Response Time Analysis for Fixed Priority Servers

Arne Hamann, Dakshina Dasari, Jorge Martinez, Dirk Ziegenbein
Robert Bosch GmbH, Germany
{arne.hamann,dakshina.dasari,jorgeluis.martinezgarcia,dirk.ziegenbein}@de.bosch.com

ABSTRACT

In the automotive domain, existing scheduling primitives do not suffice for complex integration scenarios involving heterogeneous applications with diverse timing requirements. Thus, there is a renewed interest to establish server based scheduling as a mainstream scheduling paradigm in automotive software development and to that end, we provide a response time analysis that provides tighter bounds than the existing state-of-the-art approaches for a system containing deferrable and periodic servers. Our proposed approach can also analyze periodic tasks with offsets and tasks with backlogged activations. We further perform experiments on LITMUS^{RT} in order to demonstrate the tightness of the proposed response time analysis.

1 INTRODUCTION

An important design trend in the automotive domain is that disparate functions, formerly hosted on dedicated electronic control units (ECUs) featuring simple microcontrollers, are now being consolidated into fewer more powerful domain-control units. Furthermore, emerging applications like advanced driver-assistance systems (ADAS) and autonomous driving applications may co-exist with classical control applications on the same hardware platform. Consequently, an imminent design challenge is to integrate and co-execute these heterogeneous applications with diverse timing and safety requirements on the shared platform, while efficiently using the platform's resources as well as providing temporal and spatial isolation to individual applications.

An important use-case scenario is the consolidation and integration of existing (legacy) applications. Each such application is realized using an OSEK compliant RTOS and comprises several periodic tasks. This is the main motivation why we focus in this paper on periodic task sets with offsets that are scheduled inside reservations.

Currently, automotive applications are scheduled using well established mechanisms like fixed priority preemptive (FPP) task scheduling (as in the AUTOSAR OS), time division multiplexing (TDM) or via custom table-driven schedulers. However, considering the upcoming complex integration scenarios as well as the heterogeneous nature of applications that involves more dynamic workloads, as well as safety and predictability requirements, existing mechanisms do not suffice, in that they are not inherently designed with the holistic view to monitor and manage the resource usage of applications for ensuring temporal isolation. Reservation based scheduling (RBS) is a well understood mechanism to compose applications in an efficient and constructively correct manner, since it combines scheduling together with budget enforcements to meet the requirements of temporal isolation. Hence there is a renewed interest in exploring RBS as the scheduling paradigm for integrating use-cases, both at operating-system and hypervisor level, in the context of the aforementioned domain-centralized control units.

RBS is essentially hierarchical in nature. At the top level, a set of "servers" are scheduled by a global scheduling mechanism, and then internally, each server hosts and schedules a set of child tasks. A server is therefore an abstract scheduling container, but with the additional property that it is associated with a budget which is replenished according to a replenishment mechanism (for example periodically). Tasks can only execute using the capacity of their parent server. Once the budget of the server is consumed, the server (and thereby the executing task) is suspended. Thus, by design, in a system composed of different servers, the interference to tasks within one server from other servers is bounded and secondly, minimum temporal guarantees are enforced.

While component-based design is quite commonly adopted by system designers, the associated component-based scheduling provided by RBS has not gained comparable traction in the industry. We found that a key barrier to directly adopting existing techniques is that computed response times are too pessimistic, even for simple scenarios involving purely periodic tasks, when compared to the values observed at runtime. In the automotive domain in particular, due to the typically high system utilization required for efficiency, this pessimism becomes prohibitive. Moreover, existing analyses generally assume that a system integrator does not have knowledge of all the tasks in each of the components/servers. While this is valid, we also need approaches where we can exploit the internal information within servers to derive tighter response times. In case of deferrable servers this "white box" view is even indispensable to calculate correct worst-case response times.

2 BACKGROUND AND RELATED WORK

2.1 Fixed Priority Servers

We consider FPP servers like the Deferrable and Periodic server [16] in which the contained tasks are scheduled according to fixed priorities. FPP servers are promising candidates for practical deployments given their simple implementation and given that they are well understood. The *Periodic Server* is replenished periodically to its full budget and when invoked, serves all requests that arrive before its capacity is exhausted (up to the remaining capacity from their time of arrival). The capacity of an active (scheduled to execute) periodic server without ready tasks diminishes as time elapses and any request that arrives after its capacity is exhausted, is served in the next replenishment period. The *Deferrable Server (DS)* is replenished periodically like the periodic server, but it is bandwidth-preserving, i.e. it retains any unused capacity C_s till the end of its period, thus deferring its execution. However, this property of the DS can lead to the infamous *double hit effect*, where the server may contiguously execute for a period of time equalling twice its capacity (C_s at the end of one period and immediately another C_s time units at the beginning of the next replenishment period), and thereby

RTNS '18, October 10–12, 2018, Chasseneuil-du-Poitou, France

ACM ISBN 978-1-4503-6463-8/18/10…$15.00
https://doi.org/10.1145/3273905.3273927

impose a greater interference on a lower priority server/task than an equivalent periodic task.

In order to dimension these servers in a way for the contained tasks to meet their deadlines, as well as to ensure efficient system utilization, a tight response time analysis of the enclosed tasks is necessary. If a server could provide resources continuously to its child tasks, a regular response time analysis as in [10] would be sufficient. However since a server behaves more like a processor with fractional capacity, the above analysis is not directly applicable given that the execution resource is available in a non-contiguous manner i.e., C_s execution units may be availed every T_s time units.

2.2 Earlier Work on Server-Based Scheduling

Bernat and Burns [3] carried out a review of fixed priority servers and compared them on the basis of metrics like responsiveness, utilization and discussed the impact of server parameter selection. A common approach to compute worst-case response times (WCRT) in a hierarchical systems is to determine the worst-case response time by constructing a worst-case arrival pattern or a *critical instant*. For example in [14], the authors define that the critical instant for a task occurs when it arrives such that the server's capacity is depleted by lower priority tasks and its request and that of higher priority tasks of the same server occur exactly C_s time units after the replenishment. Lipari and Bini [12], used a server availability function in order to determine the WCRT of a task by making the same assumption as in [14]. Both methods are pessimistic because they assume that the budget is always available as late as possible in each replenishment interval. In [14] task periods are even constrained to be greater or equal to server periods which restricts the applicability of their analysis. Almeida [1] further improved the work of [12] by considering the maximum latency that a server could suffer and a periodic pattern that resembles the critical instant in [14] and [12]. Davis and Burns [8], based on the work of [1], made use of the busy-window analysis to provide tighter bounds by refining the critical instant scenario. All the above works assumed that task are indepenent. Later in [9], Burns et.al extended their work to deal with resource sharing. However, we show that their approach is pessimistic for the analysis of a system with a single server and the pessimism increases with the number of servers. Shin et al [15] introduced a compositional method that derives either the capacity or period, from the timing requirements of its child tasks so that the server is schedulable, if and only if its tasks are. It also assumes one of the server parameters is known apriori.

In the intra-server case, the common reason for the pessimism of the aforementioned methods is that none of them verify if the critical instant indeed occurs and subsequently, this pessimism propagates to inter-server analysis. Another common shortcoming of all approaches that extend the busy window analysis is the assumption that all servers will indeed get the guaranteed C_s time units in T_s time units. For deferrable servers, this can only be ensured by considering all task interleavings up to the system wide hyperperiod to determine if (coinciding) double hits occur. These analyses are thereby based on the assumption that servers' budgets are guaranteed, which is not trivial to ascertain.

Like real-time calculus (RTC) [17], our approach is based on the demand/service abstraction. However the major difference is that, while RTC considers the demand/service curve in the time-interval domain (any window of interval length t) and so loses precise information about task offsets and inter-job dependencies [13], we consider the demand/service curve in the time domain (in the interval $[0, t]$). A time-domain-based formalism is inevitable so as to obtain a precise response time analysis of reservation-based scheduling that can deal with offsets and backlogged tasks.

The above-mentioned analyses do not consider tasks with offsets. Offset assignment has been adopted to reduce the output jitter of a task, establish precedence constraints and shorten worst-case response times [2]. The aforementioned approaches also do not handle backlogged tasks. A task τ_i is said to be backlogged if its pending workload delays the execution of one of its jobs, so that multiple instances of the same task can coexist, referred to as *multiple task activations* in AUTOSAR [7]. Consequently, a backlogged-task- as well as an offset-aware response time analysis of hierarchical systems is of paramount importance to analyze real systems.

2.3 Problem Statement and Contributions

Given a system consisting of multiple tasks which contained within periodic or deferrable servers, wherein each task is periodic and may have a non-zero offset, and the servers are scheduled using a fixed priority preemptive (FPP) scheduling mechanism while tasks within servers are also scheduled with a FPP mechanism, determine the worst-case response time (WCRT) of any given task. With this work, we make the following contributions.

