Integrated Schedulers for a Predictable Interrupt Management on Real-Time Kernels

S. Sáez and A. Crespo

Department of Computer Engineering
Universidad Politécnica de Valencia
{ssaez,alfons}@disca.upv.es

Abstract. To analyse the timeliness behaviour of a real-time system is one its key aspects. A big effort has been performed by the real-time community to develop accurate and more general schedulability analysis that can ensure the correct execution of the system. However, few works have analysed the side effects introduced by the scheduler and undesired execution of *Interrupt Service Routines*. Previous works addressed the interrupt interference by proposing an *Integrated Interrupt Model* that avoids unpredictable disturbance from external interrupts. Even so, the scheduling overhead due to the unnecessary activation of low priority tasks remains still unaddressed in this model. This work proposes a new *Virtual* implementation of an *Integrated Interrupt Event-Driven Scheduler* that copes with this pending issue. It also analyses the behaviour of the commonly used dual queue scheme under this kind of schedulers and proposes a more appropriated data structure to avoid unnecessary overheads.

1 Introduction

Real-time computing systems cope with its intrinsic complexity by decomposing the system software in a set of concurrent tasks with timing constraints. These timings constraints, that must be met for correct operation, are usually represented by a deadline and a task period. To guarantee such constraints, extensive research has been performed on schedulability analysis of real-time systems [1]. Schedulability tests are designed to take into account the system workload characteristics and the kind of scheduler used by the real-time operating system. Fixed-priority scheduler is one of the most popular and widely accepted real-time schedulers, and therefore it is present in almost all the commercial real-time operating systems [2]. However, a wide gap still exists between scheduling theory and its implementation in operating system kernels, since the system scheduler is usually assumed to be executed without any kind of overhead.

Few works have analysed the side effects introduced by the scheduler and the associated operating systems routines [3–5]. These works analyse the temporal behaviour of real-time schedulers classifying them into *event-driven*, that relies on an external hardware device that generates interrupts upon task arrivals, or *timer-driven*, that uses periodic interrupts from a programmable hardware timer

to execute the scheduler at fixed intervals. Katcher *et al.* emphasize the importance of integrating timer interrupts into the scheduling scheme in order to avoid unnecessary interrupts, i.e., timer interrupt that will wake up a low priority task while a higher priority task is in execution. This *integrated interrupt model* has been further extended by Leyva-del-Foyo *et al.* in [6–8] to include all hardware interrupts that can be produced in a real-time kernel. These authors introduce an integrated model for task and interrupt management and propose different hardware and software implementation approaches. This integrated model is carefully analysed under each proposed implementation and the utilization bound reductions compared with the one obtained from a non-integrated model.

The integration of hardware interrupts and real-time tasks gives rise to a real-time task set composed by *Hardware Activated Tasks* and *Software Activated Tasks* respectively [8]. The release of Harware Activated Task is strictly controlled by an *Interrupt Hardware Abstraction Layer* in [7] or more loosely controlled using an optimistic approach in [8]. However, the Software Activated Tasks are normally released by means of a timer interrupt, and this fact is not addressed in these works. For example, on [7] is suggested that the timer interrupt still could be considered an Interrupt Service Routine, since it will never be handled by the application. This approach, although reduces the unnecessary overhead introduced by the Interrupt Hardware Abstraction Layer, could give rise to an unnecessary invocations of the system scheduler when the task that is woken up has a lower priority than the currently executing task. An integrated event-driven scheduler, as proposed by Katcher *et al.*, could cope with this scenario whenever the *Hardware Interrupt Controller* was able to manage multiple prioritized timers. However, this capability is usually not available in the hardware interrupt controller integrated in nowadays processors.

This work proposes to complete the integrated interrupt model with an integrated event-driven scheduler that modifies the hardware timer to interrupt only when a higher priority task has to be woken up. This kind of scheduler could avoid unnecessary task activations, reducing the disturbance introduced by the operating system kernel. The adequacy of traditional scheduling implementations based on a dual queue scheme, a ready-queue and a wait-queue, to implement this new kind of scheduler is also addressed. Finally, a new fixed-priority scheduler based on a Cartesian tree [9] is proposed to overcome the detected drawbacks of the traditional dual queue approach.

The rest of the paper is organized as follows: next section describes the system model and the notation used in the rest of the paper. Section 3 introduces the model of integrated interrupt management. Then section 4 presents how the scheduler can be integrated in this new model. Section 5 presents dual-queue scheduling schemes, its computational cost and main drawbacks when used in an integrated interrupt model. Section 6 presents a new scheduler that avoids previous drawbacks. Finally section 7 presents some additional considerations and then section 8 summarise some of the main results.

