement an payback mechanism in the budget replenishment event. These event handlers are provided by RELTEQ as presented in Section IV-A.

1) Resource Locking: The lock operation first updates the subsystem's local ceiling to the highest local priority to prevent other tasks within the subsystem from interfering during the execution of the critical section. The lock operation in pseudo-code can be denoted as follows:

Pseudo-code 4 void HSRP_lock($Resource*$ r);
1: $updateSubsystemCeiling()$;
2: $S_s.lockedResourceCounter++$;
3: *SRPMutexLock*$(r, 0)$;

2) Resource Unlocking: Unlocking a resource means that the system/subsystem ceiling must be updated and the SRP mutex must be released. Additionally, in case that overrun budget, X_s, is consumed and no other global resource is locked within the same subsystem, we need to inform the scheduler that overrun has ended. Optionally, the amount of consumed overrun budget is stored to support payback upon the next replenishment. The unlock operation in pseudo-code is is shown in Pseudo-code 5.

The command $setSubsystemBudget(0)$ performs two actions: (*i*) the server is blocked to prevent the scheduler from rescheduling the server, and (*ii*) the budget depletion event is removed from RELTEQ's virtual event queue.

Pseudo-code 5 void HSRP_unlock($Resource*$ r);
1: $updateSubsystemCeiling()$;
2: $S_s.lockedResourceCounter--$;
3: **if** $S_s.lockedResourceCounter == 0$ **and** $S_s.inOverrun$ **then**
4: **if** $S_s.paybackEnabled$ **then**
5: $S_s.consumedOverrun = X_s - Q_{remaining}$;
6: **end if**
7: $setSubsystemBudget(0)$;
8: **end if**
9: *SRPMutexUnlock*(r);

3) Budget Depletion: We extend the event handler corresponding to a budget depletion by a conditional enlargement of the budget of the size X_s, with $X_s > 0$, i.e. in pseudo code:

Pseudo-code 6 on budget depletion:
1: **if** $S_s.lockedResourceCounter > 0$ **then**
2: $setSubsystemBudget(X_s)$;
3: $S_s.inOverrun =$ **true**;
4: **end if**

Note that $setSubsystemBudget(X_s)$ inserts a new event in RELTEQ's virtual event queue. Furthermore, we postpone server inactivation.

4) Budget Replenishment: When a server is still consuming overrun budget while its normal budget is replenished, the overrun state of this server is reset. Additionally, to support the optionally enabled payback mechanism, we replace the budget replenishment line in the corresponding event handler. The replenished budget is decreased with the consumed overrun budget in the previous period, i.e. in pseudo code:

Pseudo-code 7 on budget replenishment:
1: **if** $S_s.inOverrun$ **then**
2: **if** $S_s.paybackEnabled$ **then**
3: $S_s.consumedOverrun = X_s - Q_{remaining}$;
4: **end if**
5: $S_s.inOverrun =$ **false**
6: **end if**
7: $setSubsystemBudget(Q_s - S_s.consumedOverrun)$;
8: $S_s.consumedOverrun = 0$;

IX. Comparing SIRAP and HSRP

In this section we compare both implementations for HSRP and SIRAP. First, we present a brief overview of our test platform. Next, we compare the implementations of HSRP and SIRAP and demonstrate their effectiveness by means of an example system. Finally, we investigate the system overhead of the synchronization protocol's corresponding primitives.

A. Experimental Setup

In our experiments we use the cycle-accurate OpenRISC simulator provided by the OpenCores project [26]. Within this project an open-source hardware platform is developed. The hardware architecture comprises a scalar processor and basic peripherals to provide basic functionality [27]. The OpenRISC simulator allows simple code analysis and system performance evaluation. Recently, we created a port for μC/OS-II to the

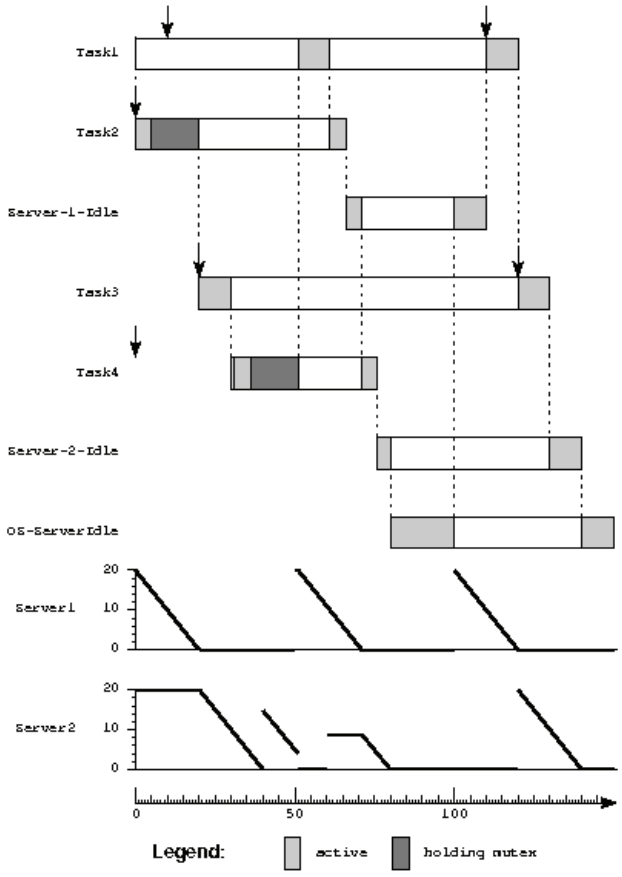


Fig. 4. Example trace for HSRP using the overrun and payback mechanisms.

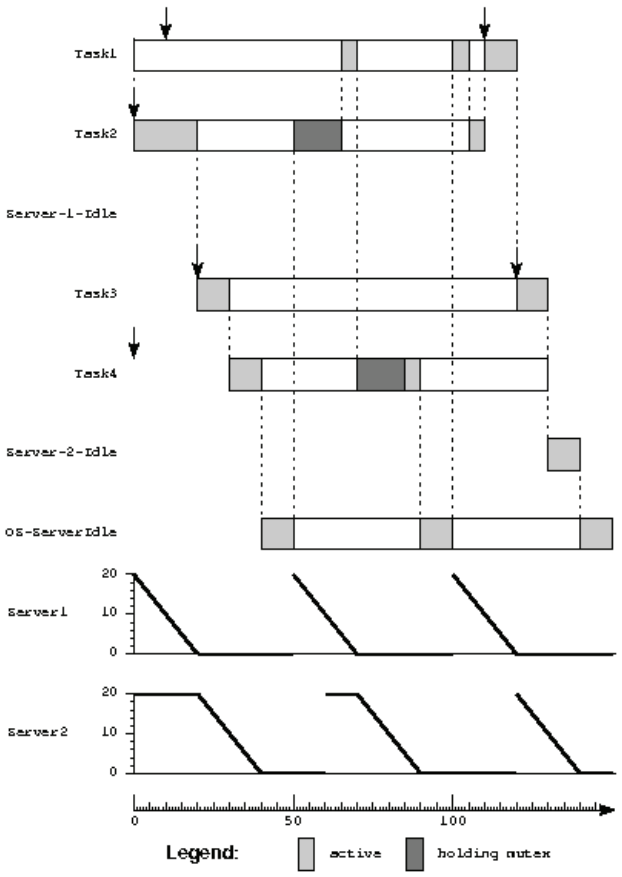


Fig. 5. Example trace for SIRAP using the skipping mechanism. Note that skipping occurs as normal task activation in the execution behaviour of a task.

OpenRISC platform, and extended the toolchain with visualization tools, which make it possible to plot a real-time system's behaviour [28, 29].

B. Protocol Comparison

To demonstrate the behavioural difference between HSRP and SIRAP, consider an example system comprised of two subsystems (see Table I) each with two tasks (see Table II) sharing a single global resource R_1. Note that the subsystem/task with the lowest number has the highest priority and that the computation times of tasks are denoted similarly as in Section III-B[2]. The local resource ceilings of R_1 are chosen to be equal to the highest local priority for SIRAP, while for HSRP this is the default setup.

TABLE I
EXAMPLE SYSTEM: SUBSYSTEM PARAMETERS

Subsystem	Period (P_s)	Budget (Q_s)	Max. blocking (X_s)
Server 1	50	20	15
Server 2	60	20	15

Inherent to the protocol, HSRP immediately grants access to a shared resource and allows the task to overrun its server's budget for the duration of the critical section, see Figure 4. Server 2 replenishes its budget with X_s at time 40. At time 50 task 4 releases R_1 and the remainder of X_s is discarded.

TABLE II
EXAMPLE SYSTEM: TASK PARAMETERS

Server	Task	Period	Computation time	Phasing
Server 1	Task 1	100	10	10
Server 1	Task 2	150	5+15+5	-
Server 2	Task 3	100	10	10
Server 2	Task 4	200	5+15+5	-

The normal budget of server 2 is reduced with its consumed overrun at the next replenishment (time 60).

Contrary, SIRAP postpones resource access when the budget is insufficient, as illustrated in Figure 5. SIRAP's spinlocking implementation is visualized as a longer normal execution time compared to HSRP, e.g. see the execution of task 2 in time interval $(10, 20]$. Figure 6 shows the behaviour of the example system when server 1 selects the SIRAP protocol and server 2 selects the HSRP (with payback).

C. Measurements and Results

In this section we investigate the overhead of the synchronization primitives of HSRP and SIRAP. Current analysis techniques do not account for overhead of the corresponding synchronization primitives, although these overheads become of relevance upon deployment of such a system. All our measurements are independent of the number of subsystems

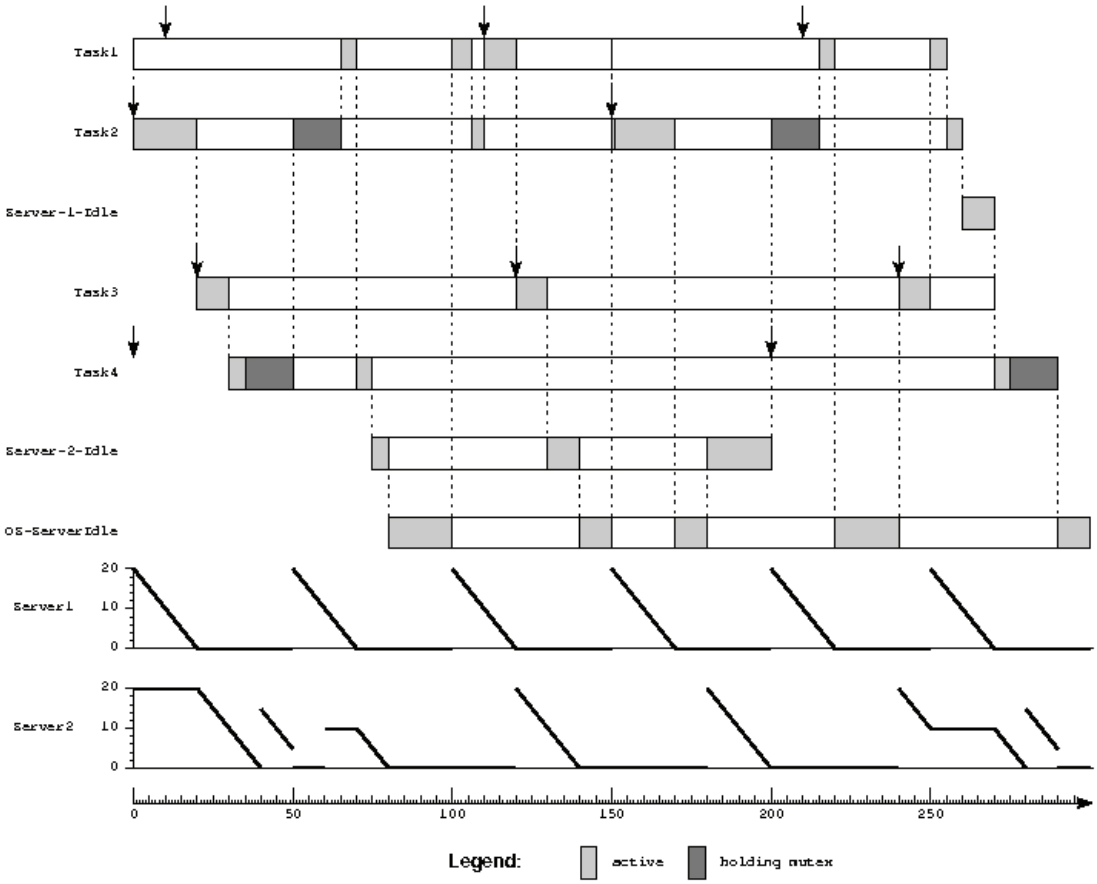


Fig. 6. Example trace combining SIRAP (server 1) and HSRP with payback (server 2) to access a single shared resource.

and tasks within a system.

