Statistical quality analysis of schedulers under soft-real-time constraints

Hilbrandt Baarsma

Pierre Jansen, Johann Hurink
University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
{h.e.baarsma,p.g.jansen,j.hurink}@utwente.nl

April 4, 2007

Abstract

This paper describes an algorithm to determine the performance of real-time systems with tasks using stochastic processing times. Such an algorithm can be used for guaranteeing Quality of Service of periodic tasks with soft real-time constraints. We use a discrete distribution model of processing times instead of worst case times like in hard real-time systems. Such a model gives a more realistic view on the actual requirements of the system. The presented algorithm works for all deterministic scheduling systems, which makes it more general than existing algorithms and allows us to compare performance between these systems. To demonstrate our method, we make a comparison between the performance of the well known scheduling algorithms Earliest Deadline First and Rate Monotonic. We show that the complexity of our method can compete with other algorithms that work for a wide range of schedulers.

Keywords: Scheduling, Soft-Real-Time, Stochastic, Dynamic Programming.
AMS classification: 68M20

1 Introduction

Many modern devices have to be able to process streams of data. These streams often consist of tasks arriving at regular intervals, where each task has to be processed within a fixed real time interval (RT). With the increasing use of RT systems, the techniques for building these systems have been described in many papers. In particular, scheduling techniques for single processor systems have been studied extensively for optimality [7] [14] [13] [9], for feasibility conditions [3] [2] [13] [9], and for worst case response [15]. In this paper we focus on single processor systems, but it is likely that the underlying ideas of these techniques can also be applied in a distributed RT environment [14] [11].

If several streams have to be processed on a single processor, there is a risk that some tasks may not complete before their deadlines. In hard real-time systems, this is not acceptable. However, with the increase of all kinds of multimedia devices, where occasionally missing a deadline can be tolerated, soft deadlines become more acceptable. In this context it is of importance to know how many deadlines are missed. Finding the number of missed deadlines is easy when dealing with deterministic processing times, but we run into problems when processing times are stochastic. Tasks might for example have a certain amount of jitter on their processing times. We still might use the deterministic methods to check just the worst case scenarios. However, using the worst cases, may lead to an oversized system that, in reality, is idle most of the time. Therefore, if missing a deadline occasionally is acceptable, it becomes an important question how we can scale our system to achieve the performance level that we tolerate, in terms of missed deadlines.

There exist several scheduling methods that can deal with periodic tasks and try to prevent deadlines being missed (see e.g. [13]). Although fast algorithms exist to check if a given scheduling method will result in a deadline being missed, there are few efficient algorithms [10] to calculate the expected number of missed deadlines. This paper describes such an algorithm with three important features:

1. It can take into account the effect of variation in

processing times.

2. It can compare hard vs. soft-real-time.

3. It can compare different scheduling techniques.

Using this algorithm, we can evaluate and compare many different scenarios. We can use most priority driven scheduling methods like EDF (Earliest Deadline First), FIFO (first in first out), DM [13] (Deadline Monotonic), RM (Rate Monotonic), and timeline priority driven scheduling. We can also either enable or disable preemption at will. In the following sections we demonstrate how these features are realized and how they can be used.
The remainder of this paper has the following structure. We start with introducing related work in Section 2. Section 3 deals with the model we use and the assumptions we make. In Section 4 we explain an algorithm that checks schedulability. The complexity of this algorithm is discussed in Section 5. Some simulations have been run with this algorithm and the results are detailed in Section 6. The paper ends with some concluding remarks.

2 Related Work

Many computer systems have to deal with periodic tasks. Tasks like this can consist of things like audio or video streams, sensor data, control signals or radio signals. Usually periodic tasks have deadlines associated with them, specifying the time a task should be ready. This introduces the so-called deadline driven scheduling methods, like EDF, other scheduling methods work with static priorities, like RM. Distinctions between real-time paradigms can be made based on the desired behavior of the system. The most common is hard real-time, where all deadlines have to be met. Systems in which occasionally missing a deadline is acceptable are classified as soft real-time. Unfortunately this classification is not very specific. It does not mention how many deadlines can be missed, or what happens if a task misses its deadline. Alternative classifications are described in [4], [5] and [1] where we encounter terms like weakly-hard real-time, probabilistic hard real-time and firm real-time. Some papers also use alternative definitions for "soft real-time". Most of these definitions allow for some deadlines to be missed, but want to maintain a certain service level. This is exactly the area where our algorithm can be used to calculate performance.
Many studies in this area look at approximations of the performance in some way, by using (worst-case) estimates. This includes for example Probabilistic Time Demand Analysis (PTDA) [16] and Stochastic Time Demand Analysis (STDA) [8]. STDA only works with fixed priority schedulers, however, and focusses on finding a lower bound on the number of missed deadlines, while PDTA is even more limited.
In [10] a method for obtaining an exact performance measure of a periodic real-time system is introduced. It allows for all priority-driven scheduling methods and assumes a discrete distribution of processing times, similar to our method. The main difference is in the complexity of the algorithms, as we will explained in more detail in Section 5. An advantage of our algorithm is that it is able to handle non preemptive scheduling and tasks with dynamic priorities.

