An Improved Rate-Monotonic Admission Control and Its Applications

Sylvain Lauzac, Rami Melhem, *Fellow*, *IEEE*, and
Daniel Mossé, *Member*, *IEEE Computer Society*

Abstract—Rate-monotonic scheduling (RMS) is a widely used real-time scheduling technique. This paper proposes RBound, a new admission control for RMS. RBound has two interesting properties. First, it achieves high processor utilization under certain conditions. We show how to obtain these conditions in a multiprocessor environment and propose a multiprocessor scheduling algorithm that achieves a near optimal processor utilization. Second, the framework developed for RBound remains close to the original RMS framework (that is, task dispatching is still done via a fixed-priority scheme based on the task periods). In particular, we show how RBound can be used to guarantee a timely recovery in the presence of faults and still achieve high processor utilization. We also show how RBound can be used to increase the processor utilization when aperiodic tasks are serviced by a priority exchange server or a deferrable server.

Index Terms—Real-time, scheduling, rate monotonic, operating systems.

1 Introduction

Tasks in a real-time system must produce functionally correct results in a timely manner. This implies that the tasks submitted to the system have known timing requirements. Many of these *real-time tasks* (such as in process control systems) are periodic and modeled as follows: At the beginning of each period, a new instance of the task is generated and is immediately available for processing. The processing of each task instance must be completed by the end of the task's period, called the *deadline* of the instance. Typically, the requirements of a periodic real-time task τ_i are characterized by a period T_i and a worst-case computation time C_i [16].

Given this task model, a real-time system must ensure that each task instance will complete before its deadline. This is done by using an *admission control* and a *scheduling policy* for the real-time system. The admission control is an algorithm that depends on the scheduling policy and ensures that only tasks that will meet their deadlines are accepted into the system. The scheduling policy determines which task instance is to be processed next.

One of the most widely used uniprocessor scheduling policies for preemptive periodic real-time tasks is the *rate-monotonic scheduling* (RMS), proposed by Liu and Layland in [16]. RMS associates each task τ_i with a fixed priority $p_i = 1/T_i$. At any time, the available task with the highest priority is processed. It is assumed that preemption time is negligible. An admission control for RMS based on processor utilization is also given in [16]. The *utilization* of task τ_i is defined to be C_i/T_i and the utilization of a task set is the sum of the utilizations of all tasks in the task set. This admission control compares the utilization of the task set to a bound that depends only on the number of tasks in the set and shows that a set of m tasks will not miss any deadline if

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq U_{LL}(m) = m(2^{1/m} - 1). \quad (1)$$

Until recently, rate-monotonic scheduling theory has been mainly developed for uniprocessor environments. However, multiprocessor systems are increasingly being used for real-time applications because of their capability for high performance, reliability, and extensibility [22]. As more real-time systems are now deployed for common application domains, such as the aerospace and automotive industries, *efficiency* requirements become a prime concern. Unfortunately, the problem of optimally scheduling a set of preemptive periodic tasks on a multiprocessor system using fixed priorities is known to be NP-complete [15]. Therefore, we address the problem of designing an efficient multiprocessor rate-monotonic scheduling algorithm.

Section 2 proposes RBound, a uniprocessor schedulability condition for RMS. Section 3 shows why RBound is well adapted to multiprocessor environments. The efficiency of RBound is measured by the average processor utilization achieved and is compared with other multiprocessor scheduling algorithms. When RMS is implemented on real-life systems, several issues must be addressed, such as aperiodic task servicing and shared resources access. A large body of work has been developed to provide solutions to these problems for uniprocessor RMS. To benefit from these solutions in the multiprocessor context, it is important to develop a multiprocessor RMS solution that remains close to the uniprocessor problem. In order to illustrate the

- *S. Lauzac is with Akamai Technologies, 1011 Western Ave., Seattle, WA 98104. E-mail: slauzac@akamai.com.*
- *R. Melhem and D. Mossé are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260. E-mail: {melhem, mosse}@cs.pitt.edu.*

Manuscript received 9 Apr. 1999; revised 2 Mar. 2001; accepted 21 Sept. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 109569.

portability of our approach, we consider the following two problems: error recovery and aperiodic task servicing.[1] Section 4 shows how RBound can be used to provide timeliness guarantees in the presence of faults and analyzes the performance of the proposed algorithms. Section 5 shows how RBound can improve the efficiency of aperiodic task servicing. The conclusion is presented in Section 6.

2 Uniprocessor Admission Control

This section describes RBound, a utilization bound for RMS that uses information about the tasks periods to obtain high processor utilization, while keeping the execution model the same as RMS (that is, task dispatching is still done via a fixed-priority scheme based on the task periods).

U_{LL}, given in (1), is a general bound used for admission control of RMS-based systems that only requires information about the number of tasks to be admitted in the system. If more information about the tasks characteristics is used, a better admission control can be obtained. For example, the exact characterization described by Lehoczky et al. in [13] uses C_i and T_i values for admission control. In [2], Burchard et al. use knowledge about the periods of the tasks to obtain better admission control for RMS tasks. Clearly, using more information increases the complexity of the admission control since there are more variables to take into consideration during admission control. The complexity of an admission control is particularly important in real-time systems with dynamically arriving tasks where the scheduling overhead must remain low. This is even more relevant in multiprocessor systems, where the admission control may be evaluated several times per task (until a suitable processor is found). On the other hand, using more information during the admission control may help elevate the processor utilization since a more accurate procedure can be used. As we can see, there may be a tradeoff between the overheads of an admission control scheme and its accuracy; we intend to investigate this tradeoff, for uniprocessor systems in this section and for multiprocessor systems in the next section.

To keep the complexity of RBound low, instead of using information about all the tasks periods, this information is captured in a single value r called *period ratio*, which is equal to the ratio of the largest to the smallest period among all tasks in the system.

Given an input task set $\mathcal{T}$, RBound first transforms $\mathcal{T}$ into $\mathcal{T}'$ according to algorithm ScaleTaskSet given in Fig. 1. ScaleTaskSet finds an equivalent task set where the ratio between maximum and minimum periods is less than 2. An admission control based on RBound is then applied to $\mathcal{T}'$. We will show in Lemma 2 that if $\mathcal{T}'$ is *schedulable* (that is, there exists a RM schedule in which no task misses any deadline), then $\mathcal{T}$ is also schedulable. To prove Lemma 2, we use Lemma 1 stated below and proven in [2].

Lemma 1. *Given a task set $\mathcal{T} = \{(C_1, T_1), (C_2, T_2), \ldots, (C_m, T_m)\}$, ordered by increasing periods if $\mathcal{T}$ cannot be scheduled on one processor with RMS, the task set $\{(2C_1, 2T_1), (C_2, T_2), \ldots, (C_m, T_m)\}$ cannot be scheduled on one processor with RMS either.*

1. ScaleTaskSet (In: $\mathcal{T}$, Out: $\mathcal{T}'$)
2. begin
3. sort the tasks in $\mathcal{T}$ by increasing period
4. for $i = 1$ to $m - 1$ do
5. $T_i' = T_i 2^{\lfloor \log \frac{T_m}{T_i} \rfloor}$
6. $C_i' = C_i 2^{\lfloor \log \frac{T_m}{T_i} \rfloor}$
7. endfor
8. sort the tasks in $\mathcal{T}'$ by increasing period
9. return ($\mathcal{T}'$)
10. end

Fig. 1. Algorithm ScaleTaskSet.

Lemma 2. *Given a task set $\mathcal{T}$, let $\mathcal{T}'$ be the task set resulting from the application of the algorithm ScaleTaskSet to $\mathcal{T}$. If $\mathcal{T}'$ is schedulable on one processor using RMS, then $\mathcal{T}$ is schedulable on one processor with RMS.*

Proof. The intuition behind this proof is that, for each task τ_i, C_i and T_i are modified in lines 5 and 6. This modification can be viewed as doubling C_i and T_i $\lfloor \log \frac{T_m}{T_i} \rfloor$ times. Each time C_i and T_i are doubled, the converse of Lemma 1 ensures that, if the modified task set is schedulable, then the input task set is also schedulable.

Formally, we use the auxiliary algorithm in Fig. 2, which computes the same function as the algorithm in Fig. 1. Note that, at the end of this algorithm, all tasks have $T_m/2 < T_i \leq T_m$ because the loop ends when the smallest period is larger than $T_m/2$. Therefore, at each iteration of the algorithm, $T_1 < T_m/2 \Rightarrow 2T_1 < T_m$.

The proof proceeds by induction: For the base case, note that the first iteration of the algorithm is immediately proven by the converse of Lemma 1. For the induction step, assume that the lemma holds for the resulting task set after the first k iterations of the loop. This means that $\exists i | T_i < T_m/2$ (if not, the algorithm would have stopped) such that the lemma holds. Therefore, the converse of Lemma 1 will guarantee that the lemma holds after one more iteration of the algorithm, which doubles T_1 and C_1, for the task with the smallest T_i.

1. ScaleTaskSet_aux (In: $\mathcal{T}$, Out: $\mathcal{T}'$)
2. begin
3. sort the tasks in $\mathcal{T}$ by increasing period
4. while $T_1 \leq T_m/2$ do
5. $C_i = 2C_i$
6. $T_i = 2T_i$
7. sort the tasks in $\mathcal{T}$ by increasing period
8. endwhile
9. $\forall i, T_i' = T_i; C_i' = C_i;$
9. return ($\mathcal{T}'$)
10. end

Fig. 2. Auxiliary Algorithm ScaleTaskSet.