2 System Model and Notation

The notation used in this work is based on the one presented in [7] but with some small differences. It can be summarized as follows:

HAT *Hardware Activated Task.* A task that is released by an external interrupt.
SAT *Software Activated Task.* A task that is activated by a timer interrupt or by other task.[1]
τ_x Real-Time task with period T_x, worst-case execution time C_x and priority p_x.
r_x Next activation instant of task τ_x while it is in the wait queue.
$H(i)$ Set of tasks with a priority higher than task τ_i. $H_{\mathrm{SAT}}(i)$ are the SATs that belongs to $H(i)$. A similar definition holds for $H_{HAT}(i)$.
$H_{\mathrm{SAT}}(i) \bigcap H_{HAT}(i) = \emptyset$.
$L(i)$ Set of tasks with a priority lower than or equal to task t_i. $L_{\mathrm{SAT}}(i)$ and $L_{HAT}(i)$ are defined as in $H(i)$.
R **and** W Represents the tasks present in the *ready queue* or *wait queue* respectively. The number of tasks in these queues will be represented as N^R and N^W. Intersection with task sets $L(i)$ and $H(i)$ will be denoted by using superscripts, e.g., $H^R(i) = R \bigcap H(i)$.

The notation for system overheads is shown next:

δ^{isr} Total processing time for the enter and leave code of an *Interrupt Service Routine*, including basic communication with the *Hardware Interrupt Controller* (interrupt ack, end-of-interrupt, etc.).
δ^{hic} Communication cost with the *Hardware Interrupt Controller* to mask undesired external interrupts.
δ^{ctx} Time required to perform a task context switch, including CPU registers context, MMU context (if required), etc.
δ^{Sched} Processing time required to determine the next task to be executed. It can be differentiated in $\delta^{\mathrm{Sched-A}}$, when it is computed during a task activation, or $\delta^{\mathrm{Sched-T}}$, if it is computed when the active task is going to be suspended. It can also be suffixed as $\delta_T^{\mathrm{Sched}}$ or $\delta_I^{\mathrm{Sched}}$ to differentiate between *traditional* schedulers and *integrated interrupt* schedulers proposed in this work.

3 Integrated Interrupt Management Model

An extended explanation of the *Integrated Interrupt Model* is presented in this section along with the issues that still remains unaddressed. A complete explanation of this model can be found in [7, 8].

The Integrated Interrupt Model proposes to use a common priority space where real-time tasks and traditional Interrupt Service Routines (ISR) are mapped. This gives rise to a set of *Hardware Activated Tasks* (HAT) and *Software*

[1] However, this work only considers the software activated task released by a timer interrupt.

Activated Tasks (SAT) which timely execution can be analysed using well known methods [10, 1]. This model avoids to perform a probably unbounded number of ISR executions while high priority tasks are in execution, but also introduces additional system overheads.

3.1 Hardware Activated Tasks

The conversion of the existing ISRs into HATs allows the system to control when a given external interrupt is attended with respect to traditional real-time tasks, i.e. SATs, and the rest of HATs by means of a correct priority assignment. The disturbance associated to the execution of ISRs, that previously affect to all priority levels due to the independence of priority spaces, is moved to a given priority level in the common priority space where higher priority tasks will not be affected. The reduction of CPU utilization due to the interrupt disturbance for a given task τ_i is denoted in [7] as U_{iS}. That work also concludes that the time required to process the enter and leave code of the ISR, δ^{isr}, is replaced by a usually larger time required to perform the context switches of the implied HAT, δ^{ctx}. Whether the utilization reduction due to this new overhead is lower than the previous lost of utilization U_{iS} or not, allows the system designer to determine the adequacy of the integrated interrupt model for a given real-time system.

This integrated priority space for tasks and interrupts requires to change the interrupt priority level[2] of the *Hardware Interrupt Controller* (HIC) each time a new task enters into or exits from execution to avoid the undesired activation of HATs. If the HIC is not included inside the CPU encapsulation, then the communication with this external device through the input/output subsystem could carried out an important overhead that also has to be incorporated into the schedulability tests. Moreover, if the HIC does not support an interrupt priority level, a *Virtual Custom Programmable Interrupt Controller* can be incorporated to *Interrupt Hardware Abstraction Layer* to offer this functionality. As this Virtual Custom PIC emulates the availability of the interrupt priority level by means of *interrupt masking* on the real HIC, additional overheads are introduced to compute the adequate mask for each priority. These additional operations performed during task activation and deactivation are the ones that introduce a higher context switch time. A complete analysis of these overheads can be found in [7, 8].

3.2 Software Activated Tasks

Although the integrated model allows the system designer to control the activation of HATs by means of properly assigned priorities, the correct activation of SATs remains unaddressed. SATs are theoretically activated by software mechanisms such as semaphores, mutexes, barriers, execution time timers, timing events and so on. While the tasks activated by semaphores, mutexes, barriers

[2] This level specifies the minimum priority of an interrupt for the CPU to be notified

and suspension objects are released by code executed directly or indirectly by other task, in the case of temporal events, it is the ISR of the timer interrupt the final responsible of producing the timed event. In such a way, a subset of the SATs, in particular the real-time periodic tasks, are indirectly activated by the occurrence of a hardware interrupt. After this SAT activation, the scheduler is executed in order to determine if the recently activated task has a higher priority than the one is currently under execution. If the activated task has a lower priority, then the current task is resumed. However, an unnecessary activation procedure has been performed that has introduced an unexpected overhead during a higher task execution. This overhead is similar to the one produced by hardware interrupts in a non-integrated model, but with the additional overhead of executing the system scheduler.

