A Server-based Approach for Overrun Management in Multi-Core Real-Time Systems

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*—This paper presents a server-based framework for task overrun management in multi-core real-time systems. Unlike most existing scheduling methods which usually assume a single upper bound of the Worst-Case Execution Time (WCET) for each task, our approach targets scenarios with task overruns. The main idea of our framework is to employ Synchronized Deferrable Servers (SDS) to deal with globally scheduled task overruns, while a partitioned scheduling approach is applied on regular task executions. Moreover, we provide a deterministic Worst-Case Response Time (WCRT) analysis focusing on hard timing constraints, along with a probabilistic analysis of Deadline Miss Ratio (DMR) for soft real-time applications. In the evaluation phase, we have implemented two types of experiments evaluating different timing constraints.**

I. Introduction

A. Motivation

The motivation of this paper originates from the overrun problem in some industrial real-time embedded systems. For some applications the estimated Worst-Case Execution Time (WCET) may be violated during runtime. For example, path planning and perception tasks in mobility robots have diversity in their execution times. As a result, if we do not handle task overruns in a proper way, the reliability of a system can be seriously impacted. This kind of problem has been addressed in [1], where a resource reservation methodology is proposed. Due to the usage of resource reservation mechanisms, systems can adapt themselves to runtime changes. Under resource reservation mechanisms, each task (or task set) can have a reserved processor bandwidth, such that it will not be influenced by emergent incidents (e.g. overruns from other tasks). Nevertheless, as far as we know, for most existing resource reservation mechanisms, the scenario that the execution of a certain task exceeds its reserved bandwidth has not been well concerned from the schedulability point of view, especially looking at the context of multi-core real-time systems. In other words, due to the task isolation by resource reservation mechanisms, the overrun of a task may not impact the execution of other tasks. However, the actual overrun of this particular task is not well managed in most existing approaches, and as a result the schedulability for the task with overruns may become very low.

On another hand, in many real-time control applications, the execution time of a task may vary greatly [2]. For those applications, a task usually has a short execution time during most of the time, while the WCET rarely occurs. As a result, the use of the WCET can obviously affect the performance optimization of a system. This problem can be even worse in soft real-time systems. Therefore, we need a scheduling framework which can handle the above problem and at the same time guarantee system schedulability. Some works targeting a similar problem have been proposed (e.g. [2][3]). In those works, each task gets a reserved bandwidth based on their mostly occurred execution time, while the executions exceeding the reserved capacity (e.g. when the WCET occurs) are handled by an overrun management. Unfortunately, those works only focus on single-core systems.

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

In order to tackle the overrun problem in the setting of multi-core systems, in this paper, we present a new Synchronized Deferrable Server (SDS) based hierarchical scheduling framework to handle task overruns. Moreover, schedulability analysis is provided targeting both hard and soft real-time applications.

B. Contributions

1). We propose an SDS-based scheduling approach for multi-core real-time systems, which, we believe, is the first framework concerning the holistic schedulablity of both regular task executions and overruns in multi-core real-time systems. In our framework, task overruns are only processed in the globally shared servers. By isolating overruns from regular task executions, each task can get a guaranteed service for its regular execution, and overruns can be processed by the spare slacks in the system via SDSs. As a result, our framework can spare many tasks from missing their deadlines due to the occurrence of overruns in the system. This framework can be utilized in different ways. For example, the concept of task overruns in this paper can be considered as extra fault recovering executions from a fault-tolerant perspective, or another critical level of executions from a mixed-criticality point of view.

2). A deterministic Worst-Case Response Time (WCRT) analysis for our framework is provided. This analysis can provide a sufficient schedulability test regarding hard timing constraints.

3). We also propose a method to compute an upper bound of the Deadline Miss Ratio (DMR) of each task in our framework targeting soft real-time applications. The estimated DMR can be utilized for variant purposes. For example, it can be used to predict temporal drops of frame rates for path planning and perception tasks.

4). Regarding different timing constraints, we have implemented a number of experiments. We explicitly examined the effects of different system parameters. The comparisons between different allocation algorithms and different server bandwidth selection schemes are provided along with discussions regarding the observations. According to the experimental results, our global overrun management framework always outperforms the local overrun-handling solutions.

C. Related Work

A lot of researchers have been working on task scheduling problems for multi-core real-time systems. The traditional scheduling methods for multi-core systems can be categorized into two types: partitioned and global scheduling. In partitioned scheduling, tasks are statically allocated on different processors. Once a task is assigned to a certain processor, it cannot migrate to any other processors. Due to this feature, the tasks in the same processor can always be scheduled under uniprocessor scheduling algorithms (e.g. Rate Monotonic (RM) [4], Deadline Monotonic (DM) [5], Earliest Deadline First (EDF) [6]). The disadvantage of partitioned scheduling is that the task allocation problem can be considered as a bin packing problem which is NP-hard [7]. Under global scheduling, tasks are usually controlled in a common queue shared among processors, and a task can migrate from one processor to another. In this case, those well-developed theories for uniprocessors may not be valid anymore, because different features of multi-core systems (e.g. parallel executions) need to be taken into account. Another type of approach which is known as semi-partitioned scheduling has also been developed (e.g. [8][9]). This type of scheduling is a combination of the traditional partitioned and global scheduling, where some tasks are partitioned on different processors and the other tasks are globally scheduled.

Some works have also been proposed regarding hierarchical scheduling approaches (e.g. [1][10][11][12]). Under hierarchical scheduling, a certain processor bandwidth (i.e. usually denoted by a *server*) can be reserved for different purposes (e.g. to handle aperiodic tasks), and different schedulers can be integrated in one system. Due to the usage of resource reservation mechanisms, systems can adapt themselves to runtime changes. Under resource reservation mechanisms, each task (or task set) can have a reserved processor bandwidth, such that it will not be influenced by emergent incidents (e.g. overruns from other tasks). In [13], the authors proposed a Synchronized Deferrable Server (SDS) based hierarchical scheduling method, where the migrating tasks are processed only in the globally shared servers. Our framework can be considered as a type of hierarchical scheduling employing SDSs, where both partitioned and global scheduling are involved.

Several works focusing on the overrun problems have been proposed in literature. In [2], the authors present an approach to handle task overruns targeting the EDF algorithm. They proposed a rate adaption method based on the Constant Bandwidth Server algorithm [14] to fully utilize a processor. In [3], the authors compared several scheduling algorithms considering task overruns. These algorithms include both classical priority based (DM and EDF) and server based scheduling algorithms. However, most of the existing scheduling frameworks considering task overruns only focus on single core systems, and our framework targets multi-core systems.

The rest of this paper is organized as follows. Section II introduces the system model and the scheduling policy of our framework. In Section III, we propose a deterministic WCRT analysis, as well as a probabilistic analysis of DMRs. The results and observations of our experiments are discussed in Section IV. Finally, in Section V, we conclude the paper and provide some thoughts about future works.

