Non-Preemptive Interrupt Scheduling for Safe Reuse of Legacy Drivers in Real-Time Systems

Tullio Facchinetti, Giorgio Buttazzo, Mauro Marinoni, and Giacomo Guidi
University of Pavia, Italy
{tullio.facchinetti,giorgio.buttazzo, mauro.marinoni, giacomo.guidi}@unipv.it

Abstract

Low-level support of peripheral devices is one of the most demanding activities in a real-time operating system. In fact, the rapid development of new interface boards causes a tremendous effort at the operating system level for writing and testing low-level drivers for supporting the new hardware. The possibility of reusing legacy drivers in real-time systems would offer the great advantage of keeping the rate of changes with a small programming effort. Since typical legacy drivers are written to execute in a non-preemptive fashion, a suitable operating system mechanism is needed to protect real-time application tasks from unpredictable bursty interrupt requests.

In this paper we present a novel approach suitable for scheduling interrupt service routines. Main features of the method include: high priority of the handler, non preemptive execution, bandwidth reservation for the application tasks, and independence of the interrupt service policy from the scheduling policy adopted for the application tasks.

1. Introduction

One of the main problems of reusing legacy drivers in a real-time system is that most interrupt handlers disable the interruption capability of the processor, executing long portions of code at the highest priority in a non-preemptive fashion. As a consequence, a bursty sequence of interrupts may introduce long blocking delays, which would cause hard tasks to miss their deadlines and soft tasks to increase their response time. Under such an execution model for the interrupts, an off-line guarantee of real-time constraints could require the system to run with a very low utilization.

On the other hand, enabling the preemption of an interrupt service routine would increase the efficiency of resource utilization, but could jeopardize the correct management of the device. For example, many device drivers require a tight management of the delays between consecutive instructions. Hence, a preemption could introduce undesired delays, causing potential system instability or inconsistency. In many cases, a small jitter in the activation of the interrupt handler (executed in a non preemptive fashion) is tolerated by the application, and it is often safer than having a preemption during the execution of the handler.

Clearly, a more predictable behavior of the system could be achieved through a careful programming of the interrupt handlers and an off-line guarantee of the real-time tasks. However, such a solution requires a tremendous effort in terms of low-level programming and testing, due to the complexity of most modern I/O devices. Considering the huge amount of open source code currently available for almost all kinds of off-the-shelf peripherals, a very appealing option would be to easily integrate the existing code inside a real-time system, while still guaranteeing the real-time constraints of the application, as well as a stable behavior of the driver code.

A device driver is often modeled as an aperiodic task, that is, a sequence of jobs with known worst-case execution time (WCET) and unknown arrival times. To protect real-time application tasks from possible overruns due to bursty interrupt arrivals, drivers can be handled through an aperiodic server that bounds the processor demand to a given maximum utilization (the server bandwidth). Several aperiodic service mechanisms have been proposed in the real-time literature, both under fixed priority [7, 12] and dynamic priority systems [13, 1]. Unfortunately, most of the proposed approaches assume a fully preemptive system, where the server can suspend the execution of the driver at any time, either because of the arrival of a new job with a higher priority, or because the server budget is exhausted.

LeVasseur et al. [8] presented a method for reusing unmodified device drivers via virtual machine and proposed a heuristic approach to avoid preemptions into device driver's code. However, they did not consider applications with timing constraints, hence their method is not suitable for real-time systems. Abeni and Buttazzo [1] considered the case in which the served job may use critical sections of code to share mutually exclusive resources with other tasks, but the fully non preemptive case was not addressed.

Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)
0-7695-2400-1/05 $20.00 **IEEE**

A possible solution for ensuring a non-preemptive execution of an interrupt handler is to assign it the highest possible priority. This can easily be done under a static priority scheduling algorithm.

Katcher et al. [6] analyzed the overhead introduced by interrupt management and other kernel mechanisms; however, their analisys addresses fixed priority systems only. Leyva-del Foyo and Mejia-Alvarez [8] proposed an integrated approach to interrupt management with related analysis, but their method requires special hardware support.

Under the Earliest Deadline First (EDF) algorithm [10], Jeffay and Stone [5] presented a feasibility test for guaranteeing the schedulability of periodic task sets running in the presence of an interrupt handler executed at the highest priority level. Although this approach guarantees a non preemptive execution of the driver, letting the driver running at the highest priority level could be too restrictive for the periodic task set, in terms of feasibility. Allowing a certain amount of activation delay for the drivers would increase the bandwidth of the application tasks, while guaranteeing a safe non preemptive execution of the interrupt routines.

