JAIST Repository
https://dspace.jaist.ac.jp/

Title	DPSC: A Novel Scheduling Strategy for Overloaded Real-Time Systems
Author(s)	CHENG, Zhuo; ZHANG, Haitao; TAN, Yasuo; LIM, Azman Osman
Citation	2014 IEEE 17th International Conference on Computational Science and Engineering (CSE): 1017-1023
Issue Date	2014-12
Type	Conference Paper
Text version	author
URL	http://hdl.handle.net/10119/13472
Rights	This is the author's version of the work. Copyright (C) 2014 IEEE. 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE), 2014, 1017-1023. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Description	

Japan Advanced Institute of Science and Technology

DPSC: A Novel Scheduling Strategy for Overloaded Real-Time Systems

Zhuo Cheng, Haitao Zhang, Yasuo Tan, and Azman Osman Lim
School of Information Science, JAIST
Nomi, Ishikawa 923-1292, Japan
{chengzhuo, zhanghaitao, ytan, aolim}@jaist.ac.jp

***Abstract*—For real-time systems, the correctness of system behavior depends on not only the computed results but also on the time at which results are produced. This requires tasks in such systems to be completed before their deadlines. However, when workload is heavy, the system may become overloaded. Under such condition, some tasks may miss their deadlines. When this problem happens, it is important to minimize the degrees of system performance degradation. To achieve this objective, the design of scheduling algorithm is crucial. In this paper, we focus on designing on-line scheduling algorithm to maximize the total number of tasks that meet their deadlines. The idea of dynamic programming is used to present a dynamic programming scheduling (DPS) algorithm. In each time, DPS makes an optimum choice for currently known task set. As the uncertainty of new arriving tasks, DPS cannot make optimum choice for the set of overall tasks. To deal with this uncertainty, by applying a congestion control mechanism, a dynamic programming scheduling with congestion control (DPSC) is introduced. Three widely used scheduling algorithms and their corresponding deferrable scheduling (DS) methods are discussed and compared with DPSC. Simulation results reveal that DPSC can effectively improve system performance.**

Keywords*—*cyber-physical systems, overloaded real-time systems, dynamic programming, congestion control.

I. Introduction

Cyber-Physical Systems (CPSs) are integrations of computation and physical processes [1]. With their rapid developments, CPSs have enlivened many critical areas for human life such as transportation, energy, and health. In CPS, as the dynamic nature of physical processing, sensitivity to timing and concurrency become central features of system behavior [1] [2]. These features make typical CPSs as multi-tasking real-time systems. In such a system, a task is required to be completed before a specified time instant which is called deadline. The execution order of tasks is set by a scheduler. Under ideal workload condition, scheduler with a proper scheduling algorithm can make all tasks meet their deadlines. However, in practical environment, system workload may vary widely. Once system workload becomes too heavy so that there does not exist a feasible scheduling algorithm can make all the tasks meet their deadlines, we say the system is *overloaded.*

To deal with the overloaded problem, two different kinds of real-time systems, hard and soft real-time systems, have different objectives. In hard real-time systems, a task missing deadline is treated as fatal fault, which may result in catastrophic. For example, in aircraft control system, a task missing deadline may result in aircraft crash. For such a system, the primary objective is to prevent the overloaded problem from occurring [3]. Unlike the hard real-time systems, soft real-time systems tolerate some tasks missing deadlines, and treat task missing deadline as performance degradation. When the overloaded problem happens, the objective is to minimize the degrees of the degradation. To achieve this objective, the design of scheduling algorithms is crucial, as different scheduling algorithms will lead to different degrees of performance degradation.

To design a scheduling algorithm for overloaded soft real-time systems, many objectives described in [5], [6] can be considered. For example, (*i*) maximizing the total number of tasks that meet deadlines, (*ii*) maximizing the effective processor utilization, (*iii*) maximizing the obtained value of completed tasks. The last two objectives are frequently adopted and studied in literature (e.g., [3], [7]). Compared with them, the first one is rarely studied, especially for the design of *on-line scheduling* (scheduler has no knowledge of a task until it makes request to execute). This objective motivates our work. In this paper, we focus on studying on-line scheduling for overloaded real-time systems with uniprocessor. Our objective is to maximize the total number of tasks that meet their deadlines. This objective is reasonable upon the application that when a missed deadline corresponds to a disgruntled customer, and the aim is to keep as many customers satisfied as possible [4].

There are mainly two contributions in this paper. (*i*) Utilize dynamic programming method to present dynamic programming scheduling (DPS) algorithm which can make optimum choice for currently known task set. (*ii*) Extend DPS to dynamic programming scheduling with congestion control (DPSC) in which congestion control mechanism is introduced. With the help of congestion control mechanism, DPSC can dynamically deal with the uncertainty of new arriving tasks. This uncertainty is the biggest challenge in the design of on-line scheduling algorithm. This idea gives a feasible method for on-line scheduling algorithm to meet this challenge.

The remainder of this paper is organized as follows. Section II summarizes the related work in this area. In section III, we present the system model and give the definition of overloaded systems. In section IV, through an example, we have studied the performance of three typical scheduling algorithms, and illustrated the necessity of a new scheduling algorithm. The details of the proposed DPS algorithm are described in section V. The extended algorithm DPSC which includes congestion control mechanism is presented in section VI. Section VII

TABLE I: Symbols and definitions

Symbol	Definition
$\mathcal{T}$	set of real-time tasks
τ_i	real-time task, $\tau_i \in \mathcal{T}$, where i is index of the task
r_i	the request time instant of τ_i
c_i	the execution time of τ_i
d_i	the deadline of τ_i
rc_i	the remaining execution time of τ_i
t	system time instant
$\mathcal{T}^c$	set of successfully completed real-time tasks

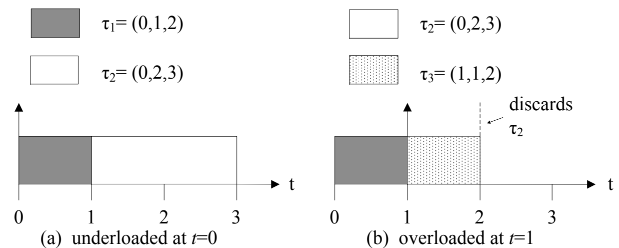

Fig. 1: Example for *underloaded* & *overloaded*

shows the simulation results. Concluding remarks are given in section VIII.

