Journal of Systems Architecture 55 (2009) 90–101

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Power saving and fault-tolerance in real-time critical embedded systems

Rodrigo M. Santos [a,*], Jorge Santos [b], Javier D. Orozco [a]

[a] Dep. de Ing. Electrica y Computadoras Universidad Nacional del Sur/CONICET Av. Alem 1253, 8000 Bahia Blanca, Argentina
[b] Dep. de Ing. Electrica y Computadoras Universidad Nacional del Sur Av. Alem 1253, 8000 Bahia Blanca, Argentina

ARTICLE INFO

Article history:
Received 23 February 2006
Received in revised form 8 July 2008
Accepted 30 September 2008
Available online 8 October 2008

Keywords:
Real-time
Energy-aware
Fault-tolerance
Embedded systems

ABSTRACT

In this paper, a method with the double purpose of reducing the consumption of energy and giving a deterministic guarantee on the fault tolerance of real-time embedded systems operating under the Rate Monotonic discipline is presented. A lower bound exists on the slack left free by tasks being executed at their worst-case execution time. This deterministic slack can be redistributed and used for any of the two purposes. The designer can set the trade-off point between them. In addition, more slack can be reclaimed when tasks are executed in less than their worst-case time. Fault-tolerance is achieved by using the slack to recompute the faulty task. Energy consumption is reduced by lowering the operating frequency of the processor as much as possible while meeting all time-constraints. This leads to a multifrequency method; simulations are carried out to test it versus two single frequency methods (nominal and reduced frequencies). This is done under different trade-off points and rates of faults' occurrence. The existence of an upper bound on the overhead caused by the transition time between frequencies in Rate Monotonic scheduled real-time systems is formally proved. The method can also be applied to multicore or multiprocessor systems.

1. Introduction

Embedded computing is an established engineering discipline with principles and knowledge at its core. An embedded system is any computer that is a component in a larger system and relies on its own microprocessor [1]. Embedded systems represent the major market for microprocessors: in 2002 about 10^9 microprocessors (approximately half of the world production) found their way into embedded systems.

Although embedded systems have been in use for decades, in the last few years there has been a definite tendency to utilise them in more real-time complex applications in which more total computing power, fault-tolerance, reduced energy consumption and precisely timed behaviors are desirable. According to Lee [2], the time is ripe to build a 21st century embedded computer science.

Saving energy is particularly important in embedded mobile systems because, given the battery technology and the workload of a processor, a way to obtain larger intervals between successive battery's charges is through lower energy consumption. In CMOS technology, extensively used in today's processors, the power undergoes an approximate cubic reduction with the supply voltage. Because frequency and voltage are in an approximate linear relationship and the time necessary to execute a given task increases linearly with diminishing frequencies, the energy necessary to execute a given task diminishes quadratically with voltage [3,4].

In the embedded systems considered in this paper, the set of real-time tasks is close to the classic Liu and Layland model [5]: a set of independent, preemptible, periodic real-time tasks, each one characterized by an execution time, a period and a deadline. However, bearing real-world applications in mind, tasks have a worst case execution time and a minimal interarrival time instead of a fixed execution time and a period. Since there is a deterministic worst case, it is possible to calculate the time necessary for the processing of all tasks in each hyperperiod, defined as the Least Common Multiple, LCM, of all the periods. The remaining time, called slack, can be advanced by a proper redistribution and used, whatever the application, when deemed convenient. On top of that, more slack, often called gain-time, can be reclaimed when tasks are executed in less than their worst-case time or increase their interarrival time. Both slacks, redistributed and reclaimed, can be devoted to saving energy by executing tasks at lower speeds. The method, a collaborative relationship between the operating system and the application [6] falls within the workload-driven policies as defined in [7].

In [8], the notions of *k*-schedulability and singularity are amply discussed in the context of using the slack to schedule mixed and reward-based systems under the Rate Monotonic discipline. Later, the slack was used to provide deterministic guarantees on

* Corresponding author. Tel.: +54 291 4595181; fax: +54 291 4595154.
E-mail addresses: ierms@criba.edu.ar (R.M. Santos), iesantos@criba.edu.ar (J. Santos), jorozco@uns.edu.ar (J.D. Orozco).

1383-7621/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2008.09.001

fault-tolerance without missing hard deadlines [9]. In this paper, a method based on those notions is used to reduce energy consumption by lowering the speed at which real-time tasks are executed. However, as pointed out in [10], power aware embedded systems working at reduced voltages are prone to soft-errors (single-event upsets) which diminish their reliability. The cause for that is that lowering the voltage reduces the amount of the external perturbation required to upset the value stored at a circuit node. For example, in an accelerated soft-error testing experiment on a 4-Mbit SDRAM memory, the authors found that the number of errors increased from 57 to 658 (a 1154% increment), when the operational voltage was reduced from 5 V to 4 V (a 20% reduction). Therefore, a method that combines power saving and fault-tolerance appears to be particularly useful in critical embedded systems, especially bearing in mind that the designer can choose the trade-off point between both applications.

The rest of this paper is organised as follows: in Section 2, previous related work is reviewed. In Section 3, a short presentation of the theoretical foundation of *k*-schedulability and its application to the design of fault-tolerant systems is made; the contents of this section are not new but are essential to the understanding of the original contributions which follow. In Section 4, the method is applied to the scheduling of real-time systems executing at reduced speeds to save energy; this may be combined with fault-tolerance at different trade-off points; how to take into account the switching time between frequencies in real world processors is also analysed. In Section 5, the proposed method is evaluated by simulations with randomly generated sets of real-time tasks; relative savings vs utilisation factors for different trade-off points and rates of faults' occurrence are determined. In Section 6, the method is extended to multiprocessors and multicore systems and, finally, in Section 7, conclusions are drawn.

2. Related work

The list of discussed papers is representative of the state of the art. The problem of simultaneously saving energy and tolerating faults has been treated in [11–13]. Unsai et al. [11] use a task set model consisting of primary and secondary tasks executing in a set of loosely coupled processors under the Application-Level Fault-Tolerance scheme. Secondary tasks may be identical to the primary tasks or they may be a scaled down version of them. They introduce heuristics that attempt to finish executing primaries as early as possible while delaying the execution of secondaries as much as possible. This results in fewer secondaries having to run, with a corresponding saving of energy. The method is not applicable to the task model used in this paper, which is closer to the classic model presented in [5] and to more common real-world applications.

In [13,12], fault-tolerance and power saving are achieved via checkpointing and dynamic voltage scaling, respectively. In the checkpointing method, the system saves its state in a secure device at definite instants and when a fault is detected the system rolls back to the most recent checkpoint and resumes normal execution. In [13], a fixed priority discipline and equidistant checkpoints are used, whereas in [12], a non-uniform spacing of checkpoints (measured in CPU cycles) is introduced. In both cases, prime factors affecting the efficiency of the method are the overhead incurred in each checkpoint and the number of allowed checkpoints. An example of a single task system with different utilisation factors and different overheads, admitting one failure per task execution, is given in [12]. For a low utilisation factor and a very low overhead, 0.3% and 0.5%, respectively, the energy saving is 64%; it diminishes to 0% for values of 0.6 and 5%, respectively. Also, the placement of checkpoints measured in CPU cycles may be not feasible from the point of view of the software execution.

In [4], four energy saving methods are presented. The first one, called Sys-Clock, is a single frequency one. First, the authors prove that a processor will consume minimum energy if it runs at a constant frequency and completes the execution of the workload exactly at its deadline. As explained later in the present paper, the method can be combined with fault-tolerance if, when calculating the single frequency, the time necessary to recompute faulty tasks is taken into account. The algorithm to calculate the frequency, which is optimal in the sense that it minimizes energy consumption, is described. The optimality, however, is based on the availability of a continuous spectrum of frequencies, something that does not happen in real processors, in which only a reduced set of operating frequencies is available. The processor must then operate at the available frequency equal to or bigger than the frequency obtained by the calculations. If no operating frequency is available between the calculated and the nominal one, the processor operates without any energy saving. Two other methods use multiple frequencies; the fourth one cannot be used in practical applications because of the complexity of the required scheduling calculations.

Regarding power saving alone, different Dynamic Voltage Scaling algorithms, DVS, have been proposed for single processor systems operating under the Earliest Deadline First, EDF, and the Rate Monotonic, RM, scheduling disciplines. Pillai et al. [3] redistribute slack by using a single frequency to execute all tasks. If gain-time appears, it is used to lower the operating frequency in the interval between its appearance and the next deadline. The process is repeated at each scheduling point and there is no deterministic redistributed slack to be used in recuperating faulty tasks.

Qadi et al. [14] present a dynamic voltage scaling algorithm supporting the canonical sporadic task model defined in [15]. The method, however, is designed only for the EDF priority discipline and it is not applicable to RM.

