# Achieving Scalability in Real-Time Systems

*Giorgio Buttazzo*
Scuola Superiore Sant'Anna, Pisa

**Running real-time applications with a variable-speed processor can result in scheduling anomalies and permanent overloads. A proposed computational model varies task response times continuously with processor speed, enabling the system to predictably scale its performance during voltage changes.**

Most embedded computing systems—including cell phones, wearable computers, cameras, sensor networks, portable multimedia players, GPS-based navigation devices, video game consoles, and smart toys—are powered by batteries. Energy consumption is a critical aspect of these systems, directly affecting both performance and lifetime.

Unfortunately, high performance and a long battery life are conflicting objectives: To achieve high performance, the system must operate at high speed, which consumes more power, whereas achieving a long battery life requires the system to consume less energy, which means operating at lower power.

A high-level strategy can exploit this dependency depending on current application needs. For example, the system can degrade performance to achieve longer battery duration or it can increase performance at the expense of higher energy consumption.

In the presence of timing and resource constraints, however, a real-time system's performance does not always improve as processor speed increases. Similarly, when reducing processor speed, the delivered service's quality might not always degrade as expected. The "Speed-Performance Tradeoff Anomalies" sidebar explains this dilemma in more detail.

Although many researchers have looked at reducing energy consumption of real-time applications,[1-6] the problem of scaling performance with speed variations requires further exploration. A proposed computational model varies task response times continuously with processor speed, enabling the system to predictably scale its performance during voltage changes. This model implements two key mechanisms at the kernel level:

- Nonblocking communication buffers prevent the scheduling anomalies caused by speed variations and allow data exchange among periodic tasks with nonharmonic period relations.
- To cope with permanent overloads caused by speed reductions, an elastic task-scheduling approach automatically adjusts task rates based on a set of coefficients, which the developer can assign during the design phase based on task importance.

To simplify comparison of task schedules executed at different clock frequencies, within a range $[f_{min}, f_{max}]$, all quantities of interest, such as computation time and utilization, are expressed as a function of speed, defined as the normalized frequency $S = f/f_{max}$. Thus, the validity range for normalized speed is $[S_{min}, S_{max}]$, where $S_{min} = f_{min}/f_{max}$ and $S_{max} = 1$. In addition, for clarity, task execution times are considered to be inversely proportional to processor speed—they are thus modeled as $C_i(S) = C_i(1)/S$, where $C_i(1)$ is the task execution time at the maximum processor speed. However, the proposed solutions do not depend on that condition and apply to different execution time models. Finally, in all scheduling illustrations, the vertical axis represents processor speed.

## REAL-TIME TASK-SCHEDULING PROBLEMS

Several problems can occur when a real-time application runs on a variable-speed processor. When tasks share access to mutually exclusive resources or execute

## Speed-Performance Tradeoff Anomalies

In a computer system, 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 complementary metal-oxide semiconductor circuits, reducing the supply voltage can achieve a quadratic power savings at the expense of a roughly linear frequency reduction.[1-3]

The speed and voltage at which the processor operates can control the amount of energy (power × time) that a portable system consumes. Most current processors are thus designed to work at different voltage levels, enabling applications to run at different speeds.

Theoretically, increasing processor speed should cause application tasks to finish earlier and thereby improve system performance. However, this is not always the case. Ronald L. Graham[4] showed that several scheduling anomalies can arise when running real-time applications on multiprocessor systems. When tasks share mutually exclusive resources, such anomalies can also appear in a uniprocessor system due to blocking factors.

Conversely, decreasing voltage to save energy consumption should cause an application to run slower in a controlled fashion, wherein all tasks increase their response times according to some predefined strategy—for example, depending on their priority level. Again, this cannot be achieved when the computational activities have interdependencies due to synchronization conditions or shared resources.

In addition, reducing processor speed increases the application tasks' computation time, which, as Figure A shows, can lead to resource contention and processor overload. If the overload is permanent, the application can behave quite unpredictably and the system can experience abrupt performance degradation.