1. A systematic methodology that mechanically transforms RBound to RM is still needed to claim that RBound can actually profit from all the extensions created for RM. The error recovery and aperiodic task servicing are intended as illustrative and useful examples.

At the end of the algorithm, all tasks have $T_i > T_m/2$. Clearly, since the algorithm is only executed when $T_1 \leq T_m/2$ and only T_1 is doubled at each iteration of the algorithm, it cannot be the case that $T_i > T_m$. Therefore, $\forall i, T_m/2 < T_i \leq T_m$. □

We now focus on the schedulability of the scaled task set $\mathcal{T}'$. The proof of Theorem 4 in [16] shows that, if the ratio between any two tasks period is less than 2, the computation times that minimize the utilization are

$$C'_i = T'_{i+1} - T'_i \;\; (i = 1, \ldots, m-1) \;\text{ and }\; C'_m = 2T'_1 - T'_m.$$

It is easy to see that rewriting the task set utilization with these computation times yields

$$\sum_{i=1}^{m} \frac{C'_i}{T'_i} = \sum_{i=1}^{m-1} \left\lceil \frac{T'_{i+1}}{T'_i} \right\rceil + 2\frac{T'_1}{T'_m} - m. \tag{2}$$

In order to have an admission control criteria that does not depend on the periods of all of the tasks, we need to derive a least upper bound on the processor utilization given by (2). Given a task set of m tasks, let $r = \frac{T'_m}{T'_1}$, where T'_1 and T'_m are the minimum and maximum periods in $\mathcal{T}'$ and $r < 2$. Let $U(r, m)$ be the utilization rewritten from (2):

$$U(r, m) = \sum_{i=1}^{m-1} \left\lceil \frac{T'_{i+1}}{T'_i} \right\rceil + \frac{2}{r} - m. \tag{3}$$

$U(r, m)$ depends on r, m, and the periods $T'_1, \ldots, T'_m$. We define $U_{RBound}(r, m)$ to be the minimum of $U(r, m)$ with respect to the periods, that is, the least upper bound on the processor utilization for a given m and r. Lemma 3 gives an expression for $U_{RBound}(r, m)$.

Lemma 3. *Consider a set of m tasks ordered by increasing periods, where the minimum and maximum periods are fixed and known and the ratio r of any two periods is less than 2. The minimum processor utilization for this task set, $U_{RBound}(r, m)$, is*

$$U_{RBound}(r, m) = (m-1)(r^{1/(m-1)} - 1) + 2/r - 1. \tag{4}$$

Proof. $U(r, m)$ is minimum when its derivative with respect to T'_i is null for $i = 2, \ldots, m-1$.

$$\begin{aligned} &\forall i \,|\, 1 < i < m, \;\; \frac{\partial U(r,m)}{\partial T'_i} = 0 \;\; \Rightarrow -\frac{T'_{i+1}}{T'^2_i} + \frac{1}{T'_{i-1}} = 0 \\ &\Rightarrow \frac{T'_i}{T'_{i-1}} = \frac{T'_{i+1}}{T'_i}. \end{aligned} \tag{5}$$

This system of equations gives

$$\frac{T'_2}{T'_1} = \frac{T'_3}{T'_2} = \ldots = \frac{T'_m}{T'_{m-1}} = r^{1/(m-1)}$$

which can be used to rewrite (3) as

$$U_{RBound}(r, m) = (m-1)r^{1/(m-1)} + 2/r - m, \tag{6}$$

which yields (4) to complete the proof. □

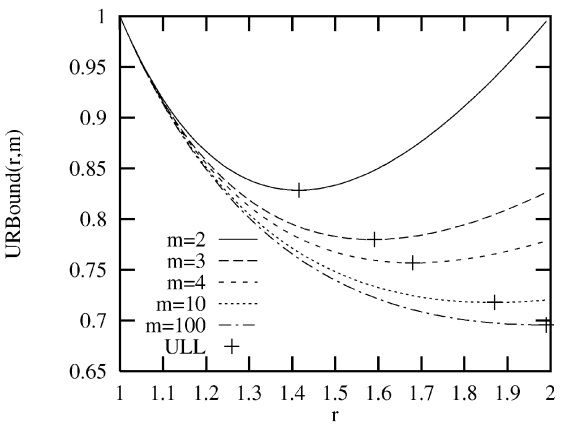

Fig. 3. Minimum utilization of RBound for different values of m.

Corollary 1. *When $m \to \infty$ the minimum achievable processor utilization $U_{RBound}(r)$ approaches $\ln r + 2/r - 1$.*

Fig. 3 shows the processor utilization U_{RBound} as a function of r. For a given m, the original RMS utilization bound U_{LL} ((1)) is the minimum of the curve $U_{RBound}(r, m)$. This implies that RBound always achieves a processor utilization better than or equal to the original RMS utilization bounds.

We can now state a necessary condition for the schedulability of a rate-monotonic task set on a uniprocessor.

Theorem 1. *Given a task set $\mathcal{T}$, let $\mathcal{T}'$ be the task set resulting from the application of the algorithm ScaleTaskSet to $\mathcal{T}$. If*

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq (m-1)(r^{1/(m-1)} - 1) + 2/r - 1, \tag{7}$$

where $r = T'_m/T'_1$, then $\mathcal{T}$ is schedulable on one processor with RMS.

Proof. For each task τ'_i $(1 \leq i < m)$ in $\mathcal{T}'$, $T'_m/2 < T'_i \leq T'_m$ and, therefore, the ratio between any two periods is less than 2. Since $\mathcal{T}'$ is also ordered by increasing periods, Lemma 3 holds and implies the schedulability of $\mathcal{T}'$. From Lemma 2, we know that the schedulability of $\mathcal{T}'$ implies the schedulability of $\mathcal{T}$. □

It should be noted that RBound is only a necessary condition for schedulability; in other words, if it is impossible to schedule $\mathcal{T}'$ with RBound, we cannot infer anything about the schedulability of $\mathcal{T}$. In order to evaluate how good of an approximation RBound is, we compare the processor utilization given by (4) with the processor utilization given by (3). We define $U_M(r, m)$ to be the maximum of $U(r, m)$ with respect to the periods. If $U_M(r, m)$ is close to $U_{RBound}(r, m)$ then (4) is a good approximation of (3) since $U(r, m)$ is between $U_{RBound}(r, m)$ and $U_M(r, m)$. Theorem 2 shows that RBound has the highest possible utilization when all tasks have a period ratio of 2 (i.e., each task has either a maximal or a minimal period).

Theorem 2. *Consider a set of m tasks ordered by increasing periods, where the minimum and maximum periods are fixed and known and the ratio r of any two periods is less than 2.*

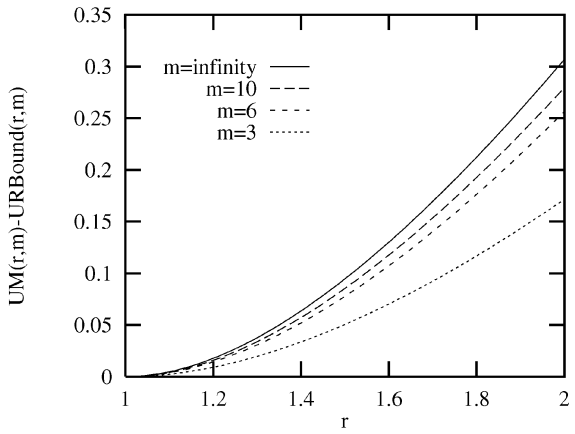

Fig. 4. Difference between upper and lower bounds for RBound.

The utilization for this task set is maximum when, for any k $(1 < k < m)$,

$$\begin{aligned} T'_i &= T'_1 \ \ (for i = 1, \ldots, k) \ \text{ and} \\ T'_j &= T'_m \ \ (for j = k+1, \ldots, m). \end{aligned} \tag{8}$$

In this case,

$$U_M(r,m) = r + 2/r - 2.$$

Proof. When the periods are given by (8), we have from (3)

$$\begin{aligned} U_M(r,m) &= \sum_{i=1}^{k-1} \left\lceil \frac{T'_1}{T'_1} \right\rceil + \frac{T'_m}{T'_1} + \sum_{j=k+1}^{m-1} \left\lceil \frac{T'_m}{T'_m} \right\rceil + 2\frac{T'_1}{T'_m} - m \\ &= r + 2/r - 2. \end{aligned} \tag{9}$$

We now show that the utilization for any task set is less or equal to (9). From (3),

$$\begin{aligned} U(r,m) &= \sum_{i=1}^{m-1} \left\lceil \frac{T'_{i+1}}{T'_i} \right\rceil + \frac{2}{r} - (m-1) - 1 \\ &= \sum_{i=1}^{m-1} \left\lceil \frac{T'_{i+1} - T'_i}{T'_i} \right\rceil + \frac{2}{r} - 1. \end{aligned} \tag{10}$$

Since the tasks are ordered by increasing periods, we have $T'_1 \leq T'_i$ $(\forall 1 \leq i \leq m)$, which implies, from (10),

$$\begin{aligned} U(r,m) &\leq \sum_{i=1}^{m-1} \left\lceil \frac{T'_{i+1} - T'_i}{T'_1} \right\rceil + \frac{2}{r} - 1 \\ &= \frac{\sum_{i=1}^{m-1} [T'_{i+1} - T'_i]}{T'_1} + \frac{2}{r} - 1 \\ &= \frac{T'_m - T'_1}{T'_1} + \frac{2}{r} - 1 \\ &= r + 2/r - 2 = U_M(r,m). \end{aligned} \tag{11}$$

□

It is worth noting from Fig. 4 that, when r is close to 1, the difference between $U_M(r,m)$ and $U_{RBound}(r,m)$ is very small, which means that the period ratio r accurately captures the information from the tasks periods.