(1) We highlight the shortcomings in the existing analysis and hence motivate the need for a different approach.
(2) We propose a novel approach for response time analysis based on the demand/supply curve abstraction. We advance the state of the art (SoA) by considering periodic tasks with offsets and allowing backlogged tasks.
(3) We show via experiments that the proposed approach dominates the SoA in terms of tightness of the computed bounds.
(4) We also evaluate the tightness of our proposed approach against experiments conducted on LITMUS^{RT} [5, 6].

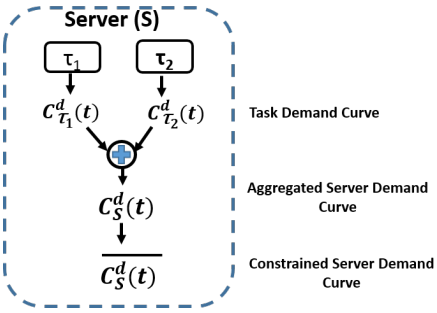


Figure 1: The total workload demand of tasks in a server is abstracted by a constrained demand curve.

3 SYSTEM MODEL AND OVERVIEW

In this work, we consider the analysis of a simple two-level hierarchical system consisting of deferrable and periodic servers. We assume that the servers are first scheduled by a FPP algorithm and then tasks within servers are also scheduled using a FPP mechanism. In our analysis, we assume tasks are independent. We also assume that the system designer has information regarding the tasks in each of the servers.

A server S is characterized by its budget C_s and its replenishment period T_s and consists of a set τ of periodic tasks $\{\tau = \tau_1, \tau_2 \ldots \tau_n\}$.

We model each task τ_i with offset O_i, worst-case execution time C_i and period P_i. The $q-th$ job of τ_i is denoted by $\tau_{i,q}$ and its release time $a_{i,q}$ is computed as $a_{i,q} = O_i + (q-1)\cdot P_i$ for all $q \geq 1$. Multiple instances of a task could co-exist (backlogged instances are possible).

The approach is based on the demand/service abstraction and the analysis is carried out in different stages:

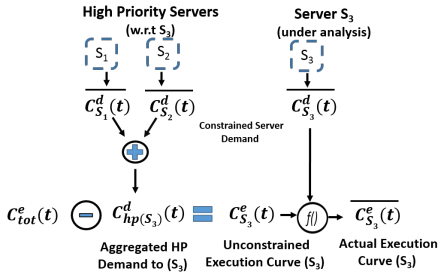


Figure 2: Overview of the analysis

(1) Model task demand: We first model the execution requirement (demand) of each task in a server.
(2) Model server demand: Then for every server, we aggregate the demand of all its constituent tasks. Given that a server S cannot provide more than C_s time units in an interval equal to T_s, we shape the demand by deferring any extra task request in any replenishment interval (RI) to the next RI. We call this the "constrained demand" of the server. Figure 1 illustrates these steps for a given server and the concept is described in Section 5.
(3) Model available server supply: In a fixed priority preemptive set-up, a server under analysis can only execute when the higher priority servers are not executing. Consider Figure 2, in which server S_3 is under analysis. We compute the aggregated constrained demand of the higher priority servers S_1 and S_2. The intervals where S_3 executes is computed by "deducting" this aggregated higher priority demand from the total available execution supply to obtain the "unconstrained execution curve" of the server.
(4) Determine the Actual Execution Curve: However the server cannot effectively execute in all of these "unconstrained execution curve" intervals due to the budget constraints as well as its internal workload demand pattern. Therefore we derive the actual execution curve where the server feasibly executes from the unconstrained execution curve by factoring in its budget constraints and its own constrained server demand (Section 6).
(5) Determine the response time of tasks within the server: After determining the intervals where the server will execute, we then distribute these intervals amongst the constituent tasks of the server on a priority basis while considering the task demand curves and thereafter compute the response time of the task (Section 7).

4 BASIC DEFINITIONS AND OPERATIONS

We now introduce windows and curves which form the building blocks of the analysis and are used to model execution intervals.

Definition 1. *Window: A window W_p has a start time $W_p.s$ and an end time $W_p.e$ such that $W_p.s < W_p.e$ and its length $W_p.l$ is computed as $W_p.l = W_p.e - W_p.s$. Any window with $W_p.e \leq W_p.s$ is empty. We also denote a window W_p by a more descriptive tuple $W(W_p.s, W_p.e)$. Moreover, two windows are said to be equal if they have the same start and end time.*

Definition 2. *Overlap between windows: The existence of overlap between two windows W_p and W_q is decided as follows:*

$$W_p \Omega W_q = \begin{cases} \text{false} & \text{if } W_p.e < W_q.s || W_q.e < W_p.s \\ \text{true} & \text{otherwise} \end{cases} \quad (1)$$

Definition 3. *Curve: A curve of length t, $C_p(t)$, is modelled by a set of non overlapping windows $C_p(t) = \{W_{p,1}, \ldots, W_{p,n} | W_{p,n}.e \leq t\}$ ordered by their start times and its total capacity, $|C_p(t)|$, is the sum of its constituent window lengths, i.e. $|C_p(t)| = \sum_{j=1}^{n} W_{p,j}.l$.*

Definition 4. *Aggregation of two windows: The aggregation operation over a pair of overlapping windows W_p and W_q is given by $W_p \oplus W_q$ and is computed as follows: $W_p \oplus W_q = W(x, y)$, where $x = \min(W_p.s, W_q.s)$ and $y = \min(W_p.s, W_q.s) + W_p.l + W_q.l$.*

Example 1. *Consider windows $W_1(2, 4)$ and $W_2(3, 6)$ with lengths 2 and 3. Then $W_1 \oplus W_2$ returns a new window $W_3(2, 7)$ with length 5.*

The aggregation operation ensures that there are no two windows with the same start, since it merges them. In the case of overlapping or adjacent windows, it creates a new window whose execution length is the sum of the window lengths.

Note that in the rest of the document we use the operator $A \cup B$ to describe a special set union between these sets implying that we add all the members of the two sets including *duplicates*. We now define the operation to aggregate two curves to arrive at a set of windows with no overlaps.

Definition 5. *Aggregation of two curves: Given two curves $C_p(t)$ and $C_q(t)$ the aggregated curve is given by $C_k(t) = C_p(t) \oplus C_q(t)$, where $C_k(t)$ is defined iteratively starting from the initial ordered set*

$$C_k(t) = C_p(t) \cup C_q(t)$$

using the following rule until $\nexists (W_a, W_b) \in C_k(t) | W_a \Omega W_b = \text{true}$.

$$\forall W_a, W_b \in C_k(t) \text{ with } W_a \Omega W_b = \text{true} :$$
$$C_k(t) = C_k(t) \setminus \{W_a, W_b\} \cup \{W_a \oplus W_b\}$$

This operation first considers all windows in both curves including the duplicates and starts by aggregating each pair of overlapping windows until no overlapping windows remain. We aggregate each overlapping window pair, add the aggregated window(s) to the resulting set, while discarding the original windows of the pair.

Example 2. *Consider the curves $C_1(t) = \{(0, 4)\}$ and $C_2(t) = \{(0, 1), (5, 6), (10, 11)\}$ for $t = 11$. We start with the curve $C_3(t) = C_1(t) \cup C_2(t) = \{(0, 1), (0, 4), (5, 6), (10, 11)\}$, and replace overlapping windows, $(0, 1)$ and $(0, 4)$, by $(0, 1) \oplus (0, 4) = (0, 5)$, reducing $C_3(t)$ to $\{(0, 5), (5, 6), (10, 11)\}$. Replacing the next overlapping elements of the curve, $(0, 5)$ and $(5, 6)$, by $(0, 5) \oplus (5, 6) = (0, 6)$, yields $C_3(t) = C_1(t) \oplus C_2(t) = \{(0, 6), (10, 11)\}$.*

Next we define the delta operation which tries to fit a demand window W_q into a supply window W_p. A request of W_q starting at $W_q.s$ can only be serviced by W_p if either there is an overlap or the supply window appears later ($W_q.s < W_p.e$). When the supply window W_p starts later than the demand window W_q as in Figure 3b, the start of W_q is conceptually moved to the right to overlap with W_p. If W_p ends earlier than the start of W_q, the demand window cannot be fitted in, and the overlap is an empty window.

