Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
The Journal of Systems and Software 78 (2005) 47–55
The Journal of Systems and Software
www.elsevier.com/locate/jss

A least upper bound on the fault tolerance of real-time systems

Rodrigo M. Santos *, Jorge Santos, Javier D. Orozco

Departamento de Ingenieria Electrica y Computadoras, Universidad Nacional del Sur/CONICET, Av Alem 1253, 8000 Bahía Blanca, Argentina

Received 31 March 2003; received in revised form 15 November 2004; accepted 28 November 2004
Available online 24 December 2004

Abstract

This paper presents a method to deal with the reexecution of tasks in a hard real-time system subject to temporary faults. The set of tasks follows the Liu and Layland model: they are periodic, independent and preemptible. Time is considered to be slotted. The system is said to be k-schedulable if it is schedulable in spite of the fact that in the interval between its release and its deadline, every task admits that k slots are devoted to uses other than its first execution. In this case, the k slots are used to reexecute tasks subject to temporary faults. Since the value of k can be easily determined, a least upper bound on all the possible combinations of faults that the system can tolerate while meeting the hard time-constraints, follows immediately. The method is bandwidth preserving and the expression of the bound is a diophantic inequality relating k, the execution time and the period of each task. The method is compared to methods proposed by other authors to solve the same problem and it is evaluated through extensive simulations performed on random generated sets of tasks.
© 2004 Elsevier Inc. All rights reserved.

Keywords: Real-time systems; Fault-tolerance; Schedulability

1. Introduction and motivation

Real-time systems are those in which results must not only be correct from an arithmetic-logical point of view but also produced before a certain instant, called *deadline*. If no deadline can be missed, the system is said to be *hard* as opposed to *soft* in which some deadlines may be missed. When all time constraints are met, the system is said to be *schedulable*.

Real-time systems, as any other human engineered system, may be subject to functional faults. Until a few years ago typical applications of hard real-time systems were restricted to cases in which missing a deadline could have severe consequences including the loss of human lives. Later, the frontier between hard and soft systems shifted in order to incorporate more applications into the hard realm (e.g. video games and multimedia servers). However it is obvious that significant differences persist in the consequences: a systematic malfunctioning of a video game may result, at most, in a commercial loss whereas a sporadic malfunctioning of a space probe may result not only in a huge economic loss but also in the loss of many years of efforts. The Cassini probe to Saturn and its 30^{+} moons, for example, took seven years to arrive in the vicinity of the planet and cost US$ 300 millions. In voyages of this type, temporary disturbances may be caused, for instance, by protons and cosmic rays (Campbell et al., 1992). Therefore, it is particularly important to be able to determine the system's fault tolerance understood as the combination of faults that do not preclude its performance according to real-time specifications. That is the motivation behind this paper.

Faults may be grouped in two classes: permanent and temporary. If they are corrected, some form of redundancy, either spatial or temporal, is always used

* Corresponding author. Tel.: +54 291 459 5181; fax: +54 291 459 5154.
E-mail address: ierms@criba.edu.ar (R.M. Santos).

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.11.021

(Liberato et al., 1999). Permanent faults are long lasting faults usually caused by hardware malfunctioning that cannot be remedied; they are corrected by using spatial redundancy such as spare units. Temporary faults, commonly short and much more frequent than permanent ones, can also be corrected by means of spatial redundancy: hardware modules are triplicated and results voted. The voting circuitry is also triplicated. However, this kind of faults is more often corrected by employing temporal redundancy in the form of the recomputation of the failed task (Gosh et al., 1998).

Because of their generalised use in real-time applications, single processor systems handling sets of periodic, independent and preemptible tasks are of particular interest. The least common multiple of the periods is called the *hyperperiod.* Since the execution time and the period of each task are known, these systems are deterministic and it is possible to calculate the time necessary for the normal processing of all the tasks, which will be the same in each hyperperiod. The remaining time, called *slack* can be devoted to other chores, like, for example, recovering from a fault. Therefore, every method designed to make a system fault tolerant by temporal redundancy consists essentially in making the best use of the available slack.

In this paper a method to deal with the reexecution of tasks subject to temporary faults is presented. Basically it is a tool to redistribute slack and, as such, it was previously applied to solve other problems, like, for example, scheduling mixed sets in which hard deterministic periodic tasks share the processor with non-hard stochastic aperiodic tasks or scheduling sets of tasks composed of hard mandatory and reward-based optional subtasks (Santos et al., 2004). In this paper, the tool is used to determine a least upper bound on the possible combinations of tolerated faults.

