Schedulability Analysis of Applications with Stochastic Task Execution Times

SORIN MANOLACHE, PETRU ELES and ZEBO PENG
Linköping University, Sweden

In the past decade, the limitations of models considering fixed (worst case) task execution times have been acknowledged for large application classes within soft real-time systems. A more realistic model considers the tasks having varying execution times with given probability distributions. Considering such a model with specified task execution time probability distribution functions, an important performance indicator of the system is the expected deadline miss ratio of the tasks and of the task graphs. This article presents an approach for obtaining this indicator in an analytic way.

Our goal is to keep the analysis cost low, in terms of required analysis time and memory, while considering as general classes of target application models as possible. The following main assumptions have been made on the applications which are modelled as sets of task graphs: the tasks are periodic, the task execution times have given generalised probability distribution functions, the task execution deadlines are given and arbitrary, the scheduling policy can belong to practically any class of non-preemptive scheduling policies, and a designer supplied maximum number of concurrent instantiations of the same task graph is tolerated in the system.

Experiments show the efficiency of the proposed technique for monoprocessor systems.

Categories and Subject Descriptors: B.8.0 [**Hardware**]: Performance and Reliability—*General*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Performance attributes*; D.4.7 [**Software**]: Operating Systems—*Organization and Design*; G.3 [**Mathematics of Computing**]: Probability and Statistics—*Markov Processes*

General Terms: Performance, Theory

Additional Key Words and Phrases: Schedulability analysis, soft real-time systems, stochastic task execution times

1. INTRODUCTION

The design process of embedded real-time systems typically starts from an informal specification together with a set of constraints. This initial informal specification is then captured as a more rigorous model, formulated in one or several modelling languages [Powell Douglass et al. 1996; Sarkar et al. 1995]. During the system level design space exploration phase, different architecture, mapping and scheduling alternatives are assessed in order to meet the design requirements and possibly optimise the values of certain quality or cost indicators [De Micheli and Gupta 1997; Beck and Siewiorek 1998; Lee et al. 1999; Potkonjak and Rabaey 1999; Wolf

1994; Ernst 1998]. The existence of accurate, fast and flexible automatic tools for performance estimation in every design phase is of capital importance for cutting down design process iterations, time, and cost.

Performance estimation tools used in the early design stages do not benefit from detailed information regarding the design and, hence, can provide only rough estimates of the final performance of the system to be implemented. In the later design stages, before proceeding to the synthesis and/or integration of the software, hardware and communication components of the system, it is important that the system performance, as predicted by the estimation tools based on the now more detailed system model, is accurate with respect to the real performance of the manufactured and deployed product. An accurate performance estimation at this stage would leverage the design process by allowing the designer to efficiently explore several design alternatives. Such a performance estimation algorithm is the topic of this article.

Historically, real-time system research emerged from the need to understand, design, predict, and analyse safety critical applications such as plant control and aircraft control, to name a few [Liu and Layland 1973; Kopetz 1997; Buttazzo 1997]. Therefore, the community focused on hard real-time systems, where breaking a timeliness requirement is intolerable as it may lead to catastrophic consequences. In such systems, if not all deadlines are guaranteed to be met, the system is said to be unschedulable. The schedulability analysis of hard real-time systems answers the question whether the system is schedulable or not [Audsley et al. 1995; Fidge 1998; Balarin et al. 1998; Stankovic and Ramamritham 1993]. The only way to ensure that no real-time requirement is broken is to make conservative assumptions about the systems, such as, for example, that every task instantiation is assumed to run for a worst case time interval, called the worst case execution time (WCET) of the task.

Applications belonging to a different class of real-time systems, the soft real-time systems, are considered to still correctly function even if some timeliness requirements are occasionally broken, as this leads to a tolerable reduction of the service quality [Buttazzo et al. 1998]. For example, multimedia applications, like JPEG and MPEG encoding, audio encoding, etc., exhibit this property [Abeni and Buttazzo 1998]. In this context, we would be interested in the *degree* to which the system meets its timeliness requirements rather than in the binary answer provided by the hard real-time system schedulability analysis. In our work, this degree is expressed as the *expected ratio of missed deadlines* for each task graph or task.

The execution time of a task is a function of application-dependent, platform-dependent, and environment-dependent factors. The amount of input data to be processed in each task instantiation as well as its value and type (pattern, configuration) are application-dependent factors. The type of processing unit that executes a task is a platform-dependent factor influencing the task execution time. If the time needed for communication with the environment (database lookups, for example) is to be considered as a part of the task execution time, then network load is an example of an environmental factor influencing the task execution time.

Input data amount and type may vary, as for example is the case for differently coded MPEG frames. Platform-dependent characteristics, like cache memory be-

haviour, pipeline stalls, write buffer queues, may also introduce a variation in the task execution time. Thus, obviously, all of the enumerated factors influencing the task execution time may vary. Therefore, a model considering variable execution time would be more realistic as the one considering fixed, worst case execution times. In a more general model of task execution times, arbitrary task execution time probability distribution functions (ETPDFs) are considered. These distributions can be extracted from performance models [van Gemund 1996] by means of analytic methods or simulation and profiling [van Gemund 2003; Gautama and van Gemund 2000; Gautama 1998]. Obviously, the fixed task execution time model is a particular case of such a stochastic one.

An approach based on a worst case execution time model would implement the task on an expensive system which guarantees the imposed deadline for the worst case situation. This situation, however, will usually occur with a very small probability. If the nature of the application is such that a certain percentage of deadline misses is affordable, a cheaper system, which still fulfils the imposed quality of service, can be designed.

This article proposes an algorithm for obtaining the expected ratio of missed deadlines per task graph or task, given a set of task graphs with the following assumptions: the tasks are periodic, the task execution times have given generalised probability distribution functions, the task execution deadlines are given and arbitrary, the scheduling policy belongs to practically any class of non-preemptive scheduling policies, and a designer supplied maximum number of concurrent instantiations of the same task graph is tolerated in the system.

The sequel of the article is structured as follows. The next section surveys related work and comparatively comments on our approach. Section 3 captures the problem formulation. Section 4 discusses our algorithm in detail. Section 5 presents experiments and the last section draws the conclusions.

2. RELATED WORK

Atlas and Bestavros [Atlas and Bestavros 1998] extend the classical rate monotonic scheduling policy [Liu and Layland 1973] 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 Buttazzo's work [Abeni and Butazzo 1999] 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.

Tia *et al.* [Tia et al. 1995] assume a task model composed of independent tasks. Two methods for performance analysis are given. One of them is just an estimate and is 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 particular model.

Zhou *et al.* [Zhou et al. 1999] and Hu *et al.* [Hu et al. 2001] root their work in Tia's. However, they do not intend to give per-task guarantees, but characterise 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 very small task sets due to complexity reasons.

De Veciana *et al.* [de Veciana et al. 2000] address a different type of problem. Having a task graph and an imposed deadline, their goal is to determine the path that has the highest probability to violate the deadline. In this case, the problem is reduced to a non-linear optimisation problem by using an approximation of the convolution of the probability densities.

Lehoczky [Lehoczky 1996] 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 underlying stochastic process weakly converges to a reflected Brownian motion with drift. As far as we are aware, the heavy traffic theory fails yet to smoothly apply to real-time systems. Not only that there are cases when such a reflected Brownian motion with drift limit does not exist, as shown by Dai and Wang [Dai and Wang 1993], but also the heavy traffic phenomenon is observed only for processor loads close to 1, leading to very long (infinite) queues of ready tasks and implicitly to systems with very large latency. This aspect makes the heavy traffic phenomenon undesirable in real-time systems.

Other researchers, such as Kleinberg *et al.* [Kleinberg et al. 2000] and Goel and Indyk [Goel and Indyk 1999], apply approximate solutions to problems exhibiting stochastic behaviour but in the context of load balancing, bin packing and knapsack problems. Moreover, the probability distributions they consider are limited to a few very particular cases.

Kim and Shin [Kim and Shin 1996] considered applications implemented on multiprocessors and modelled them as queueing networks. They restricted the task execution times to exponentially distributed ones, which reduces the complexity of the analysis. The tasks were considered to be scheduled according to a particular policy, namely first-come-first-served (FCFS). The underlying mathematical model is then the appealing continuous time Markov chain.

Díaz *et al.* [Díaz et al. 2002] derive the expected deadline miss ratio from the probability distribution function of the response time of a task. The response time is computed based on the system-level backlog at the beginning of each hyperperiod, i.e. the residual execution times of the jobs at those time moments. The stochastic process of the system-level backlog is Markovian and its stationary solution can be computed. Díaz *et al.* consider only sets of independent tasks and the task execution times may assume values only over discrete sets. In their approach, complexity is mastered by trimming the transition probability matrix of the underlying Markov chain or by deploying iterative methods, both at the expense of result accuracy. According to the published results, the method is exercised only on extremely small task sets.

Kalavade and Moghé [Kalavade and Moghé 1998] consider task graphs where the

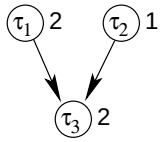


Fig. 1. Application example

task execution times are arbitrarily distributed over discrete sets. Their analysis is based on Markovian stochastic processes too. Each state in the process is characterised 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.

Our work is mostly related to the ones of Zhou *et al.* [Zhou et al. 1999], Hu *et al.* [Hu et al. 2001], Kalavade and Moghé [Kalavade and Moghé 1998] and Díaz *et al* [Díaz et al. 2002]. It differs mostly by considering less restricted application classes. As opposed to Kalavade and Moghé's work and to Díaz *et al.*'s work, we consider *continuous* ETPDFs. In addition to Díaz *et al.*'s approach, we consider task sets with dependencies among tasks. Also, we accept a much larger class of scheduling policies than the fixed priority ones considered by Zhou and Hu. Moreover, our original way of concurrently constructing and analysing the underlying stochastic process, while keeping only the needed stochastic process states in memory, allows us to consider larger applications [Manolache et al. 2001].

