Automazione e Strumentazione, Anno LIV, N. 1, pp. 88-95, Gennaio 2006.

Real-time Operating System Support for Energy-Aware Computing

Giorgio C. Buttazzo
University of Pavia, Italy
Email: buttazzo@unipv.it

KEYWORDS: real-time systems, energy-aware computing, dynamic voltage scaling, scheduling anomalies, embedded systems.

Abstract

Most of today's embedded systems (e.g., cell phones, cameras, sensor networks, smart toys, portable multimedia players, GPS-based navigators, etc.) are powered by batteries and are required to operate as long as possible with a desired level of performance. To balance speed versus power consumption, most of today's microprocessors can work under different voltage levels. In such a way, the performance of a system can be degraded to achieve a longer battery duration, or it can be increased when the battery level is high. Unfortunately, however, in the presence of timing and resource constraints, the performance of a real-time system does not always improve as the speed of the processor is increased. Similarly, when reducing the processor speed, the quality of the delivered service may not always degrade as expected.

This paper presents the potential problems that may arise in a voltage-controlled real-time system and proposes an approach that allows to develop real-time applications, whose performance can be scaled in a controlled fashion as a function of the processor speed.

1 Introduction

The number of embedded computing systems that are operated by battery is constantly increasing. Examples include cell phones, wearable computers, cameras, sensor networks, portable multimedia players, GPS-based navigators, video games, and smart toys. Most of such systems must run under real-time constraints, which determine the quality of service delivered to the user. An important issue in these systems is the possibility of controlling their energy consumption, which directly affects their lifetime, as well as their performance.

Unfortunately, achieving high performance and long lifetime are contrasting objectives. In fact, as illustrated in Figure 1, a high performance requires the system to operate at a high speed, which means high power consumption, whereas, in order to last long, the system has to consume low energy, which means operating at low power.

Such a dependency can be used by a high-level strategy to balance performance versus lifetime depending on the current application needs. Thus, for example the system performance can be degraded to achieve a longer battery duration, or it can be increased when the battery level is high.

In a computer system, the power consumption is related to the voltage at which the circuits operate according to an increasing convex function, whose precise form depends on the specific technology. For example, in CMOS circuits, the power consumption due to dynamic switching dominates the power lost to static leakage [12, 19] and the dynamic portion P_d of power consumption is given by

$$P_d = \alpha_T \cdot C_{load} \cdot f_C \cdot V_{dd}^2, \tag{1}$$

where α_T is the activity factor expressing the amount of switching, C_{load} is the capacitance load, f_C is the clock frequency, and V_{dd}^2 is the supply voltage. However, the voltage also affects the maximum frequency at

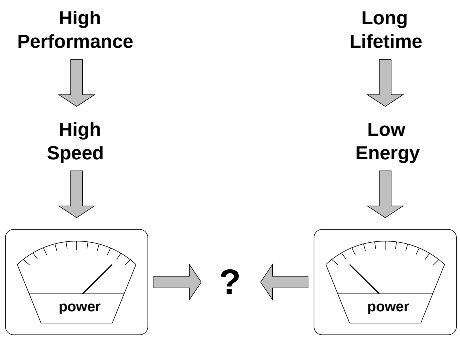

Figure 1. High Performance and long lifetime are contrasting objectives, which may change dynamically during system operation.

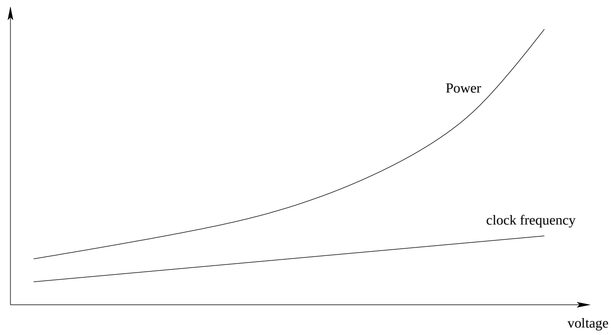

Figure 2. Power consumption and maximum clock frequency as a function of the supply voltage.

which the processor clock can run. In particular, circuit delay depends on the supply voltage as

$$D = k\frac{V_{dd}}{(V_{dd} - V_t)^2}, \tag{2}$$

where k is a constant and V_t is the threshold voltage (i.e., the minimum voltage that can be supplied to the processor allowing full and correct functionality) [3].

Equation (1) and (2) express that reducing the supply voltage can achieve a quadratic power saving at the expense of a roughly linear frequency reduction, as illustrated in Figure 2.

Hence, the amount of energy (power x time) consumed by a portable system can be controlled through the speed and voltage at which the processor operates [14]: we could decide to operate with a high performance for a short period of time, or with a lower performance for a longer duration, or select a suitable intermediate behavior. To exploit such a possibility, most of the current processors are designed to work under different voltage levels, thus enabling applications to run at different speeds.

