In Proceedings of the First IEEE Workshop on Real-Time Applications, New York, NY, May 1993.

Probabilistic Job Scheduling for Distributed Real-time Applications

Azer Bestavros   Dimitrios Spartiotis
Computer Science Department
Boston University
Boston, MA 02215

Abstract

We describe a heuristic for dynamically scheduling time-constrained tasks in a distributed environment. When a task is submitted to a node, the scheduling software tries to schedule the task locally so as to meet its deadline. If that is not feasible, it tries to locate another node where this could be done with a high probability of success. Nodes in the system inform each other about their state (viz-a-viz the availability of free cycles) using a combination of broadcasting and gossiping. The performance of the proposed protocol is evaluated both analytically and via simulation. Based on our findings, we argue that keeping a diverse availability profile and using passive bidding (through gossiping) are both advantageous to distributed scheduling for real-time systems.

1 Introduction

Loosely coupled, time-critical distributed systems are used to control physical processes in complex applications, such as controllers in aviation systems and nuclear power plants. Most tasks in such systems have strict execution deadlines. Depending on the strictness of the execution deadline, tasks are categorized as *critical* and *essential* [15], [16]. If missing a task's deadline is catastrophic, then the task's deadline is considered to be *hard*, and the task is categorized as *critical*. The characteristics of such tasks are well known in advance, and all the resources they request are usually preallocated so that they always meet their execution deadlines. *Essential* tasks are those tasks whose deadline if not met, will not cause a catastrophe, but will result in the degradation of the system performance. Since these tasks are the results of random events (e.g., pressing of a button) their characteristics do not become known until the time they occur. Their deadlines are considered to be *soft*, and occasionally missing such deadlines can be tolerated.

In this paper, we present a decentralized algorithm for scheduling randomly submitted tasks (in the presence of other critical tasks) in a loosely coupled distributed system. We start with an overview of the system model followed by a description of the proposed heuristic. Next we present our simulation results and conclude with future research directions.

2 The System Model

We model a distributed time-critical system as a set of nodes connected via a communication network. Each node consists of two processors: one is for scheduling/executing tasks and the other is for communication. Each node is associated with a (possibly empty) set of periodic tasks, which possess hard execution deadlines. We assume that the deadline for a periodic requests is the beginning of the next period. Thus, a periodic task can be described by the pair (C_i, P_i), where C_i is the required execution time each period P_i. The characteristics of periodic tasks are known *a priori*. This enables them to be scheduled off-line during system startup.

In addition to periodic tasks, sporadic tasks with strict execution deadlines may be submitted dynamically. We describe a sporadic task by the triplet (A_j, C_j, D_j), where A_j is the time task T_j makes its request for computation, C_j is the units of execution time required, and D_j is the time the computation has to finish (deadline). The characteristics of sporadic tasks are not known a priori; they become known when submitted for execution. Upon submission, the node tries to schedule the sporadic task locally. If not successful, the task is forwarded for remote execution on a different node.

An algorithm for scheduling aperiodic tasks is composed of a *transfer policy* and a *location policy* [7]. Our transfer policy is to forward a sporadic task to another node if the amount of idle processor time until the task's deadline is less than the task computational requirements. Otherwise, the task is guaranteed execution on the node to which it was initially assigned. The task transfer decision is made dynamically and is based on the current state of the node and the characteristics of the task. The location policy dictates the way the target node is selected. This is described in the next section.

For scheduling periodic tasks on a single processor, we use the Earliest Deadline First (EDF) which is a dynamic, preemptive scheduling algorithm. For a given task set $\mathcal{T}$, with n periodic tasks, a necessary and sufficient condition for the EDF to feasibly schedule the task set, is $U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$. Since the characteristics of the periodic tasks are known a priori, we can guarantee their schedulability by simply computing the *utilization factor* U, during the system setup.