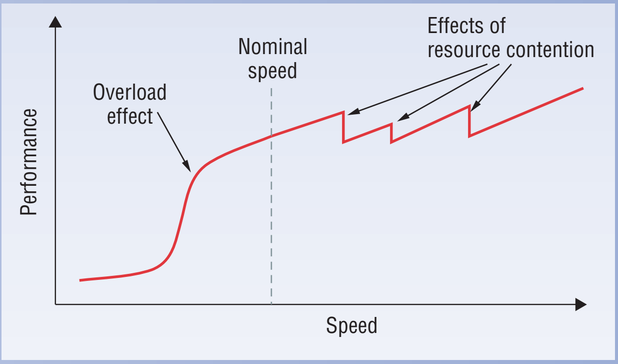

*Figure A. System performance does not increase monotonically with processing speed due to overload conditions and resource contention.*

### References

1. A.P. Chandrakasan and R.W. Brodersen, *Low Power Digital CMOS Design*, Kluwer Academic, 1995.
2. I. Hong et al., "Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors," *Proc. 19th IEEE Real-Time Systems Symp.*, IEEE CS Press, 1998, pp. 178-187.
3. H. Aydin et al., "Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics," *Proc. 13th Euromicro Conf. Real-Time Systems*, IEEE CS Press, 2001, p. 225.
4. R.L. Graham, "Bounds on the Performance of Scheduling Algorithms," chapt. 5, *Computer and Job-Shop Scheduling Theory*, E.G. Coffman Jr., ed., John Wiley & Sons, 1976, pp. 165-227.

in a nonpreemptive fashion, response times can actually increase when the processor runs at higher speeds. In addition, decreasing speed can cause a permanent overload that degrades system performance. Such problems, if not properly handled, make it impossible to control a real-time system's performance as a function of the voltage and limit the use of real-time scheduling algorithms for resource optimization.

## Shared resources

Figure 1 illustrates a simple example in which two tasks, $\tau_1$ and $\tau_2$, share a common resource. Task $\tau_1$ has a higher priority, arrives at time $t = 2$, and has a relative deadline $D_1 = 7$. Task $\tau_2$, 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 $S_1$, $\tau_1$ has a computation time $C_1 = 6$

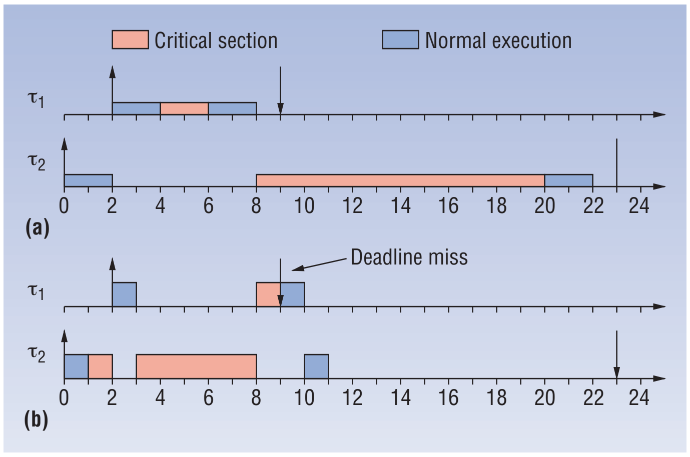

*Figure 1. Task-scheduling anomaly in the presence of resource constraints. (a) Task $\tau_1$ meets its deadline when the processor is executing at a certain speed $S_1$, but (b) misses its deadline when the speed is doubled.*

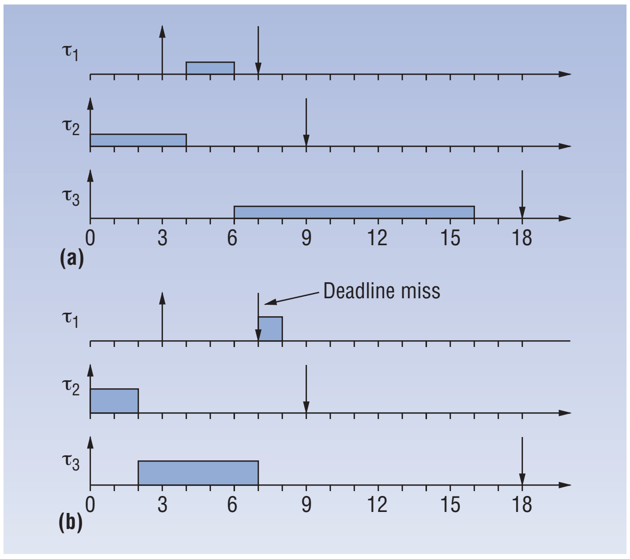

***Figure 2. Task-scheduling anomaly in the presence of nonpreemptive tasks. (a) Task* $\tau_1$ *meets its deadline when the processor is executing at speed* $S_1$*, but (b) misses its deadline when the speed is doubled.***