The interference due to unnecessary SAT activations, U_i^{SAT}, is the decrease of least upper utilization bound at priority level p_i and can be computed as follows:

$$U_i^{\mathrm{SAT}} = \sum_{\tau_k \in L_{\mathrm{SAT}}(i)} \frac{\delta^{\mathrm{isr}} + \delta_T^{\mathrm{Sched-A}}}{T_k} \tag{1}$$

where $\delta_T^{\mathrm{Sched-A}}$ can be typically decomposed in the following actions: to remove the time event from the wait queue, $\delta_{wq}^{\mathrm{Sched-A}}$, to insert the newly activated task τ_k into the ready queue, $\delta_{rq}^{\mathrm{Sched-A}}$, and to check the top of the ready queue to find out which task will be the next running task, $\delta_{next}^{\mathrm{Sched-A}}$.

$$\delta_T^{\mathrm{Sched-A}} = \delta_{wq}^{\mathrm{Sched-A}} + \delta_{rq}^{\mathrm{Sched-A}} + \delta_{next}^{\mathrm{Sched-A}} \tag{2}$$

While $\delta_{next}^{\mathrm{Sched-A}}$ can be considered constant when a new SAT has been activated, the other two overhead terms, $\delta_{wq}^{\mathrm{Sched-A}}$ and $\delta_{rq}^{\mathrm{Sched-A}}$, depends on the data structures used to maintain the ready and wait queues. These overheads are analysed in section 5.

This scheduling overhead can also be expressed as an additional blocking time, that higher priority tasks can suffer from lower priority tasks, and it should be taken into account during the Response Time Analysis [1] when traditional interrupt models are used. The next equation summarises this additional blocking time:

$$B^{\mathrm{SAT}}(i) = |L_{\mathrm{SAT}}(i)| \times (\delta^{\mathrm{isr}} + \delta_T^{\mathrm{Sched-A}}) \tag{3}$$

In order to avoid this effect, the integrated interrupt model has to grant that no timer interrupt that activates a lower priority task ($\tau_k \in L_{\mathrm{SAT}}(i)$) will be raised during the execution of a higher priority task τ_i. This can be achieved using an *Integrated Interrupt Event-Driven Scheduling* scheme as it is proposed by Katcher *et al.* in [3]. Next section briefly introduces this kind of schedulers and proposes a software-based implementation.

4 Integrated Interrupt Event-Driven Schedulers

In *Integrated Interrupt Event-Driven Scheduling* systems all the tasks are initiated by external interrupts which have priorities that fully match the software task priorities. Upon the activation of each task τ_i an interrupt is posted to the processor that only starts the corresponding ISR if the priority p_i is higher than the priority of the currently running task τ_r, i.e., $\tau_i \in H(r)$. If task τ_i does not belong to $H(r)$, then the activation interrupt of task τ_i remains pending. This behaviour requires a special hardware within the processor or a HIC that holds the active task's priority in a register and raises a real interrupt only when the pending interrupt with the highest priority has a priority greater than the current priority level. A similar *Custom Programmable Interrupt Controller* (CPIC) is described in [6] with some differences: As all the tasks are considered HATs, the CPIC has to provide enough hardware timers, with its associated priority level, to implement all periodic real-time tasks in the system.

As the functionalities to be provided by the HIC are not commonly available in current processors or PICs, an approach similar to the *Virtual CPIC* presented in [7] is proposed next.

4.1 Virtual Integrated Interrupt Event-Driven Schedulers

This work proposes the use of a *Virtual Integrated Interrupt Event-Driven Scheduler* (VIIED Scheduler). Under this kind of scheduler, while the current task τ_r is running, the HIC is programmed to raise only timer interrupts belonging to activation instants of higher priority tasks, $\{r_h : \tau_h \in H_{\mathrm{SAT}}(r)\}$. The activation instants that belong to lower priority tasks, $\{r_l : \tau_l \in L_{\mathrm{SAT}}(r)\}$, are ignored, including the case in which an activation instant r_l is closer than the closest r_h. This behaviour prevents the scheduler to wake up a task with a lower priority than the current one. Thus, if a VIIED scheduler is incorporated to the integrated interrupt mode presented in [7, 8], a fully integrated interrupt model can be achieved and the system can avoid disturbances shown in sections 3.1 and 3.2.