When increasing the speed, we would expect all the application tasks to finish earlier, in order to improve system's performance. Unfortunately this is not always the case. In [17], Graham showed that several scheduling anomalies may arise when running real-time applications on multiprocessor systems. When tasks share mutually exclusive resources, such anomalies may also arise in a uniprocessor system, as it will be shown in the next section.

Conversely, when voltage is decreased to save energy consumption, we would like the application to run slower in a controlled fashion, where all tasks increase their response times according to some predefined strategy (e.g., depending on their priority level). For reasons similar to the ones described above, this may not always be achieved in the presence of shared resources.

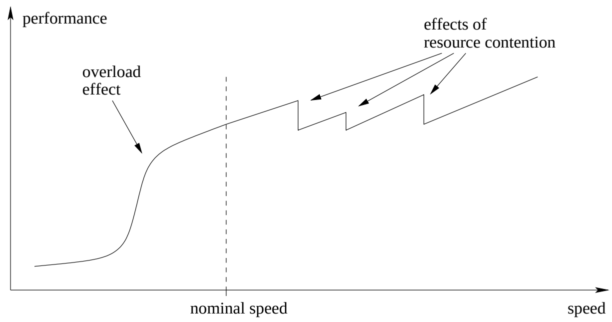

Figure 3. System performance does not monotonically increase with the processing speed due to overload conditions and resource contention.

In addition, when the processor speed is decreased, all tasks increase their computation time, so the processor may experience an overload condition. If the overload is permanent, then the application behavior may be quite unpredictable. Such an apparently anomalous behavior between performance and speed is illustrated in Figure 3.

The problem of achieving scalable applications in processors with variable speed has recently been addressed by some authors. Al Mok [24] illustrated the potential problems that can occur in a real-time system with variable speed when tasks are non preemptive, but no solution has been proposed to achieve scalability.

Yao et al. [29] described an optimal off-line scheduling algorithm to minimize the total energy consumption while meeting all timing constraints, but no on-line voltage change is assumed. Non-preemptive power-aware scheduling is investigated in [18].

The problem of minimizing the energy consumption in a set of periodic tasks with different power consumption characteristics has been solved by Aydin et al. [3], who proposed an algorithm to find the optimal processor speed for each task. However, tasks are assumed to be independent.

Aydin et al. [4] investigated the problem of scheduling hard real-time tasks using dynamic voltage scaling and proposed an algorithm to compute the optimal processor speed which allows to minimize energy consumption.

In [23], Melhem at al. proposed several scheduling techniques to reduce energy consumption of real-time applications in power-aware operating systems, but the scalability problem is not considered.

In this paper, we propose a computational model which allows to achieve scalability during voltage changes, in order to run real-time applications whose performance can be scaled as a function of the processor speed.

The rest of the paper is organized as follows: Section 2 states our terminology and notation. Section 3 introduces the problem to be solved and presents some scheduling anomalies that may arise when running real-time applications at different speeds. Section 4 illustrates the task model. Section 5 presents a kernel communication mechanism which allows data sharing among periodic that preserves scalability. Section 6 describes a technique to cope with permanent overload conditions caused by speed reduction, so preventing unpredictable performance degradations. Finally, Section 7 states our conclusions and future work.

2 Terminology and notation

Throughout the paper, a task set is denoted as $\mathcal{T}$ and consists of n tasks. A generic task τ_i is characterized by a potentially infinite sequence of instances (jobs), with periodic or aperiodic release times. Each job of task τ_i executes for a maximum execution time C_i and has to finish within a relative deadline D_i, expressed with respect to the release time. The specific task model will be presented in Section 4, after introducing the problems to be solved.

To simplify the comparison between schedules executed at different clock frequencies, within a range $[f_{min}, f_{max}]$, all the quantities of interest (computation time, utilization, etc.) will be expressed as a function of speed, defined as the normalized frequency $S = f/f_{max}$. Hence, the validity range for the normalized speed is $[S_{min}, S_{max}]$, where $S_{min} = f_{min}/f_{max}$ and $S_{max} = 1$.

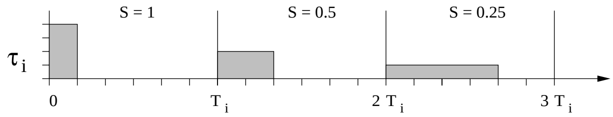

Figure 4. Notation used for representing task executions at different speeds.

For the sake of clarity, task execution times are considered to be inversely proportional to the processor speed; hence, they are modeled as $C_i(S) = C_i(1)/S$, where $C_i(1)$ is the task execution time at the maximum processor speed. However, it is worth noting that all the problems and solutions presented in this paper also apply to a different execution time model.

Finally, in all scheduling examples, the processor speed is represented on the vertical axis. For example, Figure 4 illustrates the execution of three consecutive jobs of a tasks τ_i, running at speed $S = 1$, $S = 0.5$, and $S = 0.25$, respectively.