In this paper, we propose a novel service method for handling interrupt activities, with the following characteristics:

- The handler is always executed in a non preemptive fashion, but the server limits its bandwidth consumption through a suitable budget management that allows guaranteeing the other real-time activities.
- A hierarchical scheduling approach [9] is used to make the interrupt server independent of the scheduling policy, so that either fixed or dynamic priority assignments can be used for the application tasks.
- The server can be tuned to balance its responsiveness versus its bandwidth consumption.
- The mechanism can be efficiently implemented to reduce the extra overhead required in capacity-based servers to set the timers for the budget management.
- Finally, the context-switch overhead introduced by the interrupt requests can be easily taken into account in the guarantee test for the application tasks.

2. Server description

An interrupt request I_i is modeled as an interrupt service routine (ISR) with its own worst-case computation time C_i. The server is defined by 3 parameters: a maximum budget Q_{max}, a bandwidth U, and a budget threshold Q_θ. The server also keeps two state variables: its current budget $Q(t) \leq Q_{max}$ and an activity state $\Phi(t)$, which can have three values:

- exe. The server is in this state when it executes an ISR;
- $ready$. The server is ready when there are no pending interrupt requests and a new incoming request can be executed immediately without any activation delay;
- $idle$. The server is idle when a new request cannot be immediately executed because the previous requests consumed the available budget below the threshold Q_θ. In this state, the budget is recharged according to a given replenishment rule, until the maximum level Q_{max} is reached or a new request arrives.

The maximum budget (Q_{max}) is the upper bound for the current budget and limits the number of ISRs that can be consecutively executed by the server. The budget $Q(t)$ is decreased while an ISR is executing to keep track of the remaining budget that can be allocated to other requests. To prevent any preemption of the server, the budget is allowed to be negative. When no request is executing, $Q(t)$ is recharged at a constant rate.

The U parameter specifies the percentage of processor allocated to the server, which leaves a bandwidth $1 - U$ to the application tasks. The value of U directly influences the server budget $Q(t)$, which increases at rate U when the server is ready or idle, and decreases at rate $1 - U$ when the server is executing. A higher value of U makes the budget to decrease more slowly, thus allowing the execution of a higher number of ISRs before starting the recharge. On the contrary, decreasing U makes the budget to increase more slowly, thus letting more space for the application tasks.

The budget threshold Q_θ ($0 \leq Q_\theta \leq Q_{max}$) defines the budget level above which the server can start executing pending requests after an idle period. In other words, when the budget is exhausted ($Q < 0$) a new request can only be started when the budget is replenished up to Q_θ. However, if $Q > 0$ and the server is ready, an ISR can be executed even though $Q \leq Q_\theta$.

Decreasing the value of Q_θ decreases the latency of the ISR, while increasing Q_θ decreases the overhead introduced by the server during IRQ bursts. Such a dependency is better explained in Section 3.2.

While the server is $idle$, the ISRs that cannot be executed due to the bandwidth limitations are sent to a ready queue, which can be handled by an arbitrary discipline. Multiple queues can also be maintained to handle ISR classes with different latency requirements.

Then, they are fetched from the queue when the processor can be safely assigned to the server, meaning that the execution of an interrupt service does not jeopardize the temporal requirements of the application tasks.

Two examples of server execution are reported in Figure 1 to better illustrate the budget management mechanism and the server state transitions.

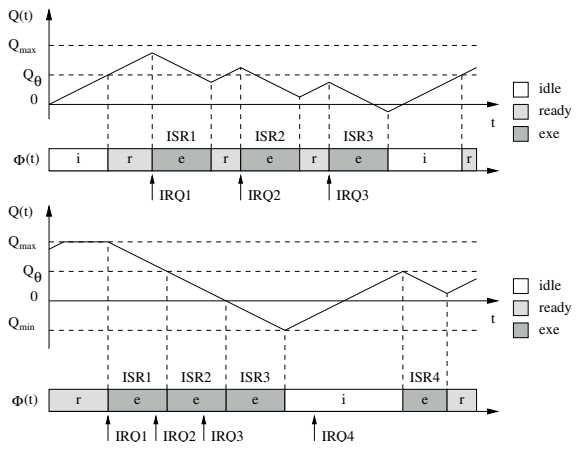

Figure 1. Samples server budget behavior.