3 Problem Definition

We consider a task set Ω with n different periodic tasks. Each task $\tau_i \in \Omega$ has a period T_i, a phase ϕ_i and a deadline interval D_i. The period T_i denotes the time distance between two consecutive occurrences of task τ_i. We assume each task will be released at the start of its period and has to be finished within $D_i \leq T_i$ time units. Furthermore, the processing time C_i of task τ_i is a stochastic variable with distribution F_i. The release times and deadlines of the occurrences of the periodic tasks are determined by the first release of the task and, thus, can be calculated in advance. The processing time C_i^k of the kth occurrence of τ_i is a realization of the of the distribution F_i and is assumed to be independent of other occurrences of this task. Let r_i^k (d_i^k) be the release time (deadline) of the kth occurrence of task τ_i. For reasons of simplicity we often use r_i (d_i) to denote the release time (deadline) of the current occurrence of τ_i. In this paper we assume that if a task still needs processing time when it reaches its deadline, that load will be discarded. This is, however, not a hard restriction on our algorithm, but it does help to reduce processing times by reducing the number of states.
The approach presented in this paper assumes the number of possible realizations for F_i to be finite. Although this a restriction, in practice this assumption can be made most of the time. One reason is that time often is discretized, but, furthermore , in many applications a given process can be characterized in sufficient detail by only a few different processing times. The phase ϕ_i of a task determines how the release moments of two different tasks are related to each other. If all tasks have $\phi = 0$, then all tasks are released at $t = 0$. For every k and i we have the property $r_i^k \bmod T_i = \phi_i$. In this paper we are mainly concerned with the case where all phases are zero, since this is

usually (close to) the worst case.
The task set Ω now has to be performed by a single processor. For this we assume a scheduling method SM to be given. This scheduling method SM may be any priority driven scheduling approach (e.g. EDF [13] scheduling, a widely accepted form of deadline driven scheduling) but more complex methods are possible as well. Furthermore, SM can include the use of preemption if necessary.
The problem now is to develop an algorithm which is able to evaluate the performance of SM on a given task set Ω.The performance is measured as the estimated number of missed deadlines.

4 A performance analyzer

This section describes the proposed algorithm for solving the problem stated in the previous section. First we give some examples of technical problems faced and then we present the ideas of the proposed algorithm.
When looking at stochastic processing times, there are some hurdles when calculating the probability that a deadline will be missed. For example, suppose we have two tasks, τ_1 and τ_2, that are both released at time 0 and τ_1 has a smaller deadline ($d_1 < d_2$). The probability that τ_2 will meet its deadline is not simply $P(C_1+C_2 \leq d_2)$. The problem is that τ_1 might already have missed its deadline. Since it would stop processing at its deadline, τ_1 would not take C_1 but d_1 time. Hence the formula should be $P(C_1 \leq d_1) * P(C_1+C_2 \leq d_2) + P(C_1 \geq d_1) * P(d_1 + C_2 \leq d_2)$.
The situation is even more complicated if τ_2 arrives in the system (at r_2) later than τ_1 but before d_1, i.e. $r_1 < r_2 < d_1$; see Figure 1. In this figure, arrows pointing up represent releases, whereas the arrows pointing down are deadlines and the grey areas depict the processing times of a task. We assume that $r_1 = 0$.

