

An Exact Stochastic Analysis of Priority-Driven Periodic Real-Time Systems *

Kanghee Kim
Chang-Gun Lee

José Luis Díaz
Daniel F. García

Lucia Lo Bello
Sang Lyul Min

José María López
Orazio Mirabella

*The contents of this technical report were edited in 2005 in order to fix some minor errors.

Abstract

This paper describes a stochastic analysis framework for general priority-driven periodic real-time systems. The proposed framework accurately computes the response time distribution of each task in the system, thus making it possible to determine the deadline miss probability of individual tasks, even for systems with a maximum utilization factor greater than 1. The framework is uniformly applied to general priority-driven systems, including fixed-priority systems (such as Rate Monotonic) and dynamic-priority systems (such as Earliest Deadline First), and can handle tasks with arbitrary relative deadlines and execution time distributions. In the framework, both an exact method and approximation methods to compute the response time distributions are presented and compared in terms of analysis accuracy and complexity. We prove that the complexity of the exact method is polynomial in terms of the number of jobs in a hyperperiod of the task set and the maximum length of the execution time distributions, and show that the approximation methods can significantly reduce the complexity without loss of accuracy.

An Exact Stochastic Analysis of Priority-Driven Periodic Real-Time Systems

Kanghee Kim[†] José Luis Díaz[‡] Lucia Lo Bello[§] José María López[‡]
Chang-Gun Lee[¶] Daniel F. García[‡] Sang Lyul Min[†] Orazio Mirabella[§]

I. INTRODUCTION

Most recent research on hard real-time systems has used the periodic task model [1] in analyzing the schedulability of a given task set where tasks are released periodically. Based on this periodic task model, various schedulability analysis methods for priority-driven systems have been developed to provide a deterministic guarantee that all the instances, called *jobs*, of every task in the system meet their deadlines, assuming that every job in a task requires its worst case execution time [1], [2], [3].

Although this deterministic timing guarantee is needed in hard real-time systems, it is too stringent for soft real-time applications that only require a probabilistic guarantee that the deadline miss ratio of a task is below a given threshold. For soft real-time applications, we need to relax the assumption that every instance of a task requires the worst case execution time in order to improve the system utilization. This is also needed for probabilistic hard real-time systems [4] where a probabilistic guarantee close to 0% suffices, i.e. the overall deadline miss ratio of the system should be below a hardware failure ratio.

Progress has recently been made in the analysis of real-time systems under the stochastic assumption that jobs from a task require variable execution times. Research in this area can be categorized into two groups depending on the approach used to facilitate the analysis. The methods in the first group introduce a worst-case assumption to simplify the analysis (e.g., the critical instant assumption in Probabilistic Time Demand Analysis [5] and Stochastic Time Demand Analysis [6], [7]) or a restrictive assumption (e.g., the heavy traffic condition in the Real-Time Queueing Theory [8], [9]). Those in the second group, on the other hand, assume a special scheduling model that provides isolation between tasks so that each task can be analyzed independently of the other tasks in the system (e.g., the reservation-based system addressed in [10] and Statistical Rate Monotonic Scheduling [11]).

^{*†} Kanghee Kim (khkim@archi.snu.ac.kr) and Sang Lyul Min (symin@dandelion.snu.ac.kr), *School of Computer Science and Engineering, Seoul National University* (Seoul, 151-742 Korea). This work was supported in part by the Ministry of Science and Technology under the National Research Laboratory program and by the Ministry of Education under the BK21 program. Also, for this work, the ICT at Seoul National University provided research facilities.

[‡] José Luis Díaz, José María López and Daniel F. García ([jdiaz, chechu, daniel}@atc.uniovi.es](mailto:{jdiaz, chechu, daniel}@atc.uniovi.es)), *Departamento de Informática, Universidad de Oviedo* (33204, Gijón, Spain)

[§] Lucia Lo Bello and Orazio Mirabella ([llobello, omirabel}@diit.unict.it](mailto:{llobello, omirabel}@diit.unict.it)), *Dipartimento di Ingegneria Informatica e delle Telecomunicazioni, Facoltà di Ingegneria, Università di Catania* (Viale A. Doria 6, 95125 Catania, Italy)

[¶] Chang-Gun Lee (cglee@ee.eng.ohio-state.edu), *Department of Electrical Engineering, Ohio State University* (2015 Neil Avenue, Columbus, OH 43210, U.S.A.)

In this paper, we describe a stochastic analysis framework that does not introduce any worst-case or restrictive assumptions into the analysis, and is applicable to general priority-driven real-time systems. The proposed framework builds upon Stochastic Time Demand Analysis (STDA) in that the techniques used in the framework to compute the response time distributions of tasks are largely borrowed from the STDA. However, unlike the STDA, which focuses on particular execution scenarios starting at a critical instant, the proposed framework considers all possible execution scenarios in order to obtain the exact response time distributions of the tasks. Moreover, while the STDA addresses only fixed-priority systems such as Rate Monotonic [1] and Deadline Monotonic [12], our framework extends to dynamic-priority systems such as Earliest Deadline First [1]. The contributions of the paper can be summarized as follows:

- The framework gives the *exact* response time distributions of the tasks. It assumes neither a particular execution scenario of the tasks such as critical instants, nor a particular system condition such as heavy traffic, in order to obtain accurate analysis results considering all possible execution scenarios for a wide range of system conditions.
- The framework provides a *unified* approach to addressing general priority-driven systems, including both fixed-priority systems such as Rate Monotonic and Deadline Monotonic, and dynamic-priority systems such as Earliest Deadline First. We neither modify the conventional rules of priority-driven scheduling, nor introduce other additional scheduling rules such as reservation scheduling, in order to analyze the priority-driven system as it is.

In our framework, in order to consider all possible execution scenarios in the system, we analyze a whole hyperperiod of the given task set (which is defined as a period whose length is equal to the least common multiple of the periods of all the tasks). In particular, to handle even cases where one hyperperiod affects the next hyperperiod, which occurs when the maximum utilization of the system is greater than 1, we take the approach of modelling the system as a Markov process over an infinite sequence of hyperperiods. This modelling leads us to solve an infinite number of linear equations, so we present three different methods to solve it: one method gives the exact solution, and the others give approximated solutions. We compare all these methods in terms of analysis complexity and accuracy through experiments. It should be noted that our framework subsumes the conventional deterministic analysis in the sense that, by modelling the worst case execution times as single-valued distributions, it always produces the same result as the deterministic analysis on whether a task set is schedulable or not.

The rest of the paper is organized as follows. In Section II, the related work is described in detail. In Section III, the system model is explained. Sections IV and V describe the stochastic analysis framework including the exact and the approximation methods. In Section VI, the complexity of the methods is analyzed, and in Section VII, a comparison between the solutions obtained by the methods is given, together with other analysis methods proposed in literature. Finally, in Section VIII, we conclude the paper with directions for future research.

II. RELATED WORK

Several studies have addressed the variability of task execution times in analyzing the schedulability of a given task set. Research in this area can be categorized into two groups depending on the approach taken to make the analysis possible. The methods in the first group [5], [6], [7], [8], [9], [13], [14] introduce a worst-case or restrictive assumption to simplify the analysis. Those in the second group [10], [11] assume a special scheduling model that provides isolation between tasks so that each task can be analyzed independently of other tasks in the system.

Examples of analysis methods in the first group include Probabilistic Time Demand Analysis (PTDA) [5] and Stochastic Time Demand Analysis (STDA) [6], [7], both of which target fixed-priority systems with tasks having arbitrary execution time distributions. PTDA is a stochastic extension of the Time Demand Analysis [2] and can only deal with tasks with relative deadlines smaller than or equal to the periods. STDA, on the other hand, which is a stochastic extension of General Time Demand Analysis [3], can handle tasks with relative deadlines greater than the periods. Like the original time demand analysis, both methods assume the critical instant where the task being analyzed and all the higher priority tasks are released or arrive at the same time. Although this worst-case assumption simplifies the analysis, it only results in an upper bound on the deadline miss probability, the conservativeness of which depends on the number of tasks and the average utilization of the system. Moreover, both analyses are valid only when the maximum utilization of the system does not exceed 1.

Other examples of analysis methods in the first group are the method proposed by Manolache et al. [13], which addresses only uniprocessor systems, and the one proposed by Leulseged and Nissanke [14], which extends to multiprocessor systems. These methods, like the one presented in this paper, cover general priority-driven systems including both fixed-priority and dynamic-priority systems. However, to limit the scope of the analysis to a single hyperperiod, both methods assume that the relative deadlines of tasks are shorter than or equal to their periods and that all the jobs that miss the deadlines are dropped. Moreover, in [13], all the tasks are assumed to be non-preemptable to simplify the analysis.

The first group also includes the Real-Time Queueing Theory [8], [9], which extends the classical queueing theory to real-time systems. This analysis method is flexible, in that it is not limited to a particular scheduling algorithm and can be extended to real-time queueing networks. However, it is only applicable to systems where the heavy traffic assumption (i.e., the average system utilization is close to 1) holds. Moreover, it only considers one class of tasks such that the interarrival times and execution times are identically distributed.

Stochastic analysis methods in the second group include the one proposed by Abeni and Buttazzo [10], and the method with Statistical Rate Monotonic Scheduling (SRMS) [11]. Both assume reservation-based scheduling algorithms so that the analysis can be performed as if each task had a dedicated (virtual) processor. That is, each task is provided with a guaranteed budget of processor time in every period [10] or super-period (the period of the next low priority task, which is assumed to be an integer multiple of the period of the task in SRMS) [11]. So, the deadline miss probability of a task can be analyzed independently of the other tasks, assuming the

guaranteed budget. However, these stochastic analysis methods are not applicable to general priority-driven systems due to the modification of the original priority-driven scheduling rules or the use of reservation-based scheduling algorithms.

III. SYSTEM MODEL

We assume a uniprocessor system that consists of n independent periodic tasks $S = \{\tau_1, \dots, \tau_n\}$, each task τ_i ($1 \leq i \leq n$) being modeled by the tuple (T_i, Φ_i, C_i, D_i) , where T_i is the period of the task, Φ_i its initial phase, C_i its execution time, and D_i its relative deadline. The execution time is a discrete random variable* with a given probability mass function (PMF), denoted by $f_{C_i}(\cdot)$, where $f_{C_i}(c) = \mathbb{P}\{C_i=c\}$. The execution time PMF can be given by a measurement-based analysis such as automatic tracing analysis [15], and stored as a finite vector, whose indices are possible values of the execution time and the stored values are their probabilities. The indices range from a minimum execution time C_i^{\min} to a maximum execution time C_i^{\max} . Without loss of generality, the phase Φ_i of each task τ_i is assumed to be smaller than T_i . The relative deadline D_i can be smaller than, equal to, or greater than T_i .

Associated with the task set, the system utilization is defined as the sum of the utilizations of all the tasks. Due to the variability of task execution times, the minimum U^{\min} , maximum U^{\max} , and the average system utilization \bar{U} are defined as $\sum_{i=1}^n C_i^{\min}/T_i$, $\sum_{i=1}^n C_i^{\max}/T_i$, and $\sum_{i=1}^n \bar{C}_i/T_i$, respectively. In addition, a hyperperiod of the task set is defined as a period of length T_H , which is equal to the least common multiple of the task periods, i.e., $T_H = \text{lcm}_{1 \leq i \leq n}\{T_i\}$.

Each task gives rise to an infinite sequence of jobs, whose release times are deterministic. If we denote the j -th job of task τ_i by $J_{i,j}$, its release time $\lambda_{i,j}$ is equal to $\Phi_i + (j-1)T_i$. Each job $J_{i,j}$ requires an execution time, which is described by a random variable following the given PMF $f_{C_i}(\cdot)$ of the task τ_i , and is assumed to be independent of other jobs of the same task and those of other tasks. However, throughout the paper we use a single index j for the job subscript, since the task that the job belongs to is not important in describing our analysis framework. On the other hand, we sometimes additionally use a superscript for the job notation, to express the hyperperiod that the job belongs to. That is, we use $J_j^{(k)}$ to refer to the j -th job in the k -th hyperperiod.

The scheduling model we assume is a general priority-driven preemptive one that covers both fixed-priority systems such as Rate Monotonic (RM) and Deadline Monotonic (DM), and dynamic-priority systems such as Earliest Deadline First (EDF). The only limitation is that once a priority is assigned to a job, it never changes, which is called a job-level fixed-priority model [16]. According to the priority, all the jobs are scheduled in such a way that, at any time, the job with the highest priority is always served first. If two or more jobs with the same priority are ready at the same time, the one that arrived first is scheduled first. We denote the priority of job J_j by a priority value p_j . Note that a higher priority value means a lower priority.

*Throughout this paper, we use a calligraphic typeface to denote random variables, e.g., \mathcal{C} , \mathcal{W} , and \mathcal{R} , and a non-calligraphic typeface to denote deterministic variables, e.g., C , W , and R .