3. NOTATION AND PROBLEM FORMULATION

This section introduces the notation used throughout the article and gives the problem formulation.

3.1 Notation

Let $T = \{\tau_1, \tau_2, \ldots, \tau_N\}$ be a set of N *tasks* and $G = \{G_1, G_2, \ldots, G_h\}$ denote h *task graphs*. A task graph $G_i = (V_i, E_i \subset V_i \times V_i)$ is a directed acyclic graph (DAG) whose set of vertices V_i is a non-empty subset of the set of tasks T. The sets V_i, $1 \leq i \leq h$, form a partition of T. There exists a directed edge $(\tau_j, \tau_k) \in E_i$ if and only if the task τ_k is dependent on the task τ_j. This dependency imposes that the task τ_k is executed only after the task τ_j has completed execution.

Let $G_i = (V_i, E_i)$ and $\tau_k \in V_i$. Then let ${}^\circ\tau_k = \{\tau_j : (\tau_j, \tau_k) \in E_i\}$ denote the set of predecessor tasks of the task τ_k. Similarly, let $\tau_k^\circ = \{\tau_j : (\tau_k, \tau_j) \in E_i\}$ denote the set of successor tasks of the task τ_k. If ${}^\circ\tau_k = \varnothing$ then task τ_k is a *root*. If $\tau_k^\circ = \varnothing$ then task τ_k is a *leaf*.

Let $\Pi = \{\pi_i \in \mathbb{N} : \tau_i \in T\}$ denote the set of *task periods*, or task inter-arrival times, where π_i is the period of task τ_i. Instantiation $u \in \mathbb{N}$ of task τ_i demands execution (is released) at time moment $u \cdot \pi_i$. The period π_i of any task τ_i is assumed to be a common multiple of all periods of its predecessor tasks (π_j divides π_i, where $\tau_j \in {}^\circ\tau_i$). Let k_{ij} denote $\frac{\pi_i}{\pi_j}$, $\tau_j \in {}^\circ\tau_i$. Instantiation $u \in \mathbb{N}$ of task τ_i may start executing only if instantiations $u \cdot k_{ij}, u \cdot k_{ij} + 1, \ldots, (u+1) \cdot k_{ij} - 1$ of tasks τ_j, $\forall \tau_j \in {}^\circ\tau_i$, have completed.

Let us consider the example in Figure 1. The circles indicate the tasks, the num-

bers outside the circles indicate the task periods. In this example, instantiation 0 of task τ_1 and instantiations 0 and 1 of task τ_2 have to complete in order instantiation 0 of task τ_3 to be ready to run. Instantiation 1 of task τ_1 and instantiations 2 and 3 of task τ_2 have to complete in order instantiation 1 of task τ_3 to be ready to run. However, there is no execution precedence constraint between instantiation 0 of task τ_1 and instantiations 0 and 1 of task τ_2 on one hand and instantiation 1 of task τ_3 on the other hand.

π_{G_j} will denote the period of the task graph G_j and π_{G_j} is equal to the least common multiple of all π_i, where π_i is the period of τ_i and $\tau_i \in V_j$.

The model, where task periods are integer multiples of the periods of predecessor tasks, is more general than the model assuming equal task periods for tasks in the same task graph. This is appropriate, for instance, when modelling protocol stacks. For example, let us consider a part of baseband processing on the GSM radio interface [Mouly and Pautet 1992]. A data frame is assembled out of 4 radio bursts. One task implements the decoding of radio bursts. Each time a burst is decoded, the result is sent to the frame assembling task. Once the frame assembling task gets all the needed data, that is every 4 invokations of the burst decoding task, the frame assembling task is invoked. This way of modelling is more modular and natural than a model assuming equal task periods which would have crammed the four invokations of the radio burst decoding task in one task. We think that more relaxed models, with regard to relations between task periods, are not necessary, as such applications would be more costly to implement and are unlikely to appear in common engineering practice.

The real-time requirements are expressed in terms of relative deadlines. Let $\Delta_T = \{\delta_i \in \mathbb{N} : \tau_i \in T\}$ denote the set of *task deadlines*. δ_i is the deadline of task τ_i. Let $\Delta_G = \{\delta_{G_j} \in \mathbb{N} : 1 \leq j \leq h\}$ denote the set of task graph deadlines, where δ_{G_j} is the deadline of task graph G_j. If there is at least one task $\tau_i \in V_j$ that has missed its deadline δ_i, then the entire graph G_j missed its deadline.

If $D_i(t)$ denotes the number of missed deadlines of the task τ_i (or of the task graph G_i) over a time span t and $A_i(t)$ denotes the number of instantiations of task τ_i (task graph G_i) over the same time span, then $\lim_{t\to\infty} \frac{D_i(t)}{A_i(t)}$ denotes the *expected deadline miss ratio* of task τ_i (task graph G_i).

Let Ex_i denote an execution time of an instantiation of the task τ_i. Let $ET = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_N\}$ denote a set of N *execution time probability density functions*. ϵ_i is the probability density of the execution time of task τ_i. The execution times are assumed to be statistically independent. All the tasks are assumed to execute on the same processor. In this case, the inter-task communication time is comprised in the task execution time.

If a task misses its deadline, the real-time operating system takes a decision based on a designer-supplied *late task policy*. Let $Bounds = \{b_1, b_2, \ldots, b_h\}$ be a set of h integers greater than 0. The late task policy specifies that at most b_i instantiations of the task graph G_i are allowed in the system at any time. If an instantiation of graph G_i demands execution when b_i instantiations already exist in the system, the instantiation with the earliest arrival time is discarded (eliminated) from the system. An alternative to this late task policy will be discussed in Section 5.5

In the common case of more than one task mapped on the same processor, the

designer has to decide on a *scheduling policy*. Such a scheduling policy has to be able to unambiguously determine the running task at any time on that processor.

Let an *event* denote a task arrival or discarding. In order to be acceptable in the context described in this article, a scheduling policy is assumed to preserve the sorting of tasks according to their execution priority between consecutive events (the priorities are allowed to change in time, but the sorting of tasks according to their priorities is allowed to change only at event times). All practically used priority based scheduling policies [Liu and Layland 1973; Buttazzo 1997; Fidge 1998; Audsley et al. 1995], both with static priority assignment (rate monotonic, deadline monotonic) and with dynamic assignment (earlier deadline first (EDF), least laxity first (LLF)), fulfill this requirement.

The scheduling policy is nevertheless restricted to non-preemptive scheduling. This limitation is briefly discussed in Section 4.1.

3.2 Problem formulation

3.2.1 *Input.* The following data is given as an input to the analysis procedure:

—The set of task graphs G,

—The set of task periods Π,

—The set of task deadlines Δ_T and the set of task graph deadlines Δ_G,

—The set of execution time probability density functions ET,

—The late task policy *Bounds*, and

—The scheduling policy.

3.2.2 *Output.* The result of the analysis is the set $Missed_T = \{m_{\tau_1}, m_{\tau_2}, \ldots, m_{\tau_N}\}$ of expected deadline miss ratios for each task and the set $Missed_G = \{m_{G_1}, m_{G_2}, \ldots, m_{G_h}\}$ of expected deadline miss ratios for each task graph.

4. ANALYSIS ALGORITHM

The goal of the analysis is to obtain the expected deadline miss ratios of the tasks and task graphs. These can be derived from the behaviour of the system. The behaviour is defined as the evolution of the system through a *state space* in time. A *state* of the system is given by the values of a set of variables that characterise the system. Such variables may be the currently running task, the set of ready tasks, the current time and the start time of the current task, etc.

Due to the considered periodic task model, the task arrival times are deterministically known. However, because of the stochastic task execution times, the completion times and implicitly the running task at an arbitrary time instant or the state of the system at that instant cannot be deterministically predicted. The mathematical abstraction best suited to describe and analyse such a system with random character is the stochastic process.

In this section, we first sketch the stochastic process construction and analysis procedure based on a simplified example. Then the memory efficient construction of the stochastic process underlying the application is detailed. Third, the algorithm is refined in order to handle multiple concurrently active instantiations of the same task graph. Finally, the complete algorithm is presented.

4.1 The underlying stochastic process

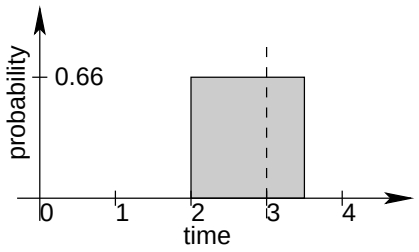
Let us define LCM as the least common multiple of the task periods. For simplicity of the exposition, we first assume that at most one instantiation of each task graph is tolerated in the system at the same time ($b_i = 1$, $\forall 1 \leq i \leq h$). In this case, all the late tasks are discarded at the time moments $LCM, 2 \cdot LCM, \ldots, k \cdot LCM, \ldots$ because at these moments new instantiations of all tasks arrive. The system behaves at these time moments as if it has just been started. The time moments $k \cdot LCM$, $k \in \mathbb{N}$ are called renewal points. Regardless of the chosen definition of the state space of the system, the system states at the renewal points are equivalent to the initial state which is unique and deterministically known. Thus, the behaviour of the system over the intervals $[k \cdot LCM, (k+1) \cdot LCM)$, $k \in \mathbb{N}$, is statistically equivalent to the behaviour over the time interval $[0, LCM)$. Therefore, in the case when $b_i = 1$, $1 \leq i \leq h$, it is sufficient to analyse the system solely over the time interval $[0, LCM)$.