The take-off point of this paper, defined as the latest findings by previous authors, is made clear in Section 2. In Section 3, the redistribution of slack in a system operating under the Rate Monotonic priority discipline is analysed and the notions of *k-schedulability* and *singularity*, conceptual bases of the method, are introduced. In Section 4, the new approach is applied to the calculation of the least upper bound on the fault-tolerance of the system; the theoretical foundation is formally proved. In Section 5, extensive simulations and their results to test the performance of the method are presented. Finally, in Section 6, conclusions are drawn.

2. Related work: the take-off point

In what follows the related work to be considered cover methods proposed by other authors to recover faults using time redundancy. The take-off point of this paper consists in the latest findings by those authors.

Pandya and Malek (1998), for example analysed the tolerance to a single fault of a real-time Rate Monotonic scheduled set of tasks. When the fault occurs, all unfinished tasks are executed. They also proved that the recovery is always possible if the total utilization factor is less than or equal to 0.5, a very stringent condition.

Burns et al. (1996) presented an exact schedulability test for fault tolerant sets of real-time tasks under specified failure hypothesis. The test gives guarantees even for sets with a total utilization factor higher than 0.5. The method, however, does not provide a general bound valid for all cases, and each combination of tolerated faults requires a separate schedulability analysis.

Gosh et al. (1998) derived upper bounds for the utilization factor of fault tolerant real-time sets of tasks, Rate Monotonic scheduled, for different faults. The bounds are sufficient although not necessary and are in general lower than those obtained with the method proposed in this paper.

Servers, originally proposed to improve the response time of aperiodic tasks sharing the processor with the real-time set (Sprunt et al., 1989; Strosnider et al., 1995), can also be used for recovering failed tasks. However, they provide bounds with lower utilization factors than those provided by the methods presented by Gosh et al. (1998).

Ramos-Thuel and Strosnider (1995) extended the analysis presented in (Lehozcky et al., 1989) to provide fault tolerance. Basically this is done by adding the recovery load to the normal load as specified in the set of real-time tasks. The schedulability test to this augmented set is then applied. Again, each combination of faults requires a separate schedulability analysis and no general bound is given.

The method herein proposed gives a worst-case bound on all the possible combination of faults tolerated by the system while meeting the time constraints. The bound is higher than those provided by other methods. Slack is redistributed by k–scheduling the real-time set. Combinations that include multiple recoveries for critical or very critical tasks can be easily explored.

3. Rate monotonic schedulability and the redistribution of slack

When two or more periodic tasks compete for the use of the processor, some rules must be applied to allocate its use. This set of rules is called *priority discipline.* In a static fixed priority discipline all tasks are assigned a priority once for all. If tasks are ordered by decreasing rates or, what is the same, by increasing periods, the discipline is called *Rate Monotonic*, notated RM. The task with the shortest period has the highest priority. Some additional rule must also be provided to break ties.

Liu and Layland (1973) proved that Rate Monotonic is optimal among the Fixed Priority disciplines. It is supported by the US Department of Defence and consequently adopted, among others, by IBM, Honeywell, Boeing, General Electric, General Dynamics, Magnavox, Mitre, NASA, Naval Air Warfare Center, Paramax and McDonnell Douglas (Obenza, 1993). No doubt it is a de facto standard, at least in the US, and hence the importance of its use in research papers whose results may be immediately applied in technical developments.

Several methods have been proposed for testing the RM schedulability of real-time systems (Liu and Layland, 1973; Joseph and Pandya, 1986; Lehozcky et al., 1989). All have in common the fact that the ordering relation to define priorities is based on increasing periods, with some additional rules to break ties. In the Empty Slots method (Santos and Orozco, 1993), a discretization of (Joseph and Pandya, 1986), time is considered to be slotted and the duration of one slot is taken as the unit of time. Slots are notated t and numbered 1, 2, ... The expressions *at the beginning of slot t* and *instant t* mean the same. Tasks are preemptible at the beginning of slots. As proved in (Liu and Layland, 1973), the worst case of load occurs when all tasks are released simultaneously at $t = 1$.

The sets of tasks to be processed follow the Liu and Layland model: they are periodic, independent and preemptible. A set S(n) of n tasks, ordered by decreasing rates, is completely specified as $S(n) = \{(C_1, T_1, D_1), (C_2, T_2, D_2), \ldots, (C_n, T_n, D_n)\}$, 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 task i, denoted τ_i, respectively. It should be noted that T_1 and T_n denote the minimum and the maximum periods, respectively. For the sake of simplicity in the analises that follow it is assumed that tasks are released at the beginning of slots, that execution times, periods and deadlines are multiples of the slot time and that deadlines are equal to periods. These restrictions can be easily relaxed.