The response time for each job J_j is represented by \mathcal{R}_j and its PMF by $f_{\mathcal{R}_j}(r) = \mathbb{P}\{\mathcal{R}_j=r\}$. From the job response time PMFs, we can obtain the response time PMF for any task by averaging those of all the jobs belonging to the task. The task response time PMFs provide the analyst with significant information about the stochastic behavior of the system. In particular, the PMFs can be used to compute the probability of deadline misses for the tasks. The deadline miss probability DMP_i of task τ_i can be computed as follows:

$$DMP_i = \mathbb{P}\{\mathcal{R}_i > D_i\} = 1 - \mathbb{P}\{\mathcal{R}_i \leq D_i\} \quad (1)$$

IV. STOCHASTIC ANALYSIS FRAMEWORK

A. Overview

The goal of the proposed analysis framework is to accurately compute the *stationary* response time distributions of all the jobs, when the system is in the steady state. The stationary response time distribution of job J_j can be defined as follows:

$$\lim_{k \rightarrow \infty} f_{\mathcal{R}_j}^{(k)} = f_{\mathcal{R}_j}^{(\infty)}$$

where $f_{\mathcal{R}_j}^{(k)}$ is the response time PMF of $J_j^{(k)}$, the j -th job in the k -th hyperperiod. In this section, we will describe how to compute the response time distributions of all the jobs in an arbitrary hyperperiod k , and then, in the following section, explain how to compute the stationary distributions, which are obtained when $k \rightarrow \infty$. We start our discussion by explaining how the response time \mathcal{R}_j of a job J_j is determined.

The response time of a job J_j is determined by two factors. One is the pending workload that delays the execution of J_j , which is observed immediately prior to its release time λ_j . We call this pending workload *backlog*. The other is the workload of jobs that may preempt J_j , which are released after J_j . We call this workload *interference*. Since both the backlog and the interference for J_j consist of jobs with a priority higher than that of J_j (i.e., with a priority value smaller than the priority value p_j of J_j), we can elaborate the two terms to p_j -backlog and p_j -interference, respectively. Thus, the response time of J_j can be expressed by the following equation

$$\mathcal{R}_j = \mathcal{W}_{p_j}(\lambda_j) + \mathcal{C}_j + \mathcal{J}_{p_j} \quad (2)$$

where $\mathcal{W}_{p_j}(\lambda_j)$ is the p_j -backlog observed at time λ_j , \mathcal{C}_j is the execution time of J_j , and \mathcal{J}_{p_j} is the p_j -interference occurring after time λ_j .

In our framework, we compute the distribution of the response time \mathcal{R}_j in two steps: *backlog analysis* and *interference analysis*. In the backlog analysis, the stationary p_j -backlog distributions $f_{\mathcal{W}_{p_j}(\lambda_j)}(\cdot)$ of all the jobs in a hyperperiod are computed. Then, in the interference analysis, the stationary response time distributions $f_{\mathcal{R}_j}(\cdot)$ of the jobs are determined by introducing the associated execution time distribution $f_{\mathcal{C}_j}(\cdot)$ and the p_j -interference effect \mathcal{J}_{p_j} into each stationary p_j -backlog distribution $f_{\mathcal{W}_{p_j}(\lambda_j)}(\cdot)$.

B. Backlog analysis algorithm

For the backlog analysis, we assume a job sequence $\{J_1, \dots, J_j\}$ in which all the jobs have a priority value smaller than or equal to p_j . It is also assumed that the

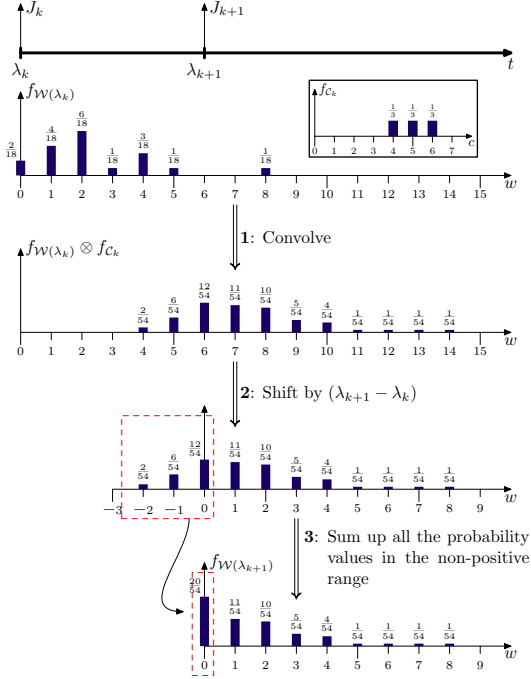


Fig. 1. An example of backlog analysis using the convolve-shrink procedure

stationary p_j -backlog distribution observed immediate prior to the release time of the first job J_1 , i.e., $f_{\mathcal{W}_{p_j}(\lambda_1)}(\cdot)$, is given. In Section V, it will be explained how the assumed stationary backlog distribution can be computed. Then the p_j -backlog distribution $f_{\mathcal{W}_{p_j}(\lambda_j)}(\cdot)$ immediate prior to the release time of J_j can be computed from $f_{\mathcal{W}_{p_j}(\lambda_1)}(\cdot)$ by the algorithm described in this subsection. For the sake of brevity, we will simplify the notation $\mathcal{W}_{p_j}(\lambda_j)$ to $\mathcal{W}(\lambda_j)$, i.e., without the subscript denoting the priority level p_j .

Let us first consider how to compute the backlog when the execution times of all the jobs are given as deterministic values. In this deterministic scenario, the backlog $W(\lambda_k)$ immediate prior to the release time of each job J_k ($1 \leq k < j$) can be expressed as follows:

$$W(\lambda_{k+1}) = \max\{W(\lambda_k) + C_k - (\lambda_{k+1} - \lambda_k), 0\} \quad (3)$$

So, once the backlog $W(\lambda_1)$ for the first job J_1 is given, the series of the backlog $\{W(\lambda_2), W(\lambda_3), \dots, W(\lambda_j)\}$ can be calculated by repeatedly applying Equation (3) along the job sequence.

Then we can explain our backlog analysis algorithm as a stochastic extension of Equation (3). Deterministic variables $W(\lambda_k)$ and C_k are translated into random variables $\mathcal{W}(\lambda_k)$ and \mathcal{C}_k , and Equation (3) is translated into a numerical procedure on the associated PMFs. This procedure can be summarized in the following three steps:

- 1) The expression “ $\mathcal{W}(\lambda_k) + \mathcal{C}_k$ ” is translated into convolution between the two PMFs of the random variables $\mathcal{W}(\lambda_k)$ and \mathcal{C}_k , respectively.

$$f_{\mathcal{W}(\lambda_k) + \mathcal{C}_k}(\cdot) = (f_{\mathcal{W}(\lambda_k)} \otimes f_{\mathcal{C}_k})(\cdot)$$

In Figure 1, for example, the arrow annotated with “Convolve” shows such a convolution operation.

- 2) The expression “ $\mathcal{W}(\lambda_k) + \mathcal{C}_k - (\lambda_{k+1} - \lambda_k)$ ” is translated into shifting the PMF $f_{\mathcal{W}(\lambda_k) + \mathcal{C}_k}(\cdot)$ obtained above by $(\lambda_{k+1} - \lambda_k)$ units to the left. In the example shown in Figure 1, the amount of the shift is 6 time units.
- 3) The expression “ $\max\{\mathcal{W}(\lambda_k) + \mathcal{C}_k - (\lambda_{k+1} - \lambda_k), 0\}$ ” is translated into summing up all the probability values in the negative range of the PMF obtained above and adding the sum to the probability of the backlog equal to zero. In the above example, the probability sum is $20/54$.

These three steps exactly describe how to obtain the backlog PMF $f_{\mathcal{W}(\lambda_{k+1})}(\cdot)$ from the preceding backlog PMF $f_{\mathcal{W}(\lambda_k)}(\cdot)$. So, starting from the first job in the given sequence, for which the stationary backlog PMF $f_{\mathcal{W}(\lambda_1)}(\cdot)$ is assumed to be known, we can compute the stationary backlog PMF of the last job J_j by repeatedly applying the above procedure along the sequence. We refer to this procedure as “convolve-shrink”.

C. Interference analysis algorithm

Once the p_j -backlog PMF is computed for each job J_j at its release time by the backlog analysis algorithm described above, we can easily obtain the response time PMF of the job J_j by convolving the p_j -backlog PMF $f_{\mathcal{W}_{p_j}(\lambda_j)}(\cdot)$ and the execution time PMF $f_{\mathcal{C}_j}(\cdot)$. This response time PMF is correct if the job J_j is non-preemptable. However, if J_j is preemptable, and there exist higher priority jobs following J_j , we have to further analyze the p_j -interference for J_j , caused by all the higher priority jobs, to obtain the complete response time PMF.

For the interference analysis, we have to identify all the higher priority jobs following J_j . These higher priority jobs can easily be found by searching for all the jobs released later than J_j and comparing their priorities with that of J_j . For the sake of brevity, we represent these jobs with $\{J_{j+1}, J_{j+2}, \dots, J_{j+k}, \dots\}$, while slightly changing the meaning of the notation λ_{j+k} from the absolute release time to the release time relative to λ_j , i.e., $\lambda_{j+k} \leftarrow (\lambda_{j+k} - \lambda_j)$.

As in the case of the backlog analysis algorithm, let us first consider how to compute the response time R_j of J_j when the execution times of all the jobs are given as deterministic values. In this deterministic scenario, the response time R_j of J_j can be computed by the following algorithm:

$$\begin{aligned}
 R_j &= W_{p_j}(\lambda_j) + C_j ; k = 1 \\
 &\text{while } R_j > \lambda_{j+k} \\
 R_j &= R_j + C_{j+k} ; k = k + 1
 \end{aligned} \tag{4}$$

The total number k of iterations of the “while” loop is determined by the final response time that does not reach the release time of the next higher priority job J_{j+k+1} . For an arbitrary value k , the final response time R_j is given as $W_{p_j}(\lambda_j) + C_j + \sum_{l=1}^k C_{j+l}$.

We can explain our interference analysis algorithm as a stochastic extension of Algorithm (4). We treat deterministic variables R_j and C_j as random variables \mathcal{R}_j

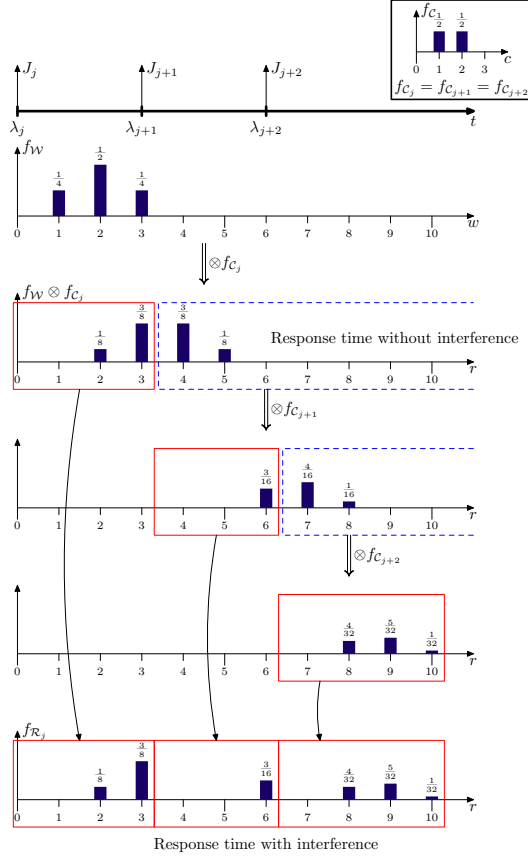


Fig. 2. An example of interference analysis using the split-convolve-merge procedure

and \mathcal{C}_j , and translate Algorithm (4) into a numerical procedure on the associated PMFs as follows:

- 1) The expression “ $\mathcal{R}_j = \mathcal{W}_{p_j}(\lambda_j) + \mathcal{C}_j$ ” is translated into $f_{\mathcal{R}_j}(\cdot) = (f_{\mathcal{W}_{p_j}(\lambda_j)} \otimes f_{\mathcal{C}_j})(\cdot)$. This response time PMF is valid in the interval $(0, \lambda_{j+1}]$. For example, in Figure 2, the first convolution \otimes shows the corresponding operation.
- 2) While $\mathbb{P}\{\mathcal{R}_j > \lambda_{j+k}\} > 0$, the expression “ $\mathcal{R}_j = \mathcal{R}_j + \mathcal{C}_{j+k}$ ” is translated into convolution between the partial PMF defined in the range (λ_{j+k}, ∞) of the response time PMF $f_{\mathcal{R}_j}(\cdot)$ calculated in the previous iteration and the execution time PMF $f_{\mathcal{C}_{j+k}}(\cdot)$. The resulting PMF is valid in the range $(\lambda_{j+k}, \lambda_{j+k+1}]$. When $\mathbb{P}\{\mathcal{R}_j > \lambda_{j+k}\} = 0$, the loop is terminated. In the example shown in Figure 2, this procedure is described by the two successive convolutions, where only two higher priority jobs J_{j+1} and J_{j+2} are assumed (In this case, all three jobs are assumed to have the same execution time distribution).

