On Optimal Hierarchical Resource Sharing in Open Environments*

Insik Shin, Moris Behnam, Thomas Nolte, Mikael Nolin
Mälardalen Real-Time Research Centre (MRTC)
Mälardalen University, 721 23 Västerås, Sweden

Abstract

This paper presents a new perspective in the context of supporting logical resource sharing under hierarchical scheduling. Our work is motivated from a tradeoff between reducing resource holding times and reducing system load (i.e., the collective processor requirements to guarantee the schedulability of hierarchical scheduling frameworks). We formulate an optimization problem that determines the resource holding times of each individual tasks (and therefore those of subsystems) with the goal of minimizing the system load subject to the system's schedulability. We present efficient algorithms to find an optimal solution to the problem, and we prove their correctness.

1 Introduction

This paper deals with hierarchical scheduling of *open real-time systems*, where subsystems are allowed to be developed and validated independently of each other in one environment, and later integrated in another environment.

Motivation. Tasks often require exclusive accesses to shared resources. When a task holds an exclusive access to a shared resource, it inherently blocks other tasks that want to access the same resource. Several synchronization protocols [2, 17, 19] provide rules on how tasks gain exclusive accesses, primarily, focusing on how to bound their blocking times. These protocols mainly achieve it through bounding of resource holding times. For example, the Stack Resource Policy (SRP) [2] allows a task within a critical section to be preempted by another task τ_i, only if there is no potential to increase the blocking time of any task with higher priority than that of τ_i. These protocols share the objective of bounding blocking times of higher-priority tasks.

Subject to schedulability, Fisher *et al.* [10] recently proposed to minimize resource holding times, allowing for preemption within critical sections. They presented an algorithm to decrease the resource holding times of tasks under the semantics of SRP, however, at the expense of increasing the blocking times of higher-priority tasks. Their basic idea is to increase the ceiling of resources as much as possible (in order to minimize resource holding times) without violating schedulability. Having shorter resource holding times is very useful in the context of HSFs, since this could allow for lower allocation of resources to subsystems.

However, when resources are shared between subsystems, a HSF needs to employ some synchronization protocol (e.g., HSRP [6], SIRAP [3]). Such protocols consider a subsystem's resource holding time as an extra requirement in terms of CPU resources needed to achieve schedulability. Therefore, defining *system load* as a quantitative measure to represent the collective CPU requirements of all subsystems, one could expect that minimizing resource holding times would be effective for reducing the system load. However, when using these

*The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

protocols, minimizing a subsystem's resource holding times may in fact increase the system load. That is, there exists a tradeoff between reducing a subsystem's resource holding time and the resulting system load. Given such a tradeoff, we introduce the *system load minimization problem* as the problem of determining the resource holding times of subsystems such that the system load can be minimized.

Approach. By assigning each subsystem with an abstract representation of its CPU requirements and resource holding times, one may consider a divide-and-conquer approach to the system load minimization problem. We denote these abstract representations as *interfaces*, and for each subsystem its interface is independently calculated. In a system, once subsystems are integrated, their interfaces are used to determine the system load. However, an independent approach is not feasible for the minimization problem; for a single subsystem we cannot determine which resource holding times that would yield the minimum system load. This is because the CPU requirement of a subsystem depends on its blocking time, and its blocking time subsequently depends on the resource holding times of other subsystems.

Hence, we employ a two-step approach to find an optimal solution to the system load minimization problem. In the first step, for each subsystem independently, we derive a set of interface candidates (i.e., a set of resource holding time candidates). In the second step, during system integration, we collect the interface candidates of all subsystems and selects one candidate for each subsystem to generate the minimum system load.

Contribution. The contribution of this paper is four-fold. First, motivated by the tradeoff between reducing resource holding times and reducing system load, we introduce a new tradeoff problem to bound resource holding times in the context of hierarchical scheduling. Second, aiming at minimizing the system load, we formulate an optimization problem to determine the resource holding times of each subsystem. Third, we present two algorithms that together find an optimal solution to the problem. The first algorithm efficiently generates a bounded number of interface candidates for each subsystem, and we prove that the interface used to generating an optimal solution is contained within this set of interfaces. The second algorithm efficiently finds an optimal solution using the interface candidates of all subsystems, and we prove its correctness. Fourth, we also extend the schedulability analysis of HSFs with HSRP [6] to be suitable for open environments.

Section 2 presents related work, followed by the system model and background in Section 3. Section 4 presents how resources are shared in our HSF. Section 5 addresses the first step of our approach to the system load minimization problem; efficiently generating interface candidates, and Section 6 resolves the second step effectively finding an optimal solution out of the candidates. Finally, Section 7 concludes.

2 Related work

Hierarchical scheduling. Over the years, there has been a growing attention to HSFs [1, 5, 7, 8, 11, 13, 14, 18, 20, 21] for real-time systems. Since Deng and Liu [7] proposed a two-level HSF for open systems, many studies have been proposed for its schedulability analysis of HSFs [11, 13, 15]. Various processor models, such as bounded-delay [16] and periodic [14, 20], have been proposed for a multi-level HSFs, and schedulability analysis techniques have been proposed for the proposed processors models [1, 5, 8, 14, 18, 20, 21]. However, none of the above studies consider supporting logical resource sharing in HSFs.

Resource sharing. To support logical resource sharing in a mutual exclusive manner, some synchronization protocols are proposed. They provide rules about how to gain access to the resource, and specifie which tasks should be blocked when trying to access the resource. To achieve predictable real-time behaviour, several protocols have been proposed including the Priority Inheritance Protocol (PIP) [19], the Priority Ceiling Protocol (PCP) [17], and the Stack Resource Policy (SRP) [2]. Fisher *et al*. [4, 10] proposed algorithms to minimize the time duration that a task locks a resource under fixed priority and EDF scheduling with SRP.

The issues of supporting resource sharing in HSFs have been considered. Deng and Liu [7] proposed the usage of non-preemptive global resource access, which bounds the maximum blocking time that a task might be subject to. It was shown in [11, 1] that traditional protocols such as SRP can be used to support local resource sharing within a subsystem. Recently, a few studies (e.g., HSRP [6], SIRAP [3], BROE [9]) have been proposed for supporting resource sharing between subsystems in HSFs. In summary, compared to the work in this paper, none of the above approaches have addressed the tradeoff between reducing a subsystem's resource holding times and the resulting system load.

3 System model and background

A HSF is introduced to support CPU time sharing among applications (subsystems) under different scheduling services. The system-level global scheduler allocates CPU time to subsystems, and the subsystem-level local schedulers subsequently schedule CPU time to their internal tasks. This framework also allows logical resource sharing between tasks in a mutually exclusive manner. Tasks can share *local* logical resources within a subsystem and *global* logical resources across subsystems. In this paper we focus on global logical resources while local logical resources can be easily supported by traditional synchronization protocols such as SRP [1, 6, 11].

3.1 Virtual processor models

The notion of real-time virtual processor (resource) model was first introduced by Mok *et al.* [16] to characterize the CPU allocations that a parent node provides to a child node in a HSF. The *CPU supply* of a virtual processor model refers to the amounts of CPU allocations that the virtual processor model can provide. The *supply bound function* of a virtual processor model calculates the minimum possible CPU supply for any given time interval of length t.