Santos and Orozco (1993) formally proved that S(n) is RM-schedulable iff

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

where $\lceil\,\rceil$ denotes the monadic ceiling operator, that is the smallest integer equal to or larger than t/T_h. The condition is intuitively clear: τ_i can be added to the system of $(i-1)$ tasks keeping the expanded system schedulable if and only if before its deadline there are enough empty slots to execute it and to give way to all the instantiations of tasks of higher priority. The last term in the right hand member is called the *work function*, denoted $W_{i-1}(t)$. It should be noted that in order to test the schedulability of the system, it suffices to test it in the interval $[1, T_n]$ because it is the most congested after the simultaneous release of all tasks at $t = 1$. The Least Common Multiple of the periods, the hyperperiod, shall be notated M. If $M = T_n$, the initial conditions are repeated every T_n slots. If $M > T_n$, the load decreases in $[T_n + 1, M]$, but in both cases $[1, T_n]$ is the critical interval.

The expression $M - W_n(M)$ gives the number of empty slots in the hyperperiod. Slots go empty only when there are not tasks with pending execution. Empty slots appearing in this way are called *background* slots. Eventually some background slots can be advanced in time, producing a redistribution of slack without jeopardizing the execution of the real-time set.

Example. Let S(5) be the system specified in Table 1.

In Fig. 1, the evolution of the system is depicted. The priority of each task decreases with increasing periods.

The schedulability test is recursive. It starts with τ_1. Since there are not tasks of higher priority, the first condition is

for $i = 1 \quad T_1 \geqslant$ least $t \mid t = C_1 = 1$

Since $T_1 = 6 \geqslant 1$, the subsystem $\{\tau_1\}$ is schedulable.

for $i = 2 \quad T_2 \geqslant$ least $t \mid t = C_2 + \lceil t/6 \rceil = 2 + \lceil t/6 \rceil$

The first slot meeting the condition is $t = 3$. Since $T_2 = 10 \geqslant 3$, the subsystem $\{\tau_1, \tau_2\}$ is schedulable, etc.

Since

$$W_5(30) = \lceil 30/6 \rceil + 2\lceil 30/10 \rceil + \lceil 30/15 \rceil + 2\lceil 30/15 \rceil + \lceil 30/15 \rceil = 19$$

$M - W_5(30) = 11$ is the number of empty background slots in the hyperperiod. As can be seen in the figure, they are the slots 9, 10, 14, 15, 23, 24 and 26–30. Also in the figure it is shown that the interval [1,8] is saturated, no background empty slots appear in it and therefore no recovery could be made in a conventionally RM scheduled system.

A hard real-time system S(n) is said to be *k-RM schedulable* if it is RM schedulable in spite of the fact that in the interval between its release and its deadline, each task admits that k slots are devoted to uses other than the execution of the original set. Santos et al. (2004) proved that a system S(n) is k-RM schedulable iff

$$\forall i \in (1, 2, \ldots n) \quad T_i \geqslant \text{ least } t \mid t = C_i + k + W_{(i-1)}(t) \tag{2}$$

Table 1
Specification of the system

i	C_i	T_i
1	1	6
2	2	10
3	1	15
4	2	15
5	1	15

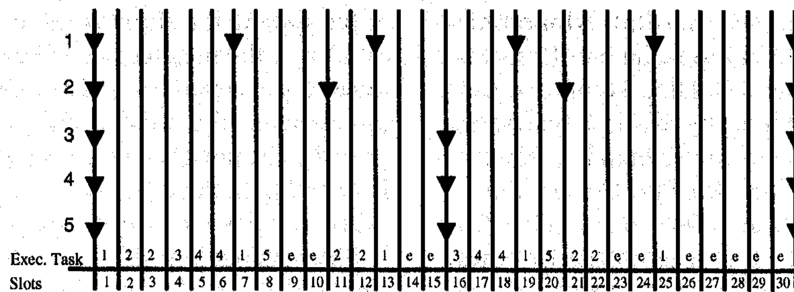

Fig. 1. Evolution of the system. e denotes an empty slot The tie between the last three tasks is broken by ordering them by name (τ_3, τ_4 and τ_5).

It must be noted that k is the minimum value in the set $\{k_i\}$ where k_i is defined as

$$k_i = \max k \mid T_i \geqslant \text{ least } t \mid t = C_i + k + W_{(i-1)}(t)$$

The complexity of the test for the k-schedulability is the same as that for the RM-schedulability that in (Santos and Orozco, 1993) has been proved to be $O(n^*T_n)$, where T_n denotes the maximum period in the system.