II. Related Work

In the literature on real-time systems, several scheduling algorithms have been proposed to deal with the overloaded problem. Some of them focus on maximizing the obtained value of completed tasks. For example, the Best-Effort approach [13] introduced a rejection policy for overloaded systems based on removing tasks with the minimum value density. It chooses the subset of tasks that maximize the value of the computation for each time unit. In [14], random criticality values are assigned to tasks. The goal is to schedule all of the critical tasks and make sure that the weight of rejected non-critical tasks is minimized. Other approaches focus on providing less stringent guarantees for temporal constraints. The elastic task model (ETM) proposed in [15] aims at increasing task periods to handle overloads in adaptive real-time control systems. In ETM, periodic tasks are able to change their execution rate to provide different qualities of service.

As to the objective that maximizes the number of task completion, some researches focus on special cases of overloaded problem. They impose certain constraints on the values of the task attributes. For example, in [4], a scheduling algorithm is proposed for the special case: equal to request times (ERT). Under that case, all the tasks have the same request time when the system is overloaded. This special case avoids the biggest challenge (the uncertainty of new arriving tasks) in designing on-line scheduling algorithm. Under that case, it is actually an overloaded problem for the design of off-line scheduling algorithm. In our approach, we does not put any constraint on the values of task attributes. By introducing congestion control mechanism, our proposed scheduling algorithm can properly deal with the uncertainty.

III. System Model & Definition

A. Notation and Assumptions

The symbols used throughout the paper are summarized in Table I. We adopt the general *firm-deadline* model proposed in [8] with uniprocessor. This model has been adopted in many studies (e.g., [4], [6]). The "firm-deadline" means only tasks completed before their deadlines are considered valuable, and any task missing its deadline is worthless to system.

The real-time system comprises a set of aperiodic real-time tasks waiting to execute. These tasks request processor to execute when they arrive in system. Each task τ_i is a 3-tuple $\tau_i = (r_i, c_i, d_i)$, where i is the index of a task, r_i is the request time instant, c_i is the required execution time, and d_i is the deadline. Symbol $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denotes the set of tasks comprised in the system, where n is the number of tasks. Task set $\mathcal{T}$ varies with the passage of time. At system time t, $\forall \tau_i \in \mathcal{T}$ meets $r_i \geq t$. Symbol rc_i represents remaining execution time of task τ_i. Initially, it equals to c_i. After τ_i has been executed for mc_i $(mc_i \leq c_i)$ time units, $rc_i = c_i - mc_i$. If $rc_i = 0$, it means τ_i has been completed. The set of successfully completed tasks $\mathcal{T}^c$ comprise all the tasks that meet their deadlines. It features that $\forall \tau_i \in \mathcal{T}^c$ has been executed c_i time units during time interval $[r_i, d_i)$. Note that, if $rc_i > d_i - t$, task τ_i should be discarded immediately, as such task cannot be able to complete successfully.

The assumptions that apply to the system model are as follows: (*i*) The scheduler can learn of a task's attributes at the time instant when it makes request, nothing is known about a task before this time. (*ii*) A task being executed on processor can be preempted by another task at any time instant, and there is no associated cost with such preemption. (*iii*) Every task is independent with the others. There is no prior bound on the time instant and number of takes which request to execute.

B. Definition

A real-time system can use different scheduling algorithms to schedule tasks in $\mathcal{T}$.

Definition ([9], [10]). *When there exists a scheduling algorithm can make all tasks meet their deadlines, the system is* ***underloaded****, and the task set is* ***feasible****. On the contrast, when there does not exist a scheduling algorithm can make all the tasks meet their deadlines, the system is* ***overloaded****, and the task set is* ***infeasible****.*

An example in Fig. 1 is used to elaborate this definition. As shown in Fig. 1 (a), at $t = 0$, $\mathcal{T} = \{\tau_1, \tau_2\}$, using earliest deadline first (EDF) algorithm to schedule $\mathcal{T}$ can make all tasks meet their deadlines, where EDF first schedules the task with the earliest deadline. Thus, the system is underloaded, and the task set is feasible. EDF algorithm proposed in literature [11] has been proven as an *optimal* scheduling algorithm. That is, if using EDF to schedule a task set cannot make all tasks meet their deadlines, no other algorithms can. Thus, EDF scheduling algorithm can be used to tell if a task set is feasible. After system passed a time unit, at $t = 1$. As shown in Fig. 1 (b), τ_1 has been successfully completed, and a new task τ_3 arrives in the system. At that time, $\mathcal{T} = \{\tau_2, \tau_3\}$. Using EDF to schedule $\mathcal{T}$ can only make τ_3 meet its deadline. Task τ_2 should be discarded at $t = 2$, as $rc_2 > d_2 - t$, where $d_2 = 3, rc_2 = 2$. Thus, the system is overloaded, and the task set is infeasible.

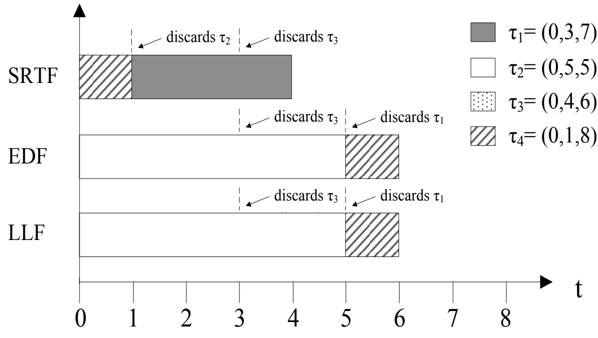

Fig. 2: Performance of scheduling algorithms

IV. Preliminary

There are many scheduling algorithms widely used in various real-time systems. Three representative scheduling algorithms are adopted as the baseline algorithms: shortest remaining time first (SRTF), EDF, and least laxity first (LLF). In this section, we use an example described in Fig. 2 to study their performance.

