Fault Tolerant Real-Time Global Scheduling on Multiprocessors *

Frank Liberato, Sylvain Lauzac, Rami Melhem, Daniel Mossé
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
(frank, lauzac, melhem, mosse)@cs.pitt.edu

Abstract

Many real-time multiprocessor scheduling techniques have been proposed to guarantee the timely execution of periodic preemptive real-time tasks. However, timeliness is usually only guaranteed in the absence of faults, which may be unacceptable for some critical systems. We therefore address the problem of multiprocessor scheduling for preemptive real-time tasks so that the timeliness of the system can be guaranteed even in the presence of faults. This work focuses on global scheduling *where tasks can migrate across processors. We consider two varieties of global multiprocessor scheduling: in the* frame-based *model, an aperiodic task set is scheduled to create a template (frame), and that schedule may be executed periodically. In the* periodic *model, each task in the set has a separate period, and is executed with no explicitly predetermined schedule. For each model, we show how to guarantee timely execution and recovery in the general case. We also propose solutions that improve upon this general case when all tasks require the same amount of time to recover from a fault.*

1 Introduction

Multiprocessor systems are increasingly being used for real-time applications because of their capability for high performance, reliability and extensibility [19]. Since most real-time applications process periodic tasks, this work only considers periodic tasks, which can be scheduled according to two main paradigms: *frame-based scheduling* or *periodic scheduling*. In a frame-based model, a schedule is built for the length of a frame and each task is assigned processing time in this schedule. The schedule is then repeated frame after frame. In a periodic scheduling paradigm, a task set is submitted to an admission control that decides if the deadlines of all instances of all the tasks can be guaranteed. If such a guarantee can be given, the task set is accepted, and scheduling decisions are made at run-time.

* Supported in part by NSF grant CCR-9308886 and DARPA under contract DABT63-96-C-0044.

Frame-based scheduling

Frame-based scheduling relies on the aperiodic scheduling problem, which can be solved exactly in polynomial time [9]. Each task in the frame is aperiodic. The task set and timeline are transformed into an instance of a maximum network flow problem, the solution to which provides a schedule for the frame if and only if one exists.

Because one can specify the earliest and latest times each task can be scheduled within a frame, the frame-based model allows multiple instances of periodic tasks with small periods to be included in multiple, non-overlapping time inteverals within the frame by treating each instance as a separate aperiodic task. This method can schedule any set of periodic tasks, if the frame length is a multiple of every period in the set.

Periodic scheduling

The periodic scheduling problem has been studied extensively for uniprocessors, but the multiprocessor problem has received less attention, partially due to early negative results. Two major strategies can be used to schedule real-time tasks on a multiprocessor: *partitioning* or *global scheduling* [7]. In a partitioning strategy each task is assigned to a processor. This effectively reduces the multiprocessor scheduling problem to scheduling problems on individual processors [7, 4], for which efficient scheduling techniques are known, such as rate-monotonic scheduling (RMS) and earliest deadline first (EDF) [15] scheduling.

In a global scheduling strategy, all task instances are stored in a single queue and, at any given time, the processors run the instances with the highest priority. Because tasks may switch processors, one might expect global schemes to achieve a higher processor utilization than partitioning schemes. However, global RMS and global EDF can fail to schedule a task set with an arbitrarily low utilization [7]. Other schemes perform better, but still achieve lower utilization than good partitioning algorithms [12]. Based on these negative results, we do not consider global RMS or global EDF further.

However, some other global scheduling techniques appear to be promising. In particular, an optimal global scheduling technique based on the concept of proportionate fairness (P-fairness) has been proposed recently [2] that can fully utilize the resources of a multiprocessor system. Our work for the periodic scheduling model is based on P-fair schedules.

Guarantees in the presence of faults

One problem with multiprocessor systems is that, as the number of processors increases, the probability of a fault increases. In critical real-time systems such as nuclear plant control and avionics control, the goal of real-time scheduling is to *guarantee* the timeliness of the tasks to avoid possible catastrophic consequences. However, this timeliness is usually only guaranteed in the absence of faults. Thus, a natural and important extension of real-time scheduling is to add time for *error recovery* to guarantee the timeliness of the system in the presence of faults. Our work addresses this for frame-based and periodic scheduling.