One could choose the following state space definition: $S = \{(\tau, W, t) : \tau \in T, W \in$ set of all multisets of $T, t \in \mathbb{R}\}$, where τ represents the currently running task, W stands for the multiset of ready tasks at the start time of the running task, and t represents the start time of the currently running task. A state change occurs at the time moments when the scheduler has to decide on the next task to run. This happens

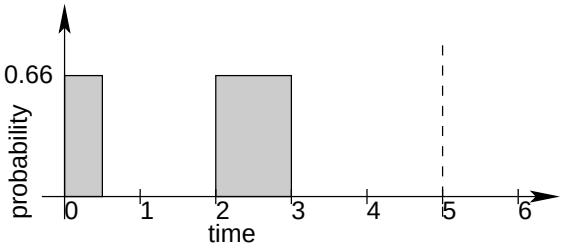
— when a task completes its execution, or
— when a task arrives and the processor is idling, or
— when the running task graph has to be discarded.

The point we would like to make is that, by choosing this state space, the information provided by a state $s_i = (\tau_i, W_i, t_i)$, together with the current time, is sufficient to determine the next system state $s_j = (\tau_j, W_j, t_j)$. The time moment when the system entered state s_i, namely t_i, is included in s_i. Because of the deterministic arrival times of tasks, based on the time moments t_j and on t_i, we can derive the multiset of tasks that arrived in the interval $(t_i, t_j]$. The multiset of ready tasks at time moment t_i, namely W_i, is also known. We also know that τ_i is not preempted between t_i and t_j. Therefore, the multiset of ready tasks at time moment t_j, prior to choosing the new task to run, is the union of W_i and the tasks arrived during the interval $(t_i, t_j]$. Based on this multiset and on the time t_j, the scheduler is able to predictably choose the new task to run. Hence, in general, knowing a current state s and the time moment t when a transition out of state s occurs, the next state s' is unambiguously determined.

The following example is used throughout this subsection in order to discuss the construction of the stochastic process. The system consists of one processor and the following application: $G = \{(\{\tau_1\}, \varnothing), (\{\tau_2\}, \varnothing)\}$, $\Pi = \{3, 5\}$, i.e. a set of two independent tasks with corresponding periods 3 and 5. The tasks are scheduled according to a non-preemptive EDF scheduling policy [Liu and Layland 1973]. In this case, LCM, the least common multiple of the task periods is 15. For simplicity, in this example it is assumed that the relative deadlines equal the corresponding periods ($\delta_i = \pi_i$). The ETPDFs of the two tasks are depicted in Figure 2. Note that ϵ_1 contains execution times larger than the deadline δ_1.



(a) ϵ_1



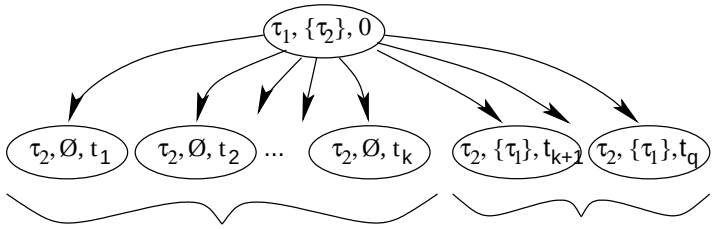
(b) ϵ_2

Fig. 2. ETPDFs of tasks τ_1 (ϵ_1) and τ_2 (ϵ_2)

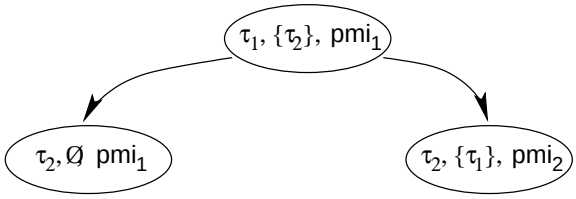
Let us assume a state representation like the one introduced above: each process state contains the identity of the currently running task, its start time and the multiset of ready task at the start time of the currently running one. For our example application, the initial state is $(\tau_1, \{\tau_2\}, 0)$, i.e. task τ_1 is running, it has started to run at time moment 0 and task τ_2 is ready to run, as shown in Figure 3(a). $t_1, t_2, \ldots, t_q$ in the figure are possible finishing times for the task τ_1 and, implicitly, possible starting times of the waiting instantiation of task τ_2. The number of next states equals the number of possible execution times of the running task in the current state. In general, because the ETPDFs are continuous, the set of state transition moments form a dense set in $\mathbb{R}$ leading to an underlying stochastic process theoretically of uncountable state space. In practice, the stochastic process is extremely large, depending on the discretisation resolution of the ETPDFs. Even in the case when the task execution time probabilities are distributed over a discrete set, the resulting underlying process becomes prohibitively large and practically impossible to solve.

In order to avoid the explosion of the underlying stochastic process, 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.

As a first step to the analysis, the interval $[0, LCM)$ is partitioned in disjunct intervals, the so-called *priority monotonicity intervals (PMI)*. A PMI is delimited



(a) Individual task completion times



(b) Intervals containing task completion times

Fig. 3. State encoding

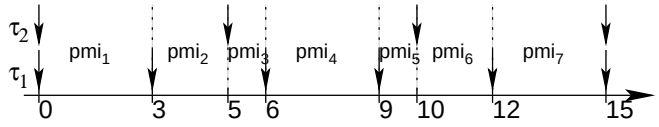


Fig. 4. Priority monotonicity intervals

by task arrival times and task execution deadlines. Figure 4 depicts the PMIs for the example above. 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. This allows the scheduler to predictably choose the next task to run regardless of the completion time within a PMI of the previously running task. All the widely used scheduling policies we are aware of (rate monotonic (RM), EDF, first come first served (FCFS), LLF, etc.) exhibit this property.

Consider a state s characterised by (τ_i, W, t): τ_i is the currently running task, it has been started at time t, and W is the multiset of ready tasks. Let us consider two next states derived from s: s_1 characterised by (τ_j, W_1, t_1) and s_2 by (τ_k, W_2, t_2). Let t_1 and t_2 belong to the same PMI. This means that no task instantiation has arrived or been discarded in the time interval between t_1 and t_2, and the relative priorities of the tasks inside the set W have not changed between t_1 and t_2. Thus, $\tau_j = \tau_k =$ the highest priority task in the multiset W, and $W_1 = W_2 = W \backslash \{\tau_j\}$. It follows that all states derived from state s that have their time t belonging to the same PMI have an identical currently running task and identical sets of ready

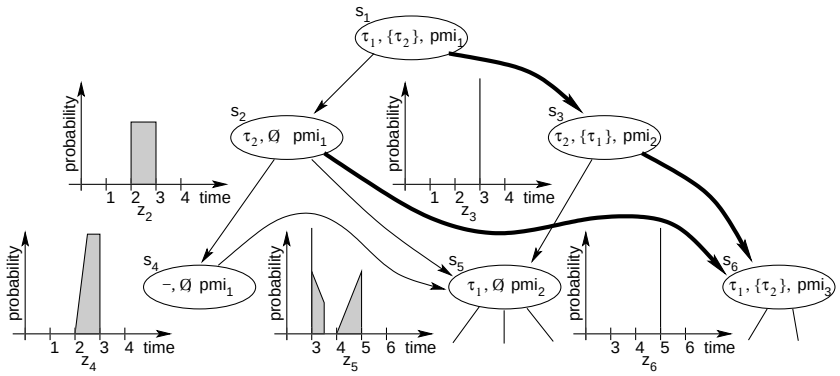


Fig. 5. Stochastic process example

tasks. 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 task that runs in state s is spanning over.

We propose a representation in which a stochastic process state is a triplet (τ, W, pmi), where τ is the running task, W the multiset of ready tasks, and pmi is the PMI containing the start time of the running task. In our example, the execution time of task τ_1 (which is in the interval $[2, 3.5]$, as shown in Figure 2(a)) is spanning over the PMIs pmi_1— $[0, 3)$—and pmi_2—$[3, 5)$. Thus, there are only two possible states emerging from the initial state, as shown in Figure 3(b).

Figure 5 depicts a part of the stochastic process constructed for our example. The initial state is $s_1 : (\tau_1, \{\tau_2\}, pmi_1)$. The first field indicates that an instantiation of task τ_1 is running. The second field indicates that an instantiation of task τ_2 is ready to execute. The third field shows the current PMI (pmi_1—$[0, 3)$). If the instantiation of task τ_1 does not complete until time moment 3, then it will be discarded. The state s_1 has two possible next states. The first one is state $s_2 : (\tau_2, \varnothing, pmi_1)$ and corresponds to the case when the τ_1 completes before time moment 3. The second one is state $s_3 : (\tau_2, \{\tau_1\}, pmi_2)$ and corresponds to the case when τ_1 was discarded at time moment 3. State s_2 indicates that an instantiation of task τ_2 is running (it is the instance that was waiting in state s_1), that the PMI is pmi_1—$[0, 3)$—and that no task is waiting. Consider state s_2 to be the new current state. Then the next states could be state $s_4 : (-, \varnothing, pmi_1)$ (task τ_2 completed before time moment 3 and the processor is idle), state $s_5 : (\tau_1, \varnothing, pmi_2)$ (task τ_2 completed at a time moment sometime between 3 and 5), or state $s_6 : (\tau_1, \{\tau_2\}, pmi_3)$ (the execution of task τ_2 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 $[0, LCM)$, i.e. $[0, 15)$ have been visited.

Let $\mathcal{P}_i$ denote the set of predecessor states of a state s_i, i.e. 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 directly be reached from state s_i. Let Z_i denote the time

when state s_i is entered. State s_i can be reached from any of its predecessor states $s_j \in \mathcal{P}_i$. Therefore, the probability $\mathsf{P}(Z_i \leq t)$ that state s_i is entered before time t is a weighted sum over j of probabilities that the transitions $s_j \rightarrow s_i$, $s_j \in \mathcal{P}_i$, occur before time t. The weights are equal to the probability $\mathsf{P}(s_j)$ that the system was in state s_j prior to the transition. Formally, $\mathsf{P}(Z_i \leq t) = \sum_{j \in \mathcal{P}_i} \mathsf{P}(Z_{ji} \leq t | s_j) \cdot \mathsf{P}(s_j)$, where Z_{ji} is the time of transition $s_j \rightarrow s_i$. Let us focus on Z_{ji}, the time of transition $s_j \rightarrow s_i$. If the state transition occurs because the processor is idle and a new task arrives or because the running task graph has to be discarded, the time of the transition is deterministically known as task arrivals and deadlines have fixed times. If, however, the cause of the state transition is a task completion, the time Z_{ji} is equal to to $Z_j + Ex_\tau$, where task τ is the task which runs in state s_j and whose completion triggers the state transition. Because Z_{ji} is a sum involving the random variable Ex_τ, Z_{ji} too is a random variable. Its probability density function, is computed as the convolution $z_j * \epsilon_\tau = \int_0^\infty z_j(t-x) \cdot \epsilon_\tau(x) dx$ of the probability density functions of the terms.

