Memory and Time-Efficient Schedulability Analysis of Task Sets with Stochastic Execution Time

Sorin Manolache, Petru Eles, Zebo Peng
Department of Computer and Information Science, Linkoping University, Sweden
{sorma, petel, zebpe}@ida.liu.se

Abstract

This paper presents an efficient way to analyse the performance of task sets, where the task execution time is specified as a generalized continuous probability distribution. We consider fixed task sets of periodic, possibly dependent, non-pre-emptable tasks with deadlines less than or equal to the period. Our method is not restricted to any specific scheduling policy and supports policies with both dynamic and static priorities. An algorithm to construct the underlying stochastic process in a memory and time efficient way is presented. We discuss the impact of various parameters on complexity, in terms of analysis time and required memory. Experimental results show the efficiency of the proposed approach.

1. Introduction

Embedded systems are digital systems meant to perform a dedicated function inside a larger system. Most embedded systems are also real-time systems. Hence, their validation has to take into consideration not only the functional aspect but also timeliness.

A typical digital systems design flow starts from an abstract description of the functionality and a set of constraints. Subsequent steps partition the functionality, allocate processing units, map the functionality to the allocated processing units and, finally, select a scheduling policy and perform timing analysis.

In safety-critical applications missing a deadline can have disastrous consequences. Hence, a conservative model for task scheduling is adopted, the worst-case execution time (WCET) model. Such systems are designed to perform according to hard time constraints even in rare borderline cases. This leads to a processor underutilization for most of the operation time.

There are several opportunities to relax some of the conservative assumptions typical to hard real-time systems. This is the case for applications where missing a deadline causes an overall quality degradation, but it is still acceptable provided that the probability of such misses is below a certain limit. Such applications are, for example, multimedia and telecommunication systems.

Another opportunity to relax the WCET model is during the early design stages. For instance, in transformational design approaches, the design phases may not be performed in the simplistic waterfall sequence described above. It may happen that information about a schedule fitness would be needed without knowing exactly the processor on which a certain task will be executed. The processor itself could be still under design and, thus, no exact information concerning execution time would be available.

Tia et al. [11] provide an example where the maximum processor utilizations are around 145%, whereas the average utilizations do not exceed 25%. The large variation in utilizations stems from the large variation of task execution times. This could be due to several reasons: architectural factors (dynamic features like caches and pipelines), causes related to functionality (data dependent branches and loops), external causes (network delays), and dependencies on input data (very strong in multimedia applications). Another source of non-exact execution time specification is the limited amount of information available. This could be the case at early design phases or with designs integrating third party components, or other customer blocks with secret functionality and insufficiently specified non-functional interfaces.

Typically, by using variable execution time models, considerable savings in system (hardware) cost can be expected. The functionality can be implemented on less powerful and cheaper processors, leading to a higher processor utilization. In the case of overload, some tasks will, most likely, fail their deadlines. The designer has to be provided with analysis tools in order to guide his or her decisions and to estimate the trade-off between cost and quality, the failure likeliness and the failure consequences. Thus, new execution time models and analysis techniques have to be developed.

Approaches based on the average case or on probabilities uniformly distributed between the best and worst case execution time have the advantage of simplicity. However, their use is limited, as they do not give information on the likeliness of particular cases. More accurate models are based on execution time probability distributions. Those distributions can be derived from statistical models of the variation sources, from legacy designs, code analysis, simulations and profiling. One of the main difficulties with probabilistic models is their solving complexity.

The aim of this work is to provide a performance analysis method for task schedules considering probabilistic models of task execution times. The methodology is not specific to any particular execution time probability distribution class or scheduling policy and, thus, it is adaptable to various applications. The result of the application analysis is the ratio of missed deadlines per task or task graph. In order to cope with the complexity problem typical to such an analysis, we have considered both execution time and

memory aspects. The algorithm is efficient in both regards and can be applied to the analysis of large task sets. We have also investigated the impact of the task set parameters (e.g. periods, dependencies, and number of tasks) on the complexity of the analysis in terms of time and memory.

The rest of the paper is structured as follows. Section 2 surveys some related work. Section 3 details our assumptions and gives the problem formulation. Section 4 introduces the underlying stochastic process and illustrates its construction and analysis on an example. Section 5 presents the analysis algorithm and in section 6 we evaluate our approach experimentally. The last section draws some conclusions.

2. Related work

Much work has been done in the field of task scheduling with fixed parameters (periods, execution times, etc.). Results are summarized in several surveys such as those by Stankovic et al. [10], Fidge [5], and Audsley et al. [3]. However, only recently researchers have focused on scheduling policies, schedulability analysis and performance analysis of tasks with stochastic parameters.

Atlas and Bestavros [2] extend the classical rate monotonic scheduling policy with an admittance controller in order to handle tasks with stochastic execution times. They analyse the quality of service of the resulting schedule and its dependence on the admittance controller parameters. The approach is limited to rate monotonic analysis and assumes the presence of an admission controller at run-time.