In order to recover from faults, some form of redundancy is usually necessary. This redundancy can consist of *spatial redundancy* (resource replication) or *temporal redundancy* (executing a recovery procedure after a fault is detected). Using temporal redundancy is problematic in real-time systems, since the execution of the recovery procedure must not hinder the timeliness of the system. It is possible to use a "slack stealing" technique [18] where unused computation time is reclaimed to recover from a fault. Because error recovery using slack-stealing depends on run-time characteristics, it is not always possible to guarantee a priori that error recovery will complete by its deadline. Other schemes guarantee error-recovery by reserving enough resources during admission control. In [8], Ghosh et al. propose an extension to RMS that allows a task to recover from a transient fault by re-executing the faulty instance. If all currently executing instances must re-execute after a single transient fault is detected, the admission control presented in [17] can be used. In [5] several admission controls are described for different recovery mechanisms. Note that time redundancy and spatial redundancy do not exclude each other [3].

The rest of the paper is organized as follows. Section 2 presents the definitions and assumptions on which this work is based. Section 3 addresses the problem of guaranteeing timely recovery on a uniprocessor for the frame-based and the periodic model. Sections 4 and 5 address the same problem in multiprocessor systems, for frame-based and periodic models respectively. The conclusion is presented in Section 6.

2 Problem definitions and assumptions

An aperiodic task can be described by a ready time R, a deadline D, a computation time C, and a recovery time V. In this paper, this is expressed as a 4-tuple (R, D, C, V). A periodic task is described by a computation time, a period and a recovery time (C, T, V), with the understanding that the task is ready at the beginning of its period and has to complete at or before the end of its period. In either the periodic or aperiodic case, a set of n tasks will be denoted by $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$. We assume that migration and context switch times are negligible and that dispatching is instantaneous. We assume that the task sets are to be scheduled on a system with p homogeneous processors. Finally, a task cannot execute in parallel with itself on multiple processors.

In the frame-based model, we assume that at most one task in each frame is affected by one transient fault. For the periodic model, we assume that an instance of a task can be affected by at most one transient fault, and that transient faults are separated by at least $\Delta \geq \max_{i=1,\ldots,n}\{T_i\}$ time units.

We will adopt the convention that a schedule for a particular task set must obey all ready-time, deadline, and computation time restrictions imposed by the task set. A task set is denoted as *feasible* only if there is a schedule for it.

2.1 Necessary conditions for error recovery from transient faults

A real-time schedule can recover from a transient fault if and only if the following three conditions are met.

C1: *In the absence of faults, each task τ_i finishes at least V_i time units before its deadline.* This ensures that there is enough time left for recovery before the deadline.

C2: *When task τ_i needs to recover, it can be allocated V_i time units for recovery.* This ensures that recovery will not fail for lack of resources.

C3: *Task recovery does not cause any other task to miss its deadline.* This ensures that the recovery operation will not violate the guarantees given to other tasks.

Enforcing these conditions is not straightforward. As an example, a well known technique for tolerating faults is to add a spare processor to execute faulty tasks [10]. This technique by itself is an inadequate solution for this problem since it does not satisfy condition **C1**. Also notice that any solution must allow a task to fail during its last instant of computation. Therefore, we assume that fault detection occurs only when a task finishes execution.

It is convenient to express any solution to the problem of fault-tolerant scheduling of n tasks in terms of a set of $n+1$ schedules, $\mathcal{S}=\{S_0, ..., S_n\}$. Schedule S_0 represents the schedule when no recovery takes place. Every other schedule, $S_i, 1 \leq i \leq n$ represents a schedule in which task τ_i fails after consuming exactly C_i units of computation time. S_0 is a schedule of $C_1, ..., C_n$ units of computation time within the timing constraints of $\tau_1, ..., \tau_n$, while $S_i, 1 \leq i \leq n$ is a schedule of $C_1, ..., C_{i-1}, C_i + V_i, C_{i+1}, ..., C_n$ within the timing constraints of $\tau_1, ..., \tau_n$.

Because it is not known a priori which, task will fail, it must be the case that each schedule S_i is the same as S_0, until the time immediately after τ_i has its last unit of computation scheduled (and a fault is detected). In order to use the solution, one could begin executing S_0. As each task τ_i completes at time t_i, it is checked for a failure. If τ_i has failed, execution proceeds along schedule S_i, from $t_i + 1$. Since S_i is identical to S_0 up to and including time t_i, this is indistinguishable from having a priori knowledge that S_i is the 'correct' schedule for the given fault. For periodic task sets, $S_i, 0 \leq i \leq n$ are produced dynamically, while for frame based scheduling they are produced statically.

2.2 Definition of optimal single fault tolerance

We will call a schedule S_0 if each of $S_1 ... S_n$ can be derived from it. We will call such an S_0 a fault tolerant schedule. We define an *optimal* algorithm for the problem of single [transient] fault tolerant scheduling under some recovery scheme, as any algorithm that always produces a schedule S_0 if one exists, given any aperiodic task set, $\mathcal{T}$.