2.1. Server rules

Budget consumption and recharging is regulated by the following rules:

1. At the system start-up $\Phi(0) = idle$ and the initial budget is set to 0, i.e., $Q(0) = 0$.
2. While $\Phi = idle$ or $\Phi = ready$, the budget increases at a constant rate U up to its maximum value. If $Q(t_1)$ is the budget at time $t_1 < t_2$, then
$$Q(t_2) = \min\{Q_{max},\ Q(t_1) + (t_2 - t_1)U\}. \quad (1)$$
3. While $\Phi = exe$, the budget decreases at a constant rate equals to $1 - U$. If $Q(t_1)$ is the budget at time $t_1 < t_2$, then
$$Q(t_2) = Q(t_1) - (t_2 - t_1)(1 - U). \quad (2)$$

The activity status of the server is determined by the current available budget, by the previous server status and by the presence or absence of pending ISRs into the ready queue. The status switches according to the following rules:

- The initial state of the server is $idle$;
- When an IRQ arrives, if Φ is exe or $idle$ the ISR is sent to the ready queue and the server maintains its current state;
- When an IRQ arrives, if $\Phi = ready$ the server starts executing the handler and $\Phi = exe$;
- When an interrupt handler terminates the execution, if $Q(t) < 0$ Φ switches from exe to $idle$; if $Q(t) \geq 0$ and the ready queue is empty, the server switches to $ready$, otherwise, if an ISR is waiting in the queue, the server keeps the exe state and starts executing the next ISR;
- When $Q(t)$ increases Φ can only be $idle$ or $ready$. If $\Phi = idle$, when $Q(t)$ reaches Q_θ and the ready queue is empty, the server switches to $ready$; if the queue is not empty it switches to exe and starts executing the first pending request. If $\Phi = ready$, when $Q(t)$ reaches Q_θ the server keeps its current status and keeps recharging up to Q_{max} if there are no IRQs to execute.

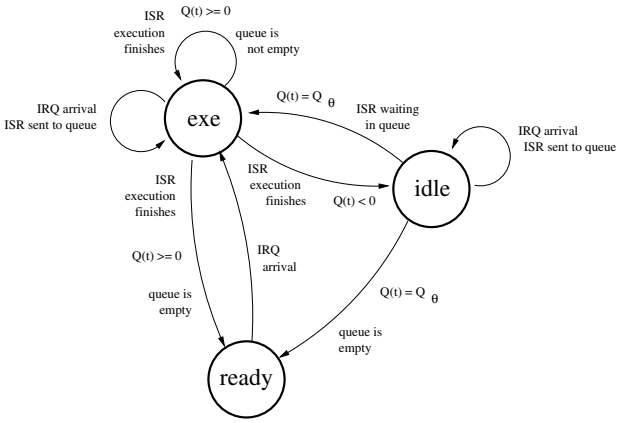

Figure 2. Server finite-states machine.

Figure 2 illustrates these rules as a finite-state machine.

3. Server Properties

The proposed interrupt server is characterized by the following interesting properties:

- the response time of every single ISR can be predicted to perform an online guarantee of incoming requests;
- the implementation overhead can be traded for the ISR latency by acting on the budget threshold Q_θ;
- the server parameters can be used to specify the bandwidth allocation within a hierarchical framework.

Such a properties will be formally addressed in the following sections.

3.1. Response time and online guarantee

The response time and the finishing time of each ISR can be determined when the corresponding IRQ is generated. This is fundamental for adopting admission control mechanisms to guarantee the system during overload conditions due to IRQ bursts. If a deadline violation is predicted, an error recovery strategy can be triggered.

To describe the algorithm for the online guarantee, we first introduce two variables: f_l keeps track of the finishing time of the last arrived IRQ, and Q_l represents the server budget at time f_l, i.e. $Q_l = Q(t_l)$.

Let us consider an interrupt request I_i triggered at time t_i. We have to calculate the f_l and Q_l variation due to the

new request arrival. In the following explanation we use the symbols f_l^{new} and Q_l^{new} to denote the updated values of the variables, whereas f_l^{old} and Q_l^{old} denote the old values. Basically, before the evaluation of each equation, the assignment $new\ value = old\ value$ has to be done in order to make the algorithm working iteratively. If the server ready queue is empty, then the server can be in one of the three activity states:

- If $\Phi = ready$ then the handler starts executing immediately, thus $f_l = t_i + C_i$ and $RT(I_i) = C_i$, and, since the budget decreases during the execution of I_i, from Equation (2) we have:

$$Q_l = Q(f_l) = Q(t_i) - (1 - U)(f_l - t_i).$$

  Notice that, in the previous equations, the new value of both f_l and Q_l (f_l^{new}, Q_l^{new}) do not depends on f_l^{old} and Q_l^{old}. This is because, from the point of view of our algorithm, when the ready queue is empty and $\Phi = ready$, the next new IRQ can be considered as the first IRQ ever occurred. For this reason we simply assign the value to f_l and Q_l.