RBound has a complexity of $O(m \log m)$ since it takes $O(m)$ to scale the tasks periods and computation times, $O(m \log m)$ to sort the tasks according to their new periods, $O(m)$ to check if the sum of the tasks utilization exceeds the bound, and $O(m)$ to map back to the original task set. This complexity is only marginally larger than the complexity of U_{LL}, which is $O(m)$.

3 Multiprocessor Admission Control

3.1 Related Work on Multiprocessor Real-Time Scheduling

Two major strategies can be used to schedule real-time tasks on a multiprocessor: *global scheduling* or *partitioning* [6]. 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. A partitioning strategy involves two algorithms: One assigns tasks to processors and the other one schedules tasks on each processor. Partitioning strategies have the advantage of reducing the multiprocessor scheduling problem to scheduling problems on individual processors for which many results are known. Furthermore, since tasks do not need to migrate across processors, partitioning usually has low scheduling overhead compared to global scheduling. Application specific constraints (such as memory requirements or dedicated input lines) can be easily integrated in a partitioning strategy by modifying the task allocation algorithm.

Most of the proposed multiprocessor algorithms with RMS priorities are based on partitioning. Global scheduling with RMS priorities suffers from the fact that it can yield an arbitrarily low processor utilization. In [6], Dhall and Liu show a set of $m+1$ tasks that cannot be scheduled on m processors with global RMS but that can be scheduled on two processors with a partitioning scheme. This task set consists of m tasks with computation 2ϵ, for some small ϵ, and period 1, and one task with computation 1 and period $1+\epsilon$. Since the m first tasks have higher priority in global RMS, they use the m processors first, causing task τ_{m+1} to miss its deadline. Not only Global RMS performs poorly in the worst case, but also its performance in the average case is mediocre [12].

Partitioning with fixed priorities is a complex problem since the utilization bound for each processor is not a constant (e.g., with RMS, it depends on the number of tasks). The first partitioning strategies with RMS scheduling were proposed by Dhall and Liu in [6]. Algorithm RMNF uses a Next-Fit bin-packing as a task assignment algorithm and RMS as a scheduling algorithm. RMNF tests if a processor can accept a new task by using the bound U_{LL} from (1). Other bin-packing algorithms have been used in conjunction with U_{LL}, including First-Fit in RMFF [6], Best-Fit in RMBF [18], and Decreasing-Utilization-First-Fit in FFDUF [5]. Because the utilization bound for RMS depends on what tasks have already been admitted, the order in which tasks are assigned to processors affects the utilization bounds. Several algorithms take advantage of the dependency between the task assignment algorithm and the scheduling algorithm. In [4], tasks are divided into classes based on their utilization. Within each class, the constraint on the tasks utilization yields a utilization bound better than

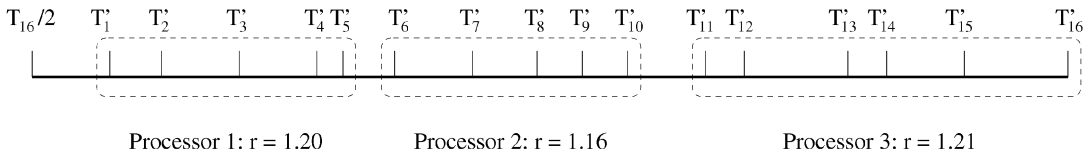

Fig. 5. Next-Fit partitioning on a scaled task set.

U_{LL}. This better bound increases the overall system utilization. Other tasks characteristics can be used to improve the schedulability bound for RMS. For example, Burchard et al. [2] present a partitioning scheme, RMGT, that uses information about the tasks periods. More recently, Han and Tyan [9] presented a $O(n^2)$ uniprocessor scheme that transforms the task periods to make the task set harmonic (and therefore the transformed task set takes full advantage of the CPU).

3.2 Partitioning with RBound

As mentioned before, a partitioning strategy involves two algorithms; one assigns tasks to processors and the other one schedules tasks on each processor. These two algorithms are not independent: Tasks should be assigned to processors such that the resulting schedule on each processor yields a high processor utilization. A property of RBound can be used to obtain this high processor utilization on a multiprocessor: As can be seen in Fig. 3, when tasks with a period ratio r close to 1 are assigned to the same processor, the resulting processor utilization is close to 1, regardless of the value of m. Therefore, the assignment of tasks to processors should be such that the period ratio r is close to 1 on each processor. In order to obtain r close to one, tasks with the closest periods in the scaled task set $\mathcal{T}'$ should be assigned to the same processor. This can easily be achieved by sorting the tasks in $\mathcal{T}'$ by increasing periods and assigning tasks to processors in a Next-Fit fashion. Fig. 5 shows that a Next-Fit partitioning on $\mathcal{T}'$ yields a small period ratio on each processor.

It is possible to further improve the performance of the task assignment algorithm by using a First-Fit packing since First-Fit searches all processors (starting from the first processor), even though some processors may have rejected a task in the past. This leads to algorithm RBound-MP given in Fig. 6. The First-Fit assignment may lead to a higher value of r, but it will not be worse than Next-Fit since the former searches a superset of the processors searched by the latter.

```
1  RBound-MP  (In: 𝒯, p)
2  begin
3      ScaleTaskSet(𝒯, 𝒯')
4      for i = 1 to m do
5          for j = 1 to p do
6              if (task τ_i can be admitted by RBound on P_j)
7                  assign τ_i to P_j
8              else
9                  if (j==p) return (REJECT)
10             endif
11         endfor
12     endfor
13 return (ACCEPT)
14 end
```

Fig. 6. Algorithm RBound-MP.

RBound-MP has a complexity of $O(m(p + \log m))$ since it takes $O(m \log m)$ to sort the task set and $O(mp)$ to assign tasks to processors.

3.3 Performance Evaluation

The performance of the proposed algorithms is measured by the average processor utilization achieved. Since there is no closed formula for the average processor utilization in multiprocessors, the performance is measured via simulation.

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. For a given U_{tot}, experiments are conducted where tasks are generated and added to the task set until the sum of their utilization exceeds U_{tot}. Once generated, a 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.

We compare the performance of several partitioning multiprocessor algorithms: RBound, RMGT [2], SRFF [9], and RMFF [6]. RMFF and SRFF use a first-fit partitioning with the utilization bound U_{LL} from (1) and the algorithm from [9], respectively. RMGT is a multiprocessor scheduling technique developed by Burchard et al. in [2]. RMGT was originally presented with a next-fit partitioning, but is used here with a first-fit partitioning to be comparable with the other algorithms.[2] We chose RMGT and SRFF as comparison algorithms for two reasons: 1) These algorithms use information about the task periods and 2) they are, to date, the most efficient partitioning schemes available for RMS tasks.

Fig. 7 shows that the performance of all partitioning schemes improve as task sets get larger since the First-Fit partitioning becomes more efficient. Furthermore, techniques developed specifically for multiprocessor

2. Indeed, we tested RMGT with first-fit and next-fit partitioning and observed that first-fit always has better performance.

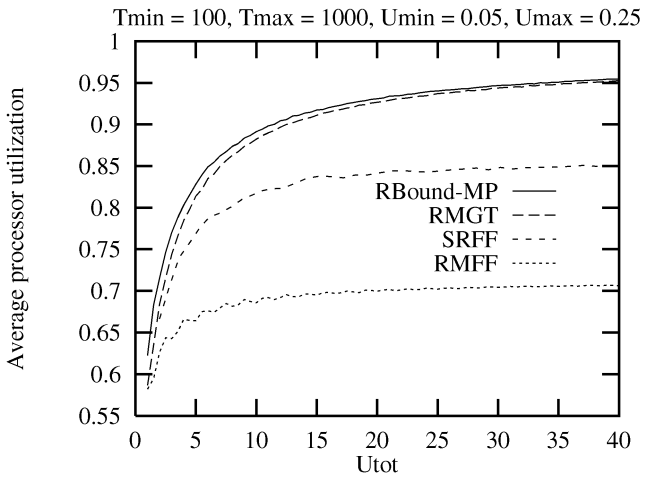

Fig. 7. Influence of the size of the task on schedulability.

environments, such as RMGT and RBound, show a sharper increase in processor utilization for large values of U_{tot}. For large total utilizations, FFSR reaches about 85 percent utilization (similar to exact characterization [13], which is not shown in the graph), whereas FFRBound and Beta are close to 95 percent. FFSR and FF-exact do not do well in multiprocessor environments because they use a bin-packing, with an overhead of approximately 15 percent loss. On the other hand, RBound and RMGT, which are designed specifically for multiprocessor environments, do much better. In the case of RBound, this sharper increase results from the fact that, as U_{tot} increases, the period ratio r on each processor decreases and thus increases $U_{RBound}(r, m)$.