As we can observe in our implementation, SIRAP induces overhead locally within a subsystem, i.e. the spinlock adds to the budget consumption of the particular task that locks the resource. HSRP introduces overhead that interferes at the global system level, i.e. the overrun mechanism requires the manipulation of event queues. The overheads introduced by the implementation of these protocols is summarized in Table III. A nice analogy of the implementation with respect to the schedulability analysis [7] is that HSRP has an overrun term at the global analysis level, while SIRAP accounts for self-blocking at the local analysis level.

TABLE III
OVERVIEW OF THE SYNCHRONIZATION PRIMITIVE'S IMPLEMENTATION COMPLEXITY FOR HSRP'S AND SIRAP'S RUN-TIME MECHANISMS.

Event	HSRP	SIRAP
Lock resource	-	spinlock
Unlock resource	overrun completion	-
Budget depletion	overrun	-
Budget replenishment	overrun completion, payback (optionally)	spinlock-completion

SIRAP's overhead consists at least of a single test for sufficient budget in case the test is passed. The overhead is at most two of such tests in case the initial test fails, i.e. one additional test is done after budget replenishment before resource access is granted. All remaining tests during spinlocking are already included as self-blocking terms in the local analysis [7]. The number of CPU instructions executed for a single test is 10 instructions on our test platform.

The best-case HSRP overhead is null in addition to the normal number of CPU instructions that are spent to increase and decrease the subsystem and system ceilings. The worst-case HSRP overhead occurs at overrun. When the budget depletes, it is replenished with the maximum allowed overrun budget, which takes 383 instructions[4]. Overrun completion can occur due to two alternative scenarios: (*i*) a task unlocks a resource while consuming overrun budget, or (*ii*) the normal budget is replenished while the subsystem consumes overrun budget. The system overhead for both cases is 966 CPU instructions. When the payback mechanism is enabled, one additional computation is done to calculate the number that needs to be paid back at the next server replenishment, i.e. a system overhead of 5 instructions. As expected, we can conclude that especially ending overrun in HSRP's unlock operation is expensive.

[4]Our current setup only uses a dedicated virtual event queue for each server to keep track of a subsystem's budget, and the queue manipulations therefore have constant system overhead. In case multiple virtual events are stored in this queue, the system overhead for inserting and removing events becomes linear in its length [10].

D. Evaluation

The synchronization protocol implementations are composed of (*i*) variable assignments, (*ii*) (sub)system ceiling stack manipulations, (*iii*) RELTEQ operations [9, 10], (*iv*) a mechanism to allow non-preemptive execution (enable/disable interrupts) and (*v*) scheduler extensions. The first three building blocks are not specifically bound to μC/OS-II. The latter two are μC/OS-II specific. Especially, the extension of the scheduler by SRP's preemption rules is eased by μC/OS-II's open-source character.

X. Discussion

In the previous section we compared the system overhead of the HSRP and SIRAP primitives. Complementary, earlier results have shown that these protocols induce different system loads depending on the chosen (sub)system parameters [7]. To optimize the overall resource requirements of a system, we would like to enable both protocols side-by-side within the same HSF, as demonstrated in Figure 6. Enabling this integration puts demands on the implementation and the schedulability analysis. From the implementation perspective, an unification of the primitive interfaces is requires. However, the choice for a particular protocol at different levels in the system impacts the complexity of the analysis.

A. Uniform Analysis

The synchronization protocols guarantee a maximal blocking time with respect to other subsystems under the assumptions that (*i*) the analysis at the local and global level is correctly performed; (*ii*) the obtained parameters are filled in correctly; and (*iii*) the subsystem behaves according to its parameters. In order to allow SIRAP and HSRP to be integrated side-by-side at the level of subsystems within a single HSF, we need to unify the (global) schedulability analysis of both protocols. Initial results based on our implementation suggests that this integration step is fairly straightforward. The analysis of this integration is left as future work.

B. Uniform Interfaces

Assuming the system analysis supports integration of HSRP and SIRAP within the same HSF at the level of subsystems, one might choose a different synchronization protocol per subsystem depending on its characteristics. Enabling this integration requires that an application programmer (*i*) can *ignore* which synchronization protocol is selected by the system, and (*ii*) cannot *exploit* the knowledge of the selected protocol. The primitive interfaces therefore need to be unified.

The interface description of SIRAP differs from HSRP, because it requires to explicitly check the remaining budget before granting access to a resource, hence the occurrence of the $holdTime$ parameter in its lock interface (see Section VII). However, the integration of HSRP and SIRAP at the level of subsystems only requires that the maximum critical sections length, X_s, within subsystem S_s is known. Assuming X_s is available from the analysis, we can easily store this information within the server-structure. This relaxes the amount of run-time information and allows to remove the $holdTime$ parameter from SIRAP's lock operation, although at the cost of budget over-provisioning due to larger self-blocking times.

XI. Conclusion

This paper describes the implementation of two alternative SRP-based synchronization protocols within a two-level fixed priority scheduled HSF to support inter-application synchronization. In such systems, several subsystems execute on a shared processor where each subsystem is given a virtual share of the processor and is responsible for local scheduling of tasks within itself. We specifically demonstrated a feasible implementation of these synchronization protocols within μC/OS-II.

First, we presented an implementation of SRP within μC/OS-II that optimizes its existing synchronization primitives by a reduced amount of source code, a simplified implementation, and optimized run-time behaviour. Next, we presented the implementation of SIRAP using a run-time skipping mechanism, and HSRP using a run-time overrun mechanism (with or without payback). We discussed the system overhead of the accompanying synchronization primitives, and how these primitives can be integrated within a single HSF.

We aim at using these protocols side-by-side within the same HSF, so that their primitives can be selected based on the relative strengths of the protocol, which depend on system characteristics [7]. We showed that enabling the full integration of both synchronization protocols at the level of subsystems is relatively straightforward.

Our current research focuses on an appropriate selection criterium that minimizes the system overhead based on the subsystem parameters. In the future we would like to investigate less restrictive ways of combining synchronization protocols within HSFs in a predictable manner, e.g. per task, resource or resource access, by extending the existing analysis techniques and corresponding tooling. Finally, we would like to further investigate (*i*) trade-offs between different design and implementation alternatives of HSFs with appropriate synchronization protocols, and (*ii*) their applicability to a wider range of server models.

References

[1] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proc. RTSS*, Dec. 2006, pp. 257–270.

[2] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in *Proc. EMSOFT*, Oct. 2007, pp. 279–288.

[3] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. RTSS*, Dec. 2003, pp. 2–13.

[4] Z. Deng and J.-S. Liu, "Scheduling real-time applications in an open environment," in *Proc. RTSS*, Dec. 1997, pp. 308–319.

[5] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.

[6] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Syst.*, vol. 9, no. 1, pp. 31–67, 1995.

[7] M. Behnam, T. Nolte, M. Åsberg, and R. Bril, "Overrun and skipping in hierarchically scheduled real-time systems," in *Proc. RTCSA*, Aug. 2009, pp. 519–526.

[8] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proc. RTSS*, Dec. 2005, pp. 389–398.

[9] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Multiplexing real-time timed events," in *Proc. ETFA*, July 2009.

[10] M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Virtual timers in hierarchical real-time systems," *Proc. WiP session of the RTSS*, pp. 37–40, Dec. 2009.
[11] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource sharing in reservation-based systems," in *Proc. RTSS*, Dec. 2001, pp. 171–180.
[12] G. Buttazzo and P. Gai, "Efficient implementation of an EDF scheduler for small embedded systems," in *Proc. OSPERT*, July 2006.
[13] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proc. OSPERT*, July 2008, pp. 63–72.
[14] N. Fisher, M. Bertogna, and S. Baruah, "The design of an EDF-scheduled resource-sharing open environment," in *Proc. RTSS*, Dec. 2007, pp. 83–92.
[15] M. Behnam, T. Nolte, M. Åsberg, and I. Shin, "Synchronization protocols for hierarchical real-time scheduling frameworks," in *Proc. CRTS*, Nov. 2008, pp. 53–60.
[16] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Scheduling of semi-independent real-time components: Overrun methods and resource holding times." in *Proc. ETFA*, Sep. 2008, pp. 575–582.
[17] R. J. Bril, U. Keskin, M. Behnam, and T. Nolte, "Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited," in *Proc. CRTS*, 2009.
[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.
[19] Micrium, "RTOS and tools," March 2010. [Online]. Available: http://micrium.com/
[20] J. J. Labrosse, *Microc/OS-II*. R & D Books, 1998.
[21] J.-H. Lee and H.-N. Kim, "Implementing priority inheritance semaphore on uC/OS real-time kernel," in *Proc. WSTFES*, May 2003, pp. 83–86.
[22] M. Bergsma, M. Holenderski, R. J. Bril, and J. J. Lukkien, "Extending RTAI/Linux with fixed-priority scheduling with deferred preemption," in *Proc. OSPERT*, June 2009, pp. 5–14.
[23] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *Proc. WPDRTS*, March 2007, pp. 1–8.
[24] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Syst.*, vol. 1, no. 1, pp. 27–60, 1989.
[25] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.
[26] OpenCores. (2009) OpenRISC overview. [Online]. Available: http://www.opencores.org/project,or1k
[27] M. Bolado, H. Posadas, J. Castillo, P. Huerta, P. Sánchez, C. Sánchez, H. Fouren, and F. Blasco, "Platform based on open-source cores for industrial applications," in *Proc. DATE*, 2004, p. 21014.
[28] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *Proc. WATERS*, July 2010.
[29] "Simulating uC/OS-II inside the OpenRISC simulator," March 2010. [Online]. Available: http://www.win.tue.nl/~mholende/ucos/

Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux

Andreu Carminati, Rômulo Silva de Oliveira, Luís Fernando Friedrich, Rodrigo Lange
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil
{andreu,romulo,lange}@das.ufsc.br
{fernando}@inf.ufsc.br

Abstract

In general purpose operating systems, such as the mainline Linux, priority inversions occur frequently and are not considered harmful, nor are avoided as in real-time systems. In the current version of the kernel PREEMPT-RT, the protocol that implements the priority inversion control is the Priority Inheritance. The objective of this paper is to propose the implementation of an alternative protocol, the Immediate Priority Ceiling, for use in drivers dedicated to real-time applications, for example. This article explains how the protocol was implemented in the real-time kernel and compares tests on the protocol implemented and Priority Inheritance, currently used in the real-time kernel.

1 Introduction

In real-time operating systems such as Linux/PREEMPT-RT [7, 8], task synchronization mechanisms must ensure both the maintenance of internal consistency in relation to resources or data structures, and determinism in waiting time for these. They should avoid unbounded priority inversions, where a high priority task is blocked indefinitely waiting for a resource that is in possession of a task with lower priority.

In general purpose systems, such as mainline Linux, priority inversions occur frequently and are not considered harmful, nor are avoided as in real-time systems. In the current version of the kernel PREEMPT-RT, the protocol that implements the priority inversion control is the Priority Inheritance (PI) [9].

The objective of this paper is to propose the implementation of an alternative protocol, the Immediate Priority Ceiling (IPC) [5, 9], for use in drivers dedicated to real-time applications, for example. In this scenario, an embedded Linux supports an specific known application that does not change task priorities after its initialization. It is not the objective of this paper to propose a complete replacement of the existing protocol, as mentioned above, but an alternative for use in some situations. The work in this article only considered uniprocessor systems.

This paper is organized as follows: section 2 presents the current synchronization scenario of the mainline kernel and PREEMPT-RT, section 3 explains about the Immediate Priority Ceiling protocol, section 4 explains how the protocol was implemented in the Linux real-time kernel, section 5 compares tests made upon the protocol implemented and Priority Inheritance implemented in the real-time kernel and section 6 presents an overhead analysis between IPC and PI.

2 Mutual Exclusion in the Linux Kernel

Since there is no mechanism in the mainline kernel that prevents the appearance of priority inversions, situations like the one shown in Figure 1 can occur very easily, where task T2, activated at $t = 1$, acquires the shared resource. Then, task T0 is activated at $t = 2$ but blocks because the resource is held by T2. T2 resumes execution, and is preempted by T1, which is activated and begins to run from $t = 5$ to $t = 11$. But task T0 misses the deadline at $t = 9$, and the resource required for its completion was only available at $t = 12$ (after the deadline).

