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 algorithms that (1) facilitate system-independent synthesis of timing-interfaces for subsystems and (2) system-level selection of interfaces to minimize CPU load. The results presented are developed for hierarchical fixed-priority scheduling of subsystems that may share logical resources (i.e., semaphores). We show that the use of shared resources results in a tradeoff problem, where resource locking times can be traded for CPU allocation, complicating the problem of finding the optimal interface configuration subject to scheduability.

This paper presents a methodology where such a tradeoff can be effectively explored. It first synthesizes a bounded set of interface-candidates for each subsystem, independently of the final system, such that the set contains the interface that minimizes system load for any given system. Then, integrating subsystems into a system, it finds the optimal selection of interfaces. Our algorithms have linear complexity to the number of tasks involved. Thus, our approach is highly suitable for adaptable and reconfigurable systems.

1 Introduction

Hierarchical scheduling has emerged as a promising vehicle simplifying the development of complex real-time software systems. Hierarchical scheduling frameworks (HSFs) provide an effective mechanism for achieving temporal partitioning, making it easier to enforce the principle of separation of concerns in the design and analysis of real-time systems. HSFs allow hierarchical CPU sharing among subsystems (applications). The whole CPU is available and shared among subsystems. Subsequently, each subsystem’s allocated CPU-share is divided among its internal tasks by the usage of an internal scheduler.

Substantial studies [8, 13, 15, 17, 11, 20, 16, 1, 10, 6, 21, 9] have been introduced for the schedulability analysis of HSFs, where subsystems are independent. For dependent subsystems, synchronization protocols [7, 3, 12] have been proposed for arbitrating accesses to logical resources (i.e., semaphore) across subsystems in HSFs. There have been a few studies [20, 10] on the system load minimization problem, which finds the minimum collective CPU requirement (i.e., system load) necessary to guarantee the schedulability of an entire HSF. However, this problem has not been addressed taking into account global (logical) resource sharing (across subsystems).

*The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.
The difficulty of finding the minimum system load substantially grows with the presence of global sharing of logical resources, in comparison to without it. Without it, it is a straightforward bottom-up process; individual subsystems develop their timing-interfaces [20], describing their minimum CPU requirements needed to ensure schedulability, and individual subsystem interfaces can easily be combined to determine the minimum system load that guarantees the schedulability of an entire HSF. However, global resource sharing produces interference among subsystems, complicating the process of finding subsystem interfaces that impose the minimum CPU requirements into the system load.

An inherent feature with global resource sharing is that a subsystem can be blocked in accessing a global shared resource, if there is another subsystem locking the resource at the moment. Such blocking imposes more CPU demands, resulting in an increase of the system load. Therefore, subsystems can reduce their resource locking time, for example, using the mechanism presented in [5], in order to potentially reduce the blocking of other subsystems towards decrease of the system load. However, in doing so, this paper presents an inconsistent consequence of reducing resource locking time; it can increase the CPU demands of the subsystem itself (locking the resource), subsequently increasing the system load. Hence, this paper introduces a potentially contradicting effect of reducing resource locking time on the system load, and it entails methods that can effectively explore such a tradeoff.

In this paper, we consider a two-step approach towards the system load minimization problem. In the first step, each subsystem generates its own interface candidates in isolation, investigating the intra-subsystem aspect of the tradeoff. In the second step, putting all subsystems together on system-level, interfaces of all subsystems are selected from their own candidates to find the minimum resulting system load, examining the inter-subsystem aspect of the tradeoff. For the first step, we present an algorithm that derives a bounded number of interface candidates for each subsystem such that it is guaranteed to carry an interface candidate that constitutes the minimum system load no matter which other subsystems it will be later integrated with. The first step allows the interface candidates of subsystems to be developed independently, making it also suitable for open environments [8], requiring no knowledge of other subsystems. For the second step, we present another algorithm that determines optimal interface selection to find the minimum system load. The complexity of both algorithms is very low ($O(n)$), making the approach good for execution during run-time, e.g., suitable for adaptable and reconfigurable systems.

In the remaining paper, Section 2 presents related work, followed by system model and background in Section 3. Section 4 presents schedulability analysis in our HSF, followed by problem formulation and solution outline in Section 5. Section 6 addresses the first step of the two-step approach; efficiently generating interface candidates, and Section 7 resolves the second step finding an optimal solution out of the candidates. Finally, Section 9 concludes.

2 Related work

This section presents related work in the areas of HSFs as well as synchronization protocols.

Hierarchical scheduling. The HSF for real-time systems, originating in open systems [8] in the late 1990’s, has been receiving an increasing research attention. Since Deng and Liu [8] introduced a two-level HSF, its schedulability has been analyzed under fixed-priority global scheduling [13] and under Earliest Deadline First (EDF) based global scheduling [15]. Mok et al. [17, 11] proposed the bounded-delay virtual processor model to achieve a clean separation in a multi-level HSF. In addition, Shin and Lee [20] introduced the periodic virtual processor model (to characterize the periodic CPU allocation behaviour), and many studies have been proposed on schedulability analysis with this model.
under fixed-priority scheduling [1, 16, 6] and under EDF scheduling [20, 21]. More recently, Easwaran et al. [9] introduced Explicit Deadline Periodic (EDP) virtual processor model. However, a common assumption shared by all above studies is that tasks are independent.

**Synchronization.** Many synchronization protocols have been introduced for arbitrating accesses to shared logical resources addressing the priority inversion problem, including Priority Inheritance Protocol (PIP) [19], Priority Ceiling Protocol (PCP) [18], and Stack Resource Policy (SRP) [2]. There have been studies on supporting resource sharing within subsystems [1, 13] in HSFs. For supporting global resource sharing across subsystems, two protocols have been proposed for periodic virtual processor model (or periodic server) based HSFs on the basis of an overrun mechanism [7] and skipping [3], and another protocol [12] for bounded-delay virtual processor model based HSFs. Bertogna et al. [5] addressed the problem of minimizing the resource holding time under SRP. In summary, compared to the work in this paper, none of the above approaches have addressed the tradeoff between how long subsystems can lock shared resources and the resulting CPU requirement required in guaranteeing schedulability.

### 3 System model and background

A Hierarchical Scheduling Framework (HSF) is introduced to support CPU resource sharing among applications (subsystems) under different scheduling services. In this paper, we are considering a two-level HSF, where the system-level global scheduler allocates CPU resources to subsystems, and the subsystem-level local schedulers subsequently schedule CPU resources to their internal tasks. This framework also allows logical resource sharing between tasks in a mutually exclusive manner.

#### 3.1 Virtual processor models

The notion of real-time virtual processor model was first introduced by Mok et al. [17] to characterize the CPU allocations that a parent node provides to a child node in a HSF. The CPU supply refers to the amounts of CPU allocations that a virtual processor 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 processor model \( \Gamma(P, Q) \) to specify periodic CPU allocations, where \( P \) is a period \( (P > 0) \) and \( Q \) is a periodic allocation time \((0 < Q \leq P)\). The supply bound function \( \text{sbf}_\Gamma(t) \) of \( \Gamma(P, Q) \) was given in [20] that computes the minimum possible CPU supply for every interval length \( t \) as follows:

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

where \( k = \max \left( \left\lfloor \frac{t - (P - Q)}{P} \right\rfloor, 1 \right) \).