To achieve this behaviour, each time the scheduler is invoked to determine the next ready task with the highest priority, τ_r, it has to examine the wait queue to find out which of the suspended tasks, τ_p, with a priority higher than p_r, has the closest activation instant. We call this task, τ_p, the *next preemptor* of τ_r and it can be expressed as:

$$\tau_p \in H^W(r) / \nexists \tau_q \in H^W(r) : r_q < r_p \tag{4}$$

Once τ_p is determined, the HIC has to be programmed to raise the next timer interrupt at time r_p. If $H^W(r)$ is empty then there is no task into the wait queue with a higher priority than the current one, and therefore, the timer interrupt does not need to be programmed. In such a case, the currently active task will run until its suspension, at which time the system scheduler is invoked again. This can only happen when the highest priority task is running.

Taking into account the new functionality that has to be implemented by the scheduler in this model, the new scheduling overheads $\delta_I^{\text{Sched}-\text{A}}$ and $\delta_I^{\text{Sched}-\text{T}}$ can be expressed as follows:

$$\delta_I^{\text{Sched}-\text{A}} = \delta_{wq}^{\text{Sched}-\text{A}} + \delta_{rq}^{\text{Sched}-\text{A}} + \delta_{next}^{\text{Sched}-\text{A}} + \delta_{np}^{\text{Sched}-\text{A}} \tag{5}$$

$$\delta_I^{\text{Sched}-\text{T}} = \delta_{rq}^{\text{Sched}-\text{T}} + \delta_{wq}^{\text{Sched}-\text{T}} + \delta_{next}^{\text{Sched}-\text{T}} + \delta_{np}^{\text{Sched}-\text{T}} \tag{6}$$

where δ_{np} is the time required to find the next preemptor on each case, and $\delta_{rq}^{\text{Sched}-\text{T}}$, $\delta_{wq}^{\text{Sched}-\text{T}}$ and $\delta_{next}^{\text{Sched}-\text{T}}$ are the execution times required to remove the active task from ready queue, to insert its next activation on the wait queue and to determine the next activate task among the ones found in the ready queue.

The efficiency of a VIIED scheduler will strongly depend on its ability to perform all the steps included in $\delta_I^{\text{Sched}-\text{A}}$ and $\delta_I^{\text{Sched}-\text{T}}$ in a fast and bounded manner. A lot of study has been carried out to analyse the behaviour and temporal costs of different kinds of priority queues, including under the real-time perspective [11]. Next section analyses common data structures used to implement the ready and wait queues on several real-time kernels. The main drawbacks that arise when they are used to implement a VIIED scheduler are also analysed.

5 Dual-Queue Scheduling Schemes

Today's real-time operating system kernels are usually based on a dual-queue scheduling scheme. This scheme uses one queue to store active tasks, called *ready queue*, and the other one to store timed events, called *wait queue*. Timed events stored in the wait queue usually has a reference to the suspended task that has to be woken up upon the arrival of such an event.

Several open source real-time kernels found in the bibliography have been analysed and the differences found among them in the scheduling scheme are mainly centred in the data structures used to implement these queues. While *PartiKle* [12] and *MarteOS* [13] use priority bitmaps plus an array of ordered linked lists to implement the ready queue and a heap to implement the wait queue, *Open Ravenscar Kernel* [14] and *Shark* [15] kernels implement both queues with ordered linked list. The only one that does not follow the dual-queue approach is RT-Linux [16] that uses a very simple scheme based on a single one unsorted queue.

5.1 Time complexity of queue operations

The ready queue operations usually required by a scheduler to implement a fixed-priority scheduling policy are: `insert`, `delete` or `delete-min`, and `find-min`. Among the structures used to implement the ready queue, the priority bitmap clearly outperforms any other data structure since it has a constant temporal cost ($\Theta(1)$) with respect to the number of ready tasks[3] in all the required operations.

[3] The temporal cost is $O(P)$, where P is the number of priority levels, but this value is fixed in a given system

The queue operation required to implement the wait queue in a conventional event-driven scheduler are the same than in the ready queue. As the normal key used for storing timed events are absolute activation instants, and they are not bounded, the priority bitmap cannot be used for wait queues. In these situations a typical data structure to be used is any binary tree that offers a good temporal behaviour on the required operations. As shown above, one of the preferred ones is the binary heap [17]. In this case, the temporal cost of `find-min` operation is $\Theta(1)$ and $\Theta(\log(n))$ for `insert` and `delete-min`. Other binary trees as AVL and Black-Red trees[17] can also be used, as they have similar temporal costs, although the constant that multiply any operation cost is usually higher than the one of binary heaps.

However, to implement a VIIED scheduler a new queue operation `find-preemptor` is required to be implemented in the wait queue, as it is shown in equation (4). This operation is not commonly available in used data structures, as it needs a second key to sort the wait queue nodes: the priority of the task to be activated. Since data structures used to implement wait queues only uses one key, the cost of finding the item that accomplishes the *next preemptor* condition has an asymptotic upper bound of $O(N^W)$, being N^W the number of timed events stored in the wait queue.