3 Problem statement

This section illustrates the problems that may arise under specific circumstances when executing a set of real-time tasks in a processor with variable speed. Such problems prevent controlling the performance of a real-time application as a function of the voltage, since a task could even increase its response time when executed at a higher speed. Typically, such scheduling anomalies arise when tasks share mutually exclusive resources or are handled by non-preemptive scheduling policies.

Figure 5 illustrates a simple example where two tasks, τ_1 and τ_2, share a common resource. Task τ_1 has a higher priority, arrives at time $t = 2$ and has a relative deadline $D_1 = 7$. Task τ_2, having lower priority, arrives at time $t = 0$ and has a relative deadline $D_2 = 23$. Suppose that, when the tasks are executed at a certain speed S_1, τ_1 has a computation time $C_1 = 6$, (where 2 units of time are spent in the critical section), whereas τ_2 has a computation time $C_2 = 18$ (where 12 units of time are spent in the critical section). As shown in Figure 5a, if τ_1 arrives just before τ_2 enters its critical section, it is able to complete before its deadline, without experiencing any blocking. However, if the same task set is executed at a double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in Figure 5b. This happens because, when τ_1 arrives, τ_2 already granted its resource, causing an extra blocking in the execution of τ_1, due to mutual exclusion.

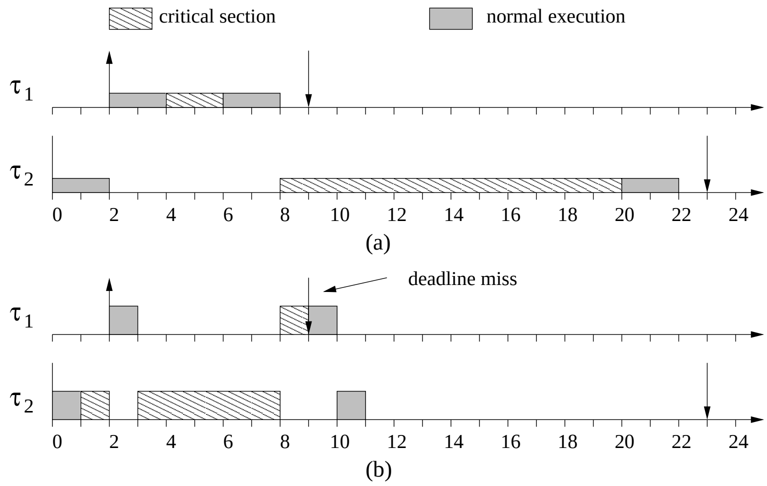

Figure 5. Scheduling anomaly in the presence of resource constraints: task τ_1 meets its deadline when the processor is executing at a certain speed S_1 (a), but misses its deadline when the speed is doubled (b).

Figure 6 illustrates another anomalous behavior occurring in a set of three real-time tasks, τ_1, τ_2 and τ_3, running in a non-preemptive fashion. Tasks are assigned a fixed priority proportional to their relative deadline, thus τ_1 is the task with the highest priority and τ_3 is the task with the lowest priority. As shown in Figure 6a, when tasks are executed at speed S_1, τ_1 has a computation time $C_1 = 2$ and completes at time $t = 6$. However, if the same task set is executed with double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in Figure 6b. This happens because, when τ_1 arrives, τ_3 already started its execution and cannot be preempted (due to the non-preemptive mode).

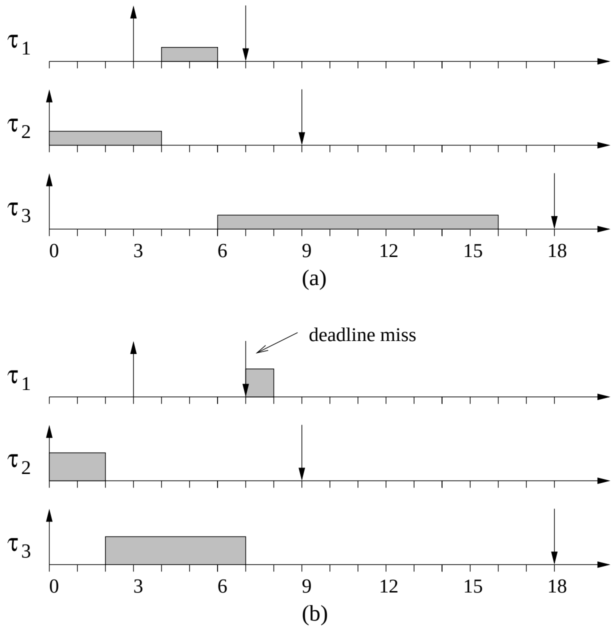

Figure 6. Scheduling anomaly in the presence of non-preemptive tasks: task τ_1 meets its deadline when the processor is executing at speed S_1 (a), but misses its deadline when the speed is doubled (b).

It is worth observing that a set of non preemptive tasks can be considered as a special case of a set of tasks sharing a single resource (the processor) for their entire execution. According to this view, each task executes as it were inside a big critical section with a length equal to the task computation time. Once a task starts executing, it behaves as it were locking a common semaphore, thus preventing all the other tasks from taking the processor.