Scordino and Lipari [16] propose a power aware scheduling using resource reservation techniques. The method, based on an algorithm called GRUB [17], reclaims the time not used in previous reservations to slow down the processor. It uses the EDF scheduling policy.

Quan et al. [18] propose two static fixed priority algorithms for scheduling real-time systems on variable voltage processors. The first one finds the minimum constant speed that can be applied throughout the execution of the whole tasks' set, shutting down the processor when it is idle. The second one produces a variable–voltage schedule which always results in lower energy consumption than the first algorithm. They assume continuous voltage variations and do not take advantage of reclaimed slack.

In [19], the authors propose an optimization algorithm to find for each task a speed that minimizes the energy consumption of the system. They show that each task can execute at constant speed in all its instances within the LCM. The optimization problem is proved to be non-linear and the computational complexity of the method is $O(n2\log n)$. The scheduling policy used is EDF. In the paper, the authors assume that frequency and voltage vary continuously and no switching costs are calculated in terms of time or energy. No fault recovery mechanism is considered for the tasks in the system.

In [20], some dynamic scheduling techniques are proposed for power aware real-time systems. The method includes three parts: (a) an off-line static calculation to compute the optimal speed assuming worst case execution times; (b) an on-line speed reduction, possible thanks to the reduction in actual execution times; (c) an on-line, adaptive and speculative speed adjustment, to anticipate early completions. The third speed reduction is not determin-

istic and therefore it cannot be used if deterministic guarantees on fault-tolerance must be given.

The problem of power saving in multiprocessor systems has been addressed by Zhu et al. [21]. They propose a technique, called Global Shared Slack Reclamation, in which the processors may share the reclaimed slack arising from a shorter execution time in one of them. The assignment of the set of tasks into the set of processors is made dynamically, subject to stochastic reductions in the execution times. Therefore, no deterministic guarantees can be given for fault-tolerance.

In this paper, a method based on the notion of k-schedulability is used to reduce energy consumption by lowering the speed at which real-time tasks are executed while providing deterministic guarantees on their fault-tolerance without missing hard deadlines. Some distinctive features of the method are that the designer can choose the deterministic guarantees of fault-tolerance to be given as well as fix the trade-off point between fault-tolerance and energy saving. The method is also evaluated and compared with other methods.

3. Rate monotonic schedulability

The RM discipline, a *de facto* standard at least in the USA [22], is used to define tasks' priorities. Several methods have been proposed for testing the RM-schedulability of real-time systems [5,23–25].

3.1. k-Schedulability

In [25], the Empty Slots Method is presented. Time is assumed to be slotted and the duration of one slot is taken as the unit of time. Slots are denoted t and numbered 1,2,... The expressions at the beginning of slot t and instant t mean the same.

The model, close to the classic Liu and Layland [5], is usual in real world applications. In it, real-time systems, $\mathbf{S}(n)$, of n independent preemptible periodic or pseudo-periodic tasks, must be scheduled to meet all time constraints. A task, τ_i, is said to be released (or instantiated) when it is ready to be executed. $\mathbf{S}(n)$ is completely specified as $\mathbf{S}(n) = \{(C_1, T_1, D_1), (C_2, T_2, D_2), \ldots (C_n, T_n, D_n)\}$, where C_i, T_i, and D_i, denote the worst case execution time, the period or the minimum interarrival time, and the deadline of τ_i relative to its release, respectively. In common applications, T_i is longer than or equal to D_i. If $T_i \leqslant T_{i+1}$, the system is RM ordered. In [5], it is formally proved that the worst case of load takes place when all the tasks request execution simultaneously.

Assuming that time is slotted and that the slot is the unit of time simplifies the theoretical approach and the schedulability calculations. To this effect, how long is a slot measured in s, the MKS unit of time, can be chosen by the designer and depends on the application. In [25], it is formally proved that $\mathbf{S}(n)$ is RM-schedulable iff

$$\forall i \in (1, \ldots n)$$
$$T_i \geqslant D_i \geqslant \text{least } t | t = C_i + \sum_{h=1}^{i-1} C_h \left\lceil \frac{t}{T_h} \right\rceil \tag{1}$$

The intuitive meaning of expression (1) is clear: τ_i will meet its time-constraint if its deadline is equal to or bigger than the time necessary to execute itself and all the tasks of higher priority, assuming that each task is executed as soon as possible. The last term on the right hand member of the expression above is called the work-function, denoted $W_{i-1}(t)$. It gives the number of slots necessary to execute all the tasks instantiated up to slot t. If M denotes the number of slots contained in a LCM interval, the expression $M - W_n(M)$ gives the number of empty slots (slack) in M. Obviously, once the set of real-time tasks is specified, the amount of slack in the hyperperiod cannot be increased. The case of deadlines longer than periods is simply handled by transposing their symbols ($D_i \geqslant T_i \geqslant \cdots$).

Note that when RM is used, it is not necessary to determine at the beginning of every slot which task will be executed in it. The assignment of tasks into the processor changes only when the execution of a task is finished or when a new task of higher priority than the one being executed, is released. What is more, if a task does not fully use its last slot, the beginning of the execution of the next task (if already released) can be advanced. A task τ_i belonging to a system $\mathbf{S}(n)$ is said to be k_i-schedulable if k_i is the largest integer satisfying

$$\forall i \in (1, \ldots, n)$$
$$T_i \geqslant D_i \geqslant \text{least } t | t = C_i + k_i + \sum_{h=1}^{i-1} C_h \left\lceil \frac{t}{T_h} \right\rceil \tag{2}$$

The intuitive meaning of expression (2) is clear: in the interval between its instantiation and its deadline, τ_i tolerates that k_i slots be used for purposes other than executing itself and giving way to higher priority tasks. In the context of this paper, those purposes are to use more time to execute the task, lowering the frequency and saving energy, or to recalculate faulty tasks. Those k_i slots are, essentially, redistributed slack. If $k = \min\{k_i\}$, the system is said to be k-schedulable.

A singularity, s, is a slot in which all the tasks released in $1, (s-1)$ have been executed. Note that $t = s - 1$ can be either an empty slot or a slot in which a last pending task completes its execution. s is a singularity even if at $t = s$ tasks are released.

In [8], it is formally proved that k slots can be given any anticipated use in the interval between each singularity and the next one without jeopardizing the real-time system. Note that since in the previous slot no task is pending execution, $t = 1$, the start of the system, is a singularity and k slots are already available for any desired use. The implementation to keep tab of the available slack is rather simple: at each singularity the operating system, OS, loads a counter with the value of k (redistributed slack) and decrements its content by 1 each time that a slot is used for purposes other than the execution of the original real-time set. Slack is available while the content is not zero. After reaching zero, the counter must wait until the next singularity to be reloaded to the value k.

The Empty Slots Method then provides the necessary and sufficient condition for the system to be schedulable, that is to meet all its time-constraints, assuming that each task is executed as soon as possible. It also gives the amount of empty slots, or slack, in the hyperperiod. The k-scheduling method goes further: it calculates the amount of slack that can be early used for purposes other than executing the original set of tasks at nominal frequency, while keeping the system schedulable.

3.2. Fault-tolerance

The application of k-schedulability to the design of fault-tolerant systems has been presented in [9]. An updated summary follows.

In this subsection, it is assumed that the k slots available after each singularity are entirely devoted to recomputing faulted tasks. The method falls within the class based on temporal redundancy and, therefore, the amount of possible recomputing is limited by the amount of redundant time available to do it. However, as will be shown later, the designer can choose how to use it, for instance providing deterministic guarantees of single or multiple recoveries to the more critical tasks. The fault-detection method, beyond the scope of this paper, is either explicit (through a pattern recognition

in which a signature is associated to a particular fault) or implicit (by some indicator caused by the fault) [26].

Assuming the worst case, only one singularity takes place in the interval $[1, T_n]$, where T_n denotes the maximum period in the tasks' set. $R_i = \lfloor k/\lceil T_n/T_i \rceil \rfloor$ represents the number of slots available to recover τ_i in each of its instantiations in the interval $[1, T_n]$. If $R_i \geqslant C_i$, the task may be recovered at least once in each instantiation. If $R_i < C_i$, the recovery can be made in only

$$p_i = \lfloor \lceil T_n/T_i \rceil / \lceil C_i/R_i \rceil \rfloor$$

instantiations out of the $\lceil T_n/T_i \rceil$ instantiations of the period, subject to the condition $R_i \lceil T_n/T_i \rceil \geqslant C_i$. The condition ensures that at least one recovery can be made.

If q_i denotes the number of instantiations in $[1, T_n]$ in which τ_i may fail once and be recomputed, the diophantic expression

$$\sum_{1}^{n} C_i \cdot q_i \leqslant k \tag{3}$$

subject to the condition $0 \leqslant q_i \leqslant \min[p_i, \lceil T_n/T_i \rceil]$, indicates a least upper bound on the combination of faults that the system can tolerate. Since $[1, T_n]$ is the most congested interval in the hyperperiod, when $T_n \neq M$, the method guarantees at least the same fault-tolerance in the intervals $[(jT_n + 1), (j+1)T_n]$, $j = 1, 2, \ldots,$ up to $(j+1)T_n = M$.