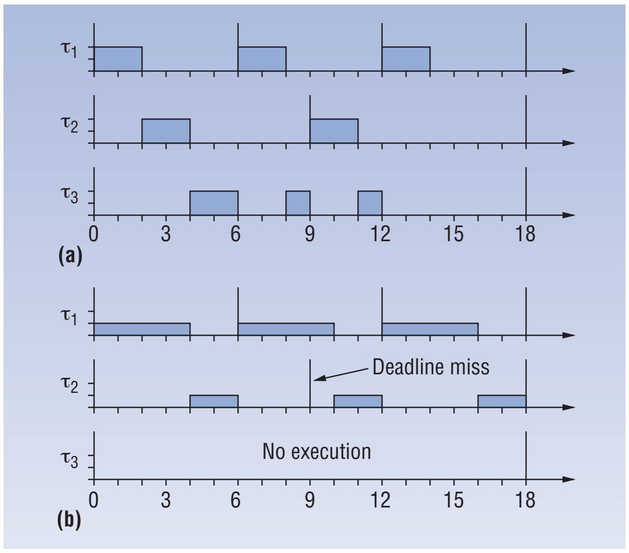

***Figure 3. Effects of a permanent overload due to processor speed reduction. In case (b), the processor is running at half-speed with respect to case (a).***

(with 2 units of time spent in the critical section), whereas $\tau_2$ has a computation time $C_2 = 16$ (with 12 units of time spent in the critical section).

As Figure 1a shows, if $\tau_1$ arrives just before $\tau_2$ enters its critical section, it can complete before its deadline, without experiencing any blocking. However, if the same task set is executed at speed $S_2 = 2S_1$, $\tau_1$ misses its deadline, as shown in Figure 1b. This occurs because, when $\tau_1$ arrives, $\tau_2$ has already granted its resource, causing extra blocking in the execution of $\tau_1$ due to mutual exclusion.

### Nonpreemptive tasks

Figure 2 illustrates anomalous behavior occurring in a set of three real-time tasks—$\tau_1$, $\tau_2$, and $\tau_3$—running in nonpreemptive mode. Tasks are assigned a fixed priority proportional to their relative deadline; thus, $\tau_1$ is the task with the highest priority and $\tau_3$ is the task with the lowest priority.

As Figure 2a shows, when tasks are executed at speed $S_1$, $\tau_1$ has a computation time $C_1 = 2$ and completes at time $t = 6$. However, if the same task set is executed at speed $S_2 = 2S_1$, $\tau_1$ misses its deadline, as Figure 2b shows. This happens because, when $\tau_1$ arrives, $\tau_3$ has already started its execution and cannot be preempted.

A set of nonpreemptive tasks can be considered as a special case of a set of tasks sharing a single, mutually exclusive resource for their entire execution. Each task executes as if it were inside a long critical section with a length equal to the task computation time. Once a task starts executing, it behaves as if it were locking a common semaphore, thus preventing the processor from executing all other tasks.

### Permanent overload

Figure 3 shows the negative effects of a permanent overload condition, caused by a speed reduction, in a set of periodic tasks $\tau_1$, $\tau_2$, and $\tau_3$. Figure 3a shows the feasible schedule produced by the rate monotonic (RM) algorithm[7] 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 3b shows the schedule that RM obtains when the processor speed is $S_2 = S_1/2$, such that all computation times are doubled. In this case, a speed reduction generates a permanent overload that causes $\tau_2$ to miss its deadline and prevents $\tau_3$ from executing.

Note that a scheduling algorithm based on absolute deadlines, such as earliest deadline first (EDF),[7] would not prevent $\tau_3$ from executing and would react to overloads by delaying all tasks' executions more evenly. Previous work compared RM and EDF in detail for different scenarios.[8]

## AVOIDING SHARED-RESOURCE BLOCKING

Cyclical asynchronous buffers[9] prevent scheduling anomalies due to speed variations and allow data exchange among periodic tasks with nonharmonic period relations. CABs are a kind of wait-free mechanism that let tasks perform read and write operations simultaneously via memory duplication. If a task $\tau_W$ wants to write a new message into a CAB while a task $\tau_R$ is reading the current message, the system creates a new buffer so that $\tau_W$ can write its message without interfering with $\tau_R$. As $\tau_W$ finishes writing, its message becomes the most recent one in the CAB. To avoid blocking, the number of buffers that a CAB must handle must equal the number of tasks that use the CAB plus one.