A. Results

SRTF: It first schedules the task with the shortest remaining time. The scheduling sequence is $\langle \tau_4, \tau_1 \rangle$. By this sequence, τ_4 and τ_1 can be completed sequentially, while τ_2 and τ_3 will miss their deadlines. SRTF schedules a task based on its "length", while another kind of scheduling algorithms uses "urgency" to schedule tasks. There are two algorithms using this method.

EDF: It uses deadline to indicate the urgency of a task. A task with shorter deadline will be scheduled first. It can complete τ_2 and τ_4 while makes τ_3 and τ_1 miss their deadlines.

LLF: Another method to indicate the urgency of a task is using laxity, which is a measure of the spare time that a task has. For τ_i, the laxity l_i is computed as $l_i = d_i - rc_i - t$. LLF first schedules task with least laxity. Notice that, at $t = 2$, τ_2 and τ_3 have the same least laxity. It is reasonable to select the task with the shortest remaining time to execute, thus τ_2 is selected to execute. Same situation happens at $t = 4$, based on the same criterion, τ_2 is selected. It can complete τ_2 and τ_4 while makes τ_3, τ_1 miss their deadlines.

B. Criterion

For the given task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ at $t = 0$, all the three scheduling algorithms can complete two tasks. In this regard, the performance of these three algorithms is the same. However, all these results are obtained based on current knowledge of task set without consideration of the impact of new arriving tasks. In practical environment, when there are new tasks arriving, the performance of the three algorithms may be different. Consider a scenario. A new task $\tau_5 = (4, 1, 5)$ arrives in system at $t = 4$. SRTF can complete three tasks τ_4, τ_1, τ_5 while EDF and LLF can only complete two tasks τ_2, τ_4 or τ_5, τ_4 (either is possible). The reason that SRTF can complete one more task is because SRTF can complete the two tasks within less time slots. When τ_5 arrives, SRTF has already completed τ_4 and τ_1, while LLF and EDF have not completed τ_2 and τ_4, which results in either τ_2 or τ_5 missing deadline. Based on this observation, we come to a criterion.

Criterion. A task set can be scheduled by different scheduling algorithms, when these algorithms can complete the same number of tasks, the one that can complete this number of tasks within less time slots makes better performance.

Based on this criterion, SRTF is considered to make better performance than EDF and LLF. All of the three scheduling algorithms achieve two as the value of task completion number. We wonder if it is the maximum value. For this simple example with only four tasks, we can enumerate all the subset of the given ready task set, and use EDF to tell if the subset of tasks is feasible. By this way, we can find three is the maximum value of task completion number with the scheduling sequence $\langle \tau_3, \tau_1, \tau_4 \rangle$. Through this example, we can see that, for overloaded real-time system, a new scheduling algorithm is needed. This motives our work. A novel dynamic programming scheduling algorithm is proposed in next section.

V. Dynamic Programming Scheduling

For a given set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ at system t, to maximize the number of task completion is a procedure to find the optimal solution of scheduling $\mathcal{T}$ to maximize $|\mathcal{T}^c|$. This procedure can be treated as choosing tasks from $\mathcal{T}$, and putting it into $\mathcal{T}^c$. Because only a task fully completed before its deadline is considered successfully completed, once choosing a task $\tau_i \in \mathcal{T}$, and putting it into $\mathcal{T}^c$, it means allocating rc_i idle time slots before d_i to τ_i. This optimization problem is similar to *0-1 knapsack problem*. Here, we use dynamic programming scheduling (DPS) to find the optimal solution.

A. Sets

1) $\mathcal{S}$: Define $\mathcal{S}[i, j]$ to be a list used to store the scheduling sequence for task set $\mathcal{T}_i \subseteq \mathcal{T}$ within the number of time slots no more than j. Task set $\mathcal{T}_i = \{\tau_1, \tau_2, \ldots, \tau_i\}$, where i is the number of task in $\mathcal{T}_i$, $0 \le i \le n$. Note $0 \le j \le \texttt{d_max}(\mathcal{T}) - t$, where d_max returns the maximum value of d_i of $\tau_i \in \mathcal{T}$, and each task τ_i in $\mathcal{S}[i, j]$ is allocated rc_i time slots.

2) $\mathcal{N}$: Define $\mathcal{N}[i, j]$ to be the number of task completion with scheduling sequence in $\mathcal{S}[i, j]$. That is $\mathcal{N}[i, j] = |\mathcal{T}_{i,j}|$, where $\mathcal{T}_{i,j}$ is a task set which stores all the tasks in $\mathcal{S}[i, j]$.

3) $\mathcal{L}$: Define $\mathcal{L}[i, j]$ to be the total number of time slots used by $\mathcal{S}[i, j]$ to complete all the tasks in $\mathcal{T}_{i,j}$. That is, $\mathcal{L}[i, j] = \sum rc_i$ for $\forall \tau_i \in \mathcal{T}_{i,j}$.

B. Optimal Targets

Our objective can be interpreted as finding $\mathcal{S}[n, \texttt{d_max}(\mathcal{T}) - t]$ to maximize $\mathcal{N}[n, \texttt{d_max}(\mathcal{T}) - t]$. When there exists more than one scheduling sequence can achieve the same maximum value of $\mathcal{N}[n, \texttt{d_max}(\mathcal{T}) - t]$, the one with smaller value of $\mathcal{L}[n, \texttt{d_max}(\mathcal{T}) - t]$ should be chosen. This criterion has been described in section IV-B.