Example. Let $S(3)$ be the system specified in the first three columns of Table 1. The last four columns contain the calculated values of k_i, R_i, p_i and $\lceil T_n/T_i \rceil$, respectively.

Since $k = 5$, the combination of faults that the system can tolerate is then given by $q_1 + 2q_2 + 3q_3 \leqslant 5$. The relevant combinations of the number of instantiations in which the different tasks can fail once and be recomputed in [1,15] is given in Table 2.

Therefore, in the interval [1,15], τ_1 can be recuperated in its three instantiations and τ_2 in one; τ_1 in two instantiations and τ_3 in one; and, finally, τ_2 and τ_3 in one instantiation each. The same goes for the interval [16,30]. It must be noted that any of the above alternatives constitutes a deterministic guarantee on the fault-tolerance of the system. As explained before, the implementation requires only one counter in which the value k is loaded at each singularity and is decremented by 1 each time a slack slot is used.

As described, the method has only one checkpoint, at the end of the execution of each task. It can be easily converted to multiple checkpoints simply by dividing a task into convenient subtasks, each one tested at the end of its execution. In expression (3) partial, instead of total, execution times should be used.

4. Energy saving

Contrary to what happens in other applications (e.g. laptop computers), the disk and the screen are not normally used in embedded systems, specially in mobile ones. However, besides its cache, the microprocessor must access its DRAM, which operates always at its nominal voltage but with timing waveforms driven by the system clock [27]. The total (processor plus DRAM) percent saving is therefore application-dependent: the relative importance of savings in the processor decreases as the application requires more accesses to DRAM. In what follows, only the savings in the microprocessor are considered. This also excludes possible savings in other consumptions, e.g. WLAN devices [28]. The method, however, can be incorporated into multiple resource managers cooperative schemes such as that described in [29].

When the nominal voltage of the processor, V_n, is reduced to a certain operating voltage, V_o, the energy consumed by a CMOS processor is reduced approximately $(V_o/V_n)^3$ times; almost all energy saving methods are based, therefore, on reducing the voltage at which the system operates.

However, diminishing supply voltage results in increased circuit delay; therefore, the nominal frequency, f_n, must be reduced to an operating frequency f_o, and, consequently, the execution time is increased (f_n/f_o) times. Usually, the values of voltage and frequency are almost linearly related. In the case of the Transmeta Crusoe processor (Transmeta Tm 5400 specifications, cited in [27]) the curve voltage vs frequency is slightly convex and can be approximated by a straight line through the end points. The approximation is pessimistic in the sense that the operating frequency obtained for a given voltage in the approximation is actually higher than that obtained from the real curve; consequently, it will produce less saving. It must be borne in mind that although it is possible to reduce frequency without reducing voltage, the converse is not true. Assuming, then, that voltage and frequency are linearly related, energy consumption per executed task, reduced $(V_o/V_n)^2$ times, is also reduced $(f_o/f_n)^2$ times.

It helps to simplify calculations if the power of the processor at nominal frequency is taken as the unit of power. Since the unit of time is the slot, the unit of energy turns out to be the energy consumed by the processor working at full power during one slot. It is assumed that when there are empty slots (no tasks to be processed) the processor is idle and works at 15% of the power consumed at the lowest available frequency, a figure based on measurements presented in [27]. Knowing the power in W and the slot-time in s, it is a trivial matter to obtain the energy consumed in MKS units (Ws).

Energy saving methods can be broadly classified in two classes: single frequency and multiple frequencies methods. Although the saving is dependent on the type of load to be processed and the processor in which it is processed, some general characteristics of both classes can be given. In the single frequency class, one frequency, deemed to be optimal or suboptimal (in the sense of consuming less energy), is calculated for the system clock, and the workload is always processed at that same frequency. In the multiple frequencies class, the system clock switches between frequencies according to certain criteria (e.g., always processing each task at the same reduced frequency or reducing frequency when slack is available); this class pays a penalty for the overhead introduced when switching frequencies.

In [4], three practical energy saving methods are proposed. The first one, Sys-Clock, is a single frequency method optimizing energy consumption; it is the best among all single frequency methods. The authors present comparative evaluations of Sys-Clock against four multifrequency methods, two by Pillai and Shin (Static Voltage Scheduling and Cycle Conserving), and two by Saewong and Rajkumar (Priority Monotonic and Dynamic Priority Monotonic). Comparisons are made by determining, in all cases, energy consumption relative to consumption at nominal frequency (with no saving at all). Since evaluations in this paper are performed

Table 1
System's specification

i	C_i	$T_i = D_i$	k_i	R_i	p_i	$\lceil T_n/T_i \rceil$
1	1	6	5	1	3	3
2	2	10	6	2	2	2
3	3	15	5	5	1	1

Table 2
Diophantic expression coefficients

q_1	q_2	q_3
3	1	0
2	0	1
0	1	1

against Sys-Clock, the results can be transitively compared to those obtained in [4].

In what follows, Sys-Clock is described in the context of an available continuous spectrum of frequencies; this means that the processor can operate at the calculated frequency, whichever it is. Then, a multiple frequencies method, based on k-schedulability and denoted kFE, is introduced, also in the context of a continuous spectrum of frequencies. The discussion focuses on the limitation imposed on both methods by the fact that in real world processors, only a small number of operating frequencies are available. The cost of the frequency switching overhead is formally proved and finally, theoretical findings are summarized and the kFE-scheduling algorithm is explained, step by step. Throughout the section, examples are given, of course not as proofs but to illustrate the theoretical conclusions.

4.1. Single frequency from a continuous spectrum

In order to process a given workload, a minimum constant operating frequency for the whole system is calculated. In [4], it is formally proved that the frequency calculated with the proposed Sys-Clock algorithm is optimal among single frequency methods in the sense that there is no other single frequency scheme that consumes less energy and guarantees all the timing constraints.

In the algorithm, given a set of n tasks, n subsets can be formed; each subset consists of a task and all the tasks that, because of having higher priority, may preempt it. Essentially, the algorithm determines first the n constant clock frequencies that minimize consumption to execute each subset. Each frequency is determined by slowing down the execution of the subset as much as possible without missing any deadline. The maximum frequency of this set of task-oriented minimum frequencies is then adopted as single frequency for the whole system. It must be borne in mind that the exact implementation of the algorithm will require that the calculated frequency is available at the processor; in other words, a continuous spectrum of frequencies should be available.

Although it is designed only to save energy, the method can be combined with fault-tolerance by incrementing the workload by the time necessary to recompute the faulty tasks guaranteed to be recovered.

It may happen that $T_n < M$, that is, the maximum period is smaller than the hyperperiod. In that case, the intervals of length T_n following the first one will be less congested and idle times will appear. The consumption of the processor in the idle state is assumed to be 15% of the non-idle consumption at that frequency.

Example. In $\mathbf{S}(3) = \{(1,6,6),(2,10,10),(3,15,15)\}$, the system of the example in Section 3.2, τ_1 is assumed to be a very critical task and must be recuperated even if it fails in all its instantiations. Because of this, the system can be seen as composed of four tasks, the original three plus the recomputations of τ_1.

The energy consumption in the hyperperiod, with the processor operating at nominal frequency, can be calculated as the sum of the workload plus 0.15 times the number of empty slots:

$$W_4(30) + 0.15 \cdot (30 - W_4(30)) = 23.20$$

The minimum single frequency, calculated with the Sys-Clock static algorithm, is $0.8\hat{6}$. Bearing in mind that $(V_o/V_n)^2 = (f_o/f_n)^2$, the energy consumption is

$$(f_o/f_n)^3[W_4(30)/(f_o/f_n) + 0.15(30 - W_4(M)/(f_o/f_n))] = 16.97$$

73.14% of that at nominal frequency with a relative saving of 26.86%. The processor is saturated in [1,15] but because the interval [16,30] is less congested, 4.61 empty slots (idle processor time) appear at the end of the interval (Fig. 1a).

4.2. Multiple frequencies from a continuous spectrum: redistributed and reclaimed slack

The use of several frequencies to save energy is always conditioned to the fact that the voltage scaling overhead at every frequency-switch is acceptable. For the sake of clarity and since the examples of this and the next subsection are slated only to illustrate the different methods, switching-times between frequencies will be neglected. How they influence real world applications is analysed in Section 4.4 and taken into account when performing comparative evaluations in Section 5.