For scheduling sporadic tasks we use the results obtained in [3]. Two implementations of the EDF, respectively called EDS and EDL, are possible such that tasks are processed, respectively, as soon as possible and as late as possible. Following the notation in [3], we introduce the availability function $f_Y^X(t)$, with respect to a task set Y, scheduled according to the scheduling algorithm X in the time interval $[0, t]$, to be

$$f_Y^X(t) = \begin{cases} 1 & \text{if the processor is idle at } t \\ 0 & \text{otherwise.} \end{cases}$$

For any time instances t_1 and t_2, the integral $\Omega_Y^X(t_1, t_2) = \int_{t_1}^{t_2} f_Y^X(t)dt$ gives the total number of units of time the processor is idle in the interval $[t_1, t_2]$. For a sporadic task set $\mathcal{T}$, with D as the maximum deadline of the sporadic tasks in $\mathcal{T}$, it holds that for any instant $t \leq D$, $\Omega_{\mathcal{T}}^{EDS}(0,t) \leq \Omega_{\mathcal{T}}^X(0,t)$, and $\Omega_{\mathcal{T}}^{EDL}(0,t) \geq \Omega_{\mathcal{T}}^X(0,t)$, where X is any preemptive scheduling algorithm. So, scheduling tasks by EDL will provide us with the largest number of idle processor cycles over the interval $[0, t]$.

In order to check the schedulability of sporadic tasks we implemented an algorithm, $ACCEPT$, that utilizes the previous results. $ACCEPT$ is invoked whenever a sporadic task arrives at a node. It looks ahead in time and decides whether the sporadic task can be accepted depending on the unfinished work until the task's deadline. $ACCEPT$ runs in time linear with respect to the number of requests until the task's deadline.

3 Heuristic Algorithm Description

In order to maximize the probability that a transfered sporadic task will meet its time constraint, each node has to gather information about the load at the other nodes in the system. Our scheme does not use this information to achieve a load balanced system. On the contrary, it allows nodes to be unequally loaded so as to get a broad spectrum of available free cycles network-wide (availability profile). This promises to increase the probability of a transfer success. Because of the diversity of the availability profile, the source node (the sender of the sporadic task) might find more than one suitable node for transfer. In this case, the target node is selected probabilistically, so as to avoid overwhelming a lightly loaded node with sporadic tasks from various sources.

Dispersal of information:

The most important information a node sends out to other nodes is the localization and duration of the node's idle times and the time interval for which this information was computed. The information about idle times changes whenever a sporadic task arrives at a node and is accepted for execution. In this case, by invoking algorithm $ACCEPT$ the node is able to compute the new localization and duration of the idle times.

Changes in the workload of a node are detected by looking at the utilization factor ρ. We define three threshold values for ρ, namely ρ_L, ρ_M and ρ_H. Whenever $\rho \leq \rho_L$ the node is considered to be lightly loaded; whenever $\rho_L < \rho \leq \rho_M$ the node is considered to have a medium load; whenever $\rho_M < \rho \leq \rho_H$ the node is heavily loaded, but it can schedule all the tasks assigned to it, and whenever $\rho > \rho_H$ the node is overloaded and cannot schedule all of its tasks, so it has to transfer some of them to other nodes. When the ρ value of a node crosses one of these thresholds, the node sends out the information described previously. The use of threshold values prevent flooding the network with redundant messages.

It is obvious that a trade-off exists between the number of threshold values and the recency of the workload information. In order to alleviate this problem, we introduced a technique called *gossiping*. Whenever a node detects the communication medium idle, and a change in its work load has occured, but not a significant one to cause broadcasting, it starts talking with its neighbors. During this process, it exchanges information about its own work load and about the work load of other nodes (accordingly it receives similar information from the neighboring nodes). A node that receives information about another node (either because of load change or gossiping) checks if the information received is newer than the one already kept. If this is the case, it updates its information table. So, two nodes involved in gossiping can exchange up-to-date information about a third node, not directly involved in their conversation.

Information updating:

On every node there is a system task that regularly computes the workload on the node and stores the information in appropriate data structures. This task runs as a periodic system task with period Q_i for node i. As soon as the communication processor detects this update, and as soon as the communication channels with its neighbors are idle, it starts gossiping. In addition to the local workload information, global workload information (about other nodes in the network) is gossiped as well.