In the real-time kernel PREEMPT-RT exists the implementation of PI (Priority Inheritance). As mentioned above, it is a mechanism used to accelerate the release of resources in real-time systems, and to avoid the effect of indefinite delay of high priority tasks that can be blocked waiting for resources held by tasks of lower priority.

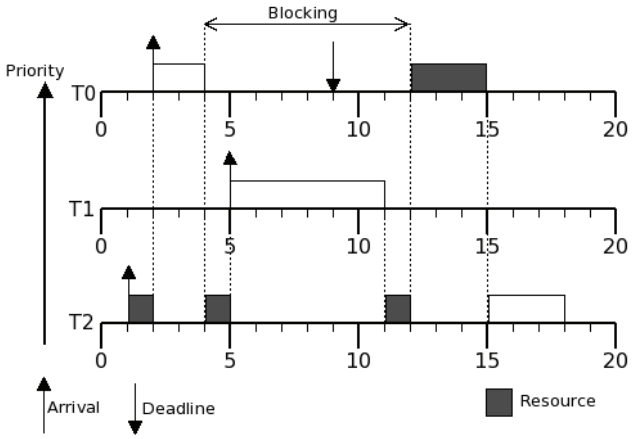


Figure 1. Priority inversion

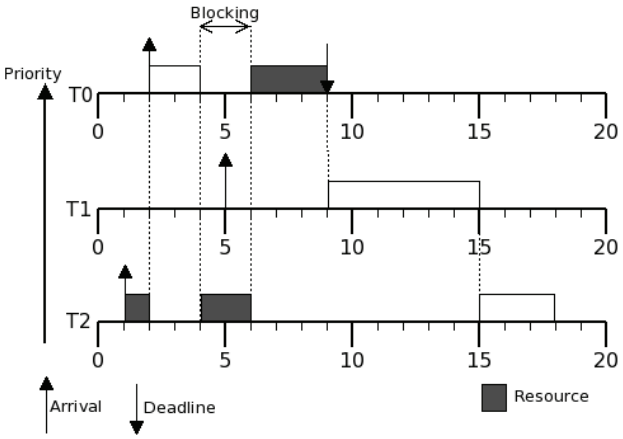


Figure 2. Priority inversion resolved by PI

In the PI protocol, a high priority task that is blocked on some resource, gives its priority to the low priority task (holding that resource), so will release the resource without suffering preemptions by tasks with intermediate priority. This protocol can generate chaining of priority adjustments (a sequence of cascading adjustments) depending on the nesting of critical sections.

Figure 2 presents an example of how the PI protocol can help in the problem of priority inversion. In this example, task T2 is activated at $t = 1$ and acquires a shared resource, at $t = 1$. Task T0 is activated and blocks on the resource held by T2 at $t = 4$. T2 inherits the priority from T0 and prevents T1 from running, when activated at $t = 5$. At $t = 6$, task T2 releases the resource, its priority changes back to its normal priority, and task T0 can conclude without missing its deadline.

Some of the problems [11] of this protocol are the number of context switches and blocking times larger than the largest of the critical sections [9] (for the high priority task), depending on the pattern of arrivals of the task set that shares certain resources.

Figure 3 is an example where protocol PI does not prevent the missing of the deadline of the highest priority task. In this example, there is the nesting of critical sections, where T1 (the intermediate priority) has the critical sections of resources 1 and 2 nested. In this example, task T0, when blocked on resource 1 at $t = 5$, gives his priority to task T1, which also blocks on resource 2 at $t = 6$. T1 in turn gives its priority to task T2, which resumes its execution and releases the resource 2, allowing T1 to use that resource and to release the resource 1 to T0 at $t = 10$. T0 resumes its execution but it misses its deadline, which occurs at $t = 13$. In this example, task T0 was blocked by a part of the time of the external critical section of T1 plus a part of the time of the critical section of T2. In a larger system the blocking time of T0 in the worst case would be the sum of many critical sections, mostly associated with resources not used by T0.

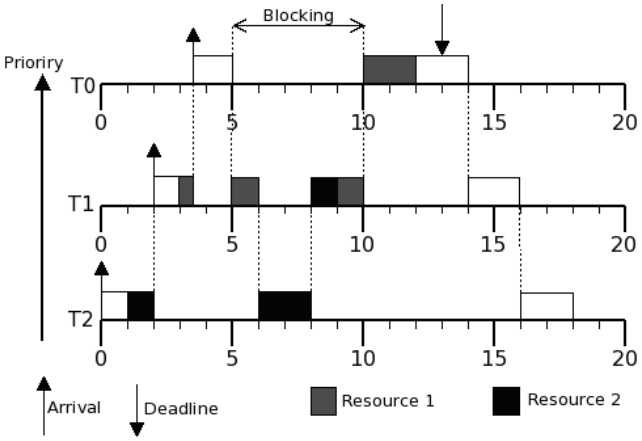


Figure 3. Priority inversion not resolved by PI

3 The Immediate Priority Ceiling Protocol

The Immediate Priority Ceiling (IPC) [2] synchronization protocol for fixed priority tasks, is a variation of Priority Ceiling Protocol [9, 1] or *Highest Locker Priority*. This protocol is an alternative mechanism for unbounded priority inversion control, and prevention of deadlocks in

uniprocessor systems.

In the PI protocol, the priority is associated only to tasks. In IPC, the priority is associated with both tasks and resources. A resource protected by IPC has a priority ceiling, and this priority is the highest priority of all task priorities that access this resource.

According to [3], the maximum block time of a task under fixed priority using shared resource protected by IPC protocol is the larger critical section of the system whose priority ceiling is higher than the priority of the task in question and is also used by a lower priority task.

What happens in IPC can be considered as preventive inheritance, where the priority is adjusted immediately when occurs a resource acquisition, and not when the resource becomes necessary to a high priority task, as in PI (you can think of PI as the IPC, but with dynamic adjustment of the ceiling). This preventive priority setting prevents low priority tasks from being preempted by tasks with intermediate priorities, which have priorities higher than low priority tasks and lower than the resource priority ceiling.

Figure 4 shows an example similar to that shown in Figure 3, but this time using IPC. In this example, the high priority task does not miss its deadline, because when task T2 acquires resource 2 at *t = 1*, its priority is raised to the ceiling of the resource (priority of T1), preventing task T1, activated at *t = 2*, from starting its execution. At *t = 3.5*, task T0 is activated and begins its execution. The task is no longer blocked because resource 1 is available. Task T0 does not miss its deadline.

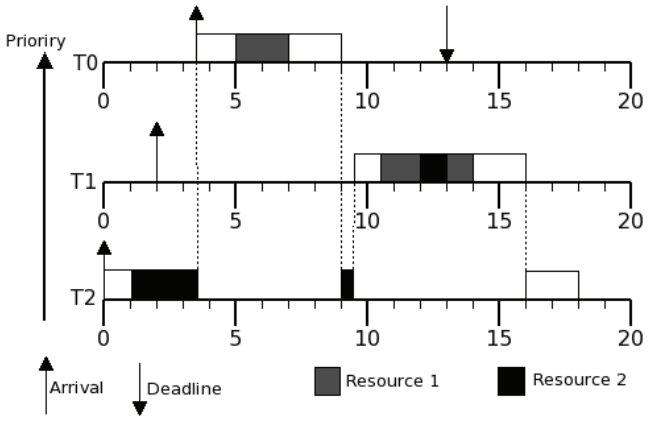


Figure 4. Priority inversion resolved by IPC

4 Description of the Implementation

The Immediate Priority Ceiling Protocol was implemented based on the code of rt_mutexes already in the patch PREEMPT-RT. The rt_mutexes are mutexes that implement the Priority Inheritance protocol. The kernel version used for implementation was the 2.6.31.6 [10] with PREEMPT-RT patch rt19. Although rt_mutexes are implemented in PREEMPT-RT for both uniprocessor and multiprocesors, our implementation of IPC considers only the uniprocessor case.

The implementation was made primarily for use in device-drivers (kernel space), as shown in Figure 5, where there is an example of tasks sharing a critical section protected by IPC and accessed through an ioctl system call to a device-driver.

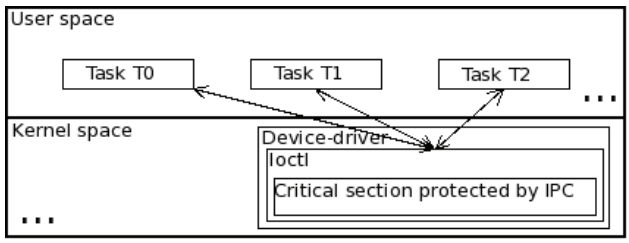


Figure 5. Diagram of interaction between the IPC protocol and tasks

The type that represents the IPC protocol was defined as *struct ipc_mutex*, and it is presented in code 1. In this structure, *wait_lock* is the spinlock that protects the access to the structure, *wait_list* is an ordered (by priorities) list that stores pending lock requests, *on_task_entry* serves to manage the locks acquired (and, consequently, control of priorities), *owner* stores a pointer to the task owner of the mutex (or null pointer if the mutex is available) and finally the *ceiling*, which stores the priority ceiling of the mutex.

Code 1 Data structure that represents a IPC mutex

```
struct ipc_mutex {
        atomic_spinlock_t wait_lock;
        struct plist_head wait_list;
        struct plist_node on_task_entry;
        struct task_struct *owner;
        int ceiling;
...
};
```

The proposed implementation presents the following

API of functions and macros:

- **DEFINE_IPC_MUTEX(mutexname, priority):** This macro is provided for the definition of a static IPC mutex, where *mutexname* is the identifier of the mutex and priority is the ceiling of the mutex, or a value in the range of 0 to 99 (according to the specification of static priorities for real-time tasks on Linux). The current version can only create mutexes with priorities set at compile time, thus, the priority ceiling should be assigned by the designer of the application. This is not a too restrictive assumption when an embedded Linux runs a known application that does not change task priorities after its initialization.

- **void ipc_mutex_lock(struct ipc_mutex * lock):** Mutex acquisition function. In uniprocessor systems this is a nonblocking function because, according to the IPC protocol, if a task requests a resource, it is because this resource is available (the owner is null). In multiprocessor systems this function can generate blocks, because the resource can be in use on other processor (the *owner* field differs from zero). In this article, only the uniprocessor version will be taken into consideration. The main role of this function is to manage the priority of the calling task along with the resource blocking, taking into account all ipc_mutexes acquired so far.

- **void ipc_mutex_unlock(struct ipc _mutex * lock):** Effects the release of the resource and the adjustment of the priority of the calling task. In multiprocessor systems, this function also makes the job of selecting the next task (by the *wait_list*) that will use the resource. What also occurs in multiprocessor systems is the effect "next owner pending" (also present in the original implementation of priority inheritance) also known as steal lock, where the task of highest priority can acquire the mutex again, even though it has been assigned to another task that did not take possession of it.

One major difference between the proposed implementation and the already existent in the PREEMPT-RT is that the latter one enables the following optimizations:

- PI can perform atomic lock: if a task attempts to acquire a mutex that is available, this operation can be performed atomically (operation atomic compare and exchange) by what is known as fast path. But this is only possible for architectures that have this type of atomic operation. Otherwise if the lock is not available and/or the architecture does not have exchange and compare instruction, the lock will not be atomic (slow path).

- PI can perform atomic unlock: when a task releases a mutex which has no tasks waiting, this operation can be performed atomically.

As mentioned earlier, these optimizations are possible only for PI mutexes. In the case of IPC, there will always be a need for verification and a possible adjustment of priority.

5 Implementation Analysis

We developed a device-driver that has the function of providing the critical sections necessary to perform the tests. This device-driver exports a single service as a service call ioctl (more specifically unlocked_ioctl, because the ioctl is protected by traditional Big Kernel Lock, which certainly would prevent the proper execution of the tests). It multiplexes the calls of the three tasks in their correspondent critical sections. This device-driver provides critical sections to run with both IPC and PI.

In order to carry out tests for the analysis of the implementation it was used a set of sporadic tasks executed in user space. The interval between activations, the resources used and the size of the critical section within the device-driver used by each task are presented in Table 1. All critical sections are executed within the function ioctl, within Linux kernel. A high-level summary of actions performed by each task (in relation to resources used) is presented in Table 2.

