Real-Time Scheduling Models: an Experimental Approach


António J. Pessoa de Magalhães
a.p.magalhaes@fe.up.pt
Fax: 22 207 4247
SAIC – DEMEGI
Faculdade de Engenharia da
Universidade do Porto - Portugal

Carlos J. A. Costa
ccosta@babbage.ipg.pt
Fax: 271 222 690
DI - ESTG
Instituto Politécnico da Guarda
Portugal

Abstract – Classical real-time scheduling models, as those commonly referred in literature, tend to ignore the impact of system tasks upon the timeliness of application tasks. However, as this paper testifies, this overhead is not negligible even for very simple systems. Accurate scheduling models from where control engineers can confidently design their real-time applications are thus required. In this paper such models are experimentally deduced and proved to be valid for the most relevant single processor scheduling schemes.

Key Words: Computer Control, Real-Time Systems, Scheduling Models, Architectures, Overhead Estimation

1 Introduction

Real-time systems must perform correctly in both the value and the time domain [1]. Computer controllers are probably the most known real-time systems. This is because a controller must conform to some time constraints in sensing the environment and performing control actions accordingly [2]. Such time constraints are usually provided in the form of deadlines [3].

Real-time scheduling aims to devise conditions for guarrantying processes deadlines in concurrent computer control applications. Current literature is rich in techniques and algorithms for real-time scheduling [4], [5], [6], [7]. Yet, most of the authors depart from load models that only cover application processes characteristics – e.g. maximum execution time, minimum time interval between execution requests, and deadlines. The computational overhead introduced by the operating system for interrupt handling, task scheduling management and context switching are usually not taken into account. As a consequence, a real-time schedule declared feasible according to a classical model may fail in practice to guarantee processes deadlines. This is particularly true when high processor utilisation is achieved and deadlines tend to be met by a small margin, if any.

The motivation for this paper is thus the notion that the mathematical frameworks provided by classical scheduling algorithms do not perfectly handle the complexity of real-time applications. Therefore, its first aim is to present augment scheduling models from where control engineers can confidently analyse a schedule and deduce if it is feasible or not. But since it is impossible to cope in here all the real-time scheduling algorithms, the paper is restricted to static and fixed priority scheduling schemes implemented on single processor systems. These are the simplest and most common scheduling schemes.

The measurement of the impact of the system software execution upon the overall system performance requires the quantification of a few timing parameters. Some operating system suppliers provide the timing values required for such quantification. However, the conditions under which such parameters were obtained are not always mentioned or are different from those of interest. Sets of experimental procedures that can bridge this gap are revealed in this paper. They result from an eclectic view on several fine-grained [8], [9], [10], [11], [12] and application oriented [13], [14], [15], [16] real-time benchmarks that can be easily tailored to any real-time single processor system. The devising of these procedures is thus another major contribution of this paper.

The remainder of the paper is organised as follows. Section 2 augments classical real-time scheduling models through the inclusion of the overhead introduced by the supporting software system. Preemptive and non-preemptive, as well as cyclical and fixed-priority scheduling strategies are covered. Section 3 provides a set of experimental procedures that enable the quantification of the overheads discussed in the previous section. Section 4 investigates how the devised scheduling models differ from classical models in a quantitative form. The results show that the computational overhead ignored in classical models can easily make real-time systems to miss their deadlines when high processor utilisation and fast response is a major concern, as it is usually the case. Section 5 ends the paper summarising the most important conclusions.
2 Qualitative Models

This section presents several qualitative scheduling models that express the impact of kernel tasks upon the timeliness of application tasks. Two kernel costs are considered: overhead and blocking. Overhead is the time taken by the kernel in performing a service on behalf of a specific task – such as invoking, resuming or terminating it. Blocking is the amount of time that a task is prevented from running due to the execution of a kernel operation that cannot be avoided – e.g. handling an interrupt that does not cause any task switching.

