A Markovian Performance Model for Resource Allocation Scheduling on GNU/Linux

Regiane Y. Kawasaki[1], Luiz Affonso Guedes[2], Diego L. Cardoso[1], Carlos R. L. Francês[1], Glaucio H. S. Carvalho[1], João C. W. A. Costa[1] and Nandamundi L. Vijaykumar[3]

[1] Department of Electrical and Computing Engineering, Federal University of Pará (UFPA), 66.075-900,
Belém, PA, Brazil
{kawasaki,diego,rfrances,ghsc}@ufpa.br,
[2] Department of Computing Engineering and Automation, Federal University of Rio Grande do Norte (UFRN), 59072-970,
Natal, RN, Brazil
affonso@dca.ufrn.br,
[3] National Institute for Space Research (INPE), Computing and Applied Mathematics Laboratory (LAC), P.O. Box 515, 12245-970,
São José dos Campos, SP, Brazil
vijay@lac.inpe.br

Abstract. The current paper addresses the problem of quality of service (QoS) provisioning in a general purpose operating system (GPOS), in this case GNU/Linux. Particularly, we propose to change the CPU allocation in that OS by reserving a percentage of a CPU capacity in order to ensure the QoS provisioning according to the QoS demand of each process. In order to investigate the effectiveness of that approach, Markovian models are proposed to represent the dynamics of the systems. Results show that the OS with reservation outperforms the system without it, but also that there is a performance tradeoff in the OS with reservation in such a way that an improvement in the QoS perceived by processes using the reserved capacity is done at a cost of a degradation in the QoS perceived by the other processes.

1 Introduction

As mentioned in the related literature ([1], [2], [3], [4] and [5]) several studies were made on the applicability of Markov models to investigate network technologies and their associated traffic. However, very few discuss the behavior of the front-end computers when the traffic turns into processes in a given operation system. For example, [6] presents a generic model for an Operating System (OS) scheduler for Non-Uniform Memory Access (NUMA) machines using the Stochastic Automata Networks (SAN) formalism. SAN is used to describe processes and processors in the OS and their behavior when processes have to be migrated.

Due to the necessity of investigating the feasibility of providing QoS to guarantee a minimum of resources to processes that need a differentiated treatment in a general purpose operating system, in case GNU/Linux, a performance model has been developed for the traditional GNU/Linux scheduler. It is important to point out that this model had to be developed as there are no such models available that deal with this issue. This proposal models the host's OS, that allows to verify the direct impact in the performance of distributed applications.

Besides the traditional GNU/Linux model, a different model that reserves a percentage of the processor time for providing attention to priority tasks has also been developed. By solving these models, their performance evaluation was compared to identify the changes in the system behavior due to reserving the resource. The contributions of this paper are: Markov model for Linux scheduler; model to reserve a resource (CPU) in this environment; and the analysis of the performance of these two strategies.

This paper is organized as follows. In Section 2 it describes the current process scheduler used in the GNU/Linux operating system. In Section 3 it shows the analytical model of Linux scheduler architecture and another model with CPU allocation. The performance analysis is based on a detailed mathematical model followed by numerical results that are presented in Section 4, admitted with or without resource reservation models. Finally, Section 5 shows the final remarks of this work.

2 Linux scheduler

Since version 2.5, the Linux scheduler has been called O(1) scheduler because all of its routines execute in constant time, no matter how many processors there are [7]. The current version of the Linux scheduler (kernel version 2.6.11) brought many advances. Amongst them, the possibility of allowing scheduling processes in multitasking systems such as Symmetric Multiprocessor (SMP) or NUMA [8].

The basic structure of the Linux scheduler is the process queue (*struct runqueue*). This *struct* is defined inside the archive *kernel/sched.c*. The current O(1) scheduler keeps a *runqueue* per processor, which is responsible for containing all the executable processes of a given processor. Thus, if a process is inserted in a *runqueue* of a specific processor, it will only run on that processor [6]. Each *runqueue* contains two priority arrays [7]: active and expired. Priority arrays are data structures composed of a priority bitmap and an array that contains one process queue for each priority.