Abeni and Butazzo's work [1] addresses both scheduling and performance analysis of tasks with stochastic parameters. Their focus is on how to schedule both hard and soft real-time tasks on the same processor, in such a way that the hard ones are not disturbed by ill-behaved soft tasks. The performance analysis method is used to assess their proposed scheduling policy (constant bandwidth server), and is restricted to the scope of their assumptions.

Spuri and Butazzo [9] propose five scheduling algorithms for aperiodic tasks. The task model they consider is one with aperiodic tasks but with fixed, worst-case execution time.

Tia et al. [11] assume a task model composed of independent tasks. Two methods for performance analysis are given. One of them is just an estimate and demonstrated to be overly optimistic. In the second method, a soft task is transformed into a deterministic task and a sporadic one. The latter is executed only when the former exceeds the promised execution time. The sporadic tasks are handled by a server policy. The analysis is carried out on this model.

Zhou et al. [12] root their work in Tia's. However, they do not intend to give per-task guarantees, but characterize the fitness of the entire task set. Because they consider all possible combinations of execution times of all requests up to a time moment, the analysis can be applied only to small task sets due to complexity reasons.

De Veciana et al. [4] address a different type of problem. Having a task graph and an imposed deadline, they determine the path that has the highest probability to violate the deadline. The problem is then reduced to a non-linear optimization problem by using an approximation of the convolution of the probability densities.

Lehoczky [8] models the task set as a Markovian process. The advantage of such an approach is that it is applicable to arbitrary scheduling policies. The process state space is the vector of lead-times (time left until the deadline). As this space is potentially infinite, Lehoczky analyses it in heavy traffic conditions, when the system provides a simple solution. The main limitations of this approach are the non-realistic assumptions about task inter-arrival and execution times.

Kalavade and Moghe [7] consider task graphs, where the task execution times are arbitrarily distributed over discrete sets. Their analysis is based on Markovian stochastic processes too. Each state in the process is characterized by the executed time and lead-time. The analysis is performed by solving a system of linear equations. Because the execution time is allowed to take only a finite (most likely small) number of values, such a set of equations is small.

Besides the differences in assumptions, our work diverges from Kalavade and Moghe's in the sense that we use pseudo-continuous execution time distributions (discretized continuous distributions) instead of being restricted to discrete sets. As the number of possible execution times becomes very high, it is infeasible to consider individual times as states. Our solution is to group execution times in equivalence classes. As a consequence, we have to use probability density convolutions for the analysis. In order to reduce the complexity in terms of memory space, the stochastic process is never stored entirely in memory, but we both construct and analyse the process at the same time.

3. Preliminaries and problem formulation

The system to be analysed is represented as a set of task graphs. A task graph is an acyclic graph with nodes representing tasks and edges capturing the precedence constraints among tasks. Precedences can be induced, for example, by data dependencies (a task processes the output of its predecessor). All tasks are executed on one single processor. They are assumed to be non-pre-emptable.

Let N be the number of tasks and let t_i, $0 \leq i < N$, denote a task. Let M be the number of task graphs and let g_i, $0 \leq i < M$ denote a task graph.

Each task is characterized by its period (inter-arrival time), assumed to be fixed, its deadline, and its execution time probability density. Let p_i and d_i, $0 \leq i < N$, denote the period and deadline of task t_i, where $d_i \leq p_i$. P, the application period, is the least common multiple of all task periods. The period of a task has to be a common multiple of the periods of its predecessors. The period of a task graph equals the least common multiple of the periods of its composing

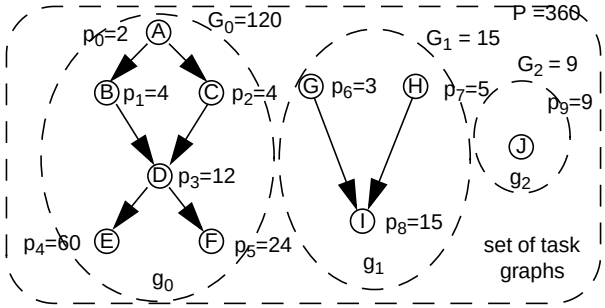

Figure 1. Set of task graphs

tasks. Let G_i, $0 \le i < M$, denote the period of the task graph g_i. A task graph g_i is activated every G_i time units.

A task consists of an infinite sequence of activations called jobs. In the sequel, we will say that a task is running when one of its jobs is running. Similarly, a task is ready when one of its jobs is ready, and a task is discarded when one of its jobs was discarded.

The task $t_k \in g_i$ is ready (pending, waiting) if and only if each of its predecessors t_j has run p_k/p_j times during the current activation of task graph g_i. If any of the jobs in a current activation of a task graph has missed its deadline, then the current task graph activation is said to have failed.

A probabilistic guarantee given for a task *t* is expressed as the ratio between the number of jobs belonging to *t* that miss their deadlines and the total number of jobs of the task *t*. A probabilistic guarantee given for a task graph *g* is expressed as the ratio between the number of the task graph's activations that fail and the number of all activations of the task graph *g*.

Figure 1 depicts a task set consisting of three task graphs g_0, g_1, and g_2. For each task and task graph the respective period is shown.