It might be the case, however, that the workload information for a node is communicated to other nodes before Q_i time units elapse. This happens whenever a new sporadic task is accepted in the time interval between two successive rounds of gossiping, or an already accepted sporadic task completes execution. Since a new task is accepted (completes) at the node, the work load on the CPU increases (decreases), in which case the new work load information is computed. If the change in the work load (which is measured by the utilization factor) crosses the threshold values for ρ then the new information is broadcast to the network. If the ρ value stays within the threshold interval, the new information is simply communicated to the node's neighbors (and, eventually, becomes known to everyone) via gossiping.

Selection of target node:

The process of selecting a target node is carried in such a way so as to maximize the probability of the transferred task being accepted. The selection is based on a prediction scheme used by the sender of the task to estimate the idle cycles (at the receiver) until the task's deadline. This estimation is based on the workload information communicated as mentioned above.

When the node has to select a target node, it does so by looking at its set of *trusted nodes*. This set includes nodes that (with high probability) will accept and schedule a transferred task. This set changes dynamically according to the workload information a node receives from the network. The trusted set is divided into two categories. The first category includes lightly loaded nodes, and the second includes nodes with a medium workload. The distinction is based on the threshold values specified earlier for ρ. It is possible for a node to move to another category or removed out of the set. From amongst the trusted nodes, the node which has the lightest load and is closest to the sender node has the highest probability of being selected as target node.

When a sporadic task arrives at a node and cannot be guaranteed execution, the node starts looking for a target node. First, it selects probabilistically either the light-load category or the medium-load category. After a category has been selected, the nodes in this category are considered. For each node in the catergory, a prediction scheme is used that approximates the number of idle cycles by the time the sporadic task arrives at the node, had the task been sent there. If the estimated idle processor cycles satisfy the task execution requirements, the node is considered a candidate target node. After the candidate target nodes have been selected, one of them is selected probabilistically as the target node, and the task is finally transferred to it. If no target node can be found, the task is kept for later re-submission.

4 Algorithm Evaluation

In this section, simulation results for the proposed algorithm are presented and compared to those obtained by other dynamic algorithms. We evaluated our protocol on a system with six nodes. The interarrival times and execution times of sporadic tasks submitted to the nodes are assumed to follow an exponential distribution, whereas their laxities are drawn from the normal distribution. We define the *laxity ratio* to be the ratio of a task's mean laxity time over its mean computation time.

To measure the *network-wide* load due to the arrival of sporadic tasks we define the demand ratio W. For a simulation of t time units, if I is the total number of idle cycles during that period on all the nodes — in the absence of any sporadic tasks — and S is the number of execution cycles requested by all the sporadic tasks occuring on every node during t, then the demand ratio is defined as $W = S/I$. Notice that this measure does not take into consideration the pattern of the arrival times of the sporadic tasks. So, even if W is less than or equal to 1.0, this does not mean that the system should be able to guarantee all the sporadic tasks that arrive, because of bursty arrivals that might have occured. In all the subsequent graphs, the X axis corresponds to the demand ratio.

To measure the performance of the algorithm, we use the *total guarantee ratio* G. Since the periodic tasks are always guaranteed, G is defined as the total number of sporadic tasks guaranteed network-wide over the total number of sporadic tasks submitted network-wide. In all the subsequent graphs, the Y axis corresponds to the guarantee ratio. Each data point in the following graphs is the average of three simulation runs.

Effect of task characteristics:

Figure 1 shows the guarantee ratio for three different sets of sporadic tasks. The laxity for all the task sets is drawn from the distribution $N(100, 50^2)$, while the task transfer delay is set to 5 time units per hop.

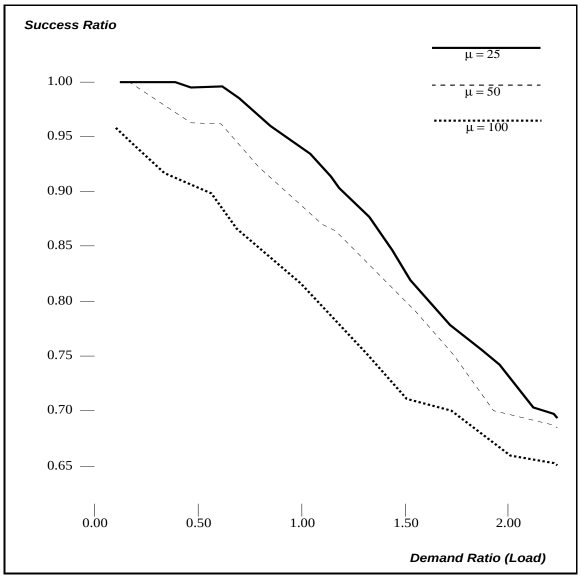

Figure 1: Effect of execution time on guarantee ratio.