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 \( \text{sbf}_\Gamma(t) \). Note that BD in Figure 1 represents the longest possible blackout duration during which the periodic virtual processor model may provide no resource allocation at all.

The service time of a resource has been defined as the duration that it takes for the resource to provide a resource supply. For a periodic resource \( \Gamma(P, Q) \), we define a service time bound function \( \text{tbf}_\Gamma(t) \) [20].
to calculate the maximum service time of $\Gamma$ required for a $t$-time-unit resource supply as follows:

$$\text{tbf}_\Gamma(t) = (P - Q) + P \cdot \left\lfloor \frac{t}{Q} \right\rfloor + \epsilon_t,$$

where

$$\epsilon_t = \begin{cases} 
P - Q + t - Q\left\lfloor \frac{t}{Q} \right\rfloor & \text{if } \left( t - Q\left\lfloor \frac{t}{Q} \right\rfloor > 0 \right) \\
0 & \text{otherwise}
\end{cases}$$

### 3.2 System model

We consider a deadline-constrained sporadic task model $\tau_i(T_i, C_i, D_i, \{c_{i,j}\})$ where $T_i$ is a minimum separation time between its successive jobs, $C_i$ is a worst-case execution time requirement, $D_i$ is a relative deadline ($C_i \leq D_i \leq T_i$), and each element $c_{i,j}$ in $\{c_{i,j}\}$ is a critical section execution time that represents a worst-case execution time requirement within a critical section of a global shared resource $R_j$. We assume that all tasks are assigned unique static priorities and are sorted according to their priorities in the order of increasing priority. Without loss of generality, we assume that the priority of a task is equal to the task ID number after sorting, and the greater a task ID number is, the higher its priority is.

A subsystem $S_s \in S$, where $S$ is the set representing the whole system of subsystems, is characterized by $\langle T_s, \mathcal{R}C_s \rangle$, where $T_s$ is a task set and $\mathcal{R}C_s$ is a set of internal resource ceilings of the global shared logical resources. We will explain the resource ceilings in Section 3.3. We assume that each subsystem has a unique static priority and subsystems are sorted in an increasing order of priority, as is the case...
with tasks. We also assume that each subsystem $S_i$ has a local Fixed-Priority Scheduler (FPS) and the system has a global FPS.

Let us define a timing-interface of a subsystem $S_i$ such that it specifies the collective real-time requirements of $S_i$. The subsystem interface is defined as $(P_s, Q_s, X_s)$, where $P_s$ is a period, $Q_s$ is a budget that represents an execution time requirement, and $X_s$ is a maximum critical section execution time of all global logical resources accessed by $S_i$. We note that $X_s$ is similar to the concept of resource holding time (RHT) in [5], however, developed for a different virtual-processor model. RHT in [5] is developed for a dedicated processor model\(^1\) (or a fractional processor model [17]), where subsystems do not preempt each other. However, our HSF is based on a time-shared (partitioned) processor model [20], where subsystem-level preemptions can take place. Therefore, $X_s$ does not represent RHT in our HSF, but indicates the worst-case execution time requirement that $S_i$ demands inside a critical section. In our periodic processor model-based HSF, when $S_i$ has the critical section execution time of $X_s$, its resource holding time is equal to the service time of $X_s$, i.e., $tbf_{(P_s, Q_s)}(X_s)$. We will explain later how to derive the values of $P_s, Q_s$ and $X_s$ for a given subsystem $S_i$.

### 3.3 Stack Resource Policy (SRP)

In this paper, we consider the SRP protocol [2] for arbitrating accesses to shared logical resources. Considering that the protocol was developed without taking hierarchical scheduling into account, we generalize its terminologies for hierarchical scheduling.

- **Resource ceiling.** Each global shared resource $R_j$ is associated with two types of resource ceilings; an internal resource ceiling ($rc_j$) for local scheduling and an external resource ceiling ($RX_s$) for global scheduling. They are defined as $rc_j = \max\{i | \tau_i \in T_s \text{ accesses } R_j\}$ and $RX_s = \max\{s | S_s \text{ accesses } R_j\}$.

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

Under SRP, a task $\tau_k$ can preempt the currently executing task $\tau_i$ (even inside a critical section) within the same subsystem, only if the priority of $\tau_k$ is greater than its corresponding subsystem ceiling. The same reasoning can be made for subsystems from a global scheduling point of view.

Given a subsystem $S_i$, let us consider how to derive the value of its critical section execution time ($X_s$). Basically, $X_s$ represents a worst-case CPU demand that internal tasks of $S_i$ may collectively request inside any critical section. Note that any task $\tau_i$ accessing a resource $R_j$ can be preempted by tasks with priority higher than the internal ceiling of $R_j$. From the viewpoint of $S_i$, let $w_j$ denote the maximum collective CPU demand necessary to complete an access of any internal task to $R_j$. Then, $w_j$ can be computed through iterative process as follows (similarly to [5]):

$$w_j^{(m+1)} = cx_j + \sum_{k=rc_j+1}^{n} \left\lfloor \frac{w_j^{(m)}}{T_k} \right\rfloor \cdot C_k,$$

(3)

where $cx_j = \max\{c_{i,j}\}$ for all tasks $\tau_i$ accessing resource $R_j$ and $n$ is the number of tasks within the subsystem. The recurrence relation given by Eq. (3) starts with $w_j^{(0)} = cx_j$ and ends when $w_j^{(m+1)} = w_j^{(m)}$ or when $w_j^{(m+1)} > D_i^s$, where $D_i^s$ is the smallest deadline of tasks $\tau_i$ accessing $R_j$. If $w_j^{(m+1)} > D_i^s$, no task $\tau_i$ is guaranteed to be schedulable, and subsequently neither is its subsystem $S_i$.

\(^1\)A processor is said to be dedicated to a subsystem, if the subsystem exclusively utilizes the processor with no other subsystems.
Then, \( X_s = \max \{ w_j \mid \forall R_j \in \mathcal{R}_s \} \), where \( \mathcal{R}_s \) is a set of global shared resources accessed by \( S_s \).

## 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 \((P_s, Q_s, X_s)\). 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 task \( \tau_i \) of a subsystem \( S_s \) is executing within the critical section of a global shared resource \( R_j \). If another task \( \tau_k \), belonging to another subsystem, is waiting for the same resource \( R_j \), this task must wait until \( S_s \) is replenished so \( \tau_i \) can continue to execute and finally release the lock on resource \( R_j \). This waiting time exposed to \( \tau_k \) can be potentially very long, causing \( \tau_k \) to miss its deadline.

In this paper, we consider a mechanism based on overrun [7] that works as follows; when the budget of the subsystem \( S_s \) expires and \( S_s \) has a task \( \tau_i \) that is still locking a global shared resource, the task \( \tau_i \) continues its execution until it releases the locked resource. The extra time that \( \tau_i \) needs to execute after the budget of \( S_s \) expires is denoted as overrun time \( \theta_s \). The maximum \( \theta_s \) occurs when \( \tau_i \) locks a resource such that \( S_s \) requests a maximum critical section execution time \((X_s)\) just before its budget \((Q_s)\) expires.