Let e_i, $0 \le i < N$, be the execution time probability density function (ETPDF) of task t_i. e_i is represented as a set of samples resulting from the discretization of the density curve. The discretization resolution is left to the designer's choice. We assume generalized probability densities of the task execution times. Figure 2 illustrates some possible ETPDFs. Density e_1 could happen if, for example, the task has only three computation paths and the variation around them is caused by hardware architecture factors. e_2 is a deterministic density (a Dirac impulse).

When a job of a task t misses its deadline it is discarded. If t has a successor in the task graph then two different policies are considered for the analysis among which the designer can choose:

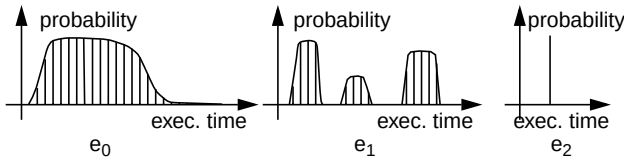

Figure 2. Exec. time probability density functions

1. The whole task graph is discarded. This means that all the jobs belonging to the current activation of the task graph are discarded. These are the ones already arrived but not yet completed and the ones that will arrive before a new activation of the task graph. This strategy is adopted in the case when the computed value of a job is critical for the continuation of the tasks in the task graph and it is a meaningful value only if the job met its deadline. In this case, it is meaningless to give per task guarantees and only per task graph guarantees will be produced as a result of the analysis. In the example in Figure 1 assume that two jobs of task A have successfully executed and B and C are now ready. Assume that B executes and misses its deadline. Then g_0 is discarded and no jobs of any of its tasks are anymore accepted until a new arrival of the entire task graph at a time moment multiple of G_0.
2. Only the missing job is discarded, but the rest of the task graph is activated normally. The successor tasks will consume either a partial result or the result from a previous execution of the discarded task. In this case, it is important to provide not only per task graph but also per task guarantees.

Task execution is considered to be non-pre-emptable. However, a method to work this around, if needed, is to define pre-emption points inside a task. For analysis, the task will be replaced by several dependent tasks with the same period and deadline as the original one, as shown in Figure 3. Let τ_i^q, $0 \le i < S$ be S tasks that resulted from task t_q, and π_i^q be their respective periods, δ_i^q their deadlines and ε_i^q their ETPDFs. Then $\pi_i^q = p_q$ and $\delta_i^q = d_q$, $0 \le i < S$. The convolution (denoted by *) of the execution time probability density functions ε_i^q has to be equal to e_q. Due to the fact that the tasks τ_i^q are dependent and have the same period, the consequence of such a task decomposition on the analysis complexity is limited, as will be shown later.

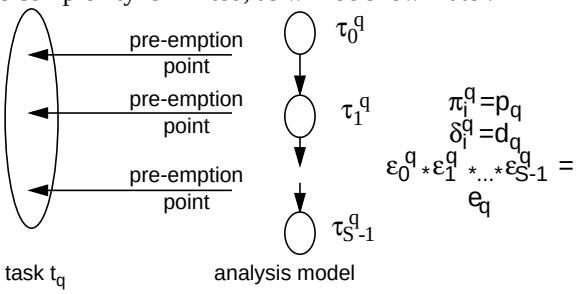

Figure 3. Introducing pre-emption points

Problem formulation

The input to the analysis algorithm is a set of task graphs and a scheduling policy. The task graphs are given according to the assumptions discussed previously. The scheduling policy is given as an algorithm to choose a next job knowing the set of ready jobs, their deadlines and the current time.

The analysis produces per task and per task graph probabilistic guarantees, as defined above.

4. The stochastic process

The analysis methodology for task sets with stochastic execution times relies on the underlying stochastic process. A stochastic process is a mathematical abstraction that characterizes a random process which proceeds in stages. The set of all possible stage outcomes in every stage forms the stochastic process state space. A stochastic process can be represented graphically in a similar manner as finite state machines. The difference is that a next state is known only probabilistically. It is assumed that the next state transition probabilities are known once the current and past states are known. If the next states and their corresponding transition probabilities depend only on the current state, then the process exhibits the Markovian property. The discrete time stochastic process that results from sampling the state space of the underlying continuous time stochastic process, at the time moments immediately following a job arrival or a job completion, forms the embedded stochastic process.

For the sequel, "process" will refer to the underlying stochastic process, and "state" will refer to the stochastic process state.

In order for the embedded process to be Markovian, certain information have to be available in a process state. A straightforward solution would be to characterize a state by the currently running task, the ready tasks and the start time of the running job. A state change would occur if the running task finished execution for some reason. The ready tasks can be deduced from the old ready tasks and the jobs arrived during the old task's execution time. The new running job can be selected considering the particular scheduling policy. In principle, there may be as many next states as many possible execution times the running job has. Hence, one factor that influences the stochastic process size is the task execution time span. This is the approach followed by Kalavade and Moghe [7] and leads to tree-like stochastic processes. Except the case that only a very small number of discrete execution times are allowed for each task, such an approach leads to an extremely huge state explosion. In our approach, we have grouped time moments into equivalence classes and, by doing so, we limited the process size explosion. Thus, practically a set of equivalent states is represented as a single state in the stochastic process. Even so, the application size is still limited by the amount of memory available for analysis. Therefore, we propose a way to perform the construction and the analysis of the process simultaneously. Consequently only a part of the process is stored in memory at any time during the analysis.