(a) W_q**'s demand partially satisfied by** W_p **with remaining demand** R_q

(b) W_q**'s demand fully satisfied by** W_p **with remaining supply** R_p

(c) W_q **completely fits into** W_p **with two remaining segments** R_p

(d) W_q **partially fits into** W_p **with remaining segment** R_q

Figure 3: Examples of delta window operator (Definition 6)

Definition 6. *The delta between two windows W_p and W_q denoted by $W_p \ominus W_q$ returns a 3-tuple (R_p, W_o, R_q), where R_p denotes the remaining unused supply of W_p and W_o denotes the overlap between W_p and W_q. Here R_q denotes the unsatisfied demand of W_q.*

(1) If the supply window arrives earlier: $W_p.e \leq W_q.s$

$$(R_p, W_o, R_q) = (W_p, \emptyset, W_q)$$

(2) else if $W_q.s < W_p.e$

$$\begin{aligned} W_o.s &= \max(W_p.s, W_q.s) \\ W_o.e &= W_o.s + \min(W_q.l, W_p.e - W_o.s) \\ W_o &= W(W_o.s, W_o.e) \\ R_q &= W(W_q.s + W_o.l, W_q.e) \\ R_p &= W(W_p.s, W_o.s) \\ R_p &= R_p \cup W(W_o.e, W_p.e) \end{aligned}$$

Figure 3 shows examples of how the delta window operator $\ominus$ fits the window W_q into W_p. Figure 3a shows a partial overlap window W_o, leaving remaining segments of, both, windows W_p and W_q. Figures 3b and 3d show cases where the overlap window W_o is deferred to the start time of W_p. In Figure 3b W_q completely fits into W_p, whereas in Figure 3d W_p is completely consumed resulting in a remaining segment of W_q. In Figure 3c W_p overspans W_q. As a result, W_q completely fits into W_p leaving 2 remaining segments of W_p.

Similarly the delta operation can be extended to two curves, in which we fit all windows in $C_q(t)$ into $C_p(t)$.

Definition 7. *Delta curve: Given two curves $C_p(t) = \{W_{p,1}, W_{p,2}, \ldots\}$ and $C_q(t) = \{W_{q,1}, W_{q,2}, \ldots\}$ the delta curve denoted as*

$$(C'_p(t), C_o(t), C'_q(t)) = C_p(t) \ominus C_q(t)$$

is defined iteratively starting from the initial curves

$$C'_p(t) = C_p(t) \text{ and } C'_q(t) = C_q(t) \text{ and } C'_o(t) = \emptyset$$

using the following rules until $C'_p(t) = \emptyset \vee C'_q(t) = \emptyset$:

(1) Find the index of first valid supply window: $j = \min(i) : W_{q,1}.s < W_{p,i}.e$
(2) Perform the delta operation: $(R_p, W_o, R_q) = W_{p,j} \ominus W_{q,1}$
(3) Remove the (partially) used supply window from the supply and add the remaining window segment: $C'_p(t) = C'_p(t) \setminus W_{p,j} \cup R_p$
(4) Add the overlap windows: $C_o(t) = C_o(t) \cup W_o$
(5) Remove the completed demand and add any incomplete demand segment: $C'_q(t) = C'_q(t) \setminus W_{q,1} \cup R_q$

Property 1. *The delta curve operation $C_p(t) \ominus C_q(t)$ constructively fits window $W_{q,i} \in C_q(t)$ into the earliest available window of the curve $C_p(t)$ (see Step 1 of Definition 7) to obtain the overlap curve $C_o(t)$.*

Property 2. *The remaining service $C'_p(t)$ has no overlap with $C_o(t)$ (see Step 3 of Definition 7).*

Example 3. *Consider the curves $C_p(40) = \{(0, 5), (12, 15), (22, 24), (30, 35)\}$ and $C_q(40) = \{(2, 4), (14, 17), (22, 24)\}$. The overlap $C_o(40) = \{(2, 4), (14, 15), (22, 24), (30, 32)\}$. The remaining segments of $C_p(t)$ once $C_q(t)$ is subtracted, $C'_p(t) = \{(0, 2), (4, 5), (12, 14), (32, 35)\}$. Note that $C'_q(t) = \emptyset$, since $C_q(t)$ can be completely fit into $C_p(t)$.*

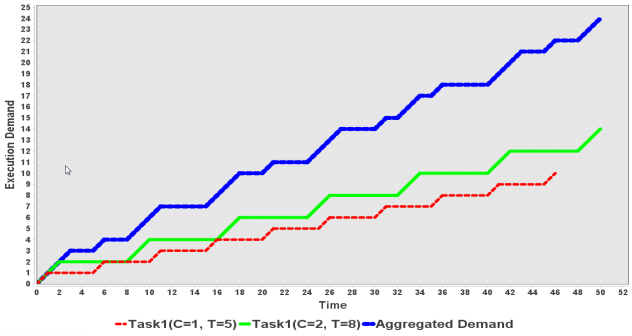


Figure 4: The aggregated demand of the two tasks in server S is $C^d_S(t) = \{(0, 3), (5, 6), (8, 11), (15, 18) \ldots\}$.

Since all the budget accounting in a server S is done on replenishment interval boundaries, we define an operation that groups all windows within each RI into one group. If a window overspans a replenishment boundary, the window is split.

Definition 8. *Split curve: Given a curve $C_p(t) = \{W_{p,1}, \ldots, W_{p,n}\}$ and a replenishment interval T_s, the split operation on the curve $C_p(t)$ distributes all its windows into disjoint groups, $\{\mathcal{G}_k\}$, producing a new curve $\tilde{C}_p(t)$, such that $\tilde{C}_p(t) = \bigcup \mathcal{G}_k$ and any window in group $\mathcal{G}_k$ always lies between $[k \cdot T_s, (k+1) \cdot T_s)$. Every window W_p in $C_p(t)$ is split by defining $k = \lfloor W_{p,i}.s / T_s \rfloor$ and using the rules:*

(1) if $W_{p,i}.e \leq (k+1) \cdot T_s$ then $W_{p,i} \subset \mathcal{G}_k$.
(2) else $W(W_{p,i}.s, (k+1) \cdot T_s) \subset \mathcal{G}_k$
Start over by splitting $W((k+1) \cdot T_s, W_{p,i}.e)$

Example 4. *Given a curve $C_p(t) = \{(2, 4), (5, 6), (7, 23), (24, 26)\}$, and $T_s = 10$, then $\mathcal{G}_0 = \{(2, 4), (5, 6), (7, 10)\}$, $\mathcal{G}_1 = \{(10, 20)\}$ and $\mathcal{G}_2 = \{(20, 23), (24, 26)\}$. Thus $\tilde{C}_p(t) = \mathcal{G}_0 \cup \mathcal{G}_1 \cup \mathcal{G}_2$.*

5 MODELLING THE EXECUTION DEMAND OF THE TASKS IN A SERVER

Definition 9. *Execution Window of a Task: An execution window of τ_i denoted by $W_{\tau_i,j}$ represents the time interval in which a job j of τ_i could execute in isolation, i.e., in the absence of server constraints and other interference.*

Definition 10. *Demand Curve of a task: The demand curve of a task τ_i up to time t denoted by $C^d_{\tau_i}(t)$, is modelled by a set of non overlapping execution windows $C^d_{\tau_i}(t) = \{W_{\tau_i,1} \ldots W_{\tau_i,n} | W_{\tau_i,n}.e \leq t\}$, ordered by their start times.*

Next, we compute the aggregated demand curve of a server up to time t. A naive way is to simply add up the demand of two tasks up to a given time t. However this can result in overestimating the demand. Hence the aggregation operation avoids this by unrolling the total demand (in case of overlaps) over the time interval by applying the aggregation operator, thus providing tight estimates.

Definition 11. *Aggregated demand curve (ADC): For a server S with associated tasks $\tau_1 \dots \tau_m$ the ADC, $C_S^d(t)$, is defined as follows:*

$$C_S^d(t) = C_{\tau_1}^d(t) \oplus \dots \oplus C_{\tau_m}^d(t)$$

where $C_{\tau_i}^d(t)$ is the demand curve of task τ_i up to time t, and $C_{\tau_i}^d(t) \oplus C_{\tau_j}^d(t)$ is computed as per Definition 5.

Example 5. *Consider server S with tasks $\tau_2(C_2 = 1, T_2 = 5)$ with task demand curve $C_{\tau_2}^d(t) = \{(0, 1), (5, 6), (10, 11), (15, 16), \dots\}$ and task $\tau_3(C_2 = 2, T_2 = 8)$ with $C_{\tau_3}^d(t) = \{(0, 2), (8, 10), (16, 18), \dots\}$. Then $C_S^d(t) = C_{\tau_2}^d(t) \oplus C_{\tau_3}^d(t) = \{(0, 3), (5, 6), (8, 11), (15, 18), \dots\}$.*

5.1 Modelling the Constrained Server Demand

As the budget accounting in a periodic and deferrable server takes place at replenishment intervals equal to T_s, we apply the split operator to the aggregated demand curve to form demand window groups $\mathcal{G}_k$. The demand in a given group is the sum of its constituent window lengths. We constrain each demand window group $\mathcal{G}_k$ such that the overall demand in each group does not exceed the server budget C_s. Any demand exceeding C_s in a given group $\mathcal{G}_k$ is moved to the next group and we proceed to constrain the demand in each group subsequently. Algorithm 1 describes the steps.

