Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems

Alan A. Bertossi, Luigi V. Mancini, and Federico Rossini

Abstract—Hard-real-time systems require predictable performance despite the occurrence of failures. In this paper, fault tolerance is implemented by using a novel duplication technique where each task scheduled on a processor has either an active backup copy or a passive backup copy scheduled on a different processor. An active copy is always executed, while a passive copy is executed only in the case of a failure. First, the paper considers the ability of the widely-used Rate-Monotonic scheduling algorithm to meet the deadlines of periodic tasks in the presence of a processor failure. In particular, the Completion Time Test is extended so as to check the schedulability on a single processor of a task set including backup copies. Then, the paper extends the well-known Rate-Monotonic First-Fit assignment algorithm, where all the task copies, included the backup copies, are considered by Rate-Monotonic priority order and assigned to the first processor in which they fit. The proposed algorithm determines which tasks must use the active duplication and which can use the passive duplication. Passive duplication is preferred whenever possible, so as to overbook each processor with many passive copies whose primary copies are assigned to different processors. Moreover, the space allocated to active copies is reclaimed as soon as a failure is detected. Passive copy overbooking and active copy deallocation allow many passive copies to be scheduled sharing the same time intervals on the same processor, thus reducing the total number of processors needed. Simulation studies reveal a remarkable saving of processors with respect to those needed by the usual active duplication approach in which the schedule of the non-fault-tolerant case is duplicated on two sets of processors.

Index Terms—Fault tolerance, hard-real-time systems, multiprocessor systems, periodic tasks, rate-monotonic scheduling, task replication.

1 Introduction

Throughout industrial computing, there is an increasing demand for more complex and sophisticated hard-real-time computing systems. In particular, fault tolerance is one of the requirements that are playing a vital role in the design of new hard-real-time distributed systems.

A variety of schemes have been proposed to support fault-tolerant computing in distributed systems, such schemes can be partitioned into two broad classes. In the first class, which employs the *passive replication* techniques, a passive backup copy of a primary task is assigned to one or more backup processors; when a primary task fails, the passive copies of the task are restarted on the backup processor, hence a passive copy is executed only in the presence of a failure. In the second class, which employs the *active replication* techniques, the same set of tasks is always executed on two or more sets of processors; every primary task has an active backup copy: if any task fails, its mirror image will continue to execute.

Many hard-real-time scheduling problems have been found to be NP-hard: most likely, there are no optimal polynomial-time algorithms for them [2], [11]. In particular, scheduling periodic tasks with arbitrary deadlines is NP-hard, even if only a single processor is available [12]. Several heuristics for scheduling periodic tasks on uniprocessor and multiprocessor systems have been proposed. Liu and Layland [10] introduced the Rate-Monotonic (RM) algorithm for preemptively scheduling periodic tasks on a single processor, under the assumption that task deadlines are equal to their periods. Joseph and Pandya [5] later derived the Completion Time Test (CTT) for checking schedulability of a set of fixed-priority tasks on a single processor. RM was generalized to multiprocessor systems by Dhall and Liu [3], who proposed, among others, the Rate-Monotonic First-Fit (RMFF) heuristic. More refined heuristics for multiprocessors were proposed by Burchard, Liebeherr, Oh, and Son [1].

It is worth noting that the RM algorithm is becoming an industry standard because of its simplicity and flexibility. It is a low overhead greedy algorithm, which is optimal among all fixed-priority algorithms. Moreover, it possesses certain advantages, for example, the implementation of efficient schedulers for aperiodic tasks, and the retiming of intervals in order to shed the load in a predictable fashion [8].

As for fault-tolerant scheduling algorithms, a dynamic programming algorithm for multiprocessors was presented in [7] which ensures that backup schedules can be efficiently embedded within the primary schedule. An algorithm was proposed in [9] which generates optimal schedules in a uniprocessor system by employing a passive replication to tolerate software failures only. The algorithms proposed in [14] are based on a bidding strategy and dynamically recompute the schedule when a processor

- *A.A. Bertossi is with the Dipartimento di Matematica, Università di Trento, Via Sommarive 14, 38050 Trento, Italy. E-mail: bertossi@science.unitn.it.*
- *L.V. Mancini is with the Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza," Via Salaria 113, 00198 Roma, Italy. E-mail: lv.mancini@dsi.uniroma1.it.*
- *F. Rossini is with Telecom Italia Mobile, Area Applicazioni Informatiche, Via Tor Pagnotta 90, 00143 Roma, Italy.*

Manuscript received 20 June 1997.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105271.

fails, in order to redistribute the tasks among the remaining nonfaulty processors. In [13], two algorithms are designed which reserve time for the processing of backup tasks on uniprocessors running fixed-priority schedulers. Finally, the techniques of backup overbooking and backup deallocation were introduced in [4] to achieve fault tolerance in multiprocessor systems, but for aperiodic nonpreemptive tasks only.

It is noted here that none of the fault-tolerant algorithms discussed above extended the RMFF algorithm or have combined in the same schedule both active and passive replication of the tasks. However, the latter idea seems potentially useful since it provides the ability to exploit the advantages of both types of replication in the same system.

Indeed, the simplest way to achieve fault tolerance in hard-real-time systems consists in using active duplication for all tasks. An active copy presents the advantages of requiring no synchronization with its primary copy—it can run before, after, or concurrently with the other copy—and of having a larger time window for execution—namely, the whole period of the task. However, using active duplication for all tasks doubles the number of processors required in the nonfault-tolerant case. In contrast, a passive copy can be executed only if a failure prevents the corresponding primary copy from completing. A passive copy has the disadvantages of having tighter timing constraints—in the worst case it is not activated until the scheduled completion time of the primary copy—and of requiring some time overhead for synchronization with the corresponding primary copy. These drawbacks can be overcome by choosing active replication when the scheduled completion time of the primary copy is close to the deadline, that is to the end of the period, and by having smaller execution times for the backup copies. Moreover, since the time overhead for synchronization is usually very small, it can be included in the execution time of the primary task. Most importantly, passive duplication has the great advantage of overbooking the processors—many passive copies whose primary copies are assigned to different processors can be scheduled on the same processor so as to share the same time interval. Indeed, under the assumption of a single processor failure, only one of such passive copies will be actually executed, namely, the passive copy whose primary copy was prevented from completing because of the failure. Moreover, if only one failure is tolerated, the space allocated to active copies whose primary copy is not assigned to the failed processor can be reclaimed as soon as a failure is detected. Passive copy overbooking and active copy deallocation allow fewer processors to be used with respect to the case in which active duplication is used for all tasks.

The present paper considers the problem of preemptively scheduling a set of independent periodic tasks on a distributed system, such that each task deadline coincides with the next request of the same task, and all tasks start in-phase. In particular, this paper extends the RMFF algorithm to tolerate failures under the assumption that processors fail in a fail-stop manner. The algorithm determines by itself which tasks must use active duplication and which can use passive duplication, preferring passive duplication whenever possible. The rest of the paper is organized as follows. Section 2 gives a formal definition of the scheduling problem and a precise specification of the fault tolerance model. Moreover, the classical RM, CTT, and RMFF algorithms are recalled. Section 3 provides a high-level description of the proposed Fault-Tolerant Rate-Monotonic First-Fit (FTRMFF) algorithm. The algorithm analysis is done in Section 4. In particular, the ability of RM to meet the deadlines in the presence of one processor failure is characterized in Section 4.1, and CTT is extended in Section 4.2 so as to check the schedulability on a single processor of a task set including backup copies. Then, such an extended CTT is used in Section 4.3 to assign task copies to processors following a First-Fit heuristic which employs passive copy overbooking and active copy space reclaiming. An algorithm to recover from a single processor failure is shown in Section 4.4, while extensions to tolerate both many processor failures and software failures are presented in Sections 4.5 and 4.6, respectively. In Section 5, simulation experiments show that the proposed FTRMFF algorithm requires fewer processors than the active duplication approach. Finally, Section 6 summarizes the work and discusses further possible extensions.

2 Background

This section gives a formal definition of the scheduling problem and a precise specification of the fault tolerance model. Moreover, important properties of the well-known RM, CTT, and RMFF algorithms are recalled.