II. System Model

A. Task Model

In our model, a multi-core system includes a set of periodic or sporadic tasks $⅂ = \{\tau_1, \tau_2, ..., \tau_N\}$. Each task can be characterized as $\tau_i = \{T_i, \overrightarrow{C_i}, P_i, D_i\}$. A period T_i denotes the minimum inter-arrival times between successive instances of a task. A matrix $\overrightarrow{C_i}$ represents the execution times of a task along with the corresponding probabilities, which will be explained in detail in the next paragraph. Each task owns a fixed priority P_i, which is used for the local scheduling on a processor (i.e. the priority of a task has nothing to do with any other tasks which are executed on another processor). The timing constraint is given by the relative deadline D_i of a task. A system is considered to be acceptable when the Deadline Miss Ratio (DMR) (i.e. the probability of the response time of a random task instance to exceed its deadline) does not exceed a certain bound $\hat{Pr}^{DMR}$. In a valid hard real-time system, all the tasks are supposed to meet their respective deadlines (i.e. the response times do not exceed their deadlines), which means that $\hat{Pr}^{DMR} = 0$. In this paper, we only assume constrained deadlines (i.e. $D_i \leq T_i$). Due to the fact that in our task model, the regular execution time of a task is given by an upper bound instead of a certain distribution, we can only generate a corresponding upper bound of the response time of a regular execution. As a result, the DMRs computed in this paper only focus on the occurrence of overruns, which is different from the concept of DMR used in some existing probabilistic analysis (e.g [15]). In other words, for each task τ_i, our analysis provides an approximated upper bound of its DMR rather than an exact DMR. Moreover, we assume that overheads caused by preemptions and migrations are negligible or included in the execution times (as discussed in [16]).

Instead of using a single upper bound of the execution time of a task, in our system model, the execution time of each task τ_i is characterized by a 2×2 matrix $\overrightarrow{C_i} = \begin{pmatrix} C_i^R & C_i^O \\ Pr_i^R & Pr_i^O \end{pmatrix}$. In most cases, the execution time of a task is bounded by a maximum *regular* execution time C_i^R, which is associated with a probability Pr_i^R (i.e. Pr_i^R represents the probability of that only the regular execution occurs without an overrun). However, sometimes because of certain factors (e.g. system errors, hardware effects, etc.), a task may include an overrun. Due to this extra execution, the maximum execution time of τ_i may be increased up to C_i^{R+} ($C_i^{R+} > C_i^R$). The upper bound of the execution time of the overrun can be acquired accordingly as $C_i^O = C_i^{R+} - C_i^R$, which is associated with a probability Pr_i^O ($Pr_i^O = 1 - P_i^R$). If C_i^O and Pr_i^O for each task τ_i equate to 0, this task model will be the same as the traditional task model with single execution time bounds. Moreover, we assume that for each task, there can be at most one instance processed in the system at each time. In other words, when a job of a task is ready to be released, if the execution of its previous instance from the same task has not been completed, the system should either discard the newly arrived instance or its previous instance (depending on specific applications). In our future work, we will further relax this assumption.

B. Deferrable Servers

We employ the concept of *Deferrable Servers (DS)* in our multi-core system model (similar to [13]). A DS can be considered as a periodic task, which has its own replenishment period T_s and capacity C_s. Within each replenishment period (T_s), a DS can execute pending workloads until all of its capacity (C_s) is consumed. At the beginning of each replenishment period, the capacity of a DS will be replenished up to C_s. In our model, we assume that each processor contains at most one DS, which has the highest priority among all the tasks located in the same processor. In other words, once a DS detects assigned workloads and meanwhile it still holds available capacity within the current replenishment period, this DS will preempt the execution of other tasks in the same processor. If a DS has consumed all of its capacity for a period, the remaining workloads need to wait until the arrival of the next replenishment period.

In this paper, we assume that the deferrable servers on different processors are synchronized. In other words, all the servers have the same initial phase (i.e. released simultaneously), and share a common replenishment period. However, different servers may have different capacities, which depend on the workloads on each processor. Therefore, we employ the same notation as in [13] to represent an m-SDS (i.e. Synchronized Deferrable Servers on m processors)[1], that is (T_s, C_s^1, C_s^2, ..., C_s^m). We also assume that the capacities in the tuple are sorted in a descending order (i.e. $\forall k$, $C_s^k \geq C_s^{k+1}$). The SDS parameters (T_s *and* C_s^i) can be selected in many different ways. As discussed above, we can consider a DS as a task with the highest priority in each processor. Therefore, using an optimal fixed-priority scheduling algorithm (e.g. Rate Monotonic (RM)), a valid T_s can be selected based on the periods of other tasks in the same processor. Given a certain utilization U_s^i of a DS, the corresponding capacity can be calculated accordingly (i.e. $C_s^i = T_s \cdot U_s^i$).

We divide the whole capacity of a processor into two parts. The first part is used to process the regular executions of partitioned tasks, where the execution time of a task is upper bounded by C_i^R. All the remaining capacity in each processor can be assigned to the SDS. Unlike [13] and [17], the servers in our model are only used to execute the overruns of tasks in a system but not any complete task, where the overrun of a task is upper bounded by C_i^O. Moreover, the regular executions of tasks are statically partitioned on different processors, while the overruns are processed on the globally shared SDSs. In other words, only the overruns of tasks can migrate from one processor to another. In this paper, we assume that the overruns are scheduled employing a stated global First-In-First-Out (FIFO) queue, where each instance in the queue may have either a *Waiting* state or a *Running* state.

[1]In our framework, the number of SDSs is smaller than or equal to the number of processors in the system (i.e. some processors may not contain any SDSs).

By isolating the overrun of a task from its continuation of the regular execution to globally shared servers, we can decrease the probability of that an overrun of a task may cause more overruns of other tasks. In other words, each task is provided a guaranteed bandwidth for its regular executions, which may not be much affected by the overruns of other tasks especially when some extremely long overruns occur.

C. Scheduling Policy

Regarding the regular execution and the overrun of a task, the scheduling mechanism is separated into two levels as follows.

1) Scheduling for Regular Executions:

- The regular executions of all the tasks are allocated on different processors using a certain allocation algorithm, such as First-Fit, Worst-Fit, Best-Fit, etc. ([7][18]).
- Within each processor, all the tasks together with the local SDS (if there is one) are scheduled using a fixed-priority uniprocessor scheduling algorithm. Note that an SDS always has the highest local priority.

2) Scheduling for Overruns:

- Once the execution time of a task exceeds a certain bound (i.e. the upper bound of its regular execution C_i^R), this task needs to release the processor immediately. The remaining execution of this task τ_i^O will be appended to the end of a stated global FIFO queue Q_g, τ_i^O will then be marked with a Waiting state.
- Once there is no job marked with Waiting ahead of τ_i^O in Q_g (i.e. all jobs ahead of τ_i^O are in the Running state), τ_i^O will be dispatched to an available SDS which has remaining capacity within the current replenishment period. Note that τ_i^O will still be kept in Q_g at this phase. If there are more than one available server, the one with the lowest index will be selected. As soon as τ_i^O is assigned to an SDS, the state of τ_i^O will be changed to Running.
- Once an SDS receives some workload, it will immediately preempt the executions of other local tasks until all of its capacity within the current replenishment period is consumed or the workload is completely processed.
- Once an SDS completes the execution of an overrun τ_i^O, τ_i^O will be removed from Q_g. If this SDS still has some remaining capacity, it will continue to process another pending workload which is the first job with a Waiting state in Q_g.
- If the capacity of an SDS is exhausted while processing τ_i^O, the remaining execution will be suspended. Then the remaining part of τ_i^O will preempt the execution of the last Running job which is queued after τ_i^O in Q_g, and the state of this preempted job will be changed to Waiting. If τ_i^O is the last Running job in Q_g (i.e. all jobs queued after τ_i^O are in the Waiting state), τ_i^O will go to the Waiting state.