Two scheduling approaches founded in most real-time kernels are considered here: event-driven and time-driven. In event-driven scheduling, interrupts denote task arrivals, and the scheduler depends on external hardware devices for interrupt generation. In time-driven implementations a cyclic timer interrupt activates the scheduler to perform its duty.

Figure 1 – Scheduling Queues

As usual, it is assumed that the system kernel includes three data structures that store the information required for scheduling support – Fig. 1. The run queue stores task control blocks (TCB) of tasks ready to run. The start queue includes TCBs of (periodic) tasks that have already executed for completion in the present period and are waiting for their next period to start again. The active queue stores the TCB of the task that is currently running on the single available processor. It is assumed that the start queue is ordered by task’s release time and the run queue is ordered by task’s priorities. It is also assumed that:

- In time driven scheduling, every time an interrupt occurs, the interrupt handler saves same registers, performs some operations and then calls the scheduler. Later, if the scheduler decides not to preempt the active task, the saved registers are restored and the active task is resumed. If a task switch is performed, the TCB of the active task is stored in the run queue according to its priority. The TCB into the head of the run queue is passed to the active queue, and the new active task executes.

- When a task ends execution, it traps the scheduler. The trap handling routine selects the new active task from the head of the run queue.

- The TCB of a periodic task is stored into the start queue on a time driven basis. This is performed by a mechanism internal or external to the kernel. TCBs are passed from the start to the run queue – also on a time driven basis – by the kernel or by the interrupt handling routine.

From these assumptions the following general parameters define the overheads and the blocking times that a predictable scheduling model must include. These apply to non-preemptive and preemptive scheduling:

- \( C_{\text{int}} \) – the time to handle an interrupt request.
- \( C_{\text{sched}} \) – the time to execute the scheduling code to determine the next task to run. It includes comparing the head of the run queue with the active task and moving TCBs from the start to the run queue.
- \( C_{\text{resume}} \) – the time to resume the active task when task switch does not occur. It includes the time to restore the registers saved by the interrupt handler.
- \( C_{\text{store}} \) – the time to save the state of the active task to a TCB, and sort it into the run queue.
- \( C_{\text{load}} \) – the time to load the new active task state from the run queue.
- \( C_{\text{trap}} \) – the time to handle the trap generated by the normal completion of a task. This includes storing the TCB of the completing (periodic) task into the start queue and selecting the head of the run queue to be the next active task.

2.1 Cyclical Scheduling

We begin by the simplest case: static scheduling. The model includes \( n \) independent tasks. Each task, \( \tau_i \), is characterised by its period, \( T_i \), and its worst-case execution time (if never interrupted), \( C_i \). The dispatcher is based on an interrupt driven real-time clock and must follow a task execution order established off-line. In the worst case scenario, no idle time exists between the execution of two consecutive tasks – Fig. 2.

\[
C_{\text{exit}} = C_{\text{trap}} + C_{\text{load}}
\]  

(1)

A blocking time also exists due to periodic timer interrupts. In the worst case such blocking takes the value:

\[
C_{\text{timer}} = C_{\text{int}} + C_{\text{sched}} + C_{\text{resume}}
\]  

(2)
The total blocking time for a task $\tau$ depends on the number of interrupts occurred during its execution. Denoting the time between two consecutive interrupts by $T_{tic}$, one finds that the total blocking time for a task $\tau$, takes the form:

$$C_{\tau} = \left[ C_i \right] \times \frac{C_{timer}}{T_{tic}}$$

(3)

When overhead and blocking terms are ignored, the cyclic execution of the $n$ tasks requires a time interval given by:

$$C_{total} = \sum_{i=1}^{n} C_i$$

(4)

However, when overhead and the blocking terms are considered the same parameter reverts to:

$$C_{total} = \sum_{i=1}^{n} \left( C_i + \left[ C_i \right] \times \frac{C_{switch}}{T_{tic}} \right) + nC_{exit}$$

(5)

Another very old and simple scheduling strategy used in traditional multitasking systems is round robin [17].