2.1 The Scheduling Problem

A *periodic task* τ_i is completely identified by a pair (C_i, T_i), where C_i is τ_i's *execution time* and T_i is τ_i's *request period*. The requests for τ_i are periodic, with constant interval T_i between every two consecutive requests, and τ_i's first request occurs at time 0. The worst case execution time for all the (infinite) requests of τ_i is constant and equal to C_i, with $C_i \leq T_i$. Periodic tasks τ_1, ..., τ_n are *independent*, that is the requests of any task do not depend on the execution of the other tasks. The *load* of a periodic task $\tau_i = (C_i, T_i)$ is $U_i \leq C_i/T_i$, while the *load of the task set* $\tau_1, \ldots, \tau_n$ is $U = \sum_{1 \leq i \leq n} U_i$.

Given n independent periodic tasks $\tau_1, \ldots, \tau_n$ and a set of *identical* processors, the *scheduling* problem consists of finding an order in which all the periodic requests of the tasks are to be executed on the processors so as to satisfy the following scheduling conditions:

(S1) *integrity* is preserved, that is, tasks and processors are sequential: each task is executed by at most one processor at a time and no processor executes more than one task at a time;

(S2) *deadlines* are met, namely, each request of any task must be completely executed before the next request of the same task, that is, by the end of its period;

(S3) the number m of processors is *minimized*.

2.2 The Fault-Tolerant Model

It is assumed that the processors belong to a distributed system and are connected by some kind of communication subsystem. The failure characteristics of the hardware are the following:

(F1) Processors fail in a fail-stop manner, that is a processor is either operational (i.e., nonfaulty) or ceases functioning;

(F2) All nonfaulty processors can communicate with each other;

(F3) Hardware provides fault isolation in the sense that a faulty processor cannot cause incorrect behavior in a nonfaulty processor; in other words, processors are independent as regard to failures;

(F4) The failure of a processor P_f is detected by the remaining nonfaulty processors after the failure, but within the instant corresponding to the closest task completion time of a task scheduled on P_f.

Note that assumption (F4) can be easily satisfied by a specific failure detection protocol as explained below, since by assumption (F1) all the processors are assumed to be fail-stop.

The *fault-tolerant scheduling* problem consists of finding a schedule for the tasks so as to satisfy the following additional condition:

(S4) *fault tolerance* is guaranteed, namely, conditions (S1)-(S3) are verified even in the presence of failures.

In order to achieve fault tolerance, two copies for each task are used, called *primary* and *backup* copies. The primary copy τ_i has its request period equal to T_i and its execution time equal to C_i, while the backup copy β_i has the same request period T_i but an execution time $D_i \neq C_i$. Although the fault-tolerant algorithm to be proposed works also when D_i is greater than or equal to C_i, in practice D_i is smaller than C_i, since backup copies usually provide a reduced functionality in a smaller execution time than the primary copies.

The primary copy of a task is always executed, while its backup copy β_i is executed according to β_i's *status*, which can be *active* or *passive*. If the status is active, then β_i is always executed, while if it is passive, then β_i is executed only when the primary copy fails. In other words, although both active and passive copies of the primary tasks are statically assigned to processors, passive backup copies are actually executed only when a failure of the corresponding primary copy occurs.

Each passive copy β_i is informed of the completion of τ_i at every occurrence of the periodic task by means of a message that the processor running τ_i sends in each period $[hT_i, (h+1)T_i]$ by τ_i's completion time to the processor assigned to the passive copy β_i. This message is small: since it must contain the indices of the primary task and of the sender and receiver processors, its size is $O(log\ n)$ bits. If the message is not received by a certain due time (to be specified in Section 3), a failure on the processor running τ_i is assumed and the passive copy β_i is scheduled. The overhead needed for such processor failure detections is mainly given by the short-message latency of the communication subsystem employed. In particular, with the current off-the-shelf technology, this overhead can be estimated in the order of few microseconds and is assumed to be included in the execution time of the primary copies. As for active copies, no implicit or explicit synchronization is assumed with their primary copies, since an active copy can run before, after, or concurrently with its primary copy.

2.3 The Rate-Monotonic Algorithm

Liu and Layland [10] proposed a fixed-priority scheduling algorithm, called *Rate-Monotonic* (RM), for solving the (nonfault-tolerant) problem stated in Section 2.1 on a single processor system, that is when $m = 1$. In their algorithm, each task is assigned a priority according to its *request rate* (the inverse of its request period)—tasks with short periods are assigned high priorities. At any instant of time, a pending task with the highest priority is scheduled. A currently running task with lower priority is preempted whenever a request of higher priority occurs, and the interrupted task is resumed later.

As an example, consider tasks τ_1 and τ_2 to be scheduled on one processor, and let $(C_1, T_1) = (1, 3)$ and $(C_2, T_2) = (3, 5)$. Task τ_1 has higher priority than τ_2, and the first request of τ_1 is scheduled during the time interval $[0, 1]$. Then the first request of τ_2 is scheduled during $[1, 3]$. At time 3, the second request of τ_1 comes, τ_2 is preempted, and τ_1 is scheduled during $[3, 4]$. Then τ_2 is resumed and scheduled during $[4, 5]$, and so on.

Liu and Layland proved the following two important results concerning fixed-priority scheduling algorithms.

Theorem 1. *The largest response time for any periodic request of τ_1 occurs whenever τ_i is requested simultaneously with the requests for all higher priority tasks.*

Theorem 2. *A periodic task set can be scheduled by a fixed-priority algorithm provided that the deadline of the first request of each task starting from a critical instant (i.e., an instant in which all tasks are simultaneously requested) is met.*

For example, a critical instant occurs when all tasks are in phase at time zero, which is called *critical instant phasing,* because it is the phasing that results in the longest response time for the first request of each task. As a consequence, to check the schedulability of any task τ_i, it is sufficient to check whether τ_i is schedulable in its first period $[0, T_i]$ when it is scheduled with all higher priority tasks.

2.4 The Completion Time Test

From Theorems 1 and 2, the following necessary and sufficient schedulability criterion was derived by Joseph and Pandya [5], as discussed also in [8].

Theorem 3. *Let the periodic tasks $\tau_1, \ldots, \tau_n$ be given in priority order and scheduled by a fixed-priority algorithm. All the periodic requests of τ_i will meet the deadlines under all task phasings if and only if:*

$$min_{0<t\leq T_i}\left\{\sum_{1\leq k\leq i} C_k\lceil t/T_k\rceil/t\right\}\leq 1.$$

The entire set of tasks $\tau_1, \ldots, \tau_n$ is schedulable under all task phasings if and only if:

$$max_{1 \le i \le n}\, min_{0 < t \le T_i} \left\{ \sum_{1 \le k \le i} C_k \lceil t/T_k \rceil / t \right\} \le 1$$

The minimization required in Theorem 3 is easy to compute in the case of the Rate-Monotonic algorithm. In fact, t needs to be checked only a finite number of times, as explained below.

Let $\tau = \{\tau_1, \ldots, \tau_i\}$, with $T_1 \le \ldots \le T_i$, be a set of tasks in phase at time zero, the *cumulative work* on a processor required by tasks in τ during $[0, t]$ is:

$$W(t, \tau) = \sum_{\tau_k \in \tau} C_k \lceil t/T_k \rceil.$$

Create the sequence of times $S_0, S_1, \ldots$ with $S_0 = \sum_{\tau_k \in \tau} C_k$, and with $S_{l+1} = W(S_l, \tau)$. If for some l, $S_l = S_{l+1} \le T_i$, then τ_i is schedulable. Otherwise, if $T_i \le S_l$ for some l, task τ_i is not schedulable. Note that S_l is exactly equal to the minimum t, $0 < t < T_i$, for which $\sum_{1 \le k \le i} C_k \lceil t/T_k \rceil \le t$ as required in Theorem 3. This schedulability test is called *Completion Time Test (CTT).*

As an immediate consequence of the above theorems, the following property holds:

Property 1. Let the Completion Time Test be satisfied for $\tau_1, \ldots, \tau_i$, and let $S_l = S_{l+1} \le T_i$ for some l. Then in any period $[hT_i, (h+1)T_i]$, with h integer, τ_i will complete no later than the instant $hT_i + S_l$.

For the sake of clarity, the quantity S_l will be denoted in the following by ψ_i since such a quantity represents the *worst-case completion time* of task τ_i in any request period T_i.