Note that in the above procedure the number of higher priority jobs we have to consider in a real system can be arbitrarily large. However, in practice, since we are often interested only in the probability of job J_j missing the deadline D_j , the set of interfering jobs we have to consider can be limited to the jobs released in the time interval $(\lambda_j, \lambda_j + D_j)$. This is because we can compute the deadline miss probability, i.e., $\mathbb{P}\{\mathcal{R}_j > D_j\}$, from the partial response time distribution defined in

the range $[0, D_j]$, i.e., $\mathbb{P}\{\mathcal{R}_j > D_j\} = 1 - \mathbb{P}\{\mathcal{R}_j \leq D_j\}$. Thus, we can terminate the “while” loop of Algorithm (4) when λ_{j+k} is greater than D_j . For the example in Figure 2, if the relative deadline D_j of J_j is 7, the deadline miss probability will be $\mathbb{P}\{\mathcal{R}_j > D_j\} = 1 - 11/16 = 5/16$.

We will refer to the above procedure as “split-convolve-merge”, since at each step the response time PMF being computed is split, the resulting tail is convolved with the associated execution time distribution, and this newly made tail and the original head are merged.

D. Backlog dependency tree

In the backlog analysis algorithm, for a given job J_j , we assumed that a sequence of preceding jobs with a priority higher than or equal to that of J_j and the stationary backlog distribution of the first job in the sequence were given. In this subsection, we will explain how to derive such a job sequence for each job in a hyperperiod. As a result, we give a *backlog dependency tree* where the p_j -backlog distributions of all the jobs in the hyperperiod can be computed by traversing the tree while applying the convolve-shrink procedure. This backlog dependency tree greatly simplifies the steady-state backlog analysis for the jobs, since it reduces the problem to computing only the stationary backlog distribution of the root job of the tree. In Section V, we will address how to compute the stationary backlog distribution for the root job.

To show that there exist dependencies between the p_j -backlog’s, we first classify all the jobs in a hyperperiod into *ground jobs* and *non-ground jobs*. A ground job is defined as a job that has a lower priority than those of all the jobs previously released. That is, J_j is a ground job if and only if $p_k \leq p_j$ for all jobs J_k such that $\lambda_k < \lambda_j$. A non-ground job is a job that is not a ground job. One important implication from the ground job definition is that the p_j -backlog of a ground job is always equal to the total backlog in the system observed at its release time. We call the total backlog *system backlog* and denote it by $\mathcal{W}(t)$, i.e., without the subscript p_j denoting the priority level. So, for a ground job J_j , $\mathcal{W}_{p_j}(\lambda_j) = \mathcal{W}(\lambda_j)$.

Let us consider the task set example shown in Figure 3(a). This task set consists of two tasks τ_1 and τ_2 with the relative deadlines equal to the periods 20 and 70, respectively. The phases Φ_i of both tasks are zero. We assume that these tasks are scheduled by EDF.

In this example, there are five ground jobs J_1, J_2, J_5, J_6 , and J_9 , and four non-ground jobs J_3, J_4, J_7 , and J_8 , as shown in Figure 3(b). That is, regardless of the actual execution times of the jobs, $\mathcal{W}_{p_1}(\lambda_1) = \mathcal{W}(\lambda_1)$, $\mathcal{W}_{p_2}(\lambda_2) = \mathcal{W}(\lambda_2)$ (which is under the assumption that $\mathcal{W}(\lambda_2)$ includes the execution time of J_1 while $\mathcal{W}(\lambda_1)$ does not), $\mathcal{W}_{p_5}(\lambda_5) = \mathcal{W}(\lambda_5)$, $\mathcal{W}_{p_6}(\lambda_6) = \mathcal{W}(\lambda_6)$, and $\mathcal{W}_{p_9}(\lambda_9) = \mathcal{W}(\lambda_9)$. On the contrary, for any of the non-ground jobs J_j , $\mathcal{W}_{p_j}(\lambda_j) \neq \mathcal{W}(\lambda_j)$. For example, $\mathcal{W}_{p_4}(\lambda_4) \neq \mathcal{W}(\lambda_4)$ if J_2 is still running until J_4 is released, since the system backlog $\mathcal{W}(\lambda_4)$ includes the backlog left by J_2 while the p_4 -backlog $\mathcal{W}_{p_4}(\lambda_4)$ does not.

We can capture backlog dependencies between the ground and non-ground jobs. For each non-ground job J_j , we search for the last ground job that is released before J_j and has a priority higher than or equal to that of J_j . Such a ground job is called the *base job* for the non-ground job. From this relation, we can observe that the p_j -backlog of the non-ground job J_j directly depends on that of the base job. For

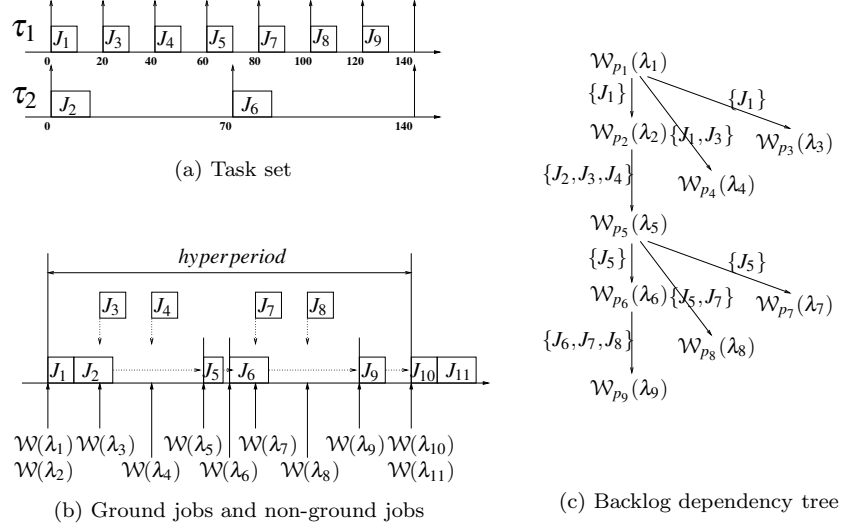


Fig. 3. An example of backlog dependency tree generation

example, for the task set shown above, the base job of J_3 and J_4 is J_1 , and that of J_7 and J_8 is J_5 . We can see that, for the non-ground job J_3 , the p_3 -backlog can be directly computed from that of the ground job J_1 by considering only the execution time of J_1 . In this computation, the existence of J_2 is ignored because J_2 has a lower priority than J_3 . Likewise, for the non-ground job J_4 , the p_4 -backlog can also be directly computed from that of the ground job J_1 in the same manner, except for the fact that, in this case, we have to take into account the arrival of J_3 in between λ_1 and λ_4 (since J_3 has a higher priority than J_4).

Note that such backlog dependencies exist even between ground jobs, and can still be captured under the concept of the base job. The base job of J_2 is J_1 , that of J_5 is J_2 , and so on. As a result, all the backlog dependencies among the jobs can be depicted with a tree, as shown in Figure 3(c). In this figure, each node represents the p_j -backlog $\mathcal{W}_{p_j}(\lambda_j)$ of J_j , each link $\mathcal{W}_{p_k}(\lambda_k) \rightarrow \mathcal{W}_{p_j}(\lambda_j)$ represents the dependency between $\mathcal{W}_{p_k}(\lambda_k)$ and $\mathcal{W}_{p_j}(\lambda_j)$, and the label on each link represents the set of jobs that should be taken into account to compute $\mathcal{W}_{p_j}(\lambda_j)$ from $\mathcal{W}_{p_k}(\lambda_k)$.

It is important to understand that this backlog dependency tree completely encapsulates all the job sequences required in computing the p_j -backlog's $\mathcal{W}_{p_j}(\lambda_j)$ of all the jobs in the hyperperiod. For example, let us consider the path from $\mathcal{W}_{p_1}(\lambda_1)$ to $\mathcal{W}_{p_8}(\lambda_8)$. We can see that the set of labels found in the path represents the exact sequence of jobs that should be considered in computing $\mathcal{W}_{p_8}(\lambda_8)$ from $\mathcal{W}_{p_1}(\lambda_1)$. That is, the job sequence $\{J_1, J_2, J_3, J_4, J_5, J_7\}$ includes all the jobs with a priority higher than or equal to that of J_8 , among all the jobs preceding J_8 . This property is applied for every node in the tree. Therefore, given the stationary root backlog distribution, i.e., $f_{\mathcal{W}_{p_1}(\lambda_1)}(\cdot)$, we can compute the stationary p_j -backlog distributions of all the other jobs in the hyperperiod by traversing the tree while applying the convolve-shrink procedure.

Finally, note that there is one optimization issue in the dependency tree. In the cases of computing $\mathcal{W}_{p_3}(\lambda_3)$ and computing $\mathcal{W}_{p_4}(\lambda_4)$, the associated job sequences are $\{J_1\}$ and $\{J_1, J_3\}$, and the former is a subsequence of the latter. In this case,

since we can obtain $\mathcal{W}_{p_3}(\lambda_3)$ while computing $\mathcal{W}_{p_4}(\lambda_4)$ with the sequence $\{J_1, J_3\}$, i.e., $\mathcal{W}_{p_3}(\lambda_3) = \mathcal{W}_{p_4}(\lambda_3)$, the redundant computation for $\mathcal{W}_{p_3}(\lambda_3)$ with the sequence $\{J_1\}$ can be avoided. This observation is also applied to the case of non-ground jobs J_7 and J_8 . It suffices to note that such redundancies can easily be removed by certain steps of tree manipulation. For more information, refer to Appendix.

E. Extension to dynamic-priority and fixed-priority systems

In this subsection, we will prove the existence of ground jobs for the job-level fixed-priority scheduling model [16]. We will also prove the existence of base jobs while distinguishing between fixed-priority systems and dynamic-priority systems.

Theorem 1. *Let $S = \{\tau_1, \dots, \tau_n\}$ be a periodic task set, in which each task generates a sequence of jobs with a deterministic period T_i and phase Φ_i . Also, let $T_H = \text{lcm}_{1 \leq i \leq n} \{T_i\}$, i.e., the length of a hyperperiod. Consider a sequence of hyperperiods the first of which starts at time t ($0 \leq t < T_H$). Then, for any t , if the relative priorities of all jobs in a hyperperiod $[t + kT_H, t + (k+1)T_H)$ coincide with those of all jobs in the next hyperperiod $[t + (k+1)T_H, t + (k+2)T_H)$ ($k = 0, 1, \dots$), it follows that*

- (a) *at least one ground job exists in any hyperperiod.*
- (b) *the same set of ground jobs are found for all the hyperperiods.**

Proof. (a) Assume that all the jobs have distinct priority values. If there exist jobs with the same priority value, they can always be reassigned distinct priority values while respecting the FCFS (First Come First Serve) principle or a user-defined principle. Then for any hyperperiod k , i.e., $[t + kT_H, t + (k+1)T_H)$, we can find a job J_j with the maximum priority value p_{max} in the hyperperiod. This guarantees that J_j has a higher priority value (or a lower priority) than all the preceding jobs released in $[t + kT_H, \lambda_j)$. Then, since the previous instance of J_j released at time $\lambda_j - T_H$ has a lower priority value than J_j , and any job released in $[\lambda_j - T_H, t + kT_H)$ has a lower priority value than the previous instance, it follows that J_j even has a higher priority value than all the jobs released in $[\lambda_j - T_H, \lambda_j)$. Likewise, it can be shown that J_j has a higher priority value than all the jobs released in $[\lambda_j - 2T_H, \lambda_j)$, $[\lambda_j - 3T_H, \lambda_j)$, and so on. Therefore, J_j is a ground job, and for any hyperperiod, there exists at least one ground job.

(b) This is straightforward from the proof of (a). □

The key point of the proof is that, in any hyperperiod, a job with the maximum priority value always has a lower priority than any preceding jobs. From this, it is easy to devise an algorithm to find all the ground jobs in a hyperperiod. First, we take an arbitrary hyperperiod and simply find the job J_j with the maximum priority value. This job is a ground one. After that, we find all the other ground jobs by searching the single hyperperiod starting at the release time of the ground job, i.e., $[\lambda_j, \lambda_j + T_H)$. In this search, we simply have to check whether a job J_l in the hyperperiod has a greater priority value than all the preceding jobs released in the hyperperiod, i.e., $\{J_j, \dots, J_{l-1}\}$.

*To be precise, the set of jobs that are ground ones agree in all hyperperiods. This does not mean that a task one job of which is classified into a ground job always produces ground jobs. A task may produce both ground jobs and non-ground jobs.

In the following, we will address the existence of the base jobs for dynamic-priority systems such as EDF.

Theorem 2. *For a system defined in Theorem 1, if the priority value $p_j^{(k)}$ of every job $J_j^{(k)}$ in the hyperperiod k (≥ 2) can be expressed as $p_j^{(k)} = p_j^{(k-1)} + \Delta$, where Δ is an arbitrary positive constant, any job in a hyperperiod can always find its base job among the preceding ground jobs in the same hyperperiod or a preceding hyperperiod.*

Proof. Since it is trivial to show that the base job of a ground job can always be found among the preceding ground jobs (actually the base job is the immediately preceding ground job), we focus only on the base job for a non-ground job.