5.2 Server Specific Window Selection

We select the feasible windows $W_{k,0}$ to $W_{k,i}$ and the partial window W_q from $\mathcal{G}_k$ that fit into C_s, depending on the server type.

Deferrable Servers. A deferrable server retains its budget till the end of the replenishment interval. We therefore select all the windows and the partial window that add upto C_s. Algorithm 2 describes how the feasible windows in each replenishment interval are calculated for a DS.

Example 6. *Consider the aggregated curve $C_S^d(t) = \{(0, 3), (5, 6), (8, 11), (15, 18)\}$ of a DS ($C_s = 2$, $T_s = 4$) as computed in Example 5. We need to constrain the budget in each replenishment interval $[k \cdot 4, (k + 1) \cdot 4]$ to C_s for $k \geq 0$. We first split the curve by T_s to obtain the groups $\mathcal{G}_0 = \{(0, 3)\}$, $\mathcal{G}_1 = \{(5, 6)\}$, $\mathcal{G}_2 = \{(8, 11)\}$, $\mathcal{G}_3 = \{(15, 16)\}$, and $\mathcal{G}_4 = \{(16, 18)\}$. We start with $\mathcal{G}_0$. Since the cumulative demand in $\mathcal{G}_0 = 3$ is greater than $C_s = 2$, we form the constrained group $\overline{\mathcal{G}_0} = \{(0, 2)\}$, shift the extra demand of 1 into a new window $W(4, 5)$, and move it to the next group $\mathcal{G}_1$. So that $\mathcal{G}_1 = \{(4, 5), (5, 6)\}$. Since the demand in $\mathcal{G}_1 = 2 \leq C_s$, we do not constrain it and $\overline{\mathcal{G}_1} = \mathcal{G}_1$. In the same fashion, we keep constraining the demand in each group of $C_S^d(t)$ to obtain $\overline{C_S^d(t)} = \overline{\mathcal{G}_1} \cup \dots \cup \overline{\mathcal{G}_4} = \{(0, 2), (4, 6), (8, 10), (12, 13), (15, 16), (16, 18)\}$.*

Periodic Servers. For a periodic server, its budget is only available in the interval $[k \cdot T_s, k \cdot T_s + C_s]$, where $k \in \mathbb{N}$. Any request arriving after $k \cdot T_s + C_s$ is served in the next replenishment interval. Algorithm 3 describes how the feasible windows in each replenishment interval are calculated for a PS.

Theorem 5.1. *The constrained demand curve $\overline{C_S^d(t)}$ computed as per Algorithm 1 redistributes the demand across replenishment periods in compliance with the budget constraints of deferrable and periodic servers.*

Algorithm 1: Computing the constrained server demand

Input: The aggregated demand curve $C_S^d(t)$

Output: The constrained demand curve $\overline{C_S^d(t)}$

(1) Form demand window groups by performing the split operation on the aggregated demand curve $C_S^d(t)$ as per Definition 8.

$$\{\mathcal{G}_0, \mathcal{G}_1, \dots \mathcal{G}_n\} = \tilde{C}_S^d(t)$$

Any window in $\mathcal{G}_k = \{W_{k,0}, \dots W_{k,m_k}\}$ lies between $[k \cdot T_s, (k + 1) \cdot T_s)$.

(2) For each group $\mathcal{G}_k \in \{\mathcal{G}_0, \mathcal{G}_1, \dots \mathcal{G}_n\}$

(a) Retain all feasible complete windows $W_{k,0}, \dots, W_{k,i}$ and possibly a feasible window segment of $W_{k,i+1}$ denoted by W_q. The remaining partial window of $W_{k,i+1}$ is denoted by W'_q.

$$(i, W_q, W'_q) = PartitionWindows(\mathcal{G}_k, ServerType)$$

$\overline{\mathcal{G}_k}$ includes all feasible windows of $\mathcal{G}_k$ and is given by:

$$\overline{\mathcal{G}_k} = \{W_{k,0}, \dots W_{k,i}\} \cup W_q.$$

(b) The set of remaining (infeasible) windows of $\mathcal{G}_k$ not covered by $\overline{\mathcal{G}_k}$ is $\{W'_q, W_{k,i+2}, \dots W_{k,m_k}\}$ with demand Δ_k computed as:

$$\Delta_k = W'_q.l + \sum_{p=i+2}^{m_k} W_{k,p}.l$$

This demand of Δ_k is added as a new execution window at the head of the next group $\mathcal{G}_{k+1}$:

$$\mathcal{G}_{k+1} = \mathcal{G}_{k+1} \cup W((k + 1)T_s, (k + 1)T_s + \Delta_k)$$

(3) Let $v = \lfloor t/T_s \rfloor$. Then the constrained demand curve is

$$\overline{C_S^d(t)} = \overline{\mathcal{G}_0} \cup \dots \cup \overline{\mathcal{G}_v}$$

return $\overline{C_S^d(t)}$

Algorithm 2: PartitionWindows

Input: $\mathcal{G}_k = \{W_{k,0}, \dots W_{k,m_k}\}$, ServerType=DS

Output: i, W_q, W'_q

(1) Find maximum index i such that the total consecutive demand of $\{W_{k,0} \dots W_{k,i}\} \leq C_s$

$$i = \max(j) \text{ with } \sum_{p=1}^{j} W_{k,p}.l \leq C_s.$$

(2) If $b = C_s - \sum_{p=1}^{i} W_{k,p}.l > 0$ and $i < m_k$ a partial window of $W_{k,i+1}$ is needed to fill up the remaining admissible budget:

$$W_q = W(W_{k,i}.s, W_{k,i}.s + b)$$

$$W'_q = W(W_{k,i}.s + b, W_{k,i}.e)$$

return (i, W_q, W'_q)

Proof. For a DS, as per Algo. 2, the cumulative demand of all windows in $\overline{\mathcal{G}_k}$ in any replenishment interval $[k*T_s, (k+1)*T_s)$ for all k, is always restricted to C_s. Likewise, for the PS as per Algo. 3,

Algorithm 3: PartitionWindows

Input: $\mathcal{G}_k = \{W_{k,0}, \ldots W_{k,m_k}\}$, ServerType=PS
Output: i, W_q, W'_q

(1) Find the maximum index i, such that each window in $\{W_{k,0} \ldots W_{k,i}\}$ ends earlier than $k \cdot T_s + C_s$

$$i = \max(j) \text{ with } W_{k,j}.e < k \cdot T_s + C_s$$

(2) If there exists $W_{k,i+1}$ such that $W_{k,i+1}.s < k \cdot T_s + C_s < W_{k,i+1}.e$, split it to obtain

$$W_q = W(W_{k,i+1}.s, k \cdot T_s + C_s)$$
$$W'_q = W(k \cdot T_s + C_s, W_{k,i+1}.e).$$

return (i, W_q, W'_q)

only those windows (and segments) that complete before $k * T_s + C_s$ are retained in $\overline{\mathcal{G}_k}$. Hence the proof holds by construction. □

Lemma 5.2. *The constrained demand curve $\overline{C^d_S(t)}$ represents the intervals in which a server executing in isolation would serve its aggregated demand (AD).*

Proof. The AD curve models the execution demand of all tasks inside a server in a given time interval. In the absence of external interference, this AD is immediately served subject to the server budget constraints in the windows defined by $\overline{C^d_S(t)}$ as per Theorem 5.1. Hence the windows in $\overline{C^d_S(t)}$ represent the time intervals in which the server executes in isolation. □

6 SERVER EXECUTION CURVES

In a system of servers scheduled by a fixed priority mechanism, a given server can only execute in the "gaps" left by high priority server execution windows, while conforming to its own budget restrictions and also considering its internal workload demand. The unconstrained execution curve of server up to time t is defined as the total remaining service after deducting constrained execution demands of higher priority servers.

6.1 Unconstrained Server Execution Curve

Consider a set of m servers $\mathcal{S} = S_1, S_2 \ldots S_m$, where the servers are indexed by their priorities and S_1 has the highest priority.

Definition 12. *Aggregated High Priority Demand Curve: For a server S_i which can be interfered by higher priority servers $S_1 \ldots S_{i-1}$, the aggregated high priority demand curve $C^d_{hp(S_i)}(t)$ is defined as :*

$$C^d_{hp(S_i)}(t) = \overline{C^d_{S_1}(t)} \oplus \ldots \oplus \overline{C^d_{S_{i-1}}(t)}$$

where the constrained server demand $\overline{C^d_{S_j}(t)}$ follows Algorithm 1.