As an example of use of CTT, consider again tasks τ_1 and τ_2 with $(C_1, T_1) = (1, 3)$ and $(C_2, T_2) = (3, 5)$ and let us check the schedulability of τ_2:

$$S_0 = 1 + 3 = 4,$$

$$S_1 = W(4, \{\tau_1, \tau_2\}) = 1\lceil 4/3 \rceil + 3\lceil 4/5 \rceil = 5,$$

and

$$S_2 = W(5, \{\tau_1, \tau_2\}) = 1\lceil 5/3 \rceil + 3\lceil 5/5 \rceil = 5.$$

Since $S_1 = S_2 \le T_2 = 5$, all the periodic requests of τ_2 will meet their deadlines.

It is worth noting that, by Theorem 3, the schedulability of lower priority tasks does not guarantee the schedulability of higher priority tasks. Therefore, in order to check the schedulability of a set of tasks, each task must get through the CTT when it is scheduled with all higher priority tasks. If tasks are picked by priority order, the schedulability test can proceed in an incremental way: CTT is performed considering tasks $\tau_1, \ldots \tau_1$ on the period $[0, T_i]$, for $i = 1, \ldots, n$, that is, by adding one task τ_i at a time to the preceding tasks $\tau_1, \ldots, \tau_{i-1}$, *without* the need to test again the schedulability of $\tau_1, \ldots, \tau_{i-1}$. In this way, as soon as ψ_i is computed, ψ_i will not change anymore, since only lower priority tasks will be considered later.

2.5 The Rate-Monotonic First-Fit

Dhall and Liu [3] generalized the RM algorithm to accommodate multiprocessor systems. In particular, they proposed the so called *Rate-Monotonic First-Fit* (RMFF) algorithm. It is a partitioning algorithm, where tasks are first assigned to processors following the RM priority order and then all the tasks assigned to the same processor are scheduled with the RM algorithm. Let $T_1 \le T_2 \le \ldots \le T_n$, the algorithm acts as follows. For $i = 1, 2, \ldots, n$, the generic task τ_i is assigned to the first processor P_j such that τ_i and the other tasks already assigned to P_j can be scheduled on P_j according to RM. If no such processor exists, the task is assigned to a new processor. Dhall and Liu showed that, using a schedulability condition weaker than CTT, RMFF uses about 2.33*U* processors in the worst case, where *U* is the load of the task set. The 2.33 worst case bound was recently lowered to 1.75 by Burchard, Liebeherr, Oh, and Son [1], using a schedulability condition stronger than that used in [3], but without using the RM priority order for task assignment, and partially using CTT. In practice, however, RMFF remains competitive, for its simplicity and efficiency. It employs the *same* priority order both for assigning tasks to processors and scheduling tasks on each processor, and requires on the average a number of processors very close to *U* when CTT is used to check for schedulability on each processor, as confirmed also by the simulation experiments exhibited in Section 5. Moreover, as shown in Section 4, it can be extended in a clean way to tolerate hardware and software failures.

3 Overview of the Fault-Tolerant RMFF Algorithm

This section provides an informal high-level description of the proposed Fault-Tolerant Rate-Monotonic First-Fit (FTRMFF) algorithm. The algorithm analysis is done in next section. For the sake of simplicity, only the extension to tolerate one processor failure is discussed hereafter. Extensions to support many processor failures or software failures will be discussed in Sections 4.5 and 4.6, respectively.

In the FTRMFF algorithm, primary and backup copies of different tasks can be assigned to the same processor. Of course, in order to tolerate a processor failure, the primary copy and the backup copy of the same task should not be assigned to the same processor. The algorithm proposed can be viewed as the RMFF algorithm applied to a task set including both primary and backup copies. Task copies, both primary and backup, are ordered by increasing periods, namely, the priority of a copy is equal to the inverse of its period. A tie between a primary copy τ_i and its backup copy β_i is broken by giving higher priority to τ_i. Thus tasks are indexed by decreasing RM priorities, and are assigned to the processors following the order:

$$\tau_1,\ \beta_1,\ \tau_2,\ \beta_2,\ \ldots,\ \tau_n,\ \beta_n.$$

CTT is used to check whether a task copy can be assigned to a processor. Thanks to Property 1 of Section 2, CTT also provides enough information to decide whether a backup copy should be active or passive. Indeed, while checking for schedulability of a primary copy τ_i, CTT also computes its worst-case completion time ψ_i. If the schedulability test for τ_i succeeds, that is when $\psi_i \le T_i$, then for each request period there are at least $T_i - \psi_i$ time units to schedule β_i as a passive copy on another processor. Let $B_i = T_i - \psi_i$ be the *recovery time* of the backup copy β_i. If $B_i \geq D_i$, then β_i may be scheduled as a passive copy, since there is enough time to execute β_i after τ_i if a processor failure prevents τ_i from being completed; otherwise β_i must be scheduled as an active copy. The algorithm prefers to schedule a backup copy as a passive copy whenever possible, so as to overbook each processor with more passive copies whose primary copies are assigned to different processors.

It is worth noting that, although tasks could be assigned to processors following any order, considering task copies by decreasing RM priorities greatly simplifies the algorithm. Indeed, such an ordering is the *same* ordering used by the RM algorithm to schedule the tasks assigned to each processor. Therefore, when a task τ_i is assigned to a processor, only lower priority tasks will be assigned later to the same processor, and the time intervals for τ_i's execution on the processor will remain unchanged. In particular, also worst case completion time ψ_i will remain unchanged. This allows to determine whether the backup copy β_i of τ_i can be scheduled as a passive copy. Clearly, with another ordering a higher priority task can be assigned to the same processor after τ_i. In this case, ψ_i needs to be recomputed and β_i must be reassigned and rescheduled. This justifies the $\tau_1, \beta_1, \tau_2, \beta_2, \ldots \tau_n, \beta_n$ order of assignment. Moreover, since the algorithm generalizes RMFF, it assigns a backup copy β_i, either passive or active, to the *first* processor P_j such that τ_i is not assigned to P_j, and β_i and the other primary and backup copies already assigned to P_j can be scheduled on P_j according to the RM algorithm for a single processor.

To find a processor a task copy can be assigned to, however, several applications of CTT are required, which take into account the situations in which no processor fails or any processor fails. The applications of the test depend on the kind (primary/backup) of the task copy to be assigned as well as on its status (active/passive) if the copy is a backup copy. There are three main assignment cases.

(A1) To assign a primary copy τ_i to a processor P_j, two conditions have to be checked.

- τ_i must be schedulable together with all the primary and active backup copies already assigned to P_j;
- τ_i must be schedulable together with all the primary copies already assigned to P_j and all the active and backup copies assigned to P_j such that their corresponding primary copies are all assigned to the same processor P_f, and this condition must be considered for *all* $P_f \neq P_j$.

The first condition takes into account the situation in which no failure occurs, while the second one takes into account the situation in which any processor other than P_j fails. Thus, as many applications of CTT as the total number of processors are required in the worst case to determine whether τ_i can be assigned to P_j. Note that the second condition use the space reserved on P_j to active copies whose primary copies are not assigned to P_f, since only one processor failure is assumed to be tolerated.

(A2) To assign an active backup copy β_i to a processor P_j, assume that the primary copy τ_i is already assigned to processor $P_p \neq P_j$ two conditions have also to be checked.

- β_i must be schedulable together with all the primary and active backup copies already assigned to P_j;
- β_i must be schedulable together with all the primary copies already assigned to P_j and all the active and backup copies assigned to P_j such that their corresponding primary copies are all assigned to P_p.

These conditions are analogous to those of (A1), with the difference that the second one takes into account the situation where the failed processor is that running the primary copy τ_i. Thus only two applications of CTT are required to determine whether β_i can be assigned to P_j.

(A3) Finally, to assign a passive backup copy β_i to a processor P_j, assuming again that the primary copy τ_i is already assigned to processor $P_p \neq P_j$, only one condition has to be tested, which is identical to the second condition of (A2). Thus only one application of CTT is needed to determine whether β_i can be assigned to P_j.