In [4], the Priority-Monotonic Frequency Assignment (PM-clock) algorithm is presented; it uses the Sys-Clock algorithm but it assigns one frequency for each task in such a way that a higher priority task will always be executed at a frequency higher than or equal to that of a lower priority task. It can be improved by a dynamic version (DPM-clock) that detects early completion times and makes use of the additional slack to reduce frequencies. In [3], the authors propose a single frequency method, Static Voltage Scaling, and a multiple frequencies one, Cycle-Conserving Real-Time Dynamic Voltage Scaling. The last one takes advantage of slack appearing when tasks execute in less than their worst-case execution time. In [4], these four multiple frequencies methods are comparatively evaluated against Sys-Clock; the metric used in the evaluation is the energy saved relative to nominal consumption.

In what follows, the multiple frequencies kFE-scheduling method (for k Fault-tolerant Energy saving), based on k-schedulability, will be presented. The value of k obtained statically for each processor is a deterministic lower bound on the amount of slack to be redistributed. It is a lower bound because there may be empty slots not captured by the method, leading to a value of k smaller than the total number of empty slots in the hyperperiod, $(M - W_n(M))$. If both applications, energy saving and fault tolerance, are sought, the slack to be used in providing a deterministic guarantee on the fault-tolerance of the system, denoted k_f, must therefore be set using only part of k. After assigning k_f to fault recovery, $k_e = k - k_f$ can be devoted to extending the execution time of other tasks. The operating frequency can therefore be lowered and voltage diminished accordingly; energy saving is then obtained. The designer is free to choose the trade-off point between both uses.

Since $t = 1$ is a singularity, the execution of the first task can immediately be extended by k_e slots without compromising the timing requirements of the real-time system. In that case, the operative frequency would be

$$f_o = (C_1/(C_1 + k_e)) \cdot f_n \tag{4}$$

In general, if task τ_i is executed after a singularity, it can be processed at a frequency

$$f_o = (C_i/(C_i + k_e)) \cdot f_n \tag{5}$$

The only implementation's requirement is that at each singularity the OS loads a counter with the value of k_e and decrements its content by 1 each time that a redistributed slack slot is used to diminish the frequency.

Example. Suppose that $k_e = 2$ and τ_i, instantiated at a singularity, has a $C_i = 2$. Therefore τ_i may execute at $f_o = 0.5f_n$ in four slots. The first two slots (the singularity and the next) correspond to the task's own slots (assigned to it when scheduling the system) and the counter is not decremented. The third and fourth slots are redistributed slack; in the third, the content is decremented to 1 and in the fourth to 0. The counter will reload a value of 2 at the next singularity.

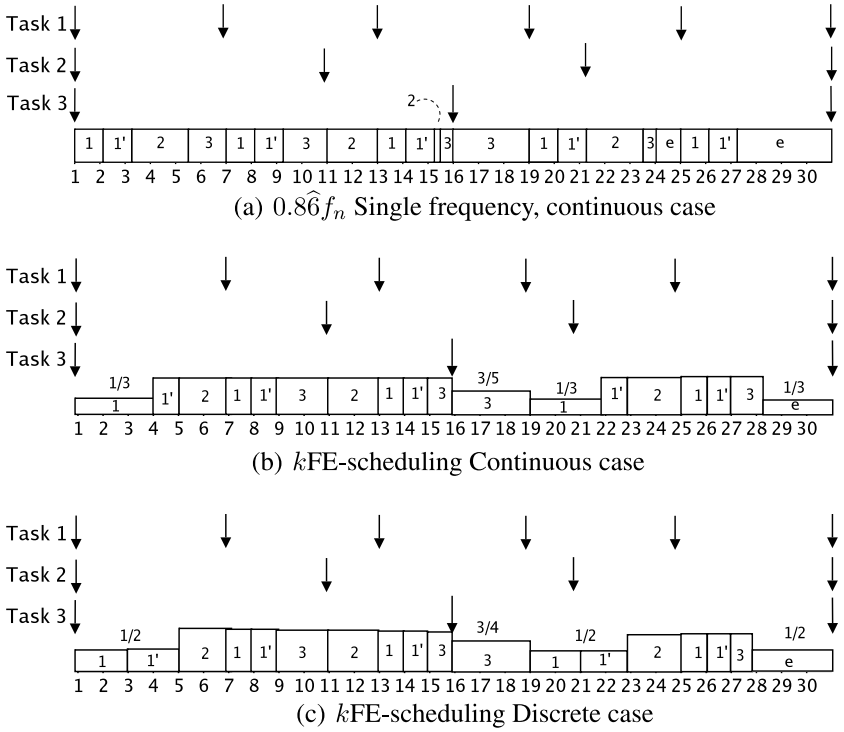


Fig. 1. Example's temporal evolutions.

Example. In $\mathbf{S}(3) = \{(1,6,6),(2,10,10),(3,15,15)\}$, the system of the previous example, the recuperation of τ_1, even if it fails in all its instantiations, is guaranteed by taking $k_f = 3$ However, since $k_e = k - k_f = 2$, the system cannot only recuperate tasks but also save energy as shown in Fig. 1b; the energy consumption in this case is 19.08, 82.24% of the consumption at full speed; the percent saving relative to nominal frequency is $(1 - (19.08/23.20))$ 100 = 17.76%. Note that the system takes advantage of singularities occurring at $t = 1$ and 15. At $t = 16$, τ_3 starts executing at $f_o = (3/5)f_n$, but before using the redistributed slack it is preempted by τ_1 that executes at $f_o = (1/3)f_n$. In order to complete its execution, τ_3 still needs 1.2 slots at nominal frequency; it uses them starting at $t = 27$. When idle, the processor switches to $f_o = (1/3)f_n$ and its consumption is 15% of the non-idle state at that frequency. The saving in this case (17.76%) is smaller than the saving obtained using the continuous spectrum single frequency method (26.86%).

Moreover, if some task takes less than its worst-case execution time, *WET*, the difference between it and the actual one, *AET* can be used to reduce the frequency in the processing of the following tasks. The method consists, then, in a reclamation of the slack generated by the reduction in execution time, $k_r = WET - AET$. The value of k_r is stochastic and depends on the probability that, for each task, $AET < WET$, holds.

The method is greedy in the sense that the available slack is devoted entirely to reduce the execution frequency of the task that follows immediately. The implementation is similar to the previous ones: a third counter is loaded with k_r, the value of the reclaimed slack or gain time, when the donating task finishes its execution; it is decremented by 1 with each reclaimed slack slot used to save energy. As in the case of slack redistribution, the nominal frequency can be reduced to

$$f_o = (C_i/(C_i + k_r)) \cdot f_n \tag{6}$$

It must be noted that singularities, when occurring, are easily detected. Since k_e is determined once for all, the calculation of f_o after each singularity imposes a light overhead. It could be argued that better results would be obtained if the available slack were spread throughout the tasks going to be executed up to the next singularity. This requires, however, a dynamic assignment which would have two shortcomings:

(a) The need to determine when the next singularity would take place, as opposed to detecting it when occurring, would impose a rather heavy overhead on the OS. On top of that, since in the model used in this paper interarrival and execution times may vary, the calculation may be pessimistic.
(b) Spreading the slack too much may lead, in the case of discrete frequencies, to waste it because the system may end operating at f_n. On the same line of reasoning, choosing a task that optimizes the use of the available slack between singularities would also impose a heavy overhead on the OS.

4.3. Single and multiple frequencies from a discrete spectrum

In the previous analysis, the fact that in real microprocessors the operating frequency cannot be varied continuously was not taken into account. In the case of the Intel PXA255, for example [30], the available operating frequencies are 400, 300 and 200 MHz (f_n, $0.75f_n$, $0.5f_n$, or simply 1, 0.75 and 0.50 if normalized to the nominal frequency). In both classes, single and multiple frequencies, this produces more energy consumption than in the theoretical continuous model because the real operating frequency must be adjusted to the available frequency, often higher than the one obtained by calculations. In [4], this energy loss is called energy quantization error.

Since frequencies will not vary continuously, the expression of the operating frequency to save energy in kFE-scheduling will then be properly expressed as

$$f_o = \lceil (C_i/(C_i + k_e))f_n \rceil \tag{7}$$

denoting the smaller frequency equal to or bigger than $(C_i/(C_i + k_e))f_n$ available in the processor. Associated to each frequency there will be an operating voltage from which the reduction in the energy consumption may be calculated.

Something similar happens in Sys-Clock, in which the operating frequency must be bigger than or equal to the optimal theoretical one obtained by calculations. The method will be called Discrete Sys-Clock, denoted DSC.

Example. The evolution of the system of the previous examples using *k*FE-scheduling to save energy, when only the frequencies 1, 0.75 and 0.50 are available, is shown in Fig. 1c). Total consumption is now 18.07, 77.89 % of that at nominal frequency with a saving of 22.11%. Note that at $t = 19$, τ_3, which is executing at $f_o = 0.75$, is preempted by τ_1. Since the three slots used up to then by τ_3 are its own, redistributed slack is still not used. It can be devoted to lowering the operational frequency of τ_1 and its recuperation after failing. Since no other frequency is available between $0.8\widehat{6}$ and 1, DSC cannot save energy because it must execute at nominal frequency.