Let us assume a case where the base job for a non-ground job $J_j^{(k)}$ is not found in the same hyperperiod k , and let $J_i^{(k)}$ be a ground job in the hyperperiod that has a higher priority value than the job $J_j^{(k)}$. That is, $J_i^{(k)}$ is not the base job of $J_j^{(k)}$. Then we can always find a previous instance $J_i^{(h)}$ of $J_i^{(k)}$ in a preceding hyperperiod h ($< k$) such that $p_i^{(h)} \leq p_j^{(k)}$, by choosing an appropriate value h that satisfies the inequality $k - h \geq (p_i^{(k)} - p_j^{(k)})/\Delta$. Since $p_i^{(k)} = p_i^{(h)} + (k - h)\Delta$, such a value h always satisfies $p_i^{(h)} \leq p_j^{(k)}$. Then, since $J_i^{(h)}$ is also a ground job (Recall Theorem 1(b)), it can be taken as the base job of $J_j^{(k)}$ if no other eligible ground job is found. Therefore, for any non-ground job J_j , we can always find the base job among the preceding ground jobs. \square

For EDF, $\Delta = T_H$, since the priority value assigned to each job is the absolute deadline.

Note that in our analysis framework it does not matter whether the base job J_i is found in the same hyperperiod (say k) the non-ground job J_j belongs to, or a preceding hyperperiod (say h), since the case where the base job J_i is found in a preceding hyperperiod simply means that the corresponding job sequence from J_i to J_j spans over the multiple hyperperiods from the hyperperiod h to k . Even in this case, since it is possible to compute the stationary backlog distribution for the root of the backlog dependency tree that originates from the hyperperiod h , through the steady-state analysis in Section V, the backlog distribution of such a non-ground job J_j can be computed along the derived job sequence.

The next possible question will be whether there exists a bound on the search range for the base jobs. Theorem 3 addresses this problem for EDF.

Theorem 3. *For EDF, it is always possible to find the base job of any non-ground job J_j in the time window $[\lambda_j - (D^{\max} + T_H), \lambda_j]$, where $D^{\max} = \max_{1 \leq i \leq n} D_i$. That is, the search range for the base job is bounded by $D^{\max} + T_H$.*

Proof. Let τ_i be the task with $D_i = D^{\max}$, and J_k an instance of τ_i . Then J_k is a ground job, since the priority value p_k is $\lambda_k + D^{\max}$, and all the previously released jobs have lower priority values. Let J_j be a non-ground job arriving at the beginning of the hyperperiod $[\lambda_k + D^{\max}, \lambda_k + D^{\max} + T_H]$. Then J_k can be taken as the base job of J_j in the worst case, since J_k is a preceding ground job that has a lower priority value than J_j . Even if we assume that the non-ground job J_j arrives at the end of the hyperperiod, i.e., at time $\lambda_k + D^{\max} + T_H$, J_k can still be taken as the base job

of J_j in the worst case. Therefore, the maximum distance between any non-ground job J_k and its base job cannot be greater than $D^{\max} + T_H$. \square

Note that if we consider a case where $D^{\max} < T_H$ (since the opposite case is rare in practice), Theorem 3 means that it is sufficient to search at most one preceding hyperperiod to find the base jobs of all the non-ground jobs in a hyperperiod.

On the contrary, in fixed-priority systems such as RM and DM, the base jobs of the non-ground jobs do not exist among the ground jobs (Recall that, for such systems, Theorem 2 does not hold, since $\Delta = 0$). In such systems, all jobs from the lowest priority task τ_n are classified as ground jobs while all jobs from the other tasks are non-ground jobs. In this case, since any ground job always has a lower priority than any non-ground job, we cannot find the base job for any non-ground job (even if all the preceding hyperperiods are searched).

Note, however, that this special case does not compromise our analysis framework. It is still possible to compute the backlog distributions of all the jobs by considering each possible priority level. That is, we can consider a subset of tasks $\{\tau_1, \dots, \tau_i\}$ for each priority level $i = 1, \dots, n$, and compute the backlog distributions of all the jobs from task τ_i , since the jobs from τ_i are all ground jobs in the subset of the tasks, and there always exist backlog dependencies between the ground jobs.

Therefore, the only difference between dynamic-priority systems and fixed-priority systems is that for the former the backlog distributions of all the jobs are computed at once with the single backlog dependency tree, while for the latter they are computed by iterative analysis over the n priority levels, which results in n backlog dependency lists.

V. STEADY-STATE BACKLOG ANALYSIS

In this section, we will explain how to analyze the steady-state backlog of a ground job, which is used as the root of the backlog dependency tree or the head of the backlog dependency list. In this analysis, for the ground job J_j , we have to consider an infinite sequence of all the jobs released before J_j , i.e., $\{\dots, J_{j-3}, J_{j-2}, J_{j-1}\}$, since all the preceding jobs contribute to the “system backlog” observed by J_j .

In Section V-A, we will prove the existence of the stationary system backlog distribution, and in Sections V-B and V-C, explain the exact and the approximation methods to compute the stationary distribution. Finally, in Section V-D, it will be discussed how to safely truncate the exact solution, which is infinite, in order to use it as the root of the backlog dependency tree.

A. Existence of the stationary backlog distribution

The following theorem states that there exists a *stationary* (or *limiting*) system backlog distribution, as long as the average system utilization \bar{U} is less than 1.

Theorem 4. *Let us assume an infinite sequence of hyperperiods, the first of which starts at the release time λ_j of the considered ground job J_j . Let $f_{\mathcal{B}_k}(\cdot)$ be the distribution of the system backlog \mathcal{B}_k observed immediate prior to the release time of the ground job $J_j^{(k)}$, i.e., at the beginning of hyperperiod k . Then, if the average system utilization \bar{U} is less than 1, there exists a stationary (or limiting) distribution $f_{\mathcal{B}_\infty}(\cdot)$ of the system backlog \mathcal{B}_k such that*

$$\lim_{k \rightarrow \infty} f_{\mathcal{B}_k} = f_{\mathcal{B}_\infty}.$$

Proof. The proof can be found in [17]. \square

For the special case where $U^{\max} \leq 1$, the system backlog distributions $f_{\mathcal{B}_k}(\cdot)$ of all the hyperperiods are identical. That is, $f_{\mathcal{B}_1} = \dots = f_{\mathcal{B}_k} = \dots = f_{\mathcal{B}_\infty}$. In this case, the stationary backlog distribution $f_{\mathcal{B}_\infty}(\cdot)$ can easily be computed by considering only the finite sequence of the jobs released before the release time of the ground job J_j . That is, we simply have to apply the convolve-shrink procedure along the finite sequence of jobs released in $[0, \lambda_j)$, assuming that the system backlog at time 0 is 0 (i.e., $\mathbb{P}\{\mathcal{W}(0)=0\} = 1$). Therefore, for the special case where $U^{\max} \leq 1$, the following steady-state backlog analysis is not needed.

B. Exact solution

For a general case where $U^{\max} > 1$, in order to compute the exact solution for the stationary backlog distribution $f_{\mathcal{B}_\infty}(\cdot)$, we show that the stochastic process defined with the sequence of random variables $\{\mathcal{B}_0, \mathcal{B}_1, \dots, \mathcal{B}_k, \dots\}$ is a Markov chain. To do this, let us express the PMF of \mathcal{B}_k in terms of the PMF of \mathcal{B}_{k-1} using the concept of conditional probabilities.

$$\mathbb{P}\{\mathcal{B}_k=x\} = \sum_y \mathbb{P}\{\mathcal{B}_{k-1}=y\} \mathbb{P}\{\mathcal{B}_k=x \mid \mathcal{B}_{k-1}=y\} \quad (5)$$

Then we can see that the conditional probabilities $\mathbb{P}\{\mathcal{B}_k=x \mid \mathcal{B}_{k-1}=y\}$ do not depend on k , since all hyperperiods receive the same sequence of jobs with the same execution time distributions. That is, $\mathbb{P}\{\mathcal{B}_k=x \mid \mathcal{B}_{k-1}=y\} = \mathbb{P}\{\mathcal{B}_1=x \mid \mathcal{B}_0=y\}$. This leads us to the fact that the PMF of \mathcal{B}_k depends only on that of \mathcal{B}_{k-1} , and not on those of $\{\mathcal{B}_{k-2}, \mathcal{B}_{k-3}, \dots\}$. Thus, the stochastic process is a Markov chain. We can rewrite Equation (5) in matrix form as follows

$$\mathbf{b}_k = \mathbf{P} \mathbf{b}_{k-1} \quad (6)$$

where \mathbf{b}_k is a column vector $[\mathbb{P}\{\mathcal{B}_k=0\}, \mathbb{P}\{\mathcal{B}_k=1\}, \dots]^\top$, i.e., the PMF of \mathcal{B}_k , and \mathbf{P} is the Markov matrix, which consists of the transition probabilities $\mathbf{P}(x,y)$ defined as

$$\mathbf{P}(x,y) = b_y(x) = \mathbb{P}\{\mathcal{B}_k=x \mid \mathcal{B}_{k-1}=y\} = \mathbb{P}\{\mathcal{B}_1=x \mid \mathcal{B}_0=y\}.$$

Thus, the problem of computing the exact solution $\boldsymbol{\pi}$ for the stationary backlog distribution, i.e., $[\mathbb{P}\{\mathcal{B}_\infty=0\}, \mathbb{P}\{\mathcal{B}_\infty=1\}, \dots]^\top$, is equivalent to solving the equilibrium equation $\boldsymbol{\pi} = \mathbf{P} \boldsymbol{\pi}$.

However, the equilibrium equation $\boldsymbol{\pi} = \mathbf{P} \boldsymbol{\pi}$ cannot be directly solved, since the number of linear equations obtained from it is infinite. Theoretically, when $k \rightarrow \infty$, the system backlog can be arbitrarily long, since $U^{\max} > 1$. This means that the exact solution $\boldsymbol{\pi}$ has an infinite length, and the Markov matrix is therefore also of infinite size. We address this problem by deriving a finite set of linear equations that is equivalent to the original infinite set of linear equations. This is possible due to the regular structure of the Markov matrix proven below.

$$\mathbf{P} = \begin{pmatrix} b_0(0) & b_1(0) & b_2(0) & \dots & b_r(0) & 0 & 0 & 0 \\ b_0(1) & b_1(1) & b_2(1) & \dots & b_r(1) & b_r(0) & 0 & 0 \\ b_0(2) & b_1(2) & b_2(2) & \dots & b_r(2) & b_r(1) & b_r(0) & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & b_r(2) & b_r(1) & \ddots \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots & b_r(2) & \ddots \\ b_0(m_r) & b_1(m_r) & b_2(m_r) & \dots & b_r(m_r) & \vdots & \vdots & \ddots \\ 0 & 0 & 0 & \dots & 0 & b_r(m_r) & \vdots & \ddots \\ 0 & 0 & 0 & \dots & 0 & 0 & b_r(m_r) & \ddots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \ddots \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Each column y in the Markov matrix \mathbf{P} is the backlog PMF observed at the end of a hyperperiod when the amount of the backlog at the beginning of the hyperperiod is y . The backlog PMF of the column y can be calculated by applying the convolve-shrink procedure (in Section IV-B) along the whole sequence of jobs in the hyperperiod, assuming that the initial backlog is equal to y . So, the regular structure found in the Markov matrix, i.e., the columns $r, r+1, r+2, \dots$, with the same backlog PMF only shifted down by one position, means that there exists a value r for the initial backlog from which onwards the backlog PMF observed at the end of the hyperperiod is always the same, only shifted one position to the right in the system. The value r is the maximum sum of all the possible idle times occurring in a hyperperiod. It is equal to

$$r = T_H(1 - U^{\min}) + W^{\min} \quad (7)$$

where W^{\min} is the system backlog observed at the end of the hyperperiod when the initial system backlog is zero and all the jobs have minimum execution times (W^{\min} is usually zero unless most of the workload is concentrated at the end of the hyperperiod). If the initial backlog is r , the whole hyperperiod is busy, and thus the backlog PMF observed at the end of the hyperperiod is simply the result of convolving the execution time distributions of all the jobs, shifted $(T_H - r)$ units to the left. The length of the backlog PMF is $(m_r + 1)$, where m_r is the index of the last non-zero element in column r . This observation is analogously applied to all cases where the initial backlog is larger than r .