The search for the higher priority is restricted to this bitmap, which uses a simple and fast algorithm called *sched_find_first_bit()*, that is, to look for the first element one ("1") within the map. When the first bit one is found ("1"), it is verified that in this row, at least one or more processes are ready for execution with that priority; then, the first one of this queue is removed and it will gain access to the processor for a given timeslice. After the execution is finished or by finalizing the task or finishing its execution time, timeslice and priority are recalculated, and it reschedules the current processes to a queue (based on

the new priority) in the expired array. Each *runqueue* has two pointers to the priority arrays. When the active array is empty, the pointers are switched: the active array passes to the expired array and vice-versa. The main advantages of this operation are: the avoidance of moving all processes to the active priority array; the execution in constant time; and keeping the scheduling algorithm with O(1) complexity.

3 Analytical model

Linux scheduler and admission control is depicted in Fig. 1. Higher and lower priority jobs arrive at the system according to two mutually independent Poisson processes with parameters λ_1 and λ_2, respectively. For the sake of simplicity, it is assumed that both services require a negative exponential service time with rate μ.

A job is removed from the active array if: (a) its processing is finished, with rate $qs_1\mu$ (for high priority jobs) and $qs_2\mu$ (for the other processes); or (b) it needs to be rescheduled to the high priority queue or low priority queue in the expired array, with rates $pr_1\mu$ or $pr_3\mu$, respectively. The scheduling of a job in the low priority queue in the active array is tied to the occupancy of the high priority queue in the active array in the sense that it will only be scheduled if the high priority queue in the active array is empty. When the processing queues are empty in an active array and there is a job to be processed in the expired array, these arrays are switched. This switching (via pointer) has an associated time of the $10^{-6}s$ [7].

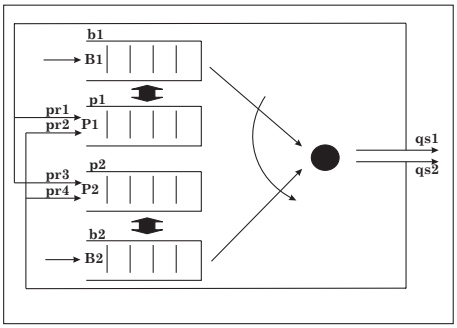


Fig. 1. System model.

Given the assumptions presented above, it a Continuous-Time Markov Chain (CTMC) [9] model of the system, whose state is defined as:

$s = (b_1, b_2, p_1, p_2, ac | 0 \leq b_1 \leq B_1; 0 \leq b_2 \leq B_2; 0 \leq p_1 \leq P_1; 0 \leq p_2 \leq P_2; ac = 0 \ \ or \ \ 1)$

Where b_1 and p_1 are the number of processes in the high priority queues; and b_2 and p_2 are the number of processes in the low priority queues; and B_i

is the buffer size of the queue i. At time, there is only one high priority queue in the active array and only one low priority queue in the active array, and the remainders are on the expired array. In order to indicate which queues are in these arrays it is used the variable ac, in such a way that if $ac = 0$, then the queues b_1 and b_2 will be in the active array and p_1 and p_2 in the expired array, and when $ac = 1$, vice-versa.

In order to evaluate the performance of the Linux scheduler, some performance measurements may be derived from the steady state probability of that CTMC. Because of the symmetry of the system only the performance measurements associated with the condition $ac = 0$ will be described, i.e., when b_1 and b_2 are in the active array, and p_1 and p_2 are in the expired array. Thus, let $p(b_1, b_2, p_1, p_2, ac)$ be the steady state probability of that Markov model, then the job blocking probability (Pbi) of a job in the queue i, it is given by the probability of its priority queue is full. Eq. (1) shows, for instance, that probability for the high priority queue in the active array. The job blocking probability for other arrays may be computed at the same way.

$$Pb_1 = \sum_{b_2=0}^{B_2} \sum_{p_1=0}^{P_1} \sum_{p_2=0}^{P_2} \pi(B_1, b_2, p_1, p_2, 0) \tag{1}$$