The performance of *k*FE-scheduling relative to DSC can be even better. The reason behind this fact is that the single frequency is calculated for the worst scenario in which all possible failures guaranteed to be recuperated actually occur. However, the assumption is pessimistic and if the failures do not take place, the best that DSC can do in the slack appearing is to operate at 15% of the non-idle processor consumption. *k*FE-scheduling, instead, makes a better use of the slack by lowering the frequency at which some tasks are executed.

Example. Let us see what happens if, in the system of the previous example, τ_1 fails in only one of its instantiations

(a) Nominal frequency: If τ_1 fails in only one instantiation, the consumption at nominal frequency is 19.8, resulting from 18 slots fully used to execute the normal workload plus one recovery and 12 empty slots consuming 15% of the nominal energy.
(b) Single continuous frequency: Since τ_1 fails only once, the workload is executed in 20.77 slots. The rest, 9.23 slots, is idle and consumption is 15% of the non-idle processor consumption at $f_o = 0.8\widehat{6}$. Total consumption is 14.87, with a saving of 24.89%.
(c) Single discrete frequency: Since $0.8\widehat{6}$ is between 0.75 and 1, it must execute at nominal frequency and no saving is obtained.
(d) *k*FE-scheduling: The consumption of the system is 11, 72 with a saving of 40.79%.

As can be seen, *k*FE-scheduling outperforms not only discrete but also continuous single frequency.

4.4. Frequency switching overheads

Transit times between frequencies are far from being negligible and must be taken into account as an overhead when choosing single vs multiple frequencies. In certain processors, frequency switching may take a relatively long time: the Compaq iPAQ, for instance, needs 20 ms to synchronize SDRAM timing [4]; other processors take much less: the Intel PXA255, for instance, has a transit time of only 500 μs [30]; however, during that period, the CPU clock stops completely until the new frequency is stabilized. It must be noted that only frequencies synthesized from the main crystal (3.6864 MHz) are considered. The processor has a sleeping mode achieved by turning off the main crystal and turning on a 32.768 kHz crystal; this oscillator, however, can take up to 10 s to stabilize. Because it takes so long, the sleeping mode cannot be used in real-time applications and it will not be considered here.

Every time a frequency switching takes place, there is also a context switching, taking a certain time. This involves the time required to maintain the context of the jobs involved as well as the time spent by the scheduler to service the interruption that triggers the switching. The problem of context switching is analysed in ([31], Real-Time Systems, p. 165) in the frame of conventional single frequency executions. It is shown there that in order to take into account the context switching overhead in job-level fixed priorities assignments, it suffices to add twice the context switching time to the execution time of each task.

Although frequency switchings, as managed by the methods described in this paper, always imply context switchings, the converse is not true. Frequency switching overhead warrants then a separate analysis. How this overhead (considerably larger than context switching's) is introduced in the schedulability calculations of the system, is formally proved in what follows.

An RM scheduled real-time system operating with a restricted set of multiple frequencies is considered. Two things must be borne in mind: (a) because there are only a few available frequencies, a task may finish its execution without consuming all the available slack. This happens because the available frequency is higher than the theoretical one calculated from a continuous spectrum; a faster processing leads to an earlier finish, leaving some slack unused. (b) a task executing at a reduced frequency may be preempted by a higher priority task before consuming all the available slack; what remains can, in turn, be used by the preempting task.

Example. τ_i of $C_i = 2$ is instantiated at an instant with $k_e = 3$. It could execute at $f_o = (C_i/(C_i + k_e))f_n = 0.4f_n$ but that frequency is not available. Instead it executes at $f_o = 0.5f_n$, the smallest available frequency bigger than or equal to $0.4f_n$. In doing so, it consumes two slots of its own and two redistributed slack slots. Therefore when it finishes its execution there is still one slack slot ready to be used by another task (Fig. 2a)).

Example. If τ_i of the previous example is preempted after executing three slots, two out of the three available slack slots, can be used by the preempting task. It may happen, however, that three slots are not enough to lower the frequency of the preempting task. In that case, it will execute at nominal frequency and the slack will be passed on to the following task, be it τ_i or some higher priority task preempting it (Fig. 2b and c).

Lemma 1. *The maximum number of frequency changes that a system may suffer between singularities is twice the number of tasks' instantiations in the interval minus 1.*

Proof. Between singularities there will be a certain number of slack slots available to reduce frequency. The first task instantiated after the singularity may start execution at a frequency different from that operating at the moment of its instantiation, requiring one frequency switch. From then on, three alternatives may occur: (a) The task finishes its execution at a slot preceding a singularity and it is therefore followed by an empty slot or by the execution of a task instantiated at the singularity. (b) The task finishes its execution and it is followed by a pending task of lower priority. (c) The task is preempted by another task of higher priority. In the three cases a frequency switching may take place. The alternatives are repeated for the second, third, etc. tasks until the last one before the next singularity. In the worst case there will be one switch associated to the first task and two switches associated to each of the following tasks. The total number of switches will therefore be twice the number of instantiations minus 1. □

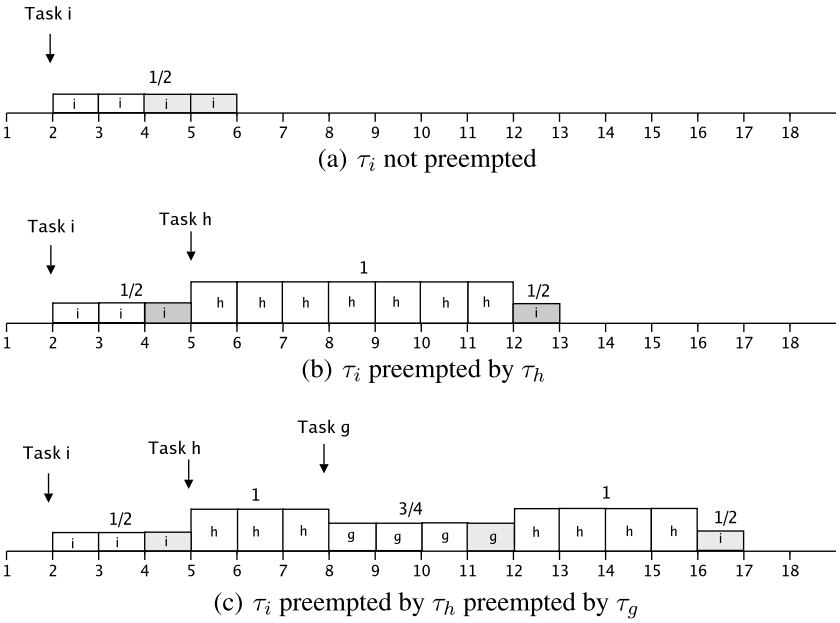


Fig. 2. Grey: slack slots. $C_i = 2$, $C_h = 7$, $C_g = 3$, $k_e = 3$. Priority order $\langle g,h,i\rangle$.

Theorem 1. *The upper bound on the overhead caused by the change in frequencies to be added to the execution time of each task, is twice the transition time.*

Proof. Immediate from the previous Lemma. □

It must be noticed that the bound is very pessimistic. It is based on the assumption that at every instantiation the task will switch frequencies. In general this will not be true. It would require not only that every task but the last one leave a residual redistributed slack for at least some of the following tasks to execute at a reduced frequency but also that the reduced frequency was different for successive tasks. In the example of Fig. 1c), for instance, there are six instantiations between the first two singularities (at $t = 1$ and $t = 16$) but only two frequency changes instead of 11. Between the second and the third singularities (at $t = 16$ and $t = 28$) there are four instantiations but only three frequency changes instead of seven.

On top of that, if in spite of incrementing the execution time to take care of frequency changes, the system is schedulable, a reclaimable slack will be produced when the double switching does not take place. As explained in Subsection 4.2, *k*FE-scheduling takes advantage of this slack to reduce the frequency of the next task to be executed.

4.5. The kFE-scheduling algorithm

The theoretical findings presented above about scheduling sets of real-time tasks in embedded processors in which fault-tolerance and energy saving are prime concerns, can be summarized as follows:

(a) Sys-Clock is optimal among all single frequency methods to save energy. Its use can be extended to tolerate faults. The minimum possible frequency is calculated and the system operates always at that frequency. Since it has been evaluated against other methods, it is a good transitive comparison base.
(b) The multifrequency *k*FE-scheduling method is based on the early capture of slack and the possibilities of using it to recompute faulty tasks or to save energy by working at a reduced operating frequency chosen from a set of available frequencies. *k* is a deterministic lower bound on the amount of available slack.
(c) If failures do not take place or tasks are executed in less than their worst case execution time, additional slack appears. The single frequency method will have a linear energy saving because the processor is idle and operates at a fraction of the nominal power. The *k*FE-method, instead, may have a quadratic saving by using the slack to operate at reduced frequencies. The overhead incurred when changing frequencies, however, must be taken into account.