### 4.2 Schedulability analysis

In this paper, we use HSRP [7] for resource synchronization in HSF. Schedulability analysis under global and local FPS with the overrun mechanism is presented in [7]. However, the presented approach is not suitable for open environments because the schedulability analysis of an internal task within a subsystem requires information of all the other subsystems. 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 \( \text{dbf}_{FP}(i, t) \) denote the demand bound function of a task \( \tau_i \) under FPS [14], i.e.,

\[
\text{dbf}_{FP}(i, t) = C_i + \sum_{\tau_k \in \text{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k,
\]

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

\[
\forall \tau_i, 0 < \exists t \leq D_i \quad \text{dbf}_{FP}(i, t) + b_i \leq \text{sbf}(t),
\]

where \( b_i \) is the maximum blocking (i.e., extra CPU demand) imposed to a task \( \tau_i \) when \( \tau_i \) is blocked by lower priority tasks that are accessing resources with ceiling greater than or equal to the priority of \( \tau_i \), and \( \text{sbf}(t) \) is the supply bound function.
Subsystem interface. We now explain how to derive the budget $Q_s$ of the subsystem interface. Given $S_s$, $RC_s$, and $P_s$, let calculateBudget($S_s$, $P_s$, $RC_s$) 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, $Q_s = \text{calculateBudget}(S_s, P_s, RC_s)$.

Global schedulability analysis. Under global FPS scheduling, we present the subsystem load bound function as follows (on the basis of a similar reasoning of Eq. (4)):

$$LBF_s(t) = DBF_s(t) + B_s,$$

where

$$DBF_s(t) = (Q_s + O_s(t)) + \sum_{S_k \in HPS(S_s)} \left\lceil \frac{t}{P_k} \right\rceil (Q_k + O_k(t)),$$

where $HPS(S_s) = \{ S_j | j > s \}$ and $O_k(t) = X_k$ and $O_s(t) = X_s$ for $t \geq 0$. Let $B_s$ denote the maximum blocking (i.e., extra CPU demand) imposed to a subsystem $S_s$, when it is blocked by lower-priority subsystems,

$$B_s = \max\{ X_j | S_j \in LPS(S_s) \},$$

where $LPS(S_s) = \{ S_j | j < s \}$.

A global schedulability condition under FPS is then

$$\forall S_s, 0 < \exists t \leq P_s LBF_s(t) \leq t$$

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 ($\alpha_s$) as

$$\alpha_s = \min_{0 < t \leq P_s} \left\{ \frac{LBF_s(t)}{t} | LBF_s(t) \leq t \right\}.$$  

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

$$load_{sys} = \max_{\forall S_s \in S }\{ \alpha_s \}.$$  

Note that $\alpha_s$ is the smallest fraction of the CPU resources that is required to schedule a subsystem $S_s$ (satisfying Eq. (9)) assuming that the global resource supply function is $\alpha t$. For example, consider a system $S$ that consists of two subsystems; $S_1$ that has interface $(10, 1, 0.5)$ and $S_2$ $(48, 1, 1)$. To guarantee the schedulability of $S_1$ and $S_2$ then $\alpha_1 = 0.25$ and $\alpha_2 = 0.198$. Then $load_{sys} = \alpha_1 = 0.25$, which can schedule both $S_1$ and $S_2$.

5 Problem formulation and solution outline

In this paper, we aim at maintaining the system load as low as possible while satisfying the real-time requirements of all subsystems in the presence of global resource sharing. To achieve this, we address the problem of developing the interfaces $(P_s, Q_s, X_s)$ of all subsystems $S_s$. In particular, assuming $P_s$ is given, we focus on determining $Q_s$ and $X_s$ such that a resulting system load ($load_{sys}$) is minimized.
subject to the schedulability of all subsystems. It is suggested from Eqs. (6) and (11) that $\text{load}_{\text{sys}}$ can be minimized by reducing $Q_s$ and $X_s$ for all subsystem $S_s$.

A recent study [5] introduced a method to reduce $X_i$. According to Eq. (3), the value of $X_s$ can decrease, when it has less interference (i.e., the summation part of Eq. (3)) from the tasks $\tau_k$ with priorities greater than the ceiling of a resource $R_j$ (i.e., $k > r_{cj}$). Such interference can be reduced by allowing fewer tasks to preempt inside the critical section of $R_j$. As proposed by [5], the ceiling of $R_j$ can be increased to its greatest possible value in order to allow no preemption inside the critical section. This way, $X_s$ can be minimized.

In this paper, we show that achieving the minimum $X_s$ of all subsystems $S_s$ does not simply produce the minimum system load, since minimizing $X_s$ may end up with a larger $Q_s$. To explain why this happens, let us assume that for a resource $R_j$, its ceiling $r_{cj}$ is $i - 1$. In this case, a task $\tau_i$ can preempt any job that is executing inside the critical section of $R_j$. Now, suppose $r_{cj}$ is increased to $i$. Then, $\tau_i$ is no longer able to preempt any job that is accessing $R_j$, and it needs to be blocked. Then, the blocking ($b_i$) of $\tau_i$ can potentially increase, and, according to Eq. (5), this may require more CPU supply (i.e., $Q_s$). Figure 2 illustrates a tradeoff between decreasing $X_s$ and increasing $Q_s$ with an example subsystem $S_s$, where $S_s$ includes 7 internal tasks and accesses 3 global resources. In the figure, each point represents a possible pair of $(X_s, Q_s)$, and the line shows the tradeoff.

In addition to such a tradeoff, there is another factor that complicates the system load minimization problem further. It is not straightforward to determine $Q_s$ and $X_s$ of $S_s$ such that they contribute to $\text{load}_{\text{sys}}$ in a minimal way. According to Eq. (8), $X_s$ can serve as the blocking of its higher-priority subsystem $S_k$ depending on the value of $X_j$ of other lower-priority subsystems $S_j$. Hence, it is impossible to determine $X_s$ and $Q_s$ in an optimal way, without knowledge of other subsystems’ interfaces.

We consider a two-step approach to the system load minimization problem. In the first step, each subsystem generates a set of interface candidates independently (with no information about other subsystems), which is suitable for subsystems to be developed in open environments. The second step is performed when subsystems are integrated to form a system. During this integration of subsystems, being aware of all interface candidates of all subsystems, only one out of all interface candidates for
each subsystem is selected (that will be used by the system-level scheduler later on) such that a resulting system load can be minimized.

6 Interface candidate generation

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 must exist an element of $IC_s$ that constitutes an optimal solution to the system load problem.

Suppose $S_s$ contains $n$ internal tasks that access $m$ global shared resources. Note that as explained in Section 5, each global resource may have up to $n$ different internal resource ceilings, and one interface candidate can be generated from each combination of $m$ resource ceilings. A brute-force solution to the interface generation problem is then to generate all possible $m^n$ interface candidates. However, not all of these $m^n$ candidates have the potential to constitute the optimal solution; those that require more CPU demand and impose greater blocking on other subsystems can be considered as replicate candidates.

Hence, we present the ICG (Interface Candidate Generation) algorithm that is not only computationally efficient, but also 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 can be denoted as $(Q_{s,j}, X_{s,j})$ where $j$ indicates interface candidate index.