In round robin scheduling two scenarios are possible: the time-slice is equal to the time interval between two consecutive ticks (i.e., it matches the timer resolution) or the time-slice is a multiple $k > 1$ of the timer period. If the time-slice is equal to the resolution of the timer, two overheads are introduced – Fig. 3. The first is related to task switching when a time-slice is exhausted – i.e., at all timer ticks. The second is a consequence of the traps caused by the completion of a task as defined in the context of static scheduling. If the time-slice is greater than the resolution of the timer, an additional blocking must be considered. This is due to interrupts that do not cause a task switch.

It is possible to define a round robin scheduling model at the light of the concepts presented for static scheduling. This is done and proved to be valid in [17]. It states that if the time-slice matches the timer resolution, then the cyclic execution of the $n$ tasks requires a time interval given by:

$$C_{total} = \sum_{i=1}^{n} \left( C_i + \left[ C_i \right] \times \frac{C_{switch}}{T_{tic}} \right) + nC_{exit}$$

(6),

where $T_{tic}$ denotes the timer period. In this case, $C_{switch} = C_{switch} + C_{timer}$ is the time required to handle an interrupt and perform a context switching. Therefore, $C_{switch} = C_{store} + C_{load}$.

In [17] is also shown that if the time-slice is greater than the period of the timer, then expression (6) gets the form:

$$C_{total} = \sum_{i=1}^{n} \left( C_i + \left[ C_i \right] \times \frac{C_{switch}}{T_{tic}} \right) + \left[ C_i \right] \times \frac{C_{timer}}{T_{tic}} + nC_{exit}$$

(7)

### 2.2 Fixed Priority Scheduling

Preemptive fixed-priority scheduling has gained an every increasing practical importance since the publication of Liu and Layland’s work [18]. Yet, only in 1989 Lehoczky et al. found an analytical condition from where the feasibility of a fixed-priority schedule can be assessed [19].

Whist the work developed by Lehoczky et al. is very important, it is not complete in the sense that it ignores the impact of the time required to perform system tasks. And there are reasons to believe that such overhead in not negligible, since interrupt handling, task switching and preemption are vital to fixed priority scheduling and may occur frequently.

Two implementations are possible for a fixed priority scheduler [20]: event-driven and time-driven. In event-driven scheduling, all tasks are initiated by internal or external events – Fig. 4. Time-driven scheduling relies on a timer that periodically interrupts the active application task and invokes the scheduler – Fig 5.
Both scenarios are deeply analysed in [17], which concludes that the following model applies to event-driven scheduling:

\[ W(t) = \sum_{j=1}^{n} \left( C_j + C_{\text{preempt}} \right) \times \left[ \frac{t}{T_j} \right] + \sum_{j=1}^{n} \left[ \frac{t}{T_j} \right] \times C_{\text{nonpreempt}} + \sum_{j=1}^{n} \left[ \frac{t}{T_j} \right] \times C_{\text{exit}} \]  \( (8) \)

When a fixed-priority scheduling is driven by the passage of the time, [17] proves that the following expression holds:

\[ W(t) = \sum_{j=1}^{n} \left( C_j + C_{\text{preempt}} \right) \times \left[ \frac{t}{T_j} \right] + \left[ \frac{t}{T_{jic}} \right] \times C_{\text{timer}} + T_{jic} + \sum_{j=1}^{n} \left[ \frac{t}{T_j} \right] \times C_{\text{exit}} \]  \( (9) \)

As it can be easily noticed, if both overhead and blocking conditions are ignored, both expressions (8) and (9) revert to the classical workload model stated in [19]:

\[ W(t) = \sum_{j=1}^{n} C_j \times \left[ \frac{t}{T_j} \right] \]  \( (10) \)

3 Quantitative Analysis

The quantification of the parameters introduced in the last section is vital to the exact characterisation of a scheduling model. Some operating systems suppliers provide the timing values required for such quantification. Yet, the conditions under which the provided system parameters were defined are not always revealed or are different from those of interest. Therefore, it is very convenient to define a set of procedures from where overhead and blocking times can be driven. These procedures – based on a set of fine-grained and application-oriented real-time benchmarks – are briefly justified in this section. The illustrating values are for a very simple hardware (an 80386 @25MHz based system) running a freeware kernel: Clock [21]. More details on these procedures can be found in [17], where the pseudo-code relating such procedures is also provided.