Definition 13. *Unconstrained Execution Curve: For a server S which has an aggregated high priority demand curve $C^d_{hp(S)}(t)$, we define its unconstrained execution curve $C^e_S(t)$ as follows:*

$$(C^e_S(t), -, -) = C^e_{tot}(t) \ominus C^d_{hp(S)}(t)$$

where $C^e_{tot}(t)$ represents the total unconstrained execution curve of the entire system, i.e. $C^e_{tot}(t) = \{W(0,t)\}$ Thus $C^e_S(t)$ represents the remaining curve of $C^e_{tot}(t)$ when $C^d_{hp(S)}(t)$ is deducted from it.

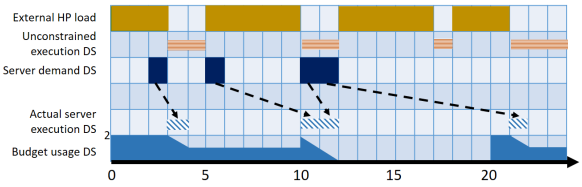


Figure 5: Computation of the Actual Server Execution Curve for DS(2,10). Note how the unconstrained execution window (17, 18) cannot be used to serve the demand (11, 12) since the server is out of budget.

Property 3. *The unconstrained execution curve $C^e_S(t)$, contains the exclusive windows in which server S may execute, without overlapping with execution windows of higher priority servers. This directly follows from Property 2 of the delta curve operator.*

Example 7. *Consider Figure 6 with aggregated HP demand $C^d_{hp(S)}(t) = \{(0,1),(4,5),(8,9),(12,13)\}$ for $t = 16$. Then $C^e_{tot}(t) = W(0,16)$, and $C^e_S(t) = \{(1,4),(5,8),(9,12),(13,16)\}$.*

In order to determine if server S guarantees its budget C_s every T_s time units, we step through all the windows in each replenishment interval up to the system wide hyperperiod (defined later) and check if the total budget, given by the sum of lengths of the windows in each interval, is at least C_s. If not, then the server cannot guarantee its budget. Such a check is especially needed when deferrable servers are used.

6.2 Actual Server Execution Curve

All the execution slots of the unconstrained execution curve $C^e_S(t)$ cannot be used by the server S since it constrained by its budget. Furthermore, the windows in which S executes are also constrained by the arrival times and demand of the internal tasks within the server. We first apply the split operation on the unconstrained execution curve $C^e_S(t)$ to obtain $\tilde{C}^e_S(t)$ so that no window in $\tilde{C}^e_S(t)$ spans across replenishment intervals. We then fit the server demand windows from $\overline{C^d_S(t)}$ into the earliest available execution windows.

To compute the server execution curve $\overline{C^e_S(t)}$, we select windows from $\tilde{C}^e_S(t)$ that can satisfy the constrained server demand $\overline{C^d_S(t)}$ while ensuring that the total consumed budget in any group in the resulting curve, $\overline{C^e_S(t)}$, does not exceed C_s.

The group index of a window W_p is given by $k = \lfloor W_p.s/T_s \rfloor$. To simplify notation, let the budget of the group in which W_p falls, be denoted by $B(W_p)$. When a supply window W_p is selected for execution, we check if its group budget limits are not violated, i.e. if $B(W_p) < C_s$. Algo. 4 describes the method of computing $\overline{C^e_S(t)}$.

Example 8. *Consider a deferrable server $S(C_s = 2, T_s = 10)$ as shown in Figure 5 with split unconstrained execution (supply) curve $\tilde{C}^e_S(t) = \{(3,5),(10,12),(17,18),(21,24)\}$ and constrained server demand curve $\overline{C^d_S(t)} = \{(2,3),(5,6),(10,12)\}$. Initially $\overline{C^e_S(t)} = \emptyset$, the modified supply $\tilde{C}^{e'}_S(t) = \tilde{C}^e_S(t)$ and the modified demand $\overline{C^{d'}_S(t)} = \overline{C^d_S(t)}$. The server fulfills the demand of $(2,3)$ by consuming one unit of the supply slot $(3,5)$ and effectively executing in the window $\overline{C^e_S(t)} = \{(3,4)\}$. We remove this used slot from $\tilde{C}^e_S(t)$ so that $C^{e'}_S(t) = \{(4,5),(10,12),(17,18),(21,24)\}$ and $\overline{C^{d'}_S(t)} = \{(5,6),(10,12)\}$. Next the demand window $(5,6)$ is served by using one unit of slot $(10,12)$, and so $\overline{C^e_S(t)} = \{(3,4),(10,11)\}$, $\tilde{C}^{e'}_S(t) = \{(4,5),(11,12),(17,18),(21,24)\}$ and $\overline{C^{d'}_S(t)} = \{(10,12)\}$. Next only the partial*

Algorithm 4: Actual Server Execution Curve

Input: $C_S^e(t)$, $\overline{C_S^d(t)} = \{W_{d,0}, W_{d,1} \ldots\}$
Output: $\overline{C_S^e(t)}$

(1) Apply the split operation on $C_S^e(t)$

$$\tilde{C}_S^e(t) = \{W_{e,0}, W_{e,1} \ldots\}$$

(2) Initialize: $\tilde{C}_S^{e'}(t) = \tilde{C}_S^e(t)$; $\overline{C_S^{d'}(t)} = \overline{C_S^d(t)}$; $\overline{C_S^e(t)} = \emptyset$
(3) Set $B(w) = 0$ for all w
(4) **while** $\tilde{C}_S^{e'}(t) = \emptyset \vee \overline{C_S^{d'}(t)} = \emptyset$ **do**
(a) Find the index of the first window $W_{e,k}$ from $\tilde{C}_S^{e'}(t)$ that potentially satisfies the demand of $W_{d,0}$.

$$j = \{\min(k) : W_{e,k}.e > W_{d,0}.s \wedge B(W_{e,k}) < C_s\}$$

(b) Compute the available budget in the group of $W_{e,j}$ as $b = C_s - B(W_{e,j})$. If the demand of $W_{d,0} > b$, split $W_{d,0}$ to obtain the valid demand segment.

$$W'_{d,0} = \begin{cases} W(W_{d,0}.s, W_{d,0}.s + b) & \text{if } W_{d,0}.l > b \\ W_{d,0} & \text{otherwise} \end{cases}$$

The residual demand is $W''_{d,0} = W(W_{d,0}.s + b, W_{d,0}.e)$
(c) Remove the original demand $W_{d,0}$ and add the residual demand $W''_{d,0}$ as the first window of the demand curve.

$$\overline{C_S^{d'}(t)} = \overline{C_S^d(t)} \setminus W_{d,0} \cup W''_{d,0}$$

(d) Fit the valid demand segment $W'_{d,0}$ into $W_{e,j}$. Return the overlap execution segment W_o, the remaining execution segment $R_{e,j}$ and the remaining demand window $R'_{d,0}$.

$$(R_{e,j}, W_o, R'_{d,0}) = W_{e,j} \ominus W'_{d,0}$$

(e) Update the budget of the replenishment group of $W_{e,j}$

$$B(W_{e,j}) = B(W_{e,j}) + W_o.l$$

(f) Update the usable execution window sets by removing the original window $W_{e,j}$ and adding the remaining execution window $R_{e,j}$ at the head position.

$$\tilde{C}_S^{e'}(t) = \tilde{C}_S^e(t) \setminus W_{e,j} \cup R_{e,j}$$

(g) Add W_o to the actual server execution curve.

$$\overline{C_S^e(t)} = \overline{C_S^e(t)} \cup W_o$$

end
return $(\overline{C_S^e(t)})$

demand $(10, 11)$ *in* $\overline{C_S^{d'}(t)}$ *is served by the supply slot* $(11, 12)$ *which yields a remaining unsatisfied demand* $\overline{C_S^{d'}(t)} = \{(11, 12)\}$, $\tilde{C}_S^{e'}(t) = \{(4, 5), (17, 18), (21, 24)\}$ *and* $\overline{C_S^e(t)} = \{(3, 4), (10, 11), (11, 12)\}$. *However, the supply slot* $(17, 18)$ *cannot be used to fulfill the remaining demand of* $(11, 12)$ *since in the interval* $[12, 20]$ *the server has already executed for 2 time units and exhausted its budget. Hence the server skips the supply window* $(17, 18)$ *and finally serves the remaining demand by consuming 1 unit of the slot* $(21, 24)$. *In this example, S effectively executes in the windows* $\overline{C_S^e(t)} = \{(3, 4), (10, 11), (11, 12), (21, 22)\}$ *with remaining demand* $\overline{C_S^{d'}(t)} = \emptyset$ *and remaining unused supply* $\tilde{C}_S^{e'}(t) = \{(4, 5), (17, 18), (22, 24)\}$.

