An Adaptive Server-Based Scheduling Framework with Capacity Reclaiming and Borrowing

Meng Liu[1], Moris Behnam[1], Shinpei Kato[2], Thomas Nolte[1]
[1]Mälardalen University, Västerås, Sweden
[2]Nagoya University, Nagoya, Japan
Email: [1]{meng.liu, moris.behnam, thomas.nolte}@mdh.se
[2]shinpei@is.nagoya-u.ac.jp

***Abstract*—In this paper, we present a new reservation based scheduling framework for soft real-time systems using EDF algorithm (called CARB-EDF). This framework has the features of Capacity Adaptation, Reclaiming and Borrowing. This framework can simplify the initial configuration of the system, where the system designer does not need to provide any estimations of task execution times. We also present a Chebyshev's inequality based predictor to estimate task execution times. A number of simulation-based experiments have been implemented. According to the results compared with some related works, our scheduling framework can provide a better performance with acceptable extra scheduling overhead.**

I. Introduction

A. Motivation

In modern real-time systems, the complexity of applications has become higher and higher. In many applications, the execution time of a task varies a lot during runtime. For example, a video decoder task can have large diversity in its execution times depending on processed content [1]. For these tasks, the Worst-Case Execution Times (WCET) are very difficult to be predicted. The WCET predictions may also become very pessimistic in reality. As a result, designing the system based on these estimations may result in a large amount of waste of system resources, especially for soft real-time systems where some deadline misses are acceptable. On another hand, some estimations can be violated during runtime, which is known as the overrun problem. This problem has been addressed in many existing works (e.g. [2] [3]). Therefore, many traditional scheduling framework based on WCETs have become less and less applicable.

In order to efficiently use the system capacity and handle the overrun problems, many works have been proposed, such as CBS [4], BEBS [5], GRUB [6], HisReWri [7] and BACKSLASH [8]. These works solve the above problem in different ways such as employing capacity reclaiming or capacity stealing mechanisms. However, these solutions still require the system designer to provide some preliminary estimations of task executions, in order to define a proper reserved server capacities. If the server capacity is selected incorrectly, the system performance can be obviously degraded, since the servers cannot adapt themselves according to runtime changes.

On another hand, feedback-controlled scheduling (e.g. [9] [10] [11]) can be used to solve the above problem, where servers can adapt their capacities according to the runtime workload. These frameworks can provide good performance in general. However, when the system experiences some sudden runtime changes (e.g. the occurrence of an extreme overrun), it may take some time for the scheduler to adapt this change. In other words, a job with overrun has a high chance to miss its deadline due to the current insufficient reserved capacity. For these cases, the capacity reclaiming and borrowing mechanisms can provide help, where this job can use reclaimed or borrowed capacity to handle the overrun.

Therefore, in this paper, we present a new scheduling framework for soft real-time system, which is based on the Earliest Deadline First (EDF) [12] scheduling algorithm with the features of capacity adaptation, reclaiming and borrowing. Under our framework, the system designers do not need to provide any estimations of task execution times, since the server capacities can adapt themselves during runtime.

B. Contribution

- We present a new reservation-based scheduling framework for soft real-time systems, which can support capacity adaption, reclaiming and borrowing. Due to the capacity adaptation mechanism, this framework can obviously decrease the requirements to the system designers.
- In order to achieve capacity adaptations, we present a Chebyshev's inequality based estimator to predict the future execution times of tasks. The parameters of this predictor are selected based on given probabilistic requirements. As far as we know, tuning these parameters in our framework is easier than in the related works while considering unknown models (i.e. distributions) of task executions.
- Introducing the concept of task criticality levels (which has not been used in the above capacity reclaiming and borrowing based approaches) can provide more controls to the system designers on server reservations.
- A number of simulation-based experiments have been implemented including both hypothetical experiments and

This work is supported by the Swedish Research Council, via the research project START.

a case study using video decoder tasks. From the experimental results, we can observe that: (1) our execution time predictor can provide a good estimation (i.e. can always satisfy the given expectations); (2) our scheduling framework can provide better performance than the related works; (3) the extra scheduling overhead of our framework is acceptable in reality.

C. Related Work

Many works have been proposed regarding dynamic reservation-based scheduling of real-time tasks. Here we discuss some of the most relevant related works. In [13] and [14], the authors present slack stealing algorithms to schedule aperiodic tasks by safely (i.e. without deadline missing) postpone the deadlines of jobs with higher priorities. However, these solutions require the prior knowledge of task execution times which are not available for many real applications. In [4], the authors introduced the Constant Bandwidth Server (CBS) under EDF scheduling algorithm, which can isolate task executions and bound the effect of task overruns by constraining runtime server bandwidth. Under CBS, tasks can borrow capacity from future server instances by extending its deadline as a price. IRIS [15], which is extended from CBS, provides a minimum budget in a fixed time interval and fairly distribute the remaining capacity to needed servers. Similarly, in CASH [16], the reclaimed capacities are organized in a global queue. Each server consumes the reclaimed capacities whose deadlines are earlier than or equal to its own, before using its own budget. Unfortunately, CASH cannot provide a good performance due to the blocking of unconsumed previous slacks. In [17], the authors present an improved version of CASH which can also support resource sharing. Some other extensions of CBS have also been proposed (e.g. BEBS [5], GRUB [6], CSS [18]). In this paper, the basic scheduling rules are extended from BACKSLASH [8], which is improved from CASH and employs both capacity reclaiming and borrowing mechanisms. BACKSLASH also uses an EDF version of HisReWri [7], which retroactively allocate reclaimed capacities to the jobs who has already borrowed capacities from future jobs. However, BACKSLASH may face to a potential problem that when a job borrows a large amount of capacity from its future jobs (e.g. when a large overrun occurs), it may cause many following jobs to miss their deadlines as a domino-effect. This problem may happen when the server capacities are not well defined. In our framework, we integrate a capacity adaptation mechanism into BACKSLASH, which can provide self-adaptations of server capacities during runtime. As a result, the scheduler can decrease the effects of improperly selected server capacities, and it can obviously decrease the requirements to system designers.

Feedback-control mechanisms have already been utilized in some real-time applications (e.g. [9] [10]). Most of these applications target on soft real-time requirements, since it is difficult to provide hard timing guarantees using runtime feedback-controls, especially when the controls are achieved by runtime statistical predictions. FC-EDF [19] uses Proportional Integral Derivative (PID) controllers, where the deadline miss ratios of admitted tasks are selected as the controlled variable. In [11], the authors present an adaptive hierarchical scheduling framework using periodic servers, where the usage state of assigned budgets are considered as the controlled variable. The workload predictor used in this framework is based on the AutoRegressive (AR) model. In [20], the author present the framework AQuoSA, which is a combination of feedback controls and CBS. Inspired by [20] and [11], we integrate a feedback-controlled capacity adaptation mechanism into BACKSLASH, where we use a different runtime statistical predictor based on Chebyshev's inequality to estimate task execution times.