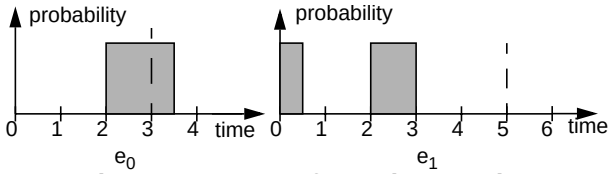

Figure 4. ETPDFs for tasks t_0 and t_1

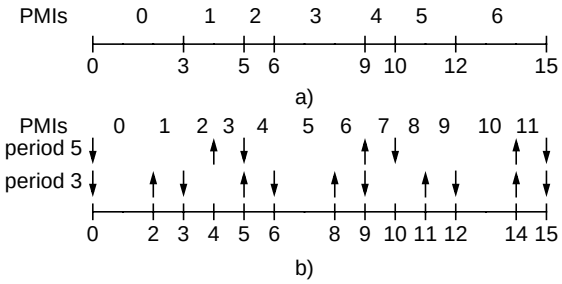

Figure 5. Priority monotonicity intervals

Due to the assumption that the tasks are discarded when they miss their respective deadlines, the analysis is performed over the interval [0, P), where P is the application period (the least common multiple of all task periods).

In the following, an example is used in order to illustrate the construction and the analysis of the stochastic process. Let t_0 and t_1 be two independent tasks scheduled according to an earliest deadline first (EDF) policy. Let $p_0 = 3$ and $p_1 = 5$ be their periods and let $d_0 = p_0$ and $d_1 = p_1$. P is then 15. The execution time probabilities are distributed as depicted in Figure 4. For simplicity, the densities were not depicted as discretized. Note that e_0 contains execution times larger than the deadline.

As a first step to the analysis, the interval [0, P) is divided in disjunct intervals, the so-called *priority monotonicity intervals (PMI)*. A PMI is delimited by the time moments a job may arrive or may be discarded. Figure 5a depicts the PMIs for the example above. If the deadlines were $d_0 = 2$ and $d_1 = 4$, then the PMIs would be as depicted in Figure 5b.

Next, the stochastic process is constructed and analysed at the same time. Let us assume the straightforward approach mentioned earlier. In this case, a stochastic process state would be characterized by the index of the task the currently running job belongs to, the start time of this job and the indexes of the waiting tasks (see Figure 6a). τ_1, τ_2, …, τ_q in Figure 6a are possible finishing times for the job of task t_0 and, implicitly, possible starting times of the waiting job of task t_1. The number of next states equals the number of possible execution times of the running job in the current state. The resulting process is extremely large (theoretically infinite, practically depending on the discretization resolution) and, in practice, unsolvable. Therefore, we would like to group as many states as possible in one equivalent state and still preserve the Markovian property.

Consider a state s_0 characterized by {i, t, w}: the current

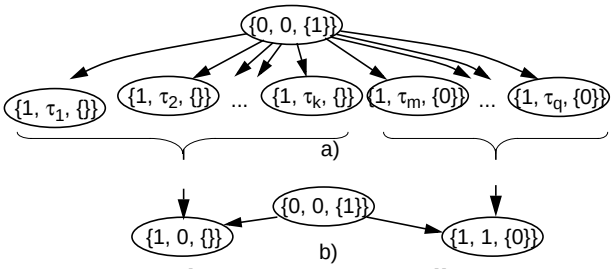

Figure 6. State encoding

job belongs to task t_i, it has been started at time t, and the waiting jobs belong to the tasks in set w. Let us consider the next states derived from s_0: s_1 characterized by {j, τ_1, w_1} and s_2 with {k, τ_2, w_2}. Let τ_1 and τ_2 belong to the same PMI. This means that no job has arrived or finished in the time interval between τ_1 and τ_2, no one has missed its deadline, and the relative priorities of the tasks inside the set w have not changed (see Section 5.1). Thus, j=k= the index of the highest priority task in the set w; w_1=w_2=w\{t_j}. It follows that all states derived from state s_0 that have their time τ belonging to the same PMI have an identical current job and identical sets of waiting jobs. Therefore, instead of considering individual times we consider time intervals, and we group together those states that have their associated start time inside the same PMI. With such a representation, the number of next states of a state *s* equals the number of PMIs the possible execution time of the job that runs in state *s* is spanning over.

We propose a representation in which a stochastic process state is a triplet {r, pmi, w}, where r is the running task index, pmi is the index of the PMI containing the running job's start time, and w the set of ready task indexes. When there is no running task, then r = -1. In our example, the execution time of task t_0 (which is in the interval [2, 3.5]) is spanning over the PMIs 0 and 1. Thus, there are only two states emerging from the initial state, as shown in Figure 6b.