For the first set, the mean execution time is 25 time units ($\mu = 0.04$) and the laxity ratio is 4. This very large laxity ratio is the reason the algorithm achieves a high guarantee ratio, even under overload conditions. For the second set, the mean execution time is 50 time units ($\mu = 0.02$) making the laxity ratio 2. Due to the larger execution times of the tasks, the guarantee ratio is not so high as in the previous case. The situation gets even worse when the execution requirements of the tasks are increased to 100 time units ($\mu = 0.01$). In this case, the laxity ratio falls down to 1. This means that a task does not get many chances for re-examination, once the first attempt to find a candidate target node fails. Also, the fact that the execution requirements are demanding, decreases the number of candidate target nodes. However, because of the probabilistic scheme, the nodes are not equally balanced, and thus the algorithm is still able to find some nodes to transfer sporadic tasks and guarantee some of them.

So, it is obvious that the task laxities play an important role on the performance of the probabilistic algorithm. Also, as it can be seen from figure 1, the guarantee ratio is not a linear function of the task parameters. Figure 2 shows the impact of the task laxities on the performance of the algorithm. Now, the mean task execution time is set to 50 time units.

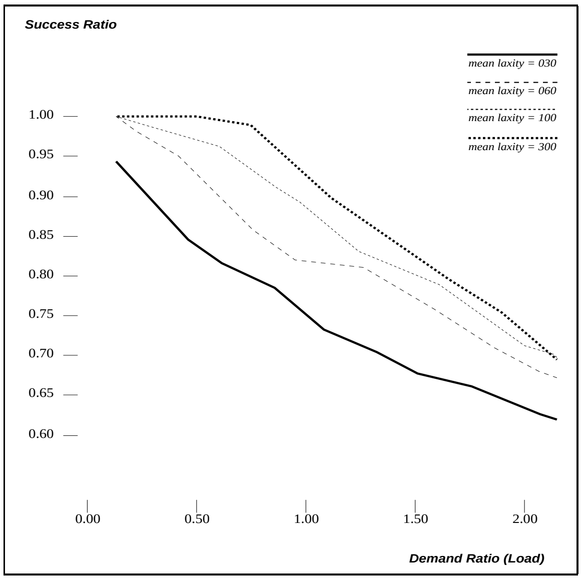

Figure 2: Effect of laxities on guarantee ratio.

We examine the following four cases:

1. The laxities are quite small with a distribution of $N(30, 15^2)$, and a laxity ratio of 0.6;
2. The laxities are moderate with a distribution of $N(60, 30^2)$, and a laxity ratio of 1.2;
3. The laxities are large with a distribution of $N(100, 50^2)$, and a laxity ratio of 2; and
4. The laxities are very large with a distribution of $N(300, 100^2)$, and a laxity ratio of 6.

Figure 2 shows that when the laxity increases the number of sporadic tasks guaranteed increases. For a moderate load of $W = 0.5$, and a laxity ratio of $0.6, G = 0.84$, while for a laxity ratio of $6, G = 1.0$. This increase in the guarantee ratio is more obvious in moderate loads. When the system becomes overloaded, this increase becomes less obvious. For example, for $W = 2.0$, increasing the laxity ratio from 0.6 to 1.2, increases the guarantee ratio from 63 % to 68 %; increasing the laxity ratio from 1.2 to 2, increases G from 68 % to 71 %, while increasing the laxity ratio from 2 to 6, increases G from 71 % to 73 % only.