- If all the servers have already exhausted their capacities, the remaining workloads of overruns need to wait until the arrival of the next replenishment period.

III. Schedulability Analysis

In this section, we present a response-time based schedulability analysis for the above system model. First, we present the calculation of the deterministic upper bounds of task WCRTs, which focuses on hard real-time constraints. In the second subsection, we propose an approach to calculate an upper bound of the DMR of each task, which can be utilized in soft real-time systems.

A. Deterministic Worst-Case Response Time Analysis

As we introduced in Section II, in our task model, the worst-case response time of a task consists of two sources: the regular execution and the overrun execution. Therefore, we can compute the WCRT of a task τ_i as

$$R_i = R_i^R + R_i^O \tag{1}$$

where R_i^R and R_i^O represent the response times of the regular execution and the overrun execution respectively. We will present the calculation of R_i^R and R_i^O in the following subsections.

In this paper, the release jitter (i.e. the timing duration between the arrival and the release of a task instance) is not considered for simplicity. Therefore, while analyzing the schedulability of a system, we only need to compare R_i with D_i.

1) WCRT of Regular Executions: As mentioned in our scheduling policy, the partitioned regular executions of tasks are scheduled using a fixed-priority uniprocessor scheduling algorithm, where an SDS can be considered as a periodic task. Therefore, we can simply utilize the existing approaches (e.g. [19][20]) to compute R_i^R.

Considering that we assume a preemptive mechanism in our model, R_i^R should[2] only consist of the upper bound of the regular execution time C_i^R and the interference I_i^R caused by other tasks in the same processor with higher priorities (including the SDS). Therefore, we can compute R_i^R as

$$\begin{aligned} R_i^R(n+1) &= C_i^R + I_i^R \\ &= C_i^R + \sum_{\substack{\forall \tau_j \in hp(\tau_i) \\ \wedge \tau_j \in S_i}} \lceil \frac{R_i^R(n)}{T_j} \rceil \cdot C_j^R + \\ &\quad (\lceil \frac{R_i^R(n) - C_s(i)}{T_s} \rceil + 1) \cdot C_s(i) \end{aligned} \tag{2}$$

where the task set S_i contains all the regular tasks assigned on the processor where task τ_i is located, and $C_s(i)$ denotes the maximum capacity of the SDS on that processor. The task set $hp(\tau_i)$ includes all the regular tasks whose priorities are higher than or equal to P_i. If there is no SDS on the processor, $C_s(i) = 0$ in Eq. 2.

[2]Note that in this paper the resource sharing problem is not taken into account for simplicity, therefore, the blocking term is not included in our calculation.

Note that while calculating the interference caused by the SDS, we cannot simply consider it as other regular tasks because of the carry-in instance of an SDS. More details can be found in [21].

Eq. 2 can be solved using a fixed point iteration solution. We can start the calculation with $R_i^R(0) = C_i^R + C_s(i)$, and iteratively compute $R_i^R(n+1)$ until: 1) $R_i^R(n+1) > D_i$, which means that τ_i has missed its deadline; 2) or $R_i^R(n+1) = R_i^R(n)$, in which case $R_i^R(n+1)$ is considered as the final result.

We need to emphasize that the above equation can only provide an upper bound of R_i^R but not the exact worst-case result due to the pessimism. Eq. 2 assumes that an SDS is completely occupied all the time, which may not be necessary in reality. As a result, the above computation gives a sufficient but not necessary schedulability test.

2) WCRT of Overruns: As introduced in Section II, the overruns of tasks are scheduled in a global FIFO queue and can be processed by the SDSs on different processors[3]. Therefore, once the overrun of a task arrives at the global FIFO queue Q_g, it only needs to wait for the executions of task overruns which are queued ahead. All the task overruns cannot preempt any other earlier arrived task overrun. Note that sometimes the execution of a task overrun τ_i^O may be suspended because the current SDS has exhausted its capacity. Then τ_i^O may preempt the execution of the jobs which are queued after τ_i^O.

The workloads queued ahead of τ_i^O is denoted by QW_i, and the workload of τ_i^O itself is upper bounded by C_i^O.

We use S_i^O to denote the time instant when τ_i^O starts to be executed. The response time of τ_i^O may consist of the processing time for the workloads which are queued ahead of τ_i^O (i.e. the time duration from the beginning of QW_i to S_i^O, denoted by r_i^H), and the processing time of τ_i^O itself (i.e. the time interval from S_i^O to the end of τ_i^O, denoted by r_i^B). We compute R_i^O as

$$R_i^O = T_s - C_s^m + r_i^H + r_i^B \tag{3}$$

where $T_s - C_s^m$ represents the maximum waiting time before the start of QW_i (i.e. denoted as the critical head $\widehat{H_k^C}$ in [13]). This upper bound may include pessimism unless all the SDSs have the same capacity (i.e. $C_s^1 = C_s^2 = ... = C_s^m$). The proof of $\widehat{H_k^C}$ can be found in Lemma 1 in [13].

Before computing r_i^H, we need to find an upper bound of the queueing workloads QW_i.

Lemma 1. *While computing QW_i, we only need to consider the queueing workloads QW_i ahead of a task overrun τ_i^O which is upper bounded by*

$$QW_i = \sum_{\substack{\forall \tau_j \in \urcorner \\ \wedge \tau_j \neq \tau_i}} C_j^O \tag{4}$$

Proof: As mentioned in Section II, we assume that each instance of a task cannot be released unless the execution of its previous instance is finished. Due to the feature of a FIFO queue, a later arrived job cannot preempt the executions of

[3]We assume that the overrun of a task cannot be executed on two or more SDSs simultaneously.

the jobs which are queued ahead in the queue. If the analyzed instance τ_i^O gets interference from two or more jobs of another task overrun τ_j^O, these instances of τ_j^O must be queued ahead of τ_i^O together in Q_g. This will apparently cause a contradiction of the above assumption. Therefore, while computing the workloads ahead of τ_i^O in Q_g, we only need to consider at most one instance of overrun for each other task in the system. ■

Once we have acquired the upper bound of QW_i, we can utilize part of the analysis presented in [13] to compute r_i^H. Both our scheduling policy and the scheduling algorithm presented in [13] can satisfy the work-conserving property (i.e. a processor will always be occupied whenever there is pending workload on it). The basic idea of the following computation is that the SDSs in the analyzed system are always busy within the time interval from the beginning of QW_i to S_i^O, and the latest occurrence of S_i^O will result in the maximum r_i^H under a given starting instant of QW_i. The proofs of the following computation can be found in [13].

$$r_i^H = (\lceil \frac{QW_i}{\sum_{i=1}^{m} C_s^i} \rceil - 1) \cdot T_s + t_{res} \tag{5}$$

where

$$t_{res} = \begin{cases} \dfrac{QW_i^{res}}{m} & if\ QW_i^{res} \leq \delta(m) \\ C_s^{k+1} + \dfrac{QW_i^{res} - \delta(k+1)}{k} & if\ \delta(k+1) < QW_i^{res} \leq \delta(k), \\ & \forall k \in [1, m-1] \end{cases} \tag{6}$$

$$QW_i^{res} = QW_i - (\lceil \frac{QW_i}{\sum_{i=1}^{m} C_s^i} \rceil - 1) \cdot \sum_{i=1}^{m} C_s^i \tag{7}$$

$$\delta(k) = \sum_{j=k}^{m} C_s^j + C_s^k \cdot (k-1), \quad \forall k \in [1, m] \tag{8}$$

In Eq. 7, $\lceil \frac{QW_i}{\sum_{i=1}^{m} C_s^i} \rceil - 1$ denotes the number of complete replenishment periods that QW_i may consume. Consequently, t_{res} represents the processing time of the residual workloads of QW_i in the last replenishment period (i.e. the capacities within this period are not exhausted). Based on t_{res}, we can compute r_i^B.