When a sorted linked list is used to implement the wait queue, the nodes can be traversed by increasing activation instant. Since the *next preemptor* condition holds for the closest timed event with an associated task priority higher than the current one, the `find-preemptor` operation can stop as soon as one node fulfils the *higher priority* condition. This gives rise to lower temporal bound of $\Omega(1)$ and an average-behaviour that depends on the distribution of timed events inserted in the wait queue. However, due to the temporal cost of the `insert` operation ($O(n)$), the linked lists are only useful for real-time system with small real-time task sets.

On the other hand, as the ordering of siblings in a binary heap is not specified by the heap property, no order about nodes can be assumed and no *in-order* traversal is possible. Then, the tight temporal cost of the operation `find-preemptor` is $\Theta(n)$. As some implementations of AVL trees can maintain an in-order linked list together with the binary tree, they could be a good substitution of binary heaps when used on VIIED schedulers with a large number of tasks.

From here on, the worst case execution time of a queue operation will be referred as Q^R_{oper} and Q^W_{oper} for the operation `oper` over the ready and wait queue respectively.

5.2 Dual-queue scheme drawbacks

As shown in the previous section, the temporal behaviour of the queue operations used by a fixed-priority real-time scheduler can be tightly bounded by using the appropriated data structures. Despite of the inability of the currently used data structures to efficiently address the `find-preemptor` operation, the major drawback of the dual-queue scheduling scheme is the use of separate queues

for ready and suspended tasks. This section describes the disadvantages derived from this dual-queue scheme.

Given a running task, τ_r, a VIIED scheduler prevents the activation of lower priority tasks by means of avoiding the unnecessary timer interrupts. However, when the *next preemptor* task τ_p is activated, the lower priority tasks which activation is pending still remain in the wait queue. Being τ_p the next preemptor of τ_r, this set of pending tasks $P_p(r)$ can defined as:

$$P_p(r) = \{\tau_k \in L^W(r)/r_k \leq r_p\} \tag{7}$$

Then the number of tasks, N_p, that have to be moved to the ready queue while activating τ_p is $N_p = |P_p(r) \cup \{\tau_p\}|$.

When evaluating the cost of removing these pending tasks, it has to be taken into account that some wait queue implementations are not able to perform a `find-min` operation without performing a set of `delete-min` operations to remove the previously activated tasks $P_p(r)$. This is the case of the commonly used *heap*. If the wait queue has this constraint, then the scheduling overhead $\delta_I^{\text{Sched}-\text{A}}$ is defined as:

$$\delta_I^{\text{Sched}-\text{A}} = N_p \times (Q^W_{\text{delete}-\text{min}} + Q^R_{\text{insert}}) + Q^R_{\text{find}-\text{min}} + Q^W_{\text{find}-\text{preemptor}} \tag{8}$$

Although the $Q^R_{\text{find}-\text{min}}$ can be considered constant, as it is known that the highest priority task is going to be τ_p, the term $N_p \times (Q^W_{\text{delete}-\text{min}} + Q^R_{\text{insert}})$, referred as *pending task processing time*, could give rise to an important execution time overhead during the activation of τ_p.

On the other hand, when the running task finishes, the system has to determine the next running task and the next preemptor. This implies to update the ready queue and wait queue, before determining the next running and preemptor tasks. This scheduling overhead is shown next:

$$\delta_I^{\text{Sched}-\text{T}} = Q^R_{\text{delete}-\text{min}} + Q^W_{\text{insert}} + Q^R_{\text{find}-\text{min}} + Q^W_{\text{find}-\text{preemptor}} \tag{9}$$

Additionally, due to the internal design of the scheduler, these scheduling overheads, $\delta_I^{\text{Sched}-\text{A}}$ and $\delta_I^{\text{Sched}-\text{T}}$, are in fact *blocking times*, since the system scheduler is commonly designed as an uninterruptible routine. Therefore, no higher priority tasks can be activated while the scheduler structures are being updated.

Schedulers implementation comparison In order to determine if this implementation of the integrated interrupt model is adequate for a given real-time system, the overhead introduced by a traditional interrupt model, δ_T, due to the tasks that have to be activated during the execution of τ_r has to be compared with the new δ_I.

When using a traditional scheduler, the maximum overhead at a priority level p_r, referred as $\delta_T(r)$, is:

$$\delta_T(r) = \delta^{\text{isr}} + 2 \times \delta^{\text{ctx}} + \delta_T^{\text{Sched}-\text{A}} + B^{\text{SAT}}(r) + \delta_T^{\text{Sched}-\text{T}} \tag{10}$$

On the other hand, from equations (8) and (9) the overhead at a priority level p_r when using a VIIED Scheduler, $\delta_I(r)$, can be summarised as:

$$\delta_I(r) = \delta^{\text{isr}} + 2 \times (\delta^{\text{ctx}} + \delta^{\text{hic}}) + \delta_I^{\text{Sched}-\text{A}} + \delta_I^{\text{Sched}-\text{T}} \tag{11}$$

Thus, the condition that has to be fulfilled by a VIIED scheduler to worth its use into an integrated interrupt model can expressed as:

$$\delta_I(r) \leq \delta_T(r), \forall \tau_r \tag{12}$$

although the condition could be applied only to the critical priority levels.