- If $\Phi = exe$ there are two cases:
  - if $Q_l^{old} \geq 0$, then

$$f_l^{new} = f_l^{old} + C_i, \quad RT(I_i) = f_l^{new} - t_i$$

$$Q_l^{new} = Q(f_l^{new}) = Q_l^{old} - (1-U)(f_l^{new} - f_l^{old})$$

  - if $Q_l^{old} < 0$, then the current pending handler is delayed until the budget will be recharged. We calculate the time t_θ at which the budget will be recharged by using Equation 1 and by imposing

$$Q_l^{new} = Q(t_\theta) = Q_\theta = Q_l^{old} + U(t_\theta - f_l^{old})$$

    we obtain

$$t_\theta = \frac{Q_\theta - Q_l^{old}}{U} + f_l^{old}.$$

    Then

$$f_l^{new} = t_\theta + C_i$$

$$RT(I_i) = f_l^{new} - t_i.$$

- If $\Phi = idle$, since the ready queue is empty, f_l^{old} and Q_l^{old} still refer to the finishing time of the last executed ISR, thus the same relations illustrated for the $\Phi = exe$ case with $Q_l^{old} < 0$ hold.

If the ready queue is not empty when IRQ_i arrives, Φ cannot be $ready$. If it is $idle$ or exe, the same relations previously illustrated can be used.

3.2. Effect of the budget threshold

The threshold Q_θ is used in our model to introduce an extra-delay between the time at which the server becomes $idle$ and the time at which it switches to $ready$ again. This is useful to limit the overhead produced by the system timers, allowing an efficient implementation of the server. The basic idea is that, when Q_θ increases, the system overhead decreases but the average ISR response time increases.

Since when $\Phi = ready$ or $\Phi = exe$ the server continuously executes the ISRs enqueued into the ready queue until $Q(t) < 0$, an efficient server implementation may update the server budget on 3 events only:

1. before the execution of the first ISR fetched from the queue;
2. after the end of the last executed ISR (when $\Phi = idle$);
3. when Φ switches from $idle$ to $ready$, because $Q(t) = Q_\theta$.

While in the first two situations the only overhead introduced by the server is the computation of the new budget, the third case requires a system timer to be implemented. The timer is set when the server becomes idle in order to trigger an event when the server has to wake up (to switch Φ to $ready$ and set $Q(t) = Q_\theta$). Since the system timers are usually implemented as interrupt routines, the overhead includes the context switch time and the execution time of the required routine. Therefore, the overhead introduced by the system timers depends on the frequency of the timer activation events.

Notice that the system timers are not managed by the server. Although they are also handled as interruptions, they are generated from the system and they require a separate mechanism to be managed. The server only handles the interrupts coming from the legacy drivers.

To discuss the effect of the threshold we consider the worst-case situation in which there is always at least one ISR waiting into the ready queue. This is the typical situation during interrupt burst conditions. We assume the duration of the system timer routine equal to C_{timer}.

Finally, we consider the non-restrictive assumption in which all the ISRs have the same duration C_{int}. This assumption requires a little explanation. Since the timers are triggered when a transition from the $idle$ state to an another state (exe or $ready$) is required, they are set when an ISR finishes its execution and $Q(t) < 0$. Moreover, the higher the frequency of timer activation the higher the overhead introduce by them. For a given Q_θ, the worst case happens when an ISR finishes at an instant t_{end} for which $\epsilon < Q(t_{end}) < 0$ for an arbitrary low ϵ, because in this case the next timer activation is the closest possible to the previous one. When $Q_\theta = 0$ and there is always an ISR waiting in the queue, the value of ϵ is affected only by the duration

of the ISR: the shorter is the ISR execution the shorter is ϵ and the higher is the overhead. When $Q_\theta > 0$ the overhead does not depend from the ISR duration only, since it depends even on the ISR arrival order and, potentially, on the ratio between Q_θ and ISR duration. Here we do not want to compare the the cases with same Q_θ when the ISR characteristics change. We want to compare the cases with different Q_θ while keeping other parameter fixed. In this situation, C_{int} can be assumed as the shortest ISR execution duration, i.e. $C_{int} = min(C_i)$.