Shin and Lee [20] proposed the periodic virtual processor model $\Gamma(P, Q)$ to characterize periodic processor allocations, where P is a period ($P > 0$) and Q is a periodic allocation time ($0 < Q \leq P$). The capacity U_Γ of a periodic virtual processor model $\Gamma(P, Q)$ is defined as Q/P.

The supply bound function $\mathtt{sbf}_\Gamma(t)$ of the periodic model $\Gamma(P, Q)$ was given in [20] to compute the minimum possible CPU supply for every interval length t as follows:

$$\mathtt{sbf}_\Gamma(t) = \begin{cases} t - (k+1)(P-Q) & \text{if } t \in [(k+1)P - 2Q, \\ & (k+1)P - Q], \\ (k-1)Q & \text{otherwise}, \end{cases} \quad (1)$$

where $k = \max\Big(\lceil (t-(P-Q))/P \rceil, 1\Big)$. Here, we first note that an interval of length t may not begin synchronously with the beginning of period P; as shown in Figure 1, the interval of length t can start in the middle of the period of a periodic model $\Gamma(P, Q)$. Figure 1 illustrates the supply bound function $\mathtt{sbf}_\Gamma(t)$.

3.2 Stack Resource Policy (SRP)

To use SRP [2] in a HSF, we extend terms associated with SRP as follows:

- *Preemption level*. Each task τ_i has a preemption level equal to $\pi_i = \mathcal{P}r_i$, where $\mathcal{P}r_i$ is the priority of τ_i. Similarly, each subsystem S_s has a preemption level equal to $\Pi_s = \mathcal{PR}_s$, where $\mathcal{PR}_s$ is the subsystem's priority.
- *Resource ceiling*. Each global shared resource R_j is associated with two types of resource ceilings; an *internal* resource ceiling for local scheduling $rc_j = \max\{\pi_i | \tau_i \text{ accesses } R_j\}$ and an *external* resource ceiling for global scheduling $RX_s = \max\{\Pi_s | S_s \text{ accesses } R_j\}$.

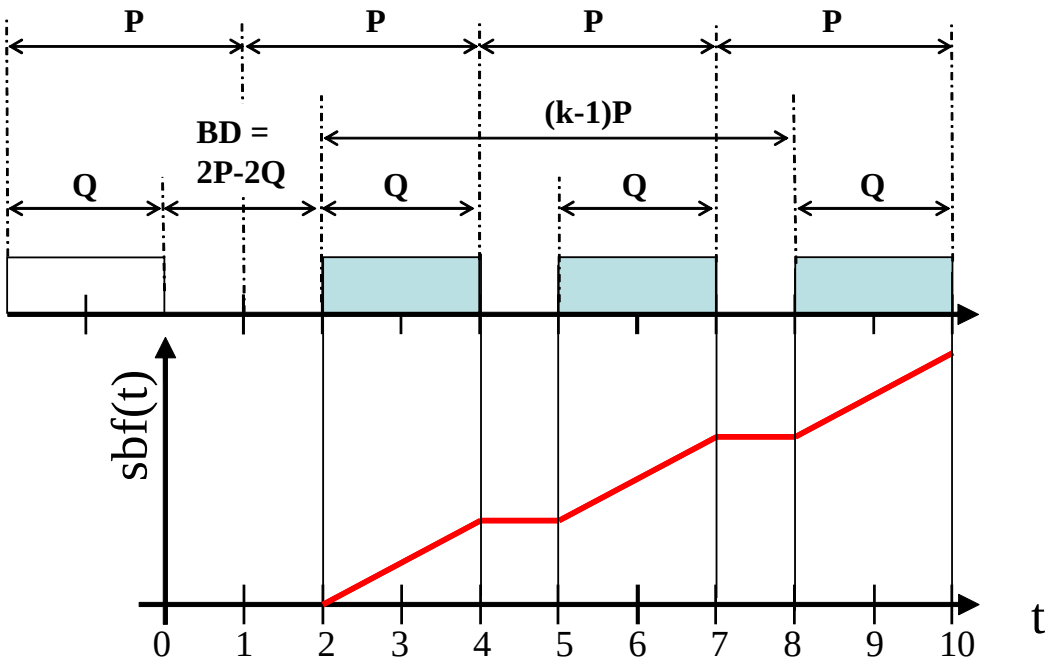

Figure 1. The supply bound function of a periodic virtual processor model $\Gamma(3, 2)$.

- *System and subsystem ceilings*. System and subsystem ceilings are dynamic parameters that change during execution. The system (subsystem) ceiling is equal to the currently locked highest external (internal) resource ceiling in the system (subsystem).

According to SRP, a job J_i generated by task τ_i can preempt the currently executing job J_k within a subsystem only if J_i is a higher-priority job of J_k and the preemption level of τ_i is greater than the current subsystem ceiling. The same reasoning can be made for subsystems from a global scheduling point of view.

3.3 System model

We consider a deadline-constrained sporadic task model $\tau_i(T_i, C_i, D_i, \mathcal{P}r_i, \{c_{i,j}\})$, where T_i is the minimum separation time between its successive jobs, C_i, is the worst-case execution time (WCET), D_i is the relative deadline ($C_i \leq D_i \leq T_i$), $\mathcal{P}r_i$ is the priority, and each element $c_{i,j}$ in $\{c_{i,j}\}$ represents the WCET of τ_i inside a critical section of the global shared resource R_j. We assume that all tasks have unique priorities and are sorted according to their priorities in the order of increasing priority.

For a shared resource R_j, the *resource holding time* h_j^l with internal resource ceiling $rc_j = l$ is defined as the maximum task execution time inside a critical section plus the interference (inside the critical section) of higher priority tasks that have preemption level greater than the internal ceiling of the locked resource. h_j^i is computed [4] using $W_j(t)$ as follows;

$$W_j(t) = cx_j + \sum_{k=rc_j+1}^{u} \lceil \frac{t}{T_k} \rceil C_k, \tag{2}$$

where $cx_j = \max\{c_{i,j}\}$ for all task τ_i uses resource R_j, i.e., the maximum execution time inside a critical section of a task among all tasks that use resource R_j, and u is the greatest internal ceiling within the subsystem.

The resource holding time h_j^l is the smallest positive time t_l^* such that

$$W_j(t_l^*) = t_l^*. \tag{3}$$

A subsystem $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the whole system of subsystems, is characterized by a task set $\mathcal{T}_s$ and a set of internal resource ceilings $\mathcal{RC}_s$ of the global shared resources. Each subsystem S_s is assumed to have a fixed-priority local scheduler (FPS local scheduler). Each subsystem S_s has an interface (the subsystem interface) that is defined as (P_s, Q_s, H_s), where P_s is a period, Q_s is an execution requirement budget, and H_s is a maximum global resource holding time, i.e., $H_s = \max\{h_j^l | \text{ for all } R_j \in \mathcal{R}_s\}$, where $\mathcal{R}_s$ is the set of global shared resources used by the internal tasks of S_s.