Let us illustrate the above based on the example depicted in Figure 5. z_2, z_3, z_4, z_5, and z_6 are the probability density functions of Z_2, Z_3, Z_4, Z_5, and Z_6 respectively. They are shown in Figure 5 to the left of their corresponding states $s_2, s_3, \ldots, s_6$. The transition from state s_4 to state s_5 occurs at a precisely known time instant, time 3, at which a new instantiation of task τ_1 arrives. Therefore, z_5 will contain a scaled Dirac impulse at the beginning of the corresponding PMI. The scaling coefficient equals the probability of being in state s_4 (the integral of z_4, i.e. the shaded surface below the z_4 curve). The probability density function z_5 results from the superposition of $z_2 * \epsilon_2$ (because task τ_2 runs in state s_2) with $z_3 * \epsilon_2$ (because task τ_2 runs in state s_3 too) and with the aforementioned scaled Dirac impulse over pmi_2, i.e. over the time 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 deadline miss of a task instantiation (the thick arrows in Figure 5, in our case). 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, let us consider the transition $s_2 \rightarrow s_6$. The system enters state s_2 at a time whose probability density is given by z_2. The system takes the transition $s_2 \rightarrow s_6$ when the attempted completion time of τ_2 (running in s_2) exceeds 5. The completion time is the sum of the starting time of τ_2 (whose probability density is given by z_2) and the execution time of τ_2 (whose probability density is given by ϵ_2). Hence, the probability density of the completion time of τ_2 is given by the convolution $z_2 * \epsilon_2$ of the above mentioned densities. Once this density is computed, the probability of the completion time being larger than 5 is easily computed by integrating the result of the convolution over the interval $[5, \infty)$. If τ_2 in s_2 completes its execution at some time $t \in [3, 5)$, then the state transition $s_2 \rightarrow s_5$ occurs (see Figure 5). The probability of this transition is computed by integrating $z_2 * \epsilon_2$ over the interval $[3, 5)$.

As can be seen, by using the PMI approach, some process states have more than one incident arc, thus keeping the 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 3(a)).

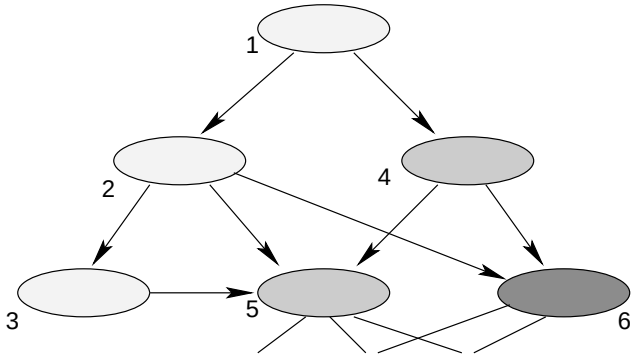


Fig. 6. State selection order

The non-preemption limitation could, in principle, be overcome if we extended the information stored in the state of the underlying stochastic process. Namely, the *residual* run time probability distribution function of a task instantiation, i.e. the PDF of the time a preempted instantion still has to run, has to be stored in the stochastic process state. This would several times multiply the memory requirements of the analysis. Additionally, preemption would increase the possible behavious of the system and, consequently, the number of states of its underlying stochastic process.

Because the number of states grows rapidly even with our state reduction approach and 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 master memory complexity is to perform the stochastic process construction and analysis simultaneously. As each arrow updates the time probability density z 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 5, the construction starts with state s_1. After its next states (s_2 and s_3) are created, their corresponding transition probabilities determined and the possible deadline miss probabilities accounted for, state s_1 can be removed from memory. Next, one of the states s_2 and s_3 is taken as current state, let us consider state s_2. The procedure is repeated, states s_4, s_5 and s_6 are created and state s_2 removed. At this moment, one would think that any of the states s_3, s_4, s_5, and s_6 can be selected for continuation of the analysis. However, this is not the case, as not all the information needed in order to handle states s_5 and s_6 are computed. More exactly, the arcs emerging from states s_3 and s_4 have not yet been created. Thus, only states s_3 and s_4 are possible alternatives for the continuation of the analysis in topological order. The next section discusses the criteria for selection of the correct state to continue with.

4.2 Memory efficient analysis method

As shown in the example in Section 4.1, only a sliding window of states is simultaneously kept in memory. All states belonging to the sliding window are stored in a priority queue. Once a state is extracted from this queue and its information

processed, it is eliminated from the memory. 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 such 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 task instantiation 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 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 6 illustrates the algorithm on the example given in Section 4.1 (Figure 5). 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.

4.3 Flexible discarding

The examples considered so far dealt with applications where at most one active instance of each task graph is allowed at any moment of time ($b_i = 1$, $1 \leq i \leq h$).

In order to illustrate the construction of the stochastic process in the case $b_i > 1$, when several instantiations of a task graph G_i may exist at the same time in the system, let us consider an application consisting of two independent tasks, τ_1 and τ_2, with periods 2 and 4 respectively. $LCM = 4$ in this case. The tasks are scheduled according to a rate monotonic (RM) policy [Liu and Layland 1973]. At most one active instantiation of τ_1 is tolerated in the system at a certain time ($b_1 = 1$) and at most two concurrently active instantiations of τ_2 are tolerated in the system ($b_2 = 2$).

Figure 7 depicts a part of the stochastic process underlying this example. It is constructed using the procedure sketched in Sections 4.1 and 4.2. The state indexes show the order in which the states were analysed (extracted from the priority queue mentioned in Section 4.2).

Let us consider state $s_6 = (\tau_2, \varnothing, [2, 4))$, i.e. the instantiation of τ_2 that arrives at time moment 0 has been started at a moment inside the PMI $[2, 4)$ and there have not been any ready tasks at the start time of τ_2. Let us assume that the finishing time of τ_2 lies past the $LCM = 4$. At time moment 4, a new instantiation of τ_2 arrives and the running instantiation is *not* discarded, as $b_2 = 2$. On one hand, if the finishing time of the running instantiation belongs to the interval $[6, 8)$, the system performs the transition $s_6 \rightarrow s_{14}$ (Figure 7). If, on the other hand, the running instantiation attempts to run past the time moment 8, then at this time moment a *third* instantiation of τ_2 would require service from the system and, therefore, the running task (the oldest instantiation of τ_2) is eliminated from the system. The transition $s_6 \rightarrow s_{19}$ in the stochastic process in Figure 7 corresponds to this latter case. We observe that when a task execution spans beyond the time moment LCM, the resulting state is not unique. The system does not behave as if it has just been restarted at time moment LCM, and, therefore, the intervals $[k \cdot LCM, (k + 1) \cdot LCM)$, $k \in \mathbb{N}$, are not statistically equivalent to

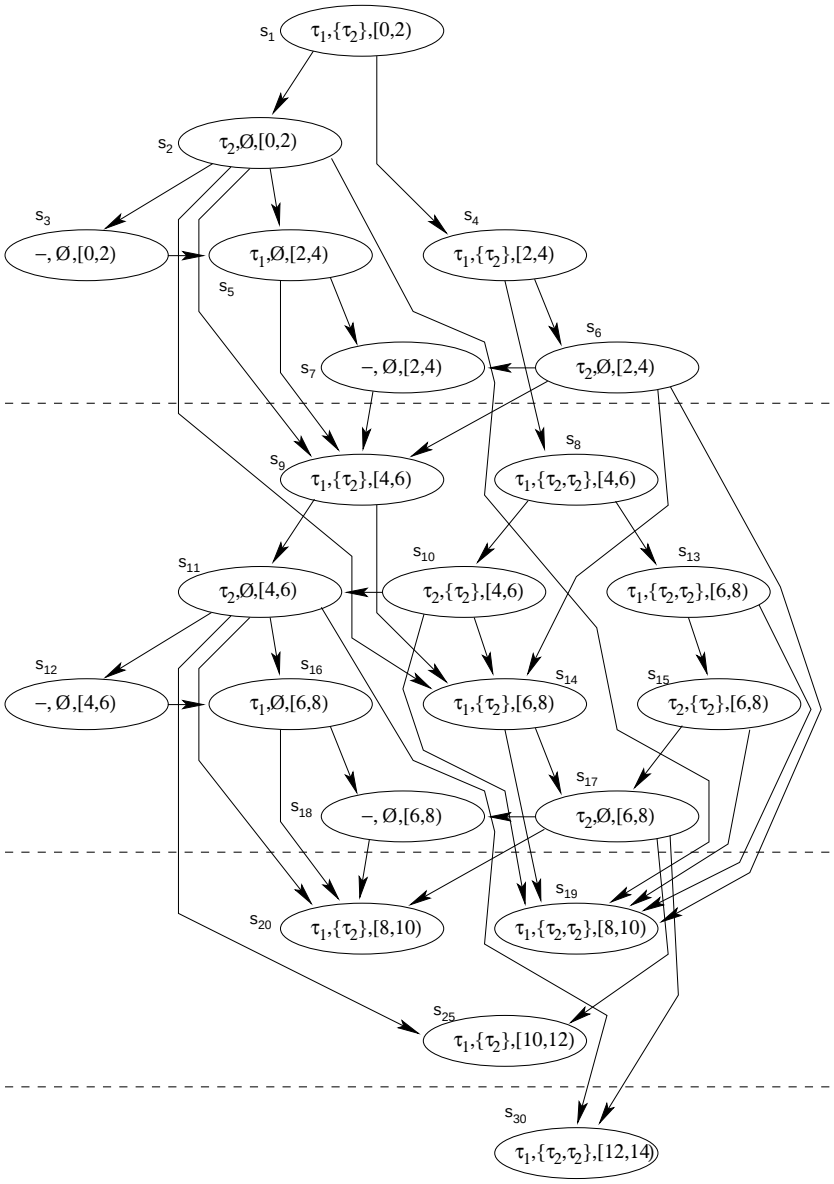


Fig. 7. Part of the stochastic process underlying the example application

the interval $[0, LCM)$. Hence, it is not sufficient to analyse the system over the interval $[0, LCM)$ but rather over several consecutive intervals of length LCM.