Example. By application of the previous definitions, in the system of the example above, $k_1 = 5$, $k_2 = 8$, $k_3 = 7$, $k_4 = 5$ and $k_5 = 4$. Therefore, $k = 4$.

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

Santos et al. (2004) proved that if a hard real-time system $S(n)$ is k-RM schedulable, k slots of an interval $[s, (s+k-1)]$ can be used to execute tasks $\notin S(n)$. In that case, the tool was tuned to optimize the execution of aperiodic non-real-time tasks by advancing their execution as much as possible in order to reduce the average delay.

When studying fault tolerance, recomputed tasks can also be conceptually considered to belong to a set different from the original one and therefore $\notin S(n)$. They can be executed in the k slots available from a singularity but not necessarily immediately after it. In fact, what is expressed in (2) is that they can be used at any time between the release and the deadline of the task. The method is therefore bandwidth preserving in the sense that slots not used immediately after the singularity are not lost but can be used later.

4. The new approach

4.1. Theoretical foundation

As explained above, the expression $M - W_n(M)$ gives the number of empty slots, commonly designated *slack*, in the hyperperiod. The total slack depends entirely on the values of M and $W_n(M)$ and it is the only time available to repeat the processing of failed tasks in all methods based on the use of time redundancy. Although no scheduling method can increase it, it can be redistributed to be used where it serves best with the proviso that the time-constraints of the original system are met. The efficiency of each method depends, essentially, on how much of the slack is used and when it is used.

Since a least upper bound, LUB, is sought, worst case conditions must be determined. The worst case assumptions are:

(a) In the interval $[1, T_n]$, there is only one singularity, the one taking place in the slot immediately before the worst case of load.
(b) The failure is detected at the end of the failed task's execution.

The first assumption means that in $[1, T_n]$ no other singularity appears and therefore only k slots are available for recovering failed tasks in the interval. The second assumption means that the failed task has consumed all the slots available for its normal execution and none of them can be used for recovery.

The recovery execution takes place immediately after the failed one subject to the availability of reshuffled slack. Faults can be detected either explicitly (through a pattern recognition in which a signature is associated to a particular fault) or implicitly (by some indirect indicator caused by the fault). Any further discussion is beyond the scope of this paper but obviously the detection process itself must be highly dependable (Maxion and Tan, 2002).

In what follows the symbol $\lfloor\,\rfloor$ denotes the monadic floor operator, that is the biggest integer equal to or smaller than the quotient within the symbol. The proposed method will now be formalised:

Lemma 1. $\lfloor k/\lceil T_n/T_i \rceil \rfloor = R_i$ *represents the number of slots available to recover* τ_i *in each of its instantiations in the interval* $[1, T_n]$.

Proof. The number of τ_i's instantiations in $[1, T_n]$ is $\lceil T_n/T_i \rceil$. If the total number of slots available for recovery in the interval is k, the expression follows

immediately. The floor is used because no fractional slots can be used at any period. □

Example. In the example of the previous section, $k=4$, $T_1=6$ and $T_n=T_5=15$. The number of τ_1's insantiations is $\lceil T_5/T_1\rceil=3$ The number of slots available for recovering failed executions of τ_1 in each of its instantiations is therefore $R_1=\lfloor 4/3\rfloor=1$. The quotient is rounded down because slots cannot be fractioned.

Lemma 2. *If $R_i \geqslant C_i$, τ_i may fail once in every instantiation and be recovered, although only C_i slots per instantiation will be needed.*

Proof. R_i, the number of slots available for recovery in each instantiation of τ_i, is larger than or equal to C_i, the time needed to execute it. Therefore the execution of the task may fail at least once per instantiation and still be recovered. □

Example. In the previous example, $R_1=1$ and $C_1=1$. Therefore the execution of τ_1 may fail once in each of its instantiations and still be recovered because there is enough available time to do it.

Lemma 3. *If $R_i < C_i$, the recovery can be made in only*

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

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

Proof. $R_i < C_i$ means that in each instantiation, the time available for recovery is smaller than the time needed to reexecute the task. Therefore, the task cannot be recovered in every instance but only in p_i of them, subject to the condition that the number of available slots is enough to execute at least one recovery. □

If $R_i < C_i$, the recovery execution time used in the calculations will be R_i, otherwise it will be C_i. Therefore, if C_i^r denotes the recovery execution time used in the calculations, $C_i^r=\min[R_i, C_i]$.

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

$$\Sigma_1^n C_i^r q_i \leqslant k \quad (3)$$

subject to the condition $0 \leqslant q_i \leqslant \min[p_i, \lceil T_n/T_i\rceil]$, indicates the combination of faults that the system can tolerate.