As soon as task copies are assigned to processors, all the copies assigned to the same processor are scheduled with the RM algorithm. However, in the absence of failures, each processor executes its primary copies and active backup copies only. When the processor assigned to β_i does not receive the synchronization message of τ_i by time $hT_i + \psi_i$, a failure of the processor running τ_i is assumed and the passive copy β_i is executed. To understand how to recover from a failure, assume τ_i is assigned to processor P_f which is detected at time θ to be failed, with θ belonging to $[hT_i, (h+1)T_i]$ for any h. If β_i is an active copy scheduled on a processor P_j, then β_i will continue to be executed and no further action is needed for β_i. If β_i is passive, then β_i becomes active on P_j starting either from θ, if the execution of τ_i was not completed by P_f before θ, or from $(h+1)T_i$, if the execution of τ_i was already completed before θ. In other words, if $\theta > hT_i + \psi_i$, then τ_i was completed before P_f's failure and there is no need to schedule β_i by time $(h+1)T_i$. If $\theta \leq hT_i + \psi_i$ then, in order to recover the lost computation of τ_i, β_i must be executed for the first time during the interval $[\theta, (h+1)T_i]$, which in general is shorter than T_i. It will be shown in the next section that β_i, the primary copies of P_j, and the backup copies of P_j meet their deadlines even in this case.

4 Analysis of the Fault-Tolerant RMFF Algorithm

In this section, necessary and sufficient schedulability criteria are proved which extend Theorem 3 to schedule a set of primary and backup copies to recover from one processor failure. Based on the proposed criteria, a fault-tolerant extension of RMFF is derived and proved to be correct.

4.1 Schedulability Criteria

In order to extend Theorem 3, consider a generic task set σ containing both primary and backup copies which must be scheduled *all together* on a single processor. The *cumulative work* $W(t,\sigma)$ during a time interval $[0,t]$ can be defined as follows:

$$W(t,\sigma)=\sum_{\tau_k\in\sigma}C_k\lceil t/T_k\rceil+\sum_{\beta_k\in\sigma}D_k\,\phi(k,t)$$

where

$$\phi(k,\ t)\ =\ \begin{cases}\lceil t/T_k\rceil & \text{if } \beta_k \text{ is active}\\ 1 & \text{if } \beta_k \text{ is passive and } t\le B_k\\ 1+\lceil (t-B_k)/T_k\rceil & \text{if } \beta_k \text{ is passive and } t> B_k\end{cases}$$

The function $\phi(k,\ t)$ gives the overall number of requests of a backup copy β_k during $[0,t]$. If β_k is an active copy, then its work is the same as a primary copy. If β_k is a passive copy, then the *first request* of β_k must be executed within β_k's recovery time, which is shorter than the task period T_k and is at least $B_k=T_k-\psi_k$ time units long, while the next requests of β_k must be executed at every period T_k. Thus, if $t\le B_k$, there is only one request of β_k to be executed within $[0,t]$ and $\phi(k,t)=1$; otherwise there is one request to be executed in $[0,B_k]$, and $\lceil (t-B_k)/T_k\rceil$ requests to be executed in $[B_k,t]$, hence $\phi(k,t)=1+\lceil (t-B_k)/T_k\rceil$.

It is worth noting that, according to Theorems 1 and 2, the previously defined $W(t,\sigma)$ takes into account the *worst* possible cumulative work, in which *all* the passive backup copies start *in phase* together with all the other primary and active backup copies, and must be executed for the first time within their recovery times, which are *shorter* than their periods. Such a definition of the work $W(t,\sigma)$ is the basis for providing fault tolerance, as shown in the following.

In order to present the exact analysis of the schedulability of a set of tasks using the RM algorithm to tolerate a processor failure, some additional definitions are needed. Let us assume that all the task copies in $\{\tau_1,\tau_2,\ldots,\tau_n\}\cup\{\beta_1,\beta_2,\ldots,\beta_n\}$ are already assigned to the processors and that the worst case completion time ψ_i of τ_i and the status (active/passive) of β_i are already determined for all i. In the following, *status*(β_i) denotes the status (active/passive) of β_i, $P(\tau_i)$ the processor assigned to the primary copy τ_i, and $P(\beta_i)$ the processor assigned to the backup copy β_i. The tasks assigned to each processor P_j are denoted as follows.

The sets *primary*(P_j) and *backup*(P_j) represent the primary and backup copies assigned to processor P_j, namely:

$$primary(P_j)=\{\tau_h\colon P(\tau_h)=P_j\},$$
$$backup(P_j)=\{\beta_h\colon P(\beta_h)=P_j\}.$$

The set *active*(P_j) includes the active backup copies assigned to processor P_j, namely,

$$active(P_j)=\{\beta_h\colon \beta_h\colon\in backup(P_j),\ status(\beta_h)=active\}$$

The set *passiveRecover*(P_j,P_f) consists of the passive copies assigned to P_j such that their primary copies are assigned to P_f. In other words, this set contains all the passive backup copies that processor P_j *must start* scheduling when a failure of processor P_f is detected. That is,

$$passiveRecover(P_j,P_f)$$
$$=\{\beta_h\colon \beta_h\in backup(P_j), P(\tau_h)=P_f,\ status(\beta_h)=passive\}.$$

In contrast, the set *activeRecover*(P_j,P_f) denotes the active copies assigned to P_j with primary copies assigned to P_f, namely, this set contains all the active backup copies that processor P_j *must keep* scheduling when P_f fails:

$$activeRecover(P_j,P_f)$$
$$=\{\beta_h\colon \beta_h\in backup(P_j),\ P(\tau_h)=P_f,\ status(\beta_h)=active\}.$$

Finally, *recover*(P_j,P_f) gives the union between the last two sets:

$$recover(P_j,P_f)$$
$$=passiveRecover(P_j,\ P_f)\cup activeRecover(P_j,\ P_f).$$

In the absence of failures, any processor P_j must execute its primary copies together with its active backup copies. In other words, the set of tasks scheduled on P_j in the fault-free case is given by $primary(P_j)\cup active(P_j)$.

Theorem 4. *Let $\sigma=primary(P_j)\cup active(P_j)$ be the set of periodic tasks given in priority order which are assigned to processor P_j. All the periodic requests of tasks in σ will meet the deadlines if and only if:*

$$\max_{\tau_k,\beta_k\in\sigma}\ \min_{0<t\le T_k}\left\{\sum_{\tau_k\in\sigma}C_k\lceil t/T_k\rceil/t+\sum_{\beta_k\in\sigma}D_k\phi(k,t)/t\right\}\le 1.$$

The proof of Theorem 4 follows from Theorem 3, since active backup copies can be regarded as additional primary copies and $\phi(k,t)=\lceil t/T_k\rceil$ by definition.

Consider now the case that a failure of processor P_f is detected at time θ. In such a case, only the backup copies in $recover(P_j,P_f)$ should be scheduled on P_j together with its primary copies. Note that under the assumption of a single processor failure, all the active copies in $active(P_j)-activeRecover(P_j,P_f)$ do not need to be executed after time θ.

Theorem 5. *Consider any processorP_j and assume that the failure of processor $P_f, j\ne f,$ is detected at time θ. Let $\sigma=primary(P_j)\cup recover(P_j,P_f)$ be the set of periodic tasks given in priority order and assigned to P_j. All the periodic requests of tasks in σ will meet the deadlines for any θ if and only if:*

$$\max_{\tau_k,\beta_k\in\sigma}\ \min_{0<t\le V_k}\left\{\sum_{\tau_k\in\sigma}C_k\lceil t/T_k\rceil/t+\sum_{\beta_k\in\sigma}D_k\phi(k,t)/t\right\}\le 1.$$

where V_k is equal to T_k for a primary copy or an active backup copy and to B_k for a passive backup copy.

Proof. A critical instant for a passive copy β_k occurs when β_k and all primary and backup copies with higher priority than β_k are released simultaneously at time θ on processor P_j. Note that the first request period of β_k may be shorter than T_k and in the worst case is equal to B_k. Since by Theorem 1 the worst case phasing occurs when all tasks are in phase at time 0, we can restrict our attention to the case in which all the tasks in σ are released at time 0 and each passive copy β_k has its period equal to B_k. Therefore, if β_k can be scheduled within

$[0, B_k]$, then can also be scheduled in any interval $[hT_k + \psi_k,\ hT_k + \psi_k + B_k] = [hT_k + \psi_k, (h+1)T_k]$.