Table 1 shows the intervals between activations expressed with a pseudo-randomness, ie, with values uniformly distributed between minimum and maximum values. This randomness was included to improve the distribution of results because, with fixed periods, arrival patterns were being limited to a much more restricted set. Table 1 also presents the sizes of the critical sections of each task. Other information shown in Table 1 is the number of activations performed for each task. For the high priority task, there were 1000 monitored activations (latency, response time, critical section time, lock time, etc). For other tasks there was no restriction on the number of activations.

The high priority task has one of the highest priorities of the system. The other tasks were regarded as medium and low priorities. But they also have absolute high priorities. All tasks have been configured with the scheduling policy SCHED_FIFO, which is one of the policies for real-time [4] available in Linux.

Even with the use of a SMP machine for testing, all tasks were set at only one CPU (CPU 0). The tests were conducted using both IPC and PI for comparison purposes.

Task	T0/High	T1/Med.	T2/Low
Priority	70	65	60
Activation interval	rand in [400,800] ms	rand in [95,190] ms	rand in [85,170] ms
Resource	R1	R1,R2	R2
Critical section size	aprox. 17 ms	aprox. 2x17 ms	aprox. 17 ms

Table 1. Configuration of the set of tasks

Task	T0/High	T1/Med.	T2/Low
Action 1	Lock(R1)	Lock(R1)	Lock(R2)
Action 2	Critical Sec.	Critical Sec.	Critical Sec.
Action 3	Unlock(R1)	Lock(R2)	Unlock(R2)
Action 4		Critical Sec.	
Action 5		Unlock(R2)	
Action 6		Unlock(R1)	

Table 2. Actions realized by tasks

Mutex R1 has been configured with priority ceiling 70 (which is the priority of task T0) and R2 has been configured with priority ceiling 65 (which is the priority of task T1).

5.1 Results of the Use of the PI mutex

With priority inheritance, the high priority task had activation latencies as can be seen in the histogram of Figure 6 appearing in the interval [20000, 30000] nanoseconds (range where the vertical bar is situated in the histogram). Because of finding the resource busy with a certain frequency (as illustrated in Figure 7, waiting time for the resource), the task was obligated to perform volunteer context switch for propagation of its priority along the chain of locks.

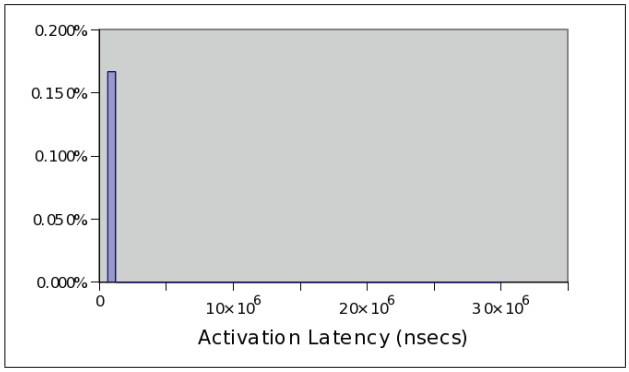


Figure 6. Histogram of activation latencies (high priority task using PI)

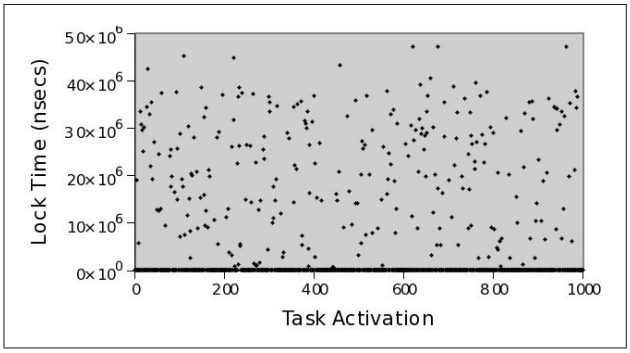


Figure 7. Blocking time (high priority task using PI)

Regarding the response time (as can be seen in the histogram of Figure 8) it was consistent with the blocking time sustained, with a maximum of nearly 3 times the size of the critical section, in accordance with the task set. It can be seen in Table 3 the worst-case response time observed is 64,157,591 ns. The theoretical worst-case response time for this test would be, with an appropriate synchronized activation, 68 ms, or 17 ms own critical section of task T0 added to 34 ms of task T1 and 17 ms of task T2. In this test, there is a good approximation of the theoretical limit.

5.2 Results of the Use of the IPC mutex

Using IPC, it can be noted in the histogram of Figure 9 that the task of highest priority presented, with low

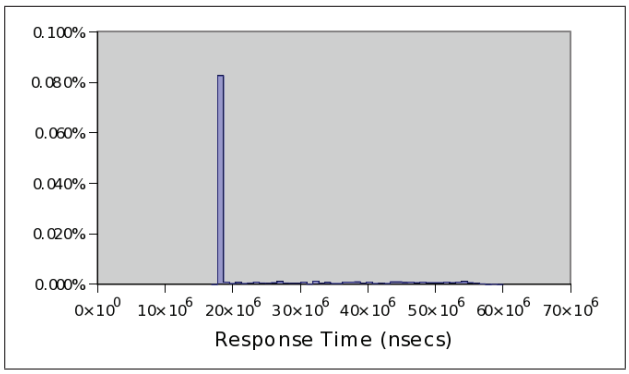


Figure 8. Histogram of response time (high priority task using PI)

Protocol:	PI	IPC
Average response time:	22,798,549 ns	21,014,311 ns
Std dev:	11,319,355 ns	8,723,159 ns
Max:	64,157,591 ns	50,811,328 ns

Table 3. Average response times and standard deviations

frequency, varying values of activation latency (seen in the tail of the histogram). Waiting times set by the resource appear in Figure 10, which is expected according to the definition of the protocol implemented. A tail appears in the histogram of response time (Figure 11) due to activation latency (higher values, but with only a few occurrences).

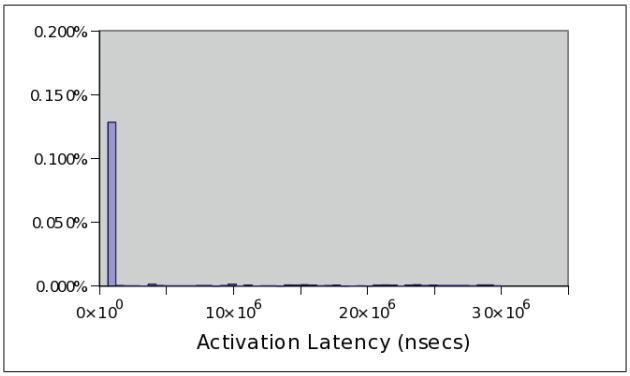


Figure 9. Histogram of activation latencies (high priority task using IPC)

As it can be seen in Table 3, the worst-case response

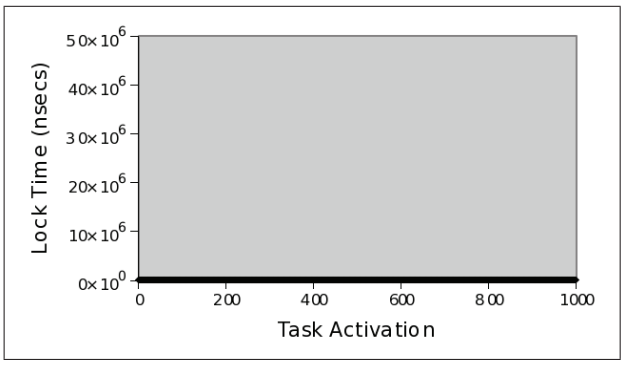


Figure 10. Blocking time (high priority task using IPC)

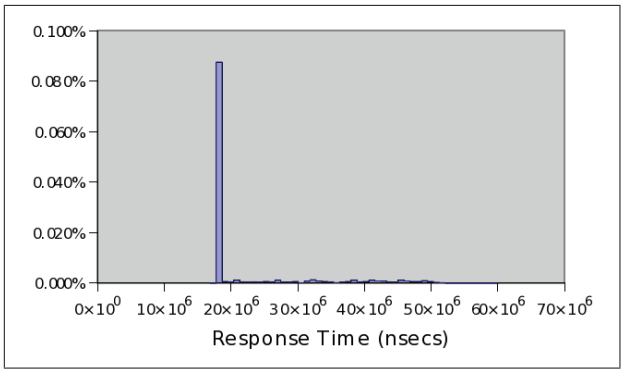


Figure 11. Histogram of response time (high priority task using IPC)

time observed is (maximum) 50,811,328 ns. In this test, the theorical limit is 51 ms, ie, 17 ms own critical section of task T0 added to 34 ms of task T1. Also in this test there is a good approximation of the theorical limit.

5.3 Comparison between PI and IPC

One can observe that, in general, IPC has behavior similar to PI. The differences appear in the lock time where, by definition, in uniprocessor systems, the resource is always available when using IPC protocol. For the PI protocol, the blocking time will appear with the primitive lock, and this time may be longer than that with IPC. In IPC the blocking time appears before the activation time, and it has a maximum length of a single critical section (in the conditions described above).

According to Table 3, the IPC protocol presented

standard deviation and average response time smaller than PI. Another important point in Table 3 is that the worst-case response time observed in the IPC test was almost a critical section smaller than the PI (the size of a critical section is 17 ms, and the difference between the worst case of IPC and PI is around 14 ms).

Figure 12 shows the tail of the response time histograms of IPC and PI combined. In this figure, the response times of the IPC protocol concentrates on lower values. For the PI, these are distributed more uniformly to higher values, indicating an average response time smaller for the IPC protocol. This histogram also indicates in its final portion that the worst case, as it was also observed in Table 3, has a difference of one critical section in favor of the IPC protocol. This difference in the worst case was reported in the figure by two vertical lines, where the distance between them is about the duration of one critical section. Table 4 summarizes the results qualitatively.

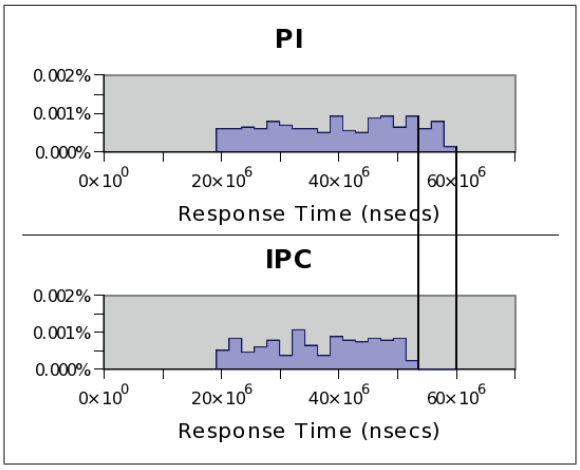


Figure 12. Histogram of the response time of the high priority task

Protocol	PI	IPC
Activation Latency	Not varied	Varied
Blocking time	Varied	Not varied
Response time	Blocking time dependent	Latency dependent

Table 4. Summary of expected results

6 Implementation Overhead

We define overhead as any decrease in the system's ability to produce useful work. Thus, for this study, the overhead will be considered as the reducing of the CPU time available to the rest of the system, given the presence of a set of higher priority tasks sharing resources protected by PI or IPC.

To evaluate the protocol implemented in terms of overhead imposed on the system, we used a set of test tasks as specified in Table 5. In the same table, it is presented the tasks configurations, some of which are identical to the tasks used to evaluate the protocol in the previous section. For example, for task T0', the size of the critical section is equal to the size of the critical section of task T0 of the previous test (represented by "== T0").

To perform an estimative of the overhead, it was created a measuring task with priority 51 (with policy SCHED_FIFO). This priority is above the default priority of threaded irq handlers [8] and softirqs [6] . This was done to keep the measuring task above the interference of the mechanisms of interrupt handling and work postponement of Linux. Every CPU time that remains (not used by the test tasks synchronized by IPC or PI) is then assigned to the measurement task. Both the measurement task and the task set synchronized by IPC or PI were fixed to a single CPU (CPU 0 in a system with 2 cores.)

The measurement task is activated before the activation of real-time tasks and ends after they terminate. In each test iteration, the measurement task runs for 9 seconds, and the higher priority tasks begin 1 second after it starts. As shown in Table 5, task T0' executes 10 activations, the others will run until the end of this task.