Proof. Immediate from the previous lemmas. □

The above is the fundamental expression of the method. It is a diophantic inequality but it also can be seen as an n-dimensional surface which is a least upper-bound on the combinations of faults tolerated by the system. If in spite of increasing the maximum q corresponding to a given task, the inequality still holds, solutions with multiple faults of some tasks in all or some instantiations can be explored.

Gosh et al. (1998) give the conditions to ensure recovery:

(1) In order to recover a task τ_i, failed at its instantiation j ($j=1,2,\ldots$) at least C_i slots must be available in the interval $[jT_i, (j+1)T_i]$.
(2) The recovery of a failed task must be performed before its deadline.
(3) The recovery of a task must not cause any other task to miss its deadline.

The fundamental expression of the proposed method (3) goes beyond the first condition because it makes possible the determination of the LUB on all the combinations of faults that the system can tolerate. The basic k-schedulability inequality (2) allows the calculation of k, a fundamental parameter in determining the upper bound. If (2) holds, the second and third conditions above are met.

Example. In the next two examples, the same set of tasks of Section 3 is used.

Calculated k_i, C_i^r and p_i are given in the fourth to sixth columns in Table 2. The possible values of q_i are presented in the last column. $k=\min\{k_i\}=4$. Consequently, the system can tolerate failures in any combination satisfying

$$q_1 + 2q_2 + q_3 + 2q_4 + q_5 \leqslant 4$$

Obviously, not all tasks may fail in all their instantiations in $[1, T_n]$. Some tolerated combination of faults in $[1, 15]$, are given in the following examples.

Example. If $q_1=3$, $q_2=0$, $q_3=1$, $q_4=0$, $q_5=0$, τ_1 and τ_3 may fail in all their instantiations in the interval. Other combinations of tasks that may fail in all their instantiations in the interval are τ_1 and τ_5; τ_3, τ_4 and τ_5.

Example. If $q_1=1$, $q_2=1$, $q_3=1$, $q_4=0$, $q_5=0$, τ_1 and τ_2 may fail in one instantiation each and τ_3 in its only instantiation in the interval, etc.

Table 2
Number of instantiations in which each task may fail once and be recomputed

i	C_i	T_i	k_i	C_i^r	p_i	q_i
1	1	6	5	1	3	0–3
2	2	10	6	2	2	0–2
3	1	15	7	1	1	0–1
4	2	15	5	2	1	0–1
5	1	15	4	1	1	0–1

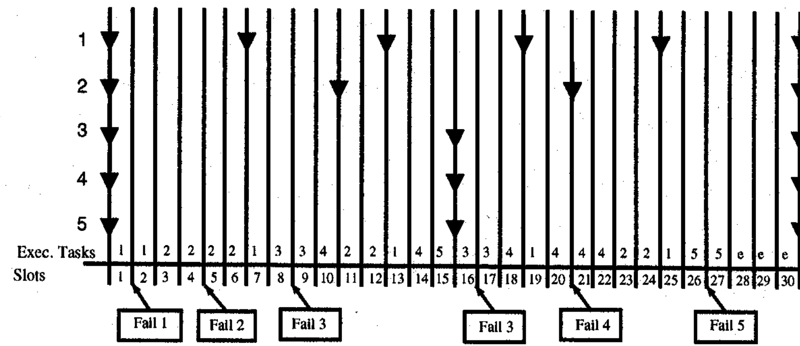

Fig. 2. The evolution of the system under some fault conditions.

In the first half of Fig. 2, the last case is shown. As can be seen, τ_1 fails in its first instantiation (at $t = 1$) and it is recovered at $t = 2$; τ_2 also fails in its first instantiation (at $t = 3, 4$) and it is recovered at $t = 5, 6$. τ_3 fails in its first instantiation (at $t = 8$) and it is recovered at $t = 9$. The four advanced slack slots corresponding to the least upper bound have then been used.

It should be noted that if no fault occurs, four empty background slots appear at $t = 9$, 10, 14 and 15, as shown in Fig. 1. What the method does is to advance the four slots and make them available for recoveries at $t = 1$, a singularity because the previous slot is an empty one. Since the use of a fifth slot would entail that some task misses its deadline, four slots is the maximum slack usable for recoveries in $[1, 15]$. In this example, therefore, the method optimises their number and their position and gives the best guarantee as a least upper bound in the interval.

Example. Let S(3) be the set of tasks specified in Table 3.

Because $p_1 = 0$, τ_1 can never be recovered. Because $R_2 < C_2$, $p_2 = 1$ and $R_2\lceil T_3/T_2\rceil = C_2$, τ_2 can be recovered in one out of its two instantiations in [1, 15].