Using the above regularity, we can derive the equivalent finite set of linear equations as follows. First, we take the first $(m_r + 1)$ linear equations from $\boldsymbol{\pi} = \mathbf{P}\boldsymbol{\pi}$, which correspond to rows 0 to m_r in the Markov matrix. The number of unknowns appearing in the $(m_r + 1)$ linear equations is $(r + m_r + 1)$, i.e., $\{\pi_0, \pi_1, \dots, \pi_{r+m_r}\}$. Next, we derive r additional equations from the fact that $\pi_x \rightarrow 0$ when $x \rightarrow \infty$, in order to complete the finite set of linear equations, i.e., $(r + m_r + 1)$ linear equations with the $(r + m_r + 1)$ unknowns. For this derivation, from rows $(m_r + 1), (m_r + 2), \dots$ in the Markov matrix, we extract the following equation:

$$\mathbf{Q}_{x+1} = \mathbf{A}\mathbf{Q}_x \quad x \geq m_r + 1 \quad (8)$$

where

$$\mathbf{Q}_x = [\pi_{x-d}, \pi_{x-d+1}, \dots, \pi_{x-1}, \pi_x, \pi_{x+1}, \dots, \pi_{x-d+m_r-1}]^T, \quad (d = m_r - r)$$

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 \\ -b_r(m_r)/b_r(0) & -b_r(m_r-1)/b_r(0) & \dots & -b_r(d+1)/b_r(0) & 1 - b_r(d)/b_r(0) & -b_r(d-1)/b_r(0) & \dots & -b_r(1)/b_r(0) & 1 \end{pmatrix}$$

Then, by diagonalizing the companion-form matrix \mathbf{A} , it can be shown that the general form of π_x is expressed as follows:

$$\pi_x = \sum_{k=1}^{m_r} a_k \lambda_k^{x-m_r-1} \quad (9)$$

where $\{\lambda_1, \lambda_2, \dots, \lambda_{m_r}\}$ are the eigenvalues obtained by the matrix diagonalization, and the associated coefficients a_k are a linear combination of $\{\pi_{r+1}, \pi_{r+2}, \dots, \pi_{r+m_r}\}$. Since it has already been proved in [17] that in Equation (9) there exist $(r+1)$ eigenvalues λ_k such that $|\lambda_k| \geq 1$, the associated coefficients a_k are equated to 0 because the condition that $\pi_x \rightarrow 0$ when $x \rightarrow \infty$ is met only in this case. As a result, $(r+1)$ additional linear equations are obtained, but since one linear equation is always degenerate, r linear equations remain*.

Therefore, the complete set of linear equations is composed of the first $(m_r + 1)$ linear equations taken from $\boldsymbol{\pi} = \mathbf{P}\boldsymbol{\pi}$, and the r linear equations obtained by equating to 0 all the coefficients a_k such that $|\lambda_k| \geq 1$. Then the set of $(r + m_r + 1)$ linear equations with the $(r + m_r + 1)$ unknowns can be solved with a numerical method, and as a result the solution for $\{\pi_0, \pi_1, \dots, \pi_{r+m_r}\}$ is obtained. Once the solution is given, we can complete the general form of π_x , since all the other unknown coefficients a_k , such that $|\lambda_k| < 1$, are calculated from the solution. Therefore, we can finally generate the infinite stationary backlog distribution with the completed general form. For more information about the above process, the reader is referred to [17], [18].

C. Approximated solutions

Markov matrix truncation method: One possible approximation of the exact solution is to truncate the Markov matrix \mathbf{P} to a finite square matrix \mathbf{P}' . That is, we approximate the problem of $\boldsymbol{\pi} = \mathbf{P}\boldsymbol{\pi}$ to $\boldsymbol{\pi}' = \mathbf{P}'\boldsymbol{\pi}'$, where $\boldsymbol{\pi}' = [\pi'_0, \pi'_1, \pi'_2, \dots, \pi'_p]$ and \mathbf{P}' is a square matrix of size $(p+1)$, which consists of the elements $\mathbf{P}(x, y)$ ($0 \leq x, y \leq p$) of the Markov matrix \mathbf{P} . The resulting equation is an eigenvector problem, from which we can calculate the approximated solution $\boldsymbol{\pi}'$ with a numerical method. Among the calculated eigenvectors, we can choose as the solution an eigenvector whose eigenvalue is the closest to 1. In order to obtain a good approximation of the exact solution $\boldsymbol{\pi}$, the truncation point p should be increased as much as possible, which makes the eigenvalue converge to 1.

*The uniqueness of the solution is guaranteed by the ergodicity of the underlying Markov chain. In [17] we proved that the Markov chain is positive recurrent and ergodic

Iterative method: Another approximation method, which does not require the Markov matrix derivation, is simple iteration of the backlog analysis algorithm for the system backlog \mathcal{B}_k over a sufficient number of hyperperiods. Since Theorem 4 guarantees that $f_{\mathcal{B}_k}(\cdot)$ converges towards $f_{\mathcal{B}_\infty}(\cdot)$, we can compute $f_{\mathcal{B}_1}, f_{\mathcal{B}_2}, \dots, f_{\mathcal{B}_k}$, in turn, until convergence occurs. That is, while monitoring the quadratic difference $\|f_{\mathcal{B}_k} - f_{\mathcal{B}_{k-1}}\|$ (def. $\|\mathbf{x} - \mathbf{y}\| = \sqrt{\sum_i (x_i - y_i)^2}$), we can continue the computation of $f_{\mathcal{B}_k}(\cdot)$'s until the difference falls below a given threshold ε .

For both approximation methods, it is important to choose the associated control parameters appropriately, i.e., the truncation point and the number of iterations, respectively. In general, as \bar{U} approaches 1, a larger value should be used for the control parameters, since the probability values of the stationary backlog distribution spread more widely. Note that, by choosing an appropriate value for the control parameters, we can achieve a trade-off between analysis accuracy and the computational overheads required to obtain the approximated solution. We will address this issue in Section VII-A.

D. Safe truncation of the exact solution

As mentioned earlier, the stationary backlog distribution has an infinite length when $U^{\max} > 1$. So, in practice, to use the infinite solution obtained by the exact method as the root of the backlog dependency tree, we have to truncate the solution at a certain point. In this subsection, we show that the use of such a truncated solution is safe in that it is “more pessimistic” than the original infinite solution, thus giving an upper bound on the deadline miss probability for each task.

Let $f'_{\mathcal{B}_\infty}(\cdot)$ be the solution obtained by truncating the original infinite solution at point M . That is, the truncated solution $f'_{\mathcal{B}_\infty}(\cdot)$ is expressed as follows:

$$f'_{\mathcal{B}_\infty}(w) = \begin{cases} f_{\mathcal{B}_\infty}(w) & w \leq M \\ 0 & w > M \end{cases}$$

The truncated solution is not a complete distribution, since the total sum of the nonzero probabilities is less than 1. In other words, the truncated solution has a “deficit” of $\sum_{w>M} f_{\mathcal{B}_\infty}(w)$. However, it is possible to say that $f'_{\mathcal{B}_\infty}(\cdot)$ is more pessimistic than $f_{\mathcal{B}_\infty}(\cdot)$ in the sense that

$$\sum_{w=0}^t f'_{\mathcal{B}_\infty}(w) \leq \sum_{w=0}^t f_{\mathcal{B}_\infty}(w) \quad \text{for any } t.$$

This means that the use of the truncated distribution $f'_{\mathcal{B}_\infty}(\cdot)$ always produces results which are more pessimistic than that of the original distribution $f_{\mathcal{B}_\infty}(\cdot)$. Thus, it leads to a higher deadline miss probability for each task than the original one.

VI. COMPUTATIONAL COMPLEXITY

In this section, we investigate the computational complexity of our analysis framework, dividing it into two parts: (1) the complexity of the backlog and interference analysis, and (2) the complexity of the steady-state backlog analysis. In this complexity analysis, to make the analysis simple and safe, we introduce two assumptions. One assumption is that we regard the deterministic releases of jobs in the

hyperperiod as random releases which follow an interarrival time distribution. So, if the total number of jobs in the hyperperiod is n , the interarrival time distribution is understood as a random distribution with a mean value $\bar{T} = T_H/n$. The other assumption is that all the jobs in the hyperperiod have execution time distributions of the same length m . This simplification is safe, since we can make execution time distributions of different lengths have the same length by zero-padding all the distributions other than the longest one.

A. Complexity of the backlog and interference analysis

To safely analyze the complexity of the backlog analysis, we assume that, for any job J_j in the hyperperiod, all the preceding jobs $\{J_1, \dots, J_{j-1}\}$ are involved in computing the p_j -backlog distribution. That is, it is assumed that the p_j -backlog distribution can only be computed by applying the convolve-shrink procedure to the stationary backlog distribution of J_1 along the whole sequence of preceding jobs. This scenario is the worst case that can happen in computing the p_j -backlog distribution, since the set of jobs required to compute the p_j -backlog distribution does not necessarily cover all the preceding jobs. So, by assuming the worst case scenario for every job J_j in the hyperperiod, we can safely ignore the complex backlog dependencies among the jobs.

Without loss of generality, assuming that the truncated length M of the stationary backlog distribution of J_1 is expressed as a multiple of the execution time distribution length m , i.e. $s \times m$, let us consider the process of applying the convolve-shrink procedure to each job in the sequence $\{J_1, \dots, J_j\}$. Each convolution operation increases the length of the backlog distribution by $(m-1)$ points, and each shrink operation reduces the length by \bar{T} points on average. Note that, if $\bar{T} \approx (m-1)$, the backlog distribution length remains constant on average, and thus the convolve-shrink procedure has the same cost for all the jobs in the sequence. However, if $\bar{T} \rightarrow 0$, which implies that U^{\max} becomes significantly high, the backlog distribution length always increases approximately by m points for each iteration. Assuming this pessimistic case for \bar{T} , the complexity of the j -th iteration of the convolve-shrink procedure is $O((s+j-1)m^2)$, since the j -th iteration is accompanied by convolution between the backlog distribution of length $(s+j-1)m$ and the associated execution time distribution of length m . So, the complexity of computing the single p_j -backlog distribution from the stationary backlog distribution is $sm^2 + (s+1)m^2 + \dots + (s+j-1)m^2$, i.e., $O(j^2m^2)$. Therefore, the total complexity* of computing the p_j -backlog distributions of all the jobs $\{J_1, \dots, J_n\}$ in the hyperperiod is $O(n^3m^2)$.

Likewise, the complexity of the interference analysis can be analyzed as follows. First, let us consider the complexity for a single job J_j . As explained above, the length of the p_j -backlog distribution of J_j for which the interference analysis is to be applied is $(s+j-1)m$, so the initial response time distribution (without any interference) will have a length of $(s+j)m$. We can assume that there exists a constant value k (called *interference degree*) that represents the maximum number of

*In this analysis, we have assumed that the associated backlog dependency tree is completely built by considering only all the jobs in a single hyperperiod. However, if more than one hyperperiod were to be considered for the complete construction of the backlog dependency tree, the term n in $O(n^3m^2)$ should be replaced with the total number of jobs in the multiple hyperperiods.

interfering jobs, within the deadlines, for any job in the hyperperiod. Then the split-convolve-merge procedure is applied k times to the initial response time distribution of J_j . We can see that the convolution at the i -th iteration of the technique has a complexity of $O((l_i - i\bar{T})m)$, where l_i is the length of the response time distribution produced by the $(i - 1)$ -th iteration. That iteration increases the response time distribution by $(m - 1)$ points. So, assuming that $\bar{T} \rightarrow 0$, we can say that the i -th iteration has a complexity of $O((s + j + i)m^2)$, since $l_i = (s + j + i - 1)m$. Thus, the complexity of applying the split-convolve-merge procedure k times to the initial response time distribution is $(s + j)m^2 + (s + j + 1)m^2 + \dots + (s + j + k - 1)m^2$, i.e. $O(k^2m^2)$. Therefore, if we consider all the n jobs in the hyperperiod, the total complexity of the interference analysis is $O(nk^2m^2)$. In particular, by assuming that $k < n$, this complexity can be expressed as $O(n^3m^2)$. This assumption is reasonable, since the fact that $k \geq n$ means that every job in the hyperperiod has a relative deadline greater than or equal to the length of the hyperperiod, which is unrealistic in practice.

B. Complexity of the steady-state backlog analysis

The complexity of the steady-state backlog analysis is different, depending on the solution method used to compute the stationary backlog distribution. First, let us investigate the complexity of the exact method. The exact method consists of three steps: Markov matrix \mathbf{P} derivation, companion-form matrix \mathbf{A} diagonalization, and solving a system of linear equations. The complexity of the Markov matrix derivation is equivalent to that of computing r times the system backlog distribution observed at the end of a hyperperiod from that assumed at the beginning of the hyperperiod, by applying the convolve-shrink procedure along the whole sequence of jobs $\{J_1, \dots, J_n\}$. So, the complexity is $O(rn^2m^2)$, since $O(n^2m^2)$ is the complexity of computing once the system backlog distribution observed at the end of the hyperperiod with n jobs. The complexity of the companion-form matrix \mathbf{A} diagonalization is $O(m_r^3)$, since the diagonalization of a matrix with size l can be solved in time $O(l^3)$ [19]. However, note that m_r is smaller than nm , since $(m_r + 1)$ denotes the length of the backlog distribution obtained by convolving n execution time distributions of length m . So, the complexity of diagonalizing the companion-form matrix \mathbf{A} can be expressed as $O(n^3m^3)$. Finally, the complexity of solving the system of linear equations is $O((m_r + r)^3)$, since a system of l linear equations can be solved in time $O(l^3)$ [20]. This complexity can also be expressed as $O((nm + r)^3)$, since $m_r < nm$. Therefore, the total complexity of the exact method is $O(rn^2m^2) + O(n^3m^3) + O((nm + r)^3)$. This complexity expression can be further simplified to $O(n^3m^3)$ by assuming that $r < nm$. This assumption is reasonable, since $r < T_H = n\bar{T}$ and we can assume that $\bar{T} < m$ when $\bar{T} \rightarrow 0$.