The mean delay of the high priority queue and the low priority queue in the active array may be computed as

$$Wb_1 = \frac{\sum_{b1=1}^{B_1} \sum_{b2=0}^{B_2} \sum_{p1=0}^{P_1} \sum_{p2=0}^{P_2} b_1 \pi(b_1, b_2, p_1, p_2, 0)}{\lambda_1 (1 - Pb_1)} \tag{2}$$

$$Wb_2 = \frac{\sum_{b1=0}^{B_1} \sum_{b2=1}^{B_2} \sum_{p1=0}^{P_1} \sum_{p2=0}^{P_2} b_2 \pi(b_1, b_2, p_1, p_2, 0)}{\lambda_2 (1 - Pb_2)} \tag{3}$$

where, Pb_2 is the job blocking probability on the low priority queue. Likewise, since, at time, only p_1 and p_2 are in the expire array, the mean delay of the high priority queue and the low priority queue may be, respectively, computed as

$$Wp_1 = \frac{\sum_{b1=0}^{B_1} \sum_{b2=0}^{B_2} \sum_{p1=1}^{P_1} \sum_{p2=0}^{P_2} p_1 \pi(b_1, b_2, p_1, p_2, 0)}{\mu(pr_1 + pr_2)(1 - Pp_1)} \tag{4}$$

$$Wp_2 = \frac{\sum_{b1=0}^{B_1} \sum_{b2=0}^{B_2} \sum_{p1=0}^{P_1} \sum_{p2=1}^{P_2} p_2 \pi(b_1, b_2, p_1, p_2, 0)}{\mu(pr_3 + pr_4)(1 - Pp_2)} \tag{5}$$

Where, Pp_1 and Pp_2 are the job blocking probability on the high and low priority queue in the expired array. The throughput of the jobs of the high priority queue and low priority queue in the active array are, respectively, given by:

$$X_1 = qs_1\mu \sum_{b_1>0}^{B_1} \sum_{b_2=0}^{B_2} \sum_{p_1=0}^{P_1} \sum_{p_2=0}^{P_2} \pi(b_1, b_2, p_1, p_2, 0) \tag{6}$$

$$X_2 = qs_2\mu \sum_{b_2>0}^{B_2} \sum_{p_1=0}^{P_1} \sum_{p_2=0}^{P_2} \pi(0, b_2, p_1, p_2, 0) \tag{7}$$

Here it is considered only jobs that finish their processing and leaving the system.

3.1 Reservation model

In this section, it is extended the model depicted previously by changing CPU allocation by means of splitting the CPU capacity into two parts: a percentage R is allocated for applications with high priority that demands QoS guarantees; and the remainder capacity $(1 - R)$ is assigned for other process. Fig. 2 shows that scheme.

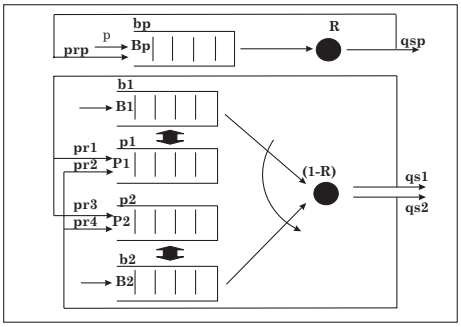


Fig. 2. Resource allocation.

The state of the CTMC of that system is defined as: $s = (b_1, b_2, p_1, p_2, bp, ac | 0 \leq b_1 \leq B_1; 0 \leq b_2 \leq B_2; 0 \leq p_1 \leq P_1; 0 \leq p_2 \leq P_2; 0 \leq b_p \leq B_p; ac = 0 or 1)$.

Where the main difference between that model and previous one consists in the random variable *bp* that represents the high priority processes, which demand QoS guarantees. Besides, the rates of the remainder processes have to be multiplied by the factor $(1 - R)$. Table 1 shows that CTMC.

Table 1. Transitions from state $s = (b_1, b_2, p_1, p_2, bp)$ to successor state t for jobs in priority policy.