Theorem 6.1. *A given server S executes in the windows defined by its actual server execution curve* $\overline{C_S^e(t)}$ *obtained in Algorithm 4, considering the interference from higher priority servers.*

Proof. We prove the above by showing that the following properties hold:

(1) $\overline{C_S^e(t)}$ does not contain windows overlapping with higher priority server executions: $\overline{C_S^e(t)}$ is the union of overlaps between the demand windows in $\overline{C_S^d(t)}$ with supply windows in $C_S^e(t)$. Each overlap $W_o \subseteq C_S^e(t)$, and therefore $\overline{C_S^e(t)} \subseteq C_S^e(t)$. By property 3, these are feasible windows in which S can execute without overlapping with higher priority server executions.
(2) In any replenishment window group of $\overline{C_S^e(t)}$, the total execution does not exceed C_s: This property holds because firstly, all supply windows $W_{e,k}$ whose execution group budget $B(W_{e,k})$ exceeds C_s are eliminated (Step 4a). Secondly, any demand window segment that, when serviced, could violate the budget of a replenishment group is returned to the demand pool and serviced in the next supply window with available group budget (Step 4b, 4c).
(3) The chosen execution windows in $\overline{C_S^e(t)}$ conform to the work conserving nature of the server: In Step 4a, the earliest feasible window with available group budget is chosen. Then, in Step 4d, the earliest position in that window is chosen. Thus, any pending demand is serviced as soon as there is available budget.
(4) The computed $\overline{C_S^e(t)}$ does not contain overlapping windows: This holds since the selected supply window in Step 4a is removed from the remaining supply pool in Step 4f.

□

7 RESPONSE TIME ANALYSIS

7.1 Determining the Span of Computation

As discussed in [2, 11, 13] it is sufficient to look at all scheduling situations up to the hyperperiod of a set of task with offsets (and without release jitter) to find the worst-case scheduling situations, and thus the worst-case response times. Afterwards, the schedule repeats itself (if the system is well dimensioned). This also holds for our system model, adding fixed priority servers to the picture that replenish their budget periodically.

We analyse all possible task and server period interleavings and compute response times until the system wide hyperperiod (SWH), computed as the least common multiple of all the server and task periods. Correspondingly, we compute the actual server execution curve until its capacity can serve all the jobs considered in the SWH. Note that for a server at a given priority, the required SWH can be further optimized since lower priority servers (and their tasks) do not have any effect on the response time, and hence their periods may be excluded from the computation of the SWH. The proposed approach is computationally efficient, since it is only necessary to record the system state (cumulative demand/service) at discrete schedule points (window boundaries).

7.2 Actual Execution Curve of a Task

The server execution curve $\overline{C_S^e(t)}$ computed as per Algo. 4 denotes the time intervals during which the server executes. These intervals are shared across its contained tasks by order of priority. Let $\{\tau = \tau_1 \ldots \tau_n\}$ denote the set of n tasks of S, indexed by decreasing

priority. To compute the worst-case response time R_i we first find the actual execution curve where task τ_i can execute.

Definition 14. *Given a server S with actual execution curve $\overline{C^e_S(t)}$ containing task τ_i, its task execution curve $\overline{C^e_{\tau_i}(t)}$, considering the interference from other higher priority tasks is computed as:*

(1) Let $C^e_{hp(\tau_i)}(t) = C^d_{\tau_0}(t) \oplus \ldots \oplus C^d_{\tau_{i-1}}(t)$ denote the aggregated high priority task demand

(2) The available execution curve of τ_i, $C^e_{\tau_i}(t)$, after subtracting $C^e_{hp(\tau_i)}(t)$:

$$(C^e_{\tau_i}(t), -, -) = \overline{C^e_S(t)} \ominus C^e_{hp(\tau_i)}(t)$$

(3) Fit the task demand curve $C^d_{\tau_i}(t)$ into $C^e_{\tau_i}(t)$ to obtain the overlap region corresponding to $\overline{C^e_{\tau_i}(t)}$.

$$(-, \overline{C^e_{\tau_i}(t)}, -) = C^e_{\tau_i}(t) \ominus C^d_{\tau_i}(t)$$

Lemma 7.1. *The actual execution curve $\overline{C^e_{\tau_i}(t)}$ computed as per Definition 14 represents the earliest time windows where τ_i executes in S, considering execution from higher priority tasks.*

Proof. By Property 2, the computed $C^e_{\tau_i}(t)$ excludes the execution windows of higher priority tasks. Then by Property 1, the demand windows of τ_i are fit in the earliest feasible remaining windows of $C^e_{\tau_i}(t)$. The resulting $\overline{C^e_{\tau_i}(t)}$, therefore, represents the earliest time windows where the jobs of τ_i finally execute, while respecting the priority order. □

7.3 Response Time of a Task

The task execution curve $\overline{C^e_{\tau_i}(t)}$ is used for executing consecutive jobs of τ_i. As per Lemma 7.1, it is sufficient to step through $\overline{C^e_{\tau_i}(t)}$ to obtain the individual job response times of τ_i. This is done by Algorithm 5 that computes the service time st that is needed by $\overline{C^e_{\tau_i}(t)}$ to provide t time units of service.

Obviously, the WCRT R_i of τ_i is given by the maximum response time of all its jobs that arrive up to the SWH (see Section 7.1).

Definition 15. *Given task τ_i with its actual execution curve $\overline{C^e_{\tau_i}(t)}$, the response time $R_{i,j}$ of its j-th job $\tau_{i,j}$ arriving at $a_{i,j}$ is computed as below:*

$$R_{i,j} = TimeToProvide(\overline{C^e_{\tau_i}(t)}, j * C_i) - a_{i,j}$$

Then, the worst-case response time R_i of τ_i is given by:

$$R_i = \max_{a_{i,j} < t} R_{i,j}$$

where t is the system wide hyperperiod.

The method works for tasks arriving at offsets and backlogged tasks, while also considering the effects of budget depletion by lower priority tasks, since the available server execution intervals are distributed first to the tasks as per task priorities and then as a second step these task execution intervals are in turn distributed by the arrival order to its jobs.

Example 9. *Consider a system with 2 deferrable servers as shown in Figure 6, with high priority server S_1 ($C = 3$, $T = 10$) and low priority server S_2 ($C = 2$, $T = 4$). While server S_1 hosts one task τ_1 ($C_1 = 1$, $T_1 = 4$), S_2 hosts τ_2 ($C_2 = 1$, $T_2 = 5$) and τ_3 ($C_3 = 2$, $T_3 = 8$), where τ_2 has a higher priority. We compute the WCRT of τ_2 where the actual execution curve of S_2 is $\overline{C^e_{S_2}(t)} = \{(1, 3), (5, 7), (9, 11), (13, 14), (15, 16)\}$ for $t = 16$. The demand curve of task τ_2 $C^d_{\tau_2}(t) = \{(0, 1), (5, 6), (10, 11), (15, 16)\}$. Since τ_2 is the highest priority task, it has the first right to the server execution windows, and executes in the windows $\overline{C^e_{\tau_2}(t)} = \{(1, 2), (5, 6), (10, 11), (15, 16)\}$.*

Here, job $\tau_{2,j}$ with $C_2 = 1$ needs $j \cdot C_2$ time units to be provided by $\overline{C^e_{\tau_2}(t)}$. $\tau_{2,1}1$ arriving at time 0 is served 1 time unit at time 2 with $R_{2,1} = 2$. Similarly 2 time units are provided to $\tau_{2,2}$ arriving at 5 at time 6 with $R_{2,2} = 1$. Thus we obtain $R_{2,3} = R_{2,4} = 1$, and on further computing upto $t = 40$ (the hyperperiod), we observe the maximum job response time is for $\tau_{2,8}$ with $R_{2,8} = 3$ and therefore $R_2 = 3$.

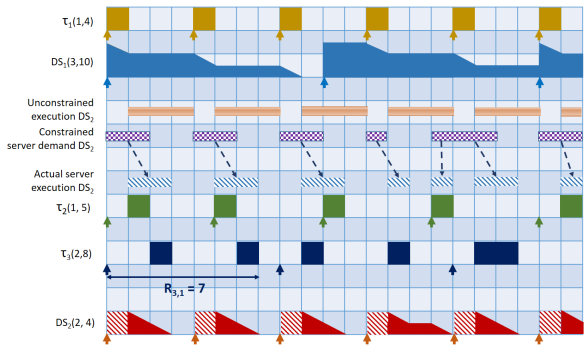


Figure 6: Response time analysis for a system of servers.