To estimate the system overhead we need to calculate the time between two consecutive timer activations. Since a timer activation occurs when the budget increases up to Q_θ, we have to evaluate the maximum interval Δ_t in which $Q(t)$ reaches Q_θ again after reaching its minimum value $Q_{min} \leq 0$. Referring to the example illustrated in Figure 3, let t_a be the time at which $Q(t_a) = Q_\theta$ and Φ switches from $idle$ to exe, let t_b be the time at which $Q(t_b) = Q_{min}$ and Φ switches from exe to $idle$, and let t_c be the time at witch $Q(t_c) = Q_\theta$ and Φ switches from $idle$ to exe again. Then, $\Delta_t = (t_c - t_a)$. As we will see later, the duration of Δ_t depends on Q_θ, C_{int} and U.

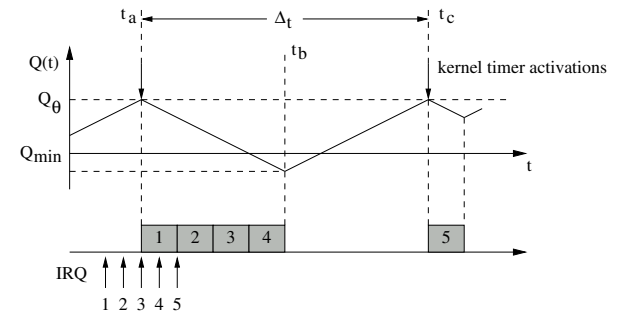

Figure 3. IRQ scheduling with threshold.

To determine the value of Q_{min}, let us consider the situation in which $Q_\theta = 0$. If there is always an ISR into the ready queue, the budget follows the behavior illustrated in Figure 4. At each ISR execution, the budget reaches the value of $Q(t_b) = Q_{min}$. From Equation 2 we have $Q(t_b) = Q(t_a) - (1-U)(t_b - t_a)$, where $Q(t_b) = Q_{min}$, $Q(t_a) = 0$ and $t_b - t_a = C_{int}$, so that $Q_{min} = -(1-U)C_{int}$.

Now let us consider the situation depicted in Figure 3, where $Q_\theta \neq 0$ such that $Q_\theta = nQ_{min} < Q_{max}$ for a given integer n. Considering the interval $[t_a, t_b]$ and Equation 2, we can write $Q(t_b) = Q(t_a) - (1-U)(t_b - t_a)$, where $Q(t_b) = Q_{min}$, $Q(t_a) = Q_\theta$. Then, we obtain

$$t_a = t_b - \frac{Q_\theta - Q_{min}}{1-U}.$$

For the interval $[t_b, t_c]$ we use Equation 1. Since we are considering the case with $\Phi = idle$, then $Q < Q_{max}$

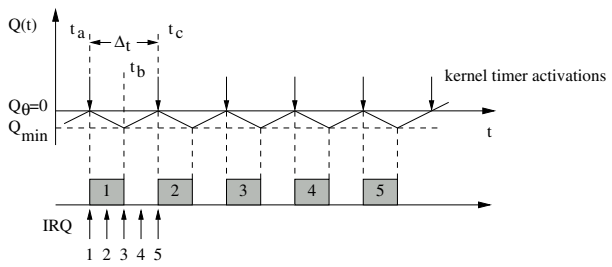

Figure 4. IRQ scheduling without threshold.

and the budget is increasing. We can say that $Q(t_c) = Q(t_b) + U(t_c - t_b)$, where $Q(t_b) = Q_{min}$, and $Q(t_c) = Q_\theta$. Then, we obtain

$$t_c = t_b + \frac{Q_\theta - Q_{min}}{U}.$$

We determine Δ_t as

$$\Delta_t = t_c - t_a = \frac{Q_\theta - Q_{min}}{U(1-U)}.$$

If U_{timer} is the bandwidth consumed by the system timers, in any time interval $[t_1, t_2]$ between two timer activations, we have

$$U_{timer} = \frac{\sum_{[t_1,t_2]} C_{timer}}{t_2 - t_1},$$

where $\sum_{[t_1,t_2]} C_{timer}$ is the total amount of time consumed for the timer management during $[t_1, t_2]$. And since $\sum_{[t_1,t_2]} C_{timer} = \frac{t_2-t_1}{\Delta_t} C_{timer}$ (because $\frac{t_2-t_1}{\Delta_t}$ represents the number of timer activations in $[t_1, t_2]$), we have

$$U_{timer} = \frac{C_{timer}}{Q_\theta + (1-U)C_{int}} U(1-U).$$

The above expression allows us to derive some interesting considerations about the threshold mechanism. To achieve the minimum response time of an ISR the threshold should be set to $Q_\theta = 0$. By doing so, we have

$$U_{timer}(Q_\theta = 0) = \frac{C_{timer}}{C_{int}} U.$$

In this case, the bandwidth consumed for the timer management becomes relevant as soon as C_{int} becomes comparable with C_{timer}. Figure 4 illustrates the schedule obtained on a burst of interrupts without threshold, that is with $Q_\theta = 0$. In such a situation, the oscillation of $Q(t)$ around 0 produces several activations of the timers used to trigger the transition of Φ from $idle$ to $ready$. If the threshold is set to $Q_\theta > 0$, the execution of some ISRs suffers from a small extra delay, but the number of timer activations decreases, since several ISRs are executed together after the same activation, as shown in Figure 3.