Successor State	Condition	Rate	Event
$(b_1+1, b_2, p_1, p_2, bp, ac)$	$(b_1 < B_1) \wedge (ac = 0)$	λ_1	A job arrives in high priority class
$(b_1, b_2+1, p_1, p_2, bp, ac)$	$(b_2 < B_2) \wedge (ac = 0)$	λ_2	A job arrives in low priority class
$(b_1-1, b_2, p_1, p_2, bp, ac)$	$(b_1 > 0) \wedge (ac = 0)$	$qs_1(1-R)\mu$	A job from high class terminates
$(b_1-1, b_2, \theta, p_2, bp, ac)$	$(b_1 > 0) \wedge (ac = 0)$ $\begin{cases} \theta = p_1+1, \text{ if } p_1 < P_1 \\ \theta = P_1, \quad \text{if } p_1 = P_1 \end{cases}$	$pr_1(1-R)\mu$	A job is rescheduled to high priority class
$(b_1-1, b_2, p_1, \theta, bp, ac)$	$(b_1 > 0) \wedge (ac = 0)$ $\begin{cases} \theta = p_2+1, \text{ if } p_2 < P_2 \\ \theta = P_2, \quad \text{if } p_2 = P_2 \end{cases}$	$pr_3(1-R)\mu$	A job is rescheduled to low priority class
$(b_1, b_2-1, p_1, p_2, bp, ac)$	$(b_1 = 0) \wedge (b_2 > 0) \wedge (ac = 0)$	$qs_2(1-R)\mu$	A job from low class terminates
$(b_1, b_2-1, \theta, p_2, bp, ac)$	$(b_1 = 0) \wedge (b_2 > 0) \wedge (ac = 0)$ $\begin{cases} \theta = p_1+1, \text{ if } p_1 < P_1 \\ \theta = P_1, \quad \text{if } p_1 = P_1 \end{cases}$	$pr_2(1-R)\mu$	A job is rescheduled to high priority class
$(b_1, b_2-1, p_1, \theta, bp, ac)$	$(b_1 = 0) \wedge (b_2 > 0) \wedge (ac = 0)$ $\begin{cases} \theta = p_2+1, \text{ if } p_2 < P_2 \\ \theta = P_2, \quad \text{if } p_2 = P_2 \end{cases}$	$pr_4(1-R)\mu$	A job is rescheduled to low priority class
$(b_1, b_2, p_1, p_2, bp+1, ac)$	$b_p < B_p$	λ_p	A job arrives in QoS priority class
$(b_1, b_2, p_1, p_2, bp-1, ac)$	$b_p > 0$	$qspR\mu$	A job from QoS class terminates
$(b_1, b_2, p_1, p_2, bp-1, ac)$	$b_p > 0$	$prpR\mu$	A job is rescheduled, but before it is decremented
$(b_1, b_2, p_1, p_2, bp+1, ac)$	$b_p < B_p$	$prpR\mu$	A job is rescheduled, but after it is incremented
$(b_1, b_2, p_1, p_2, bp, ac+1)$	$(ac = 0) \wedge ((b_1 = 0) \wedge (b_2 = 0)) \wedge ((p_1 > 0) \vee (p_2 > 0))$	mtv	Change of arrays, b_1 and b_2 become expired
$(b_1, b_2, p_1, p_2, bp, ac-1)$	$(ac = 1) \wedge ((p_1 = 0) \wedge (p_2 = 0)) \wedge ((b_1 > 0) \vee (b_2 > 0))$	mtv	Change of arrays, b_1 and b_2 become active

Transitions from state s to all possible successor states are reported in Table 1 along with their rates and conditions under which the transitions exist; the last column indicates the type of event to which a transition refers. When $ac = 0$, if a job is generated in the high priority queue in the active array, the occupancy of that queue, b_1, will increase by one unit. A rescheduled job from that queue will go to the high priority queue in the expired array with rate $pr_1(1-R)$ or to the low priority queue in the expired array with rate $pr_3(1-R)$. In the first case the job keeps the same priority and, in the latter, the priority is decreased. A job can leave the high priority queue in the active array, after finishing its processing with rate $qs_1(1-R)$. An arrival in the low priority queue in the active array takes place with rate and increases b_2 by one unit.

Since the system under analysis is finite, when a buffer (active or expired arrays) is full an incoming or rescheduled job is blocked. After switching, the queues that were in the expired array (p_1 and p_2) become active and vice-versa. The variable bp represents QoS jobs. We assumed that $mtv = 10^{-6}$. The system is symmetric, which makes quite natural the match of the other transitions of the model.

Due to the lack of space and for simplicity only performance measurements of the high priority jobs that demand QoS guarantees are presented.

$$W_{pb} = \frac{\sum_{b_1=0}^{B_1}\sum_{b_2=0}^{B_2}\sum_{p_1=0}^{P_1}\sum_{p_2=0}^{P_2}\sum_{b_p=1}^{B_P} b_p\pi(b_1,b_2,p_1,p_2,b_p,0)}{(\lambda_p + prpR\mu)(1-Pb_p)} \tag{8}$$

Where Pb_p is blocking probability of high priority jobs that demand QoS guarantees derived as Eq.(1). The throughput is given by:

$$X_{pb} = qspR\mu \sum_{b_1=0}^{B_1}\sum_{b_2=0}^{B_2}\sum_{p_1=0}^{P_1}\sum_{p_2=0}^{P_2}\sum_{b_p>0}^{B_P} \pi(b_1,b_2,p_1,p_2,b_p,0) \tag{9}$$

