Integrated Task and Interrupt Management for Real-Time Systems

LUIS E. LEYVA-DEL-FOYO, Universidad Autónoma Metropolitana—Unidad Cuajimalpa
PEDRO MEJIA-ALVAREZ, CINVESTAV-IPN
DIONISIO DE NIZ, Carnegie Mellon University

Real-time scheduling algorithms like RMA or EDF and their corresponding schedulability test have proven to be powerful tools for developing predictable real-time systems. However, the traditional interrupt management model presents multiple inconsistencies that break the assumptions of many of the real-time scheduling tests, diminishing its utility. In this article, we analyze these inconsistencies and present a model that resolves them by integrating interrupts and tasks in a single scheduling model. We then use the RMA theory to calculate the cost of the model and analyze the circumstances under which it can provide the most value. This model was implemented in a kernel module. The portability of the design of our module is discussed in terms of its independence from both the hardware and the kernel. We also discuss the implementation issues of the model over conventional PC hardware, along with its cost and novel optimizations for reducing the overhead. Finally, we present our experimental evaluation to show evidence of its temporal determinism and overhead.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling, threads*; D.4.4 [**Operating Systems**]: Communications Management—*Input/output*; D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Predictability, interrupt scheduling, programmable interrupt controller

ACM Reference Format:
Leyva-del-Foyo, L. E., Mejia-Alvarez, P., and de Niz, D. 2012. Integrated task and interrupt management for real-time systems. ACM Trans. Embed. Comput. Syst. 11, 2, Article 32 (July 2012), 31 pages.
DOI = 10.1145/2220336.2220344 http://doi.acm.org/10.1145/2220336.2220344

1. INTRODUCTION

Most real-time systems interact with the physical world. This interaction needs to be tightly synchronized in order to obtain the effect the system is trying to get in the physical world, for example, inflating the airbag of a car in a crash. When this interaction involves asynchronous events, interrupts are commonly used. Such interrupts need to have predictable timing behavior and the capacity to enable the rest of the system to be equally predictable.

This research was supported in part by a grant from NSF, CONACYT, and PROMEP from project NSF-CONACYT 42449 and project UAM-PTC-141.
Authors' addresses: L. E. Leyva-del-Foyo, Departamento de Tecnologías de la Información, Universidad Autónoma Metropolitana—Unidad Cuajimalpa, Avenida Constituyentes 1054, México, D.F., C.P. 11950; email: lleyva@correo.cua.uam.mx; P. Mejia-Alvarez, Departamento de Computación, CINVESTAV-IPN, México D.F., Av. IPN 2508 México, C.P. 07360 D.F; email: pmalvarez@delta.cs.cinvestav.mx; D. de Niz, Software Engineering Institute, 4500 Fifth Avenue, Pittsburgh, PA 15213.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1539-9087/2012/07-ART32 $15.00
DOI 10.1145/2220336.2220344 http://doi.acm.org/10.1145/2220336.2220344

Two forms of asynchronous activities are found in real-time systems: tasks and interrupt service routines (ISRs). Each of these activities has its own independent scheduling and synchronization policies and mechanisms. In particular, the interrupts mechanism schedules the execution of ISRs in response to the occurrence of external asynchronous events, called interrupt requests (IRQs). On the other hand, the operating system (OS) schedules the execution of tasks in response to their arrival to a ready queue.

In order to obtain a high-efficiency and low-latency interrupt response time, general-purpose (and also real-time) operating systems provide a set of mechanisms to handle interrupts totally independent of those used for task management. While this scheme is adequate for high-throughput systems, for example, Web and database servers, in critical real-time systems, the differences in the scheduling and synchronization between ISRs and tasks create problems of mutual interference that can jeopardize the predictability of the system.

Tasks are abstractions of the concurrency model supported by the kernel, and the responsibility of their management lies completely on the kernel itself, providing services that create, delete, communicate, and synchronize tasks. On the other hand, interrupts are abstractions of the computer's hardware and the responsibility for their management lies with the hardware logic of a specialized circuit. This hardware is responsible for providing services to assign IRQs to ISRs, perform context switching between tasks and ISRs, and enable and disable IRQs.

The interrupt-handling hardware is responsible for ISR scheduling, according to their hardware priorities. Tasks, on the other hand, are scheduled by the kernel according to their software priorities. In the traditional interrupt mechanism, hardware priorities have precedence over software priorities, because this mechanism was originally designed for general-purpose systems in which the only activities with strict timing requirements are interrupts. This arrangement provides low latency to interrupts, avoiding data losses while other tasks are executing. However, in real-time systems where all activities (tasks and interrupts) may have strict timing requirements, this scheme introduces unpredictability that may jeopardize the temporal guarantees demanded by these tasks.

The synchronization among tasks is done by mechanisms provided by the operating system (e.g., semaphores, mutexes, messages, mailboxes, etc.). In contrast, synchronization among ISRs is reduced to the mutual exclusion achieved with the help of their own hardware priorities.

In most common designs, a priority is assigned to each IRQ, allowing the arrival of higher-priority requests during the execution of an ISR. In this scheme, known as nested interrupts, each ISR is executed as a critical section with respect to itself, to lower priority ISRs, and with respect to the tasks. Although the ISRs are automatic critical sections with respect to the tasks, the opposite is not true. The mechanisms used to guarantee exclusive access to critical sections among tasks do not guarantee exclusive access of the tasks against ISRs. The mutual exclusion between ISRs and tasks is only achieved by disabling interrupts.

In order not to affect the system's response time to urgent interrupts, interrupts are disabled by priorities. That is, a disable threshold (a.k.a. IRQ level) is set to delay the occurrence of an interrupt of a priority lower or equal to the threshold, until that threshold is lowered again. In general-purpose systems, the synchronization by interrupt disabling is adequate, because no task ever needs to preempt an ISR. However, in real-time systems, it is possible for a task to have a higher priority than an ISR. Hence, a selective protection against ISR preemptions is needed to allow some ISRs to preempt a task while disallowing others.

In this article, we propose a strategy that integrates both types of asynchronous activities, resolving the previously discussed issues. The contributions are the following.

— An integrated strategy for the management of interrupts and tasks for real-time systems.
— The evaluation of the integrated scheme using utilization and response-time schedulability analysis. The purpose of this evaluation is to help the designers understand the conditions that make our scheme more valuable.
— The design of a portable low-level subsystem for interrupt management for real-time systems based on the integrated model.
— The implementation of the integrated model over a conventional PC hardware using virtual interrupt masking with an analysis that shows the applicability of this technique to real-time systems.

The rest of the work is organized as follows. Section 2 discusses the mutual interference between tasks and interrupts management in the traditional interrupt handling strategy. In Section 3, our integrated interrupt handling strategy is introduced. In Section 4, a schedulability analysis is conducted to compare both strategies. The design of a portable kernel subsystem based in this integrated model is presented in Section 5. The rationale for an implementation over conventional PC hardware, using interrupt masking is presented in Section 6, as well as the analysis of its overhead. In Section 7, we present latency and efficiency improvement using virtual interrupt masking. A detailed design with the pseudocode of the implementation is presented in Section 8. Section 9 presents our experiments to demonstrate the deterministic behavior of our integrated model. In Section 10, the related work is discussed and compared with our model. Finally, Section 11 presents our conclusions.

2. MUTUAL INTERFERENCE IN THE TRADITIONAL MODEL IN REAL-TIME SYSTEMS

Since the traditional model of interrupt handling is strongly supported by hardware, it yields a fast response to external events and a low overhead. Consequently, it has been the method of choice in most real-time operating systems. However, its use causes interference from interrupts to tasks and vice versa. This is discussed next.

2.1 Priority Interference

The assumption that the timing execution requirements of an ISR have higher importance than those of a task does not hold in real-time systems. In fact, the response-time requirement of a real-time task may be even shorter than that of some ISRs. In such a case, it may be necessary to assign a higher priority for task than to some (or all) ISRs. For example, the tasks with high priorities may be under the *disturbance* of hardware events necessary only for low-priority tasks. On the other hand, these low-priority tasks associated with interrupts may not be able to execute due to temporal overloads (e.g., due to frequent hardware events), even though their associated ISRs are being executed. This arrangement affects the capacity to meet the real-time requirements of the system, causing a potentially large priority inversion and decreasing its utilization bound.

2.2 Interrupt Latency Interference

Perhaps the most significant argument against the traditional model can be found in its main objective, that of reducing interrupt latency to the minimum possible. In order to reduce this latency, the kernel disables interrupts only for brief periods of time. Nevertheless, this approach cannot prevent the applications from disabling interrupts, because this is the only way to synchronize tasks and ISRs. As a result, the system's response time to the interrupts cannot be smaller than the maximum time in which the interrupts are disabled anywhere in the system. Since the application (or a device driver) is capable of disabling the interrupts for more time than the kernel, the worst case interrupt latency would be the sum of the latency introduced by the CPU plus the worst case time on which the interrupts are disabled by the application. Thus, even though the kernel can establish a lower bound in the interrupt latency, it cannot guarantee its worst case latency. Evidence of this fact is discussed in Carlsson et al. [2002].

2.3 Mutual Exclusion Interference

In the traditional interrupt model, mutual exclusion between tasks and interrupts is achieved by disabling interrupts. In order not to affect non-related interrupt sources, many systems disable interrupts by levels. In this scheme, while a low-priority task raises the interruption level to a medium level, in order to enter a critical section that it shares with an ISR of medium level, an interrupt of high level could be occurring to activate a high-priority task, preempting the low-priority task. This context switch activates the IRQ level previously saved for the high-priority task, potentially causing a decrease on the IRQ level. This change reenables the occurrence of medium-level IRQs, effectively destroying the *interrupt lock* of the low-priority task. In order to avoid this situation, the kernel could maintain the state of the interrupts without changes when executing a context switching. However, this approach affects the predictability of the system, because the tasks would be executed with several states of interrupts, depending on which task has been preempted. The alternative is to force the tasks to always set the IRQ level to the highest possible for disabling all preemptions. Nevertheless, this alternative increases the context switch (or preemption) latency, creating an unintended priority inversion.

2.4 Sequencing Interference

In order to minimize the unpredictability caused by interrupts, the traditional interrupt model splits the system response to interrupts into at least two parts: the ISR and a task in the OS. In this approach, an ISR will make at least one call to the kernel to indicate the occurrence of some event. This call usually enables the execution of a task of higher priority than the current task, which is in charge of responding to the external event. If a context switch is executed before the ISR completes, the rest of the ISR cannot be resumed until the interrupted task is executed, leaving the system in an unstable state. Consequently, if these OS services are invoked within an ISR, the kernel must postpone any context switch until the end of the ISR. All solutions proposed for solving this problem introduce an excessive priority inversion effect due to the context switching or exhibit a temporal behavior that is very difficult to model and to predict [Tindell 1999].

2.5 Synchronization Constructs Interference

The difference in the synchronization needs of the two asynchronous activities generates a large number of synchronization situations among tasks and interrupts that cannot be independently analyzed. This increase in the complexity of the solution increases the likelihood of design errors negatively affecting the reliability of the system.

3. INTEGRATED MECHANISM FOR TASKS AND INTERRUPTS HANDLING

In this section, we present a solution to the mutual interference problems previously discussed. Our approach includes the following features.

(1) Integration of tasks and ISRs in a single priority space.
(2) Development of a unified mechanism of scheduling and synchronization.

The integrated mechanism includes a unified and flexible space of dynamic priorities for all the activities in the system. With this model, we have *software activated tasks* (SAT) and *hardware activated tasks* (HAT). SATs are the traditional software tasks, while HATs replace the traditional ISRs to handle interrupts. Under this scheme, both task types are handled with the same mechanisms, allowing the assignment of priorities to all the activities of the real-time system in correspondence only with their timing requirements. With this approach, the following advantages are obtained.