The threshold level can be tuned to balance the handler response time versus the timer management overhead.

3.3. The hierarchical framework

The interrupt server presented in this paper is assumed to be handled using a hierarchical scheduling framework in order to decouple the analysis of the interrupt handling from the application tasks scheduling [9]. This approach enables our method to be used on top of both fixed priority and dynamic priority assignment schemes for the application tasks.

Almeida et al. [2] also addressed one form of hierarchical scheduling and presented a schedulability analysis of a non-preemptive periodic task set with fixed priority within a dedicated temporal frame, applied to the CAN bus. Feng and Mok [3] introduced a function to measure the time made available by a server, and analyzed it for a static allocation of the bandwidth. A hierarchical approach was also proposed by Shin and Lee [11] to design a server that guarantees the application tasks executed separately by different servers.

In this work, the hierarchical structure of the system is organized with an upper level, the global scheduler, that provides the execution time to a lower level, made by one or more concurrent local schedulers executing the application tasks with potentially different scheduling policies. The analysis is performed using the hierarchical partitioning approach introduced by Lipari and Bini [9].

Each server is modeled with two parameters:

(i) the maximum bandwidth α is the amount of server budget Q available on a period P, having $\alpha = \frac{Q}{P}$;

(ii) the maximum consecutive idle time (Δ).

A server described with such a parameters is guaranteed both under EDF and RM [10] with the tests reported in [9].

In our approach, the interrupt manager is located at the lower level of the hierarchical structure, so that it holds always the highest priority within the system. Then, the server assigns the computation time to the ISR only if there are requests for interrupts and there is enough bandwidth to satisfy the request. Otherwise, the bandwidth is assigned to the higher level schedulers and consumed by the application tasks.

The schedulability test used to guarantee the application tasks is based on the method presented in [9], using the α and Δ parameters. We firstly introduce the following lemma to bound the continuous execution on the server.

Lemma 1 *The maximum server continuous execution time is*

$$C_w = \max_i C_i + \frac{Q_{max}}{1-U}.$$

Proof. If the initial budget is at the maximum allowed value Q_{max}, the server can execute for a period $\frac{Q_{max}}{1-U}$ before the budget reaches the 0 value. If a new ISR with duration equal to $\max_i C_i$ arrives, then the server budget becomes negative, the server becomes idle, and the budget reaches its lowest value. Therefore, the total execution time for the worst-case sequence of interrupt requests is $C_w = \max_i C_i + \frac{Q_{max}}{1-U}$. □

Since C_w depends on Q_{max}, one of the results that comes directly from Lemma 1 is that raising the value of Q_{max} makes the server able to respond quickly to an higher number of requests before becoming $idle$.

The α and Δ parameters can be derived from the server parameters according to the following theorem.

Theorem 1 *The server idle time can be represented with the two parameters α and Δ, where $\alpha = 1 - U$ and $\Delta = \max_i C_i + \frac{Q_{max}}{1-U}$.*

Proof. If the server has a bandwidth U, as derived from property 1, its idle time has a bandwidth $1 - U$ and $\alpha = 1 - U$. Delta is the maximum delay which affects the execution time passed to the task level scheduler. Such a maximum delay corresponds to the maximum execution time of the server. From Lemma 1 we have $\Delta = C_w = \max_i C_i + \frac{Q_{max}}{1-U}$. □

4. Experimental results

To test the behavior of the proposed interrupt management mechanism, the server has been implemented as a scheduling module in the Shark real-time operating system [4]. Then the server has been used to schedule the interrupt requests coming from device drivers imported from the Linux distribution without any modifications.

To trigger the IRQ generation, we have used a personal computer (PC) connected to the target machine through a parallel port. The PC generates explicit IRQs by raising a signal on the interrupt line of the parallel port. The frequency of the IRQs is controlled by the PC, so that different bursty situations can be easily tested.