The following example illustrates the negative effects of a permanent overload condition caused by a speed reduction. In this case, decreasing the processor speed degrades the system's performance in an uncontrolled fashion.

Figure 7 illustrates an example with three tasks, τ_1, τ_2, and τ_3, in which the processor speed is decreased by a factor of 2. Figure 7a shows a feasible schedule produced by the Rate Monotonic (RM) algorithm [22] when the processor runs at a given speed S_1, where the tasks have computation times $C_1 = 2$, $C_2 = 2$, and $C_3 = 4$, respectively. Figure 7b shows the schedule obtained by RM when the processor speed is reduced by half, $S_2 = S_1/2$, so that all computation times are doubled. In this case, a speed reduction generates a permanent overload, which causes τ_2 to miss its deadline and prevents τ_3 to execute.

From the examples shown in this section, it is clear that, in order to achieve scalability as a function of speed, tasks have to be fully preemptive and cannot block on shared resources. In Section 5 we present a communication mechanism which allows tasks to exchange data asynchronously, without blocking on mutually exclusive buffers. Moreover, to avoid the negative effects of a permanent overload caused by a speed reduction, tasks periods need to be specified with some degree of flexibility, so that they can be resized to handle the overload condition. As an overload is detected, period adaptation can be performed using different methodologies.

For example, rate adaptation can be efficiently performed using the elastic task model [8, 10], according to

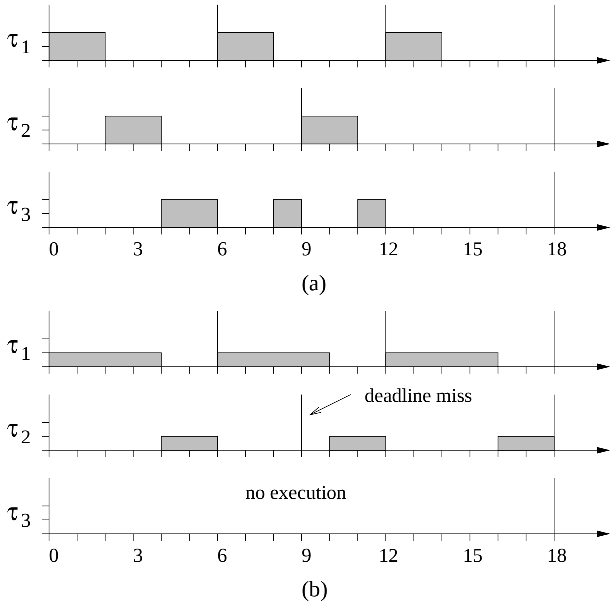

Figure 7. Effects of a permanent overload due to a speed reduction. In case (b) the processor is running at half speed with respect to case (a).

which task utilizations are treated like springs that can adapt to a given workload through period variations. The advantage of the elastic model with respect to the other methods proposed in the literature is that a new period configuration can easily be determined on line as a function of the elastic coefficients, which can be set to reflect tasks' importance. Once elastic coefficients are defined based on some design criterion, periods can be quickly computed on line depending on the current workload and the desired load level. The elastic model will be presented in the next section, whereas the elastic compression algorithm will be briefly recalled in Section 6.1.

4 Task model

In summary, the computational model adopted in this work considers a uniprocessor system whose speed S can be controlled as a function of the supplied voltage. An application consists of a set of periodic tasks, each characterized by four parameters: a worst-case computation time $C_i(S)$ (which is a function of the speed) a nominal period T_{i_0} (considered as the desired minimum period), a maximum allowed period $T_{i_{max}}$, and an elastic coefficient E_i. The elastic coefficient specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration: the greater E_i, the more elastic the task. Thus, an elastic task is denoted as:

$$\tau_i(C_i, T_{i_0}, T_{i_{max}}, E_i).$$

From a design perspective, elastic coefficients can be set equal to values which are inversely proportional to tasks' importance. In the following, T_i will denote the actual period of task τ_i, which is constrained to be in the range $[T_{i_0}, T_{i_{max}}]$. Any period variation is always subject to an *elastic* guarantee and is accepted only if there exists a feasible schedule in which all the other periods are within their range. In such a framework, tasks are scheduled by the Earliest Deadline First algorithm [22], according to which tasks with shorter absolute deadline are assigned higher priority. Hence, if $\sum \frac{C_i(S)}{T_{i_0}} \leq 1$, all tasks can be created at the minimum period T_{i_0}, otherwise the elastic algorithm is used to adapt the tasks' periods to T_i such that $\sum \frac{C_i(S)}{T_i} = U_d \leq 1$, where U_d is some desired utilization factor. To simplify the analysis we assume that tasks have a relative deadline equal to their period ($D_i = T_i$).

5 Avoiding blocking through asynchronous buffers

This section describes how blocking on shared resources can be avoided through the use of Cyclical Asynchronous Buffers [11], or CABs, a kind of wait free mechanism which allows tasks to exchange data without forcing a blocking synchronization.