Next, let us consider the complexity of the Markov matrix truncation method. In this case, since the complexity also depends on the chosen truncation point p , let us assume that the value p is given. Then we can see that the complexity* of deriving the truncated Markov matrix \mathbf{P} is $O(pn^2m^2)$, and the complexity of solving

*Note that, when the truncation point p is larger than r , the complexity is reduced to $O(rn^2m^2)$, since the last $(p - r)$ columns in the Markov matrix can be replicated from the r -th column.

the system of p linear equations through matrix diagonalization is $O(p^3)$. Thus, the total complexity is $O(pn^2m^2) + O(p^3)$.

Finally, let us consider the complexity of the iterative method. In this case, the complexity depends on the number of hyperperiods over which the backlog analysis is iterated for convergence. If the number of the hyperperiods is I , the complexity is $O(I^2n^2m^2)$, since the convolve-shrink procedure should be applied to a sequence of In jobs.

However, we cannot directly compare the complexities of all the methods, since we do not know in advance the appropriate values for the control parameters p and I that can give solutions of the same accuracy. In order to obtain insight as to how the control parameters should be chosen, we have to investigate system parameters that can affect the accuracy of the approximation methods. This issue will be addressed in the following section.

VII. EXPERIMENTAL RESULTS

In this section, we will give experimental results obtained using our analysis framework. First, we compare all the proposed solution methods to compute the stationary system backlog distribution, in terms of analysis complexity and accuracy. In this comparison, we vary the system utilization to see its effect on each solution method, and also compare the results with those obtained by Stochastic Time Demand Analysis (STDA) [6], [7]. Secondly, we evaluate the complexity of the backlog and interference analysis by experiments, in order to corroborate the complexity asymptotically analyzed in the previous section. In these experiments, while varying n (the number of jobs), m (the maximum length of the execution time distributions), \bar{T} (the average interarrival time), and k (the interference degree), we investigate their effects on the backlog and interference analysis.

A. Comparison between the solution methods

To investigate the effect of system utilization on each solution method to compute the stationary system backlog distribution, we use the task sets shown in Table I. All the task sets consist of 3 tasks with the same periods, the same deadlines, and null phases, which result in the same backlog dependency tree for a given scheduling algorithm.

The only difference in the task sets is the execution time distributions. For task sets A, B, and C, the minimum and maximum execution times for each task do not change, while the average execution time is varied. In this case, since the time needed for the backlog and interference analysis is constant, if a system backlog distribution of the same length is used as the root of the backlog dependency tree, we can evaluate the effect of the average system utilization \bar{U} on the stationary system backlog distribution. On the other hand, for task sets C, C1, and C2, the average execution time of each task is fixed, while the whole execution time distribution is gradually stretched. In this case, we can evaluate the effect of the maximum system utilization U^{\max} on the stationary system backlog distribution, while fixing the average system utilization \bar{U} .

task set	T_i	D_i	execution times			utilizations			
			C_i^{\min}	C_i	C_i^{\max}	U^{\min}	U	U^{\max}	
A	τ_1	20	20	4	6	10	.58	.82	1.27
	τ_2	60	60	12	16	22			
	τ_3	90	90	16	23	36			
B	τ_1	20	20	4	6	10	.58	.87	1.27
	τ_2	60	60	12	17	22			
	τ_3	90	90	16	26	36			
C	τ_1	20	20	4	7	10	.58	.92	1.27
	τ_2	60	60	12	17	22			
	τ_3	90	90	16	26	36			
C1	τ_1	20	20	3	7	11	.46	.92	1.38
	τ_2	60	60	10	17	24			
	τ_3	90	90	13	26	39			
C2	τ_1	20	20	2	7	12	.34	.92	1.50
	τ_2	60	60	8	17	26			
	τ_3	90	90	10	26	42			

TABLE I
TASK SETS USED IN THE EXPERIMENTS

task set		RM					EDF			
		simulation	STDA	exact	trunc	iterative	simulation	exact	trunc	iterative
A	τ_1	.0000 ± .0000	.0000	.0000			.0001 ± .0000	.0001	.0001	.0001
	τ_2	.0000 ± .0000	.0000	.0000			.0000 ± .0000	.0000	.0000	.0000
	τ_3	.0940 ± .0025	.3931	.0940	.0940	.0940	.0000 ± .0000	.0000	.0000	.0000
B	τ_1	.0000 ± .0000	.0000	.0000			.0013 ± .0002	.0013	.0013	.0013
	τ_2	.0000 ± .0000	.0000	.0000			.0005 ± .0002	.0005	.0005	.0005
	τ_3	.2173 ± .0033	.6913	.2170	.2170	.2170	.0000 ± .0001	.0000	.0000	.0000
C	τ_1	.0000 ± .0000	.0000	.0000			.0223 ± .0013	.0224	.0224	.0224
	τ_2	.0000 ± .0000	.0000	.0000			.0168 ± .0014	.0169	.0169	.0169
	τ_3	.3849 ± .0052	.9075	.3852	.3852	.3852	.0081 ± .0011	.0081	.0081	.0081
C1	τ_1	.0000 ± .0000	.0000	.0000			.0626 ± .0031	.0630	.0627	.0627
	τ_2	.0000 ± .0000	.0000	.0000			.0604 ± .0038	.0610	.0607	.0607
	τ_3	.4332 ± .0065	.9209	.4334	.4334	.4334	.0461 ± .0032	.0466	.0463	.0463
C2	τ_1	.0000 ± .0000	.0000	.0000			.1248 ± .0058	N.A.	.1250	.1250
	τ_2	.0002 ± .0001	.0018	.0002	.0002	.0002	.1293 ± .0064		.1296	.1296
	τ_3	.4859 ± .0081	.9339	N.A.	.4860	.4860	.1136 ± .0063		.1138	.1138

TABLE II
ANALYSIS ACCURACY COMPARISON BETWEEN THE SOLUTION METHODS (DEADLINE MISS PROBABILITY)

Table II summarizes the results of our stochastic analysis and, for the case of RM, also the results obtained by STDA. The table shows the deadline miss probability (DMP) for each task obtained from the stationary system backlog distribution computed by each solution method (i.e., exact, Markov matrix truncation, iterative), and the average deadline miss ratio (DMR) and standard deviation obtained from simulations. For the truncation and iterative methods, the values used for the control parameters p and I are shown in Table III (This will be explained later). The average DMR is obtained by averaging the deadline miss ratios measured from 100 simulation runs of each task set, performed during 5000 hyperperiods. To implement the exact method and the Markov matrix truncation method, we used the Intel linear algebra package called Math Kernel Library 5.2 [21].

From Table II, we can see that our analysis results are almost identical to the simulation results, regardless of the solution method used. For the case of RM, the analysis results obtained by STDA are also given, but we can observe significant

task set	SSBD computation time (seconds)						
	exact	trunc			iterative		
		$\delta=10^{-3}$	$\delta=10^{-6}$	$\delta=10^{-9}$	$\delta=10^{-3}$	$\delta=10^{-6}$	$\delta=10^{-9}$
A	.13	.00	.00	.00	.00	.00	.00
		$p=2$	$p=15$	$p=25$	$I=2$	$I=2$	$I=3$
B	.13	.00	.00	.01	.00	.00	.01
		$p=8$	$p=23$	$p=37$	$I=2$	$I=3$	$I=6$
C	.15	.01	.03	.07	.00	.01	.03
		$p=29$	$p=63$	$p=96$	$I=4$	$I=12$	$I=20$
C1	.31	.02	.10	.25	.01	.05	.21
		$p=54$	$p=115$	$p=173$	$I=7$	$I=20$	$I=35$
C2	N.A.	.07	.31	.82	.02	.23	.88
		$p=86$	$p=181$	$p=272$	$I=10$	$I=30$	$I=52$

TABLE III

ANALYSIS TIME COMPARISON BETWEEN THE SOLUTION METHODS

differences between the DMPs given by STDA and those obtained by our analysis. In the case of task τ_3 in task set A, the DMP given by STDA (39.3%) is more than four times that given by our analysis (9.4%). Moreover, as \bar{U} or U^{\max} increases, the DMP computed by STDA gets even worse. This results from the critical instant assumption made in STDA.

On the other hand, our implementation of the exact method could not provide a numerically valid result for task set C2 (in the case of RM, only for task τ_3). This is because the numerical package we used, which uses the 64-bit floating point type, may result in an ill-conditioned set of linear equations when a significantly small probability value $b_r(0)$ is used as the divisor in making the companion-form matrix \mathbf{A} (Recall from Section V-B that $b_r(0)$ is the probability that all the jobs in the hyperperiod have the minimum execution times). In the case of C2, the probability value $b_r(0)$ was 5×10^{-17} . This is also the reason why a small difference is observed between the DMP computed by the exact method and those computed by the approximation methods for task set C1, scheduled by EDF. However, note that this precision problem can be overcome simply by using a numerical package with a higher precision.

Table III shows in the case of EDF the analysis time* required by each solution method to compute the stationary system backlog distributions used to produce the results in Table II. The analysis time does not include the time taken by the backlog dependency tree generation, which is almost 0, and the time required by the backlog and interference analysis, which is less than 10 ms. The table also shows the values of the control parameters, p and I , used for the truncation and iterative methods. For fair comparison between the two approximation methods, we define an accuracy level δ to be the quadratic difference between the exact solution of the stationary system backlog distribution $SSBD_{exact}$ and the approximated solution computed by either of the methods $SSBD_{approx}$, i.e., $\delta = ||SSBD_{exact} - SSBD_{approx}||$. In the evaluation of δ , however, due to the numerical errors that can be caused by our implementation of the exact method, we do not refer to the solution given by our implementation as $SSBD_{exact}$, but to the solution obtained by infinitely applying the iterative method

*The analysis time was measured with a Unix system call called `times()` on a personal computer equipped with a Pentium Processor IV 2.0 GHz and 256 MB main memory.

to the corresponding task set until the resulting solution converges.

In Table III, we can see both the SSBD computation time and the associated control parameters used to obtain solutions with the required accuracy levels $\delta = 10^{-3}, 10^{-6}, 10^{-9}$ (The DMPs shown in Table II for the truncation and iterative methods were obtained at an accuracy level of $\delta = 10^{-6}$). From the results shown for task sets A to C, we can see that, as \bar{U} increases, the analysis time also rapidly increases for the truncation and iterative methods, while it stays almost constant for the exact method. The reason for this is that as \bar{U} increases, the probability values of the stationary backlog distribution spread more widely, so both approximation methods should compute the solution for a wider range of the backlog. That is, both methods should use a larger value for the associated control parameters, p and I , in order to achieve the required accuracy level. For the exact method, on the contrary, this spread of the stationary probability values does not affect the analysis time, since the method originally derives a general form solution from which the SSBD can be completely generated.

The above observation is analogously applied for the results from task sets C to C2. Due to the increasing U^{\max} , the SSBD spreads even more widely, so the truncation and iterative methods should increase the associated control parameters even more in order to achieve the required accuracy level. We can see that the analysis time taken by the exact method also increases, but this is not because the stationary backlog distribution spreads, but because the size of the resulting companion-form matrix \mathbf{A} becomes large due to the increasing length of the execution time distributions.

In summary, if \bar{U} and/or U^{\max} is significantly high, the approximation methods require a long computation time for high accuracy, possibly larger than that of the exact method. However, if \bar{U} is not close to 1, e.g., less than 0.8, the methods can provide highly accurate solutions at a considerably lower complexity.

B. Complexity evaluation of the backlog and interference analysis

To evaluate the complexity of the backlog and interference analysis, we generated synthetic systems, varying the system parameters n , m , and \bar{T} , while fixing \bar{U} . That is, each system generated is composed of n jobs with the same execution time distribution of length m and mean interarrival time \bar{T} . The shapes of the execution time distribution and the interarrival time distribution of the jobs are determined in such a way that the fixed average system utilization is maintained, even if they have no influence on the complexity of the backlog and interference analysis (Recall that the backlog and interference analysis time is not affected by the actual values of the probabilities composing the distributions; the probability values may only affect the analysis time of the stationary system backlog distribution by changing the average system utilization \bar{U}). We do not have to specify the interference degree k at the synthetic system generation stage, since it can be arbitrarily set prior to interference analysis of the resulting system.