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 $[(j \times T_n + 1), (j + 1)T_n]$ up to $(j + 1)T_n = M(j = 1, 2, \ldots)$. In the example of Fig. 2, for instance, at $t = 16$ another singularity takes place because the last hard pending task is executed at $t = 15$. Consequently, another four slots are again available for recovery purposes. They could be used again to recover failed executions of τ_1, τ_2 and τ_3 or in alternative schemes, for instance to recover all failed second instantiations of τ_3, τ_4 and τ_5, as shown in the second half of the figure.

Table 3
Specification of the system and number of recomputable instantiations

i	C	T	k	R_i	p_i	q_i
1	4	8	4	1	0	0
2	2	8	2	1	1	0–1
3	1	16	3	2	1	0–1

In some cases, tasks may be very critical and require multiple recoveries. For instance, if interferences with the real-time system of an orbiting satellite are long enough, they may affect not only the normal instantiation but also subsequent recoveries. In that case, allowances should be made for the tolerance of several faults after the same instantiation.

Example. In the example above, by setting $q_3 = 3$, the expression indicates that, within $[1, 15]$, the system tolerates three failed executions of τ_3 with recovery at the fourth one. The same applies in $[16, 30]$.

It should also be noted that failures in $[1, T_n]$ may occur after one or more empty slots have appeared naturally as background slots. Since an empty slot is also a singularity, k slots are again available for recoveries after it and therefore some recoveries may take place after $t = T_n$.

Example. In the previous example, τ_3, τ_2 and τ_1 may fail in that order at $t = 4$, 12 and 16, respectively, after an empty background slot appeared at $t = 10$. τ_3 is recovered at $t = 5$, τ_2 at $t = 13$ and 14, and τ_1 at $t = 16$ (after $t = T_n = 15$).

The LUB is pessimistic. k denotes only the number of redistributed empty-slots, available for recovering failed tasks in the critical period $[1, T_n]$. The fault-tolerance, however, can increase because of two reasons:

(1) The expression $T_n - W_n\ (T_n)$ gives the number of empty slots in $[1, T_n]$. It is possible that $k < T_n - W_n(T_n)$ and, in that case, the empty slots not redistributed in the critical period but appearing merely as background empty slots could also be used for recovering tasks.
(2) At the instants $jM + 1, j = 1, 2, \ldots,$ the worst case initial state of the hard real-time set of tasks is repeated, followed by a critical interval $[jM + 1, jM + T_n]$. When simulations are performed with random fault generations, the faults may take

place outside the critical intervals, with a lighter load, and consequently with more empty slots available for recovering tasks.

Example. In the example of Fig. 2, in spite of recovering one single-failed second instantiation of τ_3, τ_4 and τ_5, the non-critical interval [16,30] leaves three unused background empty slots at $t = 28$, 29 and 30. As can be seen in Fig. 3, the three tasks may actually fail twice in the interval and still be recovered. The LUB was pessimistic because, after the singularity $s = 15$, it counted only the redistributed slack, four slots, instead of the seven actually available when the background slots are counted in.

4.2. Comparison with other time-redundancy methods

According to the upper bound calculated with the method proposed by Gosh et al. (1998), the system tolerates only successive single faults in τ_3 or τ_4 or τ_5, separated by at least 30 slots.

By applying the empty slots method in a manner similar to that used by Burns et al. (1996) or by Ramos-Thuel and Strosnider (1995), exact calculations could be done using increased execution times (due to recoveries) and less pessimistic results would be obtained in certain cases. In order to recover a task that failed once in all its instantiations, its execution time should be duplicated. For instance, to test the tolerance of τ_2 to that type of faults, the expression (1) should be calculated with $C_2 = 4$. In general, to test the tolerance to faults in one task or in a combination of tasks, the expression (1) must be recalculated with the corresponding execution times augmented. On the contrary, to the best of the authors' knowledge, the method, as previously presented, is the only one that in one single expression gives a least upper bound on all the possible combinations of faults tolerated by the system.