4 Resource sharing in the HSF

4.1 Overrun mechanism

This section explains overrun mechanisms that can be used to handle budget expiry during a critical section in a HSF. Consider a global scheduler that schedules subsystems according to their periodic interfaces (P_s, Q_s, H_s). The subsystem budget Q_s is said to *expire* at the point when one or more internal (to the subsystem) tasks have executed a total of Q_s time units within the subsystem period P_s. Once the budget is expired, no new tasks within the same subsystem can initiate execution until the subsystem's budget is replenished. This replenishment takes place in the beginning of each subsystem period, where the budget is replenished to a value of Q_s.

Budget expiration can cause a problem, if it happens while a job J_i of a subsystem S_s is executing within the critical section of a global shared resource R_j. If another job J_k, belonging to another subsystem, is waiting for the same resource R_j, this job must wait until S_s is replenished so J_i can continue to execute and finally release the lock on resource R_j. This waiting time exposed to J_k can be potentially very long, causing J_k to miss its deadline.

In this paper, we consider a mechanism based on overrun [6] that works as follows; when the budget of subsystem S_s expires and S_s has a job J_i that is still locking a global shared resource, job J_i continues its execution until it releases the locked resource. The extra time that J_i needs to execute after the budget of S_s expires is denoted as *overrun time* θ. The maximum θ occurs when J_i lock a resource that gives the longest resource holding time just before the budget of S_s expires. This worst case happens when θ equals to H_s, where H_s represents the greatest global resource holding time of S_s. To solve the budget expiration problem, one simply adds the maximum overrun time θ to the subsystem budget for each subsystem S_s such that the new subsystem budget is $\widehat{Q_s} = Q_s + H_s$.

4.2 Schedulability analysis

In this paper, we use HSRP [6] for resource synchronization in HSF. Schedulability analysis under global and local FPS with the overrun mechanism is presented in [6]. However, the presented approach is not suitable for open environments. Hence, this section presents the schedulability analysis of local and global FPS using subsystem interfaces, which is suitable for open environments.

Local schedulability analysis. Let $\texttt{dbf}_{\texttt{FP}}(i,t)$ denote the demand bound function of a task τ_i under FPS [12], i.e.,

$$\texttt{dbf}_{\texttt{FP}}(i,t) = C_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \tag{4}$$

where $\texttt{HP}(i)$ is the set of tasks with higher priorites than that of τ_i. The local schedulability analysis under FPS can be then easily extended from the results of [2, 20] as follows:

$$\forall \tau_i, 0 < \exists t \leq D_i \;\; \texttt{dbf}_{\texttt{FP}}(i,t) + b_i \leq \texttt{sbf}(t), \tag{5}$$

where b_i is the longest blocking time during which a job J_i may be blocked by lower priority jobs, and $\texttt{sbf}(t)$ is the supply bound function.

Subsystem interface. We now explain how to derive the budget Q_s of the subsystem interface. Given S_s, $\mathcal{R}C_s$, and P_s, let $\mathsf{calculateBudget}(S_s, P_s, \mathcal{R}C_s)$ denote a function that calculates the smallest subsystem budget that satisfies Eq. (5) depending on the local scheduler of S_s. Such a function is similar to the one in [20]. Then, $Q_s = \mathsf{calculateBudget}(S_s, P_s, \mathcal{R}C_s)$.

Global schedulability analysis. Under global FPS scheduling, we present the system load bound function as follows (on the basis of a similar reasoning of Eq. (eq:rm-dbf)):

$$\mathtt{LBF}_s(t) = \mathtt{DBF}_s(t) + B_s \ , \text{ where} \tag{6}$$

$$\mathtt{DBF}_s(t) = (Q_s + H_s) + \sum_{S_k \in \mathtt{HPS}(S_s)} \left\lceil \frac{t}{P_k} \right\rceil \cdot (Q_k + H_k), \tag{7}$$

where $\mathtt{HPS}(S_s)$ is the set of subsystems with higher-priority than that of S_s and the system-level blocking time B_s represents the maximum blocking time during which subsystem S_s may be blocked by lower priority subsystems, i.e., $B_s = \max(H_j|$ for all $S_j \in LPS(S_s))$, $LPS(S_s)$ is the set of subsystems with lower priority than that of S_s.

A global schedulability condition under FPS is then

$$\forall S_s, 0 < \exists t \leq P_s \ \mathtt{LBF}_s(t) \leq \mathtt{t}, \text{where} \tag{8}$$

System load. As a quantitative measure to represent the minimum amount of processor allocations necessary to guarantee the schedulability of a subsystem S_s, let us define *processor request bound* (α_s) as

$$\alpha_s = \min_{0<t\leq P_s} \{\frac{\mathtt{LBF}_s(t)}{t} \mid \mathtt{LBF}_s(t) \leq t\}. \tag{9}$$

In addition, let us define the *system load* $\mathsf{load}_{\mathsf{sys}}$ of the system under global FPS as follows:

$$\mathsf{load}_{\mathsf{sys}} = \max_{\forall S_s \in \mathcal{S}} \{\alpha_s\}. \tag{10}$$

5 Interface candidate generation

In this paper, we consider a two-step approach to the system load minimization problem. In this section, we address the first step that each component generates a set of interface candidates. In the second step, when the interface candidates of all subsystems are available, one of the candidates of each subsystem will selected in order to minimize the system load.

We define the *interface candidate generation* problem as follows. Given a subsystem S_s and a set of global resources, the problem is to generate a set of interface candidates IC_s such that there exists an element of IC_s that constitutes an optimal solution to the system load problem.

As shown in the previous section, the value of Q_s is evaluated using the function $\mathsf{calculateBudget}(S_s, P_s, \mathcal{R}C_s)$. The function has 3 parameters $(S_s, P_s, \mathcal{R}C_s)$ where S_s contains the bounded number of tasks n, and $\mathcal{R}C_s$ has indicates a bounded number of resources m. A brute-force solution to the interface generation problem is to generate all possible m^n interface candidates. However, not all of these m^n candidates have the potential to constitute the optimal solution; some may require more resource demand and impose higher blocking on other subsystems and others may act as replicate interfaces.

Hence, we present an algorithm to find a correct solution to the problem. Our algorithm is computationally efficient and produces a bounded number of interface candidates. We first provide some notions and properties on which our algorithm is based. We then explain our algorithm and illustrate it. Hereinafter, we assume that P_s is given by system designer and is fixed during the whole process of generating a set of interface candidates. Therefore an interface candidate (P_s, Q_s, H_s) can be denoted as (Q_s, H_s).