Definition 1 An interface candidate $(Q_{s,i}, X_{s,i})$ is said to be redundant if there exists $(Q_{s,k}, X_{s,k})$ such that $X_{s,i} \leq X_{s,k}$ and $Q_{s,i} \leq Q_{s,k}$, where $k < i$ (denoted as $(Q_{s,i}, X_{s,i}) \leq (Q_{s,k}, X_{s,k})$). In addition, $(Q_{s,j}, X_{s,j})$ is said to be non-redundant if it is not redundant.

Suppose $(Q'_{s}, X'_{s}) \leq (Q^*_{s}, X^*_{s})$. Then, the former candidate will never yield a larger DBF_s(t) than the latter does. This immediately follows from Eqs. (6) and (7). That is, a subsystem $S_s$ will never impose more CPU requirement to the system load with $(Q'_{s}, X'_{s})$ than with $(Q^*_{s}, X^*_{s})$. The following lemma records this property.

Lemma 1 If $(Q'_{s}, X'_{s}) \leq (Q^*_{s}, X^*_{s})$, $(Q'_{s}, X'_{s})$ will never contribute more to load_{sys} than $(Q^*_{s}, X^*_{s})$ does.

Proof Since $(Q'_{s}, X'_{s}) \leq (Q^*_{s}, X^*_{s})$, we have that $Q'_{s} \leq Q^*_{s}$ and $X'_{s} \leq X^*_{s}$. Therefore, $Q'_{s} + X'_{s} \leq Q^*_{s} + X^*_{s}$. Since $(Q'_{s}, X'_{s})$ never increases DBF_s(t) (see Eq. (7)) and the blocking $B_s$, respectively, compared to $(Q^*_{s}, X^*_{s})$. It means that $(Q'_{s}, X'_{s})$ does not increase LBF_s(t) and thereby load_{sys} (see Eq. (10)), compared to $(Q^*_{s}, X^*_{s})$. That is, $(Q'_{s}, X'_{s})$ has no potential to increase load_{sys} (see Eq. (11)), compared to $(Q^*_{s}, X^*_{s})$.

Lemma 1 suggests that redundant candidates be excluded from a solution, and it reduces the number of interface candidates significantly. However, a brute-force approach to reduce redundant candidates is still computationally intractable, since the complexity of an exhaustive search is very high $O(m^n)$. We now present important properties that serve as the basis for the development of a computationally efficient algorithm.

In order to discuss some subtle properties in detail, let us further refine some of our notations with additional parameters. Firstly, the maximum blocking ($b_t$) imposed to a task $\tau_i$ can vary depending on which resource $\tau_i$ accesses. Hence, let $b_{i,j}$ denote the maximum blocking that a task with priority higher
than i can experience in accessing a resource $R_j$, i.e., $b_{i,j} = \max\{c_{k,j}\}$ for all $\tau_k \leq \tau_i$. Secondly, the maximum CPU demand ($w_j$) imposed to any task accessing a resource $R_j$ can also be different depending on the internal ceiling ($rc_j$) of $R_j$. So let $w_{j,k}$ particularly represent $w_j$ when $rc_j = k$.

The following two lemmas show the properties of redundant interfaces, suggesting insights for how to effectively exclude them.

**Lemma 2** Let $R^i$ denote a set of resources whose resource ceilings are $i$. Suppose a resource $R_k \in R^i$ yields the greatest blocking among all the elements of $R^i$. Then, it is the resource $R_k$ that requires the greatest CPU demand to complete any task’s execution inside a critical section among all elements of $R^i$, i.e.,

$$
(b_{i,k} = \max_{\forall R_j \in R^i} \{ b_{i,j} \}) \rightarrow (w_{k,i} = \max_{\forall R_j \in R^i} \{ w_{j,i} \}).
$$

**Proof** The $w_{j,i}$ depends on two parameters (see Eq. (3)); $cx_j$, which is equal to $(b_{i,j})$ since $rc_j = i$, and the interference from tasks with higher priority (the summation part denoted as $I$). Note that $I$ in invariant to difference resources $R_j \in R^i$, since it considers only the tasks with priority greater than $i$ in the summation. Then, it is clear that $w_{j,i}$ depends only on $b_{i,j}$, and it follows that the resource with the maximum $b_{i,j}$, will be consequently associated with the maximum $w_{i,j}$. □

Using Lemma 2, the following lemma particularly shows how 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 $b_{k,y} < b_{k,z}$ 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 $(Q', X')$ denote an interface candidate generated when $rc_y = k$ and $rc_z = i$, where $k < i$. Let $(Q^*, X^*)$ denote another interface candidate generated when $rc_y = rc_z = i$. We wish to show that $(Q^*, X^*) \leq (Q', X')$, i.e., $Q^* \leq Q'$ and $X^* \leq X'$.

Given $b_{i,y} < b_{i,z}$, it follows from Lemma 2 that $w_{y,i} < w_{z,i}$. This means that even though the ceiling of $R_y$ increases to $i$, it does not change the maximum blocking $(b_i)$ of tasks $\tau_i$. Therefore, it does not change the demand bound function either. As a result, $Q^* = Q'$.

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

### 6.1 ICG algorithm

**Description.** Using Lemmas 1, 2, and 3, we can reduce the complexity of a search algorithm. The algorithm shown in Figure 3 is based on these lemmas. In the beginning (at line 1), each resource ceiling $rc_j$ is set to its initial ceiling value according to SRP (without applying the technique in [5]). The algorithm then generates an interface candidate $(Q^*, X^*)$ based on the current resource ceilings (line 3 and 4). This new interface candidate is added into a list (line 6). Such addition can make some
- calculateBudget($S_s, P_s, RC_s$) returns the smallest subsystem budget that satisfies Eq. (4).
- increaseCeilingX$(RC_s)$ returns whether or not the ceiling of the resource associated with $X^*$ can be increased by one. 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 (Lemma 3).
- Interface is an array of interface candidates; each candidate is $(Q, X, RC)$.
- addInterface(Interface, $Q^*, X^*, RC_s$) adds new interface in the interface list array.
- removeRedundant(Interface) removes all redundant interfaces from the interface list.

1: $RC_s = \{rc_1, \ldots, rc_m\} \parallel rc_j$=initial ceiling of $R_j$ (SRP)
2: num = 0
3: do
4: $Q^* = $calculateBudget($S_s, P_s, RC_s$)
5: $X^* = \text{max}\{w_{1,rc_1}, \ldots, w_{m,rc_m}\}$
6: addInterface(Interface, $Q^*, X^*, RC_s$)
7: num=removeRedundant(Interface)
8: while (increaseCeilingX$(RC_s)$)
9: return (Interface, num)

Figure 3. The ICG algorithm.

candidate redundant according to Lemma 1, and those redundant candidates are removed (line 7). Let $R^*$ denote the resource that determines $X^*$ in line 3, and $v^*$ denote the value of the ceiling ($rc^*$) of $R^*$ at that moment. In line 8, the algorithm 1) increases the ceiling $rc^*$ by one 2) checks the conditions given in Lemma 3 to further increase $rc^*$ if possible, and 3) increases the ceiling of all other resources that have the same ceiling as $v^* + 1$, to the current value of $rc^*$. This way, we can further reduce redundant interface candidates.