— SATs and HATs share the same space of priorities and at any time may have any priority in the system.
— Priority interference associated with the independent priority space is avoided (Section 2.1).
— The implementation of an enter/leave protocol for disabling ISRs in the kernel is avoided, preventing mutual exclusion interference (Section 2.3).
— The error of the *broken interrupt lock* (resulting from the task switching) is eliminated (Section 2.3).
— Interrupt overloads can be handled using some scheduling techniques, such as the sporadic server.

The integration of the synchronization mechanism is obtained by handling all IRQs with a universal low-level interrupt handler (LLIH) at the lowest level. This LLIH synchronizes the interrupt disabling with the priority of the running task and converts all interrupts into synchronization events using the abstractions of communication and synchronization among tasks. In this model, the HATs remain blocked until an IRQ occurs, for example, by executing *wait*() on a semaphore or a condition variable related to the IRQ (for schemes based on communication by shared memory), or by executing *receive*() to accept messages (for message-passing schemes). When an IRQ occurs, the LLIH only unblocks the task. Key to the integrated model is the fact that the LLIH is only activated (and consume CPU) if the priority of the corresponding HAT is high enough to preempt the currently running task.

This approach provides an abstraction that assigns the low-level details of the interrupt treatment to the kernel (instead of the hardware) and eliminates the differences between HATs and tasks. The real service of the interrupt lies within the HAT, making it unnecessary for the kernel to handle interrupts any differently from tasks.

The existence of only one type of asynchronous activity and a uniform synchronization and communication mechanism between tasks and ISRs provide a solution to the mutual interference problems discussed in Section 2 with the following advantages.

— The scheduling of IRQs is not performed by the PIC but by the task scheduler of the kernel (through the unified priority scheme).
— Interrupt disabling by the application is avoided. This allows the kernel to guarantee the worst case response time to external events (Section 2.2).
— HATs are executed in an environment where they may invoke, without restrictions, any service of the kernel or use any library. This lack of restrictions prevents mutual exclusion and sequencing interference between tasks and interrupt handlers, as discussed in Sections 2.3 and 2.4.

— The development and maintenance of the system is simplified (Section 2.5), because there is only one mechanism for synchronization and communication among cooperating activities.

This integrated design allows for the use of interrupts without jeopardizing the temporal determinism of the system. Also, the decrease in the complexity of this integrated design decreases the likelihood of errors favoring the development of reliable systems. Overall, this scheme allows for the development of robust and predictable systems.

4. SCHEDULABILITY ANALYSIS

In this section, we develop a schedulability analysis to evaluate both the traditional interrupt management model and our integrated model. We first analyze the decrease in the utilization bound and the response time of the traditional model, then the decrease in the utilization bound due to context switching in the integrated model. Finally, with this analysis, we evaluate the conditions under which one model is more appropriate than the other. It is worth noting that the scheduling of interrupt requests in the integrated model can be used with any priority-based scheduling algorithm. The integrated model provides a set of independent services that can be used with static or dynamic priority algorithms. However, due to the fact that the traditional interrupts model restricts the scheduling of interrupt requests to fixed priority, from here on, we will use RM scheduling for the comparisons and evaluation of our scheme.

4.1 Decrease in the Utilization Bound in the Traditional Model

According to the real-time scheduling theory, a task t_i (from a set of N tasks) is schedulable if the following holds.

$$U_{lub} \geq U_i, \tag{1}$$

where U_{lub} is the *least upper utilization bound*, which is $i(2^{1/i}\text{-}1)$ for rate monotonic scheduling (RMS) or 1 for earliest deadline first (EDF). U_i is the CPU level-i cumulative utilization, that is, the utilization due to task t_i, plus the utilization from the interference of higher-priority tasks. This can be computed as

$$U_i = \frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j}, \tag{2}$$

where C_i and T_i denote the execution time (including any system overhead) and the period of task t_i, respectively, and $P(i)$ denotes the tasks with priorities higher than the priority of t_i.

In the traditional model, the timing disturbance of the ISRs on the scheduling of task t_i due to the separate priority space can be described using the Generalized Rate-Monotonic Scheduling Theory [Klein et al. 1989]. There are two types of such disturbances, as shown in Figure 1.

— Disturbance due to interrupts, with minimum interarrival times smaller than that of task t_i and associated with non-real-time tasks. It is called disturbance, due to non-real-time tasks. For each task t_i, let $S(i)$ be the set of ISRs t_k^S of this kind, each one with computation time C_k^S and period $T_k^S < T_i$. The utilization of an ISR t_k^S in $S(i)$ is given by C_k^S / T_k^S.
— The disturbance associated with ISRs with hard timing requirements but with minimum interarrival times greater than that of task t_i. This disturbance is known as rate monotonic priority inversion. For each task t_i, let $L(i)$ be the set of ISRs t_k^L with

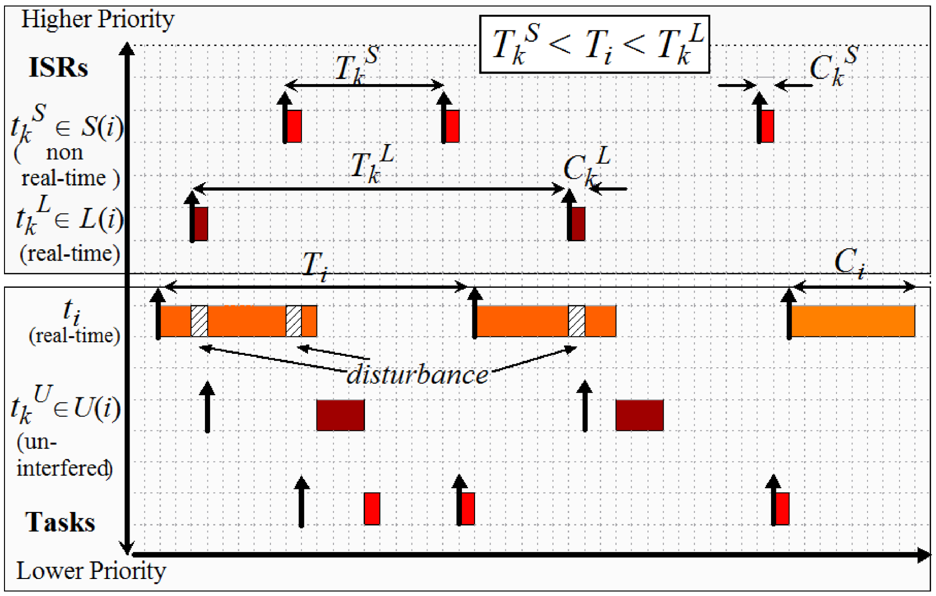

Fig. 1. Disturbances due to the separate space of tasks and ISRs.

these characteristics and C_k^L be its computation time. Since the interarrival times of these interrupts T_k^L are greater than T_i, they can preempt t_i only once, that is, at each instance. In consequence, the worst case utilization due to an ISR in $L(i)$ is C_k^L / T_i.

The equation for the utilization bound considering these two disturbances is as

$$U_i = \left(\frac{C_i}{T_i} + \sum_{j \in P(i)} \frac{C_j}{T_j} \right) + \left(\sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C_k^L \right). \tag{3}$$

The first two terms of the equation are identical to those of Equation (2). Therefore, the third and fourth terms are the decrease on the least upper utilization bound produced by the use of an independent space of interrupt priorities. Let us call this utilization decrease U_{iS}, then Equation (1) can be rewritten as

$$U_{net} = U_{lub} - U_{loss} \geq U_i, \tag{4}$$

where the utilization loss $U_{loss} = U_{iS}$ is computed by

$$U_{iS} = \sum_{k \in S(i)} \frac{C_k^S}{T_k^S} + \frac{1}{T_i} \sum_{k \in L(i)} C_k^L. \tag{5}$$

In order to minimize U_{iS}, the execution time of the ISRs (C_k^S,C_k^L) must be minimized. In this way, an ISR will perform the minimum processing necessary and activate a task. Once activated, this task will execute (as other tasks) under the control of the kernel scheduler, assigning a priority to the task according to its timings requirements.

Although this arrangement minimizes the disturbance produced by the ISRs, it does not solve the predictability problem, because it is not possible to predict the frequency of the interrupts from all devices in the system. Too many interrupts occurring during a short time interval make the system unpredictable and may cause some tasks to miss their deadlines.

In order to address this problem, some systems include additional mechanisms to set a bound on the number of the interrupts during certain time intervals [Regnier et al. 2008]. However, it is clear that these mechanisms introduce an additional overhead.

4.2 Increase in the Response Time

In the traditional scheme, the response time of an event is equal to the worst case response time of the task that communicates with the ISR. The existence of two spaces of priorities causes an increase on the response time of the tasks. The response time R_i of task t_i with execution time C_i and minimum interarrival time T_i can be computed by the following recurrence equation [Joseph and Pandya 1986].

$$R_i^n = C_i + B_i + \sum_{j \in P(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j, \tag{6}$$

where R_i^n denotes the nth iterative value ($R_i^0 = \mathrm{C}_i$), B_i is the blocking time of task t_i, and $P(i)$ is the set of tasks with higher priority than that of t_i. The third term in Equation (6) denotes the total interference suffered by t_i from tasks in the $P(i)$ set. This iterative process ends successfully when $R_i^{n-1} = R_i^n$ or unsuccessfully when $R_i^n > D_i$, where D_i denotes the deadline of task t_i. In order to consider the effect of the two spaces of independent priorities in the response time of task t_i, we must add to Equation (6) the interference of the ISR sets $S(i)$ and $L(i)$ to the response time of task t_i. Adding this interference to Equation (6), we have

$$R_i^n = \left(C_i + B_i + \sum_{j \in P(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j \right) + \left(\sum_{k \in S(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_k^S + \sum_{k \in L(i)} C_k^L \right). \tag{7}$$

The first section of Equation (7) includes three terms identical to those of Equation (6). The remaining terms (second section) denote the disturbance due to the use of an independent space of priorities on the response time R_i. However, since Equation (7) is a recurrence equation, we cannot quantify the terms of both sections separately, as is done in the utilization case (Section 4.1). It is important to note that a small increase in the second section of the equation can produce a big increase in the response time of the task.

4.3 Overhead in the Integrated Model

The drawback of the integrated model is the overhead introduced by the context switching of the HATs (that were treated before as ISRs). This overhead causes a decrease in the utilization bound.

To calculate the difference between the traditional and the integrated model, we first calculate the overhead of the traditional model. Let $H(i)$ be the set of all activities t_j^H with execution time C_j^H and minimum interarrival time T_j^H smaller than the period T_i of task t_i, which are handled by an ISR in the traditional model. Let δ^I be the total CPU time for the enter/leave code of the ISR needed to save and restore the state of the CPU and keep track of the nesting of the ISRs. Let c_j^H be the execution time from the interrupt handler itself. Then, the total execution time of an ISR in the $H(i)$ set can be computed by $C_j^H = c_j^H + \delta^I$. Therefore, Equation (2), including δ^I, can be rewritten as

$$U_i^I = \frac{C_i}{T_i} + \sum_{j \in (P(i)-H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + \delta^I}{T_J^H}. \tag{8}$$

Since all activities in the $H(i)$ set in the integrated model are treated as HATs, their context switching time must be included. Let δ^P be the context switching time. Then, the execution time C_j^H of a HAT in the $H(i)$ set can be denoted by $C_j^H = c_j^H + 2\delta^P$. In consequence, Equation (2), including δ^P, can be rewritten as

$$U_i^P = \frac{C_i}{T_i} + \sum_{j \in (P(i)-H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p}{T_J^H}. \tag{9}$$

Therefore, the decrease in utilization $\left(U_{loss} = U_i^{PI}\right)$ due to the overhead produced by the activities in the $H(i)$ set as HATs is given by

$$U_i^{PI} = U_i^P - U_i^I = \sum_{j \in H(i)} \frac{c_j^H + 2\delta^p}{T_J^H} - \sum_{j \in H(i)} \frac{c_j^H + \delta^I}{T_J^H}$$

$$U_i^{PI} = \sum_{j \in H(i)} \frac{2\delta^P - \delta^I}{T_J^H}. \tag{10}$$

The overhead of the integrated model will be smaller than the one caused by the priority inversion of the traditional model if the following condition holds.

$$U_i^{PI} < U_{iS}. \tag{11}$$

Following Equation (11), if we compare the decrease in the utilization bound of the traditional interrupt model U_{iS} (Equation (5)) against the decrease introduced by the integrated interrupt model U_i^{PI} (Equation (10)) due to the additional overhead in context switching, it is possible to observe that in most of the cases, the savings obtained using the traditional model are far smaller than those of the integrated model because of the potentially large priority inversion introduced in the traditional model.