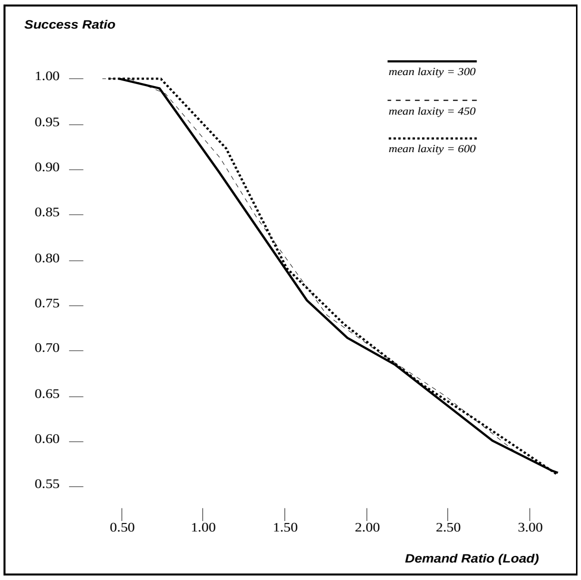

Figure 3: Effect of very large laxities on guarantee ratio.

One can also see that when the system becomes excessively overloaded, increasing the task laxity does not benefit the guarantee ratio. This is also true for medium or heavy loads. After a certain threshold value, the increase in the task laxity does not result in more sporadic tasks being guaranteed. Figure 3 shows that for $\mu = 0.02$, increasing the task laxity from $N(300, 100^2)$ to $N(450, 150^2)$, and from $N(450, 150^2)$ to $N(600, 200^2)$ does not increase the number of sporadic tasks guaranteed. The threshold value for the task laxities in this specific case is 300, thus a laxity ratio of 6.

Comparison to other algorithms:

Figure 4 shows that the performance of our algorithm is much better than that of a *no-forwarding algorithm* (NFA), and that it approaches the performance of a *delay-free algorithm*. The NFA and DFA algorithms can be thought of as defining lower and upper bounds on the attainable performance of our heuristic. Using NFA, if a sporadic task cannot be guaranteed timely execution locally, it is not forwarded. DFA, on the other hand, works exactly like our probabilistic algorithm, except that perfect information about node workloads is available at no overhead cost.

Figure 5 shows a comparison of our algorithm to an algorithm that uses a random forwarding mechanism. The task characteristics and the task communication delay are the same as before. In general, it can be seen that the probabilistic algorithm performs better than the random scheduling algorithm. This is especially true, in the cases of moderate and heavy loads. However, when the system becomes overloaded, the performance of the

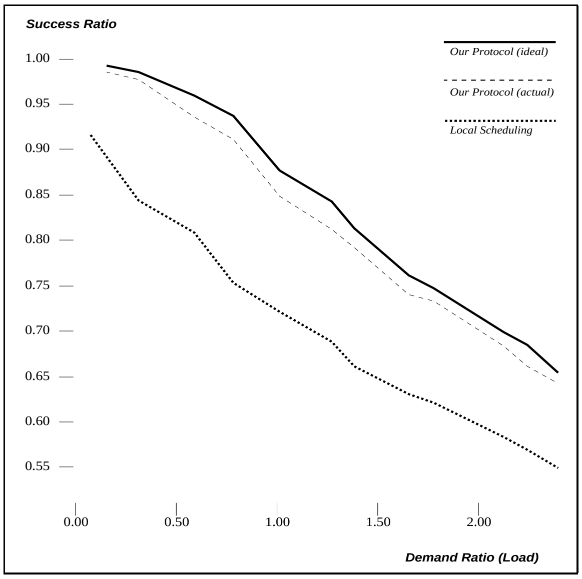

Figure 4: Lower and upper performance bounds.

two algorithms tends to coincide. This happens because nodes can no longer accept transferred tasks. So, while the probabilistic algorithm will not transfer a task, the random algorithm will transfer the task, only to have it miss its deadline remotely.