Lemma 2. *An upper bound of r_i^B can be computed as*

$$r_i^B = \begin{cases} C_i^O & if\ C_i^O \leq C_s^1 - t_{res} \\ T_s - t_{res} + CRP_i^O \cdot T_s + C_i^{res} & Otherwise \end{cases} \tag{9}$$

where

$$C_i^{res} = C_i^O - (C_s^1 - t_{res}) - CRP_i^O \cdot C_s^1 \tag{10}$$

$$CRP_i^O = \lceil \frac{C_i^O - (C_s^1 - t_{res})}{C_s^1} \rceil - 1 \tag{11}$$

Proof: As we discussed above, while τ_i^O is executing on a server, the jobs which are queued after τ_i^O may occupy other servers at the same time, that will definitely increase the response time of τ_i^O. In the worst case, we can assume that when τ_i^O is being processed on a server, there are always some jobs consuming all the other servers simultaneously. Under this assumption, we can compute the upper bound of r_i^B according to different sizes of C_i^O. The proof is given in Appendix A. ■

3) Calculation of WCRT: Based on the upper bounds of R_i^R (Eq. 2) and R_i^O (Eq. 3) computed above, we can derive an upper bound of R_i using Eq. 1. We need to confess that our analysis provides a sufficient but not necessary schedulability test due to the pessimism involved in the computation. We will further improve the analysis by decreasing pessimism in our future work.

B. Deadline Miss Ratio Calculation

In our task model, the regular execution time of a task is given by a single upper bound instead of a certain distribution. Moreover, the regular execution of each task is assumed to be mandatory, which means that the occurrence probability of τ_i^R is 100% for any task τ_i. Therefore, no matter if a task instance experiences an overrun or not, the regular WCRT is deterministically upper bounded by R_i^R. In other words, the probabilities in $\overrightarrow{C_i}$ are not involved while computing the response times of regular executions.

On the other hand, while computing R_i^O, the occurrence probability of a task overrun (i.e. Pr_i^O) can be taken into account. For simplicity, in this paper, we only consider the probabilities in QW_i-related computations, while the other parts of R_i still use the upper bounds computed in the previous subsections[4].

Lemma 1 implies that at most only one instance of each task overrun (except τ_i^O) needs to be included in QW_i. As described in our task model, the occurrence probability of a task overrun τ_i^O is given by Pr_i^O. Therefore, we can derive a discrete probability distribution of QW_i by

$$F_{QW_i} = \prod_{\substack{\forall \tau_j \in \daleth \\ \wedge \tau_j \neq \tau_i}} \Upsilon_i^O \tag{12}$$

where

$$\Upsilon_i^O = (\begin{smallmatrix} 0, & C_i^O \\ Pr_i^R, & Pr_i^O \end{smallmatrix}) \tag{13}$$

$$\prod_{k=1}^{n} = \Upsilon_1^O \otimes \Upsilon_2^O \otimes \ldots \otimes \Upsilon_n^O \tag{14}$$

$$\Upsilon_x^O \otimes \Upsilon_y^O = (\begin{smallmatrix} 0, & C_x^O, & C_y^O, & C_x^O + C_y^O \\ Pr_x^R \cdot Pr_y^R, & Pr_x^O \cdot Pr_y^R, & Pr_x^R \cdot Pr_y^O, & Pr_x^O \cdot Pr_y^O \end{smallmatrix}) \tag{15}$$

As defined above, F_{QW_i} can also be considered as a two-dimensional matrix (i.e. $F_{QW_i} = \begin{pmatrix} c_{QW_i}^1 & \cdots & c_{QW_i}^N \\ pr_{QW_i}^1 & \cdots & pr_{QW_i}^N \end{pmatrix}$, and the size is $(N, 2)$).

Apparently, in Eq. 5, QW_i is the only effective factor, since all the other factors are fixed or based on QW_i. Therefore, we can define Eq. 5 as a single variable function,

$$r_i^H(X) = (\lceil \frac{X}{\sum_{i=1}^{m} C_s^i} \rceil - 1) \cdot T_s + t_{res} \tag{16}$$

This function returns the result of r_i^H along with the t_{res} computed by the sub-equation (Eq. 6). In the same manner, Eq. 9 can be defined as a function of t_{res}, and Eq. 3 can be defined as a function of r_i^H and r_i^B. Then we can use Algorithm 1 to compute an upper bound of the DMR of τ_i.

[4]Considering the probabilities in the whole computation of R_i may result in an exponentially increased computing time. Therefore, we leave the further investigation to our future work.

Algorithm 1 Calculation of DMR_i

```
1: Initialize $\tau_i, pdf_i \leftarrow null, DMR_i \leftarrow 0$
2: $R_i^R \leftarrow Eq.\ 2$ [5]
3: $F_{QW_i} \leftarrow Eq.\ 12$
4: $(N, 2) \leftarrow sizeof(F_{QW_i})$
5: for all $k$ in [1,N] do
6:     $r_H \leftarrow 0, t_{res} \leftarrow 0, r_B \leftarrow 0, r_i \leftarrow 0$
7:     $r_H, t_{res} \leftarrow Eq.\ 5(c^k_{QW_i})$
8:     $r_B \leftarrow Eq.\ 9(t_{res})$
9:     $R_i^O \leftarrow Eq.\ 3(r_H, r_B)$
10:    $R_i \leftarrow R_i^R + R_i^o$
11:    $append\ (R_i, pr^k_{QW_i} \cdot Pr_i^O)\ to\ pdf_i$
12: end for
13: $(q, 2) \leftarrow sizeof(pdf_i)$
14: for all $j$ in [1,q] do
15:     if $pdf_i[j-1][0] > D_i$ then
16:         $DMR_i += pdf_i[j-1][1]$
17:     end if
18: end for
19: return $DMR_i$
```

According to different possible quantiles of QW_i, we can respectively compute the corresponding response times of τ_i (Algorithm 1 line 6-10). The quantiles of QW_i only make sense when the overrun of τ_i indeed occurs. Otherwise R_i^O is always 0 and R_i is constantly equal to R_i^R. Therefore, we also need to take Pr_i^O into account while deriving the distribution of R_i (Algorithm 1 line 11). Based on the distribution of R_i, we can compute an upper bound of DMR_i (Algorithm 1 line 14-18, where pdf_i denotes the created probability density function of the response times of τ_i).