In any case, it could be possible to design a hybrid model with a configuration in which some activities are treated as ISRs and others as HATs to satisfy the condition stated in Equation (11). For instance, since the timer interrupt always has the highest priority in the system and will never be handled by the application, it could be considered as an ISR. This reduces the $H(i)$ set, therefore reducing U_i^{PI}.

5. DESIGN OF THE LOW-LEVEL INTERRUPT MANAGEMENT SYSTEM

In this section, we describe the design of the integrated interrupt management system described in Section 3. The UML diagram of Figure 2 shows the relationships among the components involved in the interrupt management system. All kernel components that communicate with the interrupt management system use the iKRNLINT interface.

The interrupt management system is divided into two components: KRNLINT (kernel interrupt management) and INTHAL (interrupt hardware abstraction layer) The KRNLINT contains the hardware-independent management code. The INTHAL, on the other hand, contains the hardware-dependent management code. All communication between KRNLINT and INTHAL occurs through the iINTHAL interface.

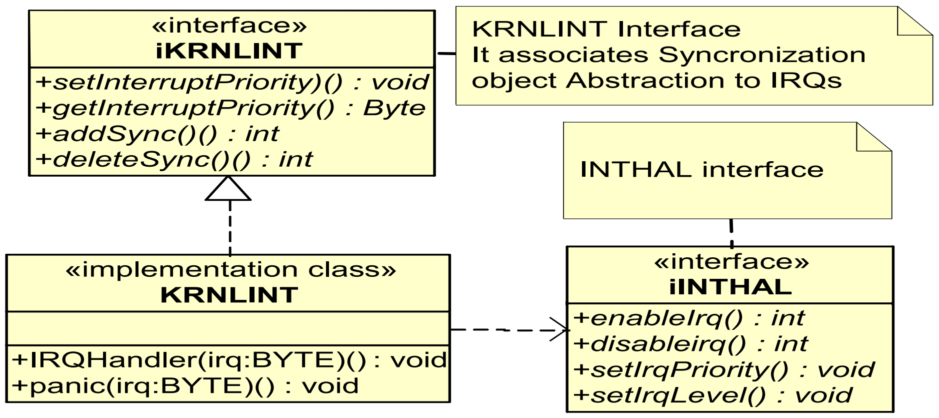

Fig. 2. INTHAL component interfaces.

5.1 Kernel Interrupt Management Component

The KRNLINT component supplies the low-level mechanisms to allow the rest of the system (specifically the scheduling modules and the synchronization and communication modules) to treat the interrupts using the same scheduling and synchronization policies than those used for the real-time tasks. The responsibilities of this component are the following.

— To enable the association of synchronization objects (i.e., semaphores, mailboxes, etc.) with each one of the IRQ lines (*addSync*() in Figure 2). These synchronization objects are identified by a single synchronization identifier (*syncId*).
— To generate a signal for the synchronization objects each time an IRQ arrives.
— To supply mechanisms for interrupt management.

The KRNLINT component creates the kernel interrupt abstractions, which are identified by a predefined interrupt identifier of type *irqId*. Each kernel interrupt is associated with a priority within the unified space.

5.2 Interrupt Hardware Abstraction Layer

The INTHAL component provides interrupt management at the lowest level. It is in charge of those aspects that depend on the interrupt hardware. The goal of this component is to provide an abstraction to make the system as independent as possible from the computer architecture. The responsibilities of this component are the following.

— To provide a set of interrupt request lines independent of the hardware architecture that go from IRQ0 to IRQn (only *n* depends on the hardware architecture).
— To provide an interface to set the priorities for each IRQx lines, independently of the interrupt hardware.
— To provide the capacity for setting an IRQ level under which interrupts are disabled.

The priorities for each IRQ can be set between 0 and the highest scheduling priority (in our implementation, it is 255). The value 0 indicates that the related IRQ is disabled.

When the system is started, all IRQs are in an *ignored* state. An IRQ changes to a *captured* state when the kernel requests attention to the IRQ explicitly by invoking an *enableIrq*() service. A *captured* IRQ can be in an *enabled* or *disabled* state. It is enabled when their IRQ level is above the current IRQ level. The activation of the captured and enabled IRQs produces the invocation of the *IRQHandler*() kernel routine. An IRQ is disabled when its level is below or equal to the current IRQ level (so the *IRQHandler*() is not invoked).

Once an IRQ is captured, its priority can be modified using *setIrqPriority*(*irq, priority*). The current IRQ level can be set at any moment using *setIrqLevel*(*priority*). All IRQs with a priority below the system IRQ level are disabled. After an IRQ has been captured and each time it is triggered, if its priority is higher than the current system IRQ level, then the control is transferred to the *IRQHandler*(*irq*) (passing the corresponding IRQ as a parameter).

This design allows the kernel to be independent of the interrupt hardware. Several alternative modules of the INTHAL can be implemented, each one for different interrupt hardware architectures. A possible implementation may use an FPGA to implement a *custom programmable interrupt controller* (CPIC) that cooperates with the kernel to jointly schedule tasks and IRQs. This implementation would introduce a minimum overhead, but it is not possible on systems with conventional interrupt hardware.

6. IMPLEMENTATION OVER CONVENTIONAL PC INTERRUPT HARDWARE

In this section, we provide an implementation over the conventional PC hardware. First, for the sake of completion, we provide an introduction of the interrupt hardware in order to understand the remainder of this section.

6.1 Conventional Interrupt Hardware Overview

Computer systems using the Intel family processors and compliant with the industry standard use an interrupt hardware composed of two programmable interrupts controllers (PIC) 8259A chips (or equivalent inside the motherboard chipset) connected in cascade (through the IRQ2 of the first 8259A – master PIC). This configuration provides 16 IRQ lines (IRQ0... IRQ15).

Each 8259A can control eight prioritized IRQ lines and has several internal 8-bit registers that affect its operation. The most important registers for our scheme are (1) the *interrupt request register* (IRRE) in which each bit set in this register indicates that the corresponding IRQ line has signaled an interrupt; (2) the *interrupt mask register* (IMRE) in which each bit set in this register indicates that the corresponding IRQ line is masked (or disabled), otherwise it is enabled (or unmasked); and (3) the *interrupt service register* (ISRE) in which each bit set in this register indicates that the corresponding IRQ is being serviced.

The bits in the ISRE register are set when the processor acknowledges the corresponding IRQ issued by the PIC. This acknowledgment protocol is controlled by the hardware. The software can only disable it by clearing the processor's global interrupt flag (disabling all interrupts). The IRRE keeps track of the interrupts that are in service in the CPU so that when an IRQ arrives, it does not cause an interrupt to the CPU while higher- or equal-priority interrupts are being serviced.

A bit in the ISR can be cleared (producing the enabling of lower or equal priority interrupts) according to the following end of interrupt (EOI) modes.

— *Normal* (*or explicit*) *EOI mode*. The ISRE's bits are cleared when the (interrupt service routine) software issues an *EOI command* to the PIC.
— *Automatic EOI mode*. The ISRE's bit, which corresponds to the requested IRQ, is automatically cleared when the CPU acknowledges the interrupt request. After that, the 8259A can send another lower or equal-priority request to the processor, interrupting the previous one.

Most commercial operating systems use the normal EOI, because it allows the PIC to control the priority of the IRQs.

6.2 Emulation Using Physical Interrupt Masking

Neither the 8259A traditional *programmable interrupt controller* (PIC) nor the modern *advanced PIC* (APIC) included in the recent PCs supports the joint scheduling with the kernel of the interrupt requests required by the integrated model.

To cope with this problem, the INTHAL cancels the built-in hardware interrupt priorities and establishes an interrupt priority scheme compatible with the kernel scheduler. This is accomplished in two stages.

(1) Cancellation of the PIC's automatic priority handling. Essentially, the INTHAL must ensure that the ISRE registers of both 8259A are set to allow all IRQs enabled explicitly by the IMREs. This is possible by capturing all ISRs and using one of the following EOI modes.
 — *EOI Mode*. Sending the *end of interrupt* command (EOI) to the 8259A controller.
 — *AEOI Mode*. Using the 8259A *automatic end of interrupt* operation mode.
(2) Software Priority Management. Once each IRQ occurs, the INTHAL must explicitly set the IMRE registers of each 8259A with a mask to disable all IRQs with lower or equal priority (including the current IRQ) and enable all IRQs with higher priorities.

This emulation is handled by the INTHAL, which is in charge of maintaining the state of a *virtual custom programmable interrupt controller* (VCPIC) capable of supporting the integrated model. The VCPIC keeps a table with the current priority for each IRQ and the current system priority level. Any time there is a change in the status of the VCPIC, the INTHAL calculates and sets the appropriate mask for the IMRE of the two 8259A interrupt controllers.

Figure 3 shows an UML sequence diagram illustrating the elements involved on the interrupt treatment and their sequence of events (and messages) produced when an interrupt occurs. In the left side of the figure, two hardware elements are depicted: the device which issues the interrupt and the PIC 8259. The other hardware element involved is the CPU. Instead of depicting the CPU, we are depicting the software elements involved in the interrupt request: the VCPIC (INTHAL) and KRNLINT components of the kernel, as well as the synchronization object associated with the interrupt (IRQx Sync Object) and the HAT which handles the request from the device (IRQxHAT).

Note that, at the right side of the figure, IRQxHAT is handling the interrupt by invoking the wait() operation on the associated object IRQxSync Object (message 1). This operation blocks the task until the IRQ occurs (message 2).

When any device issues an IRQ (message 3), the PIC and the CPU interrupt vector table allow the invocation of the LLIH in the INTHAL by sending the IRQ as an argument (message 5). The heart of this handler is the integrated priority-space emulation algorithm (message 6), which uses the 8259 IMRE and the EOI command (messages 7 and 8).

The details of this emulation algorithm will be introduced in the following sections. The final result of this algorithm is the setting of an adequate priority and the invocation of the IRQHandler() in KRNLINT (message 9).

The goal of the KRNLINT is to find the synchronization object associated with the IRQ (message 10) to invoke the signal operation on that object (message 11). The signal operation will then unblock the task waiting for the interrupt (message 12), which then will service the IRQ from the device.