For each system generated, we perform backlog and interference analysis, assuming a null backlog at the beginning of the analysis. For each of the n jobs, we measure the time taken by backlog analysis and interference analysis separately. In this measurement, the backlog analysis time for the j -th job is defined as the time

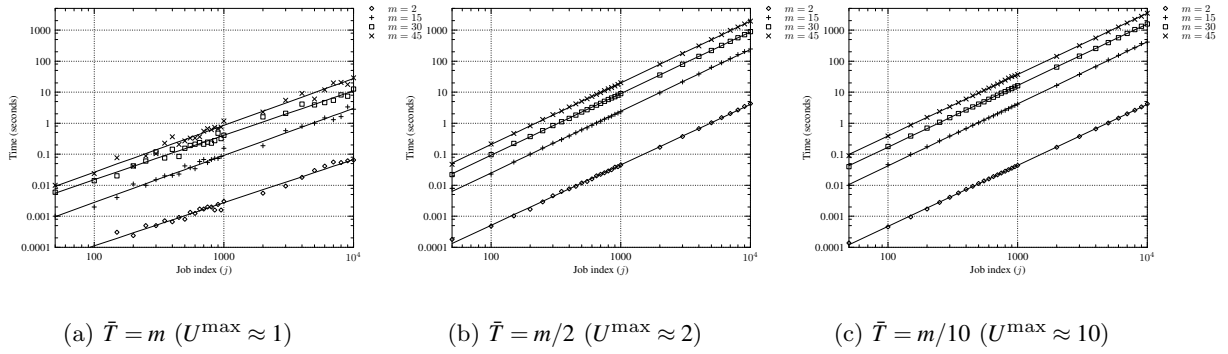


Fig. 4. Backlog analysis time

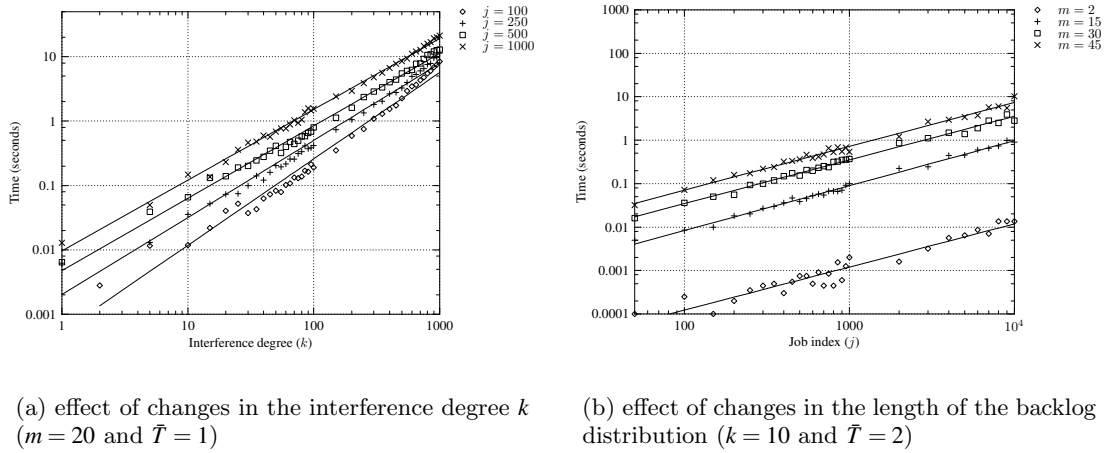


Fig. 5. Interference analysis time

taken by applying the convolve-shrink procedure from the first job J_1 (with the null backlog) to job J_j .

Figure 4 shows the backlog analysis time measured for each job J_j in seconds, while varying m and \bar{T} . Note that both the x-axis and the y-axis are in logarithmic scale. From this figure we can see that the backlog analysis time for each job increases in polynomial order $O(j^2 m^2)$, as analyzed in the previous section. However, note that, due to the backlog dependencies, the backlog analysis for the j -th job may be efficiently performed in a real system by reusing the result of the backlog analysis for some close preceding job J_i ($i < j$). So, the backlog analysis time for real jobs may be significantly lower than that expected from the figure. Moreover, also note that in the case where $\bar{T} = m$, the backlog analysis time slowly increases as the value of j increases, since the backlog distribution length rarely grows due to the large interarrival times of the jobs.

Figure 5(a) shows the interference analysis times measured for the 100th, 250th, 500th, and 1000th jobs in seconds, while only varying the interference degree k . Note that both the x-axis and the y-axis are still in logarithmic scale. From this figure, we can see that the interference analysis time for a single job also increases

in polynomial order $O(k^2m^2)$ as the interference degree increases. Note, however, that the interference degree considered before the deadline is usually very small in practice. On the other hand, Figure 5(b) shows the interference analysis times measured for each job J_j while fixing all the other system parameters. In this figure, we can indirectly see the effect of the length of the p_j -backlog distribution for the j -th job to which the interference analysis is applied. As the p_j -backlog distribution length increases, the interference analysis time also increases, but slowly.

VIII. CONCLUSIONS AND FUTURE WORK

This paper has presented a framework for the stochastic analysis of periodic real-time systems which relaxes the assumption that all tasks need their worst-case execution times, assuming instead that the execution time is a random variable with a known distribution function. For the case in which all the arrival instants are deterministic, we developed a method for deriving the exact response time distribution of each task, even for systems with a maximum utilization greater than 1. Experimental results confirmed the accuracy of the proposed analysis, even for the approximated methods. The computational complexity of the proposed analysis, polynomial with respect to the number of jobs per hyperperiod and the size of the execution time distributions, is still affordable, while allowing accurate deadline miss ratios to be derived in a much shorter time than using simulation and with greater accuracy. The stochastic analysis is directly applicable to multiprocessor systems using a partitioning scheme and common allocation algorithms such as First Fit, Best Fit, etc. In fact, a multiprocessor made up of m processors would behave like m independent uniprocessors.

When the arrival instants are not deterministic (e.g., sporadic tasks) the analysis proposed here is no longer directly applicable. This only apparently reduces the practical applicability of our framework, as in [22] we successfully investigated the possibility of obtaining “safe” *approximations* of the response time distributions instead of the exact distributions. Safe means that the probability of deadline miss derived from the approximated distribution is greater than the exact probability. It can be shown that, if each sporadic task in the system is replaced by a periodic task, with a period equal to the minimum interrelease times, our analysis can be applied to this new system, obtaining a safe approximation of the exact solution.

Future work will focus on further extensions of the framework, in order to address jitter and dependencies between the execution times.

REFERENCES

- [1] L. Liu and J. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of ACM*, vol. 20, no. 1, pp. 46–61, 1973.
- [2] J. P. Lehoczky, L. Sha, and Y. Ding, “The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior,” in *Proc. of the 10th IEEE Real-Time Systems Symposium*, 1989.
- [3] J. P. Lehoczky, “Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines,” in *Proc. of the 11th IEEE Real-Time Systems Symposium*, 1990, pp. 201–209.
- [4] G. Bernat, A. Colin, and S. Petters, “WCET Analysis of Probabilistic Hard Real-Time Systems,” in *Proc. of the 23rd IEEE Real-Time Systems Symposium*, 2002.

- [5] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.-S. Liu, "Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times," in *Proc. of the Real-Time Technology and Applications Symposium*, Chicago, Illinois, May 1995, pp. 164–173.
- [6] M. K. Gardner and J. W. Liu, "Analyzing Stochastic Fixed-Priority Real-Time Systems," in *Proc. of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Mar. 1999.
- [7] M. K. Gardner, "Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems," Ph.D. dissertation, School of Computer Science, University of Illinois, Urbana-Champaign, 1999.
- [8] J. P. Lehoczky, "Real-Time Queueing Theory," in *Proc. of the 17th IEEE Real-Time Systems Symposium*, 1996, pp. 186–195.
- [9] —, "Real-Time Queueing Network Theory," in *Proc. of the 18th IEEE Real-Time Systems Symposium*, 1997, pp. 58–67.
- [10] L. Abeni and G. Buttazzo, "Stochastic Analysis of a Reservation Based System," in *Proc. of the 9th International Workshop on Parallel and Distributed Real-Time Systems*, Apr. 2001.
- [11] A. K. Atlas and A. Bestavros, "Statistical Rate Monotonic Scheduling," in *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998, pp. 123–132.
- [12] J. Leung and J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.
- [13] S. Manolache, P. Eles, and Z. Peng, "Memory and Time-Efficient Schedulability Analysis of Task Sets with Stochastic Execution Times," in *Proc. of the 13th Euromicro Conference on Real-Time Systems*, Jun. 2001, pp. 19–26.
- [14] A. Leulseged and N. Nissanke, "Probabilistic Analysis of Multi-processor Scheduling of Tasks with Uncertain Parameter," in *Proc. of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, Feb. 2003.
- [15] A. Terrasa and G. Bernat, "Extracting Temporal Properties from Real-Time Systems by Automatic Tracing Analysis," in *Proc. of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, Feb. 2003.
- [16] J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.
- [17] J. L. Díaz, J. M. López, and D. F. García, "Stochastic Analysis of the Steady-State Backlog in Periodic Real-Time Systems," Departamento de Informática, University of Oviedo, Tech. Rep., 2003, also available at <http://www.atc.uniovi.es/research/SASS03.pdf>.
- [18] J. L. Díaz, D. F. García, K. Kim, C.-G. Lee, L. LoBello, J. M. López, S. L. Min, and O. Mirabella, "Stochastic Analysis of Periodic Real-Time Systems," in *Proc. of the 23rd Real-Time Systems Symposium*, Austin, TX, USA, 2002, pp. 289–300.
- [19] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed., ser. Johns Hopkins Studies in the Mathematical Sciences. Baltimore, MD, USA: The Johns Hopkins University Press, 1996.
- [20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge University Press, 1992.
- [21] Intel, "Intel Math Kernel Library: Reference Manual," 2001, <http://developer.intel.com/software/products/mkl>.
- [22] J. L. Díaz, J. M. López, M. García, A. M. Campos, K. Kim, and L. LoBello, "Pessimism in the stochastic analysis of real-time systems: Concept and applications," in *Proc. of the 25th Real-Time Systems Symposium*, Lisbon, Portugal, 2004, pp. 197–207.

APPENDIX: BACKLOG DEPENDENCY TREE OPTIMIZATION

Once the backlog dependency tree is derived for a given task set, the stationary p_j -backlog distribution for each job can be computed by traversing the tree while applying the convolve-shrink procedure. However, there may be a number of redundant computations in computing all the stationary p_j -backlog distributions, since the job sequence considered in the computation of the stationary p_j -backlog distribution of a job can have a common prefix with the job sequence for another job. If two separate job sequences share a common prefix, the p_j -backlog analysis for the two jobs can be optimized by eliminating the redundant computations occurring due to the common prefix. In this section, we explain how to remove such redundancies in the backlog dependency tree.

A. Backlog Dependency Tree Expansion

To optimize a backlog dependency tree, we need transform the tree in such a form that the redundancies can easily be eliminated. For example, let us consider the task set shown in Figure 6. This task set consists of three tasks τ_1 , τ_2 , and τ_3 with the relative deadlines equal to the periods, respectively. The phases ϕ_i of all the tasks are zero. We assume that these tasks are scheduled by EDF.

The backlog dependency tree derived for the task set is shown in Figure 7. This tree consists of seven ground jobs $J_{1,1}, J_{2,1}, J_{3,1}, J_{2,2}, J_{3,2}, J_{2,3}$, and $J_{1,9}$, and seven non-ground jobs $J_{1,2}, J_{1,3}, J_{1,4}, J_{1,5}, J_{1,6}, J_{1,7}$, and $J_{1,8}$. In the tree, each node is marked by adding a superscript of '*' to the p_j -backlog notation, i.e., $W_{p_i}^*(\lambda_{i,j})$, so that it can be distinguished from the intermediate nodes that will be introduced in the following.

We transform the backlog dependency tree so that the job sequence on each link can include only one job in the resulting tree. To this end, we expand each link $W_{p_i}(\lambda_i) \xrightarrow{\{J_1, \dots, J_n\}} W_{p_j}(\lambda_j)$ (assume that $J_i = J_1$) to a sequence of links: $W_{p_i}(\lambda_i) \xrightarrow{\{J_1\}} W_{p_j}(\lambda_2) \xrightarrow{\{J_2\}} \dots \xrightarrow{\{J_n\}} W_{p_j}(\lambda_j)$. Each intermediate node introduced, $W_{p_j}(\lambda_k)$ ($k = 1, \dots, n$), is the p_j -backlog observed at the release time of J_k , meaning the backlog observed at λ_k at priority level p_j . Figure 8 shows the result of expanding all the links of the backlog dependency tree in Figure 7.