In a CAB, read and write operations can be performed simultaneously without causing any blocking. Hence, a task can write a new message in a CAB while another task is reading the previous message. Mutual exclusion between reader and writer is avoided by means of memory duplication. In other words, if a task τ_W wants to write a new message into a CAB while a task τ_R is reading the current message, a new buffer is created, so that τ_W can write its message without interfering with τ_R. As τ_W finishes writing, its message becomes the most recent one in the CAB. To avoid blocking, the number of buffers that a CAB must handle has to be equal to the number of tasks that use the CAB plus one.

CABs were purposely designed for the cooperation among periodic activities running at different rates, such as control loops and sensory acquisition tasks. This approach was first proposed by Clark [13] for implementing a robotic application based on hierarchical servo-loops, and it is used in the HARTIK kernel [7] and in the SHARK kernel [16] as a basic communication support among periodic hard tasks.

In general, a CAB provides a one-to-many communication channel, which at any instant contains the latest message or data inserted in it. A message is not consumed by a receiving process, but is maintained into the CAB structure until a new message is overwritten. As a consequence, once the first message has been put in a CAB, a task can never be blocked during a receive operation. Similarly, since a new message overwrites the old one, a sender can never be blocked.

Notice that, using such a semantics, a message can be read more than once if the receiver is faster than the sender, while messages can be lost if the sender is faster than the receiver. However, this is not a problem in many control applications, where tasks are interested only in fresh sensory data rather than in the complete message history produced by a sensory acquisition task.

To insert a message in a CAB, a task must first reserve a buffer from the CAB memory space, then copy the message into the buffer, and finally put the buffer into the CAB structure, where it becomes the most recent message. This is done according to the following scheme:

```
buf_pointer = reserve(cab_id);
<copy message in *buf_pointer>
putmes(buf_pointer, cab_id);
```

Similarly, to get a message from a CAB, a task has to get the pointer to the most recent message, use the data, and then release the pointer. This is done according to the following scheme:

```
mes_pointer = getmes(cab_id);
<use message>
unget(mes_pointer, cab_id);
```

5.1 An example

To better illustrate the CAB mechanism, we describe an example in which a task τ_W writes messages in a CAB, and two tasks, τ_{R_1} and τ_{R_2}, read messages from the CAB. As it will be shown below, to avoid blocking and preserve data consistency, the CAB must contain 4 buffers. Consider the following sequence of events:

- At time t_1, task τ_W writes message M_1 in the CAB. When it finishes, it becomes the most recent data (*mrd*) in the CAB.
- At time t_2, task τ_{R_1} asks the system to read the most recent data in the CAB and receives a pointer to M_1.
- At time t_3, task τ_W asks the system to write another message M_2 in the CAB, while τ_{R_1} is still reading M_1. Hence, a new buffer is reserved to τ_W. When it finishes, M_2 becomes the most recent data in the CAB.

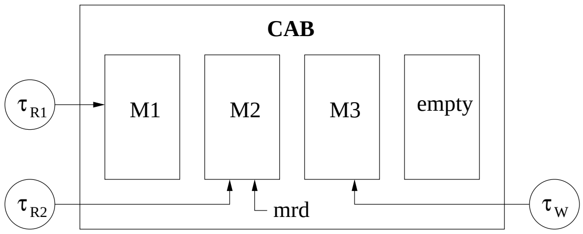

Figure 8. Buffer configuration in the CAB, at time t_5.

- At time t_4, while τ_{R_1} is still reading M_1, τ_{R_2} asks the system to read the most recent data in the CAB and receives a pointer to M_2.
- At time t_5, while τ_{R_1} and τ_{R_2} are still reading, τ_W asks the system to write a new message M_3 in the CAB. Hence, a third buffer is reserved to τ_W. When it finishes, M_3 becomes the most recent data in the CAB.
- At time t_6, while τ_{R_1} and τ_{R_2} are still reading, τ_W asks the system to write a new message M_4 in the CAB. Notice that, in this situation, M_3 cannot be overwritten (being the most recent data), hence a fourth buffer must be reserved to τ_W. In fact, if M_3 is overwritten, τ_{R_1} could ask reading the CAB while τ_W is writing, thus finding the most recent data in an inconsistent state. When τ_W finishes writing M_4 into the fourth buffer, the *mrd* pointer is updated and the third buffer can be recycled if no task is accessing it.
- At time t_7, τ_{R_1} finishes reading M_1 and releases the first buffer (which can then be recycled).
- At time t_8, τ_{R_1} asks the system to read the most recent data in the CAB and receives a pointer to M_4.

Figure 8 illustrates the situation in the example, at time t_5, when τ_W is writing M_3 in the third buffer. Notice that at this time, the most recent data (mrd) is still M_2. It will be updated to M_3 only at the end of the write operation.