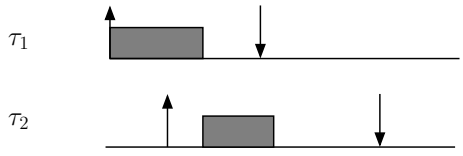

Figure 1: Example

In this case we have a certain probability that $C_1 < r_2$, leaving the processor idle for some time. Thus, adding C_1 and C_2 does not correctly describe the completion time of τ_2. Instead, we get the following more complex equation for this probability:

$$\begin{aligned} P(C_1 < r_2) &* P(r_2 + C_2 \leq d_2) \\ +P(r_2 \leq C_1 < d_1) &* P(C_1 + C_2 \leq d_2) \\ +P(C_1 \geq d_1) &* P(d_1 + C_2 \leq d_2) \end{aligned}$$

The equation is split up into three conditions. Now this is only a small model with two tasks. The equation may keep splitting with each deadline or arrival event. Although a few of these conditions might at some point be joined, the rate at which the number of conditions expands will generally grow very fast, making this type equations unmanageable.
If we use a discrete distribution, with a limited number of possible realizations for the processing times of the tasks, we can avoid some of the problems that burden the continuous case. This can be done by using dynamic programming to join the "split ups". In the given model, the system is at each time in some state with a certain probability. Let $S_t = \{s_1^t, s_2^t, \ldots s_{n_t}^t\}$ be the set of all possible states at time t. Each state $s \in S_t$ contains a list Q_s and the probability p_s of being in this state. The list Q_s consists of tasks that still need to be processed, paired with their remaining processing time. If we do not allow preemption, then the state needs two extra variables, containing the task currently being scheduled and its remaining processing time.
To be able to evaluate how a scheduling method SM processes a task set Ω in a certain time period, we have to adjust every state, to reflect how the system in that state would have evolved when scheduled by SM. For adjusting the states, the relevant times to be considered are when a new task enters the system or when a task reaches its deadline. In between two consecutive event times, t_i and t_{i-1}, no new states emerge. Only the values of the remaining processing times within the states change, according to the scheduler SM. Thus, the relevant times are given by the multiset $T = \{t_1, t_2, t_3, \ldots\}$ of all deadline and release times. This multiset has the property that $t_1 \leq t_2 \leq t_3 \leq \ldots$ and if times are equal, the deadline events occur before the release events. We define a function $F_{SM}(s_j^{t_i}, t_i, t_{i+1}) s_j^{t_i} \rightarrow s_j^{t_{i+1}}$, that describes the change of the state $s_j^{t_i} \in S_{t_i}$ within the time interval $[t_i, t_{i+1}]$ due to SM. The resulting state $F_{SM}(s_j^{t_i}, t_i, t_{i+1})$ belongs to $S_{t_{i+1}}$. Note, that the used function F depends on the scheduler used. If the set $S_t = \{s_1^t, s_2^t, \ldots s_k^t\}$ consists of all states at time t, we obtain $S_{t_{i+1}}$ by applying F_{SM} to all s_j's from S_{t_i}. The first thing F_{SM} does is to calculate $t_{i+1} - t_i$ and, if this is not zero, it updates the state according to SM. This means that some tasks in the new state have less load than they had in $s_j^{t_{i-1}}$. If the event at

t_i is a deadline of a task τ_j, F_{SM} checks if τ_j has any load left. If this is the case, task τ_j has missed its deadline, thus, the task is removed from Q_s and p_s is added to the expected number of errors. If F_{SM} has been applied to all states in S_{t_i}, we check if there are any identical states in the resulting set $S_{t_{i+1}}$. If this is the case, we merge them, i.e. if the states s_1 and s_2 have $Q_{s_1} = Q_{s_2}$, we set p_{s_1} to $p_{s_1} + p_{s_2}$ and then delete s_2 from S. Two different states can end up being identical for several reasons. For example, during $[t_i, t_{i+1}]$ both states end up being idle. How they reached this point is no longer important now.