C. Details

The details of finding the optimal solution are summarized in Alg. 1. Note $\forall \tau_i \in \mathcal{T}$ represents tasks which are waiting to execute, and features $rc_i > 0, d_i - t \ge rc_i, r_i \ge t$. The elements in the three sets are computed recursively. We obtain the element values with the increasing order of i, and with

Algorithm 1 Dynamic Programming Scheduling (DPS)

Input: $\mathcal{T}$, with known of rc_i, for all $\tau_i \in \mathcal{T}$
Output: scheduling sequence $\mathcal{S}[n, t_{max}]$, where $n = |\mathcal{T}|, t_{max} = \mathtt{d_max}(\mathcal{T}) - t$

```
1: sort T by non-descending order of d_i, such that ⟨τ_1, τ_2, ..., τ_n⟩ is a permutation
   of the tasks in T with d_i ≤ d_{i+1} for all i, 1 ≤ i < n;
2: for all 0 ≤ j ≤ t_max do
3:     N[0, j] := 0, L[0, j] := 0, S[0, j] := ⟨⟩;
4: end for
5: for all 1 ≤ i ≤ n do
6:     for all 0 ≤ j ≤ t_max do
7:         if j < rc_i then
8:             N[i, j] := N[i − 1, j], L[i, j] := L[i − 1, j], S[i, j] := S[i − 1, j]
9:         else if rc_i ≤ j ≤ d_i − t then
10:            (p, N[i, j]) := max(N[i − 1, j − rc_i] + 1, N[i − 1, j])
11:            if p = 1 then
12:                L[i, j] := L[i − 1, j − rc_i] + rc_i, S[i, j] := S[i − 1, j −
                   rc_i].push_back(τ_i)
13:            else if p = 2 then
14:                L[i, j] := L[i − 1, j], S[i, j] := S[i − 1, j]
15:            else
16:                (q, L[i, j]) := min(L[i − 1, j − rc_i] + rc_i, L[i − 1, j])
17:                if q = 2 then
18:                    S[i, j] := S[i − 1, j]
19:                else
20:                    S[i, j] := S[i − 1, j − rc_i].push_back(τ_i)
21:                end if
22:            end if
23:        else
24:            N[i, j] := N[i, d_i − t], L[i, j] := L[i, d_i − t], S[i, j] := S[i, d_i − t]
25:        end if
26:    end for
27: end for
28: return S[n, t_max]
```

the same i, obtained with the increasing order of j (line 5-6). We first sort $\mathcal{T}$ by non-descending order of d_i such that $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ is a permutation of the tasks in $\mathcal{T}$ with $d_i \leq d_{i+1}$ for all i, $1 \leq i < n$ (line 1). For the computation of $\mathcal{S}[i, j]$, based on the relation between the number of available time slots, *i.e.*, j and the interval $[rc_i, d_i - t]$, where $d_i - t$ represents the maximum number of time slots that can be allocated to τ_i, there are three different situations.

1) $j < rc_i$: It means τ_i cannot be inserted into $\mathcal{S}[i, j]$. Thus the results are the same as $\mathcal{S}[i-1, j]$, which uses the same number of time slots to schedule the task set without τ_i, *i.e.*, $\mathcal{T}_{i-1}$, where symbol $\mathcal{T}_i := \{\tau_1, \tau_2, \ldots, \tau_i\}$ denotes the set of first i tasks. Thus, $\mathcal{S}[i, j] := \mathcal{S}[i-1, j], \mathcal{N}[i, j] := \mathcal{N}[i-1, j], \mathcal{L}[i, j] := \mathcal{L}[i-1, j]$ (line 8).

2) $j > d_i - t$: As only time slots that before d_i can be allocated to τ_i, $d_i - t$ represents the maximum number of time slots that can be allocated to τ_i. As τ_i has the maximum value of d_i of τ_i in $\mathcal{T}_i$, which means it has the maximum number of time slots that can be allocated. That is, when the number of available time slots exceeds $d_i - t$, the tasks in $\mathcal{T}_i$ can only be allocated $d_i - t$ number of time slots. Thus, $\mathcal{S}[i, j] := \mathcal{S}[i, d_i - t], \mathcal{N}[i, j] := \mathcal{N}[i, d_i - t], \mathcal{L}[i, j] := \mathcal{L}[i, d_i - t]$ (line 24).

3) $rc_i \leq j \leq d_i - t$: Under this situation, two choices can be made for $\mathcal{S}[i, j]$, *i.e.*, including τ_i or not. We make the decision between the two choices based on the computed value of $\mathcal{N}[i, j]$. The one that leads larger value will be chosen (line 10). If both choices lead the same values, we choose the one that can result smaller value of $\mathcal{L}[i, j]$ (line 16). Function $\mathtt{max}(a, b)$ (line 10) returns a vector (i, v), where i is the index of larger value. If a is larger, i equals to 1. When b gets the larger value, i equals to 2. If a equals to b, assign i as -1. The Value of v is the larger value from $\{a, b\}$. If a equals to b,

i \ j	0	1	2	3	4	5	6	7	8
0 (null)	0 (0)		0 (0)		0 (0)	0 (0)			
1 (τ_2)	0 (0)		0 (0)		0 (0)	1 (5)	1 (5)		
2 (τ_3)					1 (4)		1 (4)	1 (4)	
3 (τ_1)								2 (7)	2 (7)
4 (τ_4)									3(8)

Fig. 3: Results of dynamic programming scheduling

v equals to them. Function $\mathtt{mim}(a, b)$ (line 16) has the similar operation, the difference is it returns the vector for smaller value.

For the first choice, when we insert τ_i into the list (line 12), it will occupy rc_i time slots, such that, the remaining time slots for the other available tasks, (tasks in $\mathcal{T}_{i-1}$) is $j - rc_i$. The value of task completion for tasks in $\mathcal{T}_{i-1}$ is $\mathcal{N}[i-1, j-rc_i]$. When we add one to this value, for the completion of τ_i, we get the result, *i.e.*, $\mathcal{N}[i, j] = \mathcal{N}[i-1, j-rc_i] + 1$. With the similar analysis, we can get the results as: $\mathcal{L}[i, j] = \mathcal{L}[i-1, j-rc_i] + rc_i$, and $\mathcal{S}[i, j] = \mathcal{S}[i-1, j-rc_i].\mathtt{push_back}(\tau_i)$, where $\mathcal{S}[i, j].\mathtt{push_back}(\tau_i)$ inserts τ_i to the end of the list $\mathcal{S}[i, j]$.

