Synthesis of Optimal Interfaces for Hierarchical Scheduling with Resources *

Insik Shin
Dept. of Computer Science, KAIST
Daejeon, South Korea 305-701

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 recourses (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 also suitable for adaptable and reconfigurable systems.

1 Introduction

Hierarchical scheduling has emerged as a promising vehicle for 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 [1, 5, 7, 8, 9, 10, 12, 14, 15, 17, 21, 22, 24] have been introduced for the schedulability analysis of HSFs, where subsystems are independent. For dependent subsystems, synchronization protocols [3, 6, 11] have been proposed for arbitrating accesses to logical resources (i.e., semaphore) across subsystems in HSFs. There have been a few studies [21, 9] 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 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* [21, 23], 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 [4], in order to potentially reduce the blocking of other subsystems towards decrease of the system load. However, in doing so, we present in this paper an unexpected 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

*The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS. This has been done while I. Shin was a postdoc research fellow at MRTC. Contact: insik.shin@cs.kaist.ac.kr.

1052-8725/08 $25.00 © 2008 IEEE
DOI 10.1109/RTSS.2008.34

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 [7], 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 remainder of the 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 8 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 [7] in the late 1990's, has been receiving an increasing research attention. Since Deng and Liu [7] introduced a two-level HSF, its schedulability has been analyzed under fixed-priority global scheduling [12] and under Earliest Deadline First (EDF) based global scheduling [14]. Mok *et al.* [17] proposed the bounded-delay virtual processor model to achieve a clean separation in a multi-level HSF, and schedulability analysis techniques [10, 22] have been introduced for this resource model. In addition, Shin and Lee [21, 23] 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, 15, 5] and under EDF scheduling [21, 24]. More recently, Easwaran *et al.* [8] 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, 12] 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 [6] and skipping [3], and another protocol [11] for bounded-delay virtual processor model based HSFs. Bertogna *et al.* [4] 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. Shin and Lee [21] 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 $\mathtt{sbf}_\Gamma(t)$ of $\Gamma(P, Q)$ was given in [21] that computes the minimum possible CPU supply for every interval length t as follows:

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

where $k = \max\left(\left\lceil \big(t - (P-Q)\big)/P \right\rceil, 1\right)$.

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, that belong to same subsystem, 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. Let $\texttt{HP}(i)$ returns the set of tasks with higher priorities than that of τ_i.

A subsystem $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the set representing the whole system of subsystems, is characterized by $\langle \mathcal{T}_s, \mathcal{RC}_s \rangle$, where $\mathcal{T}_s$ is a task set and $\mathcal{RC}_s$ 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_s has a local Fixed-Priority Scheduler (FPS) and the system has a global FPS. Let $\texttt{HPS}(s)$ returns the set of subsystems with higher priority than that of S_s.

Let us define a *timing-interface* of a subsystem S_s such that it specifies the collective real-time requirements of S_s. 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_s. We note that X_s is similar to the concept of *resource holding time (RHT)* in [4], however, developed for a different virtual-processor model. RHT in [4] 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 [21], where subsystem-level preemptions can take place. Therefore, X_s does not represent RHT in our HSF[2], but indicates the worst-case execution time requirement that S_s demands inside a critical section. We will explain later how to derive the values of P_s, Q_s and X_s for a given subsystem S_s.

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 $rc_j = \max\{i | \tau_i \in \mathcal{T}_s \text{ accesses } R_j\}$ 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 a currently locked resource in the system/subsystem.

Under SRP, a task τ_k can preempt the currently executing task τ_i (even inside a critical section) within the same subsystem, only if the priority of τ_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_s, 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_s may collectively request inside any critical section. Note that any task τ_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_s, 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 [4]):

$$w_j^{(m+1)} = cx_j + \sum_{k=rc_j+1}^{n} \lceil \frac{w_j^{(m)}}{T_k} \rceil \cdot C_k, \qquad (1)$$

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

Then, $X_s = \max\{w_j | \text{ for all } R_j \in \mathcal{R}_s\}$, 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

[1] A processor is said to be *dedicated* to a subsystem, if the subsystem exclusively utilizes the processor with no other subsystems.

[2] As the computation of RHT is not main focus of this paper, we refer to our technical report [20] for its computation in our HSF.

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

In this paper, we consider a mechanism based on overrun [6] that works as follows; when the budget of the subsystem S_s expires and S_s has a task τ_i that is still locking a global shared resource, the task τ_i continues its execution until it releases the locked resource. The extra time that τ_i needs to execute after the budget of S_s expires is denoted as *overrun time* θ_s. The maximum θ_s occurs when τ_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 [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 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 $\mathtt{rbf}_{\mathsf{FP}}(i,t)$ denote the request bound function of a task τ_i under FPS [13], i.e.,

$$\mathtt{rbf}_{\mathsf{FP}}(i,t) = C_i + \sum_{\tau_k \in \mathtt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \qquad (2)$$

The local schedulability analysis under FPS can be then easily extended from the results of [2, 21] as follows:

$$\forall \tau_i, 0 < \exists t \leq D_i \;\; \mathtt{rbf}_{\mathsf{FP}}(i,t) + b_i \leq \mathtt{sbf}(t), \qquad (3)$$

where b_i is the maximum *blocking* (i.e., extra CPU demand) imposed to a task τ_i when τ_i is blocked by lower priority tasks that are accessing resources with ceiling greater than or equal to the priority of τ_i, and $\mathtt{sbf}(t)$ is the supply bound function. Note that t ca be selected within a finite set of scheduling points [16].