Since all experimental results obtained show a similar behavior as a function of U_{tot}, we will next present only results for $U_{tot} = 4$ and $U_{tot} = 16$. Experiments were also conducted for $U_{tot} = 32$ and $U_{tot} = 64$ and gave the same types of results as for $U_{tot} = 16$ and are not presented here. The value of SRFF is always between that of RMFF and RBound and, therefore, for the sake of clarity, we do not further consider this algorithm.

Fig. 8 shows the algorithms performance as a function of U_{max}. RBound achieves the highest processor utilization, although it does not perform significantly better than RMGT for large task sets. Compared to RMFF, RBound greatly increases the processor utilization, showing that using a better uniprocessor admission control and assigning compatible tasks to the same processor can yield a high processor utilization (94 percent).

Although RBound achieves a high processor utilization, it is not optimal. Therefore, we ran another set of experiments to assess how far RBound is from optimality. These experiments compare RBound and First-Fit partitioning schemes where the exact characterization from Joseph and Pandya [11] is used to decide if a task can be added to a processor. The partitioning scheme with exact characterization has four algorithms, depending on the task set on which the admission control is performed: FFE (original task set), FFES (scaled task set), FFEO (original task ordered by increasing periods), and FFESO (scaled and ordered task set). Fig. 9 shows that RBound performs almost as well as FFESO and outperforms FFES, FFE, and FFEO. This shows that scaling and ordering the task set (as required by RBound) does improve the performance of the First-Fit partitioning. Furthermore, RBound has a lower complexity than partitioning schemes based on an exact characterization. Notice that the performance of RBound increases with the number of processors. We conducted experiments with $U_{tot} = 32$ and $U_{tot} = 64$ (not shown here), confirming that the performance of RBound improves with the number of processors.

4 Admission Control with Error Recovery

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 take into account *error recovery* in order to guarantee the timeliness of the system even in the presence of faults. This section first presents some related work on real-time scheduling with error recovery. It then extends RBound by adding provisions for timely error recovery from transient

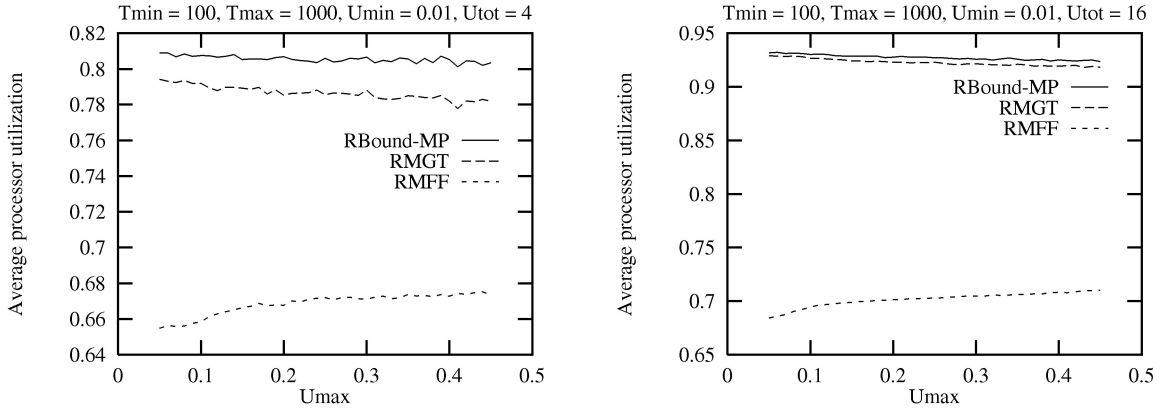

Fig. 8. Performance evaluation of RMS partitioning algorithms.

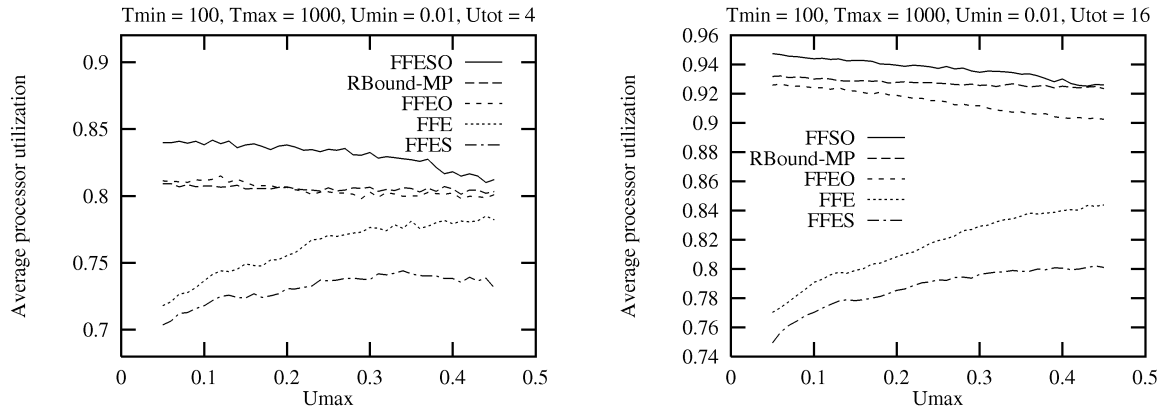

Fig. 9. Performance comparison of RBound-MP and partitioning with an exact characterization.

faults on a uniprocessor. Finally, it presents a multiprocessor real-time scheduling algorithm that provides timeliness guarantees in the presence of multiple types of faults.

4.1 Related Work on Real-Time Scheduling with Error Recovery

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). One of the most common spatial redundancies is triple modular redundancy, which executes a task on three different resources, votes upon the generated results, and achieves fault detection and masking. A less expensive spatial redundancy technique is duplication, where two copies of each task are executed, ensuring that if only one fault occurs, one of the two tasks will complete successfully. Duplication for real-time periodic tasks assumes a fault detection mechanism and has been studied by Oh and Son in [17] with the restriction that all tasks have the same period. Adding duplication for error recovery doubles the amount of resources necessary for scheduling. In [19], Oh and Son address the problem of scheduling preemptive periodic real-time tasks on a multiprocessor where tasks can have multiple versions.

Using time redundancy is problematic in real-time systems since the execution of the recovery procedure must not hinder the timeliness of the system. Ramos-Thuel and Strosnider showed how to use a "slack stealing" technique in [21], where unused computation time is reclaimed to recover from a fault. Because error recovery using slack-stealing depends on runtime characteristics, it is not always possible to guarantee that error recovery will complete by its deadline. Some 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 transient faults by reexecuting the faulty instance. If all currently executing instances must reexecute after a transient fault is detected, the admission control presented by Pandya and Malek in [20] can be used. In [3], Burns et al. describe several admission controls based on [11] for different recovery mechanisms. Note that time redundancy and space redundancy do not exclude each other. Both types of redundancy are used by Bertossi et al. in [1].

4.2 Error Recovery from Transient Faults

We first consider the problem of adding error recovery from transient faults to RBound. *A transient fault* affects only one instance of one task. We assume that, after task τ_i is affected by a transient fault, some recovery action of duration R_i has to be performed before the deadline of τ_i. This recovery model encompasses task reexecution ($R_i = C_i$), recovery blocks ($R_i = C_i'$), or restoring a safe state ($R_i = C$).

It is assumed that faults can be detected at the end of each instance [10], [25]. The time required by this fault detection mechanism can be added to the worst case computation time C_i of the task and does not hinder the timeliness of the system. We assume that transient faults are separated by a time interval Δ such that

$$\Delta \geq 2T_m, \tag{12}$$

where T_m is the largest task period. This assumption guarantees a fault-free recovery from transient faults [8].

When a fault is detected, the system enters a *recovery mode* where some recovery action must be performed before the task's deadline. This section considers two possible dispatchings during recovery mode: Rate-Monotonic Dispatching (RMD) or Slack Dispatching (SD). When RMD is used in recovery mode, each task keeps its priority based on its period and remains scheduled in the same way after the fault. When recovery is scheduled using SD, the faulty task performs its recovery by using the slack scattered in the schedule. Again, we observe a tradeoff between simplicity of dispatching and performance.