Taking into account that in the worst case scenario N_p is limited by the number of tasks with a priority lower than p_r, i.e., $N_p \leq |L_{\text{SAT}}(r)| + 1$, the inequality shown in (12) can be simplified as follows:

$$2 \times \delta^{\text{hic}} + 2 \times Q^W_{\text{find}-\text{preemptor}} \leq |L_{\text{SAT}}(r)| \times (\delta^{\text{isr}} + \delta^{\text{ctx}} + Q^R_{\text{find}-\text{min}}) \tag{13}$$

Therefore, as $Q^R_{\text{find}-\text{min}}$ has a constant execution time in an efficient ready queue implementation, the maximum temporal cost at a given priority level p_r for $Q^R_{\text{find}-\text{preemptor}}$ can be $O(|L_{\text{SAT}}(r)|)$. As currently used wait queue implementations do not use the priority to sort the timed events, the computational cost for finding the next preemptor is usually $O(N^W)$. In such cases, the system will need an additional data structure that are able to sort pending timed events in priority order to find the next preemptor with a tighter computational cost. A possible data structure to perform this task could be a priority-indexed array of sorted event queues, as it has $O(|L_{\text{SAT}}(r)|)$ for $Q^W_{\text{find}-\text{preemptor}}$.

Despite of the accomplishment of this inequality, the dependency of $\delta_I^{\text{Sched}-\text{A}}$ with the number of blocked task in $P_p(r)$ introduces an additional undesirable behaviour: the release jitter of task τ_p, defined as the interval between the expected activation r_p and the real activation $r_p + \delta^{\text{isr}} + \delta^{\text{ctx}} + \delta_I^{\text{Sched}-\text{A}}$, becomes significantly incremented.

These disadvantages are normally intrinsic to the dual-queue scheduling scheme. The next section presents a VIIED scheduling algorithm based on a single queue model that tries to avoid both problems: release jitter and pending tasks processing time.

6 A Scheduler based on Cartesian Trees

This section describes a VIIED scheduler that uses only one data structure to store ready and suspended tasks. This data structure is based on Cartesian trees and it is intended to reduce the temporal cost of finding the next preemptor, $Q_{\text{find}-\text{preemptor}}$, to constant time. It was already presented in a previous work to implement a real-time scheduler [18].

Cartesian trees were introduced and named by Vuillemin [9]. The name is derived from the Cartesian coordinate system for the plane. A Cartesian tree for a set of points has the sorted order of the points by their x-coordinates, and it has the heap property according to the y-coordinates of the points. In this work the x-coordinates will be the task priorities and the y-coordinates their activation instants. In such a way, the Cartesian tree becomes a heap structure that stores the closest activation instant in the root node. However, as each node also have a priority, the final tree has an interesting property: the left child of each node is the next task to be activated with a priority lower or equal than its parent node and the next task to be activated with a priority higher than the parent node is the right child. So, the *next preemptor* of a given task is directly located in its right child node, and therefore, to find the next preemptor of any task can be performed in constant time. We denote this usage of the Cartesian trees: *Scheduling Cartesian Tree* or SC-Tree.

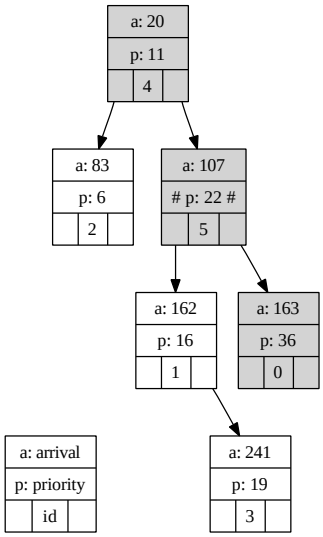

Fig. 1. A Scheduling Cartesian tree

Figure 1 shows an example of a SC-Tree with 6 tasks. Each task τ_i, is represented by three parameters: its identifier, i, its activation instant, r_i, and its priority, p_i. As it can be observed the activation instants of the nodes maintains no order but the heap property.

Let the node 4 represents the currently running task τ_4. All the suspended tasks with a higher priority, $H^W(4)$, can be found in the right sub-tree, and the lower priority ones, $L^W(4)$, in the left sub-tree. Moreover, the closest suspended task of $H^W(4)$, i.e., the next preemptor, can be found directly on the right child of node 4, i.e., the node 5. The right branch, depicted in grey in Figure

1, will be referenced as *preemptors branch*, and it contains the only tasks that can preempt the previous one in the branch during its execution. As it will be explained bellow, this preemptors branch has interesting properties.

Although this structure reduces $Q_{\text{find-preemptor}}$ to constant time, provided that a reference to the current task's node is available, this does no avoid the rest of drawbacks presented for the dual-queue scheduling schemes. So, the followed approach in this work is to use the SC-Tree not only as a heap of timed events with fast next preemptor operation, but also use the same SC-Tree as ready queue.