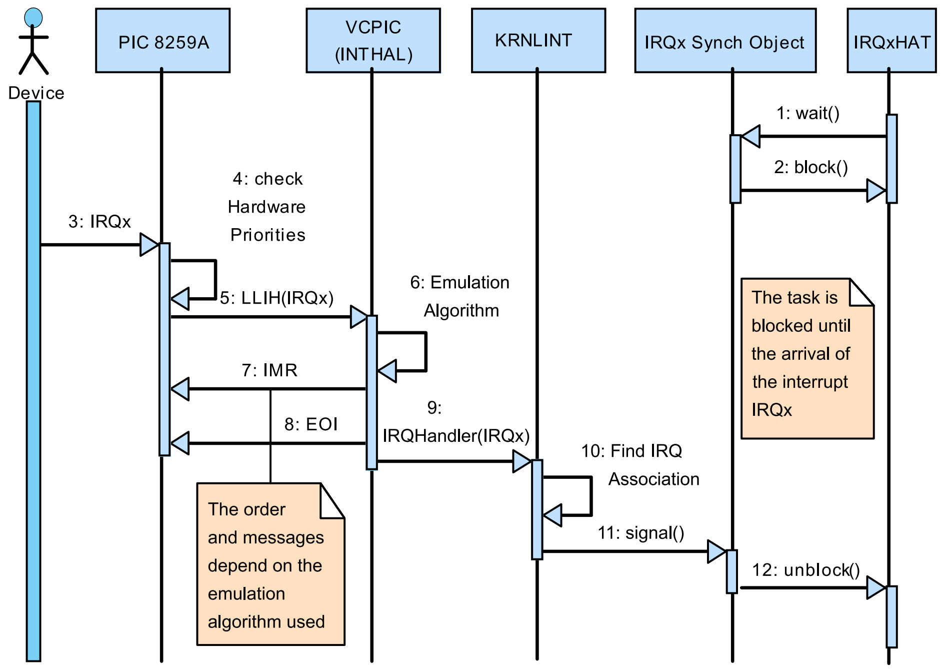

Fig. 3. Sequence of events upon an IRQ arrival (LLIH's role).

6.3 Analysis with Physical Masking

The implementation of the VCPIC introduces an additional overhead in context switching due to the computing (and setting) of the current IRQ level in the system interrupt hardware. Let δ^M be this overhead time. Then Equation (9), including δ^M, can be rewritten as

$$U_i^P = \frac{C_i + 2\delta^M}{T_i} + \sum_{j\in(P(i)-H(i))} \frac{C_j + 2\delta^M}{T_j} + \sum_{j\in H(i)} \frac{c_j^H + 2\delta^p + 2\delta^M}{T_J^H}.$$

Now the decrease in utilization $U_{loss} = U^{PI*}$ due to the overhead of the integrate model will become

$$U_i^{PI*} = \frac{2\delta^M}{T_i} + \sum_{j\in(P(i)-H(i))} \frac{2\delta^M}{T_j} + \sum_{j\in H(i)} \frac{2\delta^p + 2\delta^M - \delta^I}{T_J^H}. \tag{12}$$

As can be noted, implementing the system in this way imposes a performance penalty. It is worth nothing that even when U_i^{PI*} is not smaller than U_{iS}, this scheme has the advantages discussed in Section 3. In systems where the predictability is so important, this implementation may be an alternative to the interrupts avoidance [Kopetz et al. 1989]. These applications may trade an increase in overhead for obtaining determinism without sacrificing the benefits of the treatment of external events by interrupts.

7. USING THE VIRTUAL INTERRUPT MASKING

In this section, we introduce an alternative to the software implementation. It is a variant of the optimistic interrupt protection model that we called *virtual interrupt masking*.

7.1 Optimistic Interrupt Protection

Kernels of general-purpose OS often disable the interrupts to avoid preemptions when certain code sections modify data shared with ISRs. At the end of these critical sections, the kernel must enable the interrupts again. With the aim of speeding up this entry/leave protocol to the critical sections, a technique called *optimistic interrupt protection* (OIP) was introduced by Stodolsky et al. [1993] and consists of the following stages.

(1) When entering a critical section inside the kernel, the protocol sets a software interrupt mask to indicate what interrupts must be masked. The hardware interrupt mask is not changed.
(2) A prologue code section is located at the entry of all interrupt handlers. This prologue code checks the software interrupt mask to verify if the issued interrupt is logically masked, and if so, the execution of the remainder of the ISR is deferred to a later moment.
(3) When leaving the critical section, the protocol checks if there are any pending interrupts. If this is the case, the control is transferred to the corresponding ISR before resuming the "normal" computation.

In order to simplify the code, OIP recommends that in the event of a logically masked interrupt, besides remembering the interrupt request and before returning the control, the interrupt prologue should update the hardware mask, as specified in the software mask. In this case, after the deferred interrupts are handled as part of the leaving protocol of the critical section, the hardware interrupt mask must also be restored to its original level.

7.2 Adapting OIP to Real-Time Systems

The performance penalty analyzed in Section 6.3 can be decreased substantially if the OIP approach is adapted to our integrated model. With this technique, when the system IRQ level is raised from level A to level B, the IRQs with priority levels between A and B are not really disabled so that these undesired IRQs can occur. If any of these IRQs occurs, then the IRQ is really masked to avoid future occurrences. Let P_C be the current system priority level, then any IRQ I with priority P_i that occurs in an undesired way fulfills with the condition $P_i \leq P_C$.

Unlike the initial idea of optimistic protection, the masking of undesired IRQs (in the 8259's IMRE) after their occurrence is not optional for simplifying the implementation logic but, rather, becomes mandatory. Therefore, the occurrence of a second undesired IRQ is avoided, and the temporal predictability is guaranteed. In this case, the IRQ is recorded so that it can be issued when the priority level is low enough. Furthermore, when the system priority P_C is decreased, it is necessary to verify whether an IRQ that has been masked could occur at the new level and, in this case, modify the mask of those IRQs that should be enabled.

7.3 Adapting the Masking for the Integrated Model

There are three possible ways to carry out the masking of an undesired IRQ I in the LLIH, all of them guaranteeing a maximum bound in the priority inversion due to the disturbance of these undesired IRQs.

(1) Masking only the undesired IRQ I that occurred. This masking involves computing and setting a mask that disables the specific IRQ (without modifying the others).
(2) Masking all IRQs with priorities below or equal to P_i. This option has two advantages: (1) it is easy to implement because it is only required to set the mask

(precalculated by setIrqPriority()), and (2) when setIrqLevel() is called to raise the priority level, it does not have to calculate the mask corresponding to the new priority level (it must be done only when decreasing the priority level). Since setIrqLevel() must be executed at each context-switching, advantage (2) causes a smaller context-switching overhead. However, this option has the drawback of allowing the occurrence of other undesired IRQs (all IRQ x with priority P_x that fulfill the condition $P_c < P_x < P_i$). This would cause not only the masking of other IRQs but also would produce a larger worst case disturbance due to undesired IRQs.

(3) Masking all IRQs with priorities below or equal to PC. This option is equivalent to setting the physical IRQ level (physical mask) equal to the system (logical) priority level (logical mask). With this option, the service setIrqLevel() must compute the interrupt mask that corresponds to each level. This must be done even if the invocation raises or diminishes the current system priority level. The drawback here is a higher average-case context-switch overhead. However, it guarantees that once an undesired interrupt occurs, any other one will not occur when the system priority level is higher or equal to the current level (the first undesired IRQ masks all the others). This provides the best possible worst case in the disturbance due to undesired interrupts.

7.4 Adapting the Recording of Undesired IRQs

After an undesired interrupt occurs and after it is masked, it must be recorded to allow its occurrence only when the system priority goes below the priority of this interrupt. Here, the integrated model of interrupt management enables both VCPIC (INTHAL) and Kernel (KRNLINT) recording (described next).

7.4.1 VCPIC (INTHAL) Recording. This option is equivalent to the initial idea of the interrupt prologue in the optimistic interrupt masking (and the only one available with traditional interrupt management). In order to achieve this, it is necessary to keep an occurrence flag for each possible IRQ (called continuation in Stodolsky et al. [1993]) to record the occurrence of the undesired IRQs. When the system IRQ level goes down, the interrupt occurrence is simulated, executing IRQHandler(irq) in the INTHAL's setIrqLevel() service. This option has the advantage of making the kernel independent of the masking modes in the INTHAL.

Although this option increases the execution time δ^M of setIrqLevel(), it is not a problem for the traditional interrupt management systems (for which the optimistic masking was designed), because setIrqLevel() is not executed at every context switching (as in the integrated model) but as part of the entry/leave protocol of the kernel's critical sections. In fact, this service is less expensive than the direct handling of the IRQ level.

Due to a change to the scheme and to the kernel design objectives, for the integrated scheme and for real-time operating systems, we have the following issues.

(1) Now the setIrqLevel() is not called as part of the entry/leave protocol to the kernel's critical sections but as part of each context switch. In fact, the context switch itself constitutes a critical section; hence, it is not possible to simulate an interrupt within setIrqLevel(). Note that by using the integrated model, interrupt disabling inside the kernel is not needed. The kernel's critical sections are protected simply by disabling the preemption (modeled by immediate priority ceiling protocol [Theodore 1990]).
(2) The efficiency of the integrated scheme and the achieved schedulability is very sensitive to the worst case execution time of the setIrqLevel() service δ^M. Hence, any small increase in δ^M is an important disadvantage.

Table I. Optimistic Masking vs. Virtual Masking

Techniques	Optimistic Masking	Virtual Masking
Target systems	General-purpose OS	Real-Time OS
Physical masking	Optional, with the aim of simplifying the implementation	Mandatory, for guaranteeing the temporal predictability
Omitting the execution of an undesired IRQ	Explicit in the prologue of each ISR	Automatic in the scheduler (does not run due to its priority)
Undesired interrupt recording	Explicit in the prologue of each ISR	Automatic in the synchronization object associated to the IRQ
Deferred handler execution	Explicit in the exit protocol of the critical sections	Automatic in the kernel scheduler (when it is the highest priority)

7.4.2 Kernel (KRNLINT) Recording. A solution for these two preceding issues can be found in the model itself. Due to the integration of communication and synchronization between interrupt handlers and tasks, the VCPIC does not need to hide the occurrence of an undesired IRQ from the kernel, either to notify the kernel if an IRQ is desired or not. The kernel itself has enough information to differentiate desired from undesired interrupts (using the system priority level).

The fact is that when either a desired or undesired IRQ occurs (i.e., IRQHandler(irq) is called), the kernel signals the synchronization objects associated with that IRQ and calls the scheduler. If only signal-recording objects (semaphores, mailboxes, etc.) are associated with IRQs, then the recording of an undesired IRQ is achieved transparently. In this case, a desired IRQ would cause the preemption of the current task to execute the HAT that waits for that IRQ (because it has higher priority than the current task), while an undesired IRQ would set ready the associated HAT without preemption (because its HAT has lower priority than the current task). The HAT is automatically scheduled when the system priority level goes down.

In virtual masking mode, if an interrupt I occur with priority $P_i > P_C$, then LLIH must do the same operation as in the physical masking mode with the exception of the masking of the IRQ.

The differences between the idea of the optimistic interrupt protection [Stodolsky et al. 1993] and our adaptation for the integrated model (*virtual masking*) are summarized in Table I.

7.5 Analysis with Virtual Masking

When virtual masking is used, the masking of an IRQ may occur only if this IRQ really takes place (in undesired form), while the unmasking only takes place if, as part of a context switch to exit some activity, it is needed to enable this IRQ again.