3.1 Non-periodic Parameters

The quantification of \( C_{\text{exit}} \) is somehow trivial. It suffices to consider \( n \) non-preemptive empty tasks (i.e., tasks that do nothing – just end) and measure the time interval between the moment the first task starts execution and the moment the last task ends execution. Such time, divided by the number of tasks minus one, provides \( C_{\text{exit}} \). In our experiences we found \( C_{\text{exit}} = 185 \mu s \).

The second parameter to determine is \( C_{\text{timer}} \). Here, we must consider a task whose nominal execution time (i.e., its execution time from the beginning to end without interruptions) is well known and greater than the time interval between two consecutive timer interrupts – Fig. 6.

When the task executes under the kernel control, it is periodically interrupted. Every time an interrupt occurs, the scheduler code is executed; later the task is resumed. Thus, the difference between the execution time of the task under the kernel control and its nominal execution time gives the overload introduced by the kernel. This time difference divided by the number of interrupts occurred during task execution provides \( C_{\text{timer}} \). The number of interrupts occurred during task execution is given by:

\[ \text{Number of Interrupts} = \left[ \frac{\text{Nominal Execution Time}}{T_{i0}} \right] \]  \( (11) \)

In our experiences we found \( C_{\text{timer}} = 136 \mu s \).

![Figure 6 – Measuring \( C_{\text{timer}} \)](image)

The next parameter to quantify is \( C_{\text{switch}} \). This is obtained by forcing two tasks to switch their execution a predefined number of times. Yet, it is important to note that the scheduler executes every time a task switch occurs. Thus, the time measured includes task switch and scheduler execution times. To take care of this situation, we considered two identical tasks formed by a loop that motivates a given number of task switches. We begin by measuring the time to execute the two loops, without the kernel control. Then, a similar measure is taken when the two tasks execute under the kernel control. The difference between both measures, divided by twice the number of loops provides \( C_{\text{switch}} \). In our experiences we found \( C_{\text{switch}} = 280 \mu s \).

3.2 Periodic Parameters

To obtain \( C_{\text{preempt}} \) it suffices to consider two tasks with different priorities. Both execute for a sufficiently long time in a loop. The low priority task also includes an inner loop whose execution time is nearly one timer tick, and where it waits preemption. The high priority task suspends by a tick, allowing the kernel to execute the low priority task. When the suspension period ends, the high priority task preempts the second task. Once again, tasks are firstly executed without kernel control. Then, they run under the control of the kernel. \( C_{\text{preempt}} \) is obtained by the difference between the two execution times divided by the number of loops performed – Fig. 7. In our experiences we found \( C_{\text{preempt}} = 280 \mu s \).

The last parameter to define in the scope of the fixed priority scheduling is \( C_{\text{exit}} \). This parameter is greater than or equal to the \( C_{\text{exit}} \) considered for cyclic scheduling. This is because, in fixed-priority scheduling, there is an extra time due to the ordered
insertion of the TCB of the periodic tasks into the start queue. In this case, we considered two tasks whose priorities were defined according to the rate monotonic scheduling algorithm. The higher priority task executes for completion and reads the elapsing time before completing. Then, the low priority task executes and reads the elapsed time at the beginning. In this scenario, $C_{exit}$ is given by the difference between these time measures. In our experiences it was found $C_{exit} = 300 \mu$s.

![Figure 7 – Measuring $C_{preempt}$](image)

It is worth stating that a considerable set of experiences for both cyclical and fixed priority models was performed in order to validate the proposed models. Such experiences departed from application workloads similar to those considered for some case studies presented in real-time literature. The resulting schedules were first evaluated according to the scheduling models proposed above. Later, the task sets were executed on the experimental system, and the most relevant parameters were measured. In both cases – cyclical and fixed priority scheduling – expected and measured values were found to closely match.