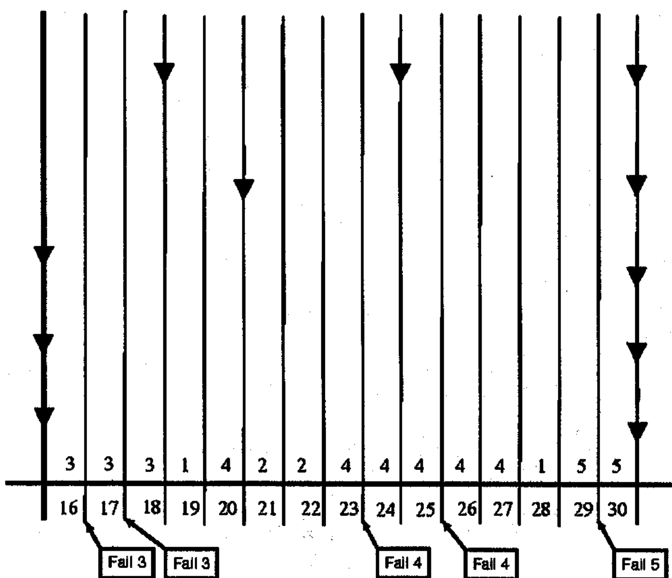

Fig. 3. The evolution of the system with some double faults in [16, 30].

5. Performance evaluation

5.1. Simulations

In what follows, the evaluation process is described to the extent that its correctness can be validated and the results replicated.

The metric used to asses the quality of service, QoS, provided by the method is the success ratio, SR, defined as the average ratio between the number of recovered tasks and the total number of faulty tasks. Evidently, SR can be seen as a figure of merit: a value of 100 means, of course, that all faulty tasks were recovered.

The two main variables affecting the SR are the Utilization Factor, UF, and the Mean Time Between Failures, MTBF. Thus they were chosen as independent variables. In order to determine the performance of the method, systematic simulations were conducted on about 10,000 randomly generated sets of tasks. Each set had 10 tasks; their periods had a uniform distribution; the sample space was [10, 20, ..., 100]. Execution times were randomly generated in such a way that the system utilization factor had a uniform distribution over the sample space [0.30, 0.31, 0.32, ..., 0.90]. Only RM schedulable systems were accepted. For each UF and each MTBF, about 165 sets were tested. Each of them was run for approximately 100,000 slots.

For the same set, faults were generated following an exponential distribution, commonly used in this kind of evaluations (Ramos-Thuel and Strosnider, 1995).

5.2. Results

The results obtained are shown in Fig. 4. A bidimensional parametric representation was chosen because it was found to be the clearest. SR is represented vs. UF (in the range 0.30–0.90 in steps of 0.01) for four values of MTBF used as parameters. Following Gosh et al. (1998) these were (measured in slots): (a) the average period, 50; (b) the maximum period, 100; (c) five times the maximum period, 500; (d) ten times the maximum period, 1000. For an MTBF = 50, for example, about 2000 faults were randomly generated for each set in each run.

5.3. Analysis

The Success Ratio starts diverging significantly from 100% only for utilization factors over 0.60 in all cases. This could be expected since low UFs mean that many empty slots are available (for UFs under 0.50, there

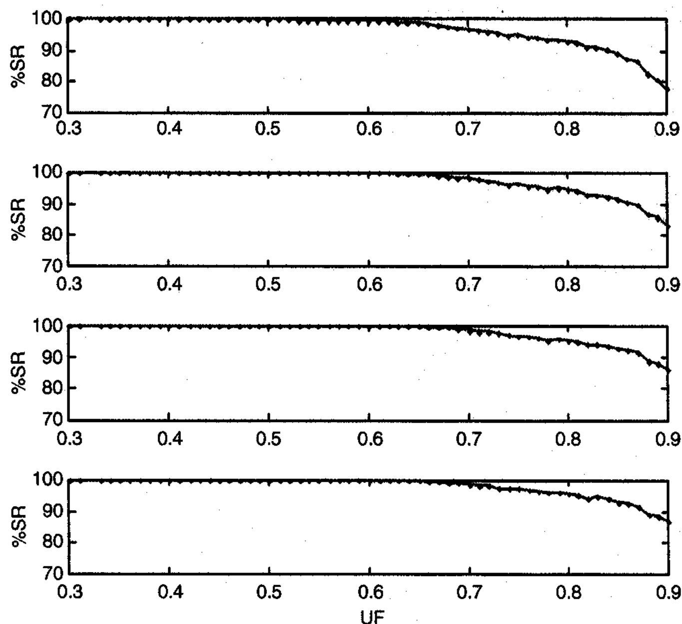

Fig. 4. Success ratio vs. utilization factor for different MTBFs (from top down, 50, 100, 500 and 1000 slots).

are more empty than busy slots). Because there are so many, empty slots appear early in the process and are immediately available for recoveries. As the UFs increase over 0.60, fewer empty slots are available and recoveries are more difficult.

As can be seen, the lowest SR is obtained for MTBF = 50 and UF = 0.9, rather stringent conditions since, firstly, an average of one failure takes place in each average period and, secondly, the slack amounts to only 10% of the whole time. However, the SR is approximately 78%, meaning that only 22% of the failures are not recovered. It is reasonable to assume that the advancement of the slack making it available for recoveries since the very beginning of the processing and its bandwidth conservation property play an important role in that result.