The *k*FE-algorithm, step by step, is then

(1) In the RM-schedulability calculations, increment the execution time of each task by twice the transition time between frequencies.
(2) If the system is still schedulable, determine the system's value of *k*.
(3) According to the sought guarantees, determine the value of k_f and consequently of k_e, devoted to fault-tolerance and energy saving, respectively.
(4) Let the system operate at nominal frequency and reduce it by using the k_e slots available after each singularity or the k_r slots available after each execution in which $AET < WET$.

5. Comparative evaluations

The *k*FE-scheduling method here proposed to save energy while providing a deterministic guarantee on a given fault-tolerance combination was evaluated by means of simulations performed on random generated sets of tasks. Each set was processed under three methods: (a) nominal frequency leading to full consumption with no savings except those coming from an idle processor consuming 15% of the energy consumed when fully loaded. (b) Discrete Sys-Clock, DSC, operating at a single frequency from a discrete spectrum, bigger than or equal to the optimal theoretical one. (c) *k*FE-scheduling. The metric used to compare cases (b)

and (c) against (a) was, as in the previous examples, the percent of energy saving relative to consumption at nominal frequency, that is $(1 - E_o/E_n)100$, where E_o and E_n denote the energy consumed to process the same load at the operating and at the nominal frequency, respectively.

Simulations were also used in [4] to compare DSC to other methods, all multifrequency (Pillai and Shin's Static Voltage Scheduling and Cycle Conserving; Saewong and Rajkumar's PM-Clock and DPM-Clock); the results obtained here may be, therefore, transitively compared to those presented in that paper. Other methods are ruled out for different reasons, e.g., no combination with fault-tolerance is possible or no fixed priority schemes are used.

The touchstone of the results presented is the repeatability of the experiments leading to them. In what follows a detailed specification of how the random sets were generated and how the simulations were carried out is presented.

5.1. Setting the simulations

In order to perform the simulations, sets composed of 10 tasks were generated with uniformly distributed periods in the sample space $[100, 110, 120, \ldots, 1100]$ and utilisation factors in the range $[0.20, 0.21, \ldots, 0.89]$. For nine tasks, selected at random, execution times were assigned to produce a task utilisation factor randomly selected for each task in the interval $[0.06, 0.12]$ of the set's utilisation factor. The execution time of the tenth task was adjusted to produce the set's final desired value. For each value of the utilisation factor, an average of more than 600 sets were run. The saving was calculated as the average of the sets' savings.

Systems were run for about ten thousand slots; this means that the task with the longest period, T_n, is executed at least nine times, usually more. Given the number of tasks and the sample space of the periods, the LCM may be in the order of 10^9. However, since $[1, T_n]$ is the most congested interval, the method guarantees at least the same fault-tolerance in successive intervals of length T_n up to the end of the hyperperiod. Therefore, in order to obtain good comparative evaluations, ten thousand slots are a satisfactory interval, long enough to produce an ample variety of faults, redistribution of slack, etc. As a matter of fact, experiments carried out over the hyperperiod show no significant differences with those restricted to 10 000 slots.

As usual in reliability theory, faults were generated using an exponential distribution [32]. The underlying assumptions are that faults are independent and that they occur at a constant rate. Two rates were selected: a low one (one fault every 30 000 slots) and a high one (one fault every 130 slots). This is practically equivalent to having systems with no faults at all or with faults saturating the guaranteed recovery capacity of the system.

Savings were determined and plotted vs utilisation factors, UF, using both methods, *k*FE and DSC, and two parameters: (a) the two rates of occurrence of faults (nil and saturating). (b) three trade-off points corresponding to three values of k_f (0.25, 0.50 and 0.75 of k). Six subfigures are therefore generated.

Switching times in multifrequency methods are an overhead that must be taken into account to make a fair comparison against single frequency methods. For every frequency switching in *k*FE, a slot was added during which the processor did not make any computation but consumed energy at the higher of the two frequencies.

5.2. Results obtained

Results are presented in Fig. 3. In all cases, relative savings are plotted against the original utilisation factor of the system, prior to its actual increment derived from the re-execution of faulty tasks.

Subfigures in the left and right columns present results for the low and the high rates of faults' occurrence, respectively. Subfigures in the three rows present relative savings for the three values of k_f. Although savings in the case of *k*FE always diminish continuously, in (a) and (b) three plateaus can be seen for DSC. As the trade-off point is shifted towards more fault-recovery and less saving, only two plateaus appear in (c) and (d). Finally, in (e) and (f) only one plateau can be seen.

5.3. Results explained

The first thing to note is that for every possible combination of fault occurrence and trade-off points, relative savings diminish as utilisation factors increase. This could be expected: in the region of low utilisation factors there is abundant slack available to reduce frequencies and, consequently, relative savings are bigger. As the utilisation factor increases, less slack is available and relative energy savings diminish. As the utilisation factor tends to 1, slack tends to zero and both saving methods converge towards no saving at all.

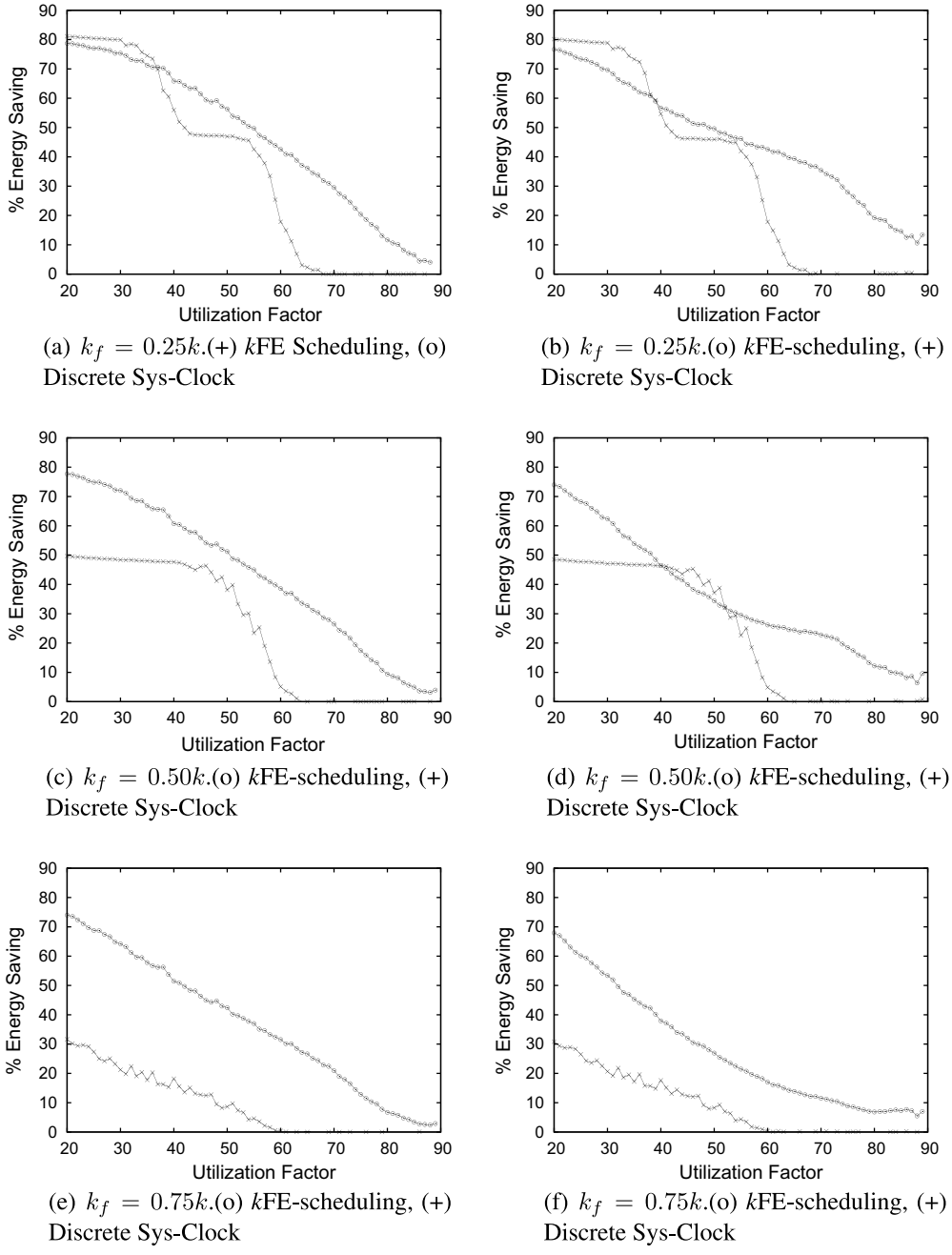
Because *k*FE can use any of the available frequencies when allowed to do so, it degrades gracefully following a continuous, almost linear, diminution. DSC, instead, presents plateaus, and the reason is that as the utilisation factor increases, less slack is available for both the reduced and the nominal single frequencies, and both savings are smaller. However, because the slack is reduced proportionately for both, the ratio E_o/E_n has very small variations and therefore the relative saving is approximately constant, generating a plateau. From a certain utilisation factor up, the system must jump to the next higher available frequency, diminishing the relative savings and generating a lower plateau.