4.2.1 Using Rate-Monotonic Dispatching: RBound/RMD

In this recovery mode, tasks are scheduled according to RMS and the faulty task recovers at its RMS priority. This section gives a condition such that, if task τ_f fails, it can execute a recovery procedure of duration R_f before its deadline. We define the recovery utilization to be

$$U_R = \max_{i=1,\ldots,m} \left\{ \frac{R_i}{T_i} \right\}. \tag{13}$$

Theorem 3. *Given a task set $\mathcal{T}$ of m tasks scheduled with RMS, when one transient fault affects any task τ_f, τ_f can execute a*

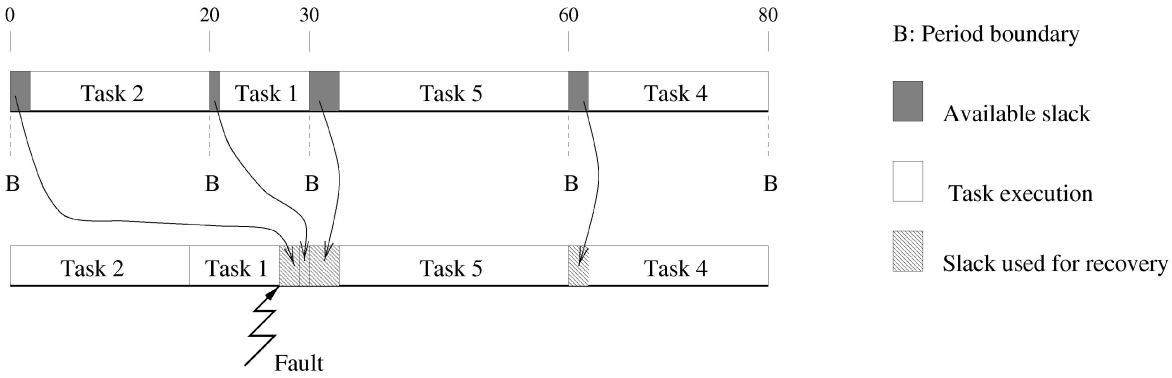

Fig. 10. Recovery with SD.

recovery action of duration R_f at its original RMS priority before its deadline if

$$\sum_{i=1}^{m}\left\lceil\frac{C_i}{T_i}\right\rceil \le U_{RBound/RMD}(r,m) = (m-1)r^{1/(m-1)} + 2/r - m - U_R, \tag{14}$$

where r is the period ratio for $\mathcal{T}$ and U_R is defined by (13).

Proof. 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 $C_f + R_f$ during the period T_f when the fault occurred. Hence, to guarantee schedulability in the presence of transient faults, it suffices to guarantee that

$$\forall f = 1,\ldots,m \quad \sum_{\substack{i=1\\ i\neq f}}^{m}\left\lceil\frac{C_i}{T_i}\right\rceil + \left\{\frac{C_f+R_f}{T_f}\right\} \le (m-1)r^{1/(m-1)} + 2/r - m \tag{15}$$

and (14) implies (15). □

Corollary 2. *When $m \to \infty$, $U_{RBound/RMD}(r)$ approaches $\ln r + 2/r - 1 - U_R$.*

4.2.2 Using Slack Dispatching: RBound/SD

It is possible to obtain a better processor utilization by reserving slack of utilization U_R in a way similar to the one presented by Ghosh et al. in [8]. This slack is distributed throughout the schedule such that, over an interval of time I between two period boundaries, the amount of slack available is $U_R I$. Note that slack dispatching differs from slack stealing in that the slack is always present in the schedule and thus is guaranteed to be available for error recovery. The top part of Fig. 10 shows a schedule where a slack of 10 percent is available between each period boundary. Before a fault is detected, the available slack is swapped with executing tasks. After a fault is detected, the swapped and available slack is used for recovery. The bottom part of Fig. 10 shows how slack is swapped with executing tasks from time 0 to 27 and is used for recovery from time 27 to 33 and 60 to 62.

In Theorem 4, we apply to RBound the same analysis technique applied in [8] to U_{LL}.

Theorem 4. *A set of m tasks with period ratio of r can be scheduled with RMS on one processor and recover from a transient fault with SD if*

$$\sum_{i=1}^{m}\frac{C_i}{T_i} \le U_{RBound/SD}(r,m) = ((m-1)(r^{1/(m-1)}-1) + 2/r - 1)(1-U_R). \tag{16}$$

Proof. The first step is to scale the original task set $\mathcal{T}$ with algorithm ScaleTaskSet. If the schedulability condition derived for the scaled task set $\mathcal{T}'$ holds, then $\mathcal{T}$ is schedulable. The analysis used in [8] shows that the processor utilization is minimum when

$$\text{for } 1 \le i \le m-1 C'_i = (T'_{i+1} - T'_i)(1-U_R) \tag{17}$$

and

$$C'_m = (2T'_1 - T'_m)(1-U_R) \tag{18}$$

(the factor $(1-U_R)$ comes from reserving slack of utilization U_R in every interval). From (17) and (18), the resulting processor utilization is

$$\sum_{i=1}^{m}\frac{C_i}{T_i} = \sum_{i=1}^{m}\frac{C'_i}{T'_i} = \left(\sum_{i=1}^{m-1}\left\lceil\frac{T'_{i+1}}{T'_i}\right\rceil + 2\frac{T'_1}{T'_m} - m\right)(1-U_R). \tag{19}$$

Following the proof of Theorem 1 with (19) instead of (3), the least upper bound for the processor utilization is (16). □

Corollary 3. *When $m \to \infty$, $U_{RBound/SD}(r)$ approaches $(\ln r + 2/r - 1)(1-U_R)$.*

A comparison with the original bound given in [8] shows that RBound/SD increases the processor utilization. By reserving slack of utilization U_R for recovery, the processor utilization $U_{G-FT-RMS}$ for a set of m tasks has been shown to be [8]

$$U_{G-FT-RMS} = m(2^{1/m} - 1)(1-U_R). \tag{20}$$

Fig. 11 shows the improvement on the processor utilization when (16) is used (curves RBound/SD) instead of (20) (curve G-FT-RMS) when $m \to \infty$. The improvement

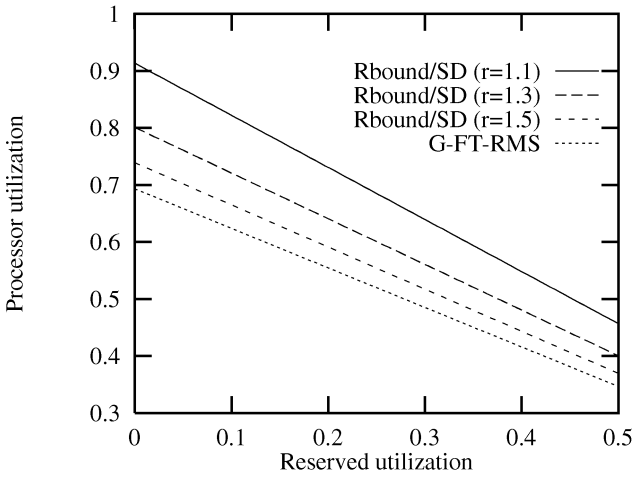

Fig. 11. Processor utilization as a function of the reserved utilization.

of the processor utilization is larger (up to 20 percent) when r is close to 1.

The average processor utilization achieved by RBound/SD and RBound/RMD is compared through simulations in Section 4.4. The following corollary shows a special case of the performance of these bounds.

Corollary 4. *When $r \to 1$, $U_{RBound/SD}$ and $U_{RBound/RMD}$ both converge to $1 - U_R$.*

RBound/SD and RBound/RMD have the same complexity as RBound since the only overhead is the computation of U_R in $O(m)$.

4.2.3 Recovery from multiple transient faults

When it is necessary to guarantee recovery from several transient faults in the time interval Δ given by (12), it is possible to increase the amount of reserved utilization U_R. When k transient faults must be tolerated within Δ, U_R is set to

$$U_R = \sum_{j=1}^{k} \left[\max_{i=1,\ldots,m}^{j} \left\{ \frac{R_i}{T_i} \right\} \right],$$

where $\max^j$ is the jth largest element of a set. Note that tolerating multiple transient faults within Δ reduces the schedulability of the task set.

Once U_R is set to the appropriate value, RBound/RMD and RBound/SD are used as described above.

4.3 Error Recovery on Multiprocessors

The work presented so far only considered error recovery from transient faults. This section extends these results to allow recovery from transient, temporary, and permanent faults. A *temporary fault* is of short duration but affects multiple tasks: Every task executing at the time at which the fault occurs is affected. For example, a temporary fault can be caused by a temporary increase in radiation levels. This increase in radiation levels causes all tasks currently executing to fail. As a consequence, several tasks may have to recover concurrently. After a temporary fault is detected, each task τ_{f_i} affected by the fault must perform a recovery action of duration R_{f_i} before its deadline. To ensure a fault-free recovery, we assume that temporary faults are separated by a time interval Δ such that (12) holds. A *permanent fault* is of long duration and affects only one processor. For example, a permanent fault can be caused by a faulty power supply. All the tasks using the faulty processor are affected by the fault. Recovery from a permanent fault requires to relocate the load of the tasks allocated to the failed processor so that, after relocation, all tasks are serviced within their deadlines.

4.3.1 Partitioning with Error Recovery from Transient Faults

To extend RBound/RMD and RBound/SD to multiprocessor systems, a partitioning scheme similar to the one from Section 3.2 is used, yielding two algorithms: RBound/RMD-MP and RBound/SD-MP. Each task is assigned to a processor according to a First-Fit bin-packing algorithm and the bounds from (14) or (16) guarantee that a task can recover from a transient fault by reexecuting on the same processor. Note that, in this case, U_R is not the same for all processors: U_R is equal to the largest task utilization among all tasks assigned to a given processor. Although RBound/RMD-MP and RBound/SD-MP use multiple processors, they do not rely on spatial redundancy to guarantee a timely error recovery.