CABs were purposely designed to enable cooperation among periodic activities running at different rates, such as control loops and sensory acquisition tasks. The soft/hard real-time kernel (SHaRK)[10] uses this mechanism as a basic communication support for periodic tasks.

In general, a CAB provides a one-to-many communication channel that at any instant contains the latest message inserted into it. A message is not consumed by a receive process, but rather is maintained in the CAB until a new message overwrites it. Consequently, once the system has put the first message into 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.

Note that, using such 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 that a sensory acquisition task produces.

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 accomplished as follows:

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

Similarly, to retrieve a message from a CAB, a task must get the pointer to the most recent message, use the data, and then release the pointer. The following code executes this action:

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

## COPING WITH PERMANENT OVERLOADS

To avoid the negative effects of a permanent overload caused by a processor speed reduction, the user must specify task periods with some degree of flexibility so that the system can resize them to remove the overload condition. The elastic task model, described in the "Elastic Task Model" sidebar, provides an efficient way to accomplish task-rate adaptation.

### Elastic tasks

Using the elastic approach, the EDF algorithm[7] assigns a higher priority to tasks with a shorter absolute deadline. Thus, if the nominal workload

$$U_0 = \sum_{i=1}^{n} \frac{C_i}{T_{i_0}}$$

is greater than 1 (there is a permanent overload in the system), the system must reduce the utilization of each task so that the total utilization becomes

$$U_d = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

This is akin to a linear spring system, in which a force $F$ compresses springs (depending on their elasticity) up to a desired total length. Figure 4 illustrates compression of the utilizations of a set of elastic tasks.

In the absence of period constraints (that is, if $T_{\max} = 8$), the utilization $U_i$ of each compressed task can be computed as follows[11]:

## Elastic Task Model

Given a uniprocessor system whose speed $S$ can be controlled as a function of the supplied voltage, the elastic task model[1] offers an efficient way to automatically adjust task rates in real-time applications. This model treats task utilizations like springs that can adapt to a given workload through period variations.

Unlike other methods, the elastic model makes it easy to determine a new period configuration dynamically as a function of elastic coefficients that reflect the tasks' importance. Once the coefficients are defined based on some design criterion, periods can be quickly computed depending on the current workload and desired load level.

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 $T_{i_0}$, the desired minimum period;
- a maximum allowed period $T_{i_{\max}}$; and
- an elastic coefficient $E_i$.

The elastic coefficient specifies the task's ability 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 that are inversely proportional to the tasks' importance. Thus, $T_i$ denotes the actual period of task $\tau_i$, which is constrained to be in the range $[T_{i_0}, T_{i_{\max}}]$; $U_{i_0}$ denotes the nominal utilization of task $\tau_i$, that is, $U_{i_0} = C_i/T_{i_0}$; and $U_{i_{\min}}$ denotes its minimum utilization, that is, $U_{i_{\min}} = C_i/T_{i_{\max}}$.

**Reference**

1. G.C. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control," *Proc. IEEE Real-Time Systems Symp.*, IEEE CS Press, 1998, pp. 286-295.

$$\forall i \;\; U_i = U_{i_0} - (U_0 - U_d)\frac{E_i}{E_v}, \tag{1}$$

where

$$E_v = \sum_{i=1}^{n} E_i. \tag{2}$$

In the presence of period constraints ($T_i = T_{imax}$), however, finding the value $T_i$ requires an iterative solution. In fact, if one or more tasks reach their maximum period during compression, the additional compression can only affect the remaining periods.

Thus, at each instant, the set $\Gamma$ of tasks can be divided into two subsets: a set $\Gamma_f$ of fixed tasks having a maximum period, and a set $\Gamma_v$ of variable tasks whose period can still be enlarged. Applying Equations 1 and 2 to the set $\Gamma_v$ of variable springs yields the following result:

$$\forall \tau_i \in \Gamma_v \;\; U_i = U_{i_0} - (U_{v_0} - U_d + U_f)\frac{E_i}{E_v}, \tag{3}$$

where

$$U_{v_0} = \sum_{\tau_i \in \Gamma_v} U_{i_0}, \tag{4}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{\min}}, \text{ and} \tag{5}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{6}$$