Similar to our deterministic WCRT calculation, the DMR_i may also include pessimism because some parts of the response time still use upper bounds (e.g. $T_s - C_s^m$ in Eq. 3).

IV. Evaluation

In this section, we present some results and observations from our experiments. We have implemented two types of experiments regarding hard and soft timing constraints respectively.

A. Experiment Settings

1) Task Generations: In this paper, we use the UUniFast-Discard [16] algorithm to generate task sets for a 4-core system. Each task set consists of n tasks ($n \in [5, 30]$ with a granularity of 5) with a total utilization $4 \times U_t$, where U_t (i.e. the average utilization bound of each processor) is selected from [0.1, 0.6] with a granularity of 0.1. The period of each task is randomly selected from a uniform distribution with the range of [50000, 100000]. The deadline of each task is assumed to equate to its period. The execution time of a task overrun (C_i^O) is selected based on a certain percentage of its regular execution time (C_i^R). This percentage is uniformly selected within the range of (0, 150%). Moreover, we assume that all the tasks in our experiments can have overruns.

[5]Note that if the response time of the regular task execution of the analyzed task has already exceeded its deadline, we directly set its DMR to 1 and terminate the algorithm immediately.

2) Allocation of Regular Executions: In the following experiments, for the regular task executions, we employ the RM scheduling algorithm within each processor. When we partition regular task executions, we take into account both the First-Fit (FF) and Worst-Fit (WF) bin-packing algorithms.

Under the FF allocation algorithm, a new task τ_i is first assigned to an unfull processor with the lowest index. Then we check the schedulability of all the tasks (based on their regular execution times) in this processor. If all the tasks can meet their deadlines, then τ_i will be kept in this processor, and we start to allocate another unassigned task. Otherwise, we need to remove τ_i from this processor, and try to assign it to the next processor. If at the end τ_i cannot be assigned to any processor, the system is considered to be unschedulable. Due to the feature of the FF algorithm, the processors are usually unevenly utilized (i.e. the processors with lower indexes are usually highly occupied, while the others may contain much less workload). In this case, the response time of a task in a heavy-loaded processor may be very close to its deadline. As a result, if the overruns of all the tasks are still executed in the processors where their regular executions are located (which is called No Overrun Management (NOM) scheduling hereinafter), they are quite likely to miss their deadlines.

Under the WF algorithm, a new task τ_i will be assigned to the processor on which can leave the most capacity left over after this allocation. In other words, if τ_i can fit in several processors (i.e. all the regular task executions on these processors are schedulable after the allocation), we choose the processor with the lowest task utilization to allocate τ_i. If τ_i cannot fit in any processor, the system will be considered as unschedulable. For most cases, the loads of the whole system are almost evenly distributed on all processors.

3) SDS Parameters Selection: The replenishment period of a set of SDS is randomly selected in the range [1000, 10000] with a granularity of 1000 (same as [13]). In our experiments, we examine three bandwidth selection schemes.

In [13], the bandwidth of an SDS is chosen from (0.15, 0.3, and 0.5). Based on randomly selected replenishment periods, SDSs with different capacities are generated. The work in [13] provides a combination of global and partitioned scheduling, where the servers are used to process globally scheduled tasks. However, in our framework, the servers are only used to execute the overruns of partitioned tasks. We want to provide a guaranteed service to each task regarding its regular execution, so that the overrun of a task may impact other regular task executions as little as possible. Therefore, we select the parameters of an SDS (i.e. replenishment period and capacity) based on the slack remaining on each processor after the task allocation process. This is similar to the idea of the Slack Stealer which is presented in [22]. The details of the selection process are illustrated in Appendix B.

In Appendix B, the SDS capacity selection is based on response time analysis, therefore, we call it Response Time based Slack Stealer (RTSS) hereinafter. Besides the RTSS, we also consider a Utilization based Slack Stealer (USS). Under the USS scheme, the bandwidth of an SDS is also selected based on the remaining capacity of the corresponding processor, but the calculation is based on utilizations rather than response times. The Liu & Layland utilization bound [4] implies that as the number of tasks goes up, the affordable total task utilization will be decreased from 1. On another hand, if we use most remaining capacity, the response time of some regular task executions may be very close to their deadline. As a result, these tasks cannot afford any overruns. Therefore, we can only partially utilize the remaining capacity. In our experiments, the SDS bandwidth selection is based on some percentage (within the range of $[0.1, 0.6]$) of the remaining capacity.

Similar to [13], we also consider an even SDS bandwidth allocation scheme (denoted by EVEN). Under this scheme, all the SDSs have the same bandwidth, which is randomly chosen from $[0.1, 0.3]$ with a granularity of 0.05.

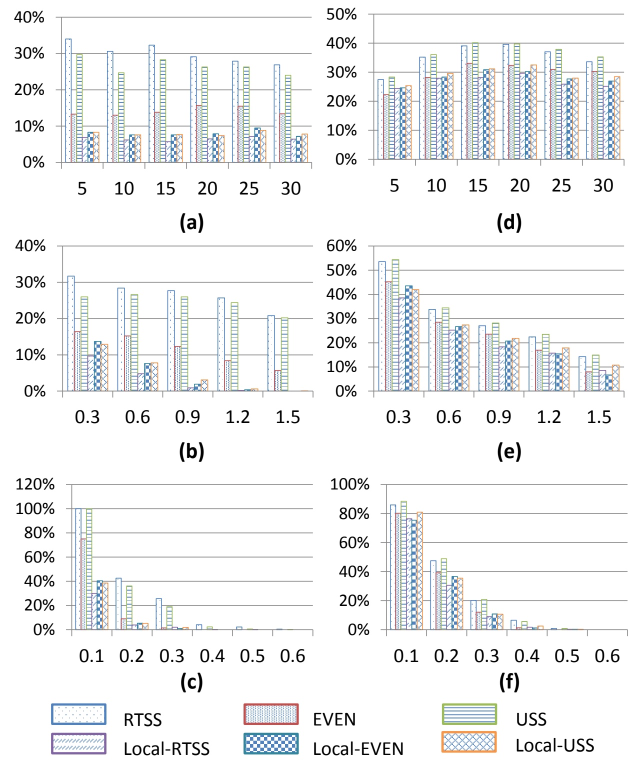

Fig. 1. Comparison between the FF (Figure a&b&c) and WF (Figure d&e&f) based framework regarding different system parameters. The y axis in these figures denotes the schedulability ratio per 1000 experimental observations. The x axis in Figure a&d denotes the number of tasks. The x axis in Figure b&e denotes the overrun percentage regarding regular executions. The x axis in Figure c&f denotes the average total task utilization for each core.

B. Experiments for Hard Real-Time Applications

In the first set of experiments, we focus on the deterministic WCRT analysis which is utilized under hard timing constraints. For each task set, we respectively apply our global SDS-based framework with different bandwidth selection schemes, as well as some local overrun management solutions presented in [3]. In [3], the authors used a Sporadic Server [23] based overrun management scheme for fixed priority systems with single core, which can be considered as a local overrun management solution in a multi-core system. Under the local overrun management scheme, the overruns of tasks are handled by the local servers (i.e. no migrations). Regarding different server bandwidth selection schemes, the results of the local management solution are marked as Local-RTSS, Local-USS and Local-EVEN. The comparison results are provided along with some observations.