4.3.2 Partitioning with Error Recovery from Temporary Faults

We observe that recovery from temporary faults is very similar to recovery from transient faults in that it can be achieved by each processor independently recovering its faulty task. Since we designed RBound/RMD-MP to reserve resources for error recovery on each processor, this makes RBound/RMD-MP able to tolerate temporary faults, as shown in the following theorem.

Theorem 5. *A task set admitted by RBound/RMD-MP can tolerate temporary faults if each processor independently recovers its faulty task using RMS dispatching.*

Proof. By contradiction. Assume that there is an instance of task τ_f that cannot perform a recovery action of duration R_f before its deadline. Since, in RBound/RMD-MP, processors are treated individually as in RBound/RMD after partitioning, the same instance would not recover from a transient fault either, violating Theorem 3. □

A similar result holds for RBound/SD-MP.

Note that an algorithm that recovers from a temporary fault also recovers from a transient fault, but the converse is not true. For example, an algorithm that recovers from a transient fault by performing the recovery on a spare processor cannot recover from a temporary fault. Also, recovery from a temporary fault does not guarantee recovery from multiple transient faults. For example, RBound/RMD-MP cannot recover from multiple transient faults occurring in the same processor within the fault interval Δ.

4.3.3 Partitioning with Error Recovery from Transient, Temporary and Permanent Faults

We present the results of this section based on RBound/RMD-MP, but a similar work can be based on RBound/

```
1  FT-RBOUND-MP (In: 𝒯, p)
2  begin
3  if (RBound/RMD-MP(𝒯, p) == REJECT)
4     return (REJECT)
5  for i = 1 to p do
6     if (Relocate(P_i) == REJECT) return (REJECT)
7  endfor
8  return(SUCCESS)
9  end

10 Relocate(P_i)
11 begin
12 for each task τ_j on P_i
13    for k = 1 to p and k ≠ i do
14       if (τ_j can be admitted by RBound on P_k)
15          Relocation[τ_j] = P_k
16       else
17          if (k = p) return (REJECT)
18       endif
19    endfor
20 endfor
21 end
```

Fig. 12. FT-RBound-MP.

SD-MP. If recovery from multiple transient faults must be guaranteed, a similar work can be derived based on the admission control given in Section 4.2.3.

We have established above that RBound/RMD-MP provides timeliness guarantees in the presence of transient and temporary faults. What remains to be done is to modify RBound/RMD-MP to incorporate recovery from permanent faults. One straightforward solution is to add a spare processor on which the tasks from a failed processor are relocated. However, it is often possible to avoid the addition of a spare processor by noting that each processor already has some processor utilization available for relocation. This available processor utilization comes from two sources: 1) Some processors have not reached their maximum utilization (this is particularly true of the last processor in a First-Fit partitioning scheme) and 2) each processor has some reserved utilization for recovery from transient/temporary faults. Algorithm FT-RBound-MP uses this observation so that, after a permanent fault occurs, the available processor utilization from nonfaulty processors is used to relocate tasks from the faulty processor. After relocation, each processor dispatches tasks according to RMS and recovery from faults (transient, temporary or permanent) is no longer guaranteed. The reserved processor utilization is said to be *overloaded* [7] since it is used by several instances for potential recovery.

Algorithm FT-RBound-MP (Fig. 12) proceeds in two steps. The first step (line 3) assigns tasks to processors according to algorithm RBound/RMD-MP and guarantees recovery from transient and temporary faults. The second step (lines 5 to 7) guarantees recovery from permanent faults by calling the procedure Relocate, which starts at line 10. This procedure looks for a relocation processor for each task assigned to the failed processor by using a First-Fit search with RBound as an admission criteria. Note that we do not use RBound/RMD as an admission criteria after a permanent fault is detected and therefore cannot guarantee further error recovery. However, if recovery from transient and temporary faults must be guaranteed, even after a permanent fault, it is possible to use RBound/RMD in line 14, at the expense of a lower schedulability. A relocation table is built in line 15 giving a relocation processor to each task. Creating this relocation table offline offers several advantages. First, when a permanent fault occurs, no cycles are wasted at runtime to decide where tasks should be relocated. Second, since the set of possibly relocated tasks is known for each processor, it is possible to have these tasks already in memory, further reducing the time needed for recovery. Third, the recovery is distributed since multiple processors participate in task relocation. Stankovic has shown in [23] that distributed recovery is more suitable for real-time systems than centralized recovery.

Hence, algorithm FT-RBound-MP guarantees that, within each processor, the utilization is such that:

1. In the absence of permanent faults, all tasks assigned to that processor can perform a timely recovery action in the presence of transient or temporary faults;
2. After a permanent fault, all tasks assigned to that processor and all tasks relocated to that processor meet their deadlines.

Algorithm FT-RBound-MP has a complexity of $O(m(p^2 + \log m))$ since it takes $O(m(p + \log m))$ to run RBound/RMD-MP and the function Relocate takes $O(mp)$ and is called $O(p)$ times.

4.4 Performance Evaluation

The average processor utilization achieved by the algorithms presented in this section is measured with the experimental setup described in Section 3.3.

Fig. 13 shows the performance of RBound/RMD-MP and RBound/SD-MP when faulty tasks τ_f recover by reexecuting ($R_f = C_f$). The performance of both algorithms decreases when U_{max} (defined in Section 3.3) increases since the cost of error recovery (the reserved utilization U_R) increases with U_{max}. We also observe on the right side of Fig. 13 that, when U_{tot} is large (i.e., r is small), there is very little difference in performance[3] between RBound/SD-MP and RBound/RMD-MP since they converge (Corollary 4). In order to assess the overhead of guaranteeing error recovery from transien faults, the performance of RBound-MP is also plotted. We see that the overhead of error recovery ranges from 5 percent ($U_{max} = 0.05$) to 37 percent ($U_{max} = 0.45$), which is a considerable improvement over a duplication scheme (100 percent overhead). A comparison with Fig. 8 shows that these algorithms achieve a higher processor utilization than RMFF (exepct for large values of U_{max}), although RMFF does not guarantee error recovery from transient faults.

Fig. 14 shows the performance of FT-RBound-MP. Using FT-RBound-MP rather than adding a spare processor to

3. Context switch times were ignored in these simulations, biasing the performance of the more complex slack dispatching algorithm.

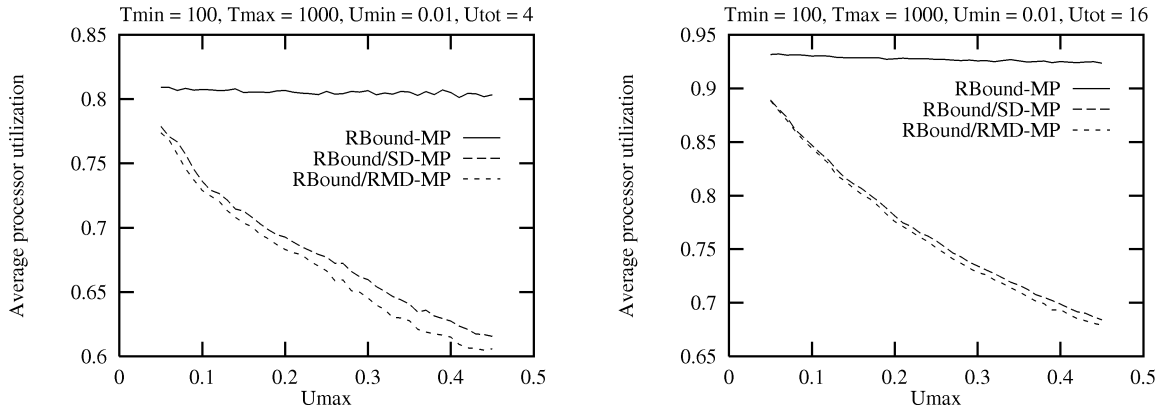

Fig. 13. Performance evaluation of partitioning algorithms with error recovery from transient faults.

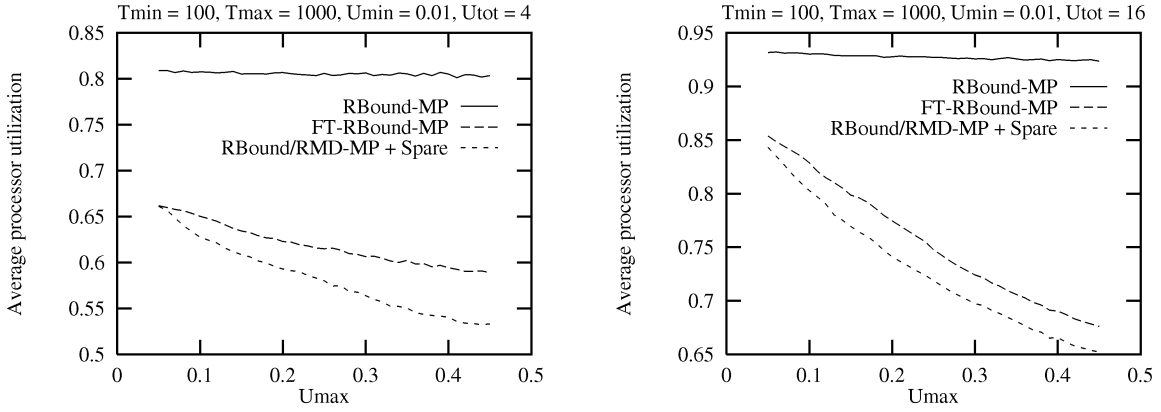

Fig. 14. Performance evaluation of partitioning algorithms with error recovery from general faults.