Any copy γ_k (with γ_k equal to τ_k or β_k), completes its execution at time $t \in [0, V_k]$ if and only if all the requests for copies with higher priority than γ_k and the request for γ_k itself are completed at time t. To evaluate the cumulative work for processing γ_k, any backup copy β_i with higher priority than γ_k can be considered as a primary copy having execution time D_i and period B_i, if β_i is passive, or T_i, if β_i is active. The number of requests of β_i in $[0, t]$ is given by $\phi(i, t)$. It follows that a necessary and sufficient condition for γ_k to meet its deadline is that the cumulative work must be less than or equal to t. Hence, the proof follows from Theorem 3. □

4.2 Fault-Tolerant CTT

Based on Theorems 4 and 5, two kinds of schedulability tests are needed, one to check for schedulability in the absence of failures, and the other to check for schedulability after a processor failure.

Given a task copy γ_i (either the primary copy τ_i or the backup copy β_i) to be assigned to a processor P_j, the first schedulability test procedure, that we call *NoFaultCTT*, tests for schedulability of γ_i on P_j together with the task copies *already* assigned to P_j in the absence of failures:

NoFaultCTT(γ_i, P_j)

1. Check whether the task set $\sigma = \gamma_i \cup primary(P_j) \cup active(P_j)$ is schedulable on P_j by means of the formula given in Theorem 4;
2. If σ is schedulable and $\gamma_i = \tau_i$, then return the worst case completion time ψ_i of τ_i.

Given γ_i to be assigned to P_j, and a processor $P_f \neq P_j$, the second procedure, called *OneFaultCTT*, tests for schedulability of γ_i on P_j together with the task copies *already* assigned to P_j in the case that processor P_f failed:

OneFaultCTT(γ_i, P_j, P_f)

1. Check whether $\sigma = \gamma_i \cup primary(P_j) \cup recover(P_j, P_f)$ is schedulable on P_j by means of the formula given in Theorem 5.

To assign a primary copy τ_i to P_j, both *NoFaultCTT*(τ_i, P_j) and *OneFaultCTT*(τ_i, P_j, P_f) must be satisfied *for all* $P_f \neq P_j$, since the recovery from the failure of any processor other than P_j must be taken into account. To assign an active backup copy β_i to P_j, instead, both *NoFaultCTT*(β_i, P_j) and *OneFaultCTT*(β_i, P_j, P_f) must be satisfied for $P_f = P(\tau_i)$ *only*, since the failure of the processor running the corresponding primary copy has to be tolerated. Finally, to assign a passive backup copy β_i to P_j, only *OneFaultCTT*$(\beta_i, P_j, P(\tau_i))$ has to be satisfied.

4.3 Fault-Tolerant RMFF

Using the *NoFaultCTT* and *OneFaultCTT* procedures previously introduced, all the task copies are assigned to the first processor in which they fit, following the ordering:

$$\tau_1,\ \beta_1,\ \tau_2,\ \beta_2,\ \ldots,\ \tau_n,\ \beta_n.$$

The assignment procedure consists of a main loop repeated for $i = 1, 2, \ldots n$, containing four consecutive steps. The first step assigns the primary copy τ_i to the first processor in which it fits. The second step establishes the recovery time B_i and the status of the backup copy β_i. Depending on β_i's status, the third or fourth step assigns the backup copy β_i to the first processor P_j in which β_i fits such that the primary copy τ_i is not assigned to the same processor. The *FTRMFF-Assignment* procedure is summarized as follows.

FTRMFF-Assignment

```
(0) Let the task copies be given by RM priority ordering: $\tau_1, \beta_1, \tau_2, \beta_2, \ldots, \tau_n, \beta_n$; Set the number $m$ of processors used to 1;
(1) Repeat the following steps for $i = 1, 2, \ldots, n$:
    (1.1) Assign the primary copy $\tau_i$ to the *first* processor $P_j$ for which $NoFaultCTT(\tau_i, P_j)$ is satisfied and $OneFaultCTT(\tau_i P_j, P_f)$ is satisfied *for all* $1 \leq f \neq j \leq m$ setting $P(\tau_i) = P_j$; if no such a processor exists, set $m$ to $m+1$ and assign $\tau_i$ to $P_m$, setting $P(\tau_i) = P_m$;
    (1.2) Let $\psi_i$ be the worst case completion time of $\tau_i$ computed in step (1.1); if $T_i - \psi_i < D_i$ then set $status(\beta_i)$ to active, else set $status(\beta_i)$ to passive;
    (1.3) If $status(\beta_i) =$ active then assign $\beta_i$ to the *first* processor $P_j \neq P(\tau_i)$ for which $NoFaultCTT(\beta_i, P_j)$ and $OneFaultCTT(\beta_i, P_j, P(\tau_i))$ are satisfied, setting $P(\beta_i) = P_j$; If no such processor exists set $m$ to $m+1$ and assign $\beta_i$ to $P_m$, setting $P(\beta_i) = P_m$;
    (1.4) If $status(\beta_i) =$ passive then assign $\beta_i$ to the *first* processor $P_j \neq P(\tau_i)$ for which $OneFaultCTT(\beta_i, P_j, P(\tau_i))$ is satisfied, setting $P(\beta_i) = P_j$; if no such processor exists set $m$ to $m+1$ and assign $\beta_i$ to $P_m$, setting $P(\beta_i) = P_m$;
(2) Return the number $m$ of processors used and the task assignment found.
```

Example 1. As an example of execution of the *FTRMFF-Assignment* algorithm, consider four primary tasks $\tau_1, ..., \tau_4$ with execution times $C_1 = 2$, $C_2 = 1$, $C_3 = 3$, $C_4 = 3$, and periods $T_1 = 5$, $T_2 = 6$, $T_3 = 8$, $T_4 = 9$, and four backup copies $\beta_1, ..., \beta_4$ such that each β_i has the same period and execution time as the primary copy τ_i, namely $D_i = C_i$. The algorithm sets the status of β_1, β_2, and β_3 to passive, while only β_4's status is set to active. The final assignment of the primary and backup copies is the following: *primary*$(P_1) = \{\tau_1, \tau_2, \tau_4\}$, *backup*$(P_1) = \emptyset$, *primary*$(P_2) = \emptyset$, *backup*$(P_2) = \{\beta_1, \beta_2, \beta_3\}$, *primary*$(P_3) = \{\tau_3\}$, and *backup*$(P_3) = \{\beta_4\}$. It is worth noting that P_2 is overbooked with many passive copies, but will never be required to execute the backup copies of both processor P_1 and processor P_3 at the same time, since at most one processor failure is assumed to occur. Therefore, P_2 gets allocated the backup copies $\{\beta_1, \beta_2, \beta_3\}$, although P_2 is not able to execute all of them together. In other words, if processor P_1 fails, then P_2 starts the execution of passive copies β_1 and β_2, while if P_3 fails, then P_2 starts the execution of β_3. Note that at least two processors are needed by any algorithm for scheduling the primary copies only, since the load U of $\tau_1, \ldots, \tau_4$ is $2/5 + 1/6 + 3/8 + 3/9 \approx 1.26$. Thus, duplicating on two sets of processors the schedule for the non-fault-tolerant case requires at least four processors to tolerate one failure. The proposed FTRMFF algorithm, instead, tolerates one failure using three processors only.

The procedure *FTRMFF-Assignment* is executed off-line and requires $O(nm^2)$ schedulability tests to be performed. Indeed, to assign a primary copy, at most m processors are tried. Each trial requires in turn one execution of *NoFaultCTT* and $m-1$ executions of *OneFaultCTT*, in the worst case. In contrast, in the non-fault-tolerant RMFF, each trial requires one execution of CTT, for a total of $O(nm)$ schedulability tests. It is worth noting that, although CTT has a pseudopolynomial complexity in the worst case, it is not too slow in practice. Moreover, since task copies are picked by RM priority order, consecutive schedulability tests on the same processor can be performed in an incremental way, as discussed in Section 2.4, thus reducing their computation time.