If the event at t_i is a release event of τ_j, then the output of F_{SM} is not one state, but several states. Each output state corresponds to a different realization of the processing time C_j of τ_j and the sum of the probabilities corresponding to these new states is equal to the probability of the original state. What F_{SM} does in this case is basically a convolution of the original states with the possible processing times of the released task.
Example: *Consider a system with tasks τ_1 and τ_2 with $r_1 = 0$, $r_2 = 1$ and $d_1 = 3$. Let both C_1 and C_2 be either 1 or 2 with probability 0.5. The list of states S starts with one state that does not contain any load. At r_1 this state splits up into two new states. Q_{s_1} has processing time of 1 left for τ_1, while in Q_{s_2} it has residual processing time 2. Both p_1 and p_2 are 0.5. The next event is at r_2. We note that one unit of time has passed and adjust the states accordingly. This results in Q_{s_1} being empty and Q_{s_2} having processing time 1 for τ_1. At this point we release the new task, splitting off the existing states. Now we will have four different states each with probability 0.25. Now the next event is d_1. We now have some states with residual processing time for more than one task. In this case it depends on which scheduling algorithm we use which task gets priority. Supposing we use EDF, we obtain 3 states with no load left and one state with processing time 1 left for τ_2. We process the deadline of τ_1, but since none of the states have any load left for τ_1 we conclude no deadlines have been missed. Now we check for identical states and see we can reduce the number of states to two. One state has a 0.75 probability and has no load left. The other state has a 0.25 probability and has to run τ_2 for one more time unit. The process we just described can be reviewed in Table 1*

Using the afore described process, we can evaluate the schedulability of the tasks by moving along the time-axis and at each deadline or release of a task adjusting all states and their probabilities. What remains, is to determine a suitable time interval over which the system is evaluated. If we consider hard real-time, the highest probability for an error to occur is if all tasks arrive at the same time, resulting in the maximum amount of load at one time. If we start by releasing all tasks at the same time and this always results in an idle system after some time with the expected numbers of missed deadlines being zero, we can stop our checking and conclude that we can schedule the task set without missing any deadlines. In all other cases, the task set cannot be scheduled safely. In the case of SRT, the situation is quite different. If we start again with a state where the load is maximal, we cannot stop the evaluation if the system is idle and take the estimated number of missed deadlines as final, unless this number is zero. The reason for this is that it is still possible to miss deadlines later on, although the probability is lower since we won't have a period with as much load as at the start. As a consequence of the above considerations, the algorithm should check a time interval equal to the least common multiple of the periods of all the tasks. At this point, the system is in the same state as it was at the start. There should not be any tasks left in our system at this point, since all our tasks have deadlines shorter than their periods.
Another problem is that if more than one deadline can be missed, the scenario where all tasks arrive at time 0 is not necessarily the worst case. This means we have to check all possible phase shifts. The maximum number of unique phase shifts depends on the greatest common divisors of the tasks. If t_0 has the longest period and n is the number of tasks then $\prod_{i=1}^{i=n} GCD(T_0, T_i)$ is the total number of unique phase shifts. Checking

event	time	probability	time left t_1	time left t_2
r_1	0	0.5	1	0
		0.5	2	0
r_2	1	0.5	0	0
		0.5	1	0
r_2	1	0.25	0	1
		0.25	0	2
		0.25	1	2
		0.25	1	1
		0.25	1	2
d_1	3	0.25	0	0
		0.25	0	0
		0.25	0	0
		0.25	0	1
d_1	3	0.75	0	0
		0.25	0	1

Table 1: States

every possible phase shift is not always necessary since introducing phase shifts does not dramatically increase the number of deadline misses most of the time. Only if all phases are possible and absolute certainty about the number of missed deadlines is required. This last condition, however, is usually contrary to the nature of soft real-time.