Algorithm 5: TimeToProvide()

Input: $\overline{C^e_{\tau_i}(t)} = \{W_{i,0}, \ldots W_{i,n}\}$, t
Output: The service time st

(1) Find maximum index k such that the cumulative service provided in $\{W_{i,0} \ldots W_{i,k}\} < t$

$$k = max(j) \text{ with } \sum_{p=1}^{j} W_{i,p}.l < t$$

(2) The remaining service is $b = t - \sum_{p=1}^{k} W_{i,p}.l$. If $b > 0$ and $k < n$, then the total service is provided in the next window $W_{i,k+1}$ at time $st = W_{i,k+1}.s + b$

return st

7.4 Remarks

Looking at all possible scheduling situations up to the SWH as provided in this paper is compulsory to judge about budget guarantees of servers in presence of deferrable servers. The reason is that it is not trivial to ensure that servers really provide C_s time units every T_s time units due to complex situations where multiple double hits of different deferrable servers coincide. The discussed SoA optimistically assumes those server guarantees in their analyses, and might, as a consequence, yield optimistic results. The following example illustrates that deferrable servers cannot be treated as "black boxes" in a scheduling analysis.

Example 10. *Consider a system comprising two deferrable servers $S_1(C_s = 1.5, T_s = 5)$ executing $\tau_1(T_1 = 11, C_1 = 3)$, and a lower priority server $S_2(C_s = 1T_s = 3)$ executing $\tau_2(T_2 = 200, C_2 = 50)$. The total server utilization of this system is 63.33%. The SoA analysis in [8] yields a WCRT of 153 for τ_2, which is optimistic since the real WCRT of 154 occurs while executing the 24-th (!) job of τ_2, released at time 4600, completes at time 4754. The reason for the optimistic results of the SoA analysis is that it does not take into account that the server guarantees of S_2 maybe violated due to rare double hits of S_1. More precisely, during the execution of the 24-th job of τ_2 only 0.5 time units of service are provided by S_2 in the intervals [4653, 4656) and [4719, 4722)*

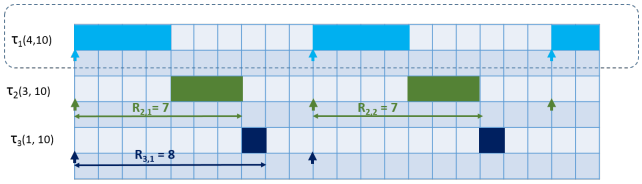


Figure 7: High priority server $S_1(C_s = 5, T_s = 10)$ hosts $\tau_1(C_1 = 4, T_1 = 10)$ and $S_2(C_s = 8, T_s = 20)$ hosts $\tau_2(C_2 = 3, T_2 = 10)$ and $\tau_3(C_1 = 1, T_1 = 10)$. τ_2 has a higher priority than τ_3

due to double hits. Thus, the system configuration does not conform to the assumptions of the SoA analyses, which is clearly not trivial to see, and the given system can, thus, not safely be analyzed by them.

Looking at all possible scheduling situations also explains the improvements of precision of our approach compared to the SoA that takes overly pessimistic assumptions as explained in the next section.

8 COMPARISON WITH THE SOA [8]

Example 11. *Consider a system with 2 DSs as shown in Figure 7. To compute the response time of τ_2, we observe the executions until SWH of 20. The task demand curves are given by: $\{C^d_{\tau_2}(t) = (0,3),(10,13)\}$ and $\{C^d_{\tau_3}(t) = (0,1),(10,11)\}$. Since the aggregated demand in the server period does not exceed its budget, the aggregated and the constrained demand curve of server S_2 are both given by $\{(0,4),(10,14)\}$. Considering interference from S_1, the unconstrained execution curve of server S_2 is given by $C^e_2(t) = \{(6,10),(16,20)\}$ while the actual execution curve considering its load and constrained demand, is $\overline{C^e_S(t)} = \{(4,8),(14,18)\}$. The task execution curve $C^e_{\tau_i}(t)$ for τ_2 is $\{(4,7),(14,17)\}$ and the response time $R_{2,1} = 7$ and $R_{2,2} = 7$ and therefore the $R_2 = 7$.*

We next compute R_2 as per the SoA approach in [8]. The approach iteratively computes the following terms until convergence.

- *The load $L(w)$ in time interval w considering its own execution + interference from higher priority tasks within the same server, solved until convergence, starting with $w = C_i$*
- *Gaps in complete server execution periods (Gap). This is given by $\lceil \frac{L(w)}{C_s} - 1 \rceil * (T_s - C_s)$*
- *Interference from the external higher priority servers (EHP) in the last segment of execution considering all server phasings.*

The response time is effectively the sum of the three components until the value converges. Finally if each task release time does not coincide with its server's replenishment time, an additional $T_s - C_s$ is added, since in the worst-case the analysis assumes that the entire server budget C_s is depleted when the task arrives.

k	w	$L(w)$	$Gap = \lceil \frac{L(w)}{8} - 1 \rceil * 12$	EHP	$R = L(w) + Gap + EHP$
0	3	3	0	5	8
1	8	3	0	10	13
2	13	3	0	10	13

Finally since the period of τ_2 = 10 does not always align with the server period of 20, their analysis adds a value of $T_s - C_s$ = 12 to 13 and thus R_2 = 25.Note that this is much higher than the period of the task (10). We also varied the offsets of each of the tasks from 0 to T_i to consider all possible interleavings and computed the response times. In no case the response time, is the value of 25 obtained.

The analysis in the SoA suffers from pessimism due to two sources of overestimation.

a) Overestimation of budget depletion by lower priority tasks: The analysis assumes that when τ_2 arrives, τ_3 depletes the entire budget and so τ_2 has to wait for $T_s - C_s$ time units. This does not happen in all cases as seen in our example.

b) Overestimation of the external server interference: This is attributed to two factors i) Assuming a higher interference from a server than it could inflict. This happens when the server is underloaded as in our example where S_1 is allocated 50% processing capacity while the contained task τ_1 needs only 40%. ii) The analysis "phases" arrival of the high priority server executions in a manner so as to inflict maximum interference. For a DS, it assumes a double hit "will occur" hence the EHP of 10. However, depending on the task and server replenishment periods double hits might not occur at all, and this can only be checked by considering the execution over the entire hyperperiod as in our analysis.

8.1 Experiment Setup

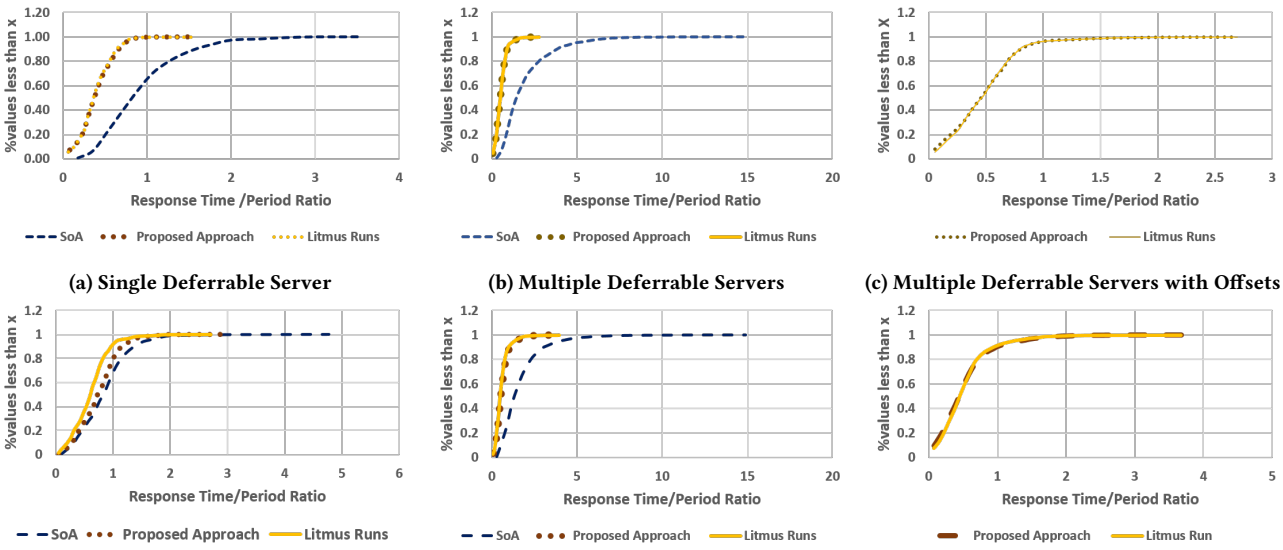
8.1.1 Task Set Parameter Generation. It was shown in [18] that the ratio between the smallest and largest task period in a task set could bias the results of a study. To address this issue, task periods are sampled from a log-uniform distribution instead of a regular uniform distribution. The task utilizations U_i are generated as per the UUnifast algorithm [4] in order that task WCETs C_i can be calculated as $C_i = U_i \cdot T_i$. Similarly, server periods T_s are sampled from a log-uniform distribution. Given n tasks and m servers, the mapping is given by randomly choosing one *composition* of n into exactly m parts. The periods of servers and tasks could be harmonic or non harmonic with respect to each other.