4.4 Recovery from a Processor Failure

Once an assignment is found by the FTRMFF algorithm, each processor P_j schedules, in the absence of a failure, the tasks in $primary(P_j) \cup active(P_j)$ according to the RM algorithm. When a failure of a processor, say P_f, is detected at time θ, the *FTRMFF-Recovery* procedure described below is invoked. The procedure takes as input P_f and θ, and produces as its output a schedule according to the RM algorithm on $m-1$ processors of at least one copy of each task, such that no task deadline is missed. The uncompleted tasks assigned to P_f are recovered by the remaining nonfaulty processors. Let β_i be a passive copy assigned to processor P_j such that $P_f = P(\tau_i)$ and θ belongs to $[hT_i, (h+1)T_i]$ for any h: β_i becomes active on P_j starting either from θ, if the execution of τ_i was not completed before θ (that is, $hT_i + \psi_i \geq \theta$), or from $(h+1)T_i$, if the execution of τ_i was already completed by P_f before θ (that is, $hT_i + \psi_i < \theta$). Note that, in the former case, β_i is executed on P_j for the first time during the interval $[\theta, (h+1)T_i]$ in order to recover the computation of τ_i lost during $[hT_i, \theta]$. Note also that, in any case, all the active backup copies of primary tasks scheduled on the nonfaulty processors are deallocated from P_j.

```
FTRMFF-Recovery($P_f, \theta$)
(1) Do the following steps in parallel for all the processors $P_j$ such that $1 \leq j \leq m$ and $j \neq f$;
    (1.1) If passiveRecover($P_j, P_f$) $\neq \emptyset$ then do the following:
        (1.1.1) Set to active the status of each $\beta_i \in$ passiveRecover($P_j, P_f$) either from time $\theta$, if $\psi_i \geq \theta \bmod T_i$, or from the next period of $\beta_i$, if $\psi_i < \theta \bmod T_i$;
        (1.1.2) Replace the schedule of $P_j$ with that produced by the RM algorithm for the set primary($P_j$) $\cup$ recover($P_j$, $P_f$);
    (1.2) If passiveRecover($P_j, P_f$) $= \emptyset$ then continue to schedule primary($P_j$) $\cup$ active($P_j$) on $P_j$ with the RM algorithm, as in the fault-free case.
```

As an example of recovery, consider the task set of Example 1. The schedule in the absence of failures is shown in Fig. 1. If a failure of processor P_1 occurs at time 0, it is detected by the other processors before the closest task completion time on P_1, namely by time $\theta = 2$. The schedule of Fig. 1 is recovered as shown in Fig. 2.

The procedure *FTRMFF-Recovery* is executed on-line and is very fast, since all the required sets, including *passiveRecover*(P_j, P_f) and *recover*(P_j, P_f), were previously computed off-line by the *FTRMFF-Assignment* procedure, which already made all the schedulability tests, too.

4.5 Tolerating Many Processor Failures

In order to tolerate many processor failures, spare processors must be employed to replace failed processors on-line. Assuming that a processor failed and that a second processor cannot fail before the first failure is recovered, the substitution of the failed processor P_f with the spare processor P_s can be done by the following *FTRMFF-Replacing* procedure. In practice, the passive backup copies of the uncompleted primary tasks of the failed processor P_f are executed by the remaining processors only once, while the spare processor P_s resumes the scheduling of the primary and active backup copies of P_f from their first requests following the failure detection time.

```
FTRMFF-Replacing($P_f, \theta$)
(1) Do the following steps in parallel for all $P_j$ such that $1 \leq j \leq m, j \neq f$, and for $j = s$ as well;
    (1.1) If $j \neq s$ and passiveRecover($P_j, P_f$) $\neq \emptyset$ then do the following:
        (1.1.1) Let unfinishedPassive($P_j$, $P_f$) $= \{\beta_i \in$ passiveRecover($P_j, P_f$): $\psi_i \geq \theta \bmod T_i\}$. Set to active the status of each $\beta_i \in$ unfinishedPassive($P_j, P_f$);
        (1.1.2) Replace the schedule of $P_j$ with that produced by the RM algorithm for the set primary($P_j$) $\cup$ unfinishedPassive($P_j, P_f$), but execute the copies in unfinishedPassive($P_j, P_f$) only once;
        (1.1.3) Restore the old schedule of $P_j$, that is that for primary($P_j$) $\cup$ active($P_j$), resuming each $\beta_i \in$ active($P_j$) starting from the first request following $\theta + T_i$;
    (1.2) If $j \neq s$ and passiveRecover($P_j, P_f$) $= \emptyset$ then continue to schedule primary($P_j$) $\cup$ active($P_j$) on $P_j$ with the RM algorithm, as in the fault-free case;
    (1.3) If $j = s$ then inherit the schedule of $P_f$, that is that for primary($P_f$) $\cup$ active($P_f$), resuming each task copy starting from the first request following $\theta$.
```

As an example, the substitution of a failed processor with a spare processor for the schedule of Fig. 2 is shown in Fig. 3. The correctness of the above procedure is proved by the following theorem.

Theorem 6. *If q spare processors are available, then the FTRMFF algorithm can tolerate $q+1$ failures, provided that a failure of processor P_f is detected within the closest completion time of the task set primary$(P_f) \cup$ active(P_f) and the time interval between two consecutive failures is three times the largest task request period.*

Proof. The scheme to tolerate more than one failure can be organized in three phases: detection, recovery and reconfiguration phases. The last two phases are performed by the *FTRMFF-Replacing* procedure.

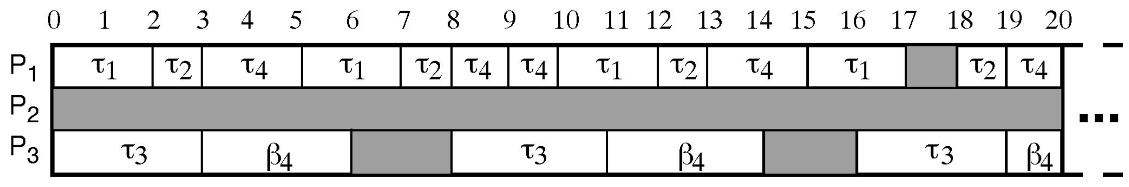

Fig. 1. The schedule in the absence of failures.

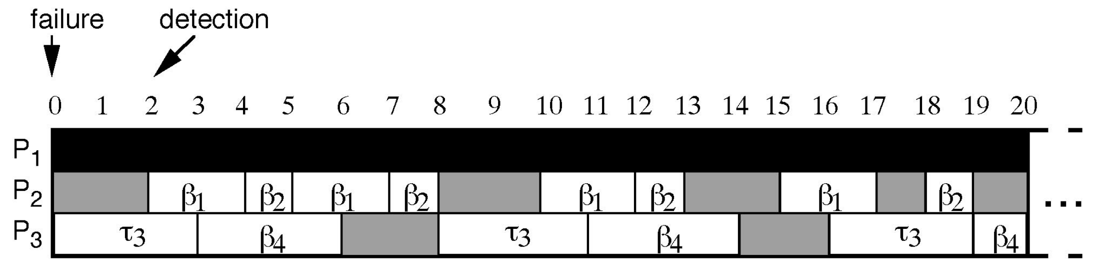

Fig. 2. Recovery from a failure of P_1 at time 0.

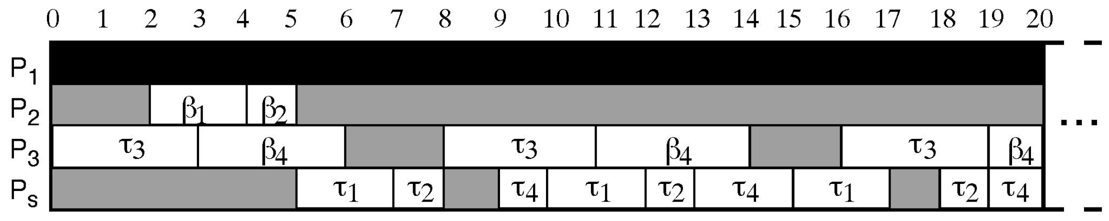

Fig. 3. Substitution of the failed processor with a spare processor.

In the detection phase, the worst case occurs when τ_n, which is the task with the longest request period, is the only one allocated to P_f and P_f fails immediately after the completion time of τ_n. This phase takes at most T_n time units, since the failure of P_f will be detected within the completion time of the next request of τ_n.