The rest of this paper is organized as follows. In Section II, we briefly introduce the background knowledge of the statistical tool which is used in our execution time predictor. In Section III, we present the details of our scheduling framework. The results of the simulation-based evaluations are presented in Section IV. Finally, Section V concludes this paper and brings some ideas of future works.

II. Background Knowledge

A. Statistical Tool

Precise probabilistic analyses always incur a large amount of calculation overhead, which may not be affordable for online scheduling frameworks. Instead, we employ some simple statistical tools to analyze the runtime information, in order to perform a feedback-controlled scheduling with acceptable overhead.

a) Mean and Standard Deviation: Mean is a simple but widely used way to present the center of a sample set (i.e. the average value), which can be calculated as:

$$\bar{x} = \frac{1}{n}\sum x_i,\ \forall x_i \in X \tag{1}$$

where $\bar{x}$ represents the mean value of the data set X consisting of n samples.

Standard deviation is used to present the spread of a sample set regarding its mean, which can be computed by:

$$\sigma = \sqrt{\frac{1}{n-1}\sum (x_i - \bar{x})^2},\ \forall x_i \in X \tag{2}$$

where σ denotes the standard deviation of the sample set X with n samples, and the mean of this data set is $\bar{x}$. A wider spread of samples may result in a larger standard deviation [21].

b) Chebyshev's Inequality: Chebyshev's inequality [22] is a famous probability theory which has been utilized in many different disciplines. Chebyshev's inequality shows that in any probability distribution, the percentage of samples, that are more than k times standard deviations away from the mean, will not exceed $1/k^2$. This inequality can be presented as:

$$Pr(|x_i - \bar{x}| \geq k\sigma) \leq \frac{1}{k^2},\ \forall x_i \in X \tag{3}$$

k	Max % beyond $k\sigma$ from $\bar{x}$	k	Max % beyond $k\sigma$ from $\bar{x}$
1	100%	2.58	15%
$\sqrt{2}$	50%	3.16	10%
2	25%	5	4%
2.24	20%	10	1%

TABLE I
EXAMPLE VALUES OF CHEBSHEV'S INEQUALITY

where $\bar{x}$ and σ are the mean and the standard deviation of the sample set X, and $Pr(A)$ represents the probability of an event A.

Generally, the inequality gives a poorer (i.e. pessimistic but safe) bound comparing to the estimate where more information of the involved distribution is provided. However, in our work, the collected samples refer to the execution times of a task, where the exact probability distribution may be very difficult to be captured. Therefore, in this paper, we use Chebyshev's inequality which can be applied on arbitrary distributions. Table I shows several example values of Chebyshev's inequality.

III. SCHEDULING FRAMEWORK

A. Task Model

In this paper, a system consists of a set of periodic and sporadic soft real-time tasks. A soft real-time task τ_i is characterized as $(T_i,\ D_i,\ cl_i)$, where T_i denotes its minimum inter-arrival time between successive jobs and D_i represents the relative deadline. We use $J_{i,p}$ to denote the pth job of task τ_i. If $J_{i,p}$ arrives at time $a_{i,p}$, the initial absolute deadline of $J_{i,p}$ (denoted as $d_{i,p}$) is $a_{i,p}+D_i$. Each task is assigned a *criticality level*, where a more time-critical task has a higher criticality level. When the system capacity is insufficient, the scheduler will provide more guarantees to the tasks with higher criticality levels. Moreover, we do not require any estimation of the Worst Case Execution Time (WCET) for each task.

As a reservation-based scheduling framework, we use rate-based servers (i.e. same as BACKSLASH [8]) in this paper, which are similar to the concept of other bandwidth servers such as CBS [4] and BEBS [5]. Each task is assigned a dedicated server. The server of task τ_i can be characterized by (Ts_i, Cs_i), where Ts_i denotes the replenishment period and Cs_i represents the server capacity/budget within one replenishment period. Theoretically, each server instance is designed for one job of the corresponding task. Therefore, Ts_i is always defined to be equal to the period of the assigned task τ_i. The server capacity Cs_i is calculated during runtime based on statistical information (more details in III-D).

Each server instance also has an absolute deadline (denoted as $ds_{i,p} = as_{i,p} + Ts_i$), where $as_{i,p}$ denotes the release time of the server instance for job $J_{i,p}$. When a job $J_{i,p}$ is released, it will activate the corresponding server instance (i.e. $as_{i,p} = a_{i,p}$). The capacity of a server instance decreases when it is serving job executions. Once the capacity of a server instance for τ_i becomes 0: (1) if there is workload pending for this server (e.g. the execution of the current job instance exceeds the reserved capacity), it can immediately recharge its capacity by Cs_i and the deadline is extended by Ts_i; (2) if there is no workload pending, it will be recharged until the release of the next job (i.e. $J_{i,p+1}$). If a server instance reaches its absolute deadline with remaining budget: (1) if there is still workload pending for this server, it can recharge its capacity up to Cs_i immediately, and the deadline is extended by Ts_i; (2) if there is no workload pending, it will be recharged until the arrival of the next job.

B. Control Model

The basic idea of the control scheme used in this paper is described in Figure 1. The system monitors the server capacity error (meaning the difference between the reserved capacity Cs_i and the prediction[1] e_i^l based on previous executions) after the completion of each job (or a number of jobs). If the error is greater than 0, which means that the current server capacity may not be enough for the executions of the coming jobs, then the capacity adaptation process is performed and a new Cs_i is assigned according to the error.

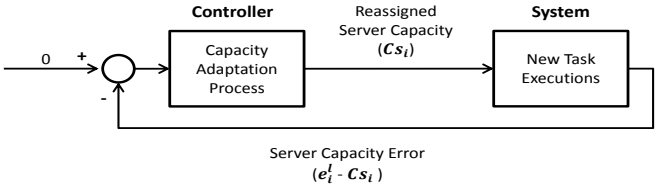

Fig. 1. The Block Diagram of the Control Scheme

The system may converge into two situations: (1) When the total task utilization is lower than 1 (i.e. no overload): Assume that no task contains deadlock, then the execution times of each task are supposed to be following a certain bounded distribution. Therefore, after a number of executions, the execution time prediction will become stable (i.e. the execution times of finished jobs are almost representative of the whole distribution). Then the system will converge with a situation that all the tasks get requested capacities regarding the runtime predictions. (2) When the total task utilization is greater than 1 (i.e. overload occurs): According to our adaptation mechanism, the tasks with higher criticality levels are provided with more guarantees. Therefore, in this case, the system may converge with a situation that the tasks with higher criticality levels are assigned enough capacities regarding the predictions, while the less critical tasks are assigned less capacity than their requests. The overloaded situation is an improper system design, which should be avoided in reality.