3 Scheduling tasks on a single processor

We first consider the problem of scheduling tasks on a uniprocessor with error recovery for both the frame-based and the periodic models.

3.1 Optimality of EDF in the presence of faults

It is well known that EDF will always produce a schedule for any aperiodic task set on a single processor, if any schedule exists in both the periodic [15] and aperiodic[6] cases. We claim that EDF is also optimal even if a transient fault can occur.

Intuitively, because EDF does not make scheduling decisions based on computation time, the computation time is free to vary during execution; EDF does not distinguish between the fault-free schedule S_0, and another schedule S_f, until the time t_f at which τ_f fails. We now prove this more formally.

Theorem 1 *EDF scheduling tolerates any single fault in any task set on a single processor, if it is possible to do so via any scheduling method.*

PROOF: Consider an aperiodic task set $\mathcal{T} = \{\tau_1, ..., \tau_n\}$, and a schedule S_0 for it produced by EDF. (Recall that any periodic task set can be transformed into an aperiodic task set, if enough instances of each task are added to reach the Least Common Multiple of the task periods.) We will consider the case when some task, $\tau_f = (R_f, D_f, C_f, V_f)$ fails. Derive a new task set $\mathcal{T}'_f = \{\tau'_1, ..., \tau'_n\}$ from $\mathcal{T}$, where

$$\forall i \in [1, n], i \neq f \quad \tau'_i = \tau_i$$
$$\tau'_f = (R_f, D_f, C_f + V_f, V_f)$$

Let S_f denote the schedule produced for $\mathcal{T}'_f$ by EDF.

Let t_f denote the time in S_0 at which τ_f completes. We claim that S_f schedules task τ'_i whenever S_0 schedules the task τ_i, up to time t_f. We prove this by contradiction. Assume the schedules differ before time t_f. Let time $t_d < t_f$ denote the earliest such time. Let τ_i represent the task scheduled in S_0 at time t_d, and τ'_j represent the task scheduled in S_f at time t_d, $i \neq j$. By definition of EDF, either τ'_i has completed by time t_d in S_f, or τ_j has completed in S_0 by time t_d. It is obvious that both situations provide a contradiction.

By conditions **C1**-**C3**, if a schedule for $\mathcal{T}'_f$ exists, it can be used to tolerate faults; S_0 and S_f schedule tasks with equal subscripts at every time at least until time t_f. If this is true for all $1 \leq \tau_f \leq n$, it is obvious that EDF scheduling is sufficient to tolerate a single transient fault in $\mathcal{T}$.

It is a simple argument that EDF scheduling is also necessary: if $\mathcal{T}'_f$ cannot be scheduled by EDF for some $1 \leq f \leq n$, then it cannot be scheduled by any method. Therefore, EDF is an optimal fault-tolerant scheduling algorithm on a single processor. □

3.2 Admission control

To use EDF as a fault-tolerant frame-based scheduling algorithm on a single processor for aperiodic task sets, one can simply verify that S_f can be scheduled for each modified task set $\mathcal{T}'_f$, $1 \leq f \leq n$. This implies that any task τ_f that fails can recover according to its unmodified EDF priority, without missing its deadline or forcing any other task τ_i to execute past D_i. This admission control is polynomial in the size of the task set, n; one simply constructs n EDF schedules for $\mathcal{T}'_1, ..., \mathcal{T}'_n$.

For the case of periodic task sets, we provide in Theorem 2 an admission control for EDF that guarantees timely recovery.

Theorem 2 *Given a task set $\mathcal{T}$ of n tasks scheduled with EDF, when one transient fault affects any task τ_f, τ_f can execute a recovery of duration V_f before its deadline if and only if*

$$\sum_{i=1}^{n}[\frac{C_i}{T_i}] + \max_{i=1,\dots,n}\{\frac{V_i}{T_i}\} \leq 1 \tag{1}$$

PROOF:
Sufficiency: We need to prove that no deadline is missed regardless of which task fails. Recovering from a transient fault for any task τ_f is equivalent to executing for $C_f + V_f$ for the period T_f when the fault occurs. Hence, to guarantee schedulability in the presence of a transient fault, it suffices to guarantee that

$$\forall f = 1,\dots,n \quad \sum_{\substack{i=1 \\ i\neq f}}^{n}[\frac{C_i}{T_i}] + \{\frac{C_f + V_f}{T_f}\} \leq 1 \tag{2}$$

Relation (1) implies Relation (2).
Necessity: Assume that there exist an algorithm that can schedule a task set rejected by EDF. Since this task set was rejected by EDF it must be that

$$\sum_{i=1}^{n}[\frac{C_i}{T_i}] + \max_{i=1,\dots,n}\{\frac{V_i}{T_i}\} > 1 \tag{3}$$