6 Rate adaptation under permanent overloads

Section 3 illustrated how the performance can be degraded when a permanent overload occurs due to a speed reduction. To avoid such a negative effect, tasks periods need to be adjusted to remove the overload condition. Rate adaptation can be performed in many ways. For example, Kuo and Mok [20] proposed a load scaling technique to degrade the workload of a system by adjusting the task periods. Tasks were assumed to be equally important and the objective was to minimize the number of fundamental frequencies to improve schedulability under static priority assignments. In [21], Lee, Rajkumar and Mercer proposed a number of policies to dynamically adjust tasks' rates in overload conditions. In [25], Nakajima showed how a multimedia activity can adapt its requirements during transient overloads by scaling down its rate or its computational demand. However, it is not clear how the QoS can be increased when the system is underloaded. In [6], Beccari et al. proposed several policies for handling overload through period adjustment; however, they did not address the problem of increasing the task rates when the processor is not fully utilized.

In this paper, task rate adjustment is performed through the elastic task model [8, 10], according to which task utilizations are treated like springs that can adapt to a given workload through period variations. The advantage of the elastic model with respect to the other methods proposed in the literature is that a new period configuration can easily be determined on line as a function of the elastic coefficients, which can be set to reflect tasks' importance. Once elastic coefficients are defined based on some design criterion, periods can be quickly computed on line depending on the current workload and the desired load level. Moreover, the elastic model can also be used in combination with a feedback mechanism, as done in [9], when system parameters are not known a priori.

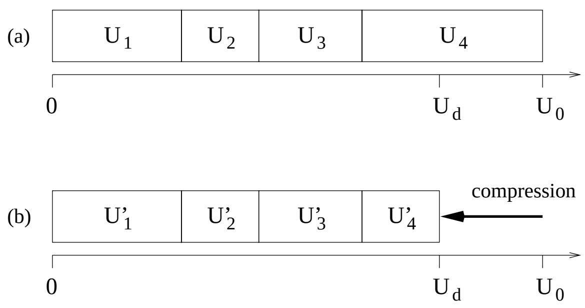

Figure 9. Compressing the utilizations of a set of elastic tasks.

6.1 The elastic approach

Whenever the total processor utilization $U_0 = \sum_{i=1}^{n} \frac{C_i}{T_{i_0}}$ is greater than one (i.e., there is a permanent overload in the system), the utilization of each task needs to be reduced so that the total utilization becomes $U_d = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$. This can be done as in a linear spring system, where springs are compressed by a force F (depending of their elasticity) up to a desired total length. The concept is illustrated in Figure 9.

As shown in [10], in the absence of period constraints (i.e., if $T_{max} = \infty$), the utilization U_i of each compressed task can be computed as follows:

$$\forall i \quad U_i = U_{i_0} - (U_0 - U_d)\frac{E_i}{E_v}. \tag{3}$$

where

$$E_v = \sum_{i=1}^{n} E_i. \tag{4}$$

In the presence of period constraints ($T_i \leq T_{i_{max}}$), however, the problem of finding the values T_i requires an iterative solution. In fact, if during compression one or more tasks reach their maximum period, the additional compression has to affect only to the remaining periods. Thus, at each instant, the set Γ of tasks can be divided into two subsets: a set Γ_f of fixed tasks having maximum period, and a set Γ_v of variable tasks whose period can still be enlarged. Applying the equations to the set Γ_v of variable springs, we have

$$\forall \tau_i \in \Gamma_v \quad U_i = U_{i_0} - (U_{v_0} - U_d + U_f)\frac{E_i}{E_v} \tag{5}$$

where

$$U_{v_0} = \sum_{\tau_i \in \Gamma_v} U_{i_0} \tag{6}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{min}} \tag{7}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{8}$$

If there exist tasks for which $U_i < U_{i_{min}}$, then the period of those tasks has to be fixed at its maximum value $T_{i_{max}}$ (so that $U_i = U_{i_{min}}$), sets Γ_f and Γ_v must be updated (hence, U_f and E_v recomputed), and equation (5) applied again to the tasks in Γ_v. If there exists a feasible solution, that is, if the desired utilization U_d is greater than or equal to the minimum possible utilization $U_{min} = \sum_{i=1}^{n} \frac{C_i}{T_{i_{max}}}$, the iterative process ends when each value computed by equation (5) is greater than or equal to its corresponding minimum $U_{i_{min}}$. In [10] it is shown that, in the worst case, the compression algorithm converges to a solution (if there exists one) in $O(n^2)$ steps, where n is the number of tasks.

The same algorithm can be used to reduce the periods when the overload is over, so adapting task rates to the current load condition to better exploit the computational resources.

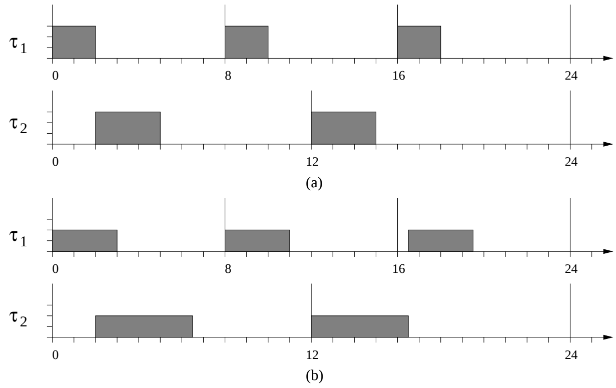

Figure 10. Problems with discrete voltage levels.