Let $\wp_i$, the set of predecessor states of a state s_i, denote the set of all states that have s_i as a next state. The set of successor states of a state s_i consists of those states that can be reached directly from state s_i. With our proposed stochastic process representation, the time moment a transition to a state s_i occurred is not determined exactly, as the task execution times are known only probabilistically. However, a probability density of this time can be deduced. Let z_i denote this density function. Then z_i can be computed from the functions z_j, where $s_j \in \wp_i$, and the ETPDFs of the tasks running in the states $s_j \in \wp_i$.

Figure 7 depicts a part of the stochastic process constructed for our example. The initial state is s_0: {0, 0, {1}}. The first field indicates that a job of task t_0 is running. The second field shows the current pmi (0), and the third field denotes that task t_1 is waiting. If the job of task t_0 does not complete until time moment 3, then it will be discarded. The state s_0 has two possible next states. The first one is state s_1: {1, 0, {}} and corresponds to the case when the job completes before time moment 3. The second one is state s_2: {1, 1, {0}} and corresponds to the case when the job was discarded at time moment 3. State s_1 indicates that a job of task t_1 is running (it is the job that was waiting in state s_0), that the pmi is 0 and that no job is waiting. Consider state s_1 to be the new current state. Then the next states could be state s_3: {-1, 0, {}} (task t_1 completed before time moment 3 and the processor is idle), state s_4: {0, 1, {}} (task t_1 completed at a time moment somewhere between 3 and 5), or state s_5: {0, 2, {1}} (the execution of task t_1 reached over time moment 5, and hence it was discarded at time moment 5). The construction procedure continues until all possible states corresponding to the time interval [0, P) have been visited.

The transition time probability density functions z_1, z_2, z_3, z_4, and z_5 are shown in Figure 7 to the left of their corresponding states. The transition from state s_3 to state s_4 occurs at a precisely known time instant, time 3, at which a new job of task t_0 arrives. Therefore, z_4 will contain a Dirac impulse at the beginning of the corresponding PMI. The probability density function z_4 results from the superposition of z_1 * e_1 (because task t_1 runs in state s_1) with z_2 * e_1 (because task t_1 runs in state s_2 too) and with the aforementioned Dirac impulse over the PMI 1, i.e. over the time interval [3, 5).

The embedded process being Markovian, the probabilities of the transitions out of a state s_i are computed exclusively from the information stored in that state s_i. For example, the probability of the transition from state s_1 to state s_4 (see Figure 7) is given by the probability that the transition occurs at some time moment in the PMI of state s_4 (the interval [3, 5)). This probability is computed by integrating z_1 * e_1 over the interval [3, 5). The probability of a task missing its deadline is easily computed from the transition probabilities of those transitions that correspond to a job discarding (the thick arrows in Figure 7).

As it can be seen, by using the PMI approach, some process states have more than one incident arc, thus keeping the

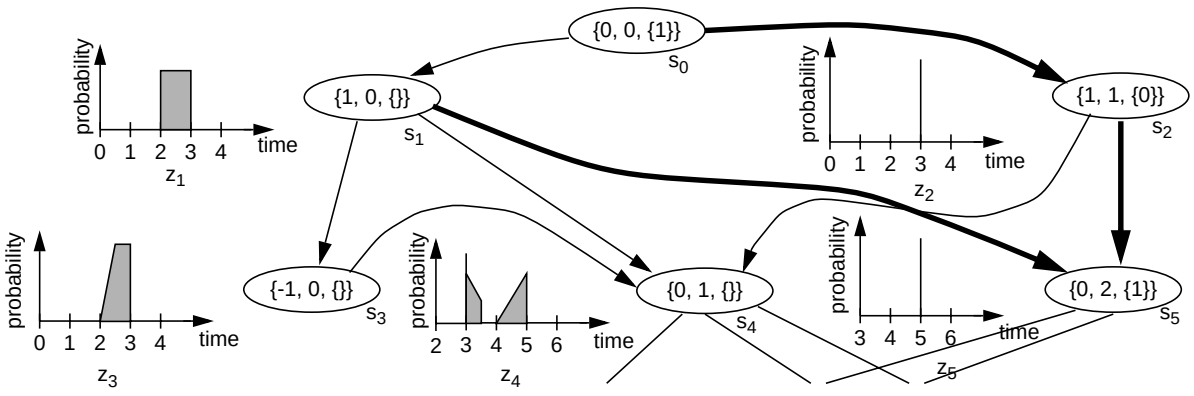

Figure 7. Stochastic process example

graph “narrow”. This is because, as mentioned, one process state in our representation captures several possible states of a representation considering individual times (see Figure 6a).