For activities with higher priority than the current task ($P(i)$ set), the worst case situation takes place when all the IRQs with smaller periods than that of the current task ($H(i)$ set) occur in an undesired form, while the activities in $P(i)$ of higher priority than that of the corresponding IRQ are being executed. In this case, each of them would cause a first writing of the mask from its handler and a second writing when the preempted activity in $P(i)$ ends. However, since this writings take place only if the IRQs occur, it should not be associated with each context switch in $P(i)$. Instead, it is enough to associate two mask writings ($2\delta^M$) to each possible activation of the activities in $H(i)$. Hence, the utilization U_i^M due to the disturbance of the HATs in the $H(i)$ set over the task t_i is computed as

$$U_i^M = \sum_{j \in H(i)} \frac{c_i^H + 2\delta^p + \gamma + 2\delta^M}{T_J^H}, \tag{13}$$

where γ is the execution time of the prologue associated to the undesired IRQs, which is needed to record their occurrence. Moreover, now it is also necessary to take into account the disturbance associated with the potential execution of the small prologues caused for attending the IRQs associated with any of the activities in the non-real-time interrupts ($S(i)$) and the real-time interrupts with larger periods than that of the current task ($L(i)$) sets (not the handling activity itself). This prologue is in charge of masking this IRQ. In this case, only one masking must be taken into account for each undesired IRQ that occurs, because the context switch of any activity in $P(i)$ never produces the unmasking of any of the IRQs associated with activities in $S(i)$ or $L(i)$. However, the number of times that this prologue may be executed depends on the way that the masking is performed (as described in the cases of Section 7.3). Then, according to these cases, Equation (9) can be transformed as follows.

(1) Masking only the current IRQ or all IRQs with priorities lower than or equal to that of the current IRQ (i.e., cases (1) and (2) in Section 7.3). In this case, each (undesired) IRQ in $S(i) \cup L(i)$ can occur only once in the worst case. Hence, Equation (9) now can be written as

$$U_i^P = \frac{C_i + |S(i) \cup L(i)| \left(\gamma + \delta^M\right)}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j}{T_j} + U_i^M. \tag{14}$$

From Equation (14), the decrease in the utilization $U_{loss} = U_i^{PI*}$ due to the overhead of the integrated model with virtual masking (using masking cases (1) or (2) of Section 7.3) is computed by

$$U_i^{PI*} = \frac{|S(i) \cup L(i)| \left(\gamma + \delta^M\right)}{T_i} + \sum_{j \in H(i)} \frac{\gamma + 2\delta^p + 2\delta^M}{T_J^H}. \tag{15}$$

Note that despite that cases (1) and (2) of Section 7.3 yield the same worst-case utilization loss, this would only occur in case (2) if all IRQs in $S(i) \cup L(i)$ occur in inverse priority order.

(2) Setting the mask that matches the current system priority P_C (i.e., case (3) of Section 7.3). In this case, only one (undesired) IRQ in $S(i) \cup L(i)$ may occur in the worst case. Hence, now Equation (9) can be written as

$$U_i^P = \frac{\gamma + \delta^M}{T_i} + \sum_{j \in (P(i) - H(i))} \frac{C_j}{T_j} + U_i^M. \tag{16}$$

Consequently, from Equation (16), the decrease in the utilization $U_{loss} = U_i^{PI*}$ due to the overhead of the integrated model with virtual masking (case (3)) can be written as

$$U_i^{PI*} = \frac{\gamma + \delta^M}{T_i} + \sum_{j \in H(i)} \frac{\gamma + 2\delta^p + 2\delta^M}{T_J^H}. \tag{17}$$

Note that, different from the traditional scheme which uses a minimal ISR and delegates the service at task level (Section 2 and Figure (1)), this virtual masking scheme is temporally predictable. It introduces a priority inversion due to a small disturbance caused by the execution of a prologue of an undesired interrupt (given by $\gamma + \delta^M$). However, as showed in Equations (15) or (17), this priority inversion is bounded. This scheme guarantees a predictable and efficient interrupt management with a very small utilization loss. Also, note that now U_i^{PI*} in Equation (17) depends only on the HATs in the $H(i)$ set and not on the SAT (as occurs in Equation (12)), making the system more scalable.

8. IMPLEMENTING THE INTEGRATED MODEL

This section presents the detailed design of the interrupt subsystem for a real-time kernel that is compliant with the integrated interrupt and task model. This subsystem can be configured in different emulation modes to satisfy the different trade-offs between cost and temporal predictability.

8.1 INTHAL Status Data

In order to keep the state of the VCPIC, the following arrays (with one element for each of the 16 IRQs) and variables are maintained.

—*IRQ_Priority*. An array of elements of a type compatible with the system priorities (byte or word) that holds the priority of each IRQ in the system.
—*IRQ_Mask*. An array of words (16 bits) with each word keeping one mask to be set in the IMREs of both 8259s when the corresponding IRQ occurs. This word unmasks the IRQ of higher priority than the matching IRQ and masks all others (including the matching IRQ).
—*Virtual_Mask_Mode*. Flag that signals the masking mode in operation: virtual (TRUE) or physical (FALSE).
—*IRQ_Level*. Byte or word that keeps the current (unified) system priority level.
—*Logical_Mask*. Interrupt mask (word value) that corresponds to the current system priority (*IRQ_Level*).
—*Physical_Mask*. Interrupt mask (word value) set in the IMREs of both 8259s. In the physical masking mode, it will always be equal to Logical_Mask, but in virtual masking mode, it may be different.

8.2 Priority Management Services

The priority management services are used to implement the unified space of priorities. The interface services provided are setIrqPriority() and setIrqLevel(), and the auxiliary services provided are set8259IMR() and setIRQMask().

The auxiliary service set8259IMR(...) must be invoked whenever it is necessary to set the mask in the interrupt hardware. It keeps the current value of the mask registers in the Physical_Mask variable so that whenever the new mask matches the mask already set, the expensive input/output operations are avoided. The algorithm used in this service is shown in Figure 4. First, it verifies whether the physical mask is different from the new mask. If this is true, it sets the masks in both 8259 IMREs and updates the values of Logical_Mask and Physical_Mask.

The setIRQMask(...) service provides support for virtual masking. As shown in Figure 4, its behavior is related to the masking mode stored in the state variable Virtual_Mask_Mode. In physical mode, it only calls the set8259IMR() service to set the mask in both 8259s. In virtual masking, it sets the mask only if it causes an IRQ enable (unmasking), otherwise only the value of the logical mask (Logical_Mask) is updated.

In physical masking mode, the value of Logical_Mask is always equal to that of the IMREs of both 8259s (and Physical_Mask), but in virtual masking mode, these values

```
set8259IMR (mask) {
  if ( mask ≠ Physical_Mask )  {
     Logical_Mask ← mask
     Physical_Mask ← mask
     IMR registers of both 8259 ← mask
  }
}

setIRQMask (mask) {
  if ( VirtualMaskMode =  TRUE)  {
     if ( Physical_Mask AND ( NOT mask ) )
       set8259IMR(mask)
     else
       Logical_Mask = mask
  }
  else {
    set8259IMR (mask)
  }
}
```

Fig. 4. Auxiliary services set8259IMR() and setIRQMask().

```
setIrqPriority (irq, priority) {
  /* Compute all masks for the priority setting*/
  IRQ_Priority[irq] ← priority
  IRQ_Mask[] ← calc_new_mask(irq,priority)
  /*  Update de current 8259 mask */
    irqMaskBit ← (1 << irq)  /*set bit in IRQ position*/
  if ( priority ≤ irqLevel ) { /* mask IRQ */
    setIRQMask( Physical_Mask  OR  irqMaskBit )
  }
  else { /* IRQ activation*/
   setIRQMask(Physical_Mask AND NOT irqMaskBit)
  }
}
```

Fig. 5. SetIrqPriority() service.

may be different. However, the bits in “1” in Physical_Mask always must be a subset of the bits in “1” in Logical_Mask. In other words, the following must hold.

$$(\text{NOT Logical\_Mask AND Physical\_Mask}) = 0$$

The setIrqPriority(irq, priority) service allows the setting of the priority level of an IRQ. Its function is to establish a correspondence between the priorities assigned to each IRQ (within the system priority space) with the value of the mask to be set in the interrupt hardware (IMREs of both 8259). Its algorithm (shown in Figure 5) keeps the IRQ_Priority and IRQ_Mask arrays (see Section 8.1). At each invocation, the entry in IRQ_Priority that corresponds with the IRQ being changed is updated. Next, the mask associated with each IRQ is obtained from the new priority configuration and stored in the IRQ_Mask array. Also, if this IRQ goes from

```
setIrqLevel (priority) {
 /* Get the new interrupt mask */
 irqi ← find_highest_prior_irq(priority)
 if (irqi = 255) /* no irq found */
   setIRQMask( 0 )
 else
   setIRQMask( IRQ_Mask[irqi] )
 IRQ_level ← priority
}
```

Fig. 6. SetIrqLevel() service.

```
CAPTURED_ENTRY(irq) {
  save_CPU_registers()
  if ( Virtual_Mask_Mode = TRUE ) {
    if (IRQ_Priority[irq] ≤ irqLevel) {
      Set8259IMR (Logical_Mask)
    } else {
      Logical_Mask ← IRQ_Mask[irq]
      IRQ_Level ← IRQ_Priority[irq]
    }
  } else {
    Set8259IMR(IRQ_Mask[irq])
    IRQ_Level ← IRQ_Priority[irq]
  }
  sendEOI
  IRQHAndler(irq) /* Enter to the kernel */
  restore_used_CPU_registers()
}
```

Fig. 7. LLIH for captured IRQs.

enabled to disabled or vice versa, then setIRQMask() is called to update the interrupt mask.

The setIrqLevel(priority) service sets the current system IRQ level and maintains the IRQ_Level variable (Figure 6). It uses the IRQ_Priority and IRQ_Mask arrays to determine the mask to be set for the new priority level. This mask is the one that disables all IRQs with a priority level lower or equal to the IRQ_Level. After computation, if this mask causes the masking or the unmasking of some IRQ, then it is set according to the masking mode (physical or virtual) using the setIRQMask().

8.3 INTHAL Low-Level Interrupt Handler (LLIH)

The INTHAL has a LLIH for each possible ISR state (captured or ignored). The control is transferred to these handlers whenever an IRQ with the associated state occurs. The handler gets the requested IRQ as an argument. The CAPTURED_ENTRY algorithm (which is the LLIH associated to the captured interrupts) is shown in Figure 7. Its main responsibilities are (1) to cancel the PICs conventional priority scheme, (2) to enforce the unified priority space, and (3) to transfer the control to the kernel interrupt handler.

When operating in the physical masking mode, if an interrupt I with priority P_i,occurs, it is because the condition $P_i >$ IRQ_Level is satisfied. In this case, all IRQs with priorities lower or equal than the IRQ_Level are masked, and the system priority level is raised to set it equal to P_i. Finally, the kernel handler (IRQHandler()) is called.

When operating in the virtual masking mode, in spite of the current system priority level IRQ_Level, the occurrence of any interrupt is possible. Here, two situations may occur.

—*An interrupt I with priority* $P_i >$ IRQ_Level occurred *(desirable interrupt)*. In this case, the same operations that took place in the physical masking mode must be carried out with the exception of the IRQ masking.
—*An interrupt I with priority* $P_i \leq$ IRQ_Level occurred *(undesirable interrupt)*. In this case, to avoid the occurrence of a second undesirable IRQ, the IMREs of both 8259s must be set using the mask which matches the priority level that was active when the IRQ took place (also, the physical mask is updated). Finally, the IRQHandler() is invoked to record its occurrence (even though the interrupt is undesired, see Section 3). Note that in this case, the current IRQ level is not modified.

8.4 Implementing the Integrated Model in an RTOS

The implementation of the integrated model in an RTOS requires the following.