Denote τ_m the task with the maximum utilization. If each instance of τ_m fails, it must execute for $C_m + V_m$. This is equivalent to scheduling a task set of utilization

$$U = \sum_{i=1}^{n}[\frac{C_i}{T_i}] + \max_{i=1,\dots,n}\{\frac{V_i}{T_i}\}$$

But Relation (3) shows that this task set has a utilization greater than 1 and cannot be scheduled on a single processor, leading to a contradiction. □

4 Frame-based scheduling with error recovery on a multiprocessor

4.1 Optimal scheduling of tasks with equal recovery times

Given an aperiodic task set $\mathcal{T}$ in which every task τ_i has the same recovery time, V, and an alternate task set $\mathcal{T}'$ which is identical to $\mathcal{T}$ except that every task τ'_i in $\mathcal{T}'$ has its deadline reduced by V ($D'_i = D_i - V_i$).

We can make the following claim:

Theorem 3 *A fault tolerant schedule for $\mathcal{T}$ exists if and only if any schedule exists for $\mathcal{T}'$.*

PROOF: By condition **C1**, it is necessary to find a schedule for $\mathcal{T}'$ to tolerate faults in $\mathcal{T}$; if no schedule for $\mathcal{T}'$ exists, it is impossible for every task in $\mathcal{T}$ to finish early enough to allow sufficient time for recovery.

We now show that if a schedule S_0 for $\mathcal{T}'$ exists, it is sufficient to tolerate faults in $\mathcal{T}$. Assume task τ'_f fails. Let V denote the recovery time. Let t_f denote the time at which τ' fails. To recover, one simply begins recovery of τ' at t_f on some processor, leaving all other processors idle until $t_f + V$, at which point one resumes the original schedule from time t_f. With this scheme, every task will complete (possibly with reexecution) at most V time units later than specified in S_0, which is not later than the original, unmodified deadline, as required by **C2** and **C3**. □

Figure 1 provides an example of this for two processors. Figure 1(a) shows the fault-free schedule S_0. If a fault occurs as indicated, Figure 1(b) shows the schedule that will actually be executed.

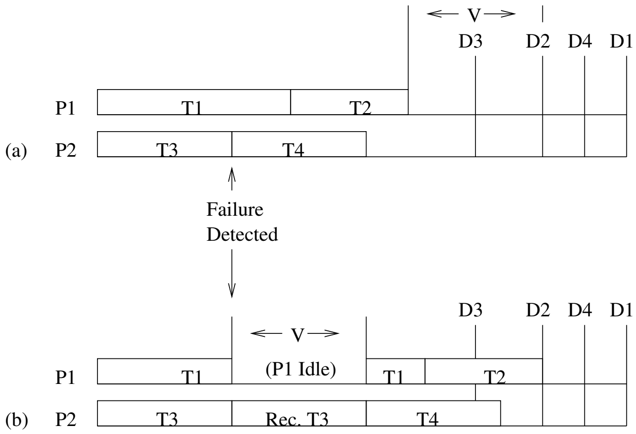

Figure 1. Recovery procedure assuming equal recovery times

4.2 Heuristic for the general case of fault tolerant scheduling

In the general aperiodic case, there may not necessarily be any relationship between the recovery times of any two tasks. For this case, we provide a heuristic to find a fault tolerant schedule. Given a task set $\mathcal{T}$, we try to find a candidate for schedule S_0, and then verify if one can derive $S_1, ..., S_n$ from it.

Since the fault-free execution of $\mathcal{T}$ in any fault-tolerant schedule necessarily must complete every task τ_i before $D_i - V_i$, an obvious starting point for an heuristic is to derive a task set $\mathcal{T}'$ from $\mathcal{T}$, by reducing the deadline of every task by its recovery time. Observe that if no schedule exists for $\mathcal{T}'$, then no fault tolerant schedule exists for $\mathcal{T}$. However, Figure 2 shows an example of a task set for two processors that can be scheduled under reduced deadline constraints but has no fault tolerant schedule, if $V_1 = C_1, V_2 = V_3 = 0$.

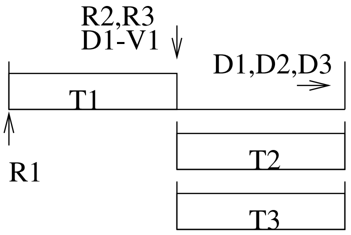

Figure 2. Task set with no FT schedule

We must now find a schedule for $\mathcal{T}'$ that can be used as S_0 for $\mathcal{T}$; i.e., $S_1, ..., S_n$ can be derived from it. No method is known that will always select such a schedule from the set of all schedules for $\mathcal{T}'$, if one exists. Our heuristic (labelled H1) is quite simple: **select *any* schedule for $\mathcal{T}'$**. To find a candidate schedule for S_0 from the modified task set, and to verify that $S_1, ..., S_n$ can be derived from it, we use the maximum network flow method [9].