During the recovery phase, all the passive copies of the uncompleted tasks assigned to P_f are executed by the non-faulty processors only once (step 1.1.2 of *FTRMFF-Replacing*), and the spare processor P_s inherits P_f's schedule starting from the first task request following the failure detection time θ (step 1.3). In other words, let θ belong to $[(h-1)T_i, hT_i]$, by Theorem 5 in the worst case any passive copy $\beta_i \in \mathit{passiveRecover}(P_j, P_f)$ is executed within $[\theta, hT_i]$. Hence, at time hT_i, the execution of passive copy β_i can be suspended on P_j, since any $\tau_i \in \mathit{primary}(P_f)$ and any $\beta_i \in \mathit{active}(P_f)$ are inherited by P_s at that time. The recovery phase requires at most T_n time units, since hT_i - $\theta < T_n$ for all i.

Finally, in the reconfiguration phase, for every non-faulty processor P_j, each copy $\beta_i \in \mathit{active}(P_j)$ which was previously suspended, is resumed starting from the first request of β_i following $\theta + T_i$ (step 1.1.3). In the worst case, at time $\theta + T_n$ the schedule of the faulty processor P_f is completely inherited by the spare processor P_s; thus the nonfaulty processors are no more required to execute any passive copy. The reconfiguration phase is completed by time $\theta + 2T_n$.

Therefore, many processors failures can be tolerated, provided that the minimum time interval between two consecutive failures is $3T_n$. If there are q spare processors, q faulty processors can be replaced by means of the *FTRMFF-Replacing* procedure, while one additional failure can be tolerated by means of the *FTRMFF-Recovery* procedure. □

4.6 Tolerating Software Failures

In addition to processor failures, a hard-real-time system can fail also due to design faults in the software. The scheduling algorithm proposed allows the combination of the recovery block technique with distributed processing to achieve a uniform treatment of both software and hardware failures. To explain the ideas of the approach, assume that two different implementations of the same task specification are provided. The two implementations, which may have different execution times, represent the primary and the backup copies of a task. It is worth noting that the backup copy must have an independent design, so that if the primary fails owing to a software error, it is highly probable that the backup has not failed (see [6] for further details). Assume also that an *acceptance test*, which is an assertion on the state of the task, is executed at the end of each copy. Since the processors are assumed fail-stop, if the acceptance test fails, it signals the presence of an error in the software. The time to execute the acceptance test is assumed to be included in the primary copy execution time. One approach to implement the recovery from software failures is as follows. The primary copy τ_i is executed on $P(\tau_i)$, and the backup copy β_i is allocated on a processor $P(\beta_i) \neq P(\tau_i)$. The result of τ_i is checked on $P(\tau_i)$ using the acceptance test. If a software error is detected in τ_i, $P(\tau_i)$ sends a notice to processor $P(\beta_i)$ to activate the backup copy β_i, only. Note that if the schedulability test was successful on $P(\beta_i)$, then β_i will meet its deadline: In case of software failure of τ_i only the time to recover τ_i is needed, while the FTRMFF algorithm returns a feasible schedule which can recover all the tasks allocated to $P(\tau_i)$. If the primary copy τ_i produces results which satisfy the acceptance test, it forwards them to the destination, and the backup copy β_i is not executed.

5 Simulation Experiments

In order to evaluate the number of processors used by the FTRMFF algorithm for scheduling both primary and backup copies, simulation experiments are performed. The execution time C_i of each primary copy τ_i is assumed to be equal to the execution time D_i of the backup copy β_i. Indeed, when $C_i > D_i$, the algorithm performs better, since fewer processors are required for the execution of the backup copies; while the case $C_i < D_i$ does not seem of practical interest, since usually the backup copies provide a reduced functionality in a shorter time than the primary copies.

Large task sets with $100 \leq n \leq 1,000$ tasks are generated. In the experiments, we vary a parameter $\alpha = max\{U_1, \cdots, U_n\}$, which represents the maximum load occurring in the task set. Each task period T_i is selected to be uniformly distributed in the interval $1 \leq T_i \leq 500$, $1 \leq i \leq n$. Each execution time C_i is also taken from a uniform distribution, but in the interval $0 \leq C_i \leq \alpha T_i$, $1 \leq i \leq n$. Three values are chosen for α, namely, 0.2, 0.5, and 0.8. Higher values for α, and hence for the execution times, are not considered, since as the average execution time tends to be half the average period, there is no time to execute a passive copy after its corresponding primary copy in case of a processor failure; therefore, active duplication is chosen for almost all tasks, and the number of processors of the resulting schedule is approximately that obtained by duplicating the schedule for the primary copies. For the chosen n and α, the experiment is repeated 30 times, and the average result is computed.

The performance metric in all the experiments is the number of processors required to assign a given task set. We compare the RMFF algorithm, which uses the CTT for schedulability testing, and the *FTRMFF-Assignment* procedure proposed in Section 4. In the outcome of the experiments, we denote with N the number of processors required by the FTRMFF algorithm for a task set consisting of both primary and backup task copies, and with M the number of processors required by the RMFF algorithm for a task set with identical primary copies and no backup copies. Since an optimal task assignment is very hard to be calculated for large task sets, we use the total load $U = U_1 + \ldots + U_n$ as a lower bound on the minimum number of processors needed to assign the primary copies. The RMFF and FTRMFF assignment algorithms were written in C++ and run on a Digital AlphaServer 2100.

The outcome of the experiments is given in Fig. 4, which shows that both N and M increase in a way that is proportional to U. For smaller values of α, M/U is very close to 1 and N/U is far from being the double of M/U, as would be if the schedule of RMFF were duplicated on two sets of processors. Thus RMFF uses a number of processors close to optimality, and FTRMFF gains benefits from passive duplication, requiring a small number of additional processors to provide fault tolerance. Clearly, as α increases, the performance of FTRMFF decreases. Moreover, Fig. 5 shows the values of $(N - M)/M$ for the same experiments, which give the *ratio of additional processors* introduced by FTRMFF to provide fault tolerance with respect to RMFF. Clearly, when active duplication is used for all tasks, this ratio becomes 1. It is observed that, when few tasks must be scheduled and α is small, few processors are needed regardless of the task execution times; hence, the various ratios shown in Fig. 5 are about the same, namely 0.35 for $n = 100$ and both $\alpha = 0.2, 0.5$. When the number of tasks increases, such ratios tend to different values, namely, about 0.3, 0.6, and 0.9 for $\alpha = 0.2$, 0.5, and 0.8, respectively. Thus, Fig. 5 illustrates a remarkable saving of processors with respect to the duplication on two sets of processors of the schedule found by RMFF, when the tasks have a small average execution time. This saving is about 70 percent for $\alpha = 0.2$ and 40 percent for $\alpha = 0.5$.

6 Concluding Remarks

This paper has considered the problem of preemptively scheduling a set of independent periodic tasks under the assumption that each task deadline coincides with the next request of the same task. The proposed FTRMFF algorithm extends the well-known Rate-Monotonic First-Fit scheduling algorithm to tolerate failures, uses a novel combined active/passive duplication scheme, and determines by itself which tasks should use passive duplication and which should use active duplication. Simulation studies revealed a remarkable saving of processors with respect to those needed by the usual active duplication approach in which the schedule of the non-fault-tolerant case is duplicated on two sets of processors. However, further research is needed, e.g., to derive an analytical worst case bound on the number of processors used by the proposed FTRMFF algorithm, or to devise schedulability conditions which are weaker but simpler than the Completion Time Test, e.g., as those proposed in [1].

This paper has assumed that there is no implicit or explicit synchronization between a primary copy and its active copy. However, if they are synchronized, processors can be used more efficiently by reclaiming the time for an active copy if its primary copy terminates successfully. This optimization is left for further work. It is worth noting that the proposed algorithm works also if some backup copies are forced to be active. Thus, further study could be devoted to consider the case in which also some backup copies are forced to be passive, that is, the status (active/passive) of a subset of backup copies is given in input to the algorithm.