Because the number of states grows rapidly and because each state has to store its probability density function, the memory space required to store the whole process can become prohibitively large. Our solution to mastering memory complexity is to perform the stochastic process construction and analysis simultaneously. As each arrow updates the time probability density of the state it leads to, the process has to be constructed in topological order. The result of this procedure is that the process is never stored entirely in memory but rather that a sliding window of states is used for analysis. For the example in Figure 7, the construction starts with state s_0. After its next states (s_1 and s_2) are created, their corresponding transition probabilities determined and the possible discarding probabilities accounted for, state s_0 can be removed from memory. Next, one of the states s_1 and s_2 is taken as current state, let us consider state s_1. The procedure is repeated, states s_3, s_4 and s_5 are created and state s_1 removed. At this moment, the arcs emerging from states s_2 and s_3 have not yet been created. Consequently, one would think that any of the states s_2, s_3, s_4, and s_5 can be selected for continuation of the analysis. Obviously, this is not the case, as not all the information needed in order to handle states s_4 and s_5 are computed (in particular those coming from s_2 and s_3). Thus, only states s_2 and s_3 are possible alternatives for the continuation of the analysis in topological order. In Section 5 we discuss the criteria for selection of the correct state to continue with.

5. Stochastic process construction and analysis

The analysis of the stochastic task set is performed in two phases:

1. Divide the interval [0, P) in PMIs.
2. Construct the stochastic process in topological order and analyse it.

5.1. Priority monotonicity intervals

The concept of PMI (called in their paper “state”) was introduced by Zhou et al. [12] in a different context, unrelated to the construction of a stochastic process.

Let A_i denote the set of time moments in the interval [0, P) when a new job of task t_i arrives and let A denote the union of all A_i. Let D_i denote the set of absolute deadlines[1] of the jobs belonging to task t_i in the interval [0, P), and D be the union of all D_i. Consequently,

$$A_i = \{x | x = k \cdot p_i, 0 \le k < P/p_i\}$$
$$D_i = \{x | x = d_i + k \cdot p_i, 0 \le k < P/p_i\}$$

If the deadline of a certain task t_i equals its period, then $A_i = D_i$ (the time moment 0 is considered, conventionally, to be the deadline of the job arrived at time moment $-p_i$).

Let $H = A \cup D$. If H is sorted in ascending order of the time moments, then a priority monotonicity interval is the interval between two consecutive time moments in H. The last PMI is the interval between the greatest element in H and P.

The only restriction imposed on the scheduling policies accepted by our approach is that inside a PMI the ordering of tasks according to their priorities is not allowed to change. The consequence of this assumption is that the next state can be determined no matter when the currently running task completes within the PMI. All the widely used scheduling policies we are aware of (RM, EDF, FCFS, etc.) exhibit this property.

5.2. The construction and analysis algorithm

The algorithm proceeds as discussed in Section 4 An essential point is the construction of the process in topological order, which allows only parts of the states to be stored in memory at any moment.

The algorithm for the stochastic process construction is depicted in Figure 8. All states belonging to the sliding window are stored in a priority queue. The key to the process construction in topological order lies in the order in which the states are extracted from this queue. First, observe that it is impossible for an arc to lead from a state with a PMI number u to a state with a PMI number v so that $v < u$ (there are no arcs back in time). Hence, a first criterion for selecting a state from the queue is to select the one with the smallest PMI number. A second criterion determines which state has to be selected out of those with the same PMI number. Note that inside a PMI no new job can arrive, and that the task ordering according to their priorities is unchanged. Thus, it is impossible that the next state s_k of a current state s_j would be one that contains waiting tasks of higher priority

```
put first state in the queue;
while queue not empty do
   s_j = extract state from the queue;
   t_i = s_j.running; -- field r of state s_j
   distribution = convolute(e_i, z_j);
   nextstatelist = next_states(s_j);
            -- consider task dependencies!
   for each s_k ∈ nextstatelist do
     compute probability of the transition
          from s_j to s_k using distribution;
     update discarding probabilities;
     update z_k;
     if s_k is not in the queue then
       put s_k in the queue;
     end if;
   end for;
   delete state s_j;
end while;
```

Figure 8. Construction and analysis algorithm

1.Except here, whenever we use the term “deadline”, we consider relative deadlines.

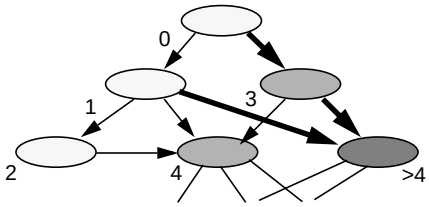

Figure 9. State selection order

than those waiting in s_j. Hence, the second criterion reads: among states with the same PMI, one should choose the one with the waiting task of highest priority.

Figure 9 illustrates the algorithm on the example given in Section 4 (Figure 7). The shades of the states denote their PMI number. The lighter the shade, the smaller the PMI number. The numbers near the states denote the sequence in which the states are extracted from the queue and processed.

6. Experimental Results

The most computation intensive part of the analysis is the computation of the convolutions. In our implementation we used the FFTW library [6] for performing convolutions based on the Fast Fourier Transform. The number of convolutions to be performed equals the number of states of the stochastic process. The memory required for analysis is determined by the maximum number of states in the sliding window. The main factors on which the stochastic process depends are P (the least common multiple of the task periods), the number of PMIs, the number of tasks, and the task dependencies.

As the selection of the next running task is unique, given the pending jobs and the time moment, the particular scheduling policy has a reduced impact on the process size. On the other hand, the task dependencies play a significant role, as they strongly influence the set of ready tasks and by this the process size. Additionally, they generate a smaller number of PMIs as they impose a certain harmony among the task periods.