For the same UF and an MTBF = 1000, the lowest frequency of simulated failures, the SR grows up to 86% (only 14% of the failed tasks are not recovered). As can be seen, although for a given UF the total available slack is always the same, as the frequency of faults decreases an improvement in performance takes place. This can be explained by the fact that some failed tasks use empty slots that in the case of more frequent faults are taken up for the recovery of previous immediate failed tasks. Again, it seems reasonable that the slack reshuffling and the bandwidth conservation are contributing causes for that result.

6. Conclusions

The method presented here is based on a redistribution of slack time trying to put it where its use can be more beneficial. The method rests on the theoretical notions of k-schedulability and singularity. It produces a least upper bound contained in an n-dimensional surface, n being the number of tasks in the real-time system. It is shown that the conditions to ensure recovery of any combination of faults below the surface are met. To the best of the authors' knowledge, it is the only method that with only one single expression allows the exploration of all the possible combinations of single or repetitive faults that the system tolerates in the interval between the beginning of the execution and the maximum tasks' period. The worst case of load, simultaneous generation of all tasks at the initial instant, is assumed. Because that interval is the most critical, the bound is pessimistic relative to faults tolerated over larger intervals, like for example the hyperperiod.

Simulations were carried out for exponentially distributed stochastic faults with four Mean Time Between Failures (the average tasks' period, the maximum period and five and ten times the maximum). Success ratios (the ratio between recovered failures and total number of failures) are represented vs. the system's utilization factor using the MTBFs as parameters.

The curves start diverging significantly from 100% only for utilization factors over 0.6 in all cases. For an UF = 0.9, the success ratio is 78% in the case of the more frequent failures and it increases up to 86% for the highest value of the MTBF. It seems reasonable to assume that the advancement of the slack, making it available for recoveries since the very beginning of the processing, and its bandwidth conservation property play an important role in that result.

Acknowledgement

The authors want to express their sincere appreciation to the anonymous referees for all their comments and suggestions, which have substantially improved the paper.

References

Burns, A., Davis, R., Punnekkat, S., 1996. Feasibility analysis of fault-tolerant real-time tasks sets. In: Proceedings of the 8th Euromicro Workshop on Real-Time Systems, pp. 29–33.

Campbell, A., McDonald, P., Ray, K., 1992. Single event upset rates in space. IEEE Transactions on Nuclear Science 39 (6), 1828–1835.

Gosh, S., Melhem, R., Mossé, D., Sen Sarma, J., 1998. Fault-tolerant rate-monotonic scheduling. Real Time Systems Journal 15, 149–181.

Joseph, M., Pandya, P., 1986. Finding response times in a real-time system. The Computer Journal 29 (5), 390–395.

Lehozcky, J., Sha, L., Ding, Y., 1989. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour.

In: Proceedings of the Real Time Systems Symposium, pp. 166–171.

Liberato, F., Lauzac, S., Melhem, R., Mossé, D., 1999. Fault-tolerant real-time global scheduling on multiprocessors. In: Proceedings 11th Euromicro Conference on Real-Time Systems, New York, pp. 252–259.

Liu, C.L., Layland, J., 1973. Scheduling algorithm for multiprograming in a hard real-time environment. Journal of the ACM 20 (1), 46–61.

Maxion, R., Tan, K., 2002. Anomaly detection in embedded systems. IEEE Transactions on Computers 51 (2), 108–120.

Obenza, J., 1993. Rate monotonic analysis for real-time systems. IEEE Computer 26 (3), 73–74.

Pandya, M., Malek, M., 1998. Minimum achievable utilization for fault-tolerant processing of periodic tasks. IEEE Transactions on Computers 47 (10), 1102–1112.

Ramos-Thuel, S., Strosnider, J.K., 1995. Scheduling fault recovery operations for time-critical applications. In: Proceedings of the 4th IFIP Conference on Dependable Computing for Critical Applications.

Santos, J., Orozco, J., 1993. Rate monotonic scheduling in hard real-time systems. Information Processing Letters 48, 39–45.

Santos, R.M., Urriza, J., Santos, J., Orozco, J., 2004. New methods for redistributing slack time in real-time systems: applications and comparative evaluation. The Journal of Systems and Software 69, 115–128.

Sprunt, B., Sha, L., Lehoczky, J., 1989. Aperiodic task scheduling for hard real-time systems. Real-Time Systems Journal 1 (1), 27–60.

Strosnider, J.K., Lehoczky, J., Sha, L., 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Transactions on Computers 44 (1), 73–91.