![Figure 8 – Overhead vs. Workload](image)

4 Overhead Analysis

Even though in the last section overhead and blocking terms were found to be very small, they can have a considerable impact upon the application tasks. This is because, in many applications, system services are performed very often, or can take a considerable time comparable to task execution times. Moreover, since each overhead and blocking term yields a particular influence upon the execution of application tasks, some workloads and scheduling schemes are expected to be more susceptible to a given parameter than to another. The present section discusses this subject and testifies its practical importance.

Let’s first consider static scheduling. In this case, and according to expression (5), application tasks can have their execution delayed due to both $C_{exit}$ and $C_{timer}$.

To understand the impact of these parameters upon the application tasks, one must consider a system that executes a task set given by a workload defined by expression (4) – i.e., an workload that is consumed in a time interval given by expression (4) when system overhead is ignored. Therefore, the impact of system tasks upon the application tasks is given by the difference between expression (5) and (4).

Since $C_{exit}$ is introduced every time a task completes execution, its overall impact increases as application tasks decrease their execution time. Such impact is depicted in Fig. 8. Each workload denotes a particular task set, each of which consisting of 8 tasks. Timer interrupts occurred at the same frequency in all the experiments (1/55ms).

From Figure 8 it can be easily concluded that, in static scheduling, system overhead is particularly notorious for short tasks. On the other hand, system overhead can be disregarded when application tasks have a long execution time and meet their deadlines by a “comfortable” margin. The same conclusions apply somehow to round robin scheduling – Fig. 9. Yet, in round robin scheduling the overload is inevitably greater than in static scheduling. This is due to the $C_{switch}$ term, which is absent in static scheduling.

![Figure 9 – Overhead vs. Scheduling Scheme](image)

The impact of $C_{timer}$ upon the execution of application tasks can also be easily testified. It suffices to trace the execution time of a given and well-known workload for different timer interrupt frequencies. The
result of such experience – conducted in the context of round robin scheduling – is provided in Fig. 10. As suspected, the overhead increases as the timer period decreases, since interrupts become more frequent. This means that when time resolution is a major concern, a considerable overhead is inevitable. This is true for any time driven scheduling scheme. Including fixed priority scheduling.

Figure 10 – Overhead vs. Time-Slice

The notion that system overload can take a considerable magnitude in time-driven fixed priority scheduling is an important conclusion. Another important conclusion is that $C_{out}$ has a considerable impact upon fixed priority application tasks that execute for a short time, since $C_{out}$ is greater in fixed priority scheduling than in static scheduling. Therefore, it is not surprising to find an overhead greater than 15% in a fixed priority schedule.

It is worth noting than in fixed priority scheduling – and according to the rate monotonic algorithm – high priority tasks tend to have a short execution time and are thus very prone to system overhead. Yet, high priority application tasks tend to meet their deadlines by a comfortable margin. As a consequence, system overhead does not usually make a high priority application task to miss its deadline.

However, the system overhead that impacts the execution of high priority application tasks also makes low priority application tasks to delay their executions. Therefore low priority tasks are also very prone to system overhead. Moreover, since low priority tasks tend to meet their deadlines by a short margin, if any, one finds that these tasks are much more prone to system overhead than high priority tasks. A set of experiments that illustrate this important conclusion can be found in [17].

5 Conclusions

The paper has shown that the overhead introduced by system tasks upon the timeliness of application tasks is not negligible even for very simple real-time systems. Therefore, augment scheduling models from where control engineers can confidently analyse a real-time schedule are required. These models were developed in this paper for the most important single processor scheduling schemes used in practice. Techniques for quantifying the overhead of system tasks upon the application tasks were also presented. Consequently, the authors of the paper feel that it has contributed to improve the practice of real-time systems in control applications. Also felt is that a similar research must be carried out in the context of distributed systems. This is because distributed systems are widely used in control applications and a larger set of system parameters – including those related to real-time communications – introduce overheads and blocking conditions upon the application tasks.

References