5 Complexity

In this section we derive the worst case time complexity of the method we introduced and compare it with the complexity of the method in [10].
Since feasibility analysis of an arbitrary periodic task system is shown to be co-NP-hard in the strong sense [12], it is not likely that there is a way to get an exact performance measure in polynomial time, even more so when we are dealing with stochastic processing times. If we'd introduce different phase shifts for tasks, the problem would be even harder, but we are ignoring those. We show in this section that our algorithm has a total complexity of $O(qnm^n)$. In this formula n is the number of tasks, m is the maximum processing time of any one task, while q represents the total number of deadline and release events in one hyperperiod.
At each of these events our algorithm performs a number of operations, but unfortunately, the number of events in one hyperperiod can grow exponentially. Given a set of n tasks $\Omega = \{\tau_1 \dots \tau_n\}$ we can calculate the hyperperiod T^H as the l.c.m. of $\{T_1 \dots T_n\}$. In the worst case, where all periods are distinct prime numbers, we have $T^H = \prod_{i=0}^{n} T_i$. The number of events in T^H, q, is equal to two times the total number of jobs, N. If N_j denotes the number of jobs of τ_j during T^H; i.e. $N_j = T^H/T_j$; we have $N = T^H \sum_{j=0}^{n} 1/T_j$. Since T^H can grow exponentially in n, the number of events we have to process, q, can also grow at this rate.
At each event, at most two actions take place. The first action that can take place if time has passed since the last event, is using the scheduling algorithm on all states to determine how they should be changed. We denote the effort of the scheduling algorithm for one state by C^n_{sched}. We assume that the time this takes is near linear in n. Another important influence on the on the complexity of our approach is the maximum number of states, $|S_t|$. This number is determined by the maximum processing time for every task. If m is the maximum processing time for all tasks τ_i, then we can derive that for any time t, $|S_t| \leq \prod_{i=0}^{n} m \leq m^n$. If we are not using preemption, $|S_t|$ is much smaller, since per state only one task can be in the middle of being processed. All other tasks are ready or have their full processing time still left. This would lead to a number of states smaller or equal to $m * (h+1)^{n-1}$, where h is the maximum number of different realizations for any task.
The second action depends on the type of event. In the case of a deadline event we have to merge identical events. This is possible in constant time assuming we already assigned memory for all possible states. In the case of a release event we add new states. If c_j^{num} is the number of possible realizations of a released task τ_j at time t_i, then $S_t \leq S_{t_{i-1}} c_j^{num}$, making this action linear in the number of states.
The above argumentation gives an approximation of the total complexity of our algorithm of $O(qnm^n)$. If we compare this to the $O(q^3m^2)$ complexity of the methods in [10], we see that if the number of events, q, is high, our performance can be much better. This is because although that paper claims a polynomial time performance, it fails to take the exponential nature of q into account. In situations where m or n is high, our algorithm might start to perform worse.
This difference in complexity of our algorithm and the algorithm of [10], is explained by the fact that the latter has a smaller state space. In their algorithm, the processing times of separate tasks can be added because the algorithm has to do a separate calculation for each deadline, while in our algorithm, the results are calculated in one run.

6 Testing and Comparing Scheduling Methods

In this section we first evaluate the performance of the introduced algorithm by applying it to uncomplicated examples from the well known EDF and RM scheduling methods using different parameters. After this evaluation, we demonstrate the abilities of our algorithm to deal with stochastic processing times. We show the difference between achieved load estimations by taking into account the stochastisity and by using worst case considerations. To show the versatility of our algorithm, we introduce versions of EDF and RM that are modified for overload situations, and compare their performance to their unmodified form.
For our tests we use instances with different parameters. The most important parameters are the system workload W, the number of tasks n, and for every task $\tau_i$the length of periods T_i, the variation in periods, and the variation in the processing times, j_i. The workload determines how much time is necessary to process all the tasks in the system over a longer period of time, and we measure it as a percentage of

the maximum utilization. The periods of the tasks influence the length of the hyper period and thus the number of events. Randomly picking periods from a small interval (in our case from 2 to 10) and then multiplying all periods with a fixed number gives the opportunity to introduce variation in the processing times, without increasing the length of the hyperperiod too much, thus keeping the number of events q limited. To generate the processing times, we first randomly generate fractions f_i for each task τ_i ($\sum f_i = 1$). Using these fractions and a given total load, we can calculate individual loads for each task resulting in a given average processing time $p_i = W \cdot T_i \cdot f_i$ for each task τ_i. For generating a distribution on the processing times, we use for these examples two processing times $p_i + j_i$ and $p_i - j_i$ for task τ_i both with probability 0.5. In our tests, j is equal to 2 for all tasks, unless otherwise indicated. If $p_i - j_i < 1$ we take a processing time of one instead. This construction of the processing times models jitter j of the processing times. All our tests were performed on a standard desktop computer with a Pentium IV type processor.

In our first test, we show how the run time of our algorithm correlates with an increasing number of tasks. We use four different scheduling methods (EDF and RM, with and without preemption) to show their effects on the run time. We use random periods between 2 and 10 and scale them by a factor of 10. The total load of the system is set to 110%. In Figure 2 we see an increase in the processing time if the number of tasks increases. This increase is the most extreme for the combination of EDF and preemption. In the cases where we use RM, our algorithm uses less computation time. The average number of states that our algorithm needs tends to be much smaller for RM, especially when preemption is used. We attribute this behavior to the high predictability of tasks with small periods, due to RM's static priorities. This predictability is increased by preemption.