The test application we have set up to evaluate the interference of the server on the application tasks consisted of a task set scheduled using the EDF scheduling policy. The task set is made by several relatively fast hard periodic tasks with a period of $1000\mu s$ and a measured computation time of $93\mu s$. The task set generates a total workload equal to $U_{app} = 0.8$.

The general interrupt arrival sequence is a burst of requests with a given duty cycle σ. We tested the server in several situations, by imposing different duty cycles to the IRQ bursts and by setting different server parameter configurations. In all the experiments we measured the ISR activation latency. The period between two consecutive bursts is always equal to 8 ms and the burst duration is changed for each experiment in order to vary the value of σ. For example, $\sigma = 40\%$ means that an IRQ burst of 3.2 ms is generated every 8 ms. The burst duration is measured by quanta of $400\mu s$, and in each quantum we randomly generate from

1 to 5 interrupt requests. These values are constrained by the physical requirements needed for the IRQ generation using the parallel port.

The ISR latencies are represented in a graph reporting on the horizontal axis every single ISR activation sorted by latency, and on the vertical axis the corresponding latency. Such a representation provides the same information of a traditional histogram, moreover it allows the visualization of several curves on the same graph, making the latency comparison easier.

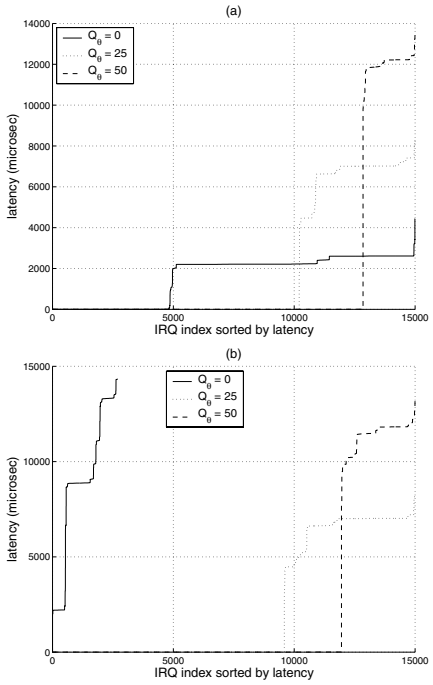

Figure 5. ISR activation latency distribution with different values of Q_θ and two different duty cycles: a) $\sigma = 30\%$ and b) $\sigma = 70\%$.

Figure 5 reports the results of an experiment in which $Q_{max} = 50, U = 0.005$ and the behavior of the latency was study with different values of Q_θ and two different duty cycles: a) $\sigma = 30\%$ and b) $\sigma = 70\%$. We notice that, for $Q_\theta = 0$, almost all the ISRs experience a high delay. Raising the value of Q_θ (25 and 50) increases the number of ISRs that are scheduled without any latency, as shown by the two curves that start flat and increase later. The difference between the two pictures is that, in Figure 5 b) the duty cycle is sufficiently high to keep the ready queue busy all the time. This implies that every new IRQ is enqueued and the delay constantly increases. Figure 5 a) shows the typical behavior of the latency when the duty cycle is not sufficiently high to make the server enqueueing all the new arrivals: increasing the value of Q_θ the number of ISR that are scheduled increases, but the worst latency increases significantly. As a result, low duty cycles and low threshold make the delay more uniformly distributed.

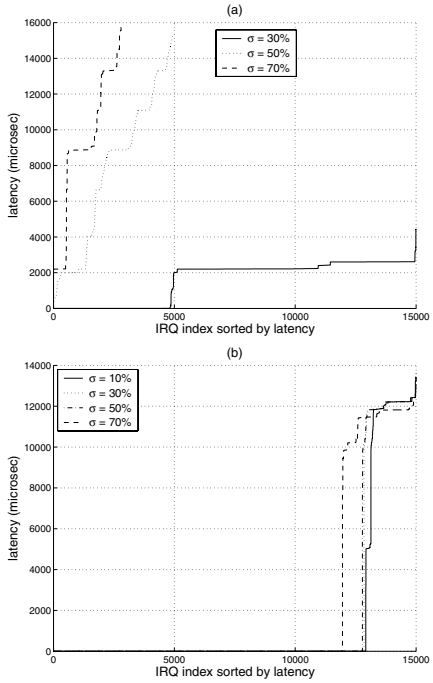

Figure 6. ISR activation latency distribution with different values of σ and two different thresholding levels: a) $Q_\theta = 0$ and b) $Q_\theta = 50$.