For the second choice, if τ_i is not added in the list (line 14), it means all the j time slots can be allocated to the other available tasks (tasks in $\mathcal{T}_{i-1}$). The value of task completion for tasks in $\mathcal{T}_{i-1}$ is $\mathcal{N}[i-1, j]$. As τ_i is not added in the list, thus $\mathcal{N}[i, j] = \mathcal{N}[i-1, j]$. With the similar analysis, we can get the results as: $\mathcal{L}[i, j] = \mathcal{L}[i-1, j]$, and $\mathcal{S}[i, j] = \mathcal{S}[i-1, j]$.

For time complexity, as double loops are used, the worst-case time complexity of DPS is $O(n \cdot t_{max})$, where $t_{max} = \mathtt{d_max}(\mathcal{T}) - t$. It is pseudo-polynomial time complexity.

D. Results

As to the example of task set depicted in Fig. 2, the computed results by Alg. 1 is shown in Fig. 3. Only results that are needed to compute the scheduling sequence are shown. In this example, task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, and the permutation of tasks in $\mathcal{T}$ by non-descending order of d is $\langle \tau_2, \tau_3, \tau_1, \tau_4 \rangle$. The first row of the table represents the index j, and the first column represents the index i corresponding to the permutation, and the corresponding task is shown in the bracket. The thick black box in the row i, $1 \leq i \leq 4$ denotes the interval $rc_i \leq j \leq d_i - t$, here $t = 0, rc_i = c_i$. The element at position (i, j) is the results of $\mathcal{N}[i, j]$, and $\mathcal{L}[i, j]$ (in the bracket).

From table, we can see the maximum number of task completion is $\mathcal{N}[4, 8] = 3$, the corresponding number of time slots used to complete such number of task is $\mathcal{L}[4, 8] = 8$. The $\mathcal{S}[4, 8]$ is denoted by the lines with arrows. The line with arrow from position $(1, 0)$ to $(2, 4)$ represents the value of the elements at $(2, 4)$ is computed from $(1, 0)$, that is, $\mathcal{N}[2, 4] = \mathcal{N}[1, 0] + 1$, $\mathcal{L}[2, 4] = \mathcal{L}[1, 0] + rc_3$, and $\mathcal{S}[2, 4] = \mathcal{S}[1, 0].\mathtt{push_back}(\tau_3)$. From the three lines with arrows, we can see τ_3, τ_1, τ_4 are inserted into list $\mathcal{S}[4, 8]$ in turn. Thus, the result of scheduling sequence is $\langle \tau_3, \tau_1, \tau_4 \rangle$, which is the optimal solution.

Each time, DPS scheduling tasks makes optimum choice for currently known task set. However, as scheduler has no

TABLE II: Result of DPS without limit of $\mathcal{S}[n, t_{max}]$

t	$\mathcal{T}$	$\mathcal{S}[n, t_{max}]$	$\mathcal{T}^c$
0	$\{\tau_1, \tau_2, \tau_3, \tau_4\}$	$\langle \boldsymbol{\tau_3}, \tau_1, \tau_4 \rangle$	$\varnothing$
1	$\{\tau_1, \tau_3, \tau_4\}$	$\langle \boldsymbol{\tau_3}, \tau_1, \tau_4 \rangle$	$\varnothing$
2	$\{\tau_1, \tau_3, \tau_4, \tau_5\}$	$\langle \boldsymbol{\tau_5}, \tau_1, \tau_4 \rangle$	$\varnothing$
3	$\{\tau_1, \tau_4, \tau_6\}$	$\langle \boldsymbol{\tau_6}, \tau_1, \tau_4 \rangle$	$\{\tau_5\}$
4	$\{\tau_1, \tau_4, \tau_7\}$	$\langle \boldsymbol{\tau_7}, \tau_4 \rangle$	$\{\tau_5, \tau_6\}$
5	$\{\tau_4\}$	$\langle \boldsymbol{\tau_4} \rangle$	$\{\tau_5, \tau_6, \tau_7\}$
6	$\varnothing$	$\varnothing$	$\{\tau_5, \tau_6, \tau_7, \tau_4\}$

TABLE III: Result of DPS with limit of $\mathcal{S}[n, t_{max}]$

t	$\mathcal{T}$	$\mathcal{S}[n, t_{max}]$	$\mathcal{T}^c$
0	$\{\tau_1, \tau_2, \tau_3, \tau_4\}$	$\langle \boldsymbol{\tau_1}, \tau_4 \rangle$	$\varnothing$
1	$\{\tau_1, \tau_3, \tau_4\}$	$\langle \boldsymbol{\tau_1}, \tau_4 \rangle$	$\varnothing$
2	$\{\tau_1, \tau_3, \tau_4, \tau_5\}$	$\langle \boldsymbol{\tau_5}, \tau_1 \rangle$	$\varnothing$
3	$\{\tau_1, \tau_4, \tau_6\}$	$\langle \boldsymbol{\tau_6}, \tau_1 \rangle$	$\{\tau_5\}$
4	$\{\tau_1, \tau_4, \tau_7\}$	$\langle \boldsymbol{\tau_7}, \tau_1 \rangle$	$\{\tau_5, \tau_6\}$
5	$\{\tau_1, \tau_4\}$	$\langle \boldsymbol{\tau_1}, \tau_4 \rangle$	$\{\tau_5, \tau_6, \tau_7\}$
6	$\{\tau_4\}$	$\langle \boldsymbol{\tau_4} \rangle$	$\{\tau_5, \tau_6, \tau_7, \tau_1\}$
7	$\varnothing$	$\varnothing$	$\{\tau_5, \tau_6, \tau_7, \tau_1, \tau_4\}$

knowledge of a task until it arrives in the system, it is doubtful that DPS can make optimum choice for the set of overall tasks.