C. Assumptions

- The framework targets on single-core systems.
- The systems considered in this paper only contain soft real-time tasks (or non-real-time tasks but no hard real-time tasks).
- Tasks are independent from each other (i.e. they do not share any resources), and do not contain any deadlock.
- Constraint deadlines are assumed for all the tasks (i.e. $D_i \leq T_i$).

[1]The calculation of e_i^l and e_i^h are presented in Section III-D.

D. Scheduling Rules

Our scheduling framework can be described in two phases: an initial phase and an runtime phase.

1) Initial Phase: First of all, the scheduler reserves some capacity Ur (i.e. presented by utilization) for future capacity adaptations, which is used to decease the calculation overhead of the scheduler (see Section III-D2a). Then the remaining processor capacity (i.e. $1 - Ur$) is evenly distributed to all the tasks. The server capacity within each replenishment period can be computed as

$$Cs_i = Us_i \times Ts_i, \tag{4}$$

where Us_i denotes the assigned server bandwidth to τ_i. The server capacity can be changed during runtime according to the runtime workload.

2) Runtime Phase: In this subsection, we present the runtime scheduling mechanism of the framework. First, we describe our capacity adaptation mechanism using a Chebyshev's inequality based estimator, which is the key issue of this framework. Then we present the scheduling rules of the whole framework which involves capacity adaptation, reclaiming and borrowing mechanisms.

a) Capacity reallocation/adaptation: The scheduler maintains two variables for each task τ_i: a variable e_i^l, which represents a lower probabilistic upper bound of the execution time of τ_i computed based on runtime statistics; and a variable e_i^h, which represents a higher probabilistic upper bound of the execution time of τ_i. The calculation of e_i^l and e_i^h will presented in next subsection.

The main principles of the capacity adaptation scheme in our framework are that:

(1). For any task τ_i, the scheduler provides at least a guarantee to afford the execution of e_i^l (i.e. $Cs_i \geq e_i^l$). If the total capacity is not enough, the tasks with higher criticality levels get more guaranteed service.

(2). If all the tasks get enough capacity for their basic request (i.e. e_i^l for τ_i), the scheduler provides more capacity for each task regarding its higher request (i.e. e_i^h for τ_i). If the total capacity is not enough, only the tasks with higher criticality levels need to be served.

When the server capacity of a task τ_i can afford its basic request e_i^l, the scheduler will not make any change since the basic principle is fulfilled. The scheduler will reallocate server capacities only when the e_i^l of a task exceeds its current assigned server capacity Cs_i (i.e. $e_i^l > Cs_i$).

The reallocation is processed as follows (Algorithm 1):

- Step 1 (line 11 - 13): First, we need to calculate how much more capacity that τ_i needs in order to guarantee the execution of the latest estimated e_i^l (line 11 & 12). Moreover, the scheduler needs to calculate the free system capacity Ur which can be used for τ_i (line 13). In the function $Recalculate\_Ur()$ (line 1 - 9), the scheduler checks the extra capacity of each task server which may cost a waste regarding the current higher request (i.e. e_p^h), and then add them to Ur (line 4 - 6).
- Step 2.1 (line 14 - 22): If the free processor capacity remained in the system is sufficient to afford the newly estimated e_i^l, we directly reassign the server capacity Cs_i to be equal to e_i^l (line 14 & 15). Then the corresponding capacity, which has just been assigned to τ_i, should be deducted from Ur (line 16).
  After the above reassignment process, if there is still some free capacity remaining, the scheduler will try to provide more capacity to τ_i (line 17 - 21). In this case, τ_i can either get the capacity of e_i^h (i.e. the higher requested capacity based on the current statistics) for each period, or just consumes all the remaining Ur.
  Finally, this reallocation process can be terminated (line 22). The capacities of other task servers will not be affected, since the scheduler just uses the free capacity to handle the reassignment of Cs_i. Therefore, sufficient free capacity Ur can obviously decrease the calculation overhead of the scheduler. Note that, if the original Ur is very large, the initial capacity of each task will become much smaller. As a result, more adaptations may be required, but the overhead of each adaptation is low. On the other hand, if Ur is very small, the initial capacity of each task will be much larger. In this case, the scheduler may need to perform fewer adaptations, however, the overhead of each adaptation may become large.
- Step 2.2 (line 24 - 37): If the free processor capacity is not enough for the reassignment of Cs_i, the scheduler will assess capacities from tasks with lower criticality levels in order to provide enough service to τ_i.
  First, the scheduler assigns all the remaining Ur to τ_i, so that there will be no waste of system capacity and the effects on other task servers can be decreased (line 24 & 25).
  Then the scheduler starts to steal the capacity from the task with the lowest criticality level (line 26 - 36). In order to avoid the problem that the reserved capacity of a task may become 0 due to no available statistical information, the capacities of the tasks with no previous executions cannot be stolen. For each less critical task τ_j (i.e. $cl_j \leq cl_i$), the scheduler steals at most a capacity of $Cs_j - e_j^l$. This is to make sure that each task has at least a guaranteed capacity of e_j^l. The scheduler assesses the less critical tasks one by one following an ascent order of the criticality levels of these tasks (line 26). This loop terminates when: (1) τ_i gets enough capacity to afford e_i^l (line 30), in which case the reallocation process terminates successfully; (2) there is no task to further steal capacity (line 37), in which case the reallocation process also terminates but τ_i cannot get enough guarantee of e_i^l. If case (2) occurs, we keep the system running, since: the following executions of τ_i may get help from reclaimed capacities; and the capacity can also be adjusted after later executions. An alternative solution is that we can allow τ_i to completely steal capacities from tasks with lower criticality levels, until τ_i gets enough capacity. In

this case, the server capacities of the tasks with lower criticality levels may become zero. Then these tasks need to wait for later executions to recover their capacities.