**Example.** We illustrate the ICG 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 $P_s = 125$. The algorithm works as shown in Table 2. The results from step 1 are $(Q_{s,1} = 26, X_{s,1} = 102)$, at step 2 $(Q_{s,1}, X_{s,1}) > (Q_{s,2}, X_{s,2})$. So $(Q_{s,1}, X_{s,1})$ is redundant (see Definition 1). That is, this interface can be removed according to Lemma 1. For the same reason, $(Q_{s,2}, X_{s,2})$ 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 $b_{2,1} > b_{2,2}$. At both steps 4 and 5, the ceiling $rc_1$ is increased by one since $X_{s,i} = w_i$ but we increase the ceiling of $rc_2$ according to Lemma 3. The algorithm selects the interface candidates from steps 3, 4 and 5.
<table>
<thead>
<tr>
<th>Step</th>
<th>( rc_1 )</th>
<th>( rc_2 )</th>
<th>( w_1 )</th>
<th>( w_2 )</th>
<th>( Q_{s,i} )</th>
<th>( X_{s,i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>13</td>
<td>102</td>
<td>51</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>13</td>
<td>52</td>
<td>51</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>13</td>
<td>7</td>
<td>51</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>12</td>
<td>6</td>
<td>52.5</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>4</td>
<td>56</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2. Example algorithm

Correctness. The following lemma proves the correctness of the ICG algorithm.

Lemma 4 Let \( \mathcal{IC} \) denote a set of up to \( n \) interface candidates that are generated by the ICG algorithm of Figure 3. There exists no non-redundant interface candidate \((Q_{s,y}, X_{s,y})\) such that \((Q_{s,y}, X_{s,y}) \notin \mathcal{IC}\).

Proof Assume that \((Q_{s,y}, X_{s,y})\) is a non-redundant interface candidate and that \(X_{s,y} = w_{k,i}\), i.e., the subsystem critical section execution 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 \(b_{i,j} > b_{i,k}\) for all \(rc_j > i\). Otherwise we could change the ceiling \(rc_k = rc_j\) according to Lemma 3, and by this \(w_{k,i} \neq X_{s,y}\).

2. There is no \(R_j\) such that \(b_{t,j} > b_{i,k}\) for all \(rc_j < i, t < i\). Otherwise \(w_{j,t} > w_{k,i}\) because when we compute the \(w_k\) and \(w_j\), the interference from higher priority tasks as well as blocking is higher for \(R_j\), and then \(w_{k,i} \neq X_{s,y}\). 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 resource ceiling \(i\), 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, \(b_{f,k}\) (where \(f \in [v, i]\)) is the maximum blocking at resource ceiling \(rc_k \in [v, i]\). Since the presented algorithm increases the ceiling of the global resource that generate the maximum subsystem critical section execution time, it will increase the ceiling of \(R_k\) when \(rc_k = v\) up to \(i\). Hence, we can guarantee that the algorithm will include the interface when \(X_{s,y} = w_{k,i}\).

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 possible resource ceilings) and there is either 0 or 1 non-redundant interface for each resource ceiling level, and the algorithm will only traverse these non-redundant interfaces. Moreover, the proposed algorithm thereby produce at most \(n\) interface candidates.

Post-processing. The ICG algorithm generates non-redundant interface candidates on the basis of Lemma 1. The notion of redundant candidate is so general that the ICG algorithm can be applicable to many synchronization protocols. In some cases, however, a set of interface candidates can be further
refined, for instance, when the overrun mechanism described in Section 4.1 is used. Consider two candidates \((Q'_s, X'_s)\) and \((Q^*_s, X^*_s)\) such that \(Q'_s + X'_s \leq Q^*_s + X^*_s\) and \(X'_s \leq X^*_s\). Then, \((Q'_s, X'_s)\) will never produce not only a larger \(DBF_s(t)\) for the subsystem \(S_s\) itself, but also a larger blocking \(B_j\) for other subsystems \(S_j\), than \((Q^*_s, X^*_s)\) does. This immediately follows from Eqs. (6)-(8). Then, we introduce the following lemma.

**Lemma 5** Consider two candidates \((Q'_s, X'_s)\) and \((Q^*_s, X^*_s)\) such that \(Q'_s + X'_s \leq Q^*_s + X^*_s\) and \(X'_s \leq X^*_s\). Then, \((Q'_s, X'_s)\) will never impose more CPU requirement to \(load_{sys}\) in any way than \((Q^*_s, X^*_s)\) does.

**Proof** Looking at Eq. (6), we can decrease \(LBF_s(t)\) to decrease the system load by decreasing the blocking \(B_s\) and/or \(DBF_s(t)\). For the blocking, \((Q'_s, X'_s)\) does not increase the blocking on the higher priority subsystems, compared to \((Q^*_s, X^*_s)\), because \(X'_s \leq X^*_s\). In addition, \((Q'_s, X'_s)\) does not increase \(DBF_s(t)\) either, because \((Q'_s + X'_s) \leq (Q^*_s + X^*_s)\) (see Eq. (7)). So we can see that \((Q^*_s, X^*_s)\) can be removed, since \((Q'_s, X'_s)\) only has a potential to increase the system load, in comparison to \((Q'_s, X'_s)\).

According to Lemma 5, a set of interface candidates generated by the ICG algorithm goes through its post-processing for further refinement, and this is very useful for the second step of our approach.

### 7 Interface selection

In this section, we consider a problem, called the **optimal interface selection** problem, that selects a **system configuration** consisting of a set of subsystem interfaces, one from each subsystem that together minimize the system load subject to the schedulability of system. We present the ICS (Interface Candidate Selection) algorithm, an algorithm that finds an optimal solution to this problem through a finite number of iterative steps.

#### 7.1 Description of the ICS algorithm

The ICS algorithm assumes that each set of interface candidates \((Q_s, X_s)\) is sorted in a decreasing order of \(X_s\). In other words, each set is sorted in an increasing order of collective demands \((Q_s + X_s)\) (see Lemma 5). Then, the first candidate \((Q_{s,1}, X_{s,1})\) has the largest critical section execution time but the smallest collective demands.

The ICS algorithm generates a finite number of system configurations through iteration steps. Each configuration is a set of individual interface candidates of all subsystems. Let \(CF_i\) denote a **configuration** that ICS generates at an \(i\)-th iteration step. For notational convenience, we introduce a variable \(f^i_k\) to denote an element of \(CF_i\), i.e., \(CF_i = \{f^i_1, \ldots, f^i_N\}\). The variable \(f^i_k\) represents the interface candidate index of a subsystem \(S_k\), indicating that the configuration in the \(i\)-th step includes \((Q_{k,f^i_k}, X_{k,f^i_k})\).

Figure 4 shows an example to illustrate the ICS algorithm, where the system contains 3 subsystems such that 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 possible configuration, and each number in the node corresponds to an interface candidate index in the order of \(S_1, S_2,\) and \(S_3\). The arrows show the possible transitions between nodes at \(i\)-th iteration step, by increasing \(f^i_k\) by 1 for each subsystem \(S_k\) one by one. We describe the ICS algorithm with this example.