We also investigated the effects of longer periods on the run time of our algorithm. For this goal, we created instances using the same basic values for the periods (i.e. leading to the same number of events, q), but scaling them by different scaling factors. The jitter has also been scaled by the same factor. With this kind of scaling, while still using integers, we obtain a finer granularity of results, leading to more precision. Scaling leads to longer processing times for larger scaling factors, because it increases the maximum processing time m (see previous section). The processing times were chosen such that we always achieved the same load (110%). In Figure 3 we see an increase in processing time for bigger scaling factors. Like in the pervious test, EDF is performing worse than RM, but to a lesser extent. Using preemption did not lead to a big increase in the number of states (and thus the running time) that could be expected from the complexity results from the previous section. It seems scaling has less effect on our algorithms running time than the number of tasks. Since scaling allows for a better granularity in approximations, this is good news.

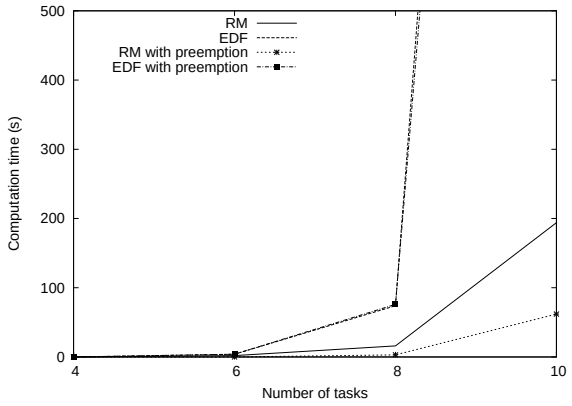

Figure 2: The effects of the number of tasks on processing time.

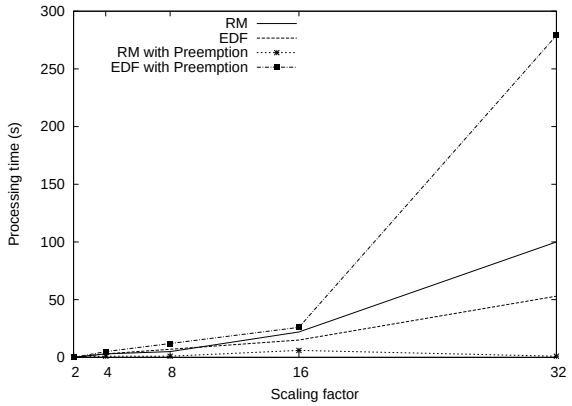

Figure 3: The effects of scaling on processing time.

In the following test we show how using worst case approximations for processing times can easily lead to

overestimating the necessary system resources. This test uses EDF with preemption, but other scheduling methods show similar behavior. In Figure 4 we give the results for a test with jitter $j = 2$ for all tasks. The top line represents the worst case scenario with processing times $p + j$, the other two lines represent the average case (processing time p) and the stochastic approximation ($p \pm j$). We can clearly see a large difference between these two lines, and the worst case scenario. It is obvious that using the worst case scenario leads to a bad performance prediction. A system designed to handle a certain load based on the worst case scenario will in reality be able to handle a load of more than 20 percent higher. As a surprising side result, we see that using the average case processing is a good approximation for estimating the number of missed deadlines. However, this might be different if processing times have a different distribution, where the the realizations of one task can be more diverse than in the setting chosen in this test.