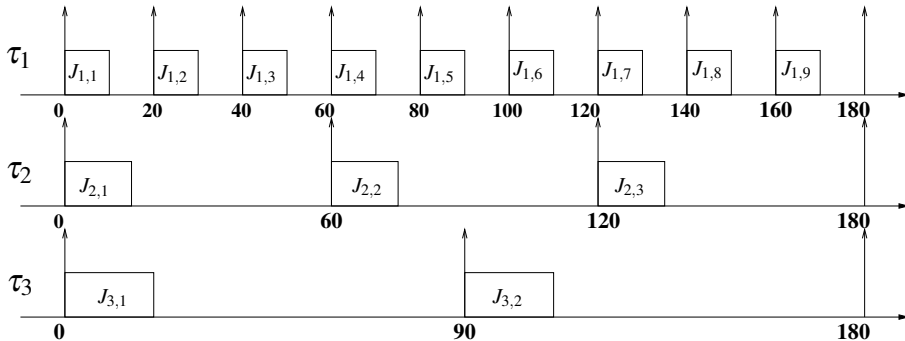


Fig. 6. A task set example

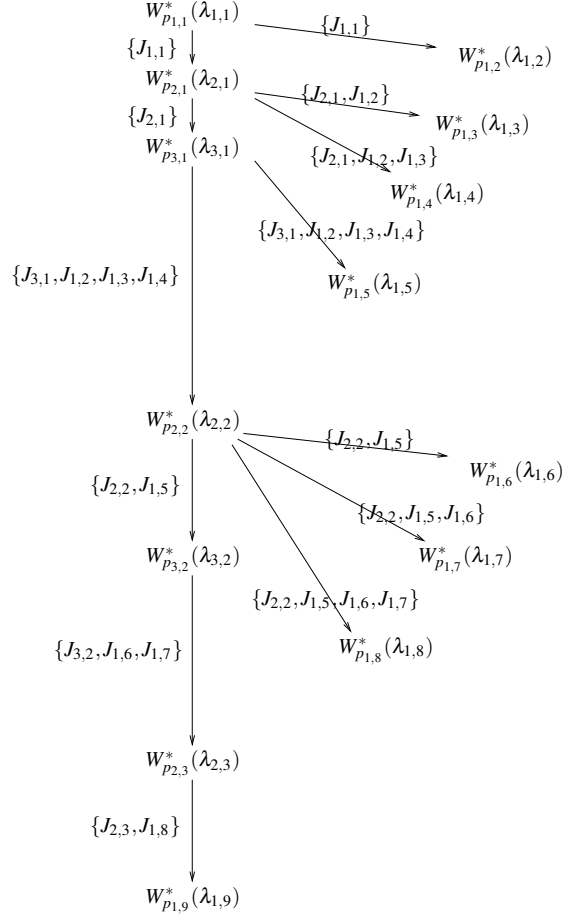


Fig. 7. The derived backlog dependency tree

In the expanded tree, we have arranged all the nodes so that the nodes representing the backlog observed at the same instant but for different priority levels can be displayed in the same row. For the release time of $J_{1,2}$, for example, we can see that there are five different nodes representing the backlog at five different priority levels, $W_{p2,2}^*(\lambda_{1,2})$, $W_{p1,5}^*(\lambda_{1,2})$, $W_{p1,4}^*(\lambda_{1,2})$, $W_{p1,3}^*(\lambda_{1,2})$, and $W_{p1,2}^*(\lambda_{1,2})$. Here, recall that the first four nodes are newly introduced intermediate ones while only the last is an original one, inherited from the original backlog dependency tree.

B. Redundancy Elimination

We can now optimize the expanded backlog dependency tree finding a common prefix shared among job sequences. In the expanded tree, let us consider the path from $W_{p2,1}^*(\lambda_{2,1})$ to $W_{p1,3}^*(\lambda_{1,3})$ and the path from $W_{p2,1}^*(\lambda_{2,1})$ to $W_{p1,4}^*(\lambda_{1,4})$. These two paths have a common prefix in the job sequences, i.e., $\{J_{2,1}, J_{1,2}\}$. Here, we can see that there is no need to separately treat the two paths originating from $W_{p2,1}^*(\lambda_{2,1})$, since $W_{p1,4}^*(\lambda_{1,2}) = W_{p1,3}^*(\lambda_{1,2})$, and $W_{p1,4}^*(\lambda_{1,3}) = W_{p1,3}^*(\lambda_{1,3})$. Thus, we can merge the two paths into a single one; that is, $W_{p2,1}^*(\lambda_{2,1}) \xrightarrow{J_{2,1}} W_{p1,3}^*(\lambda_{1,2}) \xrightarrow{J_{1,2}} W_{p1,3}^*(\lambda_{1,3}) \xrightarrow{J_{1,3}} W_{p1,4}^*(\lambda_{1,4})$ (case of complete match). Likewise, we can merge the

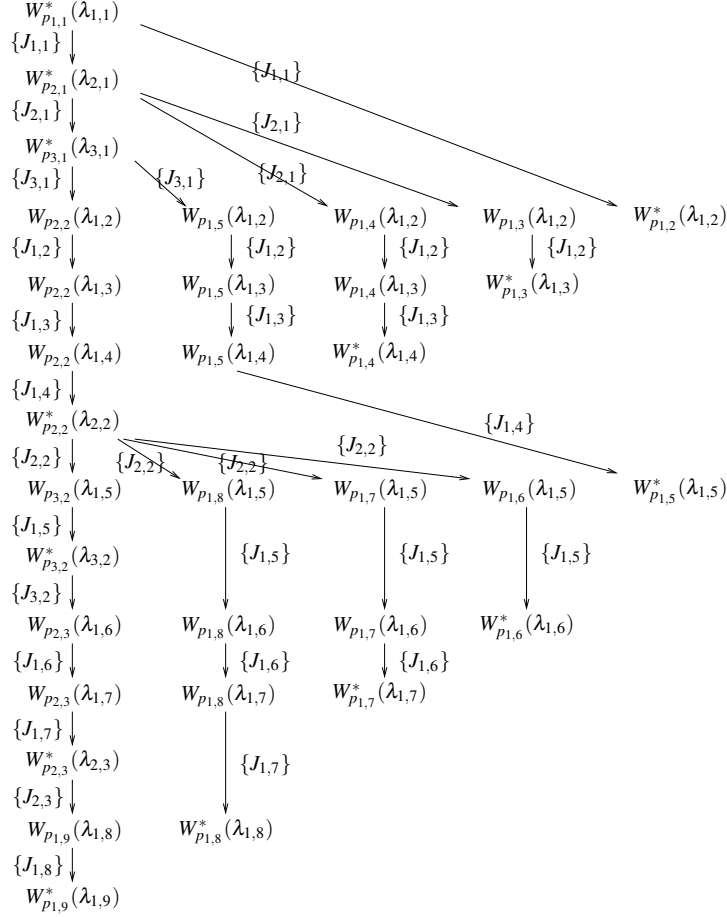


Fig. 8. The expanded backlog dependency tree

path from $W_{p_{3,1}}^*(\lambda_{3,1})$ to $W_{p_{1,5}}^*(\lambda_{1,5})$ into the main stem of the tree, since a common prefix $\{J_{3,1}, J_{1,2}, J_{1,3}, J_{1,4}\}$ exists. However, this case is not the case of complete match, because $W_{p_{1,5}}^*(\lambda_{1,5}) \neq W_{p_{2,2}}^*(\lambda_{2,2})$ due to the different release times $\lambda_{1,5}$ and $\lambda_{2,2}$. Thus, the result of the merging is $W_{p_{3,1}}^*(\lambda_{3,1}) \xrightarrow{J_{3,1}} W_{p_{2,2}}(\lambda_{1,2}) \xrightarrow{J_{1,2}} W_{p_{2,2}}(\lambda_{1,3}) \xrightarrow{J_{1,3}} W_{p_{2,2}}(\lambda_{1,4}) \xrightarrow{J_{1,4}} W_{p_{2,2}}^*(\lambda_{2,2}), W_{p_{1,5}}^*(\lambda_{1,5})$ (case of partial match). Figure 9 shows the final result of merging all the possible paths in the expanded backlog dependency tree.

Figures 10 and 11 show an algorithm to perform the merging process described above. In this algorithm, the following assumptions are made.

- A1 The input to the algorithm is an expanded backlog dependency tree, as shown in Figure 8.
- A2 Each node in the input tree has two types of attributes. One attribute is the *set of its child nodes*, and the other attribute is the *label* attached to the links originating from the node (Note that the labels of all the links originating from a node are the same, and thus there is no need to separately deal with nodes and links). In the case of a leaf node $W_{p_j}^*(\lambda_j)$, which has no link originating from it, the label is assumed to be ' $\{J_j\}$ '.
- A3 An original node $W_{p_j}^*(\lambda_j)$, which existed before the tree expansion, is classified

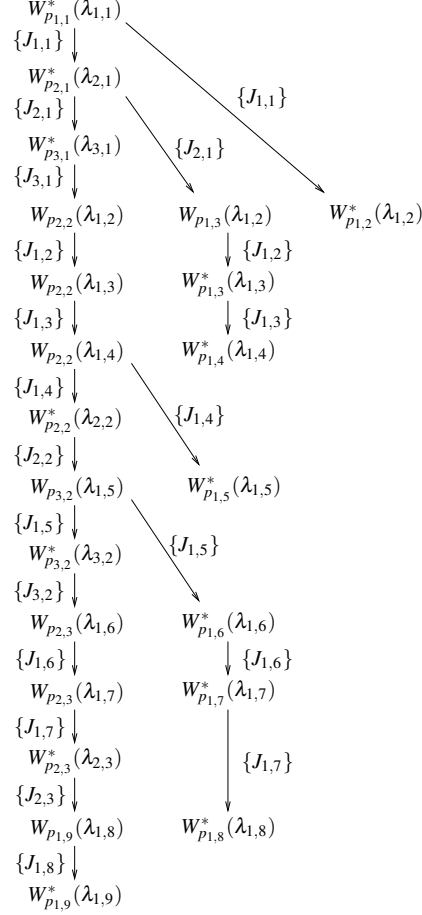


Fig. 9. The optimized backlog dependency tree

as either a *ground node* or a *non-ground node*. A ground node is a node that represents the p_j -backlog distribution of a ground job, and a non-ground node is a node that represents the p_j -backlog distribution of a non-ground job. For the ground node of ground job J_i , the set of child nodes are ordered as follows:

- The first child node is the node through which we can reach the next ground node if any.
- All the other child nodes are ordered so that through the k -th child node, we can reach the non-ground node of the non-ground job that has the k -th latest release time among all the non-ground jobs whose base job is J_i . That is, the non-ground node reachable by the first child node is for the non-ground job whose release time is the latest among all the non-ground jobs whose base job is J_i , and the non-ground node reachable by the last child node is for the non-ground job whose release time is the earliest among all the non-ground jobs.

The expanded backlog dependency tree shown in Figure 8 satisfies the above requirements.

```

01 optimize_backlog_dependency_tree(root_node)
02 {
03   ground_node = root_node;
04   while (num_of_children(ground_node) > 0) {
05     for (i = 1; i < num_of_children(ground_node); i++) { // skip ground_node → child[0].
06       [matching_node1, matching_node2] = find_common_prefix(ground_node, ground_node → child[i]);
07       if (matching_node1 ≠ ground_node) { // a common prefix is found.
08         if (num_of_children(matching_node2) == 0) // case of complete match
09           mark matching_node1 as equivalent to matching_node2;
10         else { // case of partial match
11           child_node = matching_node2 → child[0];
12           detach child_node from the parent node, matching_node2;
13           attach child_node to matching_node1 as a new child node;
14         }
15         child_node = ground_node → child[i];
16         detach child_node from the parent node, ground_node;
17         destroy the entire list starting from child_node;
18       }
19     } // end of for loop
20     if (ground_node is the last ground node)
21       break;
22     ground_node = next_ground_node_of(ground_node); // move to the next ground node.
23   } // end of while loop
24   return;
25 }
26
27 [child_node] = next_ground_node_of(ground_node)
28 {
29   child_node = ground_node → child[0];
30   while (child_node is not a ground node)
31     child_node = child_node → child[0];
32   return child_node;
33 }
34

```

Fig. 10. A backlog dependency tree optimization algorithm (part I)

```

35 [matching_node1, matching_node2] = find_common_prefix(root_node, target_node)
36 {
37     current_node1 = root_node;
38     current_node2 = target_node;
39     child_node = find_child_node_with_common_label(current_node1, current_node2);
40     if (child_node == NULL) { // only root_node is common
41         matching_node1 = root_node;
42         matching_node2 = root_node;
43         return;
44     }
45     current_node1 = child_node;
46     while (num_of_children(current_node2) > 0) {
47         child_node = find_child_node_with_common_label(current_node1, current_node2 → child[0]);
48         if (child_node == NULL)
49             break;
50         current_node1 = child_node;
51         current_node2 = current_node2 → child[0];
52     }
53     matching_node1 = current_node1;
54     matching_node2 = current_node2;
55     return;
56 }
57
58 [child_found] = find_child_node_with_common_label(parent_node, target_node)
59 {
60     child_found == NULL;
61     for (i = 0; i < num_of_children(parent_node); i++) {
62         child_node = parent_node → child[i];
63         if (child_node == target_node) // this may happen only if parent_node is a ground node.
64             break;
65         if (child_node → label == target_node → label) {
66             child_found = child_node;
67             break;
68         }
69     }
70     return child_found;
71 }
72

```

Fig. 11. A backlog dependency tree optimization algorithm (part II)