To implement this method of fault tolerant scheduling, one would generate all $n + 1$ schedules off-line, and switch between them as needed dynamically.

4.3 Evaluation of the heuristic

Our simulator produces random task sets with utilization ranging from 0% to 100% in the fault-free case. The utilization of a task set $\mathcal{T}$, $U_{\mathcal{T}}$ is given by:

$$U_{\mathcal{T}} = (\sum_i \frac{C_i}{p}) \times (max_{i\in[1,n]} D_i - min_{i\in[1,n]} V_i) \qquad (4)$$

The recovery time, V_i, for each task τ_i is equal to C_i. This models a system in which one simply re-executes a failed task.

To evaluate our heuristic, our simulator records the *Number of schedulable task sets generated*, without any consideration for fault-tolerance, and *the number of task sets that were estimated to have a fault tolerant schedule.* We consider a task set to have a fault tolerant schedule on a uniprocessor if EDF can tolerate a fault in any task. On a multiprocessor, we simply try to schedule a reduced-deadline (labelled RD) version of the task set). This measures the estimated cost of fault tolerance regardless of the heuristic chosen. Our simulation also measures *the number of task sets successfully scheduled by H1.* All results are expressed as a percentage of the number of generated task sets that can be scheduled without regard to fault tolerance.

On a single processor, Figure 3 shows a large difference between the performances of EDF and H1, by over 50% at worst. Of course, EDF is superior in complexity on a single processor. Also note that RD is necessary but not sufficient; EDF is optimal but does not meet the predictions of RD.

For two processors, (Figure 4), the performance of H1 is very close to that of RD. When the simulation was run for three or more processors, this difference disappears entirely. From this, we conclude that the necessary condition for fault-tolerant scheduling is [empirically] close to a sufficient condition as well, for $p \geq 2$. Further, an optimal scheduler would not produce significantly more successes than H1. Since no optimal scheduler is known for $p \geq 2$, we conclude that H1 is a useful heuristic.

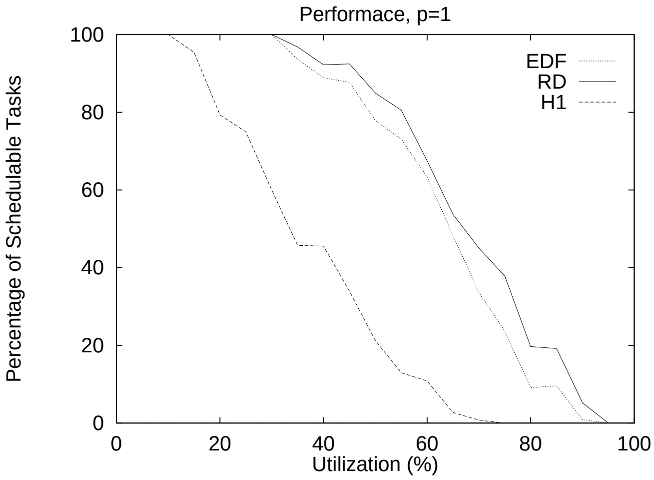

Figure 3. Uniprocessor Performance

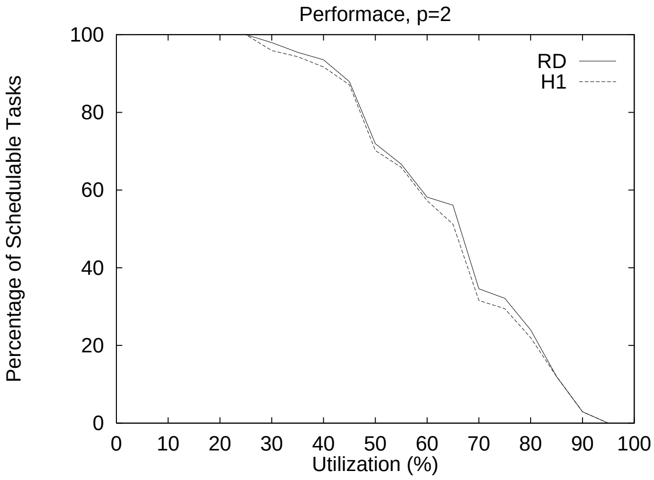

Figure 4. Two Processor Performance

5 Multiprocessor scheduling of periodic tasks with error recovery

5.1 Heuristic solution for the general case

As mentioned in Section 1, a global scheduling algorithm was recently developed, algorithm PF, which does not suffer from the poor theoretical bounds of global RMS and global EDF [1]. Algorithm PF uses the concept of slotted time where each processor executes a given task set for the length of the time slot. It is assumed that task periods and execution times are multiples of the time slot. Without loss of generality, the length of a time slot is assumed to be a unit.