**Initialization.** In the beginning, this algorithm generates an initial configuration \(CF_0\) such that it consists of the first interface candidates of all subsystems. In Figure 4, \(CF_0 = \{1, 1, 1\}\) (see line 2 of Figure 5).
**Iteration step.** The ICS algorithm transits from \((i-1)\)-th step to \(i\)-th step, increasing only one element of \(\text{CF}_{i-1}\) in value by one. In Figure 4, the arrows with bold lines illustrate the path that ICS can take. For instance, ICS moves from the initialization step \((\text{CF}_0 = \{1, 1, 1\})\) to the first step \((\text{CF}_1 = \{2, 1, 1\})\). Then, the ICS algorithm excludes the two sibling nodes of \(\text{CF}_1\) in the figure (i.e., \(\{1, 2, 1\}\) and \(\{1, 1, 2\}\)) from the remaining search space; the algorithm will never visit those nodes from this step on. This way, ICS can efficiently explore the search space. Let us describe how ICS behaves at each iteration step more formally.

Firstly, let \(\delta_i\) denote the only single element whose value increases by one between \(\text{CF}_{i-1}\) and \(\text{CF}_i\), i.e.,

\[
    f_i^k = \begin{cases} 
    f_{i-1}^k + 1 & \text{if } k = \delta_i, \\
    f_{i-1}^k & \text{otherwise.}
    \end{cases} \quad (13)
\]

In the example shown in Figure 4, \(\delta_1 = 1\).

Let us explain how to determine \(\delta_i\) at an \(i\)-th step. We can potentially increase every elements of \(\text{CF}_{i-1}\), and thereby we have at most \(N\) candidates for the value of \(\delta_i\). Here, we choose one out of at most \(N\) candidates such that a resulting \(\text{CF}_i\) can cause the system load to be minimized.

Let \(\text{load}_{\text{sys}}(i)\) denote the value of \(\text{load}_{\text{sys}}\) when a configuration \(\text{CF}_i\) is used as a system interface. We are now interested in reducing the value of \(\text{load}_{\text{sys}}(i - 1)\). Let \(s^*\) denote the subsystem \(S_{s^*}\) that has the largest processor request bound among all subsystems. That is, \(\text{load}_{\text{sys}}(i - 1) = \alpha_{s^*}\) (see Eq. (10)). We can find such \(S_{s^*}\) by evaluating the processor request bound’s of all subsystems (in line 5 of Figure 5).

By the definition of \(s^*\), we can reduce the value of \(\text{load}_{\text{sys}}(i - 1)\) by reducing the value of \(\text{LBF}_{s^*}(t)\). There are two potential ways to reduce the value of \(\text{LBF}_{s^*}(t)\). From the definition of \(\text{LBF}_{s^*}(t)\) in Eq. (6), one is to reduce its maximum blocking \(B_{s^*}\) and the other is to reduce the subsystem demands (\(\text{DBF}_{s^*}(t)\)). A key aspect of this algorithm is that it always reduces the blocking part, but does not reduce the demand...
- $IC_s$ is an array of interface candidates of subsystem $S_s$, sorted in a decreasing order of $X_s$.
- $ici_s$ is an index to $IC_s$ of subsystem $S_s$.
- $I$ 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., $load_{sys} = \alpha_{s^*}$.
- maxBlockingSubsystemToSysload($s^*$) returns a subsystem $S_{k^*}$ that produces the greatest blocking to a subsystem $S_{s^*}$. Note that $S_{s^*}$ determines the system load.

1: for all $S_s \in S$
2:   $ici_s = 1;\ I_s = IC_s[ici_s]$
3:   load_{sys}^* = 1.0;\ I^* = I$
4: do
5:   $s^* = \text{subsystemWithMaxLoad}()$
6:   $k^* = \text{maxBlockingSubsystemToSysload}(s^*)$
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. (11)
11:   if ($load_{sys} < load_{sys}^*$)
12:      $load_{sys}^* = load_{sys}$
13:   else
14:      return $I^*$ (that determines $load_{sys}^*$)
15: until(true)

Figure 5. The ICS algorithm.

part. An intuition behind is as follows: this algorithm starts from the interface candidates that have the smallest demands but the largest subsystem critical section execution 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 $B_{s^*}$ of $S_{s^*}$. It can be reduced by decreasing the $X_{k^*}$ of a subsystem $S_{k^*}$ that imposes the largest blocking to the subsystem $S_{s^*}$. We define $k^*$ in a more detail.

Let $k^*$ denote the subsystem $s_{k^*}$ that imposes the largest blocking to the subsystem $S_{s^*}$, i.e., $B_{s^*} = X_{k^*} = \max\{X_j \mid \text{for all } X_j \in LPS(s^*)\}$, where $LPS(i)$ is a set of lower-priority subsystems of $S_{s^*}$. We can find such $S_{k^*}$ easily by looking at the subsystem critical section execution times of all lower-priority subsystems of $S_{s^*}$ (in line 6 of Figure 5).

When such $S_{k^*}$ is found, it then checks whether the $X_{k^*}$ can be further reduced (in line 7 of Figure 5). If so, it is reduced (in line 8), and $CF_{i-1}$ becomes to $CF_i$ (in line 9). That is, $\delta_i = k^*$.

**Iteration termination.** The above iteration process terminates when the blocking $B_{s^*}$ of subsystem $S_{s^*}$ cannot be reduced further. The algorithm then finds the smallest value of $load_{sys}^*$ out of the values

\[load_{sys}^* = 1.0;\ I^* = I\]

\[ci = 1;\ I_s = IC_s[ci]\]

\[load_{sys}^* = 1.0;\ I^* = I\]

\[for\ all\ S_s \in S\]

\[ici_s = 1;\ I_s = IC_s[ici_s]\]

\[load_{sys}^* = 1.0;\ I^* = I\]

\[do\]

\[s^* = \text{subsystemWithMaxLoad}()\]

\[k^* = \text{maxBlockingSubsystemToSysload}(s^*)\]

\[if\ (ici_{k^*}\ can\ increase\ by\ one)\]

\[ici_{k^*} = ici_{k^*} + 1\]

\[I_{k^*} = IC_{k^*}[ici_{k^*}]\]

\[compute\ load_{sys}^*\ according\ to\ Eq.\ (11)\]

\[if\ (load_{sys} < load_{sys}^*)\]

\[load_{sys}^* = load_{sys}\]

\[else\]

\[return\ I^*\ (that\ determines\ load_{sys}^*)\]

\[until(true)\]

\[\delta_i = k^*\]

\[load_{sys}^* = \max\{X_j \mid \text{for all } X_j \in LPS(s^*)\}\]

\[load_{sys}^* = 1.0;\ I^* = I\]

\[ici_s = 1;\ I_s = IC_s[ici_s]\]

\[load_{sys}^* = 1.0;\ I^* = I\]

\[for\ all\ S_s \in S\]

\[ici_s = 1;\ I_s = IC_s[ici_s]\]

\[load_{sys}^* = 1.0;\ I^* = I\]

\[do\]

\[s^* = \text{subsystemWithMaxLoad}()\]

\[k^* = \text{maxBlockingSubsystemToSysload}(s^*)\]

\[if\ (ici_{k^*}\ can\ increase\ by\ one)\]

\[ici_{k^*} = ici_{k^*} + 1\]

\[I_{k^*} = IC_{k^*}[ici_{k^*}]\]

\[compute\ load_{sys}^*\ according\ to\ Eq.\ (11)\]

\[if\ (load_{sys} < load_{sys}^*)\]

\[load_{sys}^* = load_{sys}\]

\[else\]

\[return\ I^*\ (that\ determines\ load_{sys}^*)\]

\[until(true)\]

\[\delta_i = k^*\]

\[load_{sys}^* = \max\{X_j \mid \text{for all } X_j \in LPS(s^*)\}\]

\[\delta_i = k^*\]
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 \(S_{\delta_i}\). Then, it can repeat \(O(N \times m')\) iterations, where \(N\) is the number of subsystems and \(m'\) is the greatest number of interface candidates of a subsystem among all.