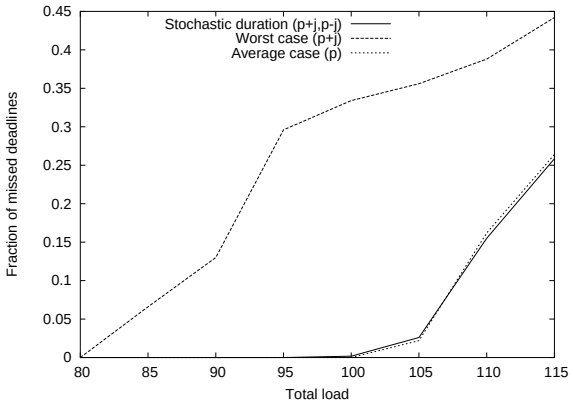

Figure 4: Stochastic processing times vs worstcase

One disadvantage of the tested scheduling methods in high load situations is that they may spend a lot of time working on tasks that will most likely not meet their deadline. Therefore in such high load situations it might be useful to look ahead. More precisely, we have modified RM and EDF such that at the time a job is about to be scheduled the scheduler checks if the avarage processing time of this task is longer than the time till its deadline. If this is the case, the job is dropped. We show this method for non-preemptive scheduling only, since a preemptive algorithm would require a much more complicated structure.
Leaving out late tasks has advantages in high load situations as can be read from Figure 5. At lower loads the advantage turns into a disadvantage, since we are dealing with averages and there might be a chance the task would not be late if it was lucky and has a short processing time. This scheduling method works well if the scheduler has information about the average processing time. For example, in an advanced system, the scheduler would be able to keep track of processing times and calculate an average after gathering sufficient data.

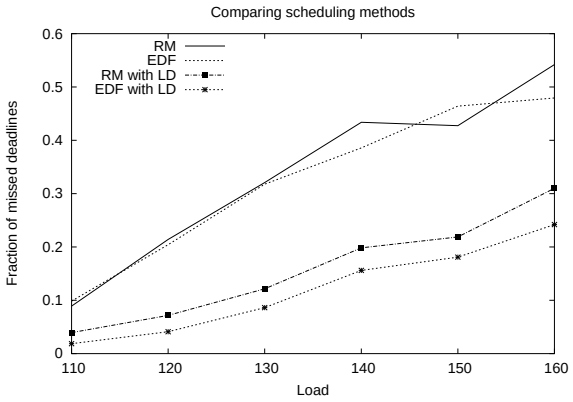

Figure 5: The effects of removing late tasks (LD).

7 Conclusion

Using methods derived from dynamic programming, allows to check effectively how many missed deadlines we may expect in a periodic system with soft real-time constraints. Although running times are still exponential in the worst case, this algorithm generally offers a significant boost in speed. Our algorithm can be modified to work with both hard and soft deadlines or deal with almost any kind of scheduling method. We have assumed discrete distributions of the processing times with a limited number of realizations. We believe these assumptions allows us to closely approach reality. Future research needs to be done, to see what kind of distributions are good approximations for the run times of different tasks.
Based upon our findings in testing several scheduling algorithms, it we believe advantageous to use specialized algorithms to schedule tasks with soft

deadlines under overload conditions more efficiently. Such algorithms would need a mechanism to prevent too much time being spent on tasks that will, with high probability, not meet their deadlines anyway.

References

[1] L. Abeni and G. Buttazzo. Qos guarantee using probabilistic deadlines. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 242–249, 1999.

[2] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *ACM Symposium on Theory of Computing*, pages 345–354, 1993.

[3] S. K. Baruah, A. K. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the Real-Time Systems Symposium*, pages 182–190, Dec 1990.

[4] Guillem Bernat, Alan Burns, and Albert Llamosi. Weakly hard real-time systems. *IEEE Trans. Comput.*, 50(4):308–321, 2001.

[5] Guillem Bernat, Antoine Colin, and Stefan M. Petters. Wcet analysis of probabilistic hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 279, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Giorgio C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[7] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Information Processing 74, proceedings of IFIP congress 74*, pages 807–813, Stockholm, Sweden, August 1974. North Holland Publishing Company. ISBN 0-7204-2803-3.

[8] Mark K. Gardner. *Probabilistic analysis and scheduling of critical soft real-time systems.* PhD thesis, University of Illinois, Urbana, Illinois, 1999.

[9] D.F K. Jeffay, Stanat, and C.U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. of the* 12^{th} *IEEE Real-Time Sys. Symp.*, pages 129–139, 1991.

[10] Kanghee Kim, Jose Luis Diaz, and Jose Maria Lopez. An exact stochastic analysis of priority-driven periodic real-time systems and its approximations. *IEEE Trans. Comput.*, 54(11):1460–1466, 2005. Member-Lucia Lo Bello and Member-Chang-Gun Lee and Member-Sang Lyul Min.

[11] H. Kopetz. *Real-Time Systems.* Kluwer Academic, Boston, MA, USA, 1997.

[12] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[14] A.K.L. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, MIT, 1983.

[15] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-time Systems*, 10(2), 1996.

[16] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *RTAS '95: Proceedings of the Real-Time Technology and Applications Symposium*, page 164, Washington, DC, USA, 1995. IEEE Computer Society.