Task	T0'	T1'	T2'	T3'	T4'	T5'	T6'
Priority	70	65	64	63	62	61	60
Activation interval	== T0	== T1	== T1	== T1	== T2	== T2	== T2
Resource	R1	R1, R2	R1, R2	R1, R2	R2	R2	R2
Critical section size	== T0	== T1	== T1	== T1	== T2	== T2	== T2
Number of activ.	10	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep

Table 5. Actions realized by tasks

To obtain the overhead estimative, the measurement

task is executed in an infinite loop incrementing a variable by the time specified above (9 seconds). The overhead will be noticed by how much the measurement task could increment a count, taking into account the execution of the set of tasks synchronized by IPC or PI. The values of the counts made by the measurement task are presented in Table 6, which was ordered to facilitate visual comparison.

IPC	PI
187,882,717	188,776,389
188,035,384	189,155,169
188,160,733	189,202,563
188,194,113	189,263,630
188,207,825	189,331,186
188,240,432	189,361,353
188,563,788	189,387,326
188,603,802	189,418,120
188,616,385	189,428,218
188,703,718	189,437,569
188,736,889	189,447,533
188,742,876	189,471,453
188,935,538	189,475,286
188,952,045	189,489,740
188,962,343	189,492,896
188,993,374	189,494,791
189,000,638	189,569,661
189,178,721	189,572,258
189,203,245	189,604,953
189,307,878	189,638,715
189,478,899	189,696,190
189,536,986	189,778,227
189,674,412	189,825,606
189,785,580	189,867,362
189,858,951	190,046,853
189,900,585	190,207,326
190,030,444	190,252,943
190,047,436	190,342,313
190,066,156	190,349,981
190,105,987	190,387,518
190,328,052	190,538,130
190,338,011	190,539,875

Table 6. Counter values of measurement task

Table 7 presents the basic statistical data related to the samples presented in Table 6.

To evaluate the results we used the statistical hypothesis test for averages with unknown variance (Student t-test). By hypothesis, the overhead of PI and IPC are equal, ie, the average of IPC and PI are equal ($H_0 : \mu_{PI} = \mu_{IPC}$).

Protocol	IPC	PI
Average(μ)	189,136,685.72	189,682,847.91
Var.(S_x^2)	524,003,070,588.27	191,603,683,258.35
Minimum	187,882,717	188,776,389
Maximum	190,338,011	190,539,875

Table 7. Basic statistics of the found values

The data presented in Table 8, which provides the data necessary for the hypothesis test, was obtained from the data showed in Table 7 plus the information of the number of samples (n = *32*)

$Sa^2_{IPC,PI}$	357,803,376,923.31
$Sa_{IPC,PI}$	598,166.68
n	32
α	0.1%
$d.f.$	60
t	-3.65

Table 8. Student's t-test data

6.1 Analysis of the Results

Because the data produced the value of *t* = 3.65, which does not belong to the region of acceptance (in t-Student distribution), the test rejects H_0 with a significance level of 0.1 %. At significance level (α) of 0.1%, the collected data indicates a difference between PI and IPC. There is a probability smaller than 0.1% that the differences observed on the presented data are from casual factors of the system only.

These differences are likely due to the fast path of the PI implementation. In IPC, there is always a need of priority verification, and this can not be performed atomically. Another point is that if a task with priority below the priority ceiling of a given resource acquires that resource, its priority has to be changed, and this may influence the overhead. As those tests show, there is a reasonable probability of tasks finding resources available, not always the priorities propagation algorithm (PI) will run. But almost always there will be priority adjustments (IPC), except for the task that defines the priority ceiling of the resource.

7 Conclusions

Task synchronization is fundamental in multitasking and/or multithread systems, specially in real-time systems. In addition to protection against race conditions, these mechanisms must prevent the emergence of uncontrolled priority inversions, which could cause the missing of deadline, leading real-time applications to present incorrect behavior, and possibly harmful consequences (depending on the application). In this context, it was proposed an alternative for some applications to the protocol implemented in the real-time Linux branch.

The IPC protocol may be suitable for dedicated applications that use architectures without instruction compare and exchange because, in this way, the implementation may not use the fast path (via atomic instructions). Another advantage of the IPC is that it generates less context switches than the PI, inducing faster response times due to switching overhead as well as lower failure rates in the TLB.

One of the disadvantages of the IPC for wider use is the need for manual determination of the priority ceiling of IPC mutexes. But this is not a problem for automation and control applications for example. Dedicated device-drivers are fully aware of the priorities of the tasks that access them, justifying the manual setting of the ceiling in this case.

As seen in the tests, the PI protocol may be more appropriate if the latency of activation is important. But if the blocking time is more relevant, IPC may be the best solution. In terms of average response time, the two solutions were similar, but IPC showed lower average response time probably due to the latency of activation being less than the waiting time of the PI. Another point in favor of the IPC protocol appears when we compare the difference in the worst-case response time observed in the tests since the IPC case was about a critical section smaller than in the PI case, as can be seen in Table 3. The PI protocol has a response time that may vary depending of the pattern of resource sharing and sequences of activation, which does not occur with IPC. Its blocking time will always be at most one critical section.

Although blocking/response times are smaller in the IPC, tests show that the overhead of IPC implemented is greater than the native PI in PREEMPT-RT. This overhead is most likely caused by the absence of a fast path in the implementation of IPC. There is a set of operations on lock/unlock that can not be executed atomically as in PI. These operations involve priority changes and tracking mutexes acquired by tasks.

As future work, we intend to implement a version with adaptive ceiling, ie, ceiling can be automatically adjusted in run-time. There is still a possibility (which was not considered in this article) to build a fast-path. To make this possible, the priority adjustments should be postponed until the eminence of a system rescheduling (through changes in the scheduler). We also intend to expand the study of the IPC protocol to multiprocessor systems.

8 Acknowledgments

To CAPES and CNPq for financial suport.

References

[1] T. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, volume 270, 1990.

[2] A. Burns and A. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison Wesley, 2001.

[3] M. Harbour and J. Palencia. Response time analysis for tasks scheduled under EDF within fixed priorities. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 200. Citeseer, 2003.

[4] C. S. IEEE, editor. *POSIX.13. IEEE Std. 1003.13-1998. Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)*. The Institute of Electrical and Electronics Engineers, 1998.

[5] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. 1980.

[6] R. Love. *Linux Kernel Development (Novell Press)*. Novell Press, 2005.

[7] I. Molnar. Preempt-rt. http://www.kernel.org/pub/linux/kernel/projects/rt - Last access 01/21, 2010.

[8] S. Rostedt and D. Hart. Internals of the RT Patch. In *Proceedings of the Linux Symposium*, volume 2007, 2007.

[9] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[10] L. Torvalds. "Linux Kernel Version 2.6.31.6", 2010. http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.31.6.tar.bz2 - Last access 03/21, 2010.

[11] V. Yodaiken. Against priority inheritance. *FSMLABS Technical Paper*, 2003. Available at `http://yodaiken.com/papers/inherit.pdf`.

The Case for Thread Migration: Predictable IPC in a Customizable and Reliable OS

Gabriel Parmer

Computer Science Department
The George Washington University
Washington, DC
gparmer@gwu.edu

Abstract

Synchronous inter-process communication (IPC) between threads is a popular mechanism for coordination in μ-kernels and component-based operating systems. Significant focus has been placed on its optimization, and consequently the efficiency of practical implementations approaches the lower limits set by hardware. This paper qualitatively compares the predictability *properties of the synchronous IPC model with those of the migrating thread model. We assess the idealized communication models, and their practical implementations both in different versions of L4, and in the* COMPOSITE *component-based OS. We study three main factors – execution accounting, communication end-points, and system customizability – and discuss the trade-offs involved in each model. We make the case that the migrating thread model is as suitable as synchronous IPC, if not more so, in configurable systems requiring strict predictability.*

1 Introduction

Component-based operating systems are an appealing foundation for embedded and real-time systems as they enable high degrees of system specialization and enhanced reliability. The system's software is decomposed into fine-grained components. Each component provides a policy, abstraction, or mechanism that is accessed by other components through its interface. A system with specific timing constraints, or that is reliant on specific resource management policies, chooses the appropriate components to satisfy those particular requirements. By segregating components into separate protection domains (provided by e.g. hardware page-tables), the reliability of the system is increased as the scope of the side-effects of faults is limited to individual components. Communication and interaction between components is conducted via inter-process communication (IPC) in which the kernel mediates control transfer between protection domains.

Many different IPC mechanisms exist including synchronous IPC between threads, and thread migration. Implementations using synchronous IPC[1] exist that are extremely efficient, approaching the performance lower-bound imposed by hardware [10]. This method is used in many systems focusing on extreme IPC performance [16, 20, 17, 19], and it is employed in at least two commercially successful OSes, QNX and OKL4[2]. To accomplish most tasks, coordination between system components is required. Thus the predictability of the IPC operation impacts the real-time characteristics of all software in a component-based system. This paper seeks to provide a qualitative analysis and comparison of the predictability properties of this established IPC mechanism with those of thread migration [7]. For invocation using thread migration, a single schedulable thread executes across components.

We base most comparisons in this paper on, first, a pure model of synchronous IPC presented in Section 2.1, and, second, a variety of implementations of L4, a mature and highly-efficient μ-kernel [11]. We choose L4 as various implementations and optimizations have been made that demonstrate many interesting trade-offs in the design of synchronous IPC. As a concrete implementation of thread migration, we compare against the COMPOSITE component-based OS.

Functionally, both synchronous IPC and thread migration often look identical to client component code. Both are made to mimic normal function invocation by a interface

[1] A note on terminology: In this paper, we will refer to synchronous IPC between threads as simply *synchronous IPC*, and will use *IPC* and *invocation* interchangeably to denote control transfer back and forth between components. Additionally, we will use common terms in the μ-kernel literature to denote components as the *client* (making an invocation), and *server* (receiving and handling the invocation).

[2] See www.qnx.com and www.ok-labs.com.

definition language [4], hiding the concrete mechanisms used for the invocation. Behaviorally, they differ greatly and in this paper we focus on these differences. Many predictable systems have been built using synchronous IPC, but we argue here that thread migration is just as strong a foundation, if not more-so for predictable, configurable, and reliable systems. We base this argument on three main factors: (1) how processing time is accounted to execution throughout the system, (2) the effects of contention on the communication end-points in the system, and (3) the effect of the invocation mechanism on the ability of the system to provide configurable and specialized services.

This paper makes the following contributions:

- identify key factors that effect the predictability and flexibility of synchronous IPC;
- analyze the migrating thread model with respect to these factors, and compare against synchronous IPC;
- suggest a number of changes to a system built on synchronous IPC, that are inspired by the migrating thread model, to increase system predictability.

This paper is organized as follows: Section 2 introduces models to describe synchronous IPC and thread migration, so that they can be compared qualitatively. Section 3 discusses the different CPU allocation and accounting properties of both models, while Section 4 investigates the properties of IPC end-point contention, and Section 5 discusses system specialization and configuration opportunities present in the migrating thread model. Section 6 discusses the limitations of the migrating thread model, while Section 7 outlines related work, and Section 8 concludes.

2 IPC Models

2.1 Synchronous IPC Between Threads

Here we introduce an idealized version of the synchronous IPC model. Though many real-world implementations do not implement it directly, it serves as the starting point for their mechanisms. In the following sections we will discuss how various implementations diverge from this strict model where appropriate.

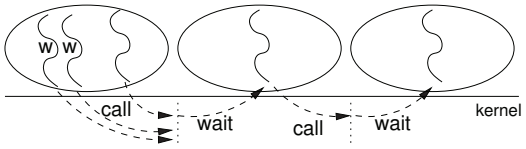


Figure 1. Synchronous IPC between threads. Threads annotated with a `w` are on a wait-queue associated with the server thread.

A system consists of a number of components, $C_a, \ldots, C_z$, each in separate protection domains. Thus communication between components must be conducted via the kernel (as switching between protection domains is typically a privileged instruction). Each component contains a number of threads $\tau_0^{C_a}, \ldots, \tau_n^{C_a}$. When thread $\tau_0^{C_a}$ wishes to harness the functionality provided by C_1, $\tau_0^{C_a}$ sends a message to $\tau_0^{C_b}$, and waits to receive a reply. This *send and receive* is often conflated into a single *call* system call. $\tau_0^{C_b}$ waits for requests from client threads, processes a request when one arrives, and replies to the client. The operations of *reply and wait* are often conflated into a single *reply_wait* system call. These API additions optimize for synchronous IPC and reduce the number of required user/kernel transitions [10]. If $\tau_0^{C_b}$ is processing and not waiting for an IPC when $\tau_0^{C_a}$ *call*s it, $\tau_0^{C_a}$ will block in a queue for $\tau_0^{C_b}$ [3], which will refer to as the *wait-queue* for a server thread.