1) Effects of System Parameters: First, we try to examine how partitioning algorithms and SDS bandwidth selection schemes can affect the schedulability of this framework. We implement several separate sets of experiments to investigate the impact of different effective factors (e.g. the number of tasks, the total task utilization, etc.). Within each experiment set, the observed factor is fixed for each sub experiment set, while all the other parameters are randomly selected as mentioned in the above subsections.

a) Number of Tasks: The first experiment set is regarding the number of tasks in the whole system. Given the total utilization of a task set, if the number of tasks is low, the utilization of each task will be high. In this case, the number of overruns is also low, but the execution times of overruns can become longer since we set the overrun time of each task based on its regular execution time. On another hand, when the number of tasks is high, the utilization of each task is mostly low. As a result, the execution time of each overrun is usually small, but the number of overruns becomes high. When we increase the number of tasks, we cannot clearly observe a trend of the schedulability changes, because the number of overruns and the execution times of overruns are balancing each other all the time.

As shown in Fig. 1.a, under the FF allocation algorithm, the RTSS and USS server bandwidth selection scheme have very close performance, which are always much better than the EVEN bandwidth selection. Because these two schemes take into account the actual remaining capacity on each core more precisely. On another hand, the local overrun management solution performs quite badly here. The main reason is that under the FF algorithm, there is always a heavy loaded core which has a very weak capability to handle overruns due to the quite limited server bandwidth. Therefore, in this case, our global overrun management framework performs much better.

As shown in Fig. 1.d, under the WF partition algorithm, the RTSS and USS bandwidth selection schemes are still better than the EVEN scheme. Under the WF algorithm, all the cores may provide much server bandwidth since the workloads are almost evenly distributed. Therefore, it is not common to see an extremely heavy loaded core in this case. That is why the local overrun management solutions are much better comparing to under the FF algorithm. However, the local overrun management still cannot outperform our method, since they do not utilize the capacities from other cores.

According to our server bandwidth selection schemes, if a core is totally free, we use the whole core for overruns; if a core has been assigned some regular task executions, we partially use the remaining capacities. In other words, under the FF algorithm, the total capacity for SDSs is usually larger than under the WF algorithm; however, the FF algorithm is strongly affected by the heavy loaded core.

Under the FF algorithm, when the number of tasks is low, there is a high probability that some cores can be completely used to handle overruns (i.e. workloads are distributed in the cores with lower indexes). However, under the WF algorithm, once the number of tasks is greater than the number of cores, there will be no free core which can be totally used for overruns. Therefore, as shown in the results, when the number of tasks is low, the FF algorithm outperforms the WF algorithm. However, when we increase the number of tasks, the occurrence of free cores under the FF algorithm becomes less frequent, but a heavy loaded core still usually exists. As a result, the WF algorithm becomes better than the FF algorithm.

b) Overrun Execution Time: This set of experiments focus on the execution times of task overruns. As shown in Fig. 1.b, under the FF algorithm, as the overrun time goes up, the schedulability of the global schemes decreases slightly. However, the performance of the local overrun management methods goes down sharply (i.e. almost non-schedulable when the overrun percentage exceeds 1.2). Under the WF algorithm (as shown in Fig. 1.e), the local overrun management methods become much better, but still limited by the local capacity on each core.

We notice that as the overrun time increases, the performance under the WF algorithm drops faster than under the FF algorithm. Under the FF algorithm, for the tasks on the heavy loaded cores whose response times are close to their deadlines, they have very weak capability to afford any overrun. Therefore, those tasks are not very sensitive to the changes of overrun times (i.e. they may always miss their deadlines no matter the overrun percentage is 0.3 or 1.2). On another hand, under the WF algorithm, heavy loaded cores do not frequently appear. As a result, the main effective factor for the performance changes in Fig. 1.b&e is the total overrun-handling capacity of a system. As we mentioned in the previous subsection, the WF algorithm has a high probability to provide less capacity than the FF algorithm, therefore, it is more sensitive to the increase of overrun times. However, when the overrun percentage is not very large (e.g. less than 0.9), the WF algorithm can always provide better performance because it can avoid the influence of heavy loaded cores.

c) Task Utilization: In this set of experiments, we try to examine the effect caused by the average total utilization of regular tasks executions per core (i.e. the utilization of the regular execution of τ_i can be computed as C_i^R/T_i). The results are shown in Fig. 1.c&f. As the system utilization goes up, the schedulability of all these methods decreases. When the average total utilization of regular executions exceeds 0.5 on each core, it is always non-schedulable.

When the utilization is very low (i.e. 0.1 per core), the global overrun management framework under the FF algorithm performs better than the WF algorithm. Similar to the previous discussion, under the FF algorithm, when the system utilization is low, there is a very high probability that some cores are totally free which can be utilized to handle overruns. However, under the WF algorithm, once the number of tasks is greater than the number of cores, there will be no empty cores in the system.

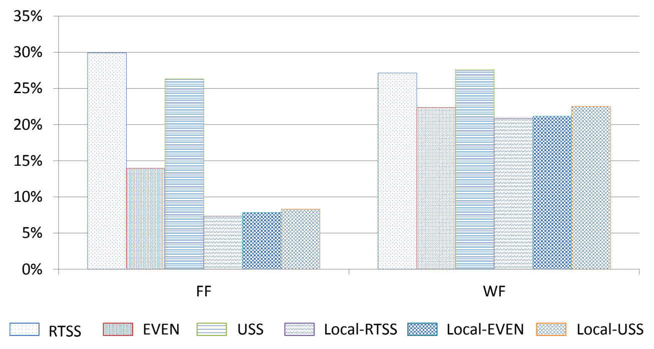

Fig. 2. FF VS. WF.

2) General Comparison: A general comparison is shown in Fig. 2. Here we summarize some key observations from the above discussions. Apparently, our framework is always better than the local overrun management solutions. Under the FF algorithm, the RTSS server bandwidth selection scheme is better than the USS approach. While under the WF algorithm, the USS scheme becomes slightly better than the RTSS approach. We need to point out that the RTSS algorithm may include more overhead than the USS scheme due to the computation of response times. Moreover, the FF algorithm has a higher probability to provide larger total overrun-handling capacity than the WF algorithm, because free cores are fully utilized. However, the WF algorithm can avoid heavy loaded cores on which the tasks may have weak capabilities to afford overruns.

3) Comparison with NOM scheduling: We also produced some experiments to compare our framework with the NOM scheduling (i.e. no overrun management). The results are shown in Fig. 3. Apparently, our framework always performs better. We need to point out that, the experiments are produced by analysis-based simulations. As we mentioned in Section III, in order to reduce the computation time, our schedulability analysis (both the deterministic and probabilistic analysis) includes some approximations which can lead to pessimism. Therefore, the actual performance of our framework can be even better. In our future work, we will further remove the pessimism in the analysis, and examine our framework from real implementations.