4 Performance study

In this section some numerical results are presented to evaluate how adequate is the Markov model to scheduling GNU/Linux with and without resource allocation policy. First, we present the performance of the Linux Markovian model. For validation purpose, Linux scheduler was simulated by using an academic version of a powerful tool named ARENA©[10]. Some measures were obtained through system calls which collect data for later analysis, minimizing the overhead in kernel (this can be obtained in www.lprad.ufpa.br/parqos). Table 2 summarizes the parameters used.

To validate the probability distributions adopted, models use input data obtained from the real system. In these data, Kolmogorov-Smirinov (K-S) goodness of fit tests were applied, using the trial version of BestFit©tool [11].

These data were used as parameters of probability distributions in question (Poisson for inter-arrivals times). The simulation results were collated with the

performance measures obtained from the real system. As the numerical results of that comparison match (very similar), the values may be considered validated for the analytical model.

Table 2. Input data.

High Priority	Measures	Low Priority:	Measures
λ_1	7	λ_2	7,3
pr_1	0,1	pr_2	0
pr_3	0,09	pr_4	0,67
Avarage Buffer	5	Avarage Buffer	5

To implement CPU allocation policy it is important to study the CPU behavior. Assuming the table above, it represents a situation where scheduler is very busy and the inputs are Poisson traffic. A new application is added in λ_1, simulating a situation of great workload. Table 3 ilustrates the Markovian model output. As expected, higher the traffic load, bigger the throughput (Fig. 3.a) and, for that reason, longer the mean waiting time, longer is the blocking probability. In the table, 0% represents the system behavior performance with just λ_1 and λ_2. λ_p is derived from λ_1(5%, 10%, 20%, 30%, 40%) and represents the impact of adding an application to the system.

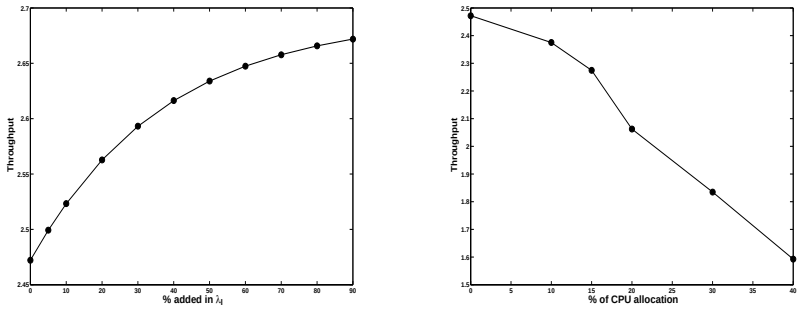
Table 3. Performance measurements.

Queue Waiting Time						
	0%	5%	10%	20%	30%	40%
Active High Priority	0,21246	0,21491	0,21687	0,21943	0,22037	0,21995
Active Low Priority	0,59056	0,60058	0,60970	0,62533	0,63785	0,64774
Expired High Priority	6,05108	6,28370	6,48294	6,79442	7,01313	7,16437
Expired Low Priority	0,85739	0,87383	0,88871	0,91386	0,93334	0,94802
Blocking Probability						
	0%	5%	10%	20%	30%	40%
Active High Priority	0,09701	0,10763	0,11827	0,13925	0,15940	0,17838
Active Low Priority	0,44159	0,44832	0,45431	0,46434	0,47214	0,47817
Expired High Priority	0,43127	0,44343	0,45346	0,46847	0,47854	0,48530
Expired Low Priority	0,45635	0,46216	0,46734	0,47591	0,48241	0,48723

The main objective of modeling kernel scheduling processes is its evaluation and study for future modifications in order to achieve a better performance for QoS applications. The extended model (resource Allocation) is being studied and tested, but it already presents interesting results like the ones in Figs 3.b and 4. It is the same test situation previously used, but with a different scheduling

process. Using CPU reservation of 40% means that 40% of process capacity is allocated for QoS applications and 60% for the rest of the applications.

Fig. 4 shows the normal system scheduler improvement by limiting the percentage of CPU (100%, 95%, 90%, 80%, 70%, 60%), and it can be observed that, smaller the percentage of CPU used, smaller the system throughput for both priority applications. For QoS application, however, bigger the percentage of CPU used (0%, 5%, 10%, 20%, 30%, 40%), bigger the system throughput.