—The INTHAL component interfaces (described in Figure 2) which constitute the interrupt management system of the integrated model must be included in the RTOS.
—All accesses to the interrupt hardware must be made through the iKRNLINT and iINTHAL interfaces.
—The CAPTURED_ENTRY algorithm (LLIH) described in Figure 7 must be implemented for each IRQ captured. The responsibilities of this algorithm are described in Section 8.3.
—The setIrqPriority(irq,priority) must be used to set the priority level of the IRQs. This function links the IRQ to the priority of the system. The setIrqLevel() service must be used by the kernel whenever a change on the current task priority level is needed. This service will notify this change to the interrupt hardware.
—A synchronization object (semaphore or mailbox) must be attached to the IRQs to be used by the interrupt API. This must be done through the KRNINT component described in Figure 2 of Section 5.1. Each time an IRQ arrives, a signal to the synchronization object must be generated.

9. EXPERIMENTAL RESULTS

The experiments were executed in the PARTEMOS real-time micro kernel. PARTEMOS was developed to support the integrated model along with the typical services found in commercial real-time operating systems. The services in PARTEMOS include task and interrupt management, synchronization services using semaphores, communication services using mailboxes, and exception handing [Leyva-del-Foyo et al. 2006a].

Two types of experiments were conducted. The first type allowed us to verify experimentally the deterministic behavior of the implementation of a single priority space with and without virtual masking. The second type was developed to compare the overhead of the different implementations of the integrated model over a conventional PC hardware. The experiments were executed on an Intel Pentium 4 PC running at 2.8GHz with 1GB of memory and 1MB of L2 cache. The *time stamp counter* of the CPU was used for measurements.

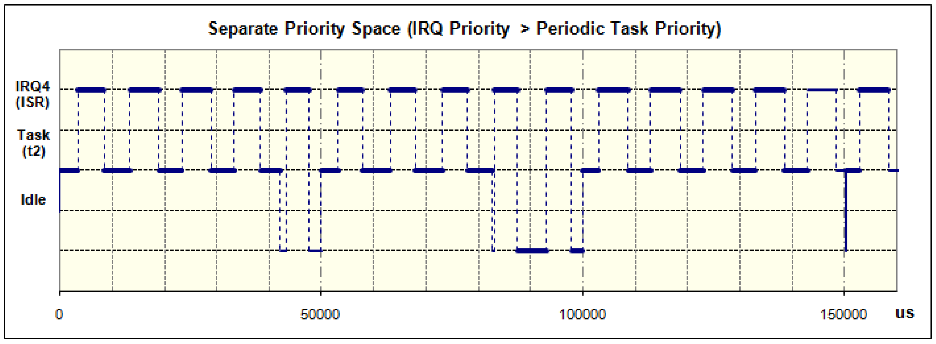

Fig. 8. Execution trace—separate priorities.

9.1 Behavior Characterization

In the first type of experiments, we conduct three different cases with different configurations of a task set consisting of the following tasks.

— t_1^S is an IRQ handler (without hard real-time requirements) that attends the serial port (receiving 100 bytes per second) with a minimum interarrival time T_1^S of 10 ms and a worst case execution time C_1^S of 5 ms (utilization $U_1^S = 0.5$).
— t_2 is a periodic hard real-time task with a period T_2 of 50 ms, a worst case execution time C_2 of 20 ms (with a utilization $U_2 = 0.4$), and a deadline of 30 ms.

In all experiments, the traces for the start and end of both activities were logged in addition to a trace for each time that the LLIH is invoked passing as a parameter the IRQ associated with the serial port (and to the HAT t_1^S).

In the first case, no integrated priority space is used, and hence the t_1^S has a higher priority than t_2. Figure 8 depicts the execution trace of the task set over a period of time. In this trace, two things are worth noting. After the first activation of task t_2 at time 0, it suffers multiple preemptions by the ISR. Also, these preemptions force t_2 to miss its deadline at time 30,000 μs. This situation is repeated for all activations of t_2 shown.

In the second and third cases, we used our integrated model to assign to task t_2 a priority higher than that of the HAT t_1^S. Note that for this particular task set, this priority configuration is the only one that guarantees the temporal requirements of the periodic task. This configuration is only possible with the integrated interrupt and task model.

In the second case, we used the emulation of the VCPIC with physical masking. Figure 9 depicts the execution trace of this case. It is worth noting that (1) the IRQ cannot preempt the periodic task t_2 (there is not a LLIH trace), hence once it is activated (at 0 μs, 50,000 μs, 10,000 μs, and 150,000 μs), it runs without disturbance. Without this disturbance, task t_2 can finish properly before its deadline in all instances, as shown in the figure (at 30,000 μs, 80,000 μs, and 13,000 μs). (2) For each period of t_2 only four IRQs are accepted and handled (instead of five that should be accepted). During the 20 ms of execution of t_2, two IRQs are issued by the serial port, but they are not attended to, because they have lower priority than task t_2. However, the hardware records one of them causing the back-to-back execution of the HAT at 20,000 μs, 70,000 μs, and 12,000 μs.

It is worth commenting about the loss of some interrupt request signals observed in Figure 9, which is caused by this priority configuration. The first comment is that at this point we have an unavoidable trade-off: in this task set, the system

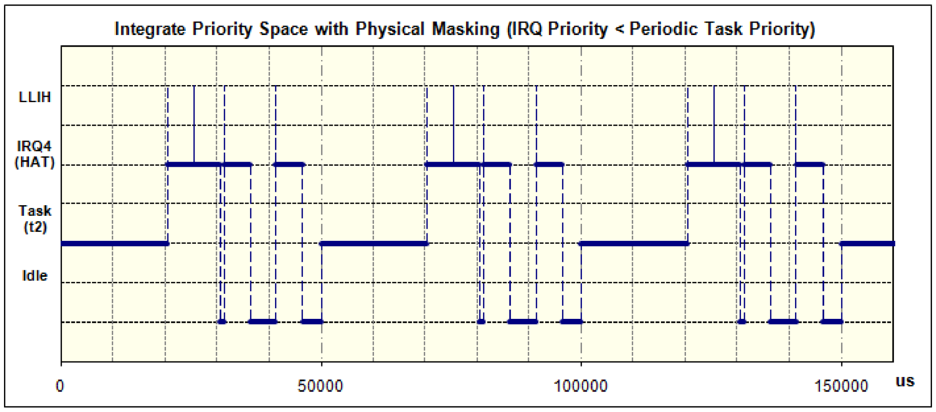

Fig. 9. Execution trace—physical masking.

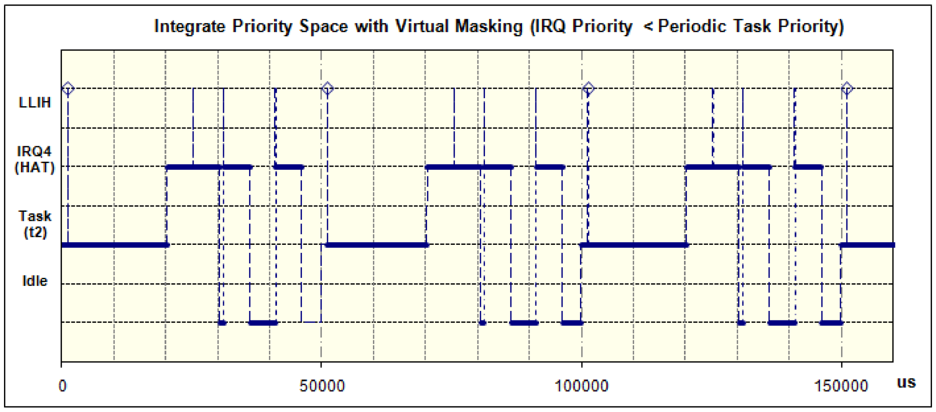

Fig. 10. Execution trace—virtual masking.

cannot guarantee the processing of all interrupts and also guarantee the meeting of the deadline of the periodic hard real-time task. Indeed, this is the reason for this priority configuration, that is, to guarantee the temporal requirements of those software-activated tasks which have hard real-time requirements, in spite of the overload caused by those hardware-activated non-real-time tasks. The second comment is that by the usage of a real-time analysis, we certainly can guarantee that interrupts (HATs) are never missed when all interrupt sources behave as expected and, at the same time, do not affect the timing requirements of the hard real-time tasks.

In the third case, we used the emulation of the VCPIC with virtual masking, as proposed in Section 7. Figure 10 depicts its execution trace. In this case, it is worth mentioning that the HAT associated with the IRQ cannot preempt periodic task t_2, hence, similar to the previous experiment, task t_2 can finish before its deadline in all instances (as shown in Figure 10 at 30,000 μs, 80,000 μs, and 13,000 μs). However, in this case, there is a difference: now the IRQ really preempts the execution of t_2. This is shown in the figure by the LLIH traces (depicted by diamonds) which occur a little later of 0 μs, 50,000 μs, 100,000 μs and 150,000 μs. Note that here, the corresponding HAT is not executed and that these undesired interrupts are not serviced at those instants of time, but instead, they are recorded (by the synchronization object) until the end of

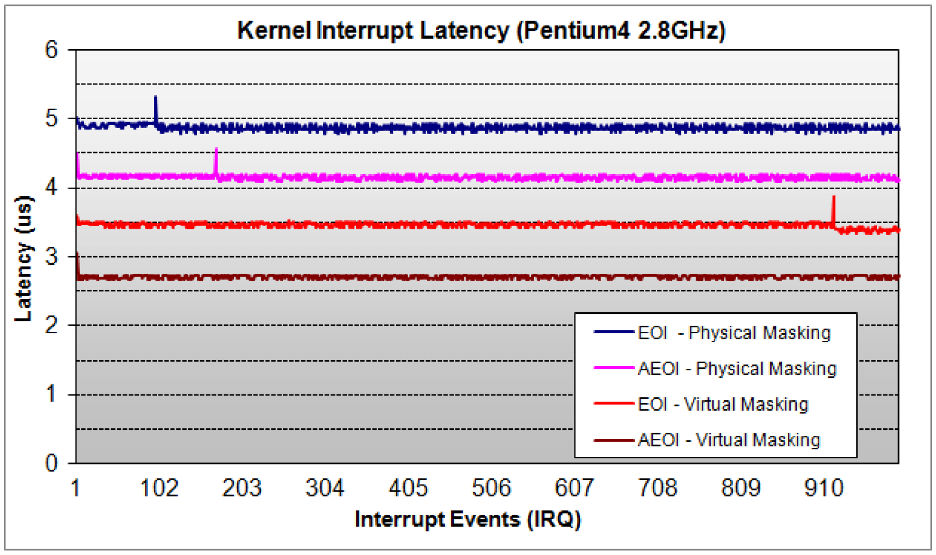

Fig. 11. Execution trace—kernel interrupt latency.

the periodic task (at 20,000 μs, 70,000 μs, and 120,000 μs). This is illustrated by the activations of the corresponding HAT (where the corresponding LLIH traces are not executed).

In this case, it is important to note that only one undesired interrupt per activation of t_2 is possible. Again, for each period of t_2, only four IRQs are accepted and handled (instead of the five IRQs that were issued by the serial port). On each execution of t_2, one IRQ is ignored. This restriction guarantees a small bound in the disturbance due to undesired interrupts, as denoted by Equation (17).

9.2 Overhead Measurement

In this section, we analyze the implementation overhead of the integrated interrupt and task model. In the integrated model, the *interrupt latency* includes the time for setting up the priority in the interrupt controller, the time for signalling the interrupt synchronization object (i.e., semaphore), and the context switch time from the interrupted task to the HAT (these timings are not included in the traditional model). Therefore, the interrupt latency of the HAT is the main overhead indicator.