Subsystem interface. We now explain how to derive the budget Q_s of the subsystem interface. Given S_s, $\mathcal{RC}_s$, and P_s, let $\mathsf{calculateBudget}(S_s, P_s, \mathcal{RC}_s)$ denote a function that calculates the smallest subsystem budget that satisfies Eq. (3) depending on the local scheduler of S_s. Such a function is similar to the one in [21]. Then, $Q_s = \mathsf{calculateBudget}(S_s, P_s, \mathcal{RC}_s)$.

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. (2)):

$$\mathtt{LBF}_s(t) = \mathtt{RBF}_s(t) + B_s \ , \text{ where} \qquad (4)$$

$$\mathtt{RBF}_s(t) = (Q_s + O_s(t)) + \sum_{S_k \in \mathtt{HPS}(s)} \left\lceil \frac{t}{P_k} \right\rceil (Q_k + O_k(t)), \qquad (5)$$

where $O_k(t) = X_k$ 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,

$$B_s = \max\{X_j |\ S_j \in \mathtt{LPS}(S_s)\}, \qquad (6)$$

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

A global schedulability condition under FPS is then

$$\forall S_s, 0 < \exists t \leq P_s \ \mathtt{LBF}_s(t) \leq \mathtt{t} \qquad (7)$$

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

$$\alpha_s = \min_{0<t\leq P_s} \{\frac{\mathtt{LBF}_s(t)}{t} \mid \mathtt{LBF}_s(t) \leq t\}. \qquad (8)$$

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

$$\mathsf{load}_{\mathsf{sys}} = \max_{\forall S_s \in \mathcal{S}} \{\alpha_s\}. \qquad (9)$$

Note that α_s is the smallest fraction of the CPU resources that is required to schedule a subsystem S_s (satisfying Eq. (7)) assuming that the global resource supply function is αt. For example, consider a system $\mathcal{S}$ 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 $\mathsf{load}_{\mathsf{sys}} = \alpha_1 = 0.25$, 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 determing Q_s and X_s such that a resulting system load ($\mathsf{load}_{\mathsf{sys}}$) is minimized

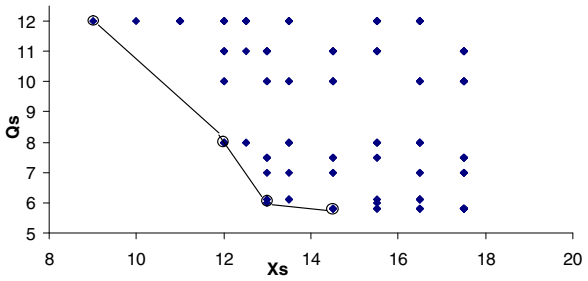

Figure 1. Tradeoff between Q_s and X_s.

subject to the schedulability of all subsystems. It is suggested from Eqs. (4) and (9) that $\mathsf{load}_{\mathsf{sys}}$ can be minimized by reducing Q_s and X_s for all subsystem S_s.

A recent study [4] introduced a method to reduce X_i. According to Eq. (1), the value of X_s can decrease, when it has less interference (i.e., the summation part of Eq. (1)) from the tasks τ_k with priorities greater than the ceiling of a resource R_j (i.e., $k > rc_j$). Such interference can be reduced by allowing fewer tasks to preempt inside the critical section of R_j. As proposed by [4], 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 rc_j is $i-1$. In this case, a task τ_i can preempt any job that is executing inside the critical section of R_j. Now, suppose rc_j is increased to i. Then, τ_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 τ_i can potentially increase, and, according to Eq. (3), this may require more CPU supply (i.e., Q_s). Figure 1 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 $\mathsf{load}_{\mathsf{sys}}$ in a minimal way. According to Eq. (6), 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 the 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 $(Q_{s,k}, X_{s,k})$ is said to be* redundant *if there exists $(Q_{s,i}, X_{s,i})$ 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, $(Q_{s,i}, X_{s,i})$ 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 $\mathtt{RBF}_s(t)$ than the latter does. This immediately follows from Eqs. (4) and (5). 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 $\mathsf{load}_{\mathsf{sys}}$ than (Q^*_s, X^*_s) does.*

Proof Suppose an interface candidate $(Q_{s,a}, X_{s,a})$ is redundant. By definition, there exists another candidate $(Q_{s,b}, X_{s,b})$ such that

- $X_{s,b} \leq X_{s,a}$ and $Q_{s,b} \leq Q_{s,a}$. So $(Q_{s,b} + X_{s,b}) <= (Q_{s,a} + X_{s,a})$. Using a redundant interface candidate will never decrease $\mathtt{RBF_s(t)}$ (see Eq. (5)) and the blocking B_s, respectively, compared to a non-redundant candidate. It means that using a redundant candidate can increases $\mathtt{