Definition 1 *An interface candidate (Q_k, H_k) is said to be* redundant *if there exists (Q_i, H_i) such that $H_i \leq H_k$ and $Q_i \leq Q_k$ and $i \neq k$ (denoted as $(Q_i, H_i) \leq (Q_k, H_k)$). If two interfaces are equivalent ($(Q_i, H_i) = (Q_k, H_k)$)), then the interface with lower index is considered as a redundant interface. In addition, (Q_i, H_i) is said to be* non-redundant *if it is not redundant.*

Lemma 1 *A redundant interface candidate does not constitute an optimal solution to the system load minimization problem.*

Proof Suppose an interface candidate (Q_a, H_a) is redundant. By definition, there exists another candidate (Q_b, H_b) such that $H_b \leq H_a$ and $Q_b \leq Q_a$. So $(Q_b + H_b) <= (Q_a + H_a)$. Using a redundant interface candidate only increases $DBF_s(t)$ (see Eq. (7)) and the blocking time B_s, respectively, compared to a non-redundant candidate. It means that using a redundant candidate can only increases $\texttt{LBF}_\texttt{s}(\texttt{t})$ and thereby $\texttt{load}_s$ (see Eq. (9)). That is, a redundant candidate only has a potential to increase $\texttt{load}_\texttt{sys}$ (see Eq. (10)). In other words, a redundant candidate cannot constitute an optimal solution to the system load minimization problem. □

Lemma 1 suggests that redundant candidates be excluded from a solution, and it reduces the number of interface candidates significantly. However, a brutal-force approach to reduce redundant candidates is still computationally intractable. The complexity of an exhaustive search is very high $O(m^n)$. We present interesting properties that help to develop a computationally efficient algorithm.

When tasks access the same shared resource, their maximum blocking time can be different depending on their preemption levels. This can happen particularly when the tasks sharing the same resources have different execution times inside critical sections. To capture this, we introduce the following notation.

X_a^b Let X_a^b be the maximum blocking time that a task of preemption level b may experience in accessing resource R_a, i.e., $X_a^b = \max\{c_{j,a}\}$ for all $j < b$.

The following lemma shows an important property: if a resource R_k implies the longest blocking time among the resources having the same resource ceilings, then R_k will provides the longest resource holding time among them.

Lemma 2 *Let $\mathcal{R}^i$ denote a set of resources whose resource ceilings are i. Suppose a resource $R_k \in \mathcal{R}^i$ yields the greatest blocking time among all the elements of $\mathcal{R}^i$. Then, the resource R_k generates the greatest resource holding time among all the elements of $\mathcal{R}^i$, i.e.,*

$$\Big(X_k^i = \max_{\forall R_j \in \mathcal{R}^i}\{X_j^i\}\Big) \rightarrow \Big(h_k^i = \max_{\forall R_j \in \mathcal{R}^i}\{h_j^i\}\Big). \tag{11}$$

Proof The resource holding time h_j^i of a resource R_j at its ceiling of i depends on two parameters (see Eq. (3)); the maximum blocking at that ceiling ($X_j^i = rx_j$) and the interference from tasks with higher preemption level (The summation part I). Since I is the same for all resources that have the same ceiling, h_j^i depends only on X_j^i as only tasks with preemption level greater than i will contribute in the summation. □

The following shows when redundant interface candidates can be generated. In other words, it indicates when we can effectively exclude redundant candidates.

Lemma 3 *Consider a resource R_y of a ceiling k ($rc_y = k$) and another resource R_z of a ceiling i ($rc_z = i$), where $k < i$. Suppose $X_y^k < X_z^k$ and $rc_y < rc_z$. Then an interface candidate generated by having the ceiling $rc_y = k + 1, .., i$ is redundant, hence it is possible to increase the ceiling of R_y to that of R_z directly (i.e., $rc_y = rc_z = i$).*

Proof Let (Q', H') denote an interface candidate generated when $rc_y = k$ and $rc_z = i$, where $k < i$. Let (Q^*, H^*) denote another interface candidate generated when $rc_y = rc_z = i$. We wish to show that $(Q^*, H^*) \leq (Q', H')$, i.e., $Q^* \leq Q'$ and $H^* \leq H'$.

Given $X_y^i < X_z^i$, it follows from Lemma 2 that $h_y^i < h_z^i$. This means that even though the ceiling of R_y increases to i, it does not change the maximum blocking time ($b(t)$) during $t \in [D_{i+1}, D_i]$. Therefore, it does not change the demand bound function either. As a result, $Q^* = Q'$.

We wish to show that $H^* \leq H'$. When the ceiling of R_y increases to i from k, its resulting resource holding time h_y^i becomes smaller than h_y^k because there will be less interference from higher priority tasks, (i.e., $h_y^i < h_y^k$). In fact, this is the only change that occurs to the resource holding time of all shared resources when rc_y increases. Hence, the maximum resource holding time H can remain the same (if $h_y^k < H'$) or decrease (if $h_y^k = H'$) after rc_y increases. That is, $H^* \leq H'$. □

Using Lemmas 1, 2, and 3, we can reduce the complexity of a search algorithm. The algorithm shown in Figure 2 is based on these Lemmas. It increases the ceiling of the resource that generates the maximum resource holding time by one step, and then checks the conditions given in Lemma 3 to further increase the ceiling of that resource if possible. It then increases the ceiling of all resources that have the same ceiling as the selected resource, to the selected resource ceiling. This way, we can reduce redundant interface candidates. Lines 8-11 checks the condition in Lemma 1. The following lemma proves the correctness of algorithm shown in Figure 2.

Lemma 4 *Let $\mathcal{IC}$ denote a set of up to n interface candidates that are generated by the algorithm of Figure 2. There exists no non-redundant interface candidate (Q_y, H_y) such that $(Q_y, H_y) \notin \mathcal{IC}$.*

Proof Assume that (Q_y, H_y) is a non-redundant interface candidate and that $H_y = h_k^i$, i.e., the resource holding time of R_k is the maximum among all global shared resources when $rc_k = i$. Then we shall prove that

1. There is no R_j such that $X_j^i > X_k^i$ for all $rc_j > i$. Otherwise we could change the ceiling $rc_k = rc_j$ according to Lemma 3, and by this $h_k^i \neq H_y$.

2. There is no R_j such that $X_j^t > X_k^i$ for all $rc_j < i, t < i$. Otherwise $h_j^t > h_k^i$ because when we compute the resource holding time of R_k and R_j, the interference from higher preemption level tasks as well as blocking is higher for R_j, and then $h_k^i \neq H_y$. If we increase the ceiling $rc_j = i$, it will not give other non-redundant interface candidates (see Lemma 2 and 3).

We can conclude that there is only one resource R_k that may generate a non-redundant interface at a preemption level i, and this is the one that imposes the highest blocking at that level. The initial ceiling of R_k is v, where $v \in [1, i]$. From Lemma 2, X_k^f (where $f \in [v, i]$) is the maximum blocking at preemption level $rc_k \in [v, i]$. Since the presented algorithm increases the ceiling of the maximum resource holding time, it will increase the ceiling of R_k when $rc_k = v$ up to i. Hence, we can guarantee that the algorithm will cover the interface when $H_y = h_k^i$. □

The proof of the previous property also shows that the complexity of the proposed algorithm is $O(n)$ since we have n tasks (which equals to the number of preemption levels) and there is either 0 or 1 non-redundant interface for each preemption level, and the algorithm will only traverse these non-redundant interfaces. Moreover, the proposed algorithm thereby produce at most n interface candidates.

5.1 Example

We illustrate the local algorithm with the following example. Consider a subsystem S_s that has six tasks as shown in Table 1. The local scheduler for the subsystem S_s is Rate Monotonic (RM) and we choose subsystem period $P_s = 100$.