Figure 1 illustrates synchronous IPC between three protection domains. A server thread, $\tau_0^{C_b}$, will become a client by harnessing the functionality of a third component and *call*ing $\tau_0^{C_c}$. We will say these nested IPCs create a *chain* of invocations. More generally, when taken together with the wait-queues for each server thread, a *dependency graph* [20] is created where threads waiting for a reply from a server thread (either because the server thread is processing on their behalf, or because they are in the wait-queue) are said to have a dependency on the server thread.

2.2 Thread Migration

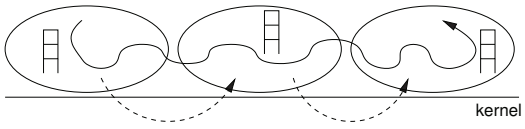


Figure 2. Thread migration. Execution contexts are spread across components, but the same schedulable entity traces invocations.

Thread migration [7] is a natural model for making invocations between components in a system. The same schedulable entity in one component continues execution in the other. The same thread, τ_0, executes through system components just as a thread in an object oriented language traverse many objects. If components are resident in the same protection domain this enables direct function invocation with little overhead [14, 15]. If system components exist in separate protection domains, then thread migration is less natural, but can still be accomplished. We will assume components in separate protection domains from now

[3] Note this is not the only option. The *call* system call can return an error code indicating the server thread is not ready for invocations. The consensus for synchronous IPC amongst surveyed implementations instead chooses the previous option.

on. In such a case, the *execution context* for a thread and the *scheduling context* are decoupled [20]. An invocation into each protected component requires a separate execution context (including C stack and register contents), but the scheduler treats the thread as a single schedulable entity. Figure 2 depicts thread migration.

2.2.1 Thread Migration in COMPOSITE

COMPOSITE is a component-based operating system focusing on enabling the efficient and predictable implementation of resource management policies, mechanisms, and abstractions as specialized user-level components [14]. Higher-level abstractions such as networking and file-systems are implemented as components, as are less-conventional low-level policies for task and interrupt scheduling [13], mutual exclusion mechanisms, and physical memory management. Components, by default, are spatially isolated from each other in separate protection domains (provided by hardware page-tables).

Components export an interface through which their functionality can be harnessed by other components. As system policies and abstractions are defined in components, invocations between components are frequent and must be both efficient and predictable.

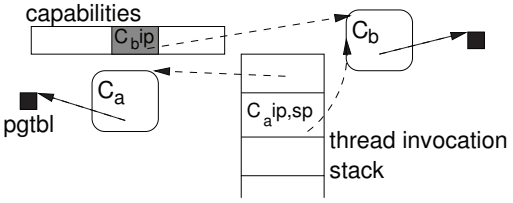


Figure 3. COMPOSITE kernel data-structures involved in an invocation. A syscall specifies a capability (associated with C_a) that yields the component to invoke (C_b). A thread's *invocation stack* saves the invoking component and enough state to return from the invocation (namely, the instruction and stack pointers).

The main kernel data-structures involved in an invocation between component C_a and C_b are depicted in Figure 3. Each component is restricted to make invocations only to components to which it has a capability [9]. Kernel capability structures link components and designate the authority to make invocations from one to the other. A thread executing in C_a that makes an invocation on a capability (via system call), will resume user-level execution in C_b, the component designated by the capability. This invocation occurs within the same thread, thus the same schedulable entity.

In addition to designating which component to execute in, a capability includes the instruction pointer in C_b to begin execution at. To maintain isolation, execution in C_b must be on a different stack from the one used in C_a. This execution stack in C_b is not chosen by the kernel. Instead, it is assumed that when the upcall is made into C_b, the first operation performed is to locate a stack to execute on. This operation we will refer to as *execution stack retrieval*. A simple implementation of this is to have a freelist of stacks in C_b, and to remove and use one upon upcall. Execution stack retrieval must be atomic to maintain freelist integrity as thread preemptions can occur at any time. COMPOSITE supports restartable atomic sequences [13] to provide this atomicity, even on processors that don't support atomic instructions. If the freelist of stacks is empty, then C_b invokes the *stack manager* component that either allocates a stack in C_b, or blocks the requesting thread until one becomes available.

As depicted in Figure 3, the structure representing a thread in COMPOSITE includes an *invocation stack*[4] which traces all invocations that have been made while in the context of that thread. Each entry in the stack includes a reference to the component being invoked, and the instruction and stack pointers to return to in the previous component. When an invoked component returns from an invocation (by invoking a static *return capability*), an item is popped off of the invocation stack, and the appropriate protection domain, stack pointer, and instruction pointer are loaded, returning execution to the invoking component (C_a). This process avoids loops, and doesn't touch user-memory so it shouldn't fault. The kernel invocation path, then, should be predictable.

Invocation arguments are passed in registers – up to 4 words on the x86 COMPOSITE implementation. Additional arguments are passed via shared memory.

IPC Efficiency in COMPOSITE: As in L4, the number of data-structures (thus cache-lines and TLB entries) touched during an invocation is small to minimize cache interference, and improve performance [10]. COMPOSITE's invocation path is implemented in C. It achieves performance on the order of optimized synchronous IPC paths also implemented in C. A component invocation takes less than 0.7 μ-seconds on both a 2.4 Ghz Pentium 4 processor, and a 1 Ghz Pentium M processor[5]. This is comparable to reported performance numbers in the μ-kernel literature [23, 15].

We believe that this demonstrates that thread-migration can be implemented without significant performance overheads compared to other techniques. Given this, the question is what are the other system factors that favor either

[4] Please note that this invocation stack is unrelated to the C stack.

[5] The average invocation overheads on these processors are similar – though they have varying clock speeds – due to significant differences in hardware overheads for user-kernel transitions, page-table switches, and relative CPU/memory speeds.

thread migration, or synchronous IPC. We investigate these in the rest of the paper.

3 CPU Allocation and Accounting

In this section, we investigate how CPU time is allocated amongst and accounted to different threads.

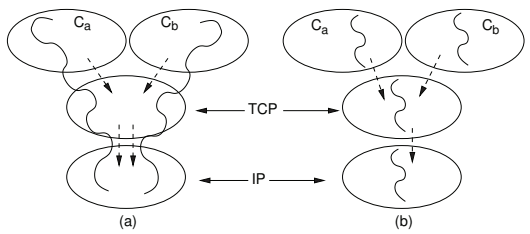


Figure 4. Invocations through components: (a) thread migration, (b) synchronous IPC.

We start with a simple system depicted in Figure 4. Application execution starts in a client component C_a and makes an IPC to C_{TCP} which, in turn, makes an IPC to C_{IP}. This could correspond to an untrusted client sending a packet through the transport and internetworking layers. Additionally, a second client component, C_b, causes the same progression of IPCs. Assume that C_a and C_b do not trust each other, and that they simply use the services provided by lower-level components.

Here we wish to investigate how CPU time is allocated and accounted throughout the system, and how it effects the policies for managing time. We investigate two models: synchronous IPC with a separate thread per component, and migrating threads where threads start in C_a and C_b and execute throughout system components. Figure 4 depicts these two situations.

From a resource allocation and accounting perspective, these two models are very different. To illustrate, assume that the amount of cycles spent processing in C_a is p_a^a. Invocations from this component result in p_{TCP}^a and p_{IP}^a cycles sent processing in C_{TCP} and C_{IP}, respectively. Additionally, the amount of time spent processing for the execution originating in C_b is p_b^b, p_{TCP}^b and p_{IP}^b, correspondingly.

3.1 Synchronous IPC Accounting and Execution

Client execution accountability: In the synchronous IPC model, the processing time spent in each component is charged to the component's thread. Thus the thread in the initial applications will be charged for their execution: p_a^a and p_b^b. However, the execution charged to $\tau_0^{C_{TCP}}$ will be $p_{TCP}^a + p_{TCP}^b$, and $\tau_0^{C_{IP}}$ will be charged for $p_{IP}^a + p_{IP}^b$. If the number of requests originating from C_a is significantly larger than those from C_b (even if the processing time in those components is small), the system scheduler will have little ability to throttle one client, or to even know which client is causing the overhead in the networking stack. The fundamental problem with this model for tracking CPU usage, and scheduling computation, is that it loses information about which client a shared component is doing processing for. Practical approaches to many of the shortcomings of the pure synchronous IPC model are discussed in Section 3.3.

Real-time task models: Aside from the inability of the scheduler to properly track client execution throughout the system, synchronous IPC does not naturally accommodate traditional real-time task execution models. It is common to assume a task has a given worst-case execution time, C, and executes periodically, with a period of T. C includes all execution time, including that which occurs in server components. The scheduler, will not see the thread using C execution time as the accounting for this execution is distributed throughout invoked threads in the system. This would make it difficult if not impossible to implement accurate aperiodic servers [22] that make invocations to other components, as budget consumption would be spread across multiple threads. An additional problem arises as the priority of a thread is often associated with its C (e.g. in rate-monotonic scheduling). It is not obvious how to assign priorities to threads throughout the system in the presence of pervasive thread dependencies. This is especially true in an open real-time system where an unknown number of nonreal-time or soft real-time tasks execute along side hard real-time tasks and they can all rely on shared servers. This problem only becomes more pronounced as the depth of the component hierarchy increases.

An application can avoid these problems by making no invocations to server threads. Unfortunately, this limits the functionality available to that application, and prevents the decomposition of the system into fine-grained components.

Priority Inversion: A server thread might have a low priority compared to a high-priority client. In such a case, a medium priority thread can cause unbounded priority inversion. To avoid these situations, great care must be taken in assigning thread priorities throughout the system. For example, [5] proposes a static structuring such that server threads always have the same or higher priority than their clients. Unfortunately, it is not clear if it generalizes in open systems. Additionally, as it requires that servers run at a higher priority, it can lead to larger scheduling interference of high priority server threads (that service predominantly low priority threads) with medium priority threads elsewhere in the system.

One might be tempted to observe that many of these problems come from having components that are relied upon and invoked by multiple other components, possibly with widely varying temporal requirements. Can't we

simply arrange the system such that there are no components that are shared between different subsystems? Unfortunately, it is difficult to not share components that drive shared peripherals (e.g. keyboards, networking cards), that share the system's physical memory between subsystems, or that schedule system's threads (assuming component-based scheduling [13]). Such sharing is unavoidable.

3.2 Migrating Threads Accounting and Execution

The migrating thread model makes it explicit which client a server component is processing for, e.g. computation in the networking stack is performed in the scheduling context of the client thread. The scheduler explicitly sees all execution performed on behalf of a specific client, and can schedule it accordingly. Thus, the execution time accounted to the thread created in C_a is $p^a_a + p^a_{TCP} + p^a_{IP}$, and likewise for the thread created in C_b. If τ_a makes a disproportionately large amount of invocations into the networking stack, it is charged directly for the processing time of those invocations (in contrast to the synchronous IPC case).

Priority Inversion: A significant complication with the migrating thread model concerns shared resources *within* a server component. If a low-priority thread takes a shared resource requiring mutual exclusivity (e.g. it is protected by a lock) priority inversion can occur if it is preempted by a high-priority thread that attempts to access the shared server resource. The solution to this is to use a resource sharing protocol that bounds the priority inversion [18]. In COMPOSITE, locking policies including those that avoid unbounded priority inversion are implemented as components.

We claim that the resource management and accounting properties of this model more closely match the intended structure of a system composed of many components. There is some precedent for this position: When a user-level process makes a system call, the execution time spent in the kernel is typically accounted to and scheduled with the credentials of the user-level thread. That is to say, that threads migrate from user- to kernel-level (though, of course, their execution contexts change).

3.3 Synchronous IPC Implementations: Accounting and Execution

Actual implementations of synchronous IPC deviate from the pure model. In this section, we discuss the relevant differences.

In synchronous IPC, the kernel switches between threads on each IPC. It is thus natural to perform scheduling on every IPC. However, the overhead of scheduling decreases IPC performance significantly. An optimization is to use *lazy scheduling* to avoid scheduling until the scheduler is explicitly invoked (e.g. via a timer-interrupt), and to do *direct process switch* whereby the system switches directly to the server thread upon IPC [10, 17] (assuming the server thread was blocked waiting for an IPC) [6]. The combination of these techniques removes scheduling related overheads from the IPC path.