In the following, we report on three sets of experiments. The aspects of interest were the stochastic process size, as it determines the analysis execution time, and the maximum size of the sliding window, as it determines the memory space required for the analysis. All experiments were performed on an UltraSPARC 10 at 450 MHz.

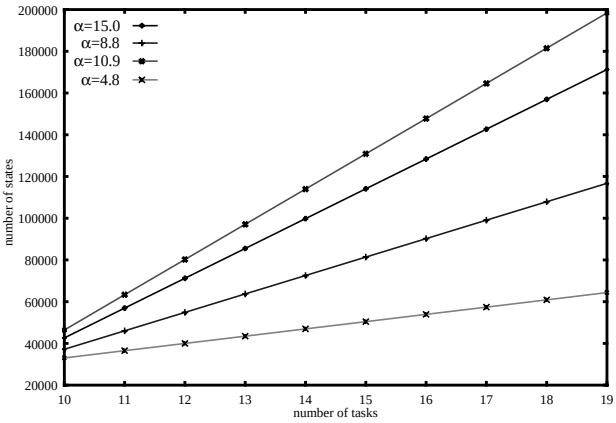

Figure 10. Experiment 1, stochastic process size

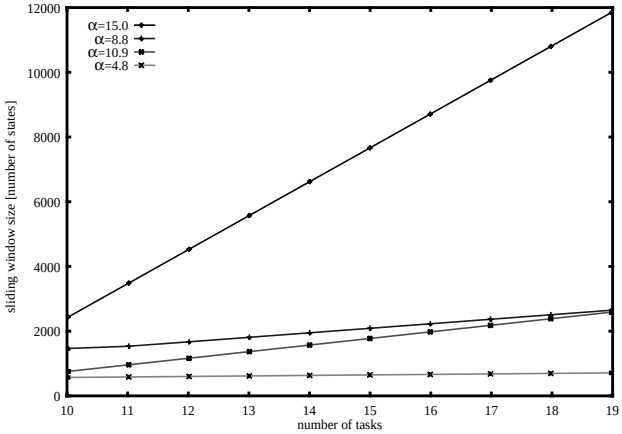

Figure 11. Experiment 1, sliding window size

In the first set of experiments we analysed the impact of the number of tasks on the process size. We considered task sets of 10 up to 19 independent tasks. P, the least common multiple of the task periods, was 360 for all task sets. We repeated the experiment four times for average values of the task periods α = 15.0, 10.9, 8.8, and 4.8 (keeping P=360). The results are shown in Figure 10. Figure 11 depicts the maximum size of the sliding window for the same task sets. As it can be seen from the diagram, the increase, both of the process size and of the sliding window, is linear. The steepness of the curves depends on the task periods (which influence the number of PMIs). It is important to notice the big difference between the process size and the maximum number of states in the sliding window. In the case for 19 tasks, for example, the process size is between 64356 and 198356 while the dimension of the sliding window varies between 373 and 11883 (16 to 172 times smaller). The reduction factor of the sliding window compared to the process size was between 15 and 1914, considering all our experiments. Because of space limitation, for the rest of paper we will concentrate only on the process size.

In the second set of experiments we analysed the impact of the application period P (the least common multiple of the task periods) on the process size. We considered 784 sets, each of 20 independent tasks. The task periods were chosen such that P takes values in the interval [1, 5040]. Figure 12 shows the variation of the average process size with the application period.

With the third set of experiments we analysed the impact of task dependencies on the process size. A task set of 200 tasks with strong dependencies (28000 arcs) among the tasks was initially created. The application period P was 360. Then 9 new task graphs were successively derived from the first one by uniformly removing dependencies between the tasks until we finally got a set of 200 independent tasks. The results are depicted in Figure 13 with a logarithmic scale for the y axis. The x axis represents the degree of dependencies among the tasks (0 for independent tasks, 9 for the initial

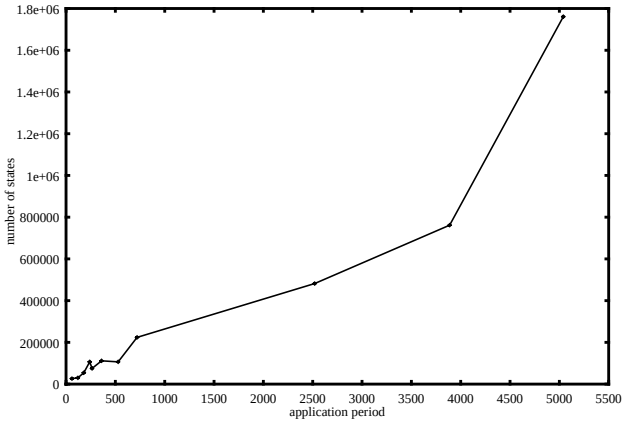

Figure 12. Experiment 2, stochastic process size

task set with the highest amount of dependencies). In this set of experiments, we used the first of the two policies defined in Section 3 for handling task graphs with dependencies.