- calculateBudget$(S_s, P_s, \mathcal{RC}_s)$ returns the smallest subsystem budget that satisfies Eq. (4).
- increaseCeilingH$^*(\mathcal{RC}_s)$ returns whether or not the ceiling of the resource associated with H^* (the current greates resource holding time) can be increased by one i.e., the ceiling of the resource associated with $H^* \neq$ maximum ceiling. If so, it increases the ceiling of the selected resource as well as the ceiling of all resources that have the same ceiling as the selected resource, motivated by Lemma 3.
- Interface is an array of interface candidates; each candidate is (Q, H, RC).

```
1:  RC_s = {rc_1, ..., rc_m}
2:  num=0
3:  Interface[num].Q = P_s
4:  do
5:      H* = max{h(rc_1), ..., h(rc_m)}
6:      Q = calculateBudget(S_s, P_s, RC_s)
7:      count = num
8:      for j = count to 0 step -1
9:          if (Q < Interface[j].Q)
10:             --num
11:         end if
12:     Interface[++num].H = H*
13:     Interface[num].Q = Q
14:     Interface[++num].RC = RC_s
15: while (increaseCeilingH*(RC_s))
16: return Interface,num()
```

Figure 2. The local algorithm.

The algorithm works as shown in Table 3. The results from step 1 are $(Q_1 = 26, H_1 = 47)$, at step 2 $(Q_1, H_1) > (Q_2, H_2)$. So (Q_1, H_1) is redundant (see Definition 1). That is, this interface can be removed according to Lemma 1. For the same reason, (Q_2, H_2) can be removed after step 3. At step 3, the rc_2 is increased directly to 4 according to Lemma 3 since $rc_1 > rc_2$ and $X_1^2 > X_2^2$. At both steps 4 and 5, the ceiling rc_1 is increased by one since $H_i = h_1$ but we increase the ceiling of rc_2 according to Lemma 3.

In summary, the algorithm generates the interface candidates shown in Table 3.

6 Interface Selection

In this section, we consider an optimization problem, called *optimal interface selection* problem, that selects a *system configuration* consisting of a set of subsystem interfaces, one from each subsystem that together minimize system load subject to the schedulability of system.

Section 6.1 presents the Greedy-GB (Greatest Blocking) algorithm, an algorithm that finds an optimal solution

$\mathcal{T}$	C_i	T_i	$\{R_j\}$	$c_{i,j}$
τ_1	8	750	R_2	4
τ_2	50	650	R_1	5
τ_3	10	600	-	0
τ_4	20	500	R_1	20
τ_5	1	165	-	0
τ_6	2	150	-	0

Table 1. Example task set parameters

$Step$	rc_1	rc_2	h_1	h_2	Q_i	H_i
1	4	1	23	47	26	47
2	4	2	23	37	26	37
3	4	4	23	7	26	23
4	5	5	22	6	30	22
5	6	6	20	4	36	20

Table 2. Example algorithm

to this problem. The proposed algorithm basically finds a set of heuristic solutions through a finite number of iteration steps. Section 6.2 shows that an optimal solution exists within the set of such heuristic solutions generated by the Greedy-GB algorithm.

6.1 Description of the Greedy-GB algorithm

The Greedy-GB algorithm relies on a couple of assumptions. It assumes the *refined* property of each subsystem's interface candidate set such that it contains no *redundant* elements (see section 5). Moreover, it assumes that each interface candidate set is sorted in a decreasing order of resource holding time (H_s). In other words, each set is sorted in an increasing order of demand ($Q_s + H_s$). Then, the first candidate has the largest resource holding time and the smallest demand.

Heuristic solution. The Greedy-GB algorithm generates a finite number of heuristic solutions through iteration steps. Each heuristic solution is a set of individual interface candidates of all subsystems, i.e.,

HS_i Let HS_i denote a *heuristic solution* that the Greedy-GB algorithm generates at an i-th iteration step. For notational convenience, we introduce a variable c_k^i to denote an element of HS_i, i.e., $\mathsf{HS}_i = \{c_1^i, \ldots, c_n^i\}$. The variable c_k^i indicates which interface candidate of a subsystem S_k is included in HS_i.

Figure 3 shows an example search space for a system consisting of 3 subsystems, where subsystem S_1 has 3 interface candidates, and two other subsystems S_2 and S_3 have 2 candidates, respectively. Each node in the graph represents a solution candidate, and each number in the node corresponds to an interface candidate index in the

Interface	P	Q	H
1	100	26	23
2	100	30.2	22
3	100	36.5	20

Table 3. Interface candidates.

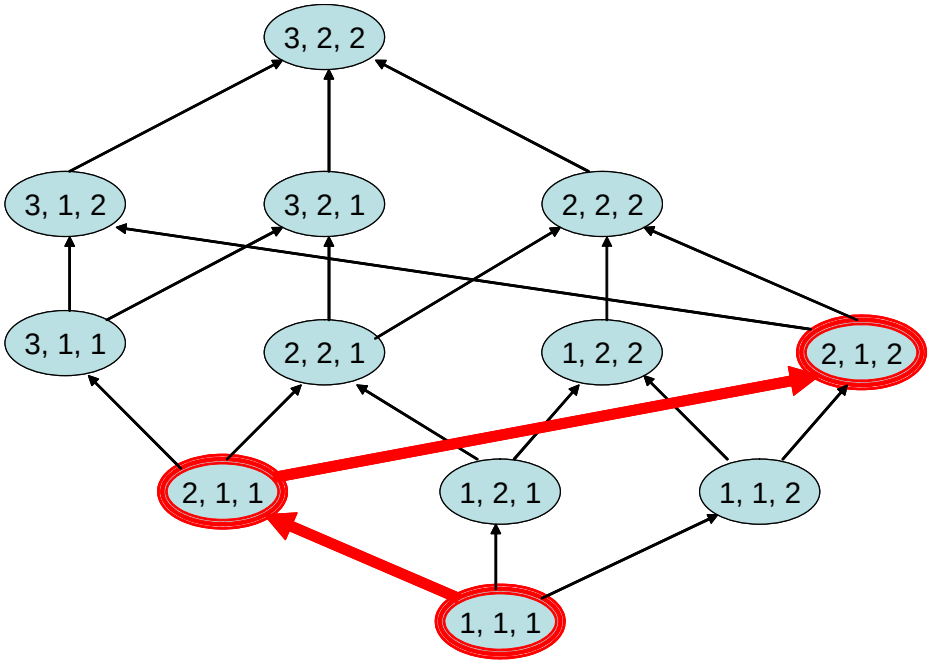

Figure 3. Search space for a system consisting of 3 subsystems.

order of S_1, S_2, and S_3. In the figure, at the second iteration step, the heuristic solution is $\mathsf{HS}_2 = \{2, 1, 2\}$, and the first element of HS_2 is $c_1^2 = 2$.

Initialization. In the beginning, this algorithm generates an initial heuristic solution HS_0 such that it consists of the first interface candidates of all subsystems. In the example shown in Figure 3, $\mathsf{HS}_0 = \{1, 1, 1\}$ (see line 2 of Figure 4).