If there are tasks for which $U_i < U_{imin}$, the system must fix the period of those tasks at its maximum value $T_{i_{\max}}$ (so that $U_i = U_{i_{\min}}$), sets $\Gamma_f$ and $\Gamma_v$ must be updated (thus, $U_f$ and $E_v$ recomputed), and Equation 3 must be applied again to the tasks in $\Gamma_v$. If there is 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 that Equation 3 computes is greater than or equal to its corresponding minimum $U_{i_{\min}}$. In the worst case, the compression algorithm converges to a solution (if one exists) in $O(n^2)$ steps, where $n$ is the number of tasks.[11]

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

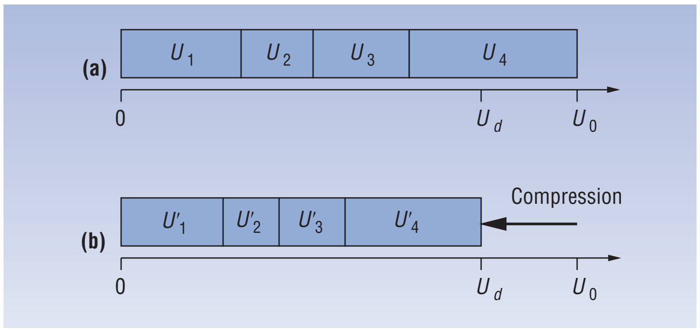

*Figure 4. Compressing the utilizations of a set of elastic tasks. (a) The system is overloaded. (b) Enlarging task periods reduces the total utilization to the desired value.*

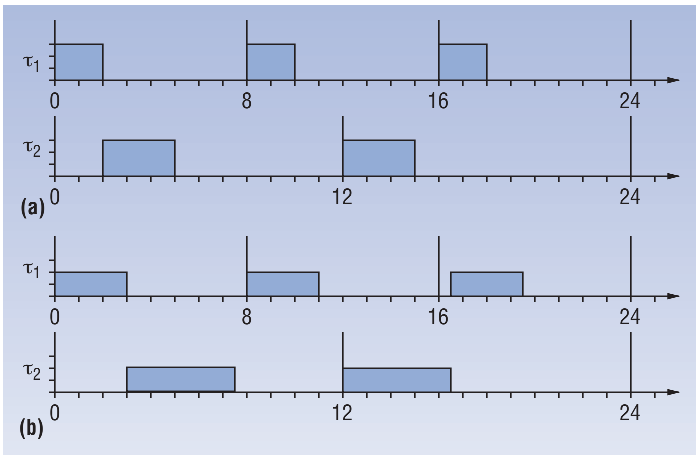

*Figure 5. Problems with discrete voltage levels. (a) Only half of the processor is utilized at its maximum speed, but the optimal speed ($S* = 0.5$) is not available. (b) The task set must run at speed $S_2 = 2/3$, wasting 25 percent of the processor.*

## Coping with discrete voltage levels

Processors with discrete voltage levels (and thus discrete speeds) might not exploit total processor utilization because the ideal speed that minimizes energy consumption while guaranteeing timing constraints might not be available. To ensure schedulability in such processors, the actual speed must be set to the closest level higher than the ideal speed. However, this means that the processor is underutilized.

Consider the situation illustrated in Figure 5 in which the processor can only run at three speed levels ($S_1 = 1$, $S_2 = 2/3$, and $S_3 = 1/3$), and the real-time application consists of two periodic tasks with a total utilization $U = 0.5$ when the processor executes at its maximum speed (Figure 5a). Clearly, running the task set at the optimal 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 $S*$) to meet timing constraints. As Figure 5b shows, when the processor executes at $S_2$, the total utilization of