In another experiment shown in Figure 6, we still set $Q_{max} = 50$ and $U = 0.005$, but we study the behavior of the latency with different values of σ and two different threshold levels: a) $Q_\theta = 0$ and b) $Q_\theta = 50$. In both graphs the curves become more abrupt as σ increases, but this behavior is much more evident for low threshold levels (case (a)). Since the speed at which the server is able to schedule the ISRs does not vary, raising the duty cycle and reducing the threshold have the join effect of keeping the ready queue always occupied, so that new IRQ arrivals are always enqueued and their latency constantly increases.

Figure 7 shows the behavior of the ISR latency while keeping the threshold and the duty cycle fixed ($Q_\theta = Q_{max}$ and $\sigma = 70\%$) and changing U and Q_{max}.

Increasing the server bandwidth U there is a reduction of the latency together with a gain on the number of ISR that are scheduled with zero latency. Raising U causes two effects: a lower slope of the budget variation during the ISR execution, which implies more time to execute the ISRs, and a higher slope of the budget variation during the bud-

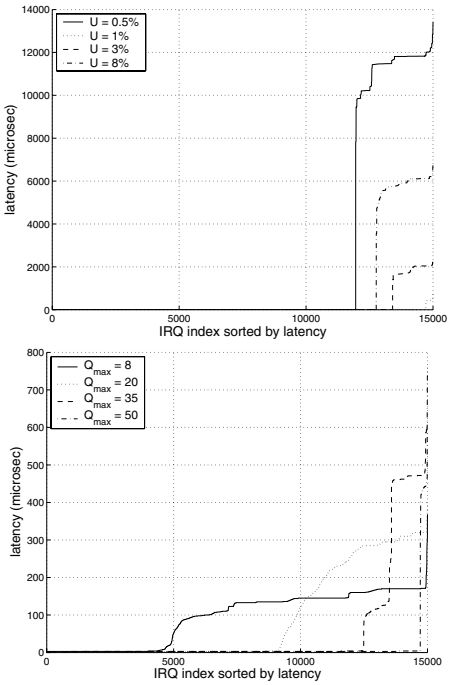

Figure 7. ISR activation latency distribution with different values of U and Q_{max}.

get recharge, which involves less time between two server activations, thus less latency for the ISRs execution.

Raising Q_{max} the curves shift to the right, increasing the number of ISRs that are served with low latencies, but also increasing the overall worst-case performance in terms of individual ISR's delay. This happens because, if Q_{max} raises, there is more time to execute ISRs consecutively, even if the slope of the budget variation does not change. On the other hand, higher values of Q_{max} implies higher values of Q_θ (we set Q_θ equal to the maximum possible value) and longer recharging periods, so that the worst-case latency increases consequently.

5. Conclusions

In this work we presented a novel approach for the efficient reuse of legacy device drivers in real-time systems.

Our method enforces a non-preemptive execution of the interrupt handlers in order to preserve the internal temporal requirements of the ISRs, that are fundamental for a predictable behavior of the system. The server runs in a hierarchical environment with an assigned bandwidth, so that the real-time application tasks can be guaranteed independently from the server. Moreover, the interrupt server policy is completely independent from the scheduling policy adopted for the application tasks.

We provided both theoretical and experimental results to show the effectiveness of our approach. While the theory validated the properties of the model, the experimental results showed the performance of the server under some realistic working situations.

The interrupt server has been integrated in the Shark real-time kernel and it is used to schedule the interrupt requests coming from device drivers imported from the Linux distribution without any modification.

References

[1] L. Abeni and G. Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, July 2004.

[2] L. Almeida, P. Pedreiras, and J. A. G. Fonseca. The ftt-can protocol: Why and how. *IEEE Transaction on Industrial Electronics*, 49(6):1189–1201, December 2002.

[3] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proc. of the 23rd IEEE Real-Time Systems Symposium*, pages 26–35, Austin, TX, USA, Dec. 2002.

[4] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proc. of the 13th IEEE Euromicro Conf. on Real-Time Systems*, pages 199–206, Delft, The Netherlands, June 2001.

[5] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 212–221, Raleigh-Durham, NC, USA, December 1993.

[6] D. Katcher, H. Arakawa, and J. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, 1993.

[7] J. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, San Jose, CA, USA, December 1987.

[8] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, USA, December 2004.

[9] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proc. of the 15th Euromicro Conf. on Real-Time Systems*, pages 151–, Porto, Portugal, July 2003.

[10] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):40–61, January 1973.

[11] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of the 24th Real-Time Systems Symposium*, pages 2–13, Cancun, Mexico, Dec. 2003.

[12] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time system. *Journal of Real-Time Systems*, 1:27–60, June 1989.

[13] M. Spuri and G. C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2):1–32, 1996.