Let an interval of the form $[k \cdot LCM, (k+1) \cdot LCM)$ be called the *hyperperiod* k and denoted H_k. $H_{k'}$ is a *lower* hyperperiod than H_k ($H_{k'} < H_k$) if $k' < k$. Consequently, H_k is a *higher* hyperperiod than $H_{k'}$ ($H_k > H_{k'}$) if $k > k'$.

For brevity, we say that a state s belongs to a hyperperiod k (denoted $s \in H_k$) if its PMI field is a subinterval of the hyperperiod k. In our example, three hyper-

periods are considered, $H_0 = [0, 4)$, $H_1 = [4, 8)$, and $H_2 = [8, 12)$. In the stochastic process in Figure 7, $s_1, s_2, \ldots, s_7 \in H_0$, $s_8, s_9, \ldots, s_{18} \in H_1$, and $s_{19}, s_{20}, s_{25} \in H_2$ (note that not all states have been depicted in Figure 7).

In general, let us consider a state s and let $\mathcal{P}_s$ be the set of its predecessor states. Let k denote the *order* of the state s defined as the lowest hyperperiod of the states in $\mathcal{P}_s$ ($k = \min\{j : s' \in H_j, s' \in \mathcal{P}_s\}$). If $s \in H_k$ and s is of order k' and $k' < k$, then s is a *back state*. In our example, s_8, s_9, s_{14}, and s_{19} are back states of order 0, while s_{20}, s_{25} and s_{30} are back states of order 1.

Obviously, there cannot be any transition from a state belonging to a hyperperiod H to a state belonging to a lower hyperperiod than H ($s \rightarrow s', s \in H_k, s' \in H_{k'} \Rightarrow H_k \leq H_{k'}$). Consequently, the set $\mathcal{S}$ of all states belonging to hyperperiods greater or equal to H_k can be constructed from the back states of an order smaller than k. We say that $\mathcal{S}$ is *generated* by the aforementioned back states. For example, the set of all states $s_8, s_9, \ldots, s_{18} \in H_1$ can be derived from the back states s_8, s_9, and s_{14} of order 0. The intuition behind this is that back states are inheriting all the needed information across the border between hyperperiods.

Before continuing our discussion, we have to introduce the notion of *similarity* between states. We say that two states s_i and s_j are similar ($s_i \sim s_j$) if all the following conditions are satisfied:

(1) The task which is running in s_i and the one in s_j are the same,
(2) The multiset of ready tasks in s_i and the one in s_j are the same,
(3) The PMIs in the two states differ only by a multiple of LCM, and
(4) $z_i = z_j$ (z_i is the probability density function of the times when the system takes a transition to s_i).

Let us consider the construction and analysis of the stochastic process, as described in Sections 4.1 and 4.2. Let us consider the moment x, when the last state belonging to a certain hyperperiod H_k has been eliminated from the sliding window. R_k is the set of back states stored in the sliding window at the moment x. Let the analysis proceed with the states of the hyperperiod H_{k+1} and let us consider the moment y when the last state belonging to H_{k+1} has been eliminated from the sliding window. Let R_{k+1} be the set of back states stored in the sliding window at moment y.

If the sets R_k and R_{k+1} contain pairwise similar states, then it is guaranteed that R_k and R_{k+1} generate identical stochastic processes during the rest of the analysis procedure (as stated, at a certain moment the set of back states unambiguously determines the rest of the stochastic process). In our example, $R_0 = \{s_8, s_9, s_{14}, s_{19}\}$ and $R_1 = \{s_{19}, s_{20}, s_{25}, s_{30}\}$. If $s_8 \sim s_{19}$, $s_9 \sim s_{20}$, $s_{14} \sim s_{25}$, and $s_{19} \sim s_{30}$ then the analysis process may stop as it reached convergence.

Consequently, the analysis proceeds by considering states of consecutive hyperperiods until the information captured by the back states in the sliding window does not change any more. Whenever the underlying stochastic process has a steady state, this steady state is guaranteed to be found.

4.4 Construction and analysis algorithm

The analysis is performed in two phases:

```
divide [0, LCM) in PMIs;
pmi_no = number of PMIs between 0 and LCM;
put first state in the priority queue pqueue;
R_old = ∅;              // R_old is the set of densities z
                        // of the back states after iteration k
(R_new, Missed) = construct_and_analyse();      // Missed is the set
                        // of expected deadline miss ratios
do
      R_old = R_new;
      (R_new, Missed) = construct_and_analyse();
while R_new ≠ R_old;


construct_and_analyse:
while ∃s ∈ pqueue such that s.pmi ≤ pmi_no do
      s_j = extract state from pqueue;
      τ_i = s_j.running;              // first field of the state
      ξ = convolute(ε_i, z_j);
      nextstatelist = next_states(s_j); // consider task dependencies!
      for each s_u ∈ nextstatelist do
            compute the probability of the transition
                                        from s_j to s_u using ξ;
            update deadline miss probabilities Missed;
            update z_u;
            if s_u ∉ pqueue then
                  put s_u in the pqueue;
            end if;
            if s_u is a back state and s_u ∉ R_new then
                  R_new = R_new ∪ {s_u};
            end if;
      end for;
      delete state s_j;
end while;
return (R_new, Missed);
```

Fig. 8. Construction and analysis algorithm

(1) Divide the interval $[0, LCM)$ in PMIs,

(2) Construct the stochastic process in topological order and analyse it.

The concept of PMI (called in their paper "state") was introduced by Zhou *et al.* [Zhou et al. 1999] in a different context, unrelated to the construction of a stochastic process. Let A denote the set of task arrivals in the interval $[0, LCM]$, i.e. $A = \{x | 0 \leq x \leq LCM, \exists 1 \leq i \leq N, \exists k \in \mathbb{N} : x = k\pi_i\}$. Let D denote the set of deadlines in the interval $[0, LCM]$, i.e. $D = \{x | 0 \leq x \leq LCM, \exists 1 \leq i \leq N, \exists k \in \mathbb{N} : x = k\pi_i + \delta_i\}$. The set of PMIs of $[0, LCM)$ is $\{[a, b) | a, b \in A \cup D \wedge \nexists x \in (A \cup D) \cap (a, b)\}$. If PMIs of a higher hyperperiod H_k, $k > 0$, are needed during the analysis, they are of the form $[a + k \cdot LCM, b + k \cdot LCM)$, where $[a, b)$ is a PMI of $[0, LCM)$.

```
next_states(s_j = (τ_i, W_i, t_i)):
nextstates = ∅;
max_exec_time = sup{t : ξ(t) > 0};        // the largest finishing time of τ_i
max_time = max{max_exec_time, discarding_time_i};        // the maximum between
                                // finishing time and discarding time of τ_i
PMI = {[lo_p, hi_p) ∈ PMIs : lo_p ≥ t_i ∧ hi_p ≤ max_time}        // the set of PMIs
                                // included in the interval [t_i, max_time]
for each [lo_p, hi_p) ∈ PMI do
        Arriv = {τ ∈ T : τ arrived in the interval [t_i, hi_p)};
        Discarded = {τ ∈ W_i : τ was discarded in the interval [t_i, hi_p)};
        Enabled = {τ ∈ T : τ becomes ready to execute as an effect of τ_i's
                completion}
        W = (W_i\Discarded) ∪ Enabled ∪ {τ ∈ Arriv : P_τ = ∅};        // add
the newly
                // arrived tasks with no predecessors, as they are
                // ready to execute, and the newly enabled ones
        select the new running task τ_u from W
                based on the scheduling policy
        W_u = W\{τ_u};
        add (τ_u, W_u, [lo_p, hi_p)) to nextstatelist;
done;
return nextstates;
```

Fig. 9. *next_states* procedure

The algorithm proceeds as discussed in Sections 4.1, 4.2 and 4.3. 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.

A global priority queue stores the states in the sliding window. The state priorities are assigned as shown in Section 4.2. The initial state of the stochastic process is put in the queue. The explanation of the algorithm is focused on the `construct_and_analyse` procedure. Each invocation of this procedure constructs and analyses the part of the underlying stochastic process which corresponds to one hyperperiod H_k. It starts with hyperperiod H_0 ($k = 0$). The procedure extracts one state at a time from the queue. Let $s_j = (\tau_i, W_i, pmi_i)$ be such a state. The probability density of the time when a transition occurs *to* s_j is given by the function z_j. The priority scheme of the priority queue ensures that s_j is extracted from the queue only after *all* the possible transitions to s_j have been considered, and thus z_j contains accurate information. In order to obtain the probability density of the time when task τ_i completes its execution, the probability density of its starting time (z_j) and the ETPDF of τ_i (ϵ_i) have to be convoluted. Let ξ be the probability density resulting from the convolution.