(a) (b)

Fig. 3. (a) Throughput behavior (b)Throughput behavior (QoS allocation policy).

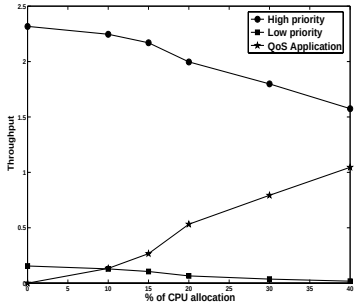


Fig. 4. Throughput behavior per flow.

5 Final remarks

Provision of QoS guarantees in a GPOS scheduler is an open problem. This paper, presented and proposed a Markovian Linux scheduler model for performance

study and, in addition, an extended model, which uses static CPU allocation policy. Through the analysis of the results, it has been concluded that the performance of the applications with QoS are greatly improved in its throughput. However, other applications have suffered some limitations. The contributions of this paper are: (1) Proposal of performance models for GPOS (GNU/Linux) scheduler; (2) Proposal of a resource (CPU) allocation scheme in an environment that needs QoS as well as showing through numerical results, obtained from its Markovian model, improvement of performance of QoS applications when compared to other applications.

Currently, we are implementing another extended model which uses dynamic CPU allocation policy. As future work, we performing experiments with Markov decision process to find optimal admission control and scheduling strategies aiming at improving the resource (CPU and memory) allocation.

This work is supported by CNPq and CAPES. Thanks to Dr. Solon Carvalho for giving permission to use the Stochastic Modeling Software (MODESTO) [12].

References

1. Leong, C. W., Zhuang, W., Cheng, Y., Wang, L.: Optimal Resource Allocation and Adaptive Call Admission Control for Voice/Data Integrated Cellular Networks, IEEE Transactions on Vehicular Technology, Vol. **55**, No. 2, March (2006), pp.654-669.
2. Zimmermann, M., Dostert, K.: Analysis and Modeling of Impulsive Noise in Broad-Band Powerline Communications,IEEE Transactions on Electromagnetic Compatibility, Vol. **44**, No. 1, February (2002), pp. 249-258.
3. Yu, J. Y., Chong, P. H. J., So, P. L., Gunawan, E.: Solutions for the Silent Node Problem in Automatic Meter Reading System Using Powerline Communications, in Proc. of the 7th International Power Engineering Conference, IPEC 2005, Nov. (2005).
4. Yuang, M. C., Po-Lung Tien, Shih, J., Chen, A.: QoS Scheduler/Shaper for Optical Coarse Packet Switching IP-Over-WDM Networks, IEEE Journal on Selected Areas in Communications, Vol. **22**, No. 9, Nov. (2004), pp.1766-1780.
5. Levey, D. B., McLaughlin, S.: Calculating Error-Free Seconds in xDSL Systems Corrupted by Impulse Noise, IEEE Communications Letters, Vol. **5**, No. 7, July (2001), pp. 319-321.
6. Chanin, R., Corrêa, M., Fernandes, P., Sales, A., Scheer, R., Zorzo, A.F.: Analytical Modeling for Operating System Schedulers on NUMA Systems, in Proc. of the 2nd International Workshop on Practical Applications of Stochastic Modelling, PASM05, University of Newcastle upon Tyne, UK, July (2005).
7. Love, R.: Linux Kernel Development, SAMS, 1st edn., (2003).
8. Hwang, K., Xu, Z.: Scalable Parallel Computing - Technology, Architecture and Programming, WCB/ McGraw-Hill, (1998).
9. Wei, W., Wang, B., Towsley, D.: Continuous-Time Hidden Markov Models for Network Performance Evaluation, Performance Evaluation, Vol.**49**, (2002), pp. 129-146.
10. Rockwell Automatation - www.arenasimulation.com, accessed in 02/15/2006.
11. Palisade - www.palisade.com/bestfit, accessed in 02/18/2006.
12. Frances, C.R., Oliveira, E., Costa, J., Santana, M., Santana, R., Bruschi, S., Vijaykumar, N., Carvalho, S.: Performance Evaluation Based on System Modelling Using Statecharts Extensions, Simulation Practice and Theory, Vol. **13**, n. 7, (2005), pp. 584-618.