Let us follow the explanation with the depicted example of Figure 1. When task τ_5 becomes active at instant 107, no modifications will be performed on the SC-Tree, but just a change in the *currently running task* reference to point to node 5, that was the previous preemptor. At this time, the node 4 is the parent of currently running task τ_5. Task τ_4 is still active but it is not the highest priority task any more. Task τ_2 is also active, but it does not matter, since has a lower priority than τ_5 and τ_4. The next preemptor will be τ_0 with an activation instant $r_0 = 163$, which is the absolute time value used to program the next timer interrupt into the HIC. If the activation instant r_0 is reached before task τ_5 completes is execution, the task τ_5 will be preempted, and the scheduler would behave as in the later case, moving the currently running task reference to task τ_0. If task τ_5 finishes is execution before r_0 and ask for delaying its execution until its next period, the node 5 is modified with the new activation instant and pushed down until its new location in the SC-Tree. The new currently running task will be τ_4, i.e., the old parent of τ_5 that still remains active. The next preemptor would be τ_0 only if the new activation instant of τ_5 is not lower than r_0.

SC-Tree computational cost As it has been explained, the structure of the SC-Tree only changes when a task finishes its execution. When a task becomes active there are no actions to perform but to change the currently running task reference to the previous preemptor (right child), so the time required to determine the next active task $Q^{SA}_{\text{find-min}}$ has a constant computational cost ($\Theta(1)$) and also to determine the next preemptor $Q^{SA}_{\text{find-preemptor}}$ once the currently running task reference has been updated. In such a way, the scheduling overhead due to task activation $\delta_I^{\text{Sched-A}}$ will be:

$$\delta_I^{\text{Sched-A}} = Q^{SA}_{\text{find-min}} + Q^{SA}_{\text{find-preemptor}} \tag{14}$$

which are both constant time. This behaviour allows the system to minimize the release jitter of a task under this VIIED scheduler.

On the other hand, when a preempting task finishes, the task that must be resumed is the task located at the parent node, so $Q^{ST}_{\text{find-min}}$ has also a constant execution time. Therefore, only the time to push down the next activation of the finished task is significant in the preemption process. This new operation `push-down` for a given finished task τ_f has a worst-case execution time $Q^{ST}_{\text{push-down}}$ that depends on the number of pending activations between *now*

and r_f. In the worst case, the number of tasks will be N^W and, therefore, the cost will be $O(N^W)$.

SC-Tree blocking time Another important advantage of the SC-Tree is that the `push-down` operation is only executed when the task finishes its execution. If some higher priority task is activated during the `push-down` operation, the operation can be preempted and resumed when the higher priority tasks finishes. Although the SC-Tree would remain in an inconsistent state until the operation was resumed, the interesting property is that the *preemptors branch* is never modified by a lower priority task, and therefore, the inconsistent SC-Tree still can be used to determine the next preemptor of the running task until pending `push-down` operations are concluded.

7 Additional considerations

This section presents some additional considerations that have to be taken into account when a fully Integrated Interrupt Model is used in a real-time system.

In the traditional interrupt model, the scheduler cost of changing the priority of the running task, e.g. when it enters/leaves a protected object and the Immediate Priority Ceiling Protocol is used, is considered negligible or has a constant asymptotic cost.

When an Integrated Interrupt Model is used, to change the priority of the current task requires to update the system priority level. To avoid unnecessary interrupts when the system priority level changes, the HIC has to be reprogrammed and the next preemptor has to be computed when the task enters and leaves the protected object. The cost of these operations can be considered excessively high to be applied each time a task access to a protected object.

However, if the base priority is p_b and the real inherited priority, p_a, is not enforced by the VIIED Scheduler, any task τ_i with a priorities $p_i \in (p_b, p_a]$ will be activated but not executed. Additional blocking times have to be computed for these tasks similar to the one presented in equations (3) and (10).

$$B^{PO}(i) = \sum_{k \in L(i)-L(b)} \delta^{\text{isr}} + Q^W_{\text{delete}-\text{min}} + Q^R_{\text{insert}} + Q^R_{\text{find}-\text{min}} \tag{15}$$

This blocking time is equivalent to the one for the traditional interrupt model but only for the task between priorities p_b and p_i. This overhead has to be compared with the overhead due to update the system priority level when the tasks enters and leaves the protected object that is presented next:

$$Q^{PO} = \delta^{\text{hic}} + Q^R_{\text{insert}} + Q^R_{\text{delete}-\text{min}} + 2 \times Q^R_{\text{find}-\text{min}} + 2 \times Q^W_{\text{find}-\text{preemptor}} \tag{16}$$

where Q^R_{insert} represents the overhead of inserting a pseudo-task at the new priority level p_a and $Q^R_{\text{delete}-\text{min}}$ is the overhead to remove this pseudo-task at

the end of the protected action. In the case of the SC-Tree, Q^R_{insert} is replaced by $Q_{\text{push-down}}$.