Iteration step. At an arbitrary i-th iteration step, the Greedy-GB algorithm takes a heuristic solution HS_{i-1} from the previous step and transforms it to another heuristic solution HS_i. Transformation is made to increase only one element of HS_{i-1} in value by one. Let us introduce a variable (δ_i) to state this more formally.

δ_i Let δ_i denote the only single element whose value increases by one between HS_{i-1} and HS_i, i.e.,

$$c_k^i = \begin{cases} c_k^{i-1} + 1 & \text{if } k = \delta_i, \\ c_k^{i-1} & \text{otherwise.} \end{cases} \tag{12}$$

In the example shown in Figure 3, $\delta_1 = 1$.

Let us explain how to determine δ_i at an i-th step. We can potentially increase every elements of HS_{i-1}, and thereby we have at most n candidates for the value of δ_i. Here, we choose one out of at most n candidates such that a resulting HS_i can cause the system load to be minimized.

Let $\mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_{i-1})$ denote the value of $\mathsf{load}_{\mathsf{sys}}$ when a heuristic solution HS_{i-1} is used as a *system interface*. We are now interested in reducing the value of $\mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_{i-1})$. We introduce a variable s_i^* that is useful to explain how to reduce $\mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_{i-1})$.

s_i^* Let s_i^* denote the subsystem $S_{s_i^*}$ that has the largest *processor request bound* among all subsystems. That is, $\mathsf{load_{sys}}(\mathsf{HS}_{i-1}) = \mathsf{load}_{s_i^*}$. We can find such $S_{s_i^*}$ by evaluating the *processor request bound*'s of all subsystems (in line 5 of Figure 4).

By the definition of s_i^*, we can reduce the value of $\mathsf{load_{sys}}(\mathsf{HS}_{i-1})$ by reducing the value of $\mathtt{LBF}_{\mathtt{s}_\mathtt{i}^*}(\mathtt{t})$. There are two potential ways to reduce the value of $\mathtt{LBF}_{\mathtt{s}_\mathtt{i}^*}(\mathtt{t})$. From the definition of $\mathtt{LBF}_{\mathtt{s}}(\mathtt{t})$ in Eq. (6), one is to reduce its maximum blocking time $B_{s_i^*}$ and the other is to reduce the subsystem demands ($\mathtt{DBF}_{s_i^*}(t)$). A key aspect of this algorithm is that it always reduces the blocking time part, but does not reduce the demand part. An intuition behind is as follows: this algorithm starts from the interface candidates that have the smallest demands but the largest resource holding times, respectively. Hence, for each interface candidate, there is no room to further reduce its demand. However, there is a chance to reduce the maximum blocking time $B_{s_i^*}$ of $S_{s_i^*}$. It can be reduced by decreasing the resource holding time of a subsystem $S_{k_i^*}$ that imposes the largest blocking time to the subsystem $S_{s_i^*}$. We define k_i^* in a more detail.

k_i^* Let k_i^* denote the subsystem $s_{k_i^*}$ that imposes the largest blocking time to the subsystem $S_{s_i^*}$, i.e., $B_{s_i^*} = H_{k_i^*} = \max\{H_j \mid \text{for all} H_s \in LPS(s_i^*)\}$[1], where $LPS(i)$ is a set of lower-priority subsystems of $S_{s_i^*}$. We can find such $S_{k_i^*}$ easily by looking at the resource holding times of all lower-priority subsystems of $S_{s_i^*}$ (in ine 6 of Figure 4).

When such $S_{k_i^*}$ is found, it then checks whether the resource holding time of $S_{k_i^*}$ can be further reduced (in line 7 of Figure 4). If so, it is reduced (in line 8), and HS_{i-1} becomes to HS_i (in line 9). That is, $\delta_i = k_i^*$.

Iteration termination. The above iteration process terminates when the blocking time B_{s*} of subsystem $S_{s_i^*}$ cannot be reduced further. The algorithm then finds the smallest value of $\mathsf{load_{sys}}$ out of the values saved during the iteration, and it returns a set of interfaces corresponding to the smallest value.

Complexity of the algorithm. During an i-th iteration, the algorithm only increases the interface candidate index of a subsystem S_{δ_i}. Then, it can repeat $\mathcal{O}(n * m')$ iterations, where n is the number of subsystems and m' is the greatest number of interface candidates of a subsystem among all.

6.2 Correctness of the Greedy-GB algorithm

In this section, we show that the Greedy-GB algorithm produces a set of heuristic solutions that contains an optimal solution. We first present notations that are useful to prove the correctness of the algorithm.

$\mathcal{AS}$. We consider the entire search space of the optimal interface selection problem. It contains all possible subsystem interfaces comprising a system configuration, and let $\mathcal{AS}$ denote it, i.e.,

$$\mathcal{AS} = IC_1 \times \cdots \times IC_n. \tag{13}$$

In the example shown in Figure 3, the entire solution space ($\mathcal{AS}$) has 12 elements.

We present some notations to denote the properties of the Greedy-GB algorithm at an arbitrary i-th iteration step.

[1]If more tan one lower priority subsystem impose same maximum blocking on Ss_i^* then we select the one with lowest priority.

- IC_s is an array of interface candidates of subsystem S_s, sorted in a decreasing order of H_s.
- ici_s is an index to IC_s of subsystem S_s
- $\mathcal{I}$ is a set of interfaces $\{I_s\}$, each of which indicated by ici_s
- subsystemWithMaxLoad() returns the subsystem S_{s^*} that has the greatest *processor request bound* among all subsystems, i.e., $\mathsf{load}_{\mathsf{sys}} = \alpha_{s^*}$.
- maxBlockingSubsystemToSysload(s^*) returns a subsystem S_{k^*} that produces the greatest blocking time to a subsystem S_{s^*}. Note that S_{s^*} determines the system load.

```
1:  for all  S_s ∈ 𝒮
2:       ici_s = 1;  I_s = IC_s[ici_s]
3:  load*_sys = 1.0;  ℐ* = ℐ
4:  do
5:       s* = subsystemWithMaxLoad()
6:       k* = maxBlockingSubsystemToSysload(s*)
7:       if (ici_k* can increase by one)
8:            ici_k* = ici_k* + 1
9:            I_k* = IC_k*[ici_k*]
10:           compute load_sys according to Eq. (10)
11:           if (load_sys < load*_sys)
12:                load*_sys = load_sys
13:                ℐ* = ℐ
14:      else
15:           return ℐ* (that determines load*_sys)
16:  until(true)
```

Figure 4. The Greedy-GB algorithm.

$\widehat{IC}_k^i$. In the beginning, the Greedy-GB algorithm has the entire search space ($\mathcal{AS}$) to explore. Basically, this algorithm gradually reduces a remaining search space to explore during iteration. For notation convenience, we introduce a variable ($\widehat{IC}_k^i$) to indicate the remaining interface candidates of a subsystem S_k to explore. By definition, c_k^i indicates which interface candidate of a subsystem S_k is selected by HS_i. This algorithm continues exploration from the interface candidate indicated by c_k^i from the end of an i-th step. Then, $\widehat{IC}_k^i$ is defined as

$$\widehat{IC}_k^i = \{c_k^i, \ldots, max_k\} \text{ for all } k = 1, \ldots, n. \tag{14}$$

where max_k is the number of interface.