Given a task set $\mathcal{T} = \{(C_1, T_1), (C_2, T_2), \ldots, (C_n, T_n)\}$ and p processors, algorithm PF accepts $\mathcal{T}$ if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq p \tag{5}$$

Relation (5) means that algorithm PF can schedule each processor up to 100% and is therefore optimal. An important property of algorithm PF is proportionate fairness, or P-fairness: the number of time slots, $\sigma(t, i)$, allocated to task τ_i during a time interval of length t is such that

$$\forall i, t : i \in [1, n], t \in N : \lfloor t \times U_i \rfloor \leq \sigma(t, i) \leq \lceil t \times U_i \rceil \tag{6}$$

In other words, algorithm PF guarantees that each task receives its allocation on a regular basis, based on its utilization U_i.

This section presents algorithm FT-PF that schedules a set of real-time tasks $\mathcal{T}= \{(C_1, T_1, V_1), (C_2, T_2, V_2), \ldots, (C_n, T_n, V_n)\}$ on p processors and recovers from transient faults. Algorithm FT-PF is based on the P-fair algorithm PF with two modifications that enforce the conditions given in Section 2.1. First, FT-PF ensures that each task finishes early enough to re-execute in case of a transient fault (condition **C1** in Section 2). Second, FT-PF uses a spare processor to satisfy conditions **C2** and **C3**.

In order to make each task finish early enough for re-execution, FT-PF assigns more resources to each task by increasing the utilization of each task. The intuition behind this technique is the following: to ensure that each task τ_i finishes its execution at least V_i time units before the deadline, we increase the utilization of the task. This way, slots are allocated closer together and all its C_i time units are allocated early enough. Formally, the next Lemma shows the amount of increase in resource allocation that is necessary to satisfy condition **C1**.

Lemma 1 *Given p processors and a task set $\mathcal{T}$, consider the task set $\mathcal{T}'$ where*

$$\forall i \in [1, n], \quad C'_i = \lceil \frac{C_i T_i}{T_i - V_i} \rceil \quad T'_i = T_i \tag{7}$$

If there is a P-fair schedule, S', generated by applying PF on $\mathcal{T}'$ for p processors, then S' is also a feasible schedule for $\mathcal{T}$ such that each task τ_i completes V_i time units before its deadline in every period.

PROOF: Since schedule S' is created by PF, Equation (6) ensures that during each interval $[(k-1)T_i, kT_i - V_i]$ ($k \in N$), task τ_i receives at least

$$\lfloor U'_i(T_i - V_i) \rfloor \geq \lfloor \frac{C_i T_i}{(T_i - V_i) T_i} (T_i - V_i) \rfloor = C_i \tag{8}$$

time units. Therefore, in every period, task τ_i completes (i.e. executes its C_i units of computation) at least V_i time units before the deadline. □

According to Lemma 1, in the worst case, a task instance of τ_i may finish exactly V_i time units before its deadline. Further, it requires V_i units to recover in case an error is detected, which implies that the utilization of the task increases to 1 at this instant. Thus, it is straightforward to use one of the p processors as a spare processor for the re-execution, which gives algorithm FT-PF (figure 5).

```
FT-PF (In: T, p)
begin
   for i = 1 to m do
      T'_i = T_i
      C'_i = ⌈ C_i T_i / (T_i − V_i) ⌉
   endfor
   for each time slot s do
      use PF to schedule T' on p − 1 processors
      if (transient fault detected for τ_f)
         recover τ_f on the spare processor
      endif
   endfor
end
```

Figure 5. Algorithm FT-PF

The correctness of FT-PF is proven in the next Theorem.

Theorem 4 *Given a task set $\mathcal{T}$ and p processors, if*

$$\sum_{i=1}^{n} [\frac{\lceil \frac{C_i T_i}{T_i - V_i} \rceil}{T_i}] + 1 \leq p \tag{9}$$

then algorithm FT-PF schedules $\mathcal{T}$ such that all the tasks in $\mathcal{T}$ always complete (including a possible recovery) before their deadlines even in the presence of transient faults.