### 7.2 Correctness of the ICS algorithm

In this section, we show that the ICS algorithm produces a set of system configurations 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.,

\[
\mathcal{AS} = IC_1 \times \cdots \times IC_n. \tag{14}
\]

In the example shown in Figure 4, the entire solution space \((\mathcal{AS})\) has 12 elements.

We present some notations to denote the properties of the ICS algorithm at an arbitrary \(i\)-th iteration step.

- **\(\hat{IC}^i_k\)** In the beginning, the ICS 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 \((\hat{IC}^i_k)\) to indicate the remaining interface candidates of a subsystem \(S_k\) to explore. By definition, \(f^i_k\) indicates which interface candidate of a subsystem \(S_k\) is selected by \(CF_i\). This algorithm continues exploration from the interface candidate indicated by \(f^i_k\) from the end of an \(i\)-th step. Then, \(\hat{IC}^i_k\) is defined as

\[
\hat{IC}^i_k = \{f^i_k, \ldots, max_k\} \text{ for all } k = 1, \ldots, n, \tag{15}
\]

where \(max_k\) is the number of interface. In the example shown in Figure 4, \(\hat{IC}^1_1 = \{2, 3\}\).

- **\(XP_i\)** 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 \(CF_i\) chosen at the \(i\)-th step. Then, \(XP_i\) is defined as

\[
XP_i = (\hat{IC}^i_1 \times \cdots \times \hat{IC}^i_n) \setminus CF_i. \tag{16}
\]

- **\(RM_i\)** In essence, the ICS 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 \(XP_i \subset XP_{i-1}\). 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 \(S_{\delta_i}\) changes from \(f^{i-1}_{\delta_i}\) to \(f^i_{\delta_i}\). Then, a subset of \(XP_i\) that contains the value of \(f^{i-1}_{\delta_i}\), is excluded at the \(i\)-th step. \(RM_i\) is defined as

\[
RM_i = (\hat{IC}^{(i-1)*}_1 \times \cdots \times \hat{IC}^{(i-1)*}_n) \setminus \{CF_{i-1}\}, \text{ where} \tag{17}
\]

\[
\hat{IC}^{(i-1)*}_k = \begin{cases} 
\{f^{i-1}_k\} & \text{if } k = \delta_i, \\
\hat{IC}^i_k & \text{otherwise.}
\end{cases} \tag{18}
\]

In the example shown in Figure 4, \(RM_1 = \{\{1, 2, 1\}, \{1, 2, 2\}, \{1, 1, 2\}\}\).

- **\(AH_i\)** Let \(AH_i\) represent a set of system configurations that the ICS algorithm selects from the first step through to an \(i\)-th step, i.e.,
\[ \text{AH}_i = \{ \text{CF}_1, \ldots, \text{CF}_j \}. \] (19)

- \( \text{AR}_i \): Let \( \text{AR}_i \) represent a set of interface candidates that the ICS algorithm excludes from the first step through to an \( i \)-th step, i.e.,

\[ \text{AR}_i = \text{RM}_{(i-1)} \cup \text{RM}_i, \quad \text{where} \quad \text{AR}_0 = \phi. \] (20)

We define partial ordering between interface candidates as follows:

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

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

**Lemma 6** At an arbitrary \( i \)-th iteration step, the ICS algorithm excludes a set of interface candidates (\( \text{RM}_i \)), and any excluded solution candidate \( r \in \text{RM}_i \) does not yield a smaller system load than that by \( \text{CF}_{i-1} \).

**Proof** As explained in Section 7.1, there are two potential ways to reduce the value of \( \text{load}_{\text{sys}}(\text{CF}_{i-1}) \) at the \( i \)-th step. One is to reduce the demand of the subsystem \( S_{s^*_i} \) (i.e., \( \text{DBF}_{s^*_i}(t) \)), and the other is to reduce its maximum blocking \( B_{s^*_i} \).

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

Secondly, we wish to show that when we change \( \text{CF}_{i-1} \) to any interface candidate \( r \in \text{RM}_i \), \( B_{s^*_i} \) does not decrease. As shown in line 6 in Figure 5, the ICS algorithm finds the subsystem \( S_{\delta_i} \) that generates the maximum blocking to for subsystem \( S_{s^*_i} \). Then, the algorithm increases \( f_{s^*_i}^{(i-1)} \) by one, if possible, to decrease \( B_{s^*_i} \). However, by definition, for all elements \( r \) of \( \text{RM}_i \), the element for the subsystem \( S_{\delta_i} \) has the value of \( f_{s^*_i}^{(i-1)} \), rather than the value of \( f_{s^*_i}^{(i)} \). This means that \( B_{s^*_i} \) never decreases when we change \( \text{CF}_{i-1} \) to \( r \). \( \square \)

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

**Lemma 7** When the ICS algorithm terminates at an arbitrary \( f \)-th step, any remaining interface candidate \( \text{xp} \in \text{XP}_f \) does not yield a smaller system load than \( \text{CF}_f \) does.

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

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

The other is to reduce the maximum blocking \( (B_{s^*_f}) \). In fact, the ICS 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 \( \text{CF}_f \) changes to any \( \text{xp} \). \( \square \)
The following lemma states that at \(i\)-th step, the remaining search space to explore decreases by \((\text{RM}_i \cup \{\text{CF}_i\})\).

**Lemma 8** At an arbitrary \(i\)-th iteration step,

\[
\text{XP}_i = \text{XP}_{i-1} \setminus (\text{RM}_i \cup \{\text{CF}_i\}).
\]

**Proof** The ICS algorithm transforms \(\text{CF}_{i-1}\) to \(\text{CF}_i\) at an \(i\)-th step by increasing the value of its \(\delta_i\)-th element. Then, we have

\[
\hat{I}_C^i_k = \begin{cases} 
\hat{I}_C^{i-1}_k \setminus \{f_{k-1}^i\} & \text{if } k = \delta_i, \\
\hat{I}_C^{i-1}_k & \text{otherwise}.
\end{cases}
\]

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

\[
\begin{align*}
\text{XP}^*_i &= \hat{I}_C^1_i \times \hat{I}_C^2_i \times \cdots \times \hat{I}_C^n_i \\
&= (\hat{I}_C^{i-1}_1 \setminus \{f_{i-1}^1\}) \times \hat{I}_C^2_i \times \cdots \times \hat{I}_C^n_i \\
&= (\{f_{i-1}^1\} \times \hat{I}_C^2_i \times \cdots \times \hat{I}_C^n_i) \\
&= \text{XP}^*_{i-1} \setminus \text{RM}^*_i \\
&= \left(\text{XP}_{i-1} \cup \{\text{CF}_{i-1}\}\right) \setminus \left(\text{RM}_i \cup \{\text{CF}_{i-1}\}\right) \\
&= \text{XP}_{i-1} \setminus \text{RM}_i.
\end{align*}
\]

That is, considering \(\text{XP}^*_i = \text{XP}_i \cup \{\text{CF}_i\}\), it follows

\[
\text{XP}_i = \text{XP}_{i-1} \setminus (\text{RM}_i \cup \{\text{CF}_i\}).
\]

\(\square\)

The following lemma states that at any \(i\)-th iteration step, the entire search space can be divided into a set of explored candidates \((\text{AH}_i)\), a set of excluded candidates \((\text{AR}_i)\), and a set of remaining candidates to explore \((\text{XP}_i)\).