Figure 9 presents an algorithmic description of the *next_state* procedure. Based on ξ, the finishing time PDF of task τ_i if task τ_i is never discarded, we compute max_exec_time, the maximum execution time of task τ_i. max_time is the maximum between max_exec_time and the time at which task τ_i would be discarded. PMI will then denote the set of all PMIs included in the interval between the start of

the PMI in which task τ_i started to run and max_time. Task τ_i could, in principle, complete its execution during any of these PMIs. We consider each PMI as being the one in which task τ_i finishes its execution. A new underlying stochastic process state corresponds to each of these possible finishing PMIs. For each PMI, we determine the multiset $Arriv$ of newly arrived tasks while task τ_i was executing. Also, we determine the multiset $Discarded$ of those tasks which were ready to execute when task τ_i started, but were discarded in the mean time, as the execution of task τ_i spanned beyond their deadlines. Once task τ_i completes its execution, some of its successor tasks may become ready to execute. These successor tasks which become ready to execute as a result of task τ_i's completion form the set $Enabled$. The new multiset of ready tasks, W, is the union of the old multiset of ready tasks except the ones that were discarded during the execution of task τ_i, $W_i \backslash Discarded$, and the set $Enabled$ and those newly arrived tasks which have no predecessor and therefore are immediately ready to run. Once the new set of ready tasks is determined, the new running task τ_u is selected from multiset W based on the scheduling policy of the application. A new stochastic process state $(\tau_u, W \backslash \{\tau_u\}, [lo_p, hi_p))$ is constructed and added to the list of next states.

The probability densities z_u of the times a transition to $s_u \in nextstatelist$ is taken are updated based on ξ. The state s_u is then added to the priority queue and s_j removed from memory. This procedure is repeated until there is no task instantiation that started its execution in hyperperiod H_k (until no more states in the queue have their PMI field in the range $k \cdot pmi\_no, \ldots, (k+1) \cdot pmi\_no$, where pmi_no is the number of PMIs between 0 and LCM). Once such a situation is reached, partial results, corresponding to the hyperperiod H_k are available and the `construct_and_analyse` procedure returns. The `construct_and_analyse` procedure is repeated until the set of back states R does not change any more.

5. EXPERIMENTAL RESULTS

The most computation intensive part of the analysis is the computation of the convolutions $z_i * \epsilon_j$. In our implementation we used the FFTW library [Frigo and Johnson 1998] 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 size of the stochastic process depends are LCM (the least common multiple of the task periods), the number of PMIs, the number of tasks N, the task dependencies, and the maximum allowed number of concurrently active instantiations of the same task graph.

As the selection of the next running task is unique, given the pending tasks and the time moment, the particular scheduling policy has a reduced impact on the process size. Hence, we use the non-preemptive EDF scheduling policy in the experiments below. 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.

The ETPDFs were randomly generated. An interval $[Emin, Emax]$ has been divided into smaller intervals. For each of the smaller intervals, the ETPDF has a constant value, different from the value over other intervals. The curve shape

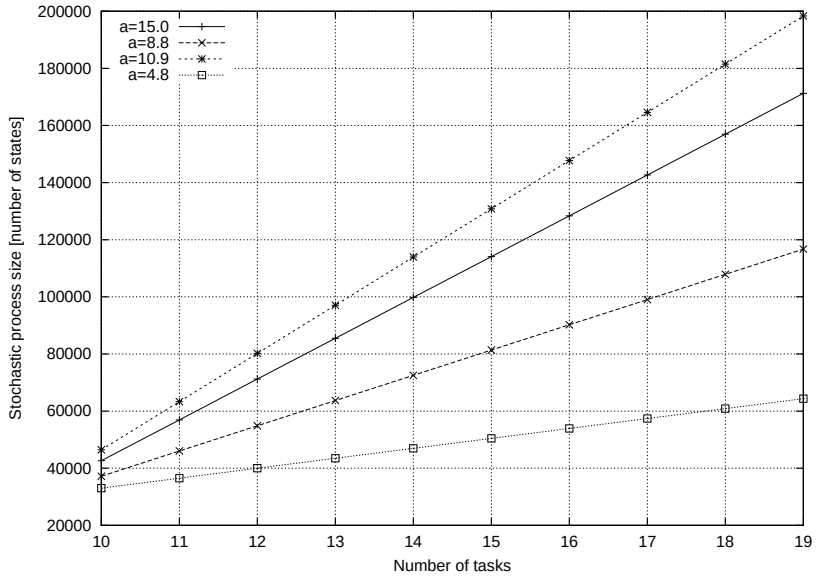


Fig. 10. Stochastic process size vs. number of tasks

has of course an influence on the final result of the analysis, but it has little or no influence on the analysis time and memory consumed by the analysis itself. The interval length $Emax - Emin$ influences the analysis time and memory, but only marginally.

The periods are randomly picked from a pool of periods with the restriction that the period of task τ has to be an integer multiple of the periods of the predecessors of task τ. The pool comprises periods in the range $2, 3, \ldots, 24$. Large prime numbers have a lower probability to be picked, but it occurs nevertheless.

In the following, we report on six sets of experiments. The first four investigate the impact of the enumerated factors (LCM, the number N of tasks, the task dependencies, the maximum allowed number of concurrently active instantiations of the same task graph) on the analysis complexity. The fifth set of experiments considers a different problem formulation. The sixth experiment is based on a real-life example in the area of telecommunication systems.

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. Both the stochastic process size and the maximum size of the sliding window are expressed in number of states. All experiments were performed on an UltraSPARC 10 at 450 MHz.

5.1 Stochastic process size vs. number of tasks

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. LCM, 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 $a = 15.0, 10.9,$

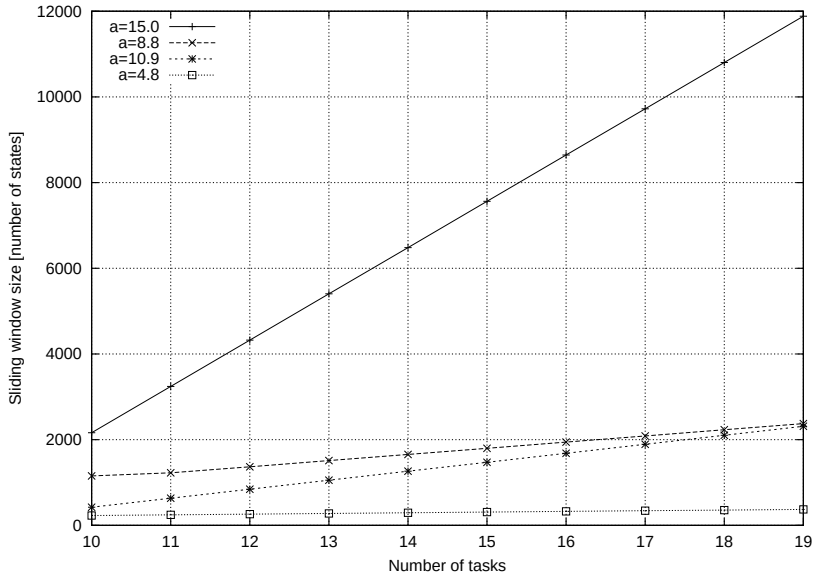


Fig. 11. Size of the sliding window of states vs. number of tasks

8.8, and 4.8 (keeping $LCM = 360$). 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 of 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.

5.2 Stochastic process size vs. application period LCM

In the second set of experiments we analysed the impact of the application period LCM (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 LCM takes values in the interval $[1, 5040]$. Figure 12 shows the variation of the average process size with the application period.

5.3 Stochastic process size vs. task dependency degree

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 LCM 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