Algorithm 1 Capacity Reallocation

```
1: Function Recalculate_Ur():
2: {
3: for all task τ_p with at least one previous execution do
4:     if Cs_p > e_p^h then
5:         Ur += (Cs_p − e_p^h) / T_p
6:         Cs_p = e_p^h
7:     end if
8: end for
9: }
10: when any task τ_i, e_i^l > Cs_i:
11: Δe_i = e_i^l − Cs_i
12: ΔU_i = Δe / T_i
13: Recalculate_Ur()
14: if Ur ≥ ΔU_i then
15:     Cs_i = e_i^l
16:     Ur −= ΔU_i
17:     if Ur > 0 then
18:         ΔU_i' = (e_i^h − Cs_i) / T_i
19:         Cs_i += min(ΔU_i' × T_i, Ur × T_i)
20:         Ur = max(0, Ur − ΔU_i')
21:     end if
22:     return Reallocation_Succcess
23: else
24:     ΔU_i −= Ur
25:     Ur = 0
26:     for all task τ_j with at least one previous execution,
        where (τ_j ≠ τ_i) ∧ (cl_j ≤ cl_i) ∧ (Cs_j > e_j^l),
        regarding an ascent order of criticality levels do
27:         if (Cs_j − e_j^l) / T_j ≥ ΔU_i then
28:             Cs_i = e_i^l
29:             Cs_j = Cs_j − ΔU_i × T_j
30:             return Reallocation_Succcess
31:         else
32:             ΔU_i −= (Cs_j − e_j^l) / T_j
33:             Cs_i += ((Cs_j − e_j^l) / T_j) × T_i
34:             Cs_j = e_j^l
35:         end if
36:     end for
37:     return Insufficient_Capacity
38: end if
```

b) Runtime parameter maintenance: As mentioned above, the scheduler needs to maintain two parameters for each task (i.e. e_i^l and e_i^h) during runtime. These parameters are computed based on the runtime statistical collections. First, we introduce how to compute these two parameters. Basically, e_i^l and e_i^h are two probabilistic estimates with different levels of expectations (i.e. Pr_i^L and Pr_i^H), which are presented in Figure 2. e_i^l is combined with a given probability Pr_i^L according to the user requirements. The probability of a job execution time of τ_i which is larger than e_i^l, should not be greater than Pr_i^L. e_i^h is a higher estimate than e_i^l, which is combined with a lower probability Pr_i^H. Similarly, the probability of a job execution time of τ_i which exceeds e_i^h, should not be greater than Pr_i^H. The computation can also be presented as follows:

$$\begin{aligned} Pr(x > e_i^l) &\leq Pr_i^L \\ Pr(x > e_i^h) &\leq Pr_i^H \end{aligned} \tag{5}$$

where x denotes the execution time of a job of τ_i, and $e_i^l < e_i^h$, $Pr_i^L > Pr_i^H$.

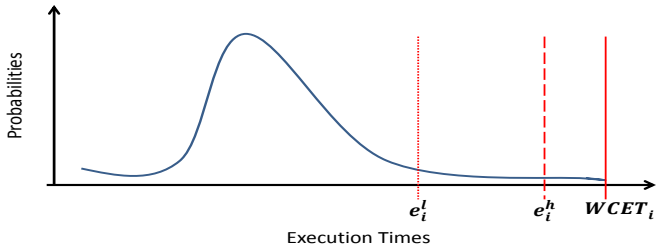

Fig. 2. e_i^l VS. e_i^h. The curve represents the execution time distribution of an example task. The x axis denotes the execution times, and the y axis denotes the corresponding probabilities.

The values of Pr_i^L and Pr_i^H depend on the requirements from the system designers. Apparently, a higher value of Pr_i^L or Pr_i^H may result in a lower value of e_i^l or e_i^h. Given specified Pr_i^L and Pr_i^H, we can approximately compute the values of k which is used in Chebyshev's inequality. In Chebyshev's inequality, both the upper and lower tails of the involved distribution are considered (i.e. the exceedances denote both the samples higher than $\overline{x} + k\sigma$ and the samples lower than $\overline{x} - k\sigma$). However, in our case, we are only interested in the exceedances that are greater than the larger estimation, which refers to the upper tail. Therefore, taking this into account, we can approximately compute k_i^H as $\sqrt{1/2Pr_i^H}$ (i.e. same manner for k_i^L). For example, in our experiments (Section IV), we set Pr_i^L to be 0.1, and Pr_i^H to be 0.04. Then the corresponding values of k_i^L and k_i^H for the Pr_r^L and Pr_i^H are 2.24 and 3.53 respectively.

After each execution of task τ_i, the corresponding execution time of this job will be recorded by the scheduler. This execution time is added into the sample set X_i which consists of all the collected execution times of τ_i. Based on Chebyshev's inequality, we can approximately compute the current estimated e_i^l and e_i^h by:

$$\begin{aligned} e_i^l &= \overline{x_i} + k_i^L \cdot \sigma_i \\ e_i^h &= \overline{x_i} + k_i^H \cdot \sigma_i \end{aligned} \tag{6}$$

where $\overline{x_i}$ and σ_i can be computed using Eq. 1 & 2.

The above calculation can be performed after the completion of each job. In this case, the scheduler can always keep the latest information of task executions, and estimations can be more precise. However, this solution may increase the overhead of the scheduling framework. On another hand, the scheduler can perform the calculation after every Ns ($Ns \geq 1$) executions. Under this solution, the calculation can be performed less frequently which can decrease the scheduling overhead. However, the scheduler needs more memory to store the latest Ns samples for each task, and the estimations may become less precise than the former solution. Therefore, the system designer can play with the value of Ns, in order to balance the trade-off between overhead and memory cost.

c) Scheduling Rules: The basic scheduling rules of our framework are extended from BACKSLASH [8] which is an EDF based scheduling algorithm. This algorithm enables both capacity reclaiming and borrowing. The rules are presented below.

Rule 1: All the tasks are scheduled using the EDF algorithm.

Rule 2: If the execution of a job does not consume the whole reserved server capacity within the corresponding replenishment period, the remaining capacity is then reclaimed by the scheduler. This reclaimed capacity (also called slack hereinafter) can be used for any other tasks in the system. However, the reclaimed slack will be expired when it reaches its deadline which is inherited from it original server. Assume that the actual execution time of job $J_{j,q}$ is $e_{j,q}$ (where $e_{j,q} < Cs_j$, the whole capacity Cs_j is available for $J_{j,q}$, and $J_{j,q}$ only uses its own reserved capacity), the reclaimed capacity can then be computed as

$$rc_{j,q} = MIN((ds_{j,q} - f_{j,q}), Cs_j - e_{j,q}) \quad (7)$$

where $f_{j,q}$ denotes the finishing time of $J_{j,p}$.

Rule 3: If there are several slacks (i.e. reclaimed by different jobs) remained in the system, the scheduler dispatches them using EDF algorithm. Moreover, a slack is always allocated to the waiting task with the earliest original deadline.