6.2 Elastic tasks and energy-aware scheduling

In processors having discrete voltage levels (hence, discrete speeds), the total processor utilization may not be exploited due to the fact that the ideal speed that minimizes energy consumption while guaranteeing timing constraints may not be available. In this case, to ensure schedulability, the actual speed must be set to the closest level higher than the ideal one. However, this means that the processor is underutilized.

For example, consider the situation illustrated in Figure 10, where a processor can only run at three speed levels ($S_1 = 1$, $S_2 = 2/3$, and $S_3 = 1/3$), and a real-time application of two periodic tasks has a total utilization $U = 0.5$ when the processor executes at its maximum speed (see Figure 10a). Clearly, running the task set at a speed $S* = 0.5$ would fully utilize the processor and significantly reduce energy consumption. However, since this speed is not available, the processor must run at speed $S_2 = 2/3$ (higher than $S*$) in order to meet timing constraints. As shown in Figure 10b, when the processor executes at S_2, the total utilization of the task set becomes $U(S_2) = 0.75$, leaving 25% of the processor unused.

In these situations, elastic scheduling can be invoked to exploit the unused utilization and run the application tasks with higher rates, so improving system performance. The schedule resulting by applying this approach is shown is Figure 11a.

As an alternative, elastic scheduling can also be used to reduce energy consumption by allowing the processor to run at a speed lower than the ideal one. For instance, in the example described above, if the speed is set at $S_3 = 1/3$, task periods can be properly enlarged to fully utilize the processor, as shown in Figure 11b. Notice that, without the elastic method, speed S_3 would cause an overload that would degrade the performance in an uncontrolled fashion.

7 Conclusions

In this paper we presented the problems that can occur when running a real-time application in a processor with variable speed. It has been shown that, when tasks share mutually exclusive resources or execute in a non-preemptive fashion, response times could even increase when the processor runs at higher speeds. In addition, when the speed is decreased, a permanent overload could degrade the system's performance in an uncontrolled fashion. Such problems, if not properly handled, would prevent controlling the performance of a real-time system as a function of the voltage and would limit the use of real-time scheduling algorithms for resource optimization (e.g., for meeting timing constraints while minimizing energy consumption).

To address these problems, we proposed a set of mechanisms that can be implemented at the kernel level to develop scalable real-time applications, whose performance can be adjusted in a controlled fashion as a function of the processor speed. In particular, the use of non blocking communication buffers (like the CABs) has two main advantages: it avoids the scheduling anomalies that may arise due to speed variations and allows data exchange among periodic tasks with non harmonic period relations.

To cope with permanent overloads caused by a speed reduction, the elastic scheduling approach provides an efficient method for automatically adjusting the task rates based on a set of coefficients, which can be assigned

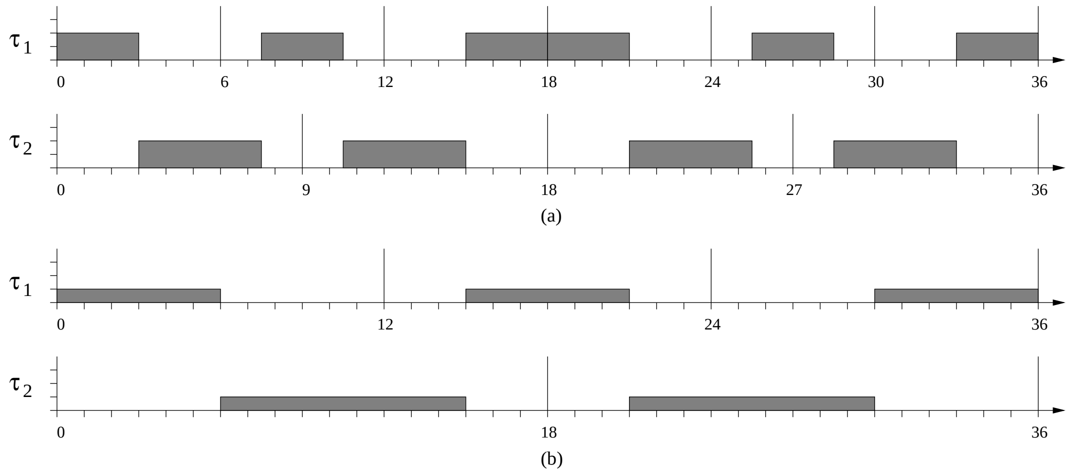

Figure 11. Elastic scheduling can be exploited to improve system performance when processor speed cannot be set at the ideal value (a), or to reduce energy consumption by decreasing periods to prevent overloads.

during the design phase based on task importance. Both methods have been implemented on top of the SHARK kernel [16] and have been experimented in a number of control applications.