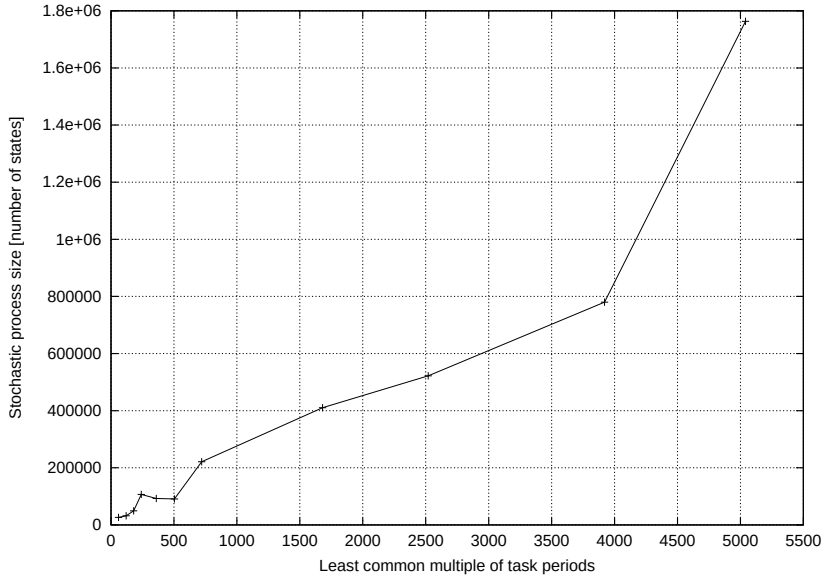


Fig. 12. Stochastic process size vs. application period LCM

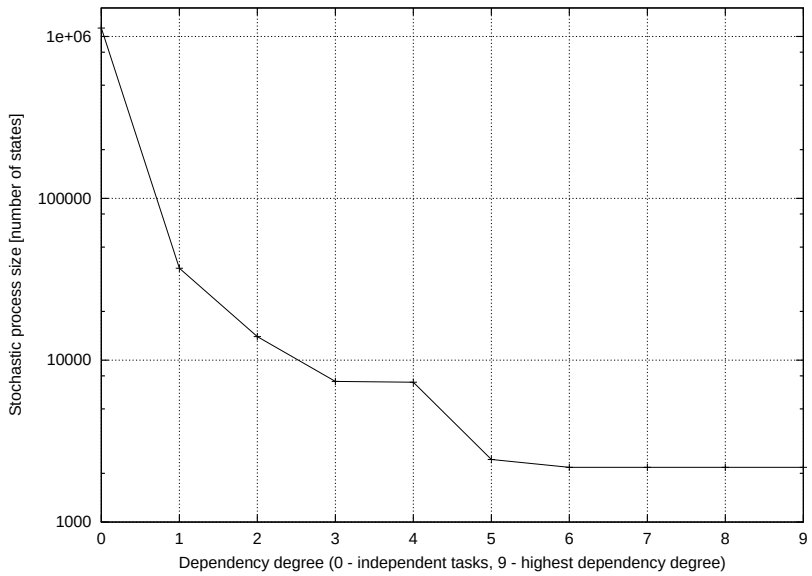


Fig. 13. Stochastic process size vs. task dependency degree

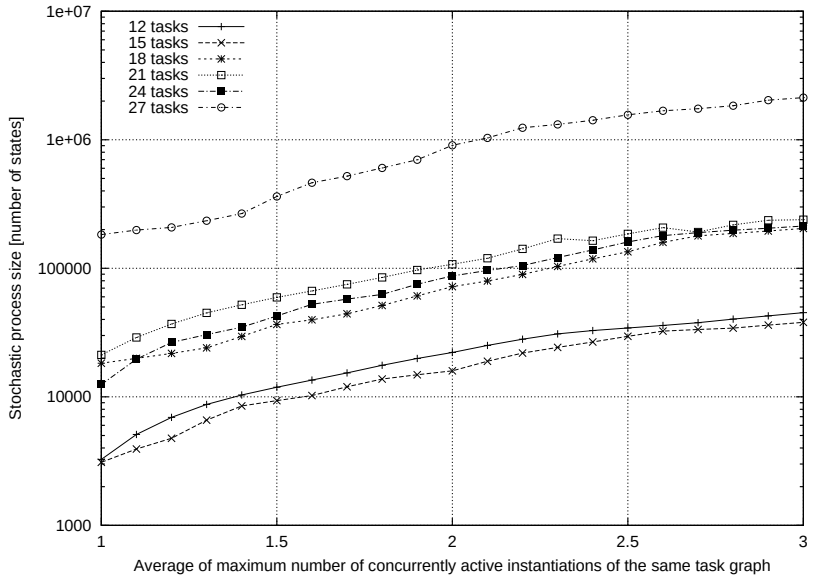


Fig. 14. Stochastic process size vs. average number of concurrently active instantiations of the same task graph

x axis represents the degree of dependencies among the tasks (0 for independent tasks, 9 for the initial task set with the highest amount of 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 this 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.

5.4 Stochastic process size vs. average number of concurrently active instantiations of the same task graph

In the fourth set of experiments, the impact of the average number of concurrently active instantiations of the same task graph on the stochastic process size was analysed. 18 sets of task graphs containing between 12 and 27 tasks grouped in 2 to 9 task graphs were randomly generated. Each task set was analysed between 9 and 16 times considering different upper bounds for the maximum allowed number of concurrently active task graph instantiations. These upper bounds ranged from 1 to 3. The results were averaged for the same number of tasks. The dependency of the underlying stochastic process size as a function of the average of the maximum allowed number of instantiations of the same task graph that are concurrently active is plotted in Figure 14. Note that the y-axis is logarithmic. Different curves correspond to different sizes of the considered task sets. It can be observed that the stochastic process size is approximately linear in the average of the maximum allowed number of concurrently active instantiations of the same task graph.

5.5 Rejection versus discarding

As formulated in Section 3.1, when there are b_i concurrently active instantiations of task graph G_i in the system, and a new instantiation of G_i demands service, the oldest instantiation of G_i is eliminated from the system. Sometimes, such a strategy is not desired, as the oldest instantiation might have been very close to finishing, and by discarding it, the invested resources (time, memory, bandwidth, etc.) are wasted.

Therefore, our problem formulation has been extended to support a late task policy in which, instead of discarding the oldest instantiation of G_i, the newly arrived instantiation is denied service (rejected) by the system.

In principle, the rejection policy is easily supported by only changing the `next_states` procedure in the algorithm presented in Section 4.4. However, this has a strong impact on the analysis complexity as shown in Table I. The significant increase in the stochastic process size (up to two orders of magnitude) can be explained considering the following example. Let s be the stochastic process state under analysis, let τ_j belonging to task graph G_i be the task running in s and let us consider that there are b_i concurrently active instantiations of G_i in the system. The execution time of τ_j may be very large, spanning over many PMIs. In the case of discarding, it was guaranteed that τ_j will stop running after at most $b_i \cdot \pi_{G_i}$ time units, because at that time moment it would be eliminated from the system. Therefore, when considering the discarding policy, the number of next states of a state s is upper bounded. When considering the rejection policy, this is not the case any more.

Moreover, let us assume that b_i instantiations of the task graph G_i are active in the system at a certain time. In the case of discarding, capturing this information in the system state is sufficient to unambiguously identify those b_i instantiations: they are the last b_i that arrived, because always the oldest one is discarded. For example, the two ready instantiations of τ_2 in the state $s_{13} = (\tau_1, \{\tau_2, \tau_2\}, [6, 8))$ in Figure 7 are the ones that arrived at the time moments 0 and 4. However, when the rejection policy is deployed, just specifying that b_i instantiations are in the system is not sufficient for identifying them. We will illustrate this by means of the following example. Let $b_i = 2$, and let the current time be $k\pi_{G_i}$. In a first scenario, the oldest instantiation of G_i, which is still active, arrived at time moment $(k-5)\pi_{G_i}$ and it still runs. Therefore, the second oldest instantiation of G_i is the one that arrived at time moment $(k-4)\pi_{G_i}$ and all the subsequent instantiations were rejected. In a second scenario, the instantiation that arrived at time moment $(k-5)\pi_{G_i}$ completes its execution shortly before time moment $(k-1)\pi_{G_i}$. In this case, the instantiations arriving at $(k-3)\pi_{G_i}$ and $(k-2)\pi_{G_i}$ were rejected but the one arriving at $(k-1)\pi_{G_i}$ was not. In both scenarios, the instantiation arriving at $k\pi_{G_i}$ is rejected, as there are two concurrently active instantiations of G_i in the system, but these two instantiations cannot be determined without extending the definition of the stochastic process state space. Extending this space with the task graph arrival times is partly responsible for the increase in number of states of the underlying stochastic process.

The fifth set of experiments reports on the analysis complexity when the rejection policy is deployed. 101 task sets of 12 to 27 tasks grouped in 2 to 9 task graphs were randomly generated. For each task set two analysis were performed, one considering

Tasks	Average stochastic process size [number of states]		Relative increase
	Discarding	Rejection	
12	2223.52	95780.23	42.07
15	7541.00	924548.19	121.60
18	4864.60	364146.60	73.85
21	18425.43	1855073.00	99.68
24	14876.16	1207253.83	80.15
27	55609.54	5340827.45	95.04

Table I. Discarding compared to rejection

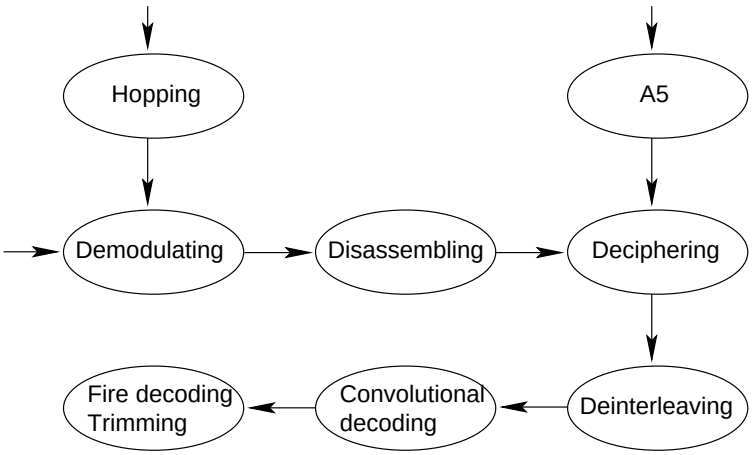


Fig. 15. Decoding of a GSM dedicated signalling channel

the discarding policy and the other considering the rejection policy. The results were averaged for task sets with the same cardinality and shown in Table I.

5.6 Decoding of a GSM dedicated signalling channel

Finally, we present an example from industry, in particular the mobile communication area. Figure 15 depicts a set of 8 tasks that co-operate in order to decode the digital bursts corresponding to a GSM 900 signalling channel [Mouly and Pautet 1992]. The incoming bursts are demodulated by the demodulation task, based on the frequency indexes generated by the frequency hopping task. The demodulated bursts are disassembled by the disassembling task. The resulting digital blocks are deciphered by the deciphering task based on a key generated by the A5 task. The deciphered block proceeds through bit deinterleaving, convolutional decoding (Viterbi decoding) and the so called fire decoding. The whole application runs on a single DSP processor and the tasks are scheduled according to fixed priority scheduling. All tasks have the same period, imposed by the TDMA scheme of the radio interface.

In this example, there are two sources of variation in execution times. The demodulating task has both data and control intensive behaviour, which can cause pipeline hazards on the deeply pipelined DSP it runs on. Its execution time prob-

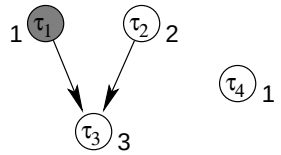
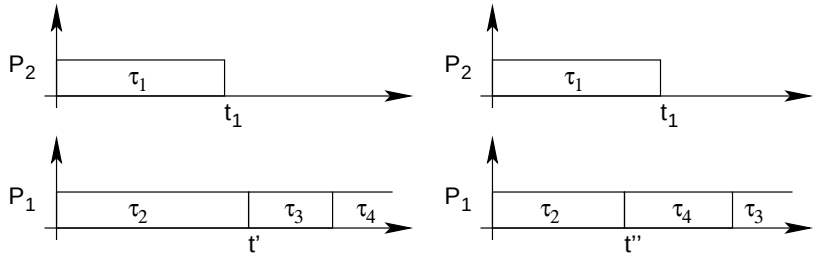


Fig. 16. Example of multiprocessor application



(a) Scenario 1 (b) Scenario 2

Fig. 17. Two execution scenarios

ability 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 A5 (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 DSP, the ratio of missed deadlines is 0 (all deadlines are met). Considering 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.