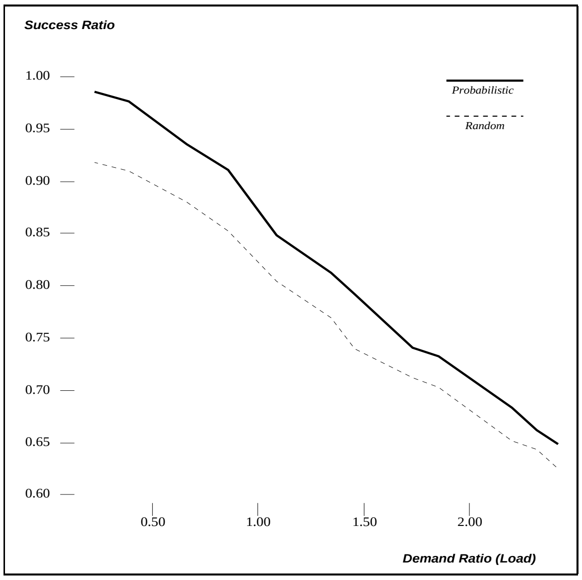

Figure 5: Comparison with random scheduling.

5 Conclusion

Dynamic scheduling of tasks with execution deadlines in a distributed time-critical environments is known to be an NP-hard problem. In this work, a heuristic approach was discussed and evaluated. The approach is a dynamic one and tries to increase the number of sporadic tasks that are accepted for execution, in the presence of critical periodic tasks that must always meet their deadlines.

The simulation results we have presented confirm the superiority of our approach. In particular, it indicates that the probabilistic algorithm performs close to the algorithm that always has up-to-date global information and forwards sporadic tasks to other nodes at no communication cost. Another conclusion drawn from the results relates the relative performances of the probabilistic and the random scheduling algorithms. Random scheduling seems to perform well, and its performance coincides with that of the probabilistic algorithm under overload conditions. Under moderate loads, the probabilistic algorithm performs better than the random algorithm, since in such conditions the proper selection of a target node is the critical issue for the overall performance of the system.

References

[1] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proc. of the Real-Time Syst. Symp.*, pp. 182-190, Lake Buena Vista, FL, December 1990. IEEE.

[2] H. Chetto and M. Chetto, "On the acceptation of non-periodic time critical tasks in distributed systems," in *Proc. 7^{th} IFAC Workshop Distributed Computer Control Systems (DCCS-86)*, Maychoss, West Germany, Oct. 30-Sept. 2, 1986.

[3] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Trans. Software Eng.*, vol. SE-15, Oct. 1989.

[4] E. G. Coffman, Jr, (editor), *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, 1976.

[5] M. Dertouzos, "Control robotics: The procedural control of physical processes," in *Proc. IFIP Congr.*, pp. 807-813, 1974.

[6] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Trans. Software Eng.*, vol SE-15, Dec. 1989.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662-675, May 1986.

[8] R. L. Graham *et al.*, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Ann. Discrete Math.*, vol. 5, pp. 287-326, 1979.

[9] J. F. Kurose and R. Chipalkatti, "Load sharing in soft real-time distributed computer systems," *IEEE Trans. Comput.*, vol. C-36, Aug. 87.

[10] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in Proc. of the 8^{th} IEEE Real-Time Systems Symposium, December 1987, pp. 261-270.

[11] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 17, no. 1, Jan. 1973.

[12] A. K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," *Ph.D. Thesis*, M.I.T., 1983.

[13] A. K. Mok and M. Dertouzos, "Multiprocessor scheduling in a hard real-time environment," in *Proc. Seventh Texas Conf. Comput. Syst.*, Nov. 1978.

[14] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," *International Conference on Distributed Computing Systems*, May-June, 1990.

[15] K. Ramamritham, J. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Trans. Comput.*, vol. C-38, Aug. 1989.

[16] B. Sprunt, L. Sha, and J. Lehoczky, "Scheduling sporadic and aperiodic events in a hard real-time system," Technical Report CMU/SEI-89-TR-11, Apr. 1989.

[17] B. Sprunt, "Aperiodic task scheduling for real-time systems," Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburg, PA, August 1990.

[18] J. A. Stankovic, K. Ramamritham, and S. Cheng , "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," *IEEE Trans. Comput.*, vol. C-34, Dec. 1985.

[19] J. K. Strosnider, "Highly responsive real-time token rings," Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburg, PA, August 1988.

[20] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.*, vol. SE-13, May 1987.

[21] W. Zhao and J. A. Stankovic, "Performance analysis of FCFS and improved FCFS scheduling algorithms for dynamic real-time computer systems," 1989.