Let us now see how the trade-off point influences savings. In order to make comparisons under the same rate of faults' occurrence, graphs to be considered are those belonging to the same column. In the left column (low rate), it can be seen that for a low k_f (subfigure a)) there is abundant slack and DSC can execute at $0.5f_n$ for low utilisation factors and at $0.75f_n$ for middle ones, respectively. From a certain utilisation factor (circa 0.65) up, the processor is constrained to operate at nominal frequency and the relative energy saving is nil.

As k_f increases, the trade-off point is shifted towards more fault recovery capacity and, consequently, there is less slack available to save energy. Because of that, DSC cannot operate at $0.5f_n$ and the first plateau disappears at (c). For high values of k_f, the plateau corresponding to $0.75f_n$ also disappears and only an approximately continuous variation takes place, as shown in (e).

The region of $k_f = 0.25$ and very low utilisation factors is the only one in which DSC shows a better performance than *k*FE. This is because DSC can operate continuously at $f_o = 0.5f_n$, whereas *k*FE can do it only during the k_e slots following a singularity. Except for that small region, *k*FE always outperforms DSC. The reason behind this, is that *k*FE profits from the fact that it can switch between the three available frequencies and, although reducing the operating frequency only for short intervals, it causes an overall reduction in energy consumption. The same conclusions are obtained by analysing the right column (subfigures (b), (d) and (f)), although in this case, high rate of faults' occurrence, DSC also produces better savings in a narrow region of mid utilisation factors for $k_f = 0.50k$.

Let us now examine how variations in the occurrence of faults influence savings. In order to make comparisons with the same trade-off points, graphs to be considered now are those belonging to the same row. Relative savings in the case of DSC are approximately equal for both rates of faults' occurrence. This is because the operating frequency must be calculated with allowances for recomputing faulty tasks. If faults take place, the corresponding slots are used for recomputing. If faults do not occur, those slots



(a) $k_f = 0.25k$.(+) kFE Scheduling, (o) Discrete Sys-Clock

(b) $k_f = 0.25k$.(o) kFE-scheduling, (+) Discrete Sys-Clock

(c) $k_f = 0.50k$.(o) kFE-scheduling, (+) Discrete Sys-Clock

(d) $k_f = 0.50k$.(o) kFE-scheduling, (+) Discrete Sys-Clock

(e) $k_f = 0.75k$.(o) kFE-scheduling, (+) Discrete Sys-Clock

(f) $k_f = 0.75k$.(o) kFE-scheduling, (+) Discrete Sys-Clock

Fig. 3. Relative savings vs utilisation factors. Results: (a), (c) and (e), low faults' rate. (b), (d) and (f) high faults' rate.

go empty and the consumption is 15% of that at the operating frequency. Exactly the same thing happens in the system operating at single nominal frequency; the consumption ratio and therefore relative savings remain approximately equal. In the kFE case, on the other hand, for a low rate of faults, empty slots (originally scheduled to recover faults that actually do not take place) and therefore singularities appear; the system switches to the lowest frequency and consumption is only 15% of consumption at that frequency. The consumption ratio is lower and the relative saving higher then in the high rate case. Larger k_f's accentuate this effect as can be seen when comparing kFE savings in (c) vs (d), and (e) vs (f).

In the kFE case, further reductions can be obtained if tasks execute in less than their worst-case time and generate a stochastic slack that can be reclaimed for that purpose. DSC takes advantage of this gain-time only by operating an idle processor at 15% of the power corresponding to the selected frequency. If the operating single frequency is not the minimum available, the multiple frequencies method, which, although only for short intervals, has the possibility of switching to the minimum one, will yield, in general, a better saving.

Comparative evaluations of DSC against four multiple frequencies methods (Pillai and Shin's Static Voltage Scheduling and Cycle Conserving; Saewong and Rajkumar's PM-Clock and DPM-Clock) were reported in [4]. Simulations with synthetic sets were performed for different UFs (0.2, 0.4, 0.6 and 0.7), different number of available frequencies (2 and 10) and two ratios of Best Case Execution Time/Worst Case Execution Time (0.5 and 1). In general, savings decrease as utilisation factors increase; savings are larger when more discrete frequencies are available. With two frequencies, DSC shows a plateau between UFs of 0.2 and 0.6, and then

an abrupt fall for UF = 0.7. All this is consistent with the results previously presented. Among the five methods tested, DSC and Cycle are the more efficient with very little differences between them. Since in the comparative evaluations presented above, *k*FE generally outperforms DSC, the result can be transitively extended to the other four methods.

6. Multicore processors and multiprocessor systems

The problem addressed in this section is how to apply the *k*FE-scheduling method to systems composed of several processors, be them on a chip (multicore) or on different chips (multiprocessor). This kind of systems are the answer to the limitation, imposed by current transistor technology, to the increase in computer power of single processors. An important byproduct is the fact that energy consumption increases by a multiplicative factor as the speed of a single processor is increased, but only by an additive factor if more processors are added to obtain the same computing power [33].

The first problem to solve when using multicore or multiprocessor systems is the assignment problem, defined as how to map a set of real-time tasks into a set of processors in such a way that no deadline is missed. Assignment methods can be broadly classified in two classes: global and partitioned. In the first class, e.g. [21], different instantiations of the same task may be executed on different processors; what is more, migrations of tasks only partially executed are permitted and, as a consequence, an instantiation preempted on a particular processor may resume execution on a different one [34]. Global methods are therefore essentially dynamic and stochastic, a decisive shortcoming from the point of view of fault-tolerance because no deterministic guarantees can be given.

In the partitioned class, tasks are assigned off-line, once for all, and all their instantiations are executed on the same processor. *k*FE-scheduling can then be applied to each processor, obtaining deterministic bounds on fault-tolerance and energy saving, the basic problem addressed in this paper.

When the system is power aware and/or fault tolerant, the assignment must tend to produce load-balanced processors, avoiding light loads in some of them and near saturation in others. This is because in near saturated processors little slack is available for redistribution and, consequently, the chances to save energy and/or to recover faulty tasks are diminished. With balanced loads, instead, the combined effect of fault-tolerance and energy saving is enhanced. Once tasks have been assigned, it will be assumed that the operating system may apply a dynamic power-coordination technology able to slow down the processors separately [35]. A set of n tasks mapped into a set of m processors generate m^n possible assignments. The assignment problem is NP-hard and in the past has been addressed by many methods belonging to the partitioned class, e.g. Heuristics [36–38], Simulated Annealing [39], Genetic Algorithms [40] and Fuzzy Algebras [41].

After obtaining a proper partition of tasks, the *k*FE-scheduling method can be applied to each of the processors, with the additional advantage that the trade-off point between power saving and fault tolerance is not necessarily the same for all processors. In the case of units engaged in the processing of more critical tasks, the use of the available slack may be shifted towards recalculating faulty tasks. If the tasks to be processed are not critical, the weight may be shifted towards energy saving.

Having all the above in mind, it can be said that the *k*FE-scheduling method can be applied to multicore or multiprocessor systems. In order to provide deterministic guarantees, a previous partition of the set of tasks, mapping them into the set of processors, must be carried out. The assignment must try to obtain load-balanced processors to enhance the possibilities of fault-tolerance and power saving. The problem is then reduced to apply the *k*FE-scheduling method to each processor, with the additional advantage that the trade-off point may be different for different processors.

7. Conclusions

The *k*FE-scheduling method, with the double purpose of saving energy and providing fault-tolerance, has been proposed. It is specially useful in embedded systems in which energy supply is critical. The method is based on the use of deterministic redistributed slack, part of which is slated to provide fault-tolerance guarantees; the rest can be used to reduce the nominal frequency by taking advantage of slack slots available after singularities, slots in which all tasks released up to the previous slot have been executed. This leads to the use of multiple frequencies. The trade-off point between both applications, fault tolerance and energy saving, can be set by the designer. As could be expected, the energy savings and the possibilities of recovering faulty tasks are greater for low processor utilisation factors, with larger amounts of slack.

Further frequency reductions can be obtained if: (a) not all faults guaranteed to be recovered actually take place, and (b) tasks execute in less than their worst case time and generate a stochastic slack that can be reclaimed for that purpose. Single frequency methods take advantage of those slacks only by operating an idle processor at 15% of full power at that frequency, a linear reduction; this saving will generally be smaller than that obtained by using the slack to diminish the operating frequency, a quadratic reduction, as *k*FE does.