6. LIMITATIONS AND EXTENSIONS

Although our proposed method is, as shown, efficiently applicable to the analysis of applications implemented on monoprocessor systems, it can handle only small scale multiprocessor applications. This section identifies the causes of this limitation and sketches an alternative approach to handle multiprocessor applications.

When analysing multiprocessor applications, one approach could be to decompose the analysis problem into several subproblems, each of them analysing the tasks mapped on one of the processors. We could attempt to apply the presented approach in order to solve each of the subproblems. Unfortunately, in the case of multiprocessors and with the assumption of data dependencies among tasks, this approach cannot be applied. The reason is that the set of ready tasks cannot be

determined based solely on the information regarding the tasks mapped on the processor under consideration. To illustrate this, let us consider the example in Figure 16. Tasks τ_2, τ_3, and τ_4 are mapped on processor P_1 and task τ_1 is mapped on processor P_2. The numbers near the tasks indicate the task priorities. For simplicity, let us assume that all tasks have the same period π, and hence there is only one priority monotonicity interval $[0, \pi)$. Let us examine two possible scenarios. The corresponding Gantt diagrams are depicted in Figure 17. At time moment 0 task τ_1 starts running on processor P_2 and task τ_2 starts running on processor P_1. Task τ_1 completes its execution at time moment $t_1 \in [0, \pi)$. In the first scenario, task τ_2 completes its execution at time moment $t' > t_1$ and task τ_3 starts executing on the processor P_1 at time moment t' because it has the highest priority among the two ready tasks τ_3 and τ_4 at that time. In the second scenario, task τ_2 completes its execution at time moment $t'' < t_1$. Therefore, at time moment t'', only task τ_4 is ready to run and it will start its execution on the processor P_1 at that time. Thus, the choice of the next task to run is not independent of the time when the running task completes its execution inside a PMI. This makes the concept of PMIs unusable when looking at the processors in isolation.

An alternative approach would be to consider all the tasks and to construct the global state space of the underlying stochastic process accordingly. In principle, the approach presented in the previous sections could be applied in this case. However, the number of possible execution traces, and implicitly the stochastic process, explodes due to the parallelism provided by the application platform. As shown, the analysis has to store the probability distributions z_i for each process state in the sliding window of states, leading to large amounts of needed memory and limiting the appropriateness of this approach to very small multi-processor applications. Moreover, the number of convolutions $z_i * \epsilon_j$, being equal to the number of states, would also explode, leading to prohibitive analysis times.

We have addressed these problems [Manolache et al. 2002] by using an approximation approach for the task execution time probability distribution functions. Approximating the generalised ETPDFs with weighted sums of convoluted exponential functions leads to approximating the underlying generalised semi-Markov process with a continuous time Markov chain. By doing so, we avoid both the computation of convolutions and the storage of the z_i functions. However, as opposed to the method presented in this paper, which produces exact values for the expected deadline miss ratios, the alternative approach [Manolache et al. 2002] generates just approximations of the real ratios.

7. CONCLUSIONS

This work proposes a method for the schedulability analysis of task sets with probabilistically distributed task execution times. Our method improves the currently existing ones by providing *exact* solutions for *larger* and *less restricted* task sets. Specifically, we allow continuous task execution time probability distributions, and we do not restrict our approach to one particular scheduling policy. Additionally, task dependencies are supported, as well as arbitrary deadlines.

The analysis of task sets under such generous assumptions is made possible by three complexity management methods:

(1) the introduction and exploitation of the PMI concept,
(2) the concurrent construction and analysis of the stochastic process, and
(3) the usage of a sliding window of states made possible by the construction in topological order.

As the presented experiments demonstrate, the proposed method can efficiently be applied to applications implemented on monoprocessor systems.

REFERENCES

Abeni, L. and Butazzo, G. 1999. QoS guarantee using probabilistic deadlines. In *Proceedings of the* 11^{th} *Euromicro Conference on Real-Time Systems*. 242–249.

Abeni, L. and Buttazzo, G. C. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings of the* 19^{th} *Real Time Systems Symposium*. 4–13.

Atlas, A. and Bestavros, A. 1998. Statistical rate monotonic scheduling. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium*. 123–132.

Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W., and Wellings, A. J. 1995. Fixed priority pre-emptive scheduling: An historical perspective. *Journal of Real-Time Systems 8,* 2-3 (March-May), 173–198.

Balarin, F., Lavagno, L., Murthy, P., and Sangiovanni-Vincentelli, A. 1998. Scheduling for embedded real-time systems. *IEEE Design and Test of Computers*, 71–82.

Beck, J. E. and Siewiorek, D. P. 1998. Automatic configuration of embedded multicomputer systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17,* 2, 84–95.

Buttazzo, G. C. 1997. *Hard Real-Time Computing Systems*. Kluwer Academic.

Buttazzo, G. C., Lipari, G., and Abeni, L. 1998. Elastic task model for adaptive rate control. In *Proceedings of the* 19^{th} *Real Time Systems Symposium*. 286–295.

Dai, J. G. and Wang, Y. 1993. Nonexistence of Brownian models for certain multiclass queueing networks. *Queueing Systems 13*, 41–46.

De Micheli, G. and Gupta, R. K. 1997. Hardware/software co-design. *Proceedings of the IEEE 85,* 3 (March), 349–365.

de Veciana, G., Jacome, M., and Guo, J.-H. 2000. Assessing probabilistic timing constraints on system performance. *Design Automation for Embedded Systems 5,* 1 (February), 61–81.

Díaz, J. L., García, D. F., Kim, K., Lee, C.-G., Lo Bello, L., López, J. M., Min, S. L., and Mirabella, O. 2002. Stochastic analysis of periodic real-time systems. In *Proceedings of the* 23^{rd} *Real-Time Systems Symposium*.

Ernst, R. 1998. Codesign of embedded systems: Status and trends. *IEEE Design and Test of Computers*, 45–54.

Fidge, C. J. 1998. Real-time schedulability tests for preemptive multitasking. *Journal of Real-Time Systems 14,* 1, 61–93.

Frigo, M. and Johnson, S. G. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. Vol. 3. 1381–1384.

Gautama, H. 1998. A probabilistic approach to the analysis of program execution time. Tech. Rep. 1-68340-44(1998)06, Faculty of Information Technology and Systems, Delft University of Technology.

Gautama, H. and van Gemund, A. J. C. 2000. Static performance prediction of data-dependent programs. In *Proceedings of the* 2^{nd} *International Workshop on Software and Performance*. 216–226.

Goel, A. and Indyk, P. 1999. Stochastic load balancing and related problems. In *IEEE Symposium on Foundations of Computer Science*. 579–586.

Hu, X. S., Zhou, T., and Sha, E. H.-M. 2001. Estimating probabilistic timing performance for real-time embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9,* 6 (December), 833–844.

Kalavade, A. and Moghé, P. 1998. A tool for performance estimation of networked embedded end-systems. In *Proceedings of the* 35^{th} *Design Automation Conference*. 257–262.

Kim, J. and Shin, K. G. 1996. Execution time analysis of communicating tasks in distributed systems. *IEEE Transactions on Computers 45,* 5 (May), 572–579.

Kleinberg, J., Rabani, Y., and Tardos, E. 2000. Allocating bandwidth for bursty connections. *SIAM Journal on Computing 30,* 1, 191–217.

Kopetz, H. 1997. *Real-Time Systems*. Kluwer Academic.

Lee, C., Potkonjak, M., and Wolf, W. 1999. Synthesis of hard real-time application specific systems. *Design Automation of Embedded Systems 4*, 215–242.

Lehoczky, J. P. 1996. Real-time queueing theory. In *Proceedings of the* 18^{th} *Real-Time Systems Symposium*. 186–195.

Liu, C. L. and Layland, J. W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM 20,* 1 (January), 47–61.

Manolache, S., Eles, P., and Peng, Z. 2001. Memory and time-efficient schedulability analysis of task sets with stochastic execution time. In *Proceedings of the* 13^{th} *Euromicro Conference on Real Time Systems*. 19–26.

Manolache, S., Eles, P., and Peng, Z. 2002. Schedulability analysis of multiprocessor real-time applications with stochastic task execution times. In *Proceedings of the* 20^{th} *International Conference on Computer Aided Design*. 699–706.

Mouly, M. and Pautet, M.-B. 1992. *The GSM System for Mobile Communication*. Palaiseau.

Potkonjak, M. and Rabaey, J. M. 1999. Algorithm selection: A quantitative optimization-intensive approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18,* 5 (May), 524–532.

Powell Douglass, B., Harel, D., and Trakhtenbrot, M. 1996. *Statecharts in Use: Structured Analysis and Object-Orientation*. Springer, 368–394.

Sarkar, A., Waxman, R., and Cohoon, J. P. 1995. *Specification-Modeling Methodologies for Reactive-System Design*. Kluwer Academic Publishers, 1–34.

Stankovic, J. and Ramamritham, K., Eds. 1993. *Advances in Real-Time Systems*. IEEE Computer Society Press.

Tia, T.-S., Deng, Z., Shankar, M., Storch, M., Sun, J., Wu, L.-C., and Liu, J. W. S. 1995. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. 164–173.

van Gemund, A. J. 1996. Performance modelling of parallel systems. Ph.D. thesis, Delft University of Technology.

van Gemund, A. J. C. 2003. Symbolic performance modeling of parallel systems. *IEEE Transactions on Parallel and Distributed Systems*. to be published.

Wolf, W. 1994. Hardware-software co-design of embedded systems. *Proceedings of the IEEE 82,* 7, 967–989.

Zhou, T., Hu, X. S., and Sha, E. H.-M. 1999. A probabilistic performace metric for real-time system design. In *Proceedings of the* 7^{th} *International Workshop on Hardware-Software Co-Design*. 90–94.