As mentioned, the execution time for the analysis algorithm strictly depends on the process size. Therefore, we showed all the results in terms of this parameter. For the set of 200 independent tasks used in the last experiment (process size 1126517) the analysis time was 745 seconds. In the case of the same 200 tasks with strong dependencies (process size 2178) the analysis took 1.4 seconds.

Finally, we considered an example from the mobile communication area. A set of 8 tasks co-operate in order to decode the digital bursts corresponding to a GSM 900 signalling channel. In this example, there are two sources of variation in execution times. One task has both data and control intensive behaviour, which can cause pipeline hazards on the deeply pipelined DSP it runs on. Its execution time probability density is derived from the input data streams and measurements. Another task will finally implement a deciphering unit. Due to the lack of knowledge about the deciphering algorithm (its specification is not publicly available), the deciphering task execution time is considered to be uniformly distributed between an upper and a lower bound. When two channels are scheduled on the same DSP, the ratio of missed deadlines is 0 (all deadlines are met). Considering

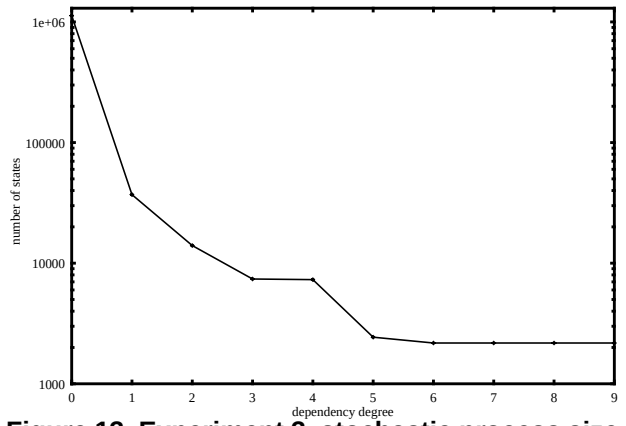

Figure 13. Experiment 3, stochastic process size

three channels assigned to the same processor, the analysis produced a ratio of missed deadlines, which was below the one enforced by the required QoS. It is important to note that using a hard real-time model with WCET, the system with three channels would result as unschedulable on the selected DSP. The underlying stochastic process for the three channels had 130 nodes and its analysis took 0.01 seconds. The small number of nodes is caused by the strong harmony among the task periods, imposed by the GSM standard.

7. Conclusions

This work proposes a method for performance analysis of task sets with probabilistically distributed execution times. The tasks are scheduled according to an arbitrary scheduling policy. The method is based on the construction of the underlying stochastic process and the analysis of this process. The stochastic process is constructed and analysed in a memory- and time-efficient way making the method applicable to large task sets. Experimental results show a very good scaling of the algorithm both in terms of memory space and execution time.

As a future work, we intend to extend our approach in order to handle sets of tasks distributed over multiprocessor systems.

References

[1] L. Abeni, G. Butazzo, *"Integrating Multimedia Applications in Hard Real-Time Systems"*, Proc. of the Real-Time Systems Symposium, 1998, pp. 4–13

[2] A. Atlas, A. Bestavros, *"Statistical Rate Monotonic Scheduling"*, Proc. of the Real-Time Systems Symposium, 1998, pp. 123–132

[3] N. C. Audsley, A. Burns, R. I. Davies, K. W. Tindell, A. J. Wellings, *"Fixed Priority Pre-emptive Scheduling: An Historical Perspective"*, Journal of Real-Time Systems, v8, 1995, pp. 173–198

[4] G. De Veciana, M. Jacome, J.H. Guo, *"Assessing Probabilistic Timing Constraints on System Performance"*, Jour. of Design Autom. for Emb. Systems, v5, 2000, pp. 61–81

[5] C. J. Fidge, *"Real-Time Schedulability Tests for Pre-emptive Multitasking"*, Journal of Real-Time Systems, v14, 1998, pp. 61–93

[6] M. Frigo, S. G. Johnson, *"FFTW: An Adaptive Software Architecture for the FFT"*, Proc. Intl. Conf. of Acoustics, Speech and Signal Processing, 1998, v. 3, p. 1381

[7] A. Kalavade, P. Moghe, *"A Tool for Performance Estimation for Networked Embedded Systems"*, Proc. of the Design Automation Conference, 1998, pp. 257–262

[8] J. P. Lehoczky, *"Real-Time Queueing Theory"*, Proc. of the Real-Time Systems Symposium, 1996, pp. 186–195

[9] M. Spuri, G. Butazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems", Journal of Real-Time Systems, v. 10, no. 2, 1996, pp. 1979–2012

[10] J. A. Stankovic, M. Spuri, M. Di Natale, G. Butazzo, *"Implications of Classical Scheduling Results for Real-Time Systems"*, IEEE Computer, June 1995, pp. 16–25

[11] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, J. W.-S. Liu, *"Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times"*, Proc. of IEEE Real-Time Techn. and Applic. Symp., 1995

[12] T. Zhou, X. Hu, E. H.-M. Sha, *"A probabilistic performance metric for Real-Time System Design"*, Proc. of the Hardw/ Softw. Co-Design Symposium, 1999, pp. 90–94