An upper bound on the overhead imposed on the execution time of each task by switching frequencies is formally proved for RM scheduled real-time systems and taken into account when calculating the schedulability of the system.

Comparative evaluations between *k*FE and Discrete Sys-Clock were performed for different trade-off points and rates of failures. *k*FE generally outperforms DSC, which, in turn, has been reported to show better performance than other multifrequency methods. The results presented in this paper can then be transitively extended. In addition, the method can be used with advantage in multiprocessor or multicore systems if a load-balancing off-line mapping of the set of tasks into the set of processors is previously carried out.

References

[1] W. Wolf, What is embedded computing?, IEEE Computer 35 (1) (2002) 136–137
[2] E. Lee, Absolutely positively on time: what would it take?, IEEE Computer 38 (7) (2005) 85–87
[3] P. Pillai, K.G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in: Proc. 18th Symposium on Operating Systems Principles, 2001, pp. 89–102.
[4] S. Saewong, R. Rajkumar, Practical voltage scaling for fixed priority RT systems, in: Proc. Ninth IEEE Real Time and Embedded Technology and Applications Symposium, 2003, pp. 106–115.
[5] G.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in hard real time environment, ACM 20 (1973) 46–61.
[6] J. Flinn, M. Satianarayanan, Energy aware adaptation for mobile applications, Operating Systems Reviews 34 (5) (1999) 48–63.
[7] S. Fabritus, V. Grigore, T. Maung, V. Loukusa, T. Mikkonen, Towards energy aware system design. <http://www.nokia.com/link?cid=EDITORIAL_1687>, Nokia, 2003.
[8] R.M. Santos, J. Urriza, J. Santos, J. Orozco, New methods for redistributing slack time in real-time systems: applications and comparative evaluations, Journal of Systems and Software 69 (1–2) (2004) 115–128.
[9] R.M. Santos, J. Santos, J. Orozco, A least upper bound on the fault tolerance of real-time systems, Journal of Systems and Software 78 (1) (2005) 47–55.
[10] V. Narayanan, Y. Xie, Reliability concerns in embedded systems designs, IEEE Computer 39 (1) (2006) 118–120.

[11] O. Unsai, I. Koren, C. Krishna, Towards energy-aware software based fault-tolerance in real-time systems, in: Proc. ISLPED'02, Monterrey, California, 2002, pp. 124–129.
[12] R. Melhem, D. Mosse, E.M. Elnozahy, The interplay of power management and fault recovery in real-time systems, IEEE Transactions on Computers 53 (2) (2004) 217–231.
[13] Y. Zhang, K. Chakrabarty, V. Swanminathan, Energy-aware fault tolerance in fixed-priority real-time embedded systems, in: Proc. IEEE International Conference on CAD, 2003, pp. 209–214.
[14] A. Qadi, S. Goddard, A dynamic voltage scaling algorithm for sporadic tasks, in: Proc. 24th International Real-Time Systems Symposium, 2003, pp. 52–62.
[15] A. Mok, Fundamental design problems of distributed systems for the hard real-time environment, Ph.D. thesis, MIT, MIT/LCS/TR-297, 1983.
[16] C. Scordino, G. Lipari, Using resource reservation techniques for power-aware scheduling, in: Proc. Fourth ACM International Conference on Embedded Software, 2004, pp. 16–24.
[17] G. Lipari, S. Baruah, Greedy reclamation of unused bandwidth in constant bandwidth servers, in: Proc. 12th Euromicro Conference on Real-Time Systems, 2000, pp. 193–200.
[18] G. Quan, X.S. Hu, Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors, in: Proc. of IEEE/ACM Design Automation Conference, 2001, pp. 828–833.
[19] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Determining optimal processor speeds for periodical real-time tasks with different power characteristics, in: Proc. 13th Euromicro Conference on Real-Time Systems, 2001, pp. 225–232.
[20] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Dynamic and aggressive scheduling techniques for power-aware real-time systems, in: Proc. IEEE Real-Time Systems Symposium, 2001, pp. 95–101.
[21] D. Zhu, R. Melhem, B. Childers, Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems, in: Proc. IEEE Real Time Systems Symposium, 2001, pp. 84–94.
[22] R. Obenza, Rate monotonic analysis for real-time systems, IEEE Computer 26 (3) (1993) 73–74.
[23] M. Joseph, P. Pandya, Finding response times in a real-time system, The Computer Journal 29 (5) (1986) 390–398.
[24] J. Lehoczky, L. Sha, Y. Ding, The rate monotonic scheduling algorithm: exact characterization and average case behavior, in: Proc. of the IEEE Real-Time Systems Symposium, 1989, pp. 163–171.
[25] J. Santos, J. Orozco, Rate monotonic scheduling in hard real-time systems, Information Processing Letters 48 (1993) 39–45.
[26] R. Maxion, K. Tan, Anomaly detection in embedded systems, IEEE Transactions on Computers 51 (2) (2002) 108–120.
[27] J. Pouwelse, P. Langendoen, H. Sips, Dynamic voltages scaling on a low-power microprocessor, in: Proc. Seventh Conf. on Mobile and Computing and Networking MOBICOM01, 2001, pp. 251–259.
[28] A. Acqaviva, T. Simunic, V. Deolalikar, S. Roy, Server controlled power management for wireless portable devices, Hewlett–Packard Company Document (2003).
[29] C. Poellabauer, K. Schwan, Energy-aware traffic shaping for wireless real-time application, in: Proc. 10th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2004, pp. 48–55.
[30] INTEL, Intel PXA255 Applications Processors Developer's Manual (2005).
[31] J.W. Liu, Real-Time Systems, Prentice Hall, 2000.
[32] N. Balakrishnan, A.P. Basu (Eds.), Exponential Distribution: Theory, Methods and Applications, Gordon and Breach Science Publishers, 1995.
[33] J.H. Anderson, S.K. Baruah, Energy-aware implementation of hard-real-time systems upon multiprocessor platforms, in: Proc. ISCA 16th International Conference on Parallel and Distributed Computing Systems, 2003, pp. 430–435.
[34] B. Andersson, S. Baruah, J. Jonsson, Static-priority scheduling in multiprocessors, in: Proc. IEEE Real Time Systems Symposium, 2001, pp. 193–202.
[35] D. Geer, Chip makers turn to multicore processors, IEEE Computer 38 (5) (2005) 11–13.
[36] S. Cheng, J. Stankovic, K. Ramamritham, Scheduling groups of tasks in distributed hard real-time systems* Coins-87-121, University of Massachusetts, Amherst, 1987.
[37] J. Xu, Multiprocessor scheduling of processes with release times, deadlines, precedence and exclusion relations, IEEE Transactions on Software Engineering 19 (2) (1993) 139–154.
[38] W. Zhao, K. Ramamritham, J. Stankovic, Preemptive scheduling under time and resource constraints, IEEE Transactions on Computers C-36 (8) (1987) 949–960.
[39] K. Tindell, A. Burns, A. Wellings, Allocating hard real-time tasks: an NP-hard problem made easy, Real-Time Systems (4) (1992) 145–165.
[40] J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975.
[41] J. Orozco, R. Cayssials, J. Santos, E. Ferro, Design of a learning fuzzy production system to solve an NP-hard real-time assignment problem, in: Proc. Eighth Euromicro Workshop on Real-Time Systems, 1996, pp. 146–150.

Rodrigo Santos received his Engineering degree in 1997 from Universidad Nacional del Sur and got his Ph.D., degree in Engineering in 2001. He has become a Researcher for The National Research Council in Argentina in 2005 and in the same year became Assistant Professor at the Department of Electrical Engineering and Computers at Universidad Nacional del Sur. His research interests are mainly related to real-time systems: QoS, Multimedia, Operating Systems and Communications. He has published his research's results in international indexed Journals and proceedings of Conferences. He is a member of several Technical Committees for conferences in the area of real-time systems and also a reviewer for several journals. He is the President for the Latin American Center of Studies in Informatics and a member of the Working Group 6.9 of IFIP. He is also an IEEE member.

Jorge Santos is Consulting Professor at the Department of Electrical Engineering and Computers, Universidad Nacional del Sur, Argentina, where he teaches courses on Real-Time Systems and on Professional Communication. He has published more than 70 papers on multivalued logics and their electronic implementation, theory of automata, systems reconfigurations, local area networks, and real-time systems. He has been head of the department and principal researcher of the National Council of Scientific and Technical Research. He is Corresponding Member of the Argentine Academy of Engineering.

Javier Orozco is Professor at the Electrical and Computer Engineering Department, Universidad Nacional del Sur and researcher of the National Council of Scientific and Technical Research Argentina. From 2001, he is the head of the department. His main interest area is in Real-Time Systems theory and applications in operating systems, architectures for embedded systems and communications. He has an active participation in several Scientific and Technical Committees and governmental boards for science and engineering promotion. He has published on local area networks, digital architectures and real-time systems theory and applications.