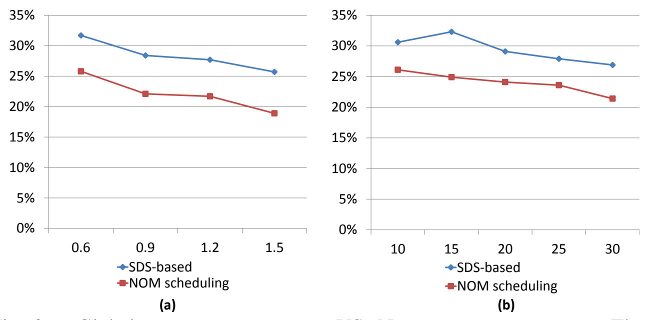

Fig. 3. Global overrun management VS. No overrun management. The y axis in these figures denotes the schedulability ratio per 1000 experimental observations. The x axis in Fig. a denotes the overrun percentage regarding regular executions. The x axis in Fig. b denotes the number of tasks. Note that the connection lines in all the figures are just used for the convenience of visualization.

C. Experiments for Soft Real-Time Applications

The second type of experiments take the probabilities of overrun occurrences into account, which focus on soft timing constraints. The systems are randomly generated in the same ways as presented in Section IV-A. The occurrence probabilities of task overruns are uniformly selected from the range [0, 5%].

As we know, the deterministic WCRT analysis assumes a guaranteed worst-case scenario where the task overruns always occur. However, the probability of that case may be very low in reality. Therefore, for soft real-time applications, we can set a DMR bound for the system under analysis. If the DMRs of all the tasks in a system do not exceed the given bound, we can still consider this system as acceptable. Due to the page limitation, here we just show some of the results. In this experiment set, we focus on the changes of the total task utilization. The regular tasks allocation is based on the FF algorithm, and the server bandwidth is selected with the USS approach. As shown in Fig. 4, when we increase the DMR bound, the average schedulability ratio of the system gains an obvious increase. That is why we believe that a probabilistic task model is very important for soft real-time systems.

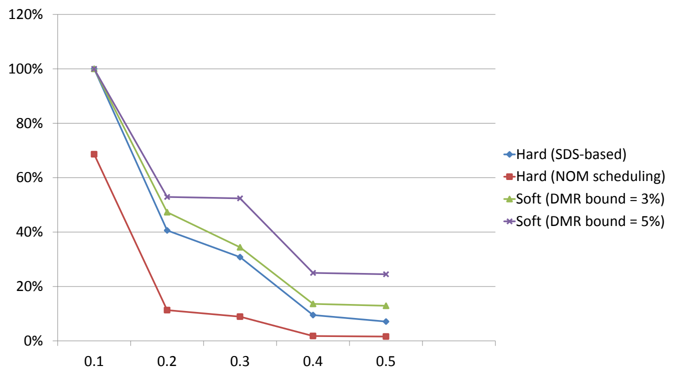

Fig. 4. Global overrun management (deterministic and probabilistic) VS. No overrun management. The y axis denotes the schedulability ratio per 1000 experimental observations. The x axis denotes the average total task utilization for each core.

V. Conclusion and Future Works

In this paper, we present a Synchronized Deferrable Server (SDS) based scheduling framework for multi-core real-time systems, where task overruns are taken into account. In our framework, task overruns are isolated from regular task executions by migration of overruns to globally shared servers. Consequently, many tasks can be saved from missing their deadlines due to the occurrence of overruns in the system. A deterministic Worst-Case Response Time (WCRT) analysis and a probabilistic Deadline Miss Ratio (DMR) computation are provided. Some systematic experiments regarding different parameters have also been implemented. According to the results of our experiments, our framework can provide better performance than the local overrun management schemes. When task overruns occur, our solution also always outperform the scheduling method without overrun management.

As mentioned above, both the deterministic WCRT analysis and the probabilistic DMR computation presented in this paper may include pessimism, because we utilize some guaranteed approximations in our analysis to decrease the complexity. Therefore, we need to further investigate some more explicit analysis to reduce the pessimism. Moreover, the experiments in this paper are produced by simulations. We will evaluate further using real implementations where the scheduling overhead will be investigated.

In our framework, the overruns are scheduled by a stated global First-In-First-Out (FIFO) queue. In our future work, we may utilize some other global scheduling mechanisms for task overruns. Then we can try to find out the optimal solutions under different system conditions. Moreover, we will also further investigate other approaches to select server bandwidth.

References

[1] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, “Resource kernels: A resource-centric approach to real-time and multimedia systems,” in *Photonics West’98 Electronic Imaging*. International Society for Optics and Photonics, 1997.

[2] M. Caccamo, G. Buttazzo, and L. Sha, “Handling execution overruns in hard real-time control systems,” *IEEE Transactions on Computers,*, vol. 51, no. 7, 2002.

[3] M. K. Gardner and J. W.-S. Liu, “Performance of algorithms for scheduling real-time systems with overrun and overload,” in *the 11th Euromicro Conference on Real-Time Systems, ECRTS’99*. IEEE, 1999.

[4] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, Jan 1973.

[5] J. Y.-T. Leung and J. Whitehead, “On the complexity of fixed-priority scheduling of periodic, real-time tasks,” *Performance evaluation*, vol. 2, no. 4, 1982.

[6] M. L. Dertouzos, “Control robotics: The procedural control of physical processes,” in *IFIP Congress, Proceedings.*, 1974.

[7] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Computing Surveys*, vol. 43, no. 4, 2011.

[8] J. H. Anderson, V. Bud, and U. C. Devi, “An edf-based scheduling algorithm for multiprocessor soft real-time systems,” in *17th Euromicro Conference on Real-Time Systems, ECRTS’05*, 2005.

[9] S. Kato and N. Yamasaki, “Semi-partitioned fixed-priority scheduling on multiprocessors,” in *15th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS’09*. IEEE, 2009.

[10] R. I. Davis and A. Burns, “Hierarchical fixed priority pre-emptive scheduling,” in *26th IEEE Real-Time Systems Symposium, RTSS’05*, 2005.

[11] P. J. Cuijpers and R. J. Bril, “Towards budgeting in real-time calculus: Deferrable servers,” in *Formal Modeling and Analysis of Timed Systems*. Springer, 2007.

[12] G. Lipari and E. Bini, “A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation,” in *31st IEEE Real-Time Systems Symposium, RTSS’10*, 2010.

[13] H. Zhu, S. Goddard, and M. B. Dwyer, “Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach,” in *32nd IEEE Real-Time Systems Symposium, RTSS’11*, 2011.

[14] L. Abeni and G. Buttazzo, “Integrating multimedia applications in hard real-time systems,” in *19th IEEE Real-Time Systems Symposium, RTSS’98*, 1998.

[15] J. Diaz, D. Garcia, K. Kim, C. Lee, L. Lo Bello, J. Lopez, S. L. Min, and O. Mirabella, “Stochastic analysis of periodic real-time systems,” in *23rd IEEE Real-Time Systems Symposium, RTSS’02*, 2002.

[16] R. I. Davis and A. Burns, “Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems,” in *30th IEEE Real-Time Systems Symposium, RTSS’09*, 2009.

[17] J. K. Strosnider, J. P. Lehoczky, and L. Sha, “The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments,” *IEEE Transactions on Computers*, vol. 44, no. 1, 1995.

[18] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, “A categorization of real-time multiprocessor scheduling problems and algorithms,” *Handbook on Scheduling Algorithms, Methods, and Models, pages*, 2004.

[19] M. Joseph and P. Pandya, “Finding response times in a real-time system,” in *Computer Journal 29(5)*, Oct. 1986.