**Lemma 9** At an arbitrary \(i\)-th step, the sets of \(\text{AR}_i\), \(\text{AH}_i\), and \(\text{XP}_i\) include all possible interface candidates.

\[
\text{AR}_i \cup \text{AH}_i \cup \text{XP}_i = \mathcal{AS}
\]

**Proof** We will prove this lemma by using mathematical induction. As a base step, we wish to show Eq. (25) is true, when \(i = 1\). Note that \(\text{AR}_0 = \emptyset\) and \(\text{AH}_0 = \{\text{CF}_0\}\). In addition, \(\text{XP}_0 = \mathcal{AS} \setminus \text{CF}_0\), according to Eq. (16). It follows that \(\text{AR}_0 \cup \text{AH}_0 \cup \text{XP}_0 = \mathcal{AP}\).

We assume that Eq. (25) is true at the \(i\)-th iteration step of the ICS algorithm. We then wish to prove that it also holds at the \((i + 1)\)-th step, i.e.,

\[
\text{AR}_i \cup \text{AH}_i \cup \text{XP}_i = \text{AR}_{i+1} \cup \text{AH}_{i+1} \cup \text{XP}_{i+1}.
\]
According to the definitions $AH_{i+1}$, $AR_{i+1}$, and $XP_{i+1}$ (see Eq. (19), (20) and (21)), we can rewrite the right-hand side of Eq. (26) as follows:

\[
AR_{i+1} \cup AH_{i+1} \cup XP_{i+1} = (AR_i \cup RM_{i+1}) \cup (AH_i \cup \{CF_{i+1}\}) \cup (XP_i \setminus (RM_{i+1} \cup \{CF_{i+1}\})) = AR_i \cup AH_i \cup XP_i.
\]

The following theorem states that the ICS algorithm produces a set of system configurations, which must contain an optimal solution.

**Theorem 10** When the ICS algorithm terminates at the $f$-th step, a set of system configurations $(AH_f)$ includes an optimal solution.

**Proof** Let $opt$ denote an optimal solution. We prove this theorem by contradiction, i.e., by showing that $opt \notin AR_f$ and $opt \notin XP_f$.

Suppose $opt \in AR_f$. Then, by definition, there should exist $RM_i$ such that $opt \in RM_i$ for an arbitrary $i \leq f$. According to Lemma 6, $load_{sys}(CF_{i-1}) < load_{sys}(opt)$, which contradicts the definition of $opt$. Hence, $opt \notin AR_f$.

Suppose $opt \in XP_f$. Then, according to Lemma 7, it should be $load_{sys}(CF_f) < load_{sys}(opt)$, which contradicts the definition of $opt$ as well. Hence, $opt \notin AR_f$.

According to Lemma 9, it follows that $opt \in CF_f$. $\square$

8 Overrun mechanism with payback

David and Burns [7] presented another overrun mechanism called overrun with payback mechanism. It works as follows, whenever overrun happens, the subsystem $S_s$ pays back $O_s$ in its next execution instant, i.e., the subsystem budget $Q_s$ will be decreased by $O_s$ for the subsystem’s execution instant following the overrun (note that only the instant following the overrun is affected).

In this section we will discuss how we can apply the ICG and ICS algorithms with a system that uses overrun with payback mechanism and how this will effect on system load. First, We will explain briefly how to analyze the local and global schedulability with this type of overrun.

**Local schedulability analysis.** We can still use Eq. (5) for the payback version of overrun where $dbf_{FP}(i,t)$ is the same as in the overrun without payback (presented in section 4.1). However, the $sbf(t)$ will be smaller in the payback version, compared to the other version of without payback. This is because the payback version may produce a longer blackout duration between two consecutive periodic processor allocations (see [4] for more details). As a consequence, the subsystem budget for a system that use overrun with payback will be greater or equal to the subsystem budget with the other version of overrun.
Global schedulability analysis. Eq. (9) is valid only if we change $O_k(t)$ in Eq. (7) such that $O_k(t) = X_k$ for $0 \leq t \leq P_k$. Then Eq. (7) using the overrun with payback mechanism can be rewritten as follows,

$$DBF_s(t) = (Q_s + X_s + \sum_{S_k \in \text{RPS}(S_s)} \left\lceil \frac{t}{P_k} \right\rceil \cdot (Q_k) + X_k)$$ (27)

The ICG algorithm presented in section 6.1 can be used without any problem with the payback version. The reason is that local schedulability for both overrun mechanisms is the same and Lemmas 2-4 are based on the local scheduling. Lemma 1 is based on the global scheduling and it is valid also with payback version of overrun.

For ICS algorithm, the possibilities to minimize $LBF_s(t)$ using overrun with payback are as follows; looking at Eq. (27) and depending on the ratio between $P_s/P_k$, if these periods are close to each others (i.e., $P_s/P_k$ is low) then then minimizing $Q_k + X_k$ of the higher priority subsystems will minimize $DBF_s(t)$ while if the $P_s$ is multiple times of the $P_k$ then it will be desired to minimize $Q_k$. The second factor is $Q_s + X_s$ of the subsystem and the third factor is $B_s$. So the difference between overrun with payback and the without payback version is that minimizing $Q_k$ can minimize the $LBF_s(t)$ which was not in the overrun without payback. And by this, Lemma 5 is not valid with the payback version of overrun and the assumption presented in section 7.1 for ICS algorithm is not valid. Hence, it calls for some appropriate changes to the post-processing part, desirably, in a way that require less subsequent changes to the ICS algorithm.

Comparing the two versions of overrun mechanism, without payback version is better than the payback version in the local schedulability and the subsystem budget will be lower when using the overrun without payback. While in the global schedulability, the payback version will be better since the demand of subsystem $S_s$ (at worst case) increases by $Q_k$ (the budget of higher priority subsystems) every period $P_k$ (see Eq. (27)). while in the overrun with payback, it increases by $Q_k + X_k$ Eq. (7). Another difference is that overrun with payback the subsystem budget $Q_s \geq X_s$ while it is not necessary when using overrun without payback.

9 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 impose extra CPU demands. However, simply reducing the blocking of subsystems does not monotonically decrease the system load, since imposing less blocking to other subsystems can impose more CPU requirements of the subsystems themselves. This paper introduced such a tradeoff and presented a two-step approach to explore the intra- and inter-subsystem aspects of the tradeoff efficiently, towards determining optimal subsystem interfaces constituting the minimum system load.

In this paper, we considered only fixed-priority scheduling, and we plan to extend our framework to EDF scheduling. Furthermore, our future work includes generalizing our framework to other synchronization protocols. For example, this paper considered only the overrun mechanism without payback [7], and we are extending towards another overrun mechanism (with-payback version) [7].

References