In the example shown in Figure 3, $\widehat{IC}_1^1 = \{2, 3\}$.

XP_i. Let us define XP_i to denote the search space remaining to explore after the end of an i-th iteration step. Note that such a remaining search space does not have to include the solution candidate HS_i chosen at the i-th step. Then, XP_i is defined as

$$\mathsf{XP}_i = (\widehat{IC}_1^i \times \cdots \times \widehat{IC}_n^i) \setminus \mathsf{HS}_i. \tag{15}$$

RM_i In essence, the Greedy-GB algorithm gradually decreases a remaining search space during iteration. That is, at an i-th step, it keeps reducing XP_{i-1} to XP_i, where $\mathsf{XP}_i \subset \mathsf{XP}_{i-1}$. Let RM_i denote a set of interface settings that is excluded from XP_{i-1} at the i-th step. Note that at the i-th step, the interface candidate of a subsystem S_{δ_i} changes from $c_{\delta_i}^{i-1}$ to $c_{\delta_i}^{i}$. Then, a subset of XP_i that contains the value of $c_{\delta_i}^{i-1}$, is excluded at the i-th step. RM_i is defined as

$$\mathsf{RM}_i = (\widehat{IC}_1^{(i-1)*} \times \cdots \times \widehat{IC}_n^{(i-1)*}) \setminus \{\mathsf{HS}_{i-1}\}, \text{ where} \tag{16}$$

$$\widehat{IC}_k^{(i-1)*} = \begin{cases} \{c_k^{i-1}\} & \text{if } k = \delta_i, \\ \widehat{IC}_k^i & \text{otherwise.} \end{cases} \tag{17}$$

In the example shown in Figure 3, $\mathsf{RM}_1 = \{\{1,2,1\},\{1,2,2\},\{1,1,2\}\}$.

AH_i Let AH_i represents a set of heuristic solutions that the Greedy-GB algorithm selects from the first step through to an i-th step, i.e.,

$$\mathsf{AH}_i = \{\mathsf{HS}_1, \ldots, \mathsf{HS}_i\}. \tag{18}$$

AR_i Let AR_i represents a set of interface candidates that the Greedy-GB algorithm excludes from the first step through to an i-th step, i.e.,

$$\mathsf{AR}_i = \mathsf{RM}_{(i-1)} \cup \mathsf{RM}_i, \tag{19}$$

where $\mathsf{AR}_0 = \phi$.

We define partial ordering between solution candidates as follows:

Definition 2 *A solution candidate* $\mathsf{sc} = \{c_1, \ldots, c_n\}$ *is said to be* strictly precedent *of another solution candidate* $\mathsf{sc}' = \{c'_1, \ldots, c'_n\}$ *(denoted as* $\mathsf{sc} \prec \mathsf{sc}'$*) if* $c_j < c'_j$ *for some* j *and* $c_k \leq c'_k$ *for all* k*, where* $1 \leq j, k \leq n$*.*

As an example, $\{1,1,1\} \prec \{1,2,1\}$.

The following lemma states that when the algorithm excludes a set of solution candidates from further exploration at an arbitrary i-th step, a set of such excluded solution candidates does not contain an optimal solution.

Lemma 5 *At an arbitrary* i*-th iteration step, the Greedy-GB algorithm excludes a set of solution candidates (*RM_i*), and any excluded solution candidate* $r \in \mathsf{RM}_i$ *does not yield a smaller system load than that by* HS_{i-1}*, i.e.,*

$$\forall r \in \mathsf{RM}_i \ \mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_{i-1}) \leq \mathsf{load}_{\mathsf{sys}}(r). \tag{20}$$

Proof As explained in Section 6.1, there are two potential ways to reduce the value of $\mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_{i-1})$ at the i-th step. One is to reduce the demand of the subsystem $S_{s_i^*}$ (i.e., $\mathtt{DBF}_{\mathtt{s}_\mathtt{i}^*}(\mathtt{t})$), and the other is to reduce its maximum blocking time $B_{s_i^*}$.

Firstly, we wish to show that the demand of $S_{s_i^*}$ does not decrease when we transform the heuristic solution of HS_{i-1} to any solution candidate $r \in \mathsf{RM}_i$. Note that each interface candidate set is sorted in an increasing order of resource requirement budget (Q). One can easily see that $\mathsf{HS}_{i-1} \prec r$. Then, it follows that $\mathtt{DBF}_{\mathtt{s}_\mathtt{i}^*}(\mathtt{t})$ never decreases when HS_{i-1} changes to r.

Secondly, we wish to show that when we change the heuristic solution of HS_{i-1} to any solution candidate $r \in \mathsf{RM}_i$, $B_{s_i^*}$ does not decrease. As shown in line 6 in Figure 4, the Greedy-GB algorithm finds the subsystem S_{δ_i} that generates the maximum blocking time to for subsystem $S_{s_i^*}$. Then, the algorithm increases $c_{\delta_i}^{i-1}$ by one, if possible, to decrease $B_{s_i^*}$. However, by definition, for all elements r of RM_i, the element for the subsystem S_{δ_i} has the value of $c_{\delta_i}^{i-1}$, rather than the value of $c_{\delta_i}^{i}$. This means that $B_{s_i^*}$ never decreases when we change HS_{i-1} to r. □

The following lemma states that when the algorithm terminates at an arbitrary f-th step, a set of remaining solution candidates does not contain an optimal solution.

Lemma 6 *When the Greedy-GB algorithm terminates at an arbitrary f-th step, any remaining solution candidate ($\mathsf{xp} \in \mathsf{XP}_f$) does not yield a smaller system load than HS_f does, i.e.,*

$$\forall \mathsf{xp} \in \mathsf{XP}_f \ \mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_f) \leq \mathsf{load}_{\mathsf{sys}}(\mathsf{xp}). \tag{21}$$

Proof As explained in the proof of lemma 5, there are two ways to reduce $\mathsf{load}_{\mathsf{sys}}$ (i.e., $\mathtt{LBF}_{\mathtt{s}_\mathtt{f}^*}(\mathtt{t})$).

One is to reduce the demand of the subsystem $S_{s_f^*}$ (i.e., $\mathtt{DBF}_{\mathtt{s}_\mathtt{f}^*}(\mathtt{t})$ in Eq. (7)). However, it does not decrease, since $\mathsf{HS}_f \prec \mathsf{xp}$ for all $\mathsf{xp} \in \mathsf{XP}_f$.

The other is to reduce the maximum blocking time ($B_{s_f^*}$). In fact, the Greedy-GB algorithm terminates at the f-th step because there is no way to decrease $B_{s_f^*}$. That is, B_f does not decrease when HS_f changes to any xp. □

The following lemma states that at any i-th step, the remaining search space to explore decreases by ($\mathsf{RM}_i \cup \{\mathsf{HS}_i\}$).