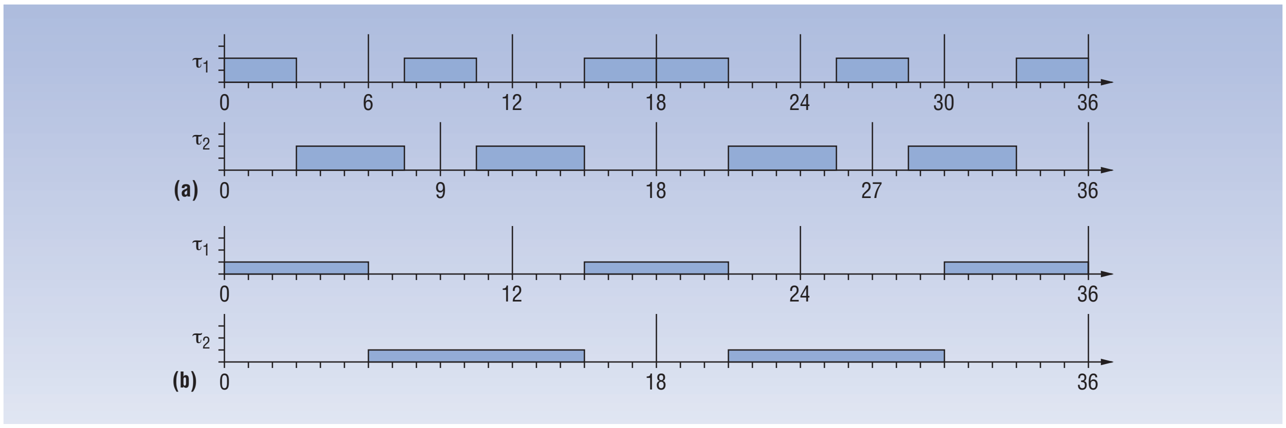

*Figure 6. Elastic scheduling. The system can exploit elastic scheduling (a) to shrink task periods and improve performance when it cannot set the speed at the ideal value or (b) to increase task periods and reduce energy consumption while preventing overloads.*

the task set becomes $U(S_2) = 0.75$, leaving 25 percent of the processor unused.

In these situations, the system can invoke elastic scheduling to exploit the unused processor capacity and run the application tasks with higher rates, thereby improving system performance. Figure 6a illustrates the schedule that results from applying this approach.

Alternatively, the system can use elastic scheduling to reduce energy consumption by allowing the processor to run at a speed lower than ideal. 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 Figure 6b shows. Note that, without applying the elastic method, speed $S_3$ would cause a permanent overload that would degrade system performance uncontrollably.

Mutually exclusive resources and nonpreemptive code can generate scheduling anomalies in a processor with dynamic voltage scaling, causing tasks to increase their response times when the processor runs at higher speeds. Even worse, decreasing the speed can cause a permanent overload that degrades system performance in an uncontrolled fashion.

Such problems can be efficiently handled through a set of kernel mechanisms, including cyclic asynchronous buffers and elastic scheduling, that let system designers scale the performance of real-time applications as a function of processor speed. As successfully done in the SHaRK kernel, both CABs and elastic scheduling can be easily implemented on top of any real-time operating system, as a middleware layer, and they should be included in current standards to develop embedded systems with real-time and energy requirements.

## References

1. F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy," *Proc. 36th Ann. Symp. Foundations of Computer Science*, IEEE CS Press, 1995, pp. 374-382.
2. I. Hong et al., "Power Optimization of Variable Voltage Core-Based Systems," *Proc. 35th Ann. Conf. Design Automation*, ACM Press, 1998, pp. 176-181.
3. H. Aydin et al., "Power-Aware Scheduling for Periodic Real-Time Tasks," *IEEE Trans. Computers*, vol. 53, no. 5, 2004, pp. 584-600.
4. R. Melhem et al., "Power Management Points in Power-Aware Real-Time Systems," chapt. 7, *Power-Aware Computing*, R. Graybill and R. Melhem, eds., Plenum/Kluwer, 2002, pp. 157-152.
5. P. Pillai and K.G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," *Proc. 18th ACM Symp. Operating Systems Principles*, ACM Press, 2001, pp. 89-102.
6. F. Zhang and S.T. Chanson, "Processor Voltage Scheduling for Real-Time Tasks with Non-Preemptible Sections," *Proc. 23rd IEEE Real-Time Systems Symp.*, IEEE CS Press, 2002, pp. 235-245.
7. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, 1973, pp. 40-61.
8. G.C. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems*, vol. 29, no. 1, 2005, pp. 5-26.
9. G.C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed., Springer, 2005.
10. P. Gai et al., "A New Kernel Approach for Modular Real-Time Systems Development," *Proc. 13th IEEE Euromicro Conf. Real-Time Systems*, IEEE CS Press, 2001, pp. 199-206.
11. G.C. Buttazzo et al., "Elastic Scheduling for Flexible Workload Management," *IEEE Trans. Computers*, vol. 51, no. 3, 2002, pp. 289-302.

***Giorgio Buttazzo** is a professor of computer engineering at the Scuola Superiore Sant'Anna, Pisa, Italy. His research interests include real-time operating systems, dynamic scheduling algorithms, multimedia systems, and neural networks. He received a PhD in computer engineering from the Scuola Superiore Sant'Anna. Buttazzo is a senior member of the IEEE Computer Society. Contact him at giorgio.buttazzo@sssup.it.*