PROOF: If no fault occurs, each task τ_i is allocated $C'_i \geq C_i$ time slots every T_i, and completes each instance before its deadline. If a transient fault occurs, we show that the three conditions given in Section 2.1 are met. First, Lemma 1 shows that condition **C1** holds. Second, since a faulty task recovers on a spare processor on which no task was scheduled, a faulty task has the resources necessary for recovery (condition **C2**). Last, since recovery is executed on a spare processor, it does not influence the other tasks, thus condition **C3** is satisfied. □

5.2 Case of equal recovery times

In the general case, increasing the task utilization according to Equation (7) is not sufficient to provide enough slack for recovery (condition **C2** and **C3**) and a spare processor is required. However, there are cases in which the increase in task utilization also satisfies conditions **C2** and **C3**. One of these cases is when all recovery times are equal

$$\forall i \in [1, n], \quad V_i = V$$

For this case, we present a modified version of FT-PF, ER-PF, that does not require a spare processor. Similarly to algorithm FT-PF, ER-PF schedules tasks by using PF with computation times as defined by Equation (7), but uses a different recovery mechanism. When a fault is detected, the system enters a recovery mode for a duration of V. In recovery mode, the faulty task executes its recovery and all other tasks are suspended. After V time units, normal execution resumes and tasks continue executing according to the schedule that would have been followed during recovery mode.

The following theorem proves the correctness of algorithm ER-PF.

Theorem 5 *Given a task set $\mathcal{T}$ and p processors, if*

$$\sum_{i=1}^{n} \left[\frac{\left\lceil \frac{C_i T_i}{T_i - V} \right\rceil}{T_i} \right] \leq p \qquad (10)$$

then algorithm ER-PF schedules $\mathcal{T}$ such that all the tasks in $\mathcal{T}$ always complete (including a possible recovery) before their deadlines even in the presence of transient faults.

PROOF: By construction, each task finishes V time units before its deadline, thus condition **C1** is met. By design of the recovery procedure, a faulty task is given V time units to recover, and condition **C2** holds. It remains to show that condition C3 is also met, that is, the delay caused by the recovery does not cause a non faulty task to miss its deadline. Consider an arbitrary task of period T that is delayed by V: during the period of T, algorithm PF schedules this task over $T - V$ time slots. From Equation (8), we know that a task scheduled in a interval of $T - V$ is allocated at least C units, thus does not miss its deadline. □

Similarly to Section 4, we see that when all recovery times are equal, enforcing condition **C1** also enforces conditions **C2** and **C3** and is therefore sufficient to guarantee recovery from transient faults.

5.3 Performance evaluation

The performance of the proposed algorithms, measured by the average processor utilization achieved, is obtained via simulation.

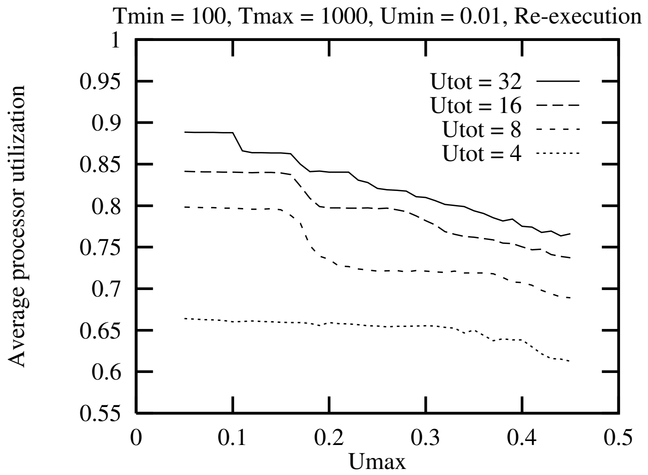

Figure 6. Performance evaluation of FT-PF

Tasks are randomly generated according to four parameters, minimum and maximum task period (T_{min} and T_{max}), minimum and maximum task utilization (U_{min} and U_{max}). For each task, the worst-case computation time C is randomly drawn from $[1, T_{min}]$ with a uniform distribution and the period T is randomly drawn from $[T_{min}, T_{max}]$ with a uniform distribution such that $U_{min} \leq \frac{C}{T} \leq U_{max}$ holds. A fifth parameter, U_{tot}, is used to generate a task set that spans several processors. Tasks are generated and added to the task set until the sum of their utilization exceeds U_{tot}. This task set is given to the scheduling algorithms and each algorithm returns the number of processors needed to accept this task set. If an admission control requires P processors to schedule a task set with total utilization U_{tot}, then we define the average processor utilization of this admission control to be U_{tot}/P. Experiments are repeated 1,000 times per algorithm and the achieved processor utilizations are averaged.