VI. Dynamic Programming Scheduling with Congestion Control

DPS schedules tasks based on the sequence stored in list $\mathcal{S}[n, t_{max}]$, where $n = |\mathcal{T}|, t_{max} = \mathtt{d_max}(\mathcal{T}) - t$. It expects all tasks in $\mathcal{S}[n, t_{max}]$ can all be completed. Nevertheless, when system is overloaded, the selected tasks usually cannot all be successfully completed. This observation gives the idea that the capacity of $\mathcal{S}[n, t_{max}]$ (i.e., the maximum number of tasks that can be added into $\mathcal{S}[n, t_{max}]$) should be limited based on the completion condition of tasks. If the number of tasks in $\mathcal{S}[n, t_{max}]$ ($\mathcal{N}[n, t_{max}]$) exceeds the capacity, based on criterion described in section IV-B, longest task should be first deleted from $\mathcal{S}[n, t_{max}]$ until meet the capacity requirement. When remaining tasks have the same length (rc_i), the tasks that first added into $\mathcal{S}[n, t_{max}]$ should be remained. This procedure will be shown through an example, which will reveal the benefit of limiting capacity.

A. DPS with limit capacity of $\mathcal{S}[n, t_{max}]$

Recall the example depicted in Fig. 2, and consider a scenario that there are three tasks $\tau_5 = (2, 1, 3)$, $\tau_6 = (3, 1, 4)$, and $\tau_7 = (4, 1, 5)$ arriving in system during time interval $[0, 8)$. The scheduling results using DPS are shown in Table II. The bold symbols in column $\mathcal{S}[n, t_{max}]$ denote which task is scheduled at corresponding time. Four tasks $\mathcal{T}^c = \{\tau_5, \tau_6, \tau_7, \tau_4\}$ can be successfully completed. It can be seen that tasks τ_1 and τ_3 are in $\mathcal{S}[n, t_{max}]$ at $t = 0$. However, both of them have not been successfully completed. If the capacity of $\mathcal{S}[n, t_{max}]$ is limited to 2, we can get the scheduling result depicted in Table III. Five tasks $\mathcal{T}^c = \{\tau_5, \tau_6, \tau_7, \tau_1, \tau_4\}$ have been completed. All the tasks that were added into $\mathcal{S}[n, t_{max}]$ have been successfully completed.

By comparing $\mathcal{S}[n, t_{max}]$ obtained under two conditions, we can see that under first condition, without limit of $\mathcal{S}[n, t_{max}]$, DPS allocates first two time slots to τ_3. However, at $t = 3$, τ_3 is preempted by a new arriving task τ_5, which makes τ_3 fail to complete. For the second condition, at $t = 0$, although τ_3 can be allocated enough idle time slots, due to the limit of $\mathcal{S}[n, t_{max}]$, only the first two tasks τ_4, τ_1 have been added into $\mathcal{S}[n, t_{max}]$. It makes the first two time slots allocated to τ_1, a shorter task than τ_3, which makes τ_1 successfully complete.

Algorithm 2 `ComputeWS`()

```
1: if a task in T^s is discarded then
2:     ws := max(⌊0.6ws⌋, 1), where max returns larger value
3: end if
4: if a task in T^s is completed then
5:     if ws ≥ wth then
6:         ws := ws + 1
7:     else
8:         ws := min(2ws, wth), where min returns smaller value
9:     end if
10: end if
```

B. Congestion Control for Window Size

From above example, it can be known that the capacity of $\mathcal{S}[n, t_{max}]$ has a great impact on the performance of DPS, and its value should be dynamically changed based on the completion condition of tasks in $\mathcal{S}[n, t_{max}]$. Here we use window size, represented by ws, to denote the value of this capacity. For concision, we use $\mathcal{T}^s$ to represent the set of tasks in $\mathcal{S}[n, t_{max}]$. As our objective is to complete all the tasks that have been added into $\mathcal{T}^s$, congestion control methods applied in network technology is feasible to achieve this objective. By introducing this method, DPS is extended to dynamic programming scheduling with congestion control (DPSC).

The detail of applying congestion control method is summarized in Alg. 2. The symbol wth (line 5) represents the threshold value of ws. It is set to the value of maximum $|\mathcal{T}^s|$ every $timer$ time units, where the maximum $|\mathcal{T}^s|$ is the number of tasks in $\mathcal{T}^s$ without limiting its capacity, and $timer$ is used to decide the time instant for updating wth. When a task in $\mathcal{T}^s$ is discarded, the value of ws is set to the 0.6 times its current value (the value of 0.6 is based on our experiments and experience), and due to the practical meaning of ws, it is defined as an integer with lower limit of 1 (line 2). When a task in $\mathcal{T}^s$ is completed, the increasing method of ws has two conditions: (*i*) if $ws \geq wth$, it keeps linear growth (line 6); (*ii*) if $ws < wth$, it keeps exponential growth. At this condition, wth has a upper bound (line 8). This procedure benefits from congestion control strategies (e.g., slow start, congestion avoidance) applied in network technology.

VII. Performance Evaluation

In this section, we present the numerical results of simulations, which are conducted to study the performance of different scheduling algorithms. The scheduling algorithms that are used to compare with DPS and DPSC are SRTF, EDF, LLF, and their corresponding deferrable scheduling (DS) methods, i.e., DS-SRTF, DS-EDF, DS-LLF. The DS method is introduced in [12]. The idea of DS is trying to complete more tasks by deferring the execution of their first selected tasks. For DS-SRTF, the procedure of it to construct scheduling list is: (*i*) select tasks based on the ascending order of rc_i, and (*i*) when a task is selected, allocate idle time slots to the task backwards from its deadline. The difference of scheduling procedure among DS-SRTF, DS-EDF, and DS-LLF is the order