To compute the cost of these operations for a given system scheduler allows the system designer to decide if it is worth following a fully integrated interrupt model or if it could be relaxed during the execution of protected actions.

8 Conclusions and Future Work

Previous works addressed the interrupt interference by proposing an *Integrated Interrupt Model* that avoids disturbance from external interrupts. However, the overhead of processing unnecessary activations of lower priority Software Activated Tasks has not been properly addressed in this model.

This work proposes a new *Virtual* implementation of an *Integrated Interrupt Event-Driven Scheduler* that avoids the hardware requirements of an Integrated Interrupt Event-Driven Scheduler. A new data structure based on Cartesian trees has been proposed to avoid the main drawbacks of implementing a Virtual Integrated Interrupt Event-Driven scheduler following a dual-queue scheduling approach. A comparison of the run-time behaviour of system schedulers used in an integrated model has been previously presented in [18].

The proposed SC-Tree scheduler has shown to be better suited for a fully integrated interrupt model, completely avoiding release jitter and scheduling blocking times that arise with conventional dual-queue schedulers.

Also a complete analysis of the implied overheads and blocking times when the integrated interrupt model includes Software Activated Tasks has been presented. This analysis allows the system designer to determine if the fully integrated interrupt management is suited for the real-time system under development.

Future work will try to extend the analysis to fully integrated interrupt management systems based on dynamic priorities and the suitability of the SC-Tree scheduler in such environments.

Acknowledgements

This work has been partially supported by the Spanish Government's projects COBAMI (DPI2011-28507-C02-02) and Hi-PartES (TIN2011-28567-C03-01-02-03) and the European Commission's MultiPARTES project (FP7-ICT-2011.3.4, Contract 287702).

References

1. Audsley, N., Burns, A., David, R., Tindell, K., Wellings, A.: Fixed priority preemptive scheduling: An historical prespective. Real-Time Systems **8**(2/3) (March/-May 1995) 173–189

2. POSIX.13: IEEE Std. 1003.13-1998. Information Technology -Standardized Application Environment Profile- POSIX Realtime Application Support (AEP). The Institute of Electrical and Electronics Engineers. (1998)
3. Katcher, D., Arakawa, H., Strosnider, J.: Engineering and analysis of fixed priority schedulers. IEEE Transactions on Software Engineering **19**(9) (September 1993) 920–934
4. Jeffay, K., Stone, D.L.: Accounting for interrupt handling costs in dynamic priority task systems. In: Proceedings of Real-Time Systems Symposium. (December 1993) 212–221
5. Burns, A., Tindell, K., Wellings, A.: Effective analysis for engineering real-time fixed priority schedulers. IEEE Transactions on Software Engineering **21**(5) (1995) 475–480
6. Leyva-Del-Foyo, L.E., Mejia-Alvarez, P.: Custom interrupt management for real-time and embedded system kernels. In: Proceedings of the Embedded Real-Time Systems Implementation (ERTSI 2004) Workshop 25th. (December 2004)
7. Leyva-Del-Foyo, L.E., Mejia-Alvarez, P., de Niz, D.: Predictable interrupt management for real time kernels over conventional PC hardware. In: RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, Washington, DC, USA, IEEE Computer Society (2006) 14–23
8. Leyva-Del-Foyo, L.E., Mejia-Alvarez, P., de Niz, D.: Predictable interrupt scheduling with low overhead for real-time kernels. Real-Time Computing Systems and Applications, International Workshop on **0** (2006) 385–394
9. Vuillemin, J.: A unifying look at data structures. Commun. ACM **23**(4) (1980) 229–239
10. Joseph, M., Pandya, P.: Finding response times in real-time systems. The Computer Journal **29**(5) (1986) 390–395
11. Mhatre, N.: A comparative performance analysis of real-time priority queues. Master's thesis, Florida State University (2001)
12. Peiro, S., Masmano, M., Ripoll, I., Crespo, A.: PaRTiKle OS, a replacement of the core of RTLinux. In: 9th Real-Time Linux Workshop. (2007)
13. Aldea-Rivas, M., Gonzalez-Harbour, M.: MaRTE OS: An ada kernel for real-time embedded applications. In: Proceedings of the 6th Ada-Europe International Conference on Reliable Software Technologies, London, UK, Springer-Verlag (2001) 305–316
14. Puente, J., Zamorano, J., Ruiz, J.F., Fernandez, R., Garcia, R.: The design and implementation of the open ravenscar kernel. ACM SIGAda Ada Letters **XXI**(1) (March 2001) 85–90
15. Gai, P., Abeni, L., Giorgi, M., Buttazzo, G.: A new kernel approach for modular real-time systems development. In: Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems. (June 2001)
16. Barabanov, M.: A linux-based realtime operating system. Master's thesis (1997)
17. Knuth, D.E.: The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)
18. Sáez, S., Lorente, V., Terrasa, S., Crespo, A.: Efficient alternatives for implementing fixed-priority schedulers. In: 10th International Conference on Reliable Software Technologies - Ada-Europe. (2005) 39–50