8.1.2 Comparison Metrics. For each of the experiments below, we compare the response times obtained through the SoA [8] (if applicable), our proposed approach and actual measurements obtained in LITMUS^{RT}. LITMUS^{RT} is a real-time extension of the Linux kernel with a focus on multiprocessor real-time scheduling and synchronization that also provides implementation of different servers. We scaled down the execution times in LITMUS^{RT} by a factor of 0.98 to account for scheduling overheads. For each of the tasks τ_i with period P_i, in each task set, we use the ratio $R'_i = R_i/P_i$ to obtain a normalized metric across different task sets, where lower values for R'_i indicate lower response times. We acknowledge that since our approach assumes that the information regarding tasks in each server is known, we can provide tighter bounds when multiple servers are involved. For each of the aforementioned approaches, we compute a cumulative density curve as seen in Figure 8b. A point (x, y) on the curve represents that $(y \cdot 100)$% of all the values lie below x. In LITMUS^{RT} we run each taskset for 2 seconds. Since the periods of the tasks and servers are in the order of few milliseconds, we can observe a significant number of jobs (more than the number of jobs considered in the analysis) and consider the maximum available value.

8.2 Different Server Scenarios

8.2.1 Intra Reservation Analysis: Single Deferrable Server. We considered 500 task sets where each task set had 5 tasks randomly generated and allocated to a single deferrable server. The tasks had zero offsets and the total load of each task set was 60%. It is clearly seen in Figure 8a that the SoA analysis estimates the response times of around 40% of all tasks to be greater than their periods, while the proposed approach calculates values that are very close to those measured in LITMUS^{RT}. In general, the response times computed by the SoA are much higher.

8.2.2 Multiple Deferrable Servers. We considered 500 task sets where each task set had 7 tasks randomly spread across 2 deferrable servers. The tasks had no offsets and the total load of each task set



(a) Single Deferrable Server

(b) Multiple Deferrable Servers

(c) Multiple Deferrable Servers with Offsets

(d) Multiple Periodic Servers

(e) Mixed Servers

(f) Mixed Servers with offsets

Figure 8: Comparison of Response Times

was 70%. It can be seen in Figure 8b that the proposed approach clearly outperforms the SoA. The latter estimates the response times of around 62% of all tasks to be larger than their periods, whereas LITMUS^{RT} and the proposed approach show that this is only the case for around 5.7% of all tasks. The experiment also demonstrates that the pessimism of existing SoA approaches keeps increasing and does not scale to multiple servers.

8.2.3 Multiple Deferrable Servers with Tasks with Offsets. We considered 500 task sets, each with a load of 70%, where each task set had 5 tasks randomly spread across 2 deferrable servers. Every task had an offset that was randomly chosen between 0 and 0.4 times its period. Our analysis also accounted for backlogged activations. In this experiment, we compared the analytic response time value ratios against those measured in LITMUS^{RT}. Since the SoA approach cannot deal with offsets, we did not compare it. It can be seen in Figure 8c that we propose an almost exact analysis.

8.2.4 Multiple Periodic Servers. Figure 8d illustrates the results for 500 task sets with a total load of 60% where 7 tasks were distributed across 3 servers. It can clearly be seen that while the proposed approach provides bounds close to that of LITMUS^{RT}, the considered SoA method calculates more pessimistic response times. Note that the results of the PS are even less pessimistic than those of the deferrable server experiments, which indicates that the SoA is better suited for the former.

8.2.5 Mixed Servers with No Task Offsets. In this experiment, we considered a mix of DS and PS. Figure 8e shows the results across 500 task sets where each task set had 10 tasks randomly assigned to one of the three servers. The type of each serve was also randomly chosen. Again the proposed approach calculates response times that are very close to the actually observed executions in LITMUS^{RT}, whereas the SoA analysis is far more conservative. The SoA approach reports that around 73% of the samples have a response time greater than that the task period, while only 9% of the samples observed on LITMUS^{RT} exhibit such a response time.

8.2.6 Mixed Servers with Task Offsets. We considered a mix of deferrable and periodic servers. Figure 8f shows the results across 500 task sets where each task set had 8 tasks randomly assigned to one of the three servers with random type. Additionally, each task had an offset which was randomly chosen between 0 and 0.4 times its period. Figure 8f shows the results that proves that we present an almost exact analysis.

8.3 Analysis Time

Each experiment set of 500 tasks took between 8 to 20 minutes on a regular Intel Core-i5-based machine. This time also included the generation of a feasible configuration as well as the execution of our proposed analysis. With regard to LITMUS^{RT}, each experiment set took around 30 minutes to complete (task runs, invocations of each set and result logging). Even though the analysis presented by [8] performs faster than our proposed approach, the time that our analysis took is very reasonable considering the tightness of the bounds.

9 SUMMARY

In this work, we propose a method based on the demand and service abstraction and demonstrate that tight bounds can be derived for computing the response times of tasks in a system comprising deferrable and periodic servers. Our approach outperforms the state-of-the-art, relaxes assumptions on the task model and also shows that it is tight with respect to the runtime values observed on LITMUS^{RT}. As future work, we will extend our analysis to consider mechanisms for slack handling.

REFERENCES

[1] Luis Almeida and Paulo Pedreiras. 2004. Scheduling Within Temporal Partitions: Response-time Analysis and Server Design. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04)*. ACM, New York, NY, USA, 95–103. https://doi.org/10.1145/1017753.1017772

[2] N. C. Audsley. 2001. On Priority Asignment in Fixed Priority Scheduling. *Inf. Process. Lett.* 79, 1 (May 2001), 39–44. https://doi.org/10.1016/S0020-0190(00)00165-4

[3] G. Bernat and A. Burns. 1999. New results on fixed priority aperiodic servers. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*. 68–78. https://doi.org/10.1109/REAL.1999.818829

[4] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the Performance of Schedulability Tests. *Real-Time Systems* 30, 1 (01 May 2005), 129–154. https://doi.org/10.1007/s11241-005-0507-9

[5] Bjorn B. Brandenburg. 2011. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. Ph.D. Dissertation. Chapel Hill, NC, USA. Advisor(s) Anderson, James H. AAI3502550.

[6] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. LITMUSRT: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of 27th IEEE International Real-Time Systems Symposium, RTSS 2006*. 111–123. https://doi.org/10.1109/RTSS.2006.27

[7] AUTOSAR consortium. 2014. AUTOSAR - Specification of Operating System. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_OS.pdf

[8] R. I. Davis and A. Burns. 2005. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*. IEEE Computer Society, Washington, DC, USA, 389–398. https://doi.org/10.1109/RTSS.2005.25

[9] R. I. Davis and A. Burns. 2006. Resource Sharing in Hierarchical Fixed Priority Pre-Emptive Systems. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*. IEEE Computer Society, Washington, DC, USA, 257–270. https://doi.org/10.1109/RTSS.2006.42

[10] M. Joseph and P. Pandya. 1986. Finding Response Times in a Real-Time System. *Comput. J.* 29, 5 (1986), 390–395. https://doi.org/10.1093/comjnl/29.5.390

[11] Joseph Y-T Leung and Jennifer Whitehead. 1982. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation* 2, 4 (1982), 237–250.

[12] G. Lipari and E. Bini. 2003. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.* 151–158. https://doi.org/10.1109/EMRTS.2003.1212738

[13] O. Redell and M. Torngren. 2002. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium.* 164–172. https://doi.org/10.1109/RTTAS.2002.1137391

[14] Saowanee Saewong, Ragunathan (Raj) Rajkumar, John P. Lehoczky, and Mark H. Klein. 2002. Analysis of Hierarchical Fixed-Priority Scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS '02)*. IEEE Computer Society, Washington, DC, USA, 173–.

[15] Insik Shin and Insup Lee. 2008. Compositional Real-time Scheduling Framework with Periodic Model. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 30 (May 2008), 39 pages. https://doi.org/10.1145/1347375.1347383

[16] J. K. Strosnider, J. P. Lehoczky, and Lui Sha. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.* 44, 1 (Jan 1995), 73–91. https://doi.org/10.1109/12.368008

[17] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. 2000. Real-time Calculus for Scheduling Hard Real-Time Systems. In *International Symposium on Circuits and Systems ISCAS 2000*, Vol. 4. Geneva, Switzerland, 101–104.

[18] Attila Zabos, Alan Burns, and Robert I. Davis. 2008. Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems. *IEEE Trans. Comput.* 57 (2008), 1261–1276. https://doi.org/doi.ieeecomputersociety.org/10.1109/TC.2008.66