Rule 4: For a released job who has the earliest deadline among all the waiting jobs, if there are some reclaimed capacities whose deadlines are earlier than or equal to the deadline of this job, it will first use those reclaimed capacities and temporarily inherit the deadlines of those reclaimed capacities. Once a reclaimed capacity reaches its deadline or it has been completely consumed, the scheduler will remove it from the system. On the other hand, if the deadline of the executing job is earlier than the reclaimed capacities, this job will first uses its own capacity.

Rule 5: When there is no reclaimed capacity remained in the system, the running job starts to use its own reserved capacity.

Rule 6: After the running job exhausts all the reclaimed capacities as well as its own reserved capacity, if this job still does not finish its execution, it will borrow capacities from the future jobs of the same task. When the running job $J_{i,p}$ is using the borrowed capacity from its future job $J_{i,p+1}$, its deadline is temporally extended to $as_{i,p} + 2 \cdot Ts_i$. This temporary deadline extension of the running job is used to guarantee that jobs from other tasks will not experience extra interference due to the capacity borrowing mechanism. Of course, this solution may punish the future jobs of τ_i by introducing unpredictable internal interference. However, this problem can be compensated by the capacity reclaiming and capacity adaptation mechanisms.

Rule 7: As presented in [8] and [7], a job which has consumed borrowed capacity from its future job, should also be able to receive future reclaimed slacks. In other words, if a job borrowed capacities from its future jobs to complete its execution, it remains in the scheduler (i.e. before it reaches its original deadline) and is still available to compete for reclaimed capacities with its original deadline. When this job gets a reclaimed slack, the scheduler will rewrite the history to pay the reclaimed capacity back to the corresponding future job whose capacity was borrowed.

Considering the integration of server capacity adaptations, we add the following rules:

Rule 8: During a runtime adaptation phase, if a task has not been executed yet (i.e. there is no previous executions of this task), the scheduler keeps its initially assigned capacity (i.e. cannot be stolen by tasks with higher criticality levels). This is to avoid the case that a job may experience no reserved capacity when it arrives.

Rule 9: The reallocation of server capacities do not affect previous jobs, since the predictions are aiming for future jobs. In other words, a reclaimed capacity can only be decreased due to the consumption of other jobs or reaching its deadline. This rule can also decrease the complexity of the scheduler by reducing the extra calculations for reclaimed slacks.

Rule 10: Assume that the previous job $J_{i,p}$ of a task τ_i has borrowed capacity (with the size of Cb) from the next coming job $J_{i,p+1}$. After the capacity adaptation process, if the newly assigned server capacity Cs_i decreases which becomes less than the originally borrowed capacity (i.e. $Cs_i < Cb$), then the scheduler considers that the capacity of $J_{i,p+1}$ is completely borrowed and the rest of the borrowed capacity (i.e $Cb - Cs_i$) is borrowed from the following jobs (i.e. $J_{i,p+2}$, $J_{i,p+3}$, ...). On the other hand, assume that the previous job $J_{i,p}$ borrowed capacities from several coming jobs, if the newly assigned capacity is increased a lot which can afford all these borrowed capacities (i.e. $Cs_i > Cb$), then the scheduler considers all these capacities are borrowed from the next coming job (i.e. $J_{i,p+1}$).

IV. Evaluation

We have implemented a number of simulation-based evaluations of our scheduling framework. First, we examine the performance of the execution time predictor used in CARB-EDF, and compare the results with the estimations generated by the predictor used in [11]. Then we evaluate the performance of the whole scheduling framework by comparing with BACKSLASH. The evaluation includes both hypothetical experiments and a case study of video decoder tasks. Finally, we examine the extra scheduling overhead of our framework due to the capacity adaptation mechanism.

A. Evaluation of the Estimator

The prediction mechanism is an important phase in feedback-controlled scheduling frameworks. In this section, we evaluate the estimator of our scheduling framework, which is used to predict the execution times of future jobs based on previous executions.

In [11], the authors used an AutoRegressive (AR) model based estimator to predict future workload. The AR model has been used to describe different time-varying processes in many areas such as signal processing, economics, etc.. We also try to use the estimator presented in [11] to predict the future execution times, and compare the estimations between this estimator and our predictor. Note that in the following experiments, the parameters of this estimator are set the same as [11]. Theoretically, tuning those parameters may provide

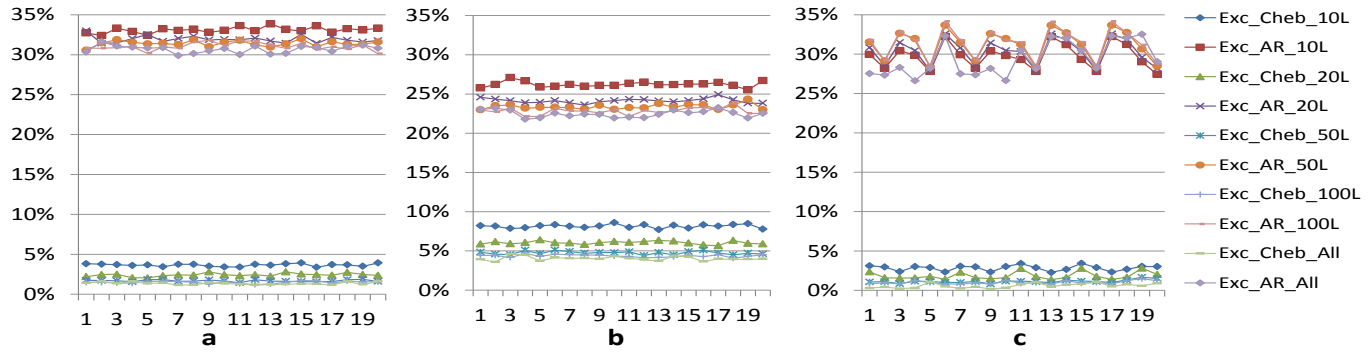

Fig. 3. The exceedance probability of the estimations regarding different sample sizes. The x axis denotes the index of experiment sets, where each set contains 5000 task executions. The y axis denotes the exceedance probability. '*Exc_Cheb*' represents the estimations predicted by our estimator, and '*Exc_AR*' represents the estimations predicted by the predictor presented in [11]. 'N L' denotes that the estimations are based on the latest N samples, and '*ALL*' denotes that the estimations are based on all the previous samples.

better estimations. However, the investigation of how to tune those parameters is out of the scope of this paper.

In these experiments, we set the k value of our predictor to be 2.24, which means that the exceedance probability is supposed to be not greater than 10%. Here an exceedance denotes that the execution time of the coming job is greater than the current prediction.

In order to perform a precise statistical estimation, a large amount of samples are necessary, because too few samples cannot provide representative predictions. However, as an on-line scheduling framework, the scheduling overhead (e.g. calculation time, memory cost, etc.) needs to be controlled, otherwise the performance of the system will be degraded. In other words, we need to decrease the required sample size while guaranteeing acceptable predictions. Therefore, we need to evaluate the effect of sample size.