Lemma 7 *At an arbitrary i-th iteration step,*

$$\mathsf{XP}_i = \mathsf{XP}_{i-1} \setminus (\mathsf{RM}_i \cup \{\mathsf{HS}_i\}). \tag{22}$$

Proof The Greedy-GB algorithm transforms HS_{i-1} to HS_i at an i-th step by increasing the value of its δ_i-th element. Then, we have

$$\widehat{IC}_k^i = \begin{cases} \widehat{IC}_k^{i-1} \setminus \{c_k^{i-1}\} & \text{if } k = \delta_i, \\ \widehat{IC}_k^{i-1} & \text{otherwise.} \end{cases} \tag{23}$$

Without loss of generality, we assume that $\delta_i = 1$. For notational convenience, let $\mathsf{XP}_i^* = \mathsf{XP}_i \cup \{\mathsf{HS}_i\}$, and $\mathsf{RM}_i^* = \mathsf{RM}_i \cup \{\mathsf{HS}_i\}$. Then, we have

$$\begin{aligned} \mathsf{XP}_i^* &= \widehat{IC}_1^i \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i \\ &= \left(\widehat{IC}_1^{i-1} \setminus \{c_1^{i-1}\}\right) \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i \\ &= \left(\widehat{IC}_1^{i-1} \times \widehat{IC}_2^{i-1} \times \cdots \times \widehat{IC}_n^{i-1}\right) \setminus \\ &\quad \left(\{c_1^{i-1}\} \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i\right) \\ &= \mathsf{XP}_{i-1}^* \setminus \mathsf{RM}_i^* \\ &= \left(XP_{i-1} \cup \{\mathsf{HS}_{i-1}\}\right) \setminus \left(\mathsf{RM}_i \cup \{\mathsf{HS}_{i-1}\}\right) \\ &= XP_{i-1} \setminus \mathsf{RM}_i \ . \end{aligned} \tag{24}$$

That is, considering $XP_i^* = XP_i \cup \{\mathsf{HS}_i\}$, it follows

$$XP_i = XP_{i-1} \setminus (\mathsf{RM}_i \cup \{\mathsf{HS}_i\}) \ . \tag{25}$$

□

The following lemma states that at any i-th iteration step, the entire search space can be divided into a set of explored candidates (AH_i), a set of excluded candidates (AR_i), and a set of remaining candidates to explore (XP_i).

Lemma 8 *At an arbitrary i-th step, the sets of AR_i, AH_i, and XP_i include all possible solution candidates, i.e.,*

$$\mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i = \mathcal{AS} \tag{26}$$

Proof We will prove this lemma by using mathematical induction. As a base step, we wish to show Eq. (26) is true, when $i = 1$. Note that $\mathsf{AR}_0 = \phi$ and $\mathsf{AH}_0 = \{\mathsf{HS}_0\}$. In addition, $\mathsf{XP}_0 = \mathcal{AS} \setminus \mathsf{HS}_0$, according to Eq. (15). It follows that $\mathsf{AR}_0 \cup \mathsf{AH}_0 \cup \mathsf{XP}_0 = AP$.

We assume that Eq. (26) is true at the i-th iteration step of the Greedy-GB algorithm. We then wish to prove that it also holds at the $(i+1)$-th step, i.e.,

$$\mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i = \mathsf{AR}_{i+1} \cup \mathsf{AH}_{i+1} \cup \mathsf{XP}_{i+1}. \tag{27}$$

According to the definitions AH_{i+1}, AR_{i+1}, and XP_{i+1} (see Eq. (18), (19) and (22)), we can rewrite the right-hand side of Eq. (27) as follows:

$$\begin{aligned} & \mathsf{AR}_{i+1} \cup \mathsf{AH}_{i+1} \cup \mathsf{XP}_{i+1} \\ = \; & \Big(\mathsf{AR}_i \cup \mathsf{RM}_{i+1}\Big) \cup \Big(\mathsf{AH}_i \cup \{\mathsf{HS}_{i+1}\}\Big) \cup \\ & \Big(\mathsf{XP}_i \setminus (\mathsf{RM}_{i+1} \cup \{\mathsf{HS}_{i+1}\})\Big) \\ = \; & \mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i \; . \end{aligned}$$

□

The following theorem states that the Greedy-GB algorithm produces a set of heuristic solutions, which must contain an optimal solution.

Theorem 9 *When the Greedy-GB algorithm terminates at the f-th step, a set of heuristic solutions (AH_f) includes an optimal solution.*

Proof Let opt denote an optimal solution. We prove this theorem by contradiction, i.e., by showing that $opt \notin \mathsf{AR}_f$ and $opt \notin \mathsf{XP}_f$.

Suppose $opt \in \mathsf{AR}_f$. Then, by definition, there should exist RM_i such that $opt \in \mathsf{RM}_i$ for an arbitrary $i \leq f$. According to Lemma 5, $\mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_{i-1}) < \mathsf{load}_{\mathsf{sys}}(opt)$, which contradicts the definition of opt. Hence, $opt \notin \mathsf{AR}_f$.

Suppose $opt \in \mathsf{XP}_f$. Then, according to Lemma 6, it should be $\mathsf{load}_{\mathsf{sys}}(\mathsf{HS}_f) < \mathsf{load}_{\mathsf{sys}}(opt)$, which contradicts the definition of opt as well. Hence, $opt \notin \mathsf{AR}_f$.

According to Lemma 8, it follows that $opt \in \mathsf{HS}_f$. □

7 Conclusion

When subsystems share logical resources in a hierarchical scheduling framework, they can block each other. In particular, when a budget expiry problem exists, such blocking can imposes extra resource demands. However, simply minimizing the blocking times (or resource holding times) of subsystems may result in increase on the system load; in this paper, we introduced such a tradeoff between reducing the resource holding times of subsystems and increasing the system load. Given such a tradeoff, we formulated the system load minimization problem, and

presented a two-step approach to the problem. As a first-step, each subsystem generates a set of solution candidates, and in the second step, the system selects one of the candidates of each subsystem in order to constitute an optimal solution. We presented efficient algorithms and proved their correctness for both steps.

In this paper, we only consider FPS. We plan to extend our framework to EDF scheduling. Another direction for future work is to extend our framework to other hierarchical synchronization protocols, such as SIRAP [3] and BROE [9].

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *EMSOFT '04*, 2004.

[2] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *EMSOFT'07*, 2007.

[4] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *Proceedings of the 15th International Workshop on Parallel and Distributed Real-Time Systems*, Long Beach, CA, March 2007.

[5] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *RTSS*, 2005.

[6] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *RTSS*, 2005.

[7] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS '97*, 1997.

[8] X. A. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *RTSS*, 2002.

[9] N. Fisher, M. Bertogna, and S. Baruah. The design of an edf-scheduled resource-sharing open environment. In *RTSS*, 2007.

[10] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in edf-scheduled systems. In *RTAS*, 2007.

[11] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *RTSS*, 1999.

[12] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS*, 1989.

[13] G. Lipari and S. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *RTAS '00*, 2000.

[14] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *ECRTS*, 2003.

[15] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard-real-time environments. In *RTSS '00*, 2000.

[16] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS '01*, 2001.

[17] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, 1988.

[18] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierar hical fixed-priority scheduling. In *ECRTS '02*, 2002.

[19] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation*, pages 909–916, Cambridge, MA, USA, November 1987.

[20] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS '03*, 2003.

[21] I. Shin and I. Lee. Compositional real-time scheduling framework. In *RTSS '04*, 2004.