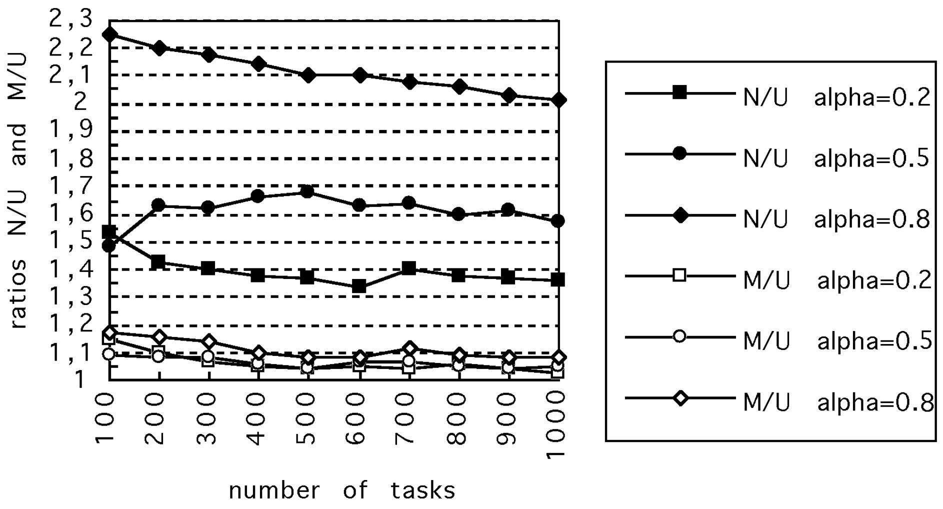

Fig. 4. Ratios between the number of processors required by FTRMFF (upper functions) or RMFF (lower functions) and the total load of the task sets.

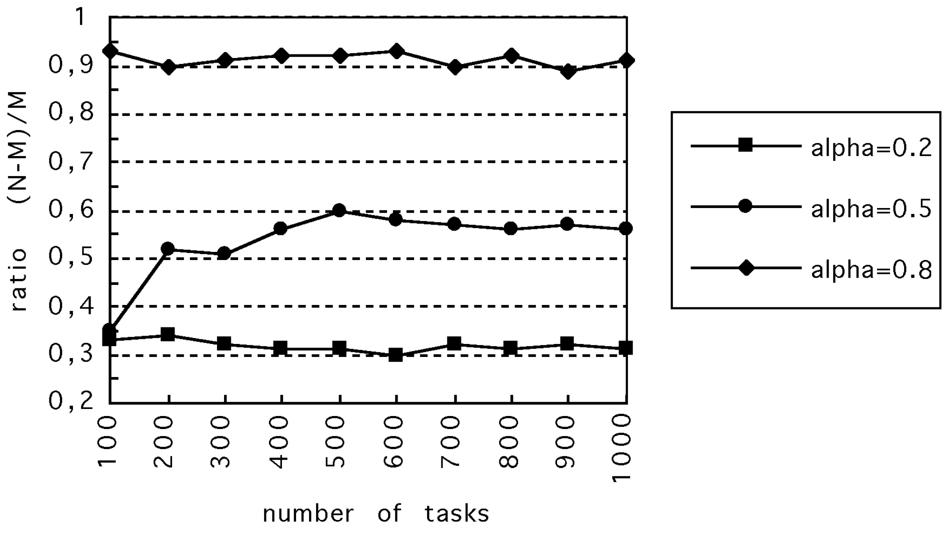

Fig. 5. Ratio $(N - M)/M$ of additional processors required by FTRMFF with respect to RMFF to provide fault tolerance.

A relevant issue is the schedule of a set of tasks subject to precedence constraints and resource requirements [8]. As a subject for future research, the combined duplication scheme proposed in the present paper could be used to extend the Rate-Monotonic First-Fit algorithm in order to tolerate failures also in the presence of resource reclaiming and task synchronization.

Finally, further research could deal with assignment strategies which are different from those considered in this paper. More precisely, tasks could be assigned to processors following an order different from the Rate-Monotonic priority one (such as the order proposed in [1]) while processors could be considered following an order other than the First-Fit one (such as assigning primary tasks to more lightly-loaded processors, thereby easing the effort to schedule their passive backup copies).

Acknowledgments

The C++ code used in the simulation experiments was written by Andrea Fusiello. This work was supported by grants from the Ministero dell'Università e della Ricerca Scientifica e Tecnologica, the Consiglio Nazionale delle Ricerche, and the Università di Trento—Progetto Speciale 1997.

References

[1] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son, "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems," *IEEE Trans. Computers,* vol. 44, no. 12, pp. 1,429-1,442, Dec. 1995.

[2] *Computer and Job/Shop Scheduling Theory,* E.G. Coffman Jr. ed., New York: John Wiley & Sons, 1976.

[3] S.K. Dhall and C.L. Liu, "On a Real-Time Scheduling Problem," *Operations Research,* vol. 26, pp. 127-140, 1978.

[4] S. Ghosh, R. Melhem, and D. Mossé, "Fault-Tolerance through Scheduling of Aperiodic Tasks in Hard-Real-Time Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 3, pp. 272-284, Mar. 1997.

[5] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer J.*, vol. 29, pp. 390-395, Oct. 1986.
[6] K.H. Kim, "Distributed Execution of Recovery Block: An Approach to Uniform Treatment of Hardware and Software Faults," *Proc. Fourth IEEE Int'l Conf. Distributed Computing Systems*, pp. 526-532, San Francisco, Calif., May 1984.
[7] C.M. Krishna and K.G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Trans. Computers*, vol. 35, no. 5, pp. 448-454, May 1986.
[8] M.H. Klein, J.P. Lehoczky , and R. Rajkumar, "Rate-Monotonic Analysis for Real-Time Industrial Computing," *Computer*, pp. 24-33, Jan. 1994.
[9] A.L. Liestman and R.H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Trans. Software Eng.*, vol. 12, no. 11, pp. 1,089-1,095, Nov. 1986.
[10] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, pp. 46-61, 1973.
[11] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and H. Shmoys, "Sequencing and Scheduling: Algorithms and Complexity," *Handbooks in Operations Research and Management Science*, vol. 4, *Logistic of Production and Inventory.* , Amsterdam: North Holland, 1993.
[12] J.Y.-T. Leung and M.L. Merrill, "A Note on Preemptive Scheduling Periodic Real-Time Tasks," *Information Processing Letters*, vol. 11, pp. 115-118, 1980.
[13] S. Ramos-Thuel and J.K. Strosnider, "Scheduling Fault Recovery Operations for Time-Critical Applications," *Proc. Fourth Int'l Conf. Dependable Computing for Critical Applications*, pp. 270-282, Jan. 1994.
[14] J.A. Stankovic, "Decentralized Decision Making for Task Reallocation in Hard-Real-Time Systems," *IEEE Trans. Computers*, vol. 38, no. 3, pp. 341-355, Mar. 1989.

Alan A. Bertossi received the Laurea degree (summa cum laude) in computer science from the University of Pisa, Italy, in 1979. Afterwards he worked as a systems programmer and designer. During 1983-1994, he was with the Department of Computer Science, University of Pisa, as a research associate first and, later, as an associate professor. Since 1995, he has been with the Department of Mathematics, University of Trento, Italy, as a professor of computer science. His main research interests are the algorithmic aspects of high-performance, parallel, distributed, fault-tolerant, and real-time systems. He has published more than 50 refereed papers (two invited) in international journals, conferences, and encyclopedias. He is currently serving as guest coeditor for two special issues of *Algorithmica* and *Discrete Applied Mathematics*, both on experimental algorithmics. His biography is included in the edition of *Who's Who in the World in 1999*.

Luigi V. Mancini received the Laurea degree in computer science from the University of Pisa, Italy, in 1983, and a PhD degree in computer science from the University of Newcastle upon Tyne, United Kingdom in 1989. From 1985-1989, he was with the University of Newcastle upon Tyne, where he worked on the reliability project led by Prof. Brian Randell. From 1989-1992, he was with the Department of Computer Science, University of Pisa, as an assistant professor. From 1993-1996, he was with the Department of Computer Science of the University of Genova, Italy, as an associate professor. Since 1996, he has been with the Department of Computer Science of the University of Roma "La Sapienza," Italy. His current research interests include distributed algorithms and systems, computer and information security, and transaction processing systems.

Federico Rossini received his Laurea degree (summa cum laude) in computer science from the University of Pisa, Italy, in 1993. After spending a year in army service, he worked as a consultant for Telecom Italia, the Italian telephone company, on database design for marketing back office. Since June 1997, he has been employed by Telecom Italia Mobile (TIM) as a software designer. Currently, he is developing and testing new software applications for the marketing and billing systems of the company.