Figure 3 shows[2] the exceedance probability of the estimations regarding different sample sizes. For Figure 3-a&b, in each experiments set the tasks have different parameters. For Figure 3-c, we use the same task for each experiment set, but start sampling with different phases. As we know, the distribution of execution times varies a lot from task to task. In the first set of experiments, we assume that the execution times follow a known distribution. In Figure 3-a, the executions are assumed to follow a normal distribution. The execution times are randomly generated based on a fixed mean and varied standard deviations, where the standard deviations change from 10% to 100% of the mean. As shown in the results, the exceedance probability of the predictions provided by the estimator from [11] is around 30%. However, the exceedance probability of the estimations from our predictor is lower than 5% which can always meet the requirement (i.e. 10%). On another hand, the same as we mentioned above, the exceedance probability of the estimations decreases as the sample size goes up. From these results, we can observe that when the estimations are predicted based on the latest 50 samples, the exceedance probability is very close to the predictions using all the previous executions. We can get a similar observation from Figure 3-b, where the executions are randomly generated following exponential distributions.

[2]In the following figures, the connection lines are just used for the convenience of visualization.

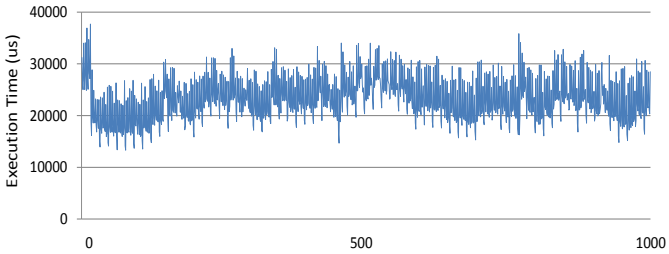

Fig. 4. The execution times of a video decoder task. The x axis denotes the index of job executions, and the y axis denotes the execution time in μs.

Besides the above hypothetical distributions, we also evaluate our predictor with a real application. The utilized application is a video decoder task which is also used in [1][23]. Figure 4 shows some example execution times of the video decoder task. Assume that the task decodes 25 frames per second, then the period of the task is 40 *ms*. The results are shown in Figure 3-c. Similar to the above experiments, the predictions from our estimator can always meet the requirement (i.e. the exceedance probability is always lower than 10%). Moreover, in this set of experiments, the predictions using the latest 50 samples are also quite acceptable, since the exceedance probabilities are very close to those of the estimations using all the samples.

In order to further evaluate the performance of the predictor, we also measured the gaps between predictions and the actual execution times. A gap denotes the opposite of an exceedance. In other words, we measure how much a prediction can be higher than the actual execution time. Apparently, a larger gap may result in a greater waste of resource. In the first set of experiments, we randomly generate task execution times based on normal distributions. As shown in Figure 5-a, while focusing on the gaps of the results, the performance of using the latest 50 samples is slightly better than using all the previous samples. We also perform a number of evaluations using the video decoder task, and the results are presented in Figure 5-b. In these experiments, we can obviously observe that using the latest 50 samples can provide tighter predictions (i.e. with less gaps). The main reason is that when we do

the predictions using all the previous executions, the means and standard deviations are very stable due to the large amount of samples. These values may also be strongly affected by extreme samples (i.e. the samples far from the mean). Therefore, these predictions are not sensitive to the latest variations of executions. On the other hand, using the latest 50 samples to do predictions will experience less effect from extreme samples since these samples will be later replaced by new executions.

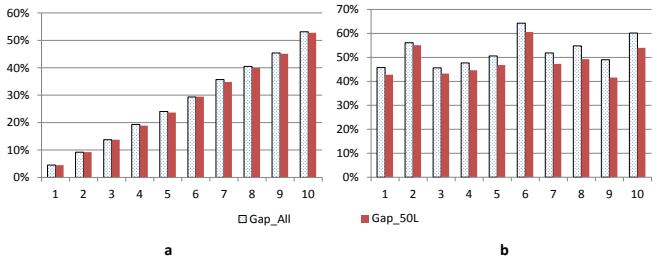

Fig. 5. The gaps between actual execution times and predictions. The *x* axis denotes the index of experiment sets, where each set contains 5000 task executions. The *y* axis denotes the average percentage of gaps regarding the actual execution times. '*50 L*' denotes that the estimations are based on the latest 50 samples, and '*ALL*' denotes that the estimations are based on all the previous samples.

According to the above experiments, we can observe that the predictions from our estimator can always meet the probabilistic requirements. Moreover, considering the overhead of the scheduling framework, we may not be able to use all the previous executions to do estimations. Instead, we can perform predictions using the latest 50 (or even 20) samples, which can also provide acceptable estimations.

B. Evaluation of the Scheduling Framework

In this subsection, we present the performance evaluation of the whole scheduling framework.

1) Hypothetical Experiments: First, we compare our framework with BACKLASH regarding task Deadline Miss Ratios (DMR). Here DMR denotes the percentage of jobs, which miss their deadlines, among all the jobs in the collected sample set. In order to perform a fair comparison, in each experiment set, the task set used in our framework is exactly the same as the set used in BACKSLASH.

In these experiments, we only consider hypothetical periodic tasks. The execution times of all the tasks are randomly generated from normal distributions. The periods of tasks are randomly selected from range [100, 1000]. For the execution time distribution of each task, the standard deviation is randomly selected from 5% to 30% of the mean. For each experiment, the average[3]total task utilization of the system is controlled within [0.8, 1.5]. The execution time predictions of our framework are based on the latest 20 executions.

For our framework, no exact task information is required (e.g. WCET). However, for BACKSLASH, we have to provide some estimation of the execution time of each task, so that we can define the server capacities. Since the task execution times are randomly generated, it is difficult to get the exact WCET (this problem is the same for many real applications). Moreover, the average utilization of the evaluated task set is quite high, as a result, if we use the WCET of each task to reserve the corresponding server capacity, the total server capacity will be much higher than the available system bandwidth (i.e. 1). Therefore, in BACKSLASH, the server capacity of each task is set according to its average execution time.

[3]During the runtime, the temporary system utilization may exceed the given average utilization.

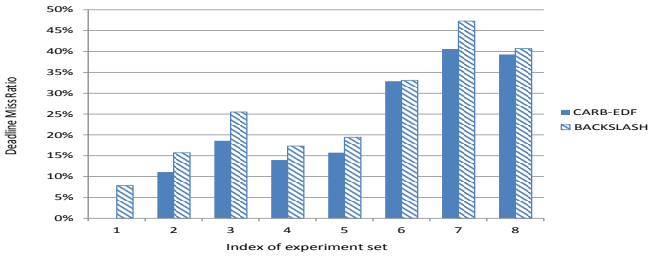

Fig. 6. The performance comparison between our framework and BACKSLASH. The x axis denotes the index of experiment sets, and the y axis represents the DMR of the whole system. For each task set, the results are collected from 50000 jobs.