In our measurements, we used a task with an endless loop that logged a trace with an identification code and a timestamp. We also used a HAT associated with the IRQ, logging a trace with another identification code and the current timestamp. The values were collected for the four combinations of the two emulation modes (physical or virtual masking) and the two EOI modes (explicit or automatic). Figure 11 plots 1,000 latency samples.[1]

In all modes, the behavior of the interrupt latency is practically stable around the average latency: 4.869 μs (for EOI with physical masking), 4.158 μs (for AEOI with physical masking), 3.469 μs (for EOI with virtual masking), and 2.716 μs (for AEOI with virtual masking). This stability is a very important factor for real-time systems. Another observation is that the average values for the virtual masking modes are

[1]We believe that the spikes in the plots are due to speculative execution and/or cache misses.

Table II. Interrupt Latency Figures and Port Operations

Emulation Mode		I/0 Write			Interrupt Latency (µs)		
EOI	Mask	EOI	IMRE	#	Min	Ave	Max
Explicit	Physical	1	2^1	**3**	4.781 μs	4.869 μs	5.318 μs
Automatic	Physical	0	2^1	**2**	4.097 μs	4.158 μs	4.564 μs
Explicit	Virtual	1	0^2	**1**	3.340 μs	3.469 μs	3.877 μs
Automatic	Virtual	0	0^2	**0**	2.672 μs	2.716 μs	3.060 μs

Notes: 1. It is assumed that it has been necessary to set the IMRE in both 8259. Often it is only necessary to establish a single IMRE. However, this measurement was done with an implementation that always sets the IMRE in both 8259 (worst case).

2. A desired IRQ is assumed. In case of an undesired one, the LLIH writes both masks. However, this is not included into the interrupt latency because the ISR is not invoked.

72.9% and 67% of the corresponding values for the physical masking, representing a significant reduction in the overhead, making the integrated model suitable even for systems with low overhead requirements. The two virtual masking modes yield better results than the two physical masking modes. The automatic EOI mode with virtual masking shows the best performance with very low worst case interrupt latency (3.06 μs).

It is interesting to analyze how the results for the interrupt latency for each emulation mode are related to the number of writing operations to the input/output port for that mode. This relationship is shown in Table II, where it is easy to identify that the number of port accesses is one of the dominant factors for the kernel interrupt latency.

9.3 Evaluation of the Experimental Results

We will compare the overhead of the port access between the integrated and the traditional model as final comments of our experiments.

(1) In the case of the traditional scheme, the interrupt latency is not affected by the overhead of any port access. Nevertheless, before leaving the ISR, it is incurred in this overhead due to the need to issue an EOI command. Consequently, the port access overhead is indeed reflected in the disturbance (or interference), causing a decrease in the utilization bound.
(2) In the case of the integrated model using the automatic EOI mode, the port operations are included in the interrupt latency (due to the need of setting the IMRE). Nevertheless, the EOI writing is eliminated completely. Consequently, since the port writing is one of the dominant factors in the execution time, we expect the overhead introduced by our scheme to be of the same order as that of the traditional interrupt scheme.
(3) If the automatic EOI mode is used in combination with the virtual masking, then (for desired interrupts) there are no port writings neither at the entry or at the exit of the HAT. Therefore, due to the need to use explicit EOI in the traditional interrupt model, we expect the overhead introduced by the integrated model indeed to be lower than that of the traditional model.

In summary, as demonstrated by Equation (17) and the experimental results, the implementation of the integrated model using the AEOI and virtual masking emulation mode allows an interrupt management scheme to be completely predictable and without overhead (for desirable interrupts). Also, the decrease in the complexity of this integrated design favors the development of reliable systems. Consequently, in real-time system kernels in which timely response to events and reliability are determining factors, the integrated model offers significant benefits.

10. RELATED WORK

Several research works propose alternatives for avoiding the difficulties of the traditional interrupt model in real-time applications. Stewart [1999] considers the indiscriminate use of ISRs one of the most common errors in real-time programming. Several real-time operating systems (e.g., MARS [Kopetz et al. 1989]) have adopted radical solutions where all external interrupts are disabled, except for those that come from the timer and propose to treat all peripherals by polling. Although this solution completely avoids the non-determinism associated with interrupts, it has the disadvantage of low efficiency in the usage of the CPU due to the busy wait in I/O operations. The advantage of our integrated scheme with respect to these proposals is that it achieves temporal determinism without significantly affecting the usage of the CPU.

Several scheduling analysis have been proposed to consider the interrupts as the activities with the top priorities in the system [Jeffay and Stone 1993; Lewandowski et al. 2007]. In Stewart and Arora [2003], the exact schedulability equation is extended to include the overhead of the interrupts in systems with static priorities, and it extended the model introduced in Lehoczky et al. [1989] to include the overhead of interrupt handling. The resulting equation evaluates the trade-offs of performing the interrupt handling inside an ISR versus postponing most of the treatment to a sporadic server [Sprunt 1990].

Several strategies have been proposed for obtaining some degree of integration among the different types of asynchronous activities. Hills [1993] proposes a "structured" interrupts treatment scheme at the task level, introducing an interface independent from the synchronization mechanism which does not consider interrupts with dynamic priorities. Kleiman and Eykholt [1995] treat interrupts as threads, but their proposal does not have the goal of achieving temporal determinism but the goal of increasing the scalability of the system in multiprocessor architectures oriented to servers' operating systems. Consequently, the interrupt threads use a separate (not unified to tasks) ranking of priority levels.

Recently, there has been some research work on interrupt handling in the context of Linux for real-time [Regnier et al. 2008]. Abeni et al. [2009] present a reservation-based approach using interrupts as threads. Other proposed schemes [Lee et al. 2010; Zhang 2009; Zhang and West 2006] also schedule interrupts as threads. Liu et al. [2010] propose a hardware scheme for supporting the combination of real-time and non-real-time interrupts. Parmer and West [2008] implement a two-level interrupts handling scheme in which the handling is deferred to *upcalls* in the user space. The upcalls are scheduled with a hierarchic and configurable scheme. Facchinetti et al. [2005] propose a scheme for the use of Linux device drivers in a real-time kernel. This scheme reserves CPU bandwidth for the execution of the ISRs without preemption.

The second generation of micro-kernels usually handles interrupts as threads with the purpose of supporting user-level device drivers. For instance, L4 handles interrupts using a so-called *interrupt handler task* (IHT) but without abstracting the interrupt hardware [Liedtke 1996]. More recently, in the process of porting L4 to architectures other than the IA32 and with the purpose of making the IHT independent of the interrupt hardware, a new L4 interrupt architecture was introduced [Dannowski et al. 2001]. An interesting side effect of this new interrupt architecture is that (in spite of the fact that the L4 micro-kernel was not designed for real-time [Ruocco 2006]) it now yields a bounded interrupt disturbance. This approach, that we call *masking/unmasking protocol*, is also used in the genirq abstraction of the recent Linux RT patch [Rostedt and Hard 2007].

With the masking/unmasking protocol, the kernel also sets a common LLIH for all IRQs. However, now the LLIH does an immediate EOI followed by the masking of the IRQ, avoiding additional occurrences of the same IRQ until it is explicitly unmasked. Then the LLIH sets ready an IHT which is scheduled according to its software priority in the same way as the rest of the tasks. When the service is finished, the IHT has the duty of explicitly unmasking the related IRQ by using an acknowledgment service provided by the kernel.

The IRQ acknowledgment (unmasking) in the IHT works as a request for an additional IRQ that only occurs as a reply to it. In consequence, this approach gets a behavior similar to that of the integrated model with virtual masking, using case (1) of masking. In the worst case, only one instance of the IRQs associated with IHTs with lower priorities than that of task t_i (that would be undesired IRQ in virtual masking) may disturb its execution. With this technique, the priority inversion, which is caused by interrupt disturbance, gets bounded due to interrupt disturbance. However, in this case, there is an additional overhead in the interrupt latency of all IRQs due to the EOI and mask-writing operation. In addition, the second mask writing is also added to the execution time of each IRQ.

To quantify this overhead, let δ^{EOI} be the execution time for issuing the EOI to the PIC; then Equation (9) can be written as

$$U_i^P = \frac{C_i + |S(i) \cup L(i)| \left(\gamma + 2\delta^M + \delta^{EOI}\right)}{T_i} + \tag{18}$$

$$\sum_{j \in (P(i)-H(i))} \frac{C_j}{T_j} + \sum_{j \in H(i)} \frac{c_i^H + 2\delta^p + \gamma + 2\delta^M + \delta^{EOI}}{T_J^H}. \tag{19}$$

Hence, the decrease in the utilization $U_{loss} = U_i^{PI*}$ due to the overhead of the masking/unmasking protocol is

$$U_i^{PI*} = \frac{|S(i) \cup L(i)| \left(\gamma + 2\delta^M + \delta^{EOI}\right)}{T_i} + \sum_{j \in H(i)} \frac{\gamma + 2\delta^p + 2\delta^M + \delta^{EOI}}{T_J^H}. \tag{20}$$

As can be noted, by comparing Equations (15) and (20), aside from the increment in the interrupt latency due to the masking of the IRQ in the LLIH, the introduced utilization loss is higher than the worst case overhead of the less optimized case of the integrated model with virtual masking.

The outcome of this comparative analysis is that the integrated model with virtual masking not only is predictable but also introduces lower overhead than that of interrupt architectures of related kernels.

Figure 12 contrasts the integrated model and the different variants of the traditional model for management of tasks and interrupts. The leftmost column of the figure shows the elements involved in the interrupt handling. The upper part of this column shows the I/O device that issues the interrupt request. These interrupt requests arrive at the hardware interrupt controller or PIC which schedules them according to their hardware priorities. The bottom part of this column shows the real-time scheduler that handles predictable software events. Above the realtime scheduler, we depict the scheduling of software entities with priorities higher than those of the real-time scheduler. The activation of these high priority software entities, the arriving of interrupt requests, and its scheduling by the PIC are all outside the control of the real-time scheduler, and hence, these are unpredictable events.

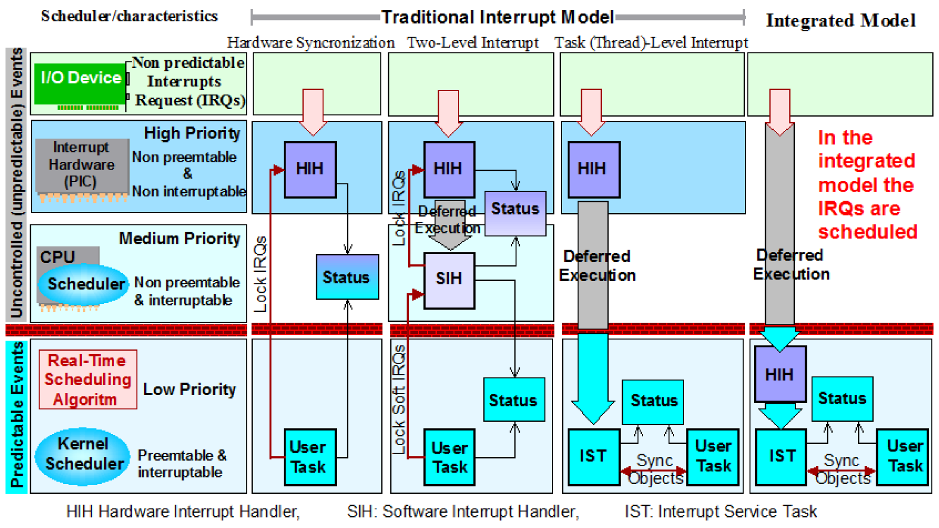
Scheduler/characteristics
Traditional Interrupt Model
Integrated Model
Hardware Syncronization
Two-Level Interrupt
Task (Thread)-Level Interrupt
Uncontrolled (unpredictable) Events
Predictable Events
I/O Device
Non predictable Interrupts Request (IRQs)
Interrupt Hardware (PIC)
High Priority
Non preemtable & Non interruptable
CPU
Scheduler
Medium Priority
Non preemtable & interruptable
Real-Time Scheduling Algoritm
Low Priority
Kernel Scheduler
Preemtable & interruptable
HIH
Status
User Task
Lock IRQs
Deferred Execution
SIH
Lock Soft IRQs
IST
Sync Objects
In the integrated model the IRQs are scheduled
HIH Hardware Interrupt Handler,
SIH: Software Interrupt Handler,
IST: Interrupt Service Task

Fig. 12. Integrated model vs. “traditional” models.