Unfortunately, The thread that is charged for execution at any point in time is not predictable. Before the execution of the scheduler, the invoking thread is charged, emulating migrating threads. However, after the scheduler is executed, the threads are scheduled separately. This unpredictability is harmful to real-time systems [16], and researchers have tested if the optimization is indeed necessary for efficiency [6]. The answer appears dependent on the frequency of IPCs. It should be noted that in such a case we are choosing between two undesirable cases: (1) unpredictable resource accounting and scheduling (via direct process switching and lazy scheduling), and (2) the problems associated with the pure synchronous IPC between threads (Section 3.1) including the associated overhead.

Side-stepping these problems, Credo [20] decouples the execution context and scheduling context of threads. Synchronous IPC between threads transfers the scheduling context to the receiving thread. This model can require walking a path in the *dependency graph* of thread synchronizations to maintain proper scheduling context assignments. Credo essentially moves the synchronous IPC regime towards a migrating thread model. Unfortunately, it does so at the cost of complexity, and it increases the worst-case execution time of invocations by requiring the walking of the dependency graph to determine current scheduling context. If the depth of this tree is not predictable, then IPC operations themselves will, in turn, not be predictable.

Discussion: Motivated by efficiency or better accounting, practical synchronous IPC implementations have moved towards the accounting and execution style of a migrating thread model. However, they do so a the cost of complexity and possible unpredictability. Systems requiring predictable IPC in which dependency graph depths cannot be statically known, would benefit from starting with a migrating thread model.

4 Communication End-Point Contention

IPC in μ-kernels and component-based OSes is directed at specific communication end-points. The end-point in synchronous IPC systems is the server thread. This thread is addressed directly from the client (i.e. by thread id), or indirectly via a capabilities [9]; the end-point is the same. For

[6] It should be noted that K42 provides synchronous IPC with direct process switch between *dispatchers* that are similar in many ways to system threads.

thread migration, the target of an invocation is the component, or protection domain, being invoked. This component can be addressed either by id, or indirectly by capability. The end-point of an invocation is important as it effects system behavior when there is contention (multiple concurrent invocations) to that end-point.

4.1 Synchronous IPC End-Point Contention

Unpredictable IPC overheads due to end-point contention: If multiple threads attempt to conduct synchronous IPC with an active server thread, they are placed in its wait-queue. When the server thread replies, the system executes the thread being replied to, or one of the threads on the wait-queue, depending on which of all of the threads has the highest priority. The execution cost of finding the next thread to execute, then, is linear in the size of the wait-queue. Thus to enable predictable IPC, the number of threads concurrently *call*ing a specific server thread must be bounded. The assumption is often that the duration of an IPC is short, thus the server thread will be preempted with only a small probability. Thus, the wait-queues should rarely grow to significant length. In general component-based systems in which even applications are decomposed into separate components, the probability of preemption in an invoked component is high, thus the consideration of wait-queue length is important[7]. Importantly, worst-case IPC costs must be considered in hard real-time systems.

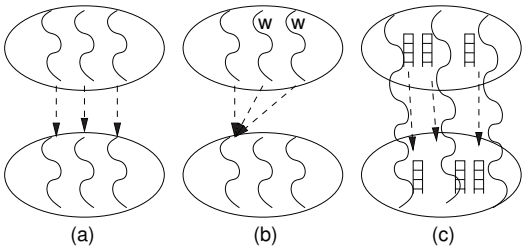


Figure 5. Invocations of and contention on various end-points. (a) All client threads invoke separate server threads. (b) Client threads invoke the same server thread, adding two to the server thread's wait-queue. (c) Thread migration: execution contexts aren't the target of invocation, thus cases similar to (b) are impossible.

Assume N threads concurrently attempt to invoke M threads in a server component. If $N = 1, M \geq 1$, then it is clear the IPC will continue without complication as the wait-queue is empty. If $M \geq N \geq 1$, and each of the N synchronizes with a separate server thread, the situation is comparable (Figure 5(a)). However, it is possible that all N invoking threads will attempt to synchronize with a single thread in the server, thus the wait-queue will be $N - 1$ long [8]. $M - 1$ server threads will remain waiting for IPC, and IPC overheads will correspondingly increase. This situation is depicted in Figure 5(b). If $N > M$, then some server thread's wait-queues will be unavoidably non-empty.

It follows that IPC predictability is dependent on if the following factors can be predicted: (1) the relative number of clients and server threads, and (2) the distribution of client invocations across server threads.

Limiting wait-queue length: Perhaps the most straightforward way to predict the maximum size of server wait-queues is to ensure that for each client thread, there is a corresponding server thread. Care is taken to only invoke a client's corresponding server thread. Though appealing in simplicity, this solution doesn't generalize for two reasons. First, the maximum number of threads in a protection domain is often bounded. Thus two components with the maximum number of threads each, would have at least twice the number of threads than are available in the server. Second, threads take up resources (e.g. memory). In the worst case such a strategy would require $T \times C$ threads, for T application threads and C components.

In a more realistic scenario, server threads are partitioned amongst different classes of client threads (with different priorities, or timing constraints). Fundamentally, client threads don't know the status of specific server threads (i.e. if specific server threads are busy or waiting for IPC). Yet on each invocation, they must answer "which server thread should I call?" Thus it is difficult, in the general case, for them to avoid invoking the same server thread.

Discussion: Predictable systems can be created using the two suggested modifications to synchronous IPC. However, in general systems with possibly malicious clients, and deep component hierarchies, it is not clear what the price of such techniques is (e.g. in memory consumption for thread context, or programmer complexity). Generally, the root problem is that the clients are forced to choose the specific execution context to process on in the server, but they don't have all information required to make that decision. That decision is best made by the server that knows its own state. In Section 4.3, we discuss possible enhancements to make this possible.

[7] Some synchronous IPC implementations disregard priority, and either switch immediately to the thread being replied to, or to the head of the wait-queue. This alleviates the problem of linear execution time in the size of the wait-queue. However, as it ignores thread priorities, it is unpredictable none-the-less.

[8] We are assuming a very specific interleaving of client threads where invocations are made before a server thread completes processing of an IPC request. This is the worst-case, and must be considered in real-time systems.

4.2 Thread Migration and End-Point Contention

For thread migration, the communication end-point being invoked is the server component[9]. As discussed in Section 2.2.1, when a component is upcalled into as the result of an invocation, the first operation it performs it to retrieve an execution stack from its local freelist. Assume N threads invoke the functions of a component in which M execution stacks (contexts) exist.

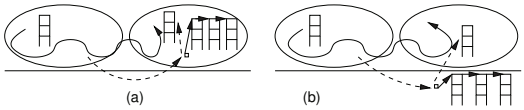


Figure 6. Retrieving execution contexts with thread migration. (a) Stacks are maintained on a freelist in the invoked component, or (b) in the kernel.

If $N \leq M$, all invocations will immediately find an execution stack to execute on. As the invocation end-point is the component, rather than specific execution contexts, so the server prevents contention on its stacks. As the execution contexts are not the target of IPC, the component has the opportunity to multiplex execution contexts as it deems appropriate. Clients will never block waiting for an execution context. Figure 6(a) depicts this operation.

If $N > M$, then contention for execution contexts is unavoidable, so the freelist of stacks will be empty for some invocations. In such cases, the thread invokes a component specializing in stack management. The stack manager allocates a new stack for immediate use, or calls the scheduler to block the thread until one becomes available. In the latter case, the stack manager implements priority inheritance to avoid unbounded priority inversion. Differentiated service between different clients or threads can be provided both at the time of execution stack retrieval, and in the stack manager by maintaining different lists of stacks for each level of service, and class of specific threads (i.e. hard real-time vs. best-effort execution contexts).

In COMPOSITE we choose to implement all policies for obtaining execution contexts at user-level in components. This enables (1) the definition of specialized policies as user-level components, and (2) the simplification of the kernel invocation path enabling its predictable execution and low number of data-structure accesses. It is possible to maintain the freelist of stacks in the kernel and assign a thread to a specific context for the duration of a component invocation. This is the approach taken by [8] which slightly complicates both kernel data-structures and the invocation path. Additionally, it places the policy for managing the stack freelist in the kernel (thus precluding the differentiated service policy described above). The benefit of this approach is that, in the case there is no available stack, it avoided the invocation into the server component. Figure 6(b) depicts this scenario.

Discussion: By changing the target of invocations from individual threads in the server to the server itself, thread migration enables the server to manage and allocate its own execution contexts. This avoids multiple client threads waiting on a single server thread while other server threads are available which will increase IPC overheads.

4.3 Synchronous IPC End-Point Enhancements

Here we propose methods for modifying synchronous IPC implementations to include many of the benefits of the migrating thread model by changing the server communication end-point.

Locating execution contexts: One benefit of the migrating thread model is that the IPC end-point is not a specific execution context, thus the system – or the invoked component – has the opportunity to choose the appropriate context itself according to specialized policies. The system (and application) designer need not carefully plan which specific client and server threads communicate with each other. We believe that slight modifications to synchronous IPC implementations would enable the same capability.

Some modern μ-kernel systems [9] use capabilities to indirectly address the thread endpoint for IPC. Given this level of indirection, it would be natural for the capability to reference not a single thread, but a collection of server threads. Figure 6(b) depicts a similar scheme. Whenever an invocation is made with the capability, a thread is dequeued and execution in the server is made on that thread. As capabilities currently hold a pointer to the thread to IPC to, the overhead of this approach should be minimal. Capabilities can include a wait-queue of threads waiting to complete IPC with one of the server threads. Alternatively, when no server thread is available to service a *call*, a exception IPC (similar to page-fault IPC) can be delivered to a corresponding execution context manager. When paired with the Credo enhancements to migrate scheduling context upon invocation, synchronous IPC becomes quite similar to thread migration indeed. The desired behavior seems better captured by thread migration.

Predictable IPC execution time: In systems where it is difficult to predict the maximum number of threads on a wait-queue for a server thread (thus the worst-case cost of an IPC), it is possible for intelligent data-structures to provide a constant-time lookup of the highest priority thread waiting for IPC. This removes the linear increase to the cost of IPC for waiting threads (though not the cost commen-

[9] More specifically, in COMPOSITE the end-point is a function within the API of the component denoted by a capability is the target.

surate with the depth of the dependency graph in Credo). The O(1) Linux scheduler (present in Linux versions 2.6 to 2.6.23) includes a data-structure enabling constant time lookup of the highest-priority thread. Though this approach will technically make IPC time predictable across all wait-queue lengths, it could impose a large cost in terms of memory usage and constant execution overheads. We leave this as an area of future study.

5 System Configurability

We discuss the ways that COMPOSITE provides the user-level definition of novel policies that rely upon the semantics of thread migration. Specifically, we discuss user-level, component-based scheduling, and Mutable Protection Domains (MPD) that enable the alteration of the protection domain configuration at run-time.

5.1 Component-Based Scheduling

In designing μ-kernels and component-based operating systems, a common goal is to include in the kernel only those concepts required to implement the system's required functionality at user-level [11]. The inclination is to remove mechanisms and policy from the (fixed) kernel and define them instead in replaceable and independently failable user-level components. Part of the motivation for this is so that the system can be configured to the largest possible breadth of application and system requirements. In real-time and embedded systems, the policies that dictate temporal behavior are amongst the most sensitive to meeting such requirements. In COMPOSITE, then, we have focuses on enabling the user-level, component-based definition of system scheduling policies [13].

To enable efficient user-level scheduling in COMPOSITE, the invocation path should not require scheduler invocation. This goal has two implications: First, the invocation path should not rely on scheduling parameters associated with threads, such as priority, as these are defined in the user-level scheduler. Second, the invocation path should not result in multiple threads becoming active, as this would imply an invocation of the user-level scheduler.

The migrating thread model satisfies both of these constraints. As no thread switches occur during the invocation path, the scheduling parameters associated with threads are not required. The only thread active during an invocation is the original scheduling context. To block or wakeup threads, invocations must be made to the scheduler component.

Pure synchronous IPC does not satisfy either of these goals. As thread switches occur on each IPC, the next thread to execute must be located, and to do so involves access to thread scheduling parameters, and dispatching between threads. This practically requires the scheduler to be kernel-resident, and for the IPC mechanism to hard-code a single scheduling policy. Additionally, some IPC operations result in the activation of multiple threads. For example, when executing a *reply_wait*, the IPC path can result in both the client and server threads, being active if there are threads on the wait-queue for the server thread. Direct process switching avoids these issues at the cost of predictable thread execution accounting.