In the future we plan to implement these techniques on top of other real-time kernels (e.g., RT-Linux and Linux-RK), as a middleware layer, to provide the basic building blocks for supporting energy-aware real-time applications.

References

[1] N. AbouGhazaleh, D. Moss, B. Childers and R. Melhem, “Toward The Placement of Power Manegement Points in Real Time Applications”, Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP’01), Barcelona, Spain, 2001.

[2] A. Allavena and D. Moss, “Scheduling of Frame-based Embedded Systems with Rechargeable Batteries”, Proceedings of the Workshop on Power Management for Real-Time and Embedded Systems, 2001.

[3] H. Aydin, R. Melhem, D.Moss and Pedro Mejia Alvarez, “Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics”, Proceedings of the Euromicro Conference on Real-Time Systems, Delft, Holland, June 2001.

[4] H. Aydin, R. Melhem, D. Moss, and Pedro Mejia Alvarez, “Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems”, Proceedings of the IEEE Real-Time Systems Symposium, December 2001.

[5] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari, “Scheduling Periodic Task Systems to Minimize Output Jitter,” Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications, Hong Kong, December 1999.

[6] G. Beccari, S. Caselli, M. Reggiani, F. Zanichelli, “Rate Modulation of Soft Real-Time Tasks in Autonomous Robot Control Systems,” *IEEE Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, June 1999.

[7] G. C. Buttazzo, ”HARTIK: A Real-Time Kernel for Robotics Applications”, Proceedings of the 14th IEEE Real-Time Systems Symposium, Raleigh-Durham, December 1993.

[8] G. C. Buttazzo, G. Lipari, and L. Abeni, “Elastic Task Model for Adaptive Rate Control,” *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 286-295, December 1998.

[9] G. C. Buttazzo and L. Abeni, “Adaptive Rate Control through Elastic Scheduling,” *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[10] G. C. Buttazzo, G. Lipari, M. Caccamo, L. Abeni, "Elastic Scheduling for Flexible Workload Management," *IEEE Transactions on Computers*, Vol. 51, No. 3, pp. 289-302, March 2002.

[11] G. C. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications - Second Edition, Springer, 2005.

[12] A. Chandrakasan and R. Brodersen, *Low Power Digital CMOS Design*, Kluwer Academic Publishers, 1995.

[13] D. Clark, "HIC: An Operating System for Hierarchies of Servo Loops," *Proceedings of IEEE International Conference on Robotics and Automation*, 1989.

[14] E. Chan, K. Govil, and H. Wasserman, "Comparing Algorithms for Dynamic Speed-setting of a Low-Power CPU", *Proceedings of the First ACM International Conference on Mobile Computing and Networking* (MOBICOM 95), November 1995.

[15] M.L. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," *Information Processing*, 74, North-Holland, Publishing Company, 1974.

[16] P. Gai, L. Abeni, M. Giorgi, G. Buttazzo, "A New Kernel Approach for Modular Real-Time Systems Development," *IEEE Proceedings of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[17] R. L. Graham: "Bounds on the Performance of Scheduling Algorithms," Chapter 5 in Computer and Job Scheduling Theory, John Wiley and Sons, pp. 165-227, 1976.

[18] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava, "Power Optimization of Variable Voltage Core-Based Systems", *Proceedings of the 35th Design Automation Conference*, 1998.

[19] I. Hong, G. Qu, M. Potkonjak, and M.B. Srivastava, "Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors", *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[20] T.-W. Kuo and A. K, Mok, "Load Adjustment in Adaptive Real-Time Systems," *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.

[21] C. Lee, R. Rajkumar, and C. Mercer, "Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach," *Proceedings of Multimedia Japan 96*, April 1996.

[22] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment," *Journal of the ACM* 20(1), 1973, pp. 40–61.

[23] R. Melhem, N. AbouGhazaleh, H. Aydin and D. Mosse, "Power Management Points in Power-Aware Real-Time Systems", In Power Aware Computing, ed. by R. Graybill and R. Melhem, Plenum/Kluwer Publishers, 2002.

[24] A. Mok, "Scalability of real-time applications," keynote address at the 7th International Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, December 2000.

[25] T. Nakajima, "Resource Reservation for Adaptive QOS Mapping in Real-Time Mach," *Sixth International Workshop on Parallel and Distributed Real-Time Systems*, April 1998.

[26] M. Spuri, and G.C. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline Scheduling", *Proceedings of IEEE Real-Time System Symposium*, San Juan, Portorico, December 1994.

[27] M. Spuri, G.C. Buttazzo, and F. Sensini, "Robust Aperiodic Scheduling under Dynamic Priority Systems", *Proc. of the IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[28] M. Spuri and G.C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-Time Systems*, 10(2), 1996.

[29] F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy," *IEEE Annual Foundations of Computer Science*, pp. 374-382, 1995.

[30] D. Zhu, R. Melhem, and B. Childers, "Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems", *Proceedings of the IEEE Real-Time Systems Symposium*, December 2001.