The other four columns of the figure show the different executable entities in the system for four different interrupt management schemes. The rightmost column represents our integrated model [Leyva-del-Foyo et al. 2006b, 2006c]. The other three middle columns represent different classes of interrupt model. In the first class (hardware synchronization), all interrupt handling is done in a hardware interrupt handler (HIH) under the control of the PIC. The synchronization between HIHs and user tasks is achieved by disabling or by locking interrupts. The two-level interrupt handling scheme splits the handling between a first-level HIH and a second-level software interrupt handler (SIH). The scheduler of these SIHs has higher priority than the real-time scheduler. In the task (thread)-level interrupt handling scheme, there is a short HIH outside the control of the real-time scheduler that activates an interrupt service task (IST) and performs the bulk of the interrupt handling.

The activities that are executed under the control of the real-time scheduler are called schedulable entities, and they all constitute the domain of predictability of the system. Ideally, in order to provide temporal guarantees, the real-time scheduler must schedule all activities in the system.

All activities executed with priorities higher than the real-time scheduler disrupt the real-time scheduling: they are non-schedulable entities and conform the domain of unpredictability of the system. In this context, Figure 12 shows a qualitative comparison between the integrated model and the other models. In the integrated model, the interrupt subsystem is free of non-schedulable entities. This is possible because, in contrast with the previous schemes (where the IRQs are not under the control of the real-time scheduler), in the integrated model, the real-time scheduler schedules all the IRQs.

Table III contrasts the integrated model and the existing alternatives for traditional interrupt handling and the interrupt avoidance (or event handling by pooling). It shows how the integrated model achieves the advantages of the last two approaches. In other words, with the integrated model, it is possible to combine the characteristics of low latency and low overhead (which are natural for an event handling by interrupt) with the characteristics of temporal predictability and feasibility for controlling overload (natural for an event handling by pooling).

Table III. Characteristic of the Integrated Model vs. Alternatives

Characteristic	**"Traditional" model of ISRs and Tasks**	**Integrated Model of Tasks and IRQs**	**interrupts avoidance**
Temporal Predictability	Impaired due to the existence of ISR (non schedulable entities).	Reinforced due to the avoidance of ISRs (non schedulable entities).	Reinforced due to the avoidance of ISRs (non schedulable entities).
Event response latency	Low latency (high responsiveness). Determined by the interrupt latency.	Low latency (high responsiveness). Determined by the interrupt latency.	Non optimum (low responsiveness). Determined by the sampling period.
Overhead	Low: determined by the context switch. Proportional to the workload.	Low: determined by the context switch. Proportional to the workload.	Appreciable, determined by the polling Task: $\frac{\text{WCET of sampling}}{\text{sampling period}}$.
Overload Control	Bad: out of the system control.	Good: controlled by the scheduler.	Good: any overload is ignored.

The integrated model proposed can be implemented in many ways other than the software-based approach presented. Indeed, in Leyva-del-Foyo and Mejia-Alvarez [2004], we advocated a hardware implementation using a *custom programmable interrupts controller* (CPIC) that cooperates with the CPU in the scheduling of IRQs and Tasks (and could be implemented with FPGAs). After the publication of our integrated model, other research has experimented with some approaches of this hardware implementation [Leyva-del-Foyo and Mejia-Alvarez 2004]. Scheler et al. [2009] present a variant of this CPIC using the peripheral control processor (PCP) of the TriCore platform. Hofer et al. [2009] present another hardware implementation of the integrated model using an ordinary hardware interrupt controller to schedule the threads as interrupts. The last approach limits the scheduling algorithm to fixed priority with the number of levels supported by the interrupt hardware. Both Scheler et al. [2009] and Hofer et al. [2009] implement a restricted task and synchronization model that is useful for low-end embedded systems. The performance results presented in Scheler et al. [2009] show that the software implementation of the integrated model indeed has lower overhead than the hardware implementation.

11. CONCLUSIONS

In this article, we analyzed the problems created by the traditional interrupt model in real-time systems and presented our solution to these problems. We highlighted that having separate scheduling and priority schemes for interrupts and tasks create multiple mutual interference problems, as discussed in Section 2. These problems have created the need for multiple workarounds that have been used in practice over the years that are inefficient and difficult to use, understand, and quantify.

To solve these problems, we presented our integrated scheduling model of interrupts and tasks that solves each of the problems discussed in Section 2. Furthermore, because the synchronization of hardware priority to software priorities can be costly, we also developed optimizations to minimize this cost. A scheduling analysis was also presented to show the improvements in the response time and utilization with respect to the traditional model. Finally, we presented our experimental evaluation that highlights the difference in determinism and interrupt latency between our model and the traditional one.

Our analysis and experiments show that the implementation of our model offers a platform that does not need workarounds to synchronize tasks and interrupts, avoiding their timing penalties without incurring any additional cost. At the same time, the programming model is simplified and easier to understand and its use leads to more-robust systems. As a result, we believe that an integrated scheduling model has the potential of becoming the defacto standard in operating systems in the future.

REFERENCES

ABENI, L., MANICA, N., AND PALOPOLI, L. 2009. Reservation-based scheduling for IRQ threads. In *Proceedings of the 11th Real-Time Linux Workshop*. 179–186.

CARLSSON M., ENGBLOM, J., ERMEDAHL, A., LINDBLAD, J., AND LISPER, B. 2002. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *Proceedings of the 2nd International Workshop on Real-Time Tools*.

DANNOWSKI, U., SKOGLUND, E., AND UHLIG, V. 2001. Interrupt handling. In *Proceedings of the 2nd Workshop on Microkernel-based Systems*.

FACCHINETTI, T., BUTTAZZO, G., MARINONI, M., AND GUIDI, G. 2005. Non-preemptive interrupt scheduling for safe reuse of legacy drivers. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*.

HILLS, T. 1993. Structured interrupts. *Oper. Syst. Rev. 27*, 1, 51–68.

HOFER, W., LOHMANN, D., SCHELER, F., AND SCHRÖDER-PREIKSCHAT, W. 2009. Sloth: Threads as interrupts. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS'09)*. 204–213.

JEFFAY, K. AND STONE, D. L. 1993. Accounting for interrupt handling cost in dynamic priority task systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. 212–221.

JOSEPH, M. AND PANDYA, P. 1986. Finding response times in real-time systems. *Comput. J. 29*, 5, 390–395.

KLEIMAN, S. AND EYKHOLT, J. 1995. Interrupts as threads. *ACM SIGOPS Oper. Syst. Rev. 21*, 2, 21–26.

KLEIN, M. H., RALYA, T., POLLACK, B., OBENZA, R., AND HARBOUR, M. G. 1989. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, Norwell, M.A.

KOPETZ, H., DAMM, A., KOZA, C., MULAZZANI, M., SCHWABI, W., SEUFT, C., AND ZAINLINGER, R. 1989. Distributed fault-tolerant Real-time systems: The MARS approach. *IEEE Micro 9*, 1, 25–40.

LEE, M., LEE, J., SHYSHKALOV, A., SEO, J., HONG, I., AND SHIN, I. 2010. On interrupt scheduling based on process priority for predictable real-time behavior. *ACM SIGBED Rev. 7*, 1.

LEHOCZKY, J., SHA, L., AND DING, Y. 1989. The rate monotonic scheduling algorithm: exact characterization and average case behaviour. In *Proceedings of the IEEE Real-Time Systems Symposium*. 166–171.

LEWANDOWSKI, M., STANOVICH, M. J., BAKER, T. P., GOPALAN, K., AND WANG, A. 2007. Modelling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. 57–68.

LEYVA-DEL-FOYO, L. E. AND MEJIA-ALVAREZ, P. 2004. Custom interrupt management for real-time and embedded system kernels. In *Proceedings of the Embedded Real-Time Systems Implementation Workshop at the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*.

LEYVA-DEL-FOYO, L. E., MEJIA-ALVAREZ, P., AND DE NIZ, D. 2006a. Abnormal events handling for dependable embedded systems. In *Proceedings of the 7th Mexican International Conference on Computer Science (ENC'06)*. 81–91.

LEYVA-DEL-FOYO, L. E., MEJIA-ALVAREZ, P., AND DE NIZ, D. 2006b. Predictable interrupt scheduling with low overhead for real time kernels. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 385–394.

LEYVA-DEL-FOYO, L. E., MEJIA-ALVAREZ, P., AND DE NIZ, D. 2006c. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. 14–23.

LIEDTKE, J. 1996. Towards real microkernels. *Commun. ACM 39*, 9, 70–77.

LIU, M., LIU, D., WANG, Y., WANG, M., AND SHAO, Z. 2010. On improving real-time interrupt latencies of hybrid operating systems with two-level hardware interrupts. *IEEE Trans. Comput.* Forthcoming.

PARMER, G. AND WEST, R. 2008. Predictable interrupt management and schedulability in the composite component-based systems. In *Proceedings of the 29th Real-Time Systems Symposium*. 232–243.

REGEHR, J. 2008. Safe and structured use of interrupts in real-time and embedded software. In *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y.-T. Leung, and S. H. Son Eds., Chapman & Hall/CRC.

REGNIER, P., LIMA, G., AND BARRETO, L. 2008. Evaluation of interrupt handling timeliness in real-time linux operating systems. *ACM SIGOPS Oper. Syst. Rev. 42*, 6, 52–63.

ROSTEDT, S. AND HARD, D. V. 2007. Internals of the RT patch. In *Proceedings of the Linux Symposium*. 161–172.

RUOCCO, S. 2006. Real-time programming and L4 microkernels. In *Proceedings of the Workshop on Operating System Platforms for Embedded Real-Time Applications*.

SCHELER, F., HOFER, W., OECHSLEIN, B., PFISTER, R., SCHRÖDER-PREIKSCHAT, W., AND LOHMANN, D. 2009. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. *The 2009 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'09)*. Grenoble, France, October 2009, 167–174.

SPRUNT, B. 1990. Aperiodic task scheduling for real-time systems. Ph.D. dissertation, Carnegie Mellon University, Pittsburg, PA.

STEWART, D. B. 1999. Twenty-five-most commons mistakes with real-time software development. In *Proceedings of Embedded Systems Conference*.

STEWART, D. B. AND ARORA, G. 2003. A tool for analyzing and fine tuning the real-time properties of an embedded system. *Trans. Softw. Eng. 29*, 4, 311–326.

STODOLSKY, D., CHEN, J. B., AND BERSHAD, B. N. 1993. Fast interrupt priority management in operating system kernels. In *Proceedings of the USENIX Symposium on Micro-Kernels and Other Kernel Architectures*. 105–110.

THEODORE, P. B. 1990. Protected records, time management and distribution. *ACM SIGAda Lett. X*, 9, 17–28.

TINDELL, K. W. 1999. RTOS interrupt handling: Common errors and how to avoid them. *Embed. Syst. Prog. Eur.*

ZHANG, Y. 2009. Prediction-based interrupt scheduling. In *Proceedings of the 30th IEEE International Real-Time Systems Symposium, Work-in-Progress Session*.

ZHANG, Y. AND WEST, R. 2006. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*. 191–201.

Received May 2009; revised July 2010; accepted October 2010