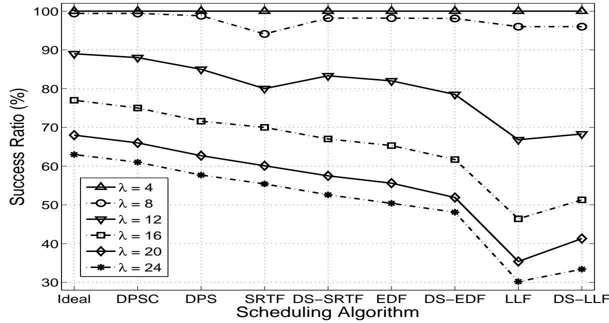

Fig. 4: Performance comparison ($4 \leq \lambda \leq 24$)

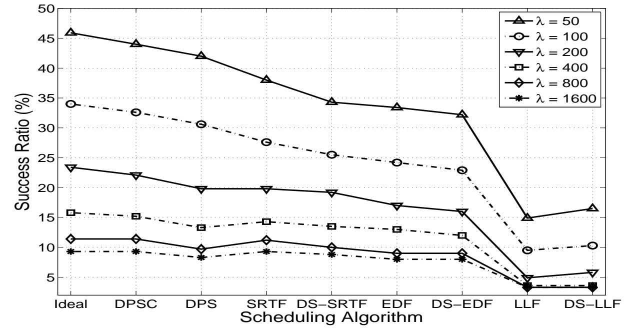

Fig. 5: Performance comparison ($50 \leq \lambda \leq 1600$)

TABLE IV: Percentage improvement of DPSC compared with other scheduling algorithms in terms of success ratio

Success Ratio	DPS (%)	SRTF (%)	DS-SRTF (%)	Ideal (%)
Average	2.3	3.0	7.2	-6.9
Maximum	16.0	17.1	25.4	0

of selecting tasks. DS-SRTF selects task with ascending order of rc_i, while DS-EDF and DS-LLF select task with ascending order of d_i and laxity respectively.

A. Simulation Settings

The metric used to evaluate the scheduling performance is *success ratio*, which is the percentage of tasks that have been successfully completed. The setting of total number of input tasks is 1000. Arriving rate λ represents the mean value of the number of tasks that arrive in the system per 100 time units. In order to evaluate the performance of scheduling algorithms under different workload conditions, the success ratio is evaluated as a function of the arriving rate λ. As the workload can be changed by λ, the attributes of tasks in our simulations are given a simple setting. For each task τ_i, c_i varies uniformly in [1 25]. The assignment of d_i is according to the equation: $d_i = r_i + sf_i \cdot c_i$, where sf_i is the slack factor that indicates the tightness of task deadline. For each task τ_i, sf_i varies uniformly in [1 16].

B. Results and Analysis

As the change rate of success ratio is quit different in different intervals of λ, the results are shown separately in two intervals of arriving rate: [4 24] and [50 1600]. When $\lambda \leq 4$, all the algorithms can make all tasks meet their deadlines. In addition to comparing with the baseline algorithms, we also want to know how far the performance of DPSC is from the upper bound in terms of success ratio. If the congestion control mechanism in DPSC could set ws perfectly to make sure that every time DPSC *just* completes all the tasks in $\mathcal{T}^s$, and no processor time slot is wasted in executing unsuccessfully completed tasks, the DPSC could achieve the ideal performance. In order to get this upper bound, we manually adjust the value of ws in DPSC. For the specific input task sets, we can get the ideal results, which are represented by Ideal in Fig. 4, Fig. 5 and table IV.

We first study the performance of the six baseline algorithms. The results are shown in Fig. 4 and Fig. 5. We can see that, DS-SRTF performs best when $\lambda \leq 12$. This means that, the idea of DS that tries to complete more tasks by deferring the execution of their first selected tasks works well under such workload condition, as under such workload condition, most of the tasks can be completed. The performance of DS also depends on the order of selecting tasks. Selects tasks with ascending order of rc_i can get satisfactory performance. When workload becomes heavier ($\lambda \geq 16$), the performance of SRTF will surpass DS-SRTF, and achieve the best performance. The reason is that, under such heavy workload, many new tasks are keeping on arriving, which makes there usually exist many tasks can be scheduled. Under such workload condition, first selecting shortest task to execute is a wise choice.

Focus next on DPS and DPSC. For DPS, compared with the baseline algorithms, it performs best when $\lambda \leq 200$. As when $\lambda > 200$, the success ratio is around 20%, which means system is severely overloaded. This condition rarely happens in practical environment. Thus, we can say DPS achieves better overall performance than all the baseline algorithms. This reveals the effectiveness of dynamic programming scheduling.

For DPSC, it achieves the best performance among all the algorithms under all the different workload conditions. Compared with DPS, this observation proves the effectiveness of congestion control mechanism. This improvement appears when $\lambda \geq 8$. This is because when $\lambda < 8$, the overload condition is not serious. The ws used to limit the capacity of $\mathcal{T}^s$, computed by congestion control mechanism, is larger than the number of tasks that can be added into $\mathcal{T}^s$. This makes these two methods have the same performance. When $\lambda \geq 800$, it can be seen that DPSC and SRTF achieve the same best performance. The reason is that, under such serious overload condition, the ws computed by function `ComputeWS()` is 1 which is its lower bound at most of the time. Under that condition, it makes the DPSC act the same as SRTF. Notice that, under such serious overload condition, as so many new tasks are keeping on arriving, first selecting a shortest task to execute can get the ideal performance in terms of success ratio.