RBound/RMD-MP (curves RBound/RMD-MP + Spare) consistently increases the processor utilization (by as much as 20 percent). The overhead of guaranteeing error recovery from transient, temporary, and permanent faults ranges from 24 percent to 38 percent. A closer analysis of the results (not observable from Fig. 14) shows that, when $U_{tot} = 4$, FT-RBound-MP succeeds less often in finding relocation processors without adding a spare processor than when $U_{tot} = 16$ since there are fewer processors to choose from. However, when FT-RBound-MP succeeds in not adding a spare processor, the gain in processor utilization is greater when $U_{tot} = 4$. These two trends compensate each other and, as a result, FT-RBound-MP exhibits a similar behavior for different values of U_{tot}.

5 Admission Control with Aperiodic Task Servicing

Although RMS is designed to service periodic tasks, many systems require servicing for both periodic and aperiodic tasks. Several solutions have been proposed in order to guarantee the timeliness of periodic tasks while servicing aperiodic tasks quickly. We focus on two techniques that solve this problem by creating an *aperiodic server*: the *priority exchange server* [14] and the *deferrable server* [24]. For both cases, we show how RBound can be used to improve the original utilization bounds.

5.1 Periodic Server with Priority Exchange

The priority exchange technique (PE) adds to the task set an aperiodic server τ_s that services the aperiodic requests as they arrive. τ_s has the highest priority and executes when an aperiodic task arrives. When there are no aperiodic tasks to service, τ_s exchanges its priority with the task of next highest priority to allow it to execute. If the periodic server τ_s has a utilization of U_s, Lehoczky et al. [14] show that, when $m \to \infty$, the processor utilization approaches

$$U_{PE}(U_s) = U_s + \ln \frac{2}{U_s + 1}. \tag{21}$$

In the same way that RBound improves upon the utilization bound U_{LL}, we can improve upon U_{PE} by using information about the tasks periods.

Theorem 6. *A set of m tasks can be scheduled with RMS on one processor with a priority exchange server of utilization $U_s \leq 2/r - 1$ if its utilization is such that*

$$\sum_{i=1}^{m} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq U_{RBound-PE}(r, m, U_s) = U_s + (m-1)(r^{1/(m-1)} - 1) + \frac{2}{(U_s+1)r} - 1, \tag{22}$$

where r is the ratio between the largest and the smallest scaled periods.

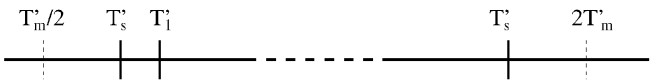

Fig. 15. Task periods after applying ScaleTaskSet.

Proof. By using algorithm ScaleTaskSet, the input task set $\mathcal{T}$ is converted into task set $\mathcal{T}'$ with a period ratio less than 2. A priority exchange server τ'_s is then added so that its period T'_s is smaller than T'_1 and larger than $T'_m/2$, as shown in Fig. 15. When scheduling the input task set after admission control, the priority exchange server is scaled down so that its period is smaller than any other period in $\mathcal{T}$.

The same analysis as in [14] shows that the computation times that minimize the processor utilization are

$$C'_s = T'_1 - T'_s \tag{23}$$

$$(\text{for } i = 1, \ldots, m-1) \;\; C'_i = T'_{i+1} - T'_i \tag{24}$$

$$C'_m = 2T'_s - T'_m \tag{25}$$

Let $r = T'_m/T'_1$ be the period ratio of the task set.

Given the computation times from (23) to (25), the resulting processor utilization bound is

$$(r, m, U_s) = U_s + \sum_{i=1}^{m-1} \left\lceil \frac{T'_{i+1}}{T'_i} \right\rceil + \frac{2T'_s}{T'_1}\frac{T'_1}{T'_m} - m. \tag{26}$$

Since, from (23), $T'_1/T'_s = U_s + 1$, (26) becomes

$$U_{RBound-PE}(r, m, U_s) = U_s + \sum_{i=1}^{m-1} \left\lceil \frac{T'_{i+1}}{T'_i} \right\rceil + \frac{2}{(U_s+1)r} - m. \tag{27}$$

The same derivation as in (4) in the proof of Lemma 3 shows that $U_{RBound-PE}$ is minimal when

$$\frac{T'_2}{T'_1} = \frac{T'_3}{T'_2} = \ldots = \frac{T'_m}{T'_{m-1}} = r^{1/(m-1)},$$

which can be used to rewrite (27) as

$$U_{RBound-PE}(r, m, U_s) = U_s + (m-1)r^{1/(m-1)} + \frac{2}{(U_s+1)r} - m \tag{28}$$

which yields (22) and completes the proof.

Note that U_s and r are not independent variables: Recall from Fig. 15 that T'_s ranges from $T'_m/2$ to T'_1. Since $U_s = T'_1/T'_s - 1$, this implies that U_s ranges from 0 to $2/r - 1$. □

Corollary 5. *When $m \to \infty$, the processor utilization $U_{RBound-PE}(r, U_s)$ approaches*

$$U_s + \ln r + 2/((U_s+1)r) - 1,$$

where $U_s \leq 2/r - 1$.

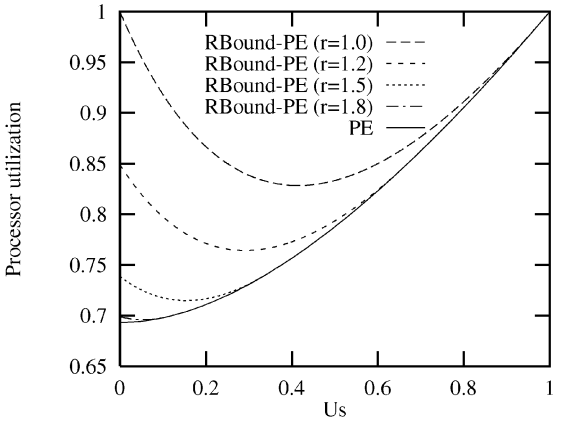

Fig. 16. Performance evaluation of RBound-PE.

Fig. 16 compares the processor utilization bounds given by $U_{PE}(U_s)$ and $U_{RBound-PE}(r, U_s)$. We observe that RBound-PE always has a processor utilization larger than or equal to PE. Furthermore, RBound-PE achieves a large processor utilization for small values of U_s, a domain in which PE yields little improvement over U_{LL}. As in the previous sections, the performance of RBound decreases as r increases and converges to the original case when $r = 2$. Note that RBound-PE merges with PE when $U_s \geq 2/r - 1$.

5.2 Aperiodic Servicing with a Deferrable Server

Another technique to service aperiodic requests is to use a *deferrable server* (DS) as proposed by Lehoczky et al. in [14]. Unlike PE, DS does not exchange its priority with other tasks, but rather holds its high priority until the end of its period. A deferrable server has a faster response time than a priority exchange server, but a lower schedulability bound:

$$U_{DS}(U_s) = U_s + \ln\frac{U_s+2}{2U_s+1}. \tag{29}$$

We can apply the same derivation technique as in Theorem 6 to improve the utilization bound for a deferrable server, based on the analysis of the deferrable server presented by Strosnider et al. in [24].

Theorem 7. *A set of m tasks can be scheduled with RMS on one processor with a deferrable server of utilization $U_s \leq (2-r)/(2r-1)$ if*

$$\sum_{i=1}^{m}\frac{C_i}{T_i} + \frac{C_s}{T_s} \leq U_{RBound-DS}(r, m, U_s) = U_s + (m-1)(r^{1/(m-1)} - 1) + \frac{U_s+2}{(2U_s+1)r} - 1, \tag{30}$$

where r is the ratio between the largest and the smallest scaled periods.

Corollary 6. *When $m \to \infty$, the processor utilization $U_{RBound-DS}(r, U_s)$ approaches*

$$U_s + \ln r + (U_s+2)/(r(2U_s+1)) - 1,$$

where $U_s \leq (2-r)/(2r-1)$.

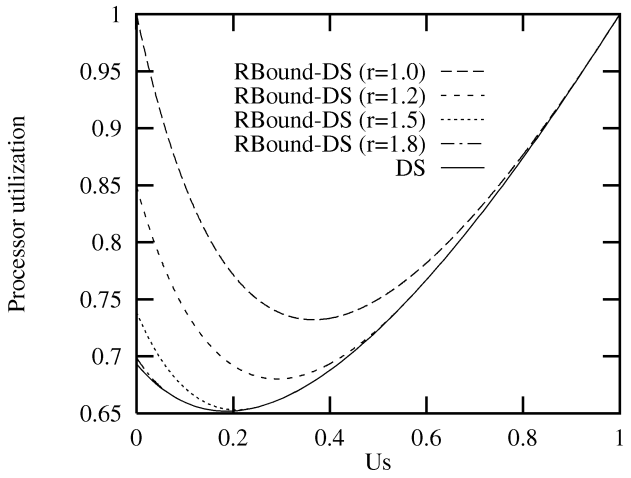

Fig. 17. Performance evaluation of RBound-DS.