[20] N. Audsley, A. Burns, M.Richardson, K. Tindell, and A. Wellings, “Applying new scheduling theory to static priority pre-emptive scheduling,” *Software Engineering Journal*, vol. 8, no. 5, 1993.

[21] J. W. S. W. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[22] J. 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.

[23] B. Sprunt, L. Sha, and J. Lehoczky, “Aperiodic task scheduling for hard-real-time systems,” *Real-Time Systems*, vol. 1, no. 1, 1989.

Appendix A

Proof of Lemma 2: We can divide r_i^B into two parts. The first part is the processing time in the period, where QW_i is just finished. The time interval t_{res} computed by Eq. 6 denotes the processing time of QW_i in its last replenishment period. Assume that τ_i^O is assigned to the server C_s^z ($1 \leq y < z \leq m$, C_s^z denotes the z^{th} SDS as well as its capacity.). Once τ_i^O has consumed the capacity of C_s^z, the processing time of τ_i^O will be increased to $C_s^z - t_{res}$. According to our scheduling policy, τ_i^O can preempt any other jobs in Q_g at this moment, since all the workloads ahead of τ_i^O have already left Q_g. Then τ_i^O will preempt the execution of another server C_s^y with a larger capacity, because under the worst-case assumption, all the servers with lower capacities should have exhausted their capacities. When the capacity of this server is also consumed, the processing time of τ_i^O will be increased to $C_s^z - t_{res} + C_s^y - C_s^z = C_s^y - t_{res}$. Then τ_i^O will occupy an even larger SDS, until the largest one (C_s^1) is consumed. Therefore, the first part of r_i^B can be computed by $C_s^1 - t_{res}$.

If $C_i^O \leq C_s^1 - t_{res}$, then $r_i^B = C_i^O$, because τ_i^O has already finished in the first replenishment period (i.e. the second part is 0).

If $C_i^O > C_s^1 - t_{res}$, we need to compute the second part of r_i^B which is the processing time after the first replenishment period. If τ_i^o is large enough, the remaining processing time (i.e. $C_i^O - (C_s^1 - t_{res})$) may occupy several complete replenishment periods (computed by Eq. 11) and a remaining execution time C_i^{res} (computed by Eq. 10). Due to the same manner of the above discussion, under the worst-case assumption, the minimum capacity that τ_i^O can consume within each replenishment period is C_s^1. Therefore, the denominator in Eq. 11 is C_s^1. Based on the above discussion, the second part of r_i^B can be calculated by $CRP_i^O \cdot T_s + C_i^{res}$.

While adding the above two parts together, we need to include the gap between them. That gap is the time interval between the time instant where all the SDS capacities have been consumed in the first replenishment period (i.e. the end of the first part), and the beginning of the second replenishment period (i.e. the start of the second part). This gap can be upper bounded as $T_s - C_s^1$. Therefore, when $C_i^O > C_s^1 - t_{res}$, we can compute r_i^B as

$$\begin{aligned} r_i^B &= C_s^1 - t_{res} + T_s - C_s^1 + CRP_i^O \cdot T_s + C_i^{res} \\ &= T_s - t_{res} + CRP_i^O \cdot T_s + C_i^{res} \end{aligned} \tag{17}$$

Appendix B

A Slack-Stealer based SDS Bandwidth Selection Scheme: For each processor, we apply Algorithm 2 to generate the capacity of its own SDS. If the processor is empty, the capacity of the processor will be fully assigned to the SDS (line 2-4). Otherwise, we need to assign the free capacity of this processor to the server, so that the created SDS will not impact too much on the schedulability of the regular executions of all the tasks on this processor.

Fig. 5 shows the way that we find the slack capacity of each processor. The basic idea is that we try to assign as much capacity as possible to an SDS, while keeping that the regular executions of all the tasks still meet their deadlines. In Fig. 5, the broken line represents the processor demand which is given by the following function

$$PD(\tau_i, t) = C_i^R + \sum_{\substack{\forall \tau_j \in hp(\tau_i) \\ \wedge \tau_j \in S_i}} \lceil \frac{t}{T_j} \rceil \cdot C_j^R \tag{18}$$

The straight solid line denotes the capacity supplied by a processor regarding time t. The first intersection point of these two lines provides the WCRT of τ_i^R, which is the theoretical base of the traditional response time analysis. If we assign some processor capacities to an SDS, the supplied capacities for the regular task executions will certainly be decreased. In other words, while assigning capacities to an SDS, we are shifting the straight line towards the downside gradually (i.e. represented by the dashed lines). We can keep shifting until the instant that a little more shifting space may result in no intersection point before D_i (i.e. τ_i^R is about to miss its deadline). This maximum shifting space provides the largest capacity of the SDS within the time duration D_i, while keeping τ_i^R still schedulable. However, if we fully utilize the above slack, it is possible that the response time τ_i becomes extremely close to its deadline. As a result, any overrun of τ_i will cause it to miss its deadline. On the other hand, if we decrease the server capacity to keep more slack, the response times of overruns may be increased accordingly. Therefore, in our experiments, we utilize the above slack with different percentages (denoted by Per^{C_s} hereinafter). This percentage is uniformly selected from the range of $[0.1, 0.6]$, which can provide the maximum schedulability according to our experimental observations. Note that for the empty processors (i.e. without any assigned regular task executions), we still fully utilize their capacities. Based on a given T_s, we can compute how much capacity can be assigned to each replenishment period (line 17). We need to apply the above process (line 6-17) on all the tasks in this processor. Then the minimum computed capacity per replenishment period is selected (line 18), so that all the regular task executions can still meet their deadlines.

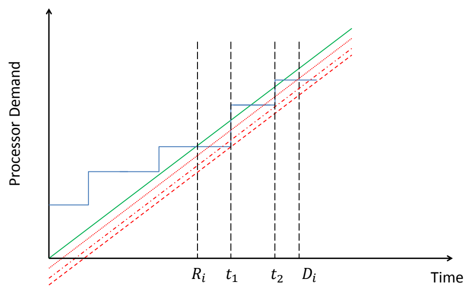

Fig. 5. SDS capacity selection.

Algorithm 2 SDS Capacity Selection

```
1: Initialize $Processor_k, T_s, C_k \leftarrow T_s, Per^{C_s}$
2: if no task on $Processor_k$ then
3:     return $C_k$
4: end if
5: for all $\tau_i$ in $Processor_k$ do
6:     $Lbound \leftarrow WCRT(\tau_i)$
7:     $Ubound \leftarrow D_i$
8:     $CheckPoints \leftarrow [\,]$
9:     $CheckPoints \Leftarrow D_i$
10:    for all $\tau_j$ in $Processor_k$ do
11:        $CheckPoints \Leftarrow ReleaseTimes(\tau_j, Lbound, Ubound)$
12:    end for
13:    $maxSlack \leftarrow 0$
14:    for all $pt$ in $CheckPoints$ do
15:        $maxSlack \leftarrow MAX(maxSlack, pt - PD(\tau_i, pt-1))$
16:    end for
17:    $maxCs \leftarrow \frac{maxSlack}{\lceil \frac{D_i}{T_s} \rceil + 1} * Per^{C_s}$
18:    $C_k \leftarrow MIN(C_k, maxCs)$
19: end for
20: return $C_k$
```