The results of 8 sets of experiments have been presented in Figure 6. As shown in the figure, in the first set of experiment, the DMR of our framework is very low (i.e. 0.01%), but the DMR of BACKSLASH is around 8%. In the worst case (i.e. the 7th set of experiment), the DMR of our framework is less than 41%, but the DMR of BACKSLASH is around 47%.

Generally, we can clearly observe that our framework outperforms BACKSLASH. The main reason is that BACKSLASH does not have capacity adaptations. The performance of BACKSLASH (and also most of the existing capacity reservation scheduling frameworks) depends on the allocation of server capacities. For a task whose execution times can frequently exceed the reserved server capacity, the DMR can become very high especially when the total system utilization is high. Because if the reclaimed capacities from other jobs are not enough, a job of this task may have to borrow from its future jobs where it needs to extend its deadline (i.e. lower its priority) as a price. This capacity borrowing can even cause a domino effect, as result, many following jobs may miss their deadlines. However, due to the capacity adaptation mechanism, our framework can decrease the occurrence of the domino effect by assigning more capacities to future jobs.

2) Case Study: In addition to the above hypothetical task sets, we also evaluate our framework using the video decoder task which is also used in Section IV-A. In the following experiments, we compare our framework with both BACKSLASH and a framework only with capacity adaptation and reclaiming mechanisms (i.e. without capacity borrowing).

Besides the video decoder task, we add three other hypothetical tasks in the system. The average total task utilization is set around 0.85. The results are presented in Table II.

In the first experiment set, the video decoder task is considered as the most important task (i.e. it has the highest criticality level, $cl_i = 4$). In this case, the DMR of the video decoder

	Set 1				Set 2				Set 3			
	cl_i	DMR_{CARB}	DMR_{BS}	DMR_{CAR}	cl_i	DMR_{CARB}	DMR_{BS}	DMR_{CAR}	cl_i	DMR_{CARB}	DMR_{BS}	DMR_{CAR}
τ_v	4	0.06%	18.28%	1.92%	3	0.11%	18.28%	4.21%	1	0.22%	18.28%	5.57%
τ_1	3	0.39%	0.16%	1.01%	4	0	0.16%	0.09%	3	0.16%	0.16%	0.36%
τ_2	2	0.29%	2.01%	0.56%	2	0.36%	2.01%	0.77	2	0.37%	2.01%	0.92%
τ_3	1	0.06%	1.1%	0.2%	1	0.04	1.1%	0.08	4	0	1.1%	0.02%

TABLE II

THE PERFORMANCE EVALUATION USING A CASE STUDY. τ_v DENOTES THE VIDEO DECODER TASK, AND THE OTHER TASKS ARE RANDOMLY GENERATED HYPOTHETICAL TASKS. DMR_{CARB} REPRESENTS THE DMR OF TASKS IN OUR FRAMEWORK, DMR_{BS} DENOTES THE DMR OF TASKS USING BACKSLASH, AND DMR_{CAR} DENOTES THE DMR OF TASKS IN THE FRAMEWORK ONLY WITH CAPACITY ADAPTATION AND RECLAIMING (WITHOUT BORROWING)MECHANISMS. A LOWER VALUE OF cl_i REPRESENTS A LOWER CRITICALITY LEVEL.

task in our framework is 0.06%, but under BACKSLASH, the DMR of this task is 18.28%. While considering the whole system, the total DMR of our framework is also much lower than BACKSLASH. Moreover, for the framework where we only keep the capacity adaptation and reclaiming mechanisms (i.e. capacity borrowing is disabled), the performance is significantly worse than our framework but still better than BACKSLASH. The main reason of the performance degradation of this framework is that runtime overruns can only be handled by limited reclaimed capacities. When the execution time distributions of some tasks have low standard deviations (i.e. less spread), the reclaimed capacities will also become low, because the reserved capacities are close to the actual execution times due to the capacity adaptation process. As a result, when a large job overrun occurs, the reclaimed capacities may not be sufficient to handle it. In this case, the job with overrun needs to use the benefits of capacity borrowing mechanism.

In the second experiment set, we switch the criticality levels of τ_v and τ_1 (i.e. lower down the criticality level of τ_v). As expected, in our framework, the DMR of the video decoder task increases to 0.11%. This is because when the total system utilization is very high, the scheduler will provide more service to the more critical tasks. As a less critical task, τ_v may not be able to get sufficient capacity. On another hand, since the criticality level of τ_1 increases, its DMR decreases to 0. However, BACKSLASH does not have the concept of criticality levels, which means that all the tasks are treated evenly. Therefore, the results of BACKSLASH do not change.

In the third experiment set, we further lower down the criticality level of τ_v by switching with τ_3 (i.e. with the lowest criticality level now). Similar to the above observations, in our framework, the DMR of τ_v increases to 0.22%, and the DMR of τ_3 decreases 0.

Besides the measurement of DMR, we also examined the tardiness [24] of the tasks under different scheduling frameworks. The tardiness of a job represents the distance between its actual finishing time and its original absolute deadline. A negative tardiness means that the job misses its deadline, and a positive tardiness means that the job can meet its deadline. The results of the video decoder task in the third experiment set are presented in Figure 7. As shown in the results, for the jobs that missed their deadlines (i.e. with negative tardiness), the largest tardiness under our framework is around 25% of the task period, while the tardiness under BACKSLASH may go beyond 100% (i.e. a video frame is displayed one frame later than its originally expected time). Moreover, the results also show that, under BACKSLASH there are many successive jobs missing deadlines together, which is known as the previously presented domino-effect due to the capacity borrowing mechanism. However, in our framework, this problem occurs much less frequently.

	U=0.5		U=1	
Task Num	Mean	STD	Mean	STD
6	0.022	0.178	0.024	0.156
8	0.021	0.151	0.044	0.206
10	0.027	0.164	0.038	0.206
20	0.031	0.172	0.058	0.237

TABLE III

THE OVERHEAD OF CAPACITY ADAPTATIONS. THE RESULTS ARE PRESENTED IN *ms*. THE EVALUATION PC USES INTEL I5-3320 CPU @ 2.6 HZ, 8G RAM, AND WINDOW 7.

According to the results of this case study, we can get the same observation as the former experiments that our framework can provide a better performance than BACKSLASH in general. On another hand, our framework can provide system designers more controls on the tasks through the concept of criticality levels. This can be very helpful for real applications where tasks with different levels of importance are integrated into one system.