The statistic results of DPSC compared with the baseline algorithms and the ideal performance are shown in table IV. As DS-SRTF and SRTF achieve the best performance under different workload conditions among all the baseline algorithms, only them are shown in the table to compare with DPSC. The values in the table are the percentages of improvement (positive number) or deterioration (negative number) of DPSC compared to the corresponding methods in terms of success

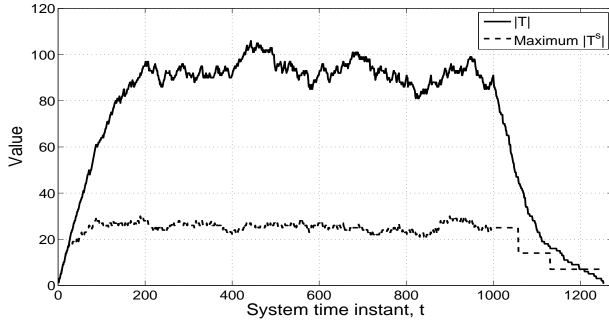

Fig. 6: Trajectories of $|\mathcal{T}|$ and maximum $|\mathcal{T}^s|$ ($\lambda = 100$)

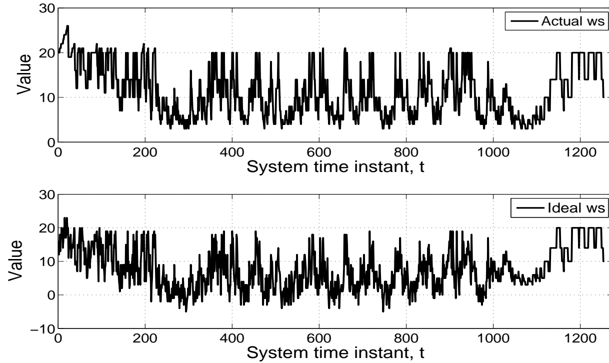

Fig. 7: Trajectories of ws under congestion control mechanism and manual operation ($\lambda = 100$)

ratio. Compared with SRTF, although the average improvement of DPSC is just 3.0% which seems not much, it is because the performance upper bound which is shown in the Ideal column. The average performance of DPSC is not far from the ideal performance (6.9% less than the ideal), which shows the effectiveness of DPSC. Moreover, relative big improvements happen at some specific workload conditions. For example, when $\lambda = 100$, DPSC can get 17.1% improvement compared with SRTF.

Fig. 6 shows the trajectories of $|\mathcal{T}|$ and maximum $|\mathcal{T}^s|$, where maximum $|\mathcal{T}^s|$ stands for the number of tasks in $\mathcal{T}^s$ without limiting its capacity. Only the results for $\lambda = 100$ is shown as an example. We can see that, although the number of tasks which are waiting to execute is large (around 90), the number of tasks that can be added into $\mathcal{T}^s$ is relatively small (around 25). Actually, even such small number of takes cannot all be completed under this kind of heavy workload.

Based on this observation, the capacity of $\mathcal{T}^s$ should be limited. The capacity value is denoted by window size ws, its trajectories are shown in Fig. 7. The actual ws stands for the actual trajectory under congestion control mechanism, and ideal ws stands for the ideal trajectory under manual manipulation. For ideal trajectory, every time, we manually manipulate ws to make sure that DPSC can *just* complete all the tasks in $\mathcal{T}^s$, and no processor time slot is wasted in executing unsuccessfully completed tasks. To achieve this, it is only possible when all tasks are known as a prior. Thus, the manual manipulation actually plays a role of clairvoyant off-line scheduling algorithm. Compare these two trajectory, we can see that the tendency is the same. This means congestion control mechanism can relatively effectively adjust ws.

VIII. Concluding Remarks

The design of scheduling algorithm is crucial for overloaded real-time systems. In this paper, we focus on maximizing the total number of tasks that meet their deadlines. To achieve this objective, a dynamic programming scheduling (DPS) algorithm was proposed. Each time DPS scheduling tasks makes optimum choice for currently known task set. But for the set of all the tasks, due to the uncertainty of new arriving tasks, DPS cannot make optimum choice. This uncertainty becomes the biggest challenge in the design of on-line scheduling algorithms. To meet this challenge, by introducing congestion control mechanism, DPSC was proposed. As shown in the the performance studies, DPSC can effectively improve system performance. This has demonstrated the effectiveness of congestion control mechanism. For the future work, an appealing direction is to adapt DPSC for such real-time systems which have probabilistic task model.

References

[1] P. Derler, E.A. Lee, and A.S. Vincentelli, "Modeling Cyber-Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012.

[2] J.C. Eidson, E.A. Lee, S. Matic, S.A. Seshia, and J. Zou, "Distributed Real-Time Software for Cyber-Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 45–59, 2012.

[3] M.K. Gardner and J.W.S. Liu, "Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload," In *Proc. of the Euromicro Conference on Real-Time Systems*, pp. 287–296, 1999.

[4] S.K. Baruah, J. Haritsa, and N. Sharma, "On-line Scheduling to Maximize Task Completions," In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 228–236, 1994.

[5] A. Burns, "Scheduling Hard Real-Time Systems: a Review," *Software Engineering Journal*, vol. 6, no. 3, pp. 116–128, 1991.

[6] S.K. Baruah and J.R. Haritsa, "Scheduling for Overload in Real-Time Systems," *IEEE Transactions on Computers*, vol. 46, no. 9, pp. 1034–1039, 1997.

[7] P. Mejia-Alvarez, R. Melhem, D. Mosse, and H. Aydin "An Incremental Server for Scheduling Overloaded Real-Time Systems," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1347–1361, 2003.

[8] J.R. Haritsa, M.J. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints," In *Proc. of the ACM Principles of Database Systems Symposium*, pp. 331–343, 1990.

[9] F. Zhang and A. Burns, "Schedulability Analysis for Real-Time Systems with EDF Scheduling," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1250–1258, 2009.

[10] R.I. Davis and A. Burns, "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems," *ACM Computing Surveys*, vol. 43, no. 5, pp. 35:1–35:44, 2011.

[11] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 40–61, 1973.

[12] M. Xiong, S. Han, K.-Y. Lam, and D. Chen, "Deferrable Scheduling for Maintaining Real-Time Data Freshness: Algorithms, Analysis, and Results," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 952–964, 2008.

[13] C.D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," Ph. D. Dissertation, Carnegie Mellon University, 1986.

[14] S. Hwang, C.M. Chen, and A.K. Agrawala, "Scheduling an Overloaded Real-Time System," In *Proc. of the IEEE International Phoenix Conference on Computers and Communications*, pp. 22–28, 1996.

[15] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control," In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 286–295, 1998.