Fig. 17 compares the processor utilization bounds given by $U_{DS}(U_s)$ and $U_{RBound-DS}(r, U_s)$. Once again, RBound-DS always has a processor utilization larger than or equal to DS and achieves a large processor utilization for small values of U_s. RBound-DS merges with DS when $U_s \geq (2-r)/(2r-1)$. Similarly to the comparison of DS and PE, RBound-DS achieves a lower utilization than RBound-PE.

6 Conclusion

Rate-monotonic scheduling (RMS) is arguably one of the most successful real-time scheduling techniques. While RMS has been principally developed in the context of uniprocessor systems, multiprocessor systems are becoming more common for real-time applications and, therefore, interest is growing in the area of multiprocessor real-time scheduling. As a consequence, this paper addresses the problem of efficiently scheduling tasks on a multiprocessor using RMS on each processor. We designed our solution to remain close to the original RMS uniprocessor technique so that the transition path to actual systems is smoother (only admission control procedures are changed). Further, we have shown two examples of RM extensions that can be directly applied to, and profit from, RBound. Further, the large body of work developed for uniprocessor RMS can be adapted to the multiprocessor case by using partitioning. A methodology for proving that any extension to RM can be immediately applied to RBound is underway.

This paper makes the following contributions to the field of rate-monotonic scheduling:

Uniprocessor scheduling. RBound is an admission control for RMS that uses information about the largest and smallest task periods to obtain a high processor utilization. We have shown that RBound always achieves a processor utilization larger than or equal to the original RMS admission control. Although RBound achieves a high processor utilization, its computational complexity remains low since information about the tasks periods is reduced to one variable. One of the main advantages of RBound is that its formalism remains close to the original RMS formalism and, thus, RBound can be easily applied whenever the original formalism applies (that is, task dispatching is still done via a fixed-priority scheme based on the task periods).

Multiprocessor scheduling. RBound isolates cases in which the processor utilization is close to the optimal. Based on this observation, we developed RBound-MP, a partitioning scheme based on RBound that schedules tasks on a multiprocessor in a way that preserves this quasi-optimal processor utilization. Experimental results indicate that RBound-MP achieves a high average processor utilization (over 90 percent).

Scheduling with error recovery. Multiprocessor RMS algorithms that provide timeliness guarantees in the presence of faults are presented. These algorithms are based on RBound and achieve a high processor utilization. Algorithm RBound/RMD-MP recovers from transient and temporary faults by reserving some processor utilization on each processor. When a fault is detected, a recovery action is guaranteed to be performed before the task's deadline. Algorithm FT-RBound-MP provides recovery from transient, temporary, and permanent faults by overloading the reserved processor utilization. Our performance evaluation indicates that these algorithms achieve a good processor utilization and often outperform other scheduling algorithms that do not provide error recovery.

Scheduling with aperiodic task servicing. The problem of guaranteeing the timeliness of periodic tasks while servicing aperiodic requests also benefits from RBound. We developed admission controls based on RBound for a priority exchange server and for a deferrable server. These proposed admission controls always achieve a processor utilization larger than or equal to the originally proposed admission controls for a priority exchange server and for a deferrable server.

Acknowledgments

This work was supported in part by the US Defense Advanced Research Projects Agency under contract DABT63-96-C-0044. An earlier version of this paper appeared in the International Symposium on Parallel Processing, 1998.

References

[1] A. Bertossi, A. Fusiello, and L. Mancini, "Fault-Tolerant Deadline-Monotonic Algorithm for Scheduling Hard-Real-Time Tasks," *Proc. Int'l Parallel Processing Symp.*, pp. 133-138, 1997.

[2] 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. 1429-1442, Dec. 1995.

[3] A. Burns, R. Davis, and S. Punnekkat, "Feasibility Analysis of Fault-Tolerant Real-Time Task Sets," *Proc. Euromicro Workshop Real-Time Systems*, pp. 29-33, 1996.

[4] S. Davari and S.K. Dhall, "An On Line Algorithm for Real-Time Tasks Allocation," *Proc. IEEE Real-Time Systems Symp.*, pp. 194-200, 1986.

[5] S. Davari and S.K. Dhall, "On a Periodic Real-Time Task Allocation Problem" *Proc. 19th Ann. Int'l Conf. System Sciences*, pp. 133-141, 1986.

[6] S.K. Dhall and C.L. Liu, "On a Real-Time Scheduling Problem," *Operations Research*, vol. 26, no. 1, pp. 127-140, 1978.

[7] S. Ghosh, R. Melhem, and D. Mossé, "Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm," *Proc. Fault Tolerant Computing Symp.*, 1994.
[8] S. Ghosh, D. Mossé, R. Melhem, and J. Sen Sarma, "Fault-Tolerant Rate-Monotonic Scheduling," *J. Real-Time Systems*, 1998.
[9] C.-C. Han and H.y. Tyan, "A Better Polynomial-Time Schedulability Test for Real-Time Fixed-Priority Scheduling Algorithms," *Proc. Real-Time Systems Symp.*, 1997.
[10] K.-H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers*, vol. 33, pp. 518-528, 1984.
[11] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer J.*, vol. 29, no. 5, pp. 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," *Proc. Euromicro Workshop Real-Time Systems*, pp. 188-195, 1998.
[13] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling: Exact Characterization and Average Case Behavior," *Proc. IEEE Real-Time Systems Symp.*, pp. 166-171, 1989.
[14] J. Lehoczky, L. Sha, and J.K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proc. IEEE Real-Time Systems Symp.*, pp. 261-270, 1987.
[15] J.Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, vol. 2, pp. 237-250, 1982.
[16] C.L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 47-61 1973.
[17] Y. Oh and S. Son, "An Algorithm for Real-Time Fault-Tolerant Scheduling in a Multiprocessor System," *Proc. Fourth Euromicro Workshop Real-Time Systems*, June 1992.
[18] Y. Oh and S.H. Son, "Tight Performance Bounds of Heuristics for a Real-Time Scheduling Problem," Technical Report CS-93-24, Univ. of Virginia, 1993.
[19] Y. Oh and S.H. Son, "Enhancing Fault-Tolerance in Rate-Monotonic Scheduling," *The J. Real-Time Systems*, vol. 7, no. 3, pp. 315-329, Nov. 1994.
[20] M. Pandya and M. Malek, "Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks," *IEEE Trans. Computers*, vol. 47, no. 10, pp. 1102-1112, Oct. 1998.
[21] S. Ramos-Thuel and J.K. Strosnider, "Scheduling Fault Recovery Operations for Time-Critical Applications," *Proc. Fourth IFIP Conf. Dependable Computing for Critical Applications*, Jan. 1995.
[22] K. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *Proc. IEEE*, vol. 82, no. 1, pp. 6-24, 1994.
[23] J.A. Stankovic, "Decentralized Decision Making for Tasks Reallocation in a Hard Real-Time System," *IEEE Trans. Computers*, vol. 38, no. 3, pp. 341-355, Mar. 1989.
[24] J. Strosnider, J. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Trans. Computers*, vol. 4, no. 1, pp. 73-91, Jan. 1995.
[25] Y.M. Yeh and T.Y. Feng, "Algorithm Based Fault Tolerance for Matrix Inversion with Maximum Pivoting," *J. Parallel and Distributed Computing*, vol. 14, pp. 373-389, 1992.

Sylvain Lauzac graduated from the Institut National Agronomique de Paris in 1992 and received the MS and PhD degrees in computer science from the University of Pittsburgh in 1995 and 2000, respectively. He is currently a member of the Content Delivery Services Group at Akamai Technologies. His research interests include fault-tolerant and real-time systems, load balancing, web services, and edge computing.

Rami Melhem received the BE degree in electrical engineering from Cairo University in 1976, the MA degree in mathematics and the MS degree in computer science from the University of Pittsburgh in 1981, and the PhD degree in computer science from the University of Pittsburgh in 1983. He was an assistant professor at Purdue University prior to joining the faculty of The University of Pittsburgh in 1986, where he is currently a professor of computer science and electrical engineering and the chair of the Computer Science Department. His research interest include real-time and fault-tolerant systems, optical interconnection networks, high performance computing, and parallel computer architectures. Dr. Melhem served on program committees of numerous conferences and workshops and was the general chair for the Third International Conference on Massively Parallel Processing Using Optical Interconnections. He was on the editorial board of the *IEEE Transactions on Computers* and served on the advisory boards of the IEEE technical committees on Parallel Processing and on Computer Architecture. He is the editor for the Plenum Book Series in Computer Science and is on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems* and *Computer Architecture Letters*. Dr. Melhem is a fellow of the IEEE and a member of the ACM.

Daniel Mossé received the BS degree in mathematics from the University of Brasilia in 1986 and the MS and PhD degrees in computer science from the University of Maryland in 1990 and 1993, respectively. He joined the faculty of The University of Pittsburgh in 1992, where he is currently an associate professor. His research interest include fault-tolerant and real-time systems, as well as networking. The major thrust of his research in the new millenium is power-aware computing and security. Dr. Mossé has served on program committees for all major IEEE-sponsored real-time related conferences and as program and general chair for the RTAS and RT Education Workshop. Typically funded by the US National Science Foundation and US Defense Advanced Research Projects Agency, his projects combine theoretical results and implementations. He is a member of the editorial board of the *IEEE Transactions on Computers* and is a member of the IEEE Computer Society the ACM.