Figure 6 shows the performance of FT-PF when tasks re-execute in order to recover from a fault (i.e., $\forall i \in [1, n], \quad V_i = C_i$). The average processor utilization decreases as U_{max} increases, since the overhead of finishing early is larger for tasks with large utilizations. When U_{tot} is large FT-PF performs better since the relative overhead of the spare processor decreases. Figure 7 compares the performance of FT-PF and ER-PF. When all recovery times are equal to 10, FT-PF has an overhead of 12% for all task utilizations. This overhead can be reduced to 6% if ER-PF is used instead. Comparison of Figure 6 and Figure 7 for $V = 50$ shows that variation in recovery times decreases the processor utilization. When tasks must re-execute for recovery, the average recovery time is 49.5 and yet the processor utilization is lower than when all tasks have a recovery time of 50.

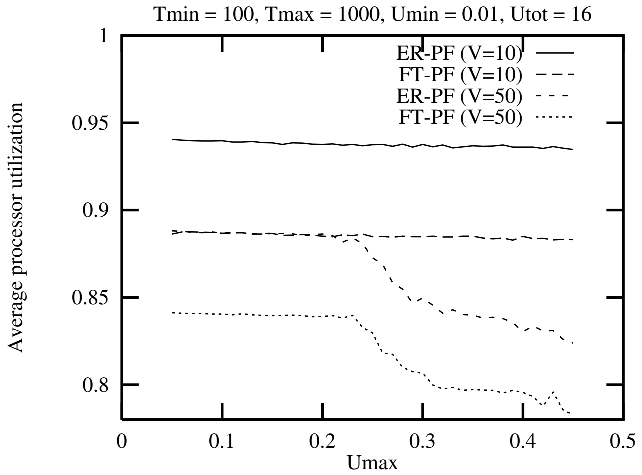

Figure 7. Comparison of FT-PF and ER-PF

6 Conclusion

Timely error recovery in real-time multiprocessor systems is an important issue for many critical systems. Our work proposes solutions to this problem for two major scheduling paradigms: frame-based scheduling and periodic scheduling. We present an optimal scheduling with error recovery for uniprocessor systems. For multiprocessor systems, we provide an exact solution for frame-based systems when all recovery times are equal, and an heuristic with a very high success rate for the general case. For periodic tasks, we propose heuristics which achieve very high processor utilization.

References

[1] S. K. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15:600–625, 1996.

[2] S. K. Baruah, J. E. Gehrke, and G. Plaxton. Fast Scheduling of Periodic Tasks on Multiple Resources. In *International Parallel Processing Symposium*, pages 280–288, 1995.

[3] A. Bertossi, A. Fusiello, and L. Mancini. Fault-Tolerant Deadline-Monotonic Algorithm for Scheduling Hard-Real-Time Tasks. In *International Parallel Processing Symposium*, pages 133–138, 1997.

[4] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Trans. on Computers*, 44(12):1429–1442, 1995.

[5] A. Burns, R. Davis, and S. Punnekkat. Feasiblity Analysis of Fault-Tolerant Real-Time Task Sets. In *Euromicro Workshop on Real-Time Systems*, pages 29–33, 1996.

[6] H. Chetto and M. Chetto. An Adaptive Scheduling Algorithm for Fault-Tolerant Real-Time Systems. *Software Engineering Journal*, pages 93–100, 1991.

[7] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

[8] S. Ghosh, R. Melhem, D. Mossé, and J. S. Sarma. Fault-Tolerant Rate-Monotonic Scheduling. *Journal of Real-Time Systems*, 1998.

[9] W. A. Horn. Some Simple Scheduling Algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[10] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison Wesley Pub. Co., 1989.

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

[12] S. Lauzac, R. Melhem, and D. Mossé. Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor. In *Euromicro Workshop on Real-Time Systems*, pages 188–195, 1998.

[13] J. Y.-T. Leung. A New Algorithm for Scheduling Periodic Real-Time Tasks. *Algorithmica*, 4:209–219, 1989.

[14] J. Y.-T. Leung and J. Whitehead. On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[15] C. L. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):47–61, 1973.

[16] Y. Oh and S. Son. Enhancing Fault-Tolerance in Rate-Monotonic Scheduling. *The Journal of Real-Time Systems*, 7(3):315–329, Nov 1994.

[17] M. Pandya and M. Malek. Minimum Achievable Utilization for Fault-tolerant Processing of Periodic Tasks. *IEEE Trans. on Computers*, 47(10):1102–1112, 1998.

[18] S. Ramos-Thuel and J. Strosnider. Scheduling Fault Recovery Operations for Time-Critical Applications. In *4th IFIP Conference on Dependable Computing for Critical Applications*, Jan 1995.

[19] K. Shin and P. Ramanathan. Real-Time Computing: A New Discipline of Computer Science and Engineering. *Proc. IEEE*, 82(1):6–24, 1994.