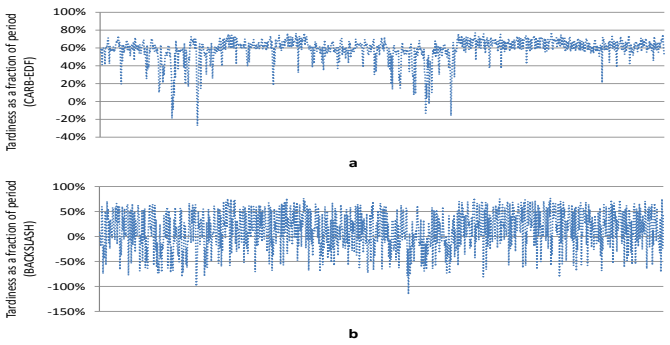

Fig. 7. The tardiness of the video decoder task under different scheduling frameworks.

3) Overhead: As shown in the above result, our framework can provide better performance than BACKSLASH due to the capacity adaptation mechanism. However, in order to achieve the capacity adaptations, some extra operations (e.g. collecting samples, calculating and updating statistical information, reallocating server capacities, etc.) need to be added to the scheduler. As an online scheduling framework, the overhead needs to be controlled so that the system performance will not be degraded. Therefore, in this section, we present the evaluation of the extra overhead of our framework.

Theoretically, the overhead is affected by both system utilization and number of tasks. Therefore, we generate a number of experiments with different values of these two system parameters. The total overhead of the adaptation process for each task is measured in real time, and the results are presented in Table III. We notice that, for most of the jobs whose executions do not cause capacity reallocation (i.e. do not satisfy Algorithm 1 line 10), the scheduler just needs to perform a few simple operations where the overhead is less than 1 μs. Due to the limitation of our simulator, we cannot capture the precise time less than 1 μs. Therefore, the overhead which is less than 1 μs is alternatively considered as 1 μs. In other words, the results shown in the table is more pessimistic than the real values.

As shown in Table III, the average overhead increases gradually as the number of tasks and system utilization go up. Generally, these overheads are relatively low. Considering a video decoder task which is used in our case study, the average execution time of each job is around 15 ms. Even in the worst experiment set presented above, the average overhead is less than 0.5% of the average job execution time. Therefore, we believe that the extra overhead due to our capacity adaptation mechanism is acceptable in reality.

V. Conclusion and Future Work

In this paper, we present a new reservation-based scheduling framework for soft real-time systems using EDF algorithm (called CARB-EDF). This framework has the features of capacity adaptation (based on statistical predictions), reclaiming and borrowing. Due to the capacity adaptation mechanism, system designers do not need to provide any estimations of task execution times in order to define proper selections of server capacities. On the other hand, the imprecise runtime predictions of task execution times can be compensated by the capacity reclaiming and borrowing mechanism. Therefore, our framework can provide a good performance in general.

A number of simulation-based experiments have been implemented including both hypothetical experiments and a case study using video decoder tasks. First, we measured the predictor used in our framework with the one presented in [11]. As shown in the results, our tunable predictor can provide better estimations. We also compare the performance of our scheduling framework with a similar framework BACKSLASH which does not have the capacity adaptation mechanism. According to the result, our framework can provide a better performance regarding both deadline miss ratios and tardiness. Finally, we examined the extra overhead of our scheduling framework regarding the capacity adaptation process. The results show that the overhead can be acceptable in reality.

In our future work, we would like to implement the presented scheduling framework in a real platform and examine the actual runtime performance. On another hand, the current framework focuses on single-core systems, we will also try to extend this framework to multi-core systems.

References

[1] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "Qos control strategies for high-quality video processing," *Real-Time Systems*, 2005.

[2] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *the SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[3] M. Caccamo, G. Buttazzo, and L. Sha, "Handling execution overruns in hard real-time control systems," *IEEE Transactions on Computers,*, 2002.

[4] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998, pp. 4–13.

[5] S. Banachowski, T. Bisson, and S. Brandt, "Integrating best-effort scheduling into a real-time system," in *25th IEEE International Real-Time Systems Symposium (RTSS'04)*, December 2004, pp. 139–150.

[6] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers," in *12th Euromicro Conference on Real-Time Systems (ECRTS'00)*, 2000, pp. 193–200.

[7] G. Bernat, I. Broster, and A. Burns, "Rewriting history to exploit gain time," in *25th IEEE International Real-Time Systems Symposium (RTSS'04)*, December 2004, pp. 328–335.

[8] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, 2005, p. 314.

[9] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, "The case for feedback control real-time scheduling," in *11th Euromicro Conference on Real-Time Systems (ECRTS'99)*, 1999, pp. 11–20.

[10] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, "Feedback–feedforward scheduling of control tasks," *Real-Time Systems*, 2002.

[11] N. M. Khalilzad, M. Behnam, and T. Nolte, "Multi-level adaptive hierarchical scheduling framework for composing real-time systems," in *19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, August 2013.

[12] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, 1973.

[13] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *Real-Time Systems Symposium (RTSS'92)*, 1992, pp. 110–123.

[14] T.-S. Tia, J. W.-S. Liu, and M. Shankar, "Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems," *Real-Time Systems*, 1996.

[15] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "Iris: a new reclaiming algorithm for server-based real-time systems," in *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, May 2004, pp. 211–218.

[16] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *21st IEEE Real-Time Systems Symposium (RTSS'09)*, 2000.

[17] M. Caccamo, G. C. Buttazzo, and D. C. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *Computers, IEEE Transactions on*, 2005.

[18] L. Nogueira and L. M. Pinho, "Capacity sharing and stealing in dynamic server-based real-time systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, 2007, pp. 1–8.

[19] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, "Design and evaluation of a feedback control edf scheduling algorithm," in *20th IEEE Real-Time Systems Symposium (RTSS'99)*, 1999, pp. 56–67.

[20] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosaadaptive quality of service architecture," *Software: Practice and Experience*, 2009.

[21] D. S. Moore and G. P. McCabe, in *Introduction to the practice of statistics*. W. H. Freeman and Company, 2006.

[22] P. Tchebichef, "Des valeurs moyennes," in *Journal de mathmatiques pures et appliques*, 1867, pp. 177–184.

[23] N. Khalilzad, M. Behnam, G. Spampinato, and T. Nolte, "Bandwidth adaptation in hierarchical scheduling using fuzzy controllers," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, June 2012, pp. 148–157.

[24] A. Srinivasan and J. Anderson, "Efficient scheduling of soft real-time applications on multiprocessors," in *15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, July 2003, pp. 51–59.