There are some indications, beyond the COMPOSITE implementation, that user-level scheduling of all system threads is best done with the migrating thread model. For example, in [21], L4 is modified to allow specific threads to control scheduling. Doing so involves migrating scheduling context with IPCs as in Credo. Additionally, due to complications created by end-point contention, the author suggests that a solution is to "construct a μ-kernel solely based on procedure call semantics".

5.2 Mutable Protection Domains

The resource accountability and execution semantics of component invocations are identical for invocations between protection domains, and between components in the same protection domain. This, along with novel mechanisms for predictably and dynamically altering protection domain structures, enables Mutable Protection Domains (MPD) [12]. MPDs recognizes that in a fine-grained component-based system, even optimized invocation paths can have significant overheads[10]. The system is able to monitor the frequency of invocations between each component. We observe that the distribution of such invocation counts is heavily tailed, and if the overhead of invocations between a small number of components is removed, the system can attain both high reliability (retaining most protection domain boundaries), while concurrently achieving significant performance improvements (up to 40%). As the distribution of invocations between components change, the system can erect and remove protection boundaries as appropriate. The goal is to maintain high reliability (to detect and isolate faults when they occur), and high performance. When a protection boundary is required for security, it should never be removed.

To retain a consistent model of thread execution accounting and scheduling in a system using synchronous IPC, thread switches would be necessary even when the components share a protection domain. The overhead of scheduling and switching between threads is higher than that of direct invocation of the destination function. For example, the cost of intra-protection domain invocations in COMPOSITE

[10] In COMPOSITE, we implement a simple web-server [14] consisting of about 25 components. Each HTTP request causes between 50 and 70 invocations depending on if it is for static or dynamic content.

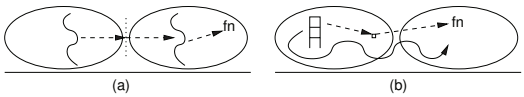


Figure 7. Inter-address space invocations of the function `fn`, using (a) synchronous IPC that required thread dispatch, and (b) migrating threads with indirect invocations through a function pointer [14].

is on the order of a C++ virtual function call, significantly faster than thread dispatch. Figure 7 depicts these two forms of invocation. Only a single execution context is required when using thread migration.

6 Thread Migration Limitations

There are a number of limitations and peculiarities both with the thread migration model, and with the COMPOSITE implementation. We discuss these in turn.

Execution Context Unavailability: When an invocation is made to a component, an execution context for that invocation must be found. This can be done in the kernel [8], or within the component itself (as in COMPOSITE). If there are no available contexts, the system must resolve this contention. Both thread migration *and* synchronous IPC must deal with this case where there are more pending invocations than there are server execution contexts. We believe this case is best dealt with by allowing the customized definition of the policies for dealing with such cases. In COMPOSITE, these policies are defined as components and they vary from having separate execution stack freelists for different client service classes (i.e. to guarantee that hard real-time tasks will always find a stack), to implementing priority inheritance. Additionally, we are currently investigating methods to balance responsiveness with execution context memory usage when allocating execution contexts to components.

Fault Recovery: The fault recovery model for server-based systems using synchronous IPC is well-known and simple to understand. When a fault occurs within a server, all threads in active IPC with that thread or server can be directly notified of that failure and act accordingly. The faulting thread and the server can independently be restarted. With thread migration, the thread that causes a fault in one component should not be destroyed as its execution context is spread across multiple components. Thus when a fault occurs in a specific component, one solution is to cause the thread to return to the invoking component, either with an error code, or an exception [3]. This model is less familiar to developers accustomed to a process-style structuring of the system.

6.1 COMPOSITE Implementation Limitations

COMPOSITE is a prototype and should not be seen as being as feature-rich as either monolithic systems such as Linux, or even mature μ-kernels such as L4. A number of design decisions simplify the implementation of thread migration in COMPOSITE.

First, the COMPOSITE kernel is non-preemptive. The IPC path assumes that no interrupts will preempt it, thus that no synchronization around kernel data-structures is required in the single-processor case. Additionally, as the invocation path does not touch user-memory, we assume that faults cannot occur. The non-preemptive assumption is justified by the general lack of expensive operations in the kernel. For example, we avoid supporting general hierarchical address space operations such as *map*, *grant*, and *unmap* [11] operations in the kernel, as complex mapping hierarchies can cause *unmap* to become expensive. Instead, we provide a simple operation to directly map a physical frame into a specific virtual location in a component. A privileged user-level component uses this simple facility to itself implement the higher-level operations. Though we believe the non-preemptive kernel implementation is a sound design decision, we cannot predict if systems that do not make such an assumption might have difficulty implementing efficient invocations using thread migration.

An additional limitation of the current COMPOSITE implementation is the fact that it only supports uniprocessors. We believe partitioning the state of the system (e.g. threads) between processors will enable efficient and predictable (and lock-free) invocations when we move COMPOSITE to multiprocessors.

7 Related Work

Thread migration is not a new method for inter-protection domain invocation. LRPC [1] describes now RPC within a single machine can be optimized by using thread migration. Ford [7] altered the invocation path in Mach to use thread migration for a significant performance improvement. Pebble [8] optimizes invocation latency by custom-compiling specialized invocation code and by using thread migration. We argue that thread migration is a predictable foundation upon which to implement finely decomposed, configurable, and reliable systems. We do not know of other work that has compared the predictability of thread migration to synchronous IPC.

8 Conclusions

In this paper, we argue the case for using a thread migration approach for predictable inter-protection domain communication in configurable and reliable systems. In doing

so, we introduce the COMPOSITE design for component invocation that uses thread migration. We make this argument in terms of three factors: (1) the desire to have a consistent processor management and accounting scheme for CPU utilization across invocations that maps well to systems in which specialized services are provided by some components to others, (2) the communication end-point abstractions provided by the kernel that have a significant effect on the bounds for IPC latency, and (3) the effect that the IPC mechanism can have on the ability of the system to provide configurable system policies (e.g. scheduling, MPD).

We argue that thread migration provides a predictable invocation foundation, and we contrast that with synchronous IPC in the general case. We argue not that previous IPC mechanisms should be abandoned, but that bringing their semantics closer to that of thread migration is beneficial for overall system predictability.

The COMPOSITE source code is available upon request.

References

[1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[2] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 223–233, New York, NY, USA, 1992. ACM.

[3] F. M. David, J. C. Carlyle, E. Chan, D. Raila, and R. H. Campbell. Exception handling in the choices operating system. In *Advanced Topics in Exception Handling Techniques in Springer Lecture Notes in Computer Science*, pages 42–61, 2006.

[4] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: a flexible, optimizing idl compiler. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 44–56, New York, NY, USA, 1997. ACM Press.

[5] K. Elphinstone. Resources and priorities. In *Proceedings of the 2nd Workshop on Microkernels and Microkernel-Based Systems*, October 2001.

[6] K. Elphinstone, D. Greenaway, and S. Ruocco. Lazy scheduling and direct process switch – merit or myths? In *Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2007.

[7] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of theWinter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, 1994.

[8] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The pebble component-based operating system. In *Proceedings of Usenix Annual Technical Conference*, pages 267–282, June 2002.

[9] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in l4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, New York, NY, USA, 2009. ACM.

[10] J. Liedtke. Improving ipc by kernel design. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 175–188, New York, NY, USA, 1993. ACM Press.

[11] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[12] G. Parmer and R. West. Mutable protection domains: Towards a component-based system for dependable and predictable computing. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 365–378, Washington, DC, USA, 2007. IEEE Computer Society.

[13] G. Parmer and R. West. Predictable interrupt management and scheduling in the Composite component-based system. In *RTSS '08: Proceedings of the 29th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, 2008.

[14] G. A. Parmer. *Composite: A Component-Based Operating System for Predictable and Dependable Computing*. PhD thesis, Boston University, Boston, MA, USA, Aug 2009.

[15] S. Reichelt, J. Stoess, and F. Bellosa. A microkernel api for fine-grained decomposition. In *5th ACM SIGOPS Workshop on Programming Languages and Operating Systems (PLOS 2009)*, Big Sky, Montana, oct 2009.

[16] S. Ruocco. A real-time programmer's tour of general-purpose l4 microkernels. In *EURASIP Journal on Embedded Systems*, 2008.

[17] Scheduling in k42, whitepaper: http://www.research.ibm.com/k42/white-papers/scheduling.pdf.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

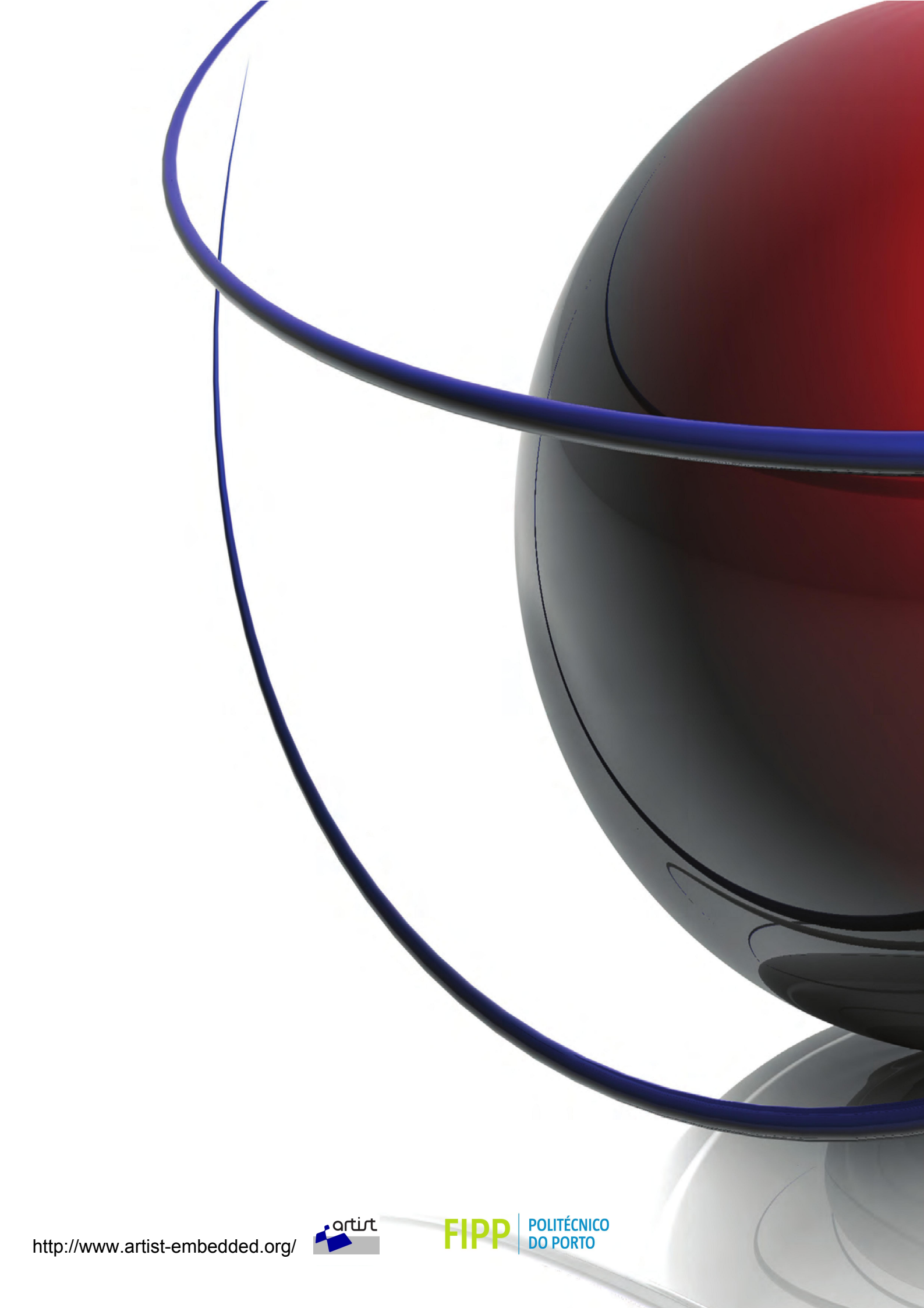
[19] J. S. Shapiro. Vulnerabilities in synchronous ipc designs. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 251, Washington, DC, USA, 2003. IEEE Computer Society.

[20] U. Steinberg, J. Wolter, and H. Hartig. Fast component interaction for real-time systems. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 89–97, Washington, DC, USA, 2005. IEEE Computer Society.

[21] J. Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *SIGOPS Oper. Syst. Rev.*, 41(4):59–68, 2007.

[22] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[23] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.



http://www.artist-embedded.org/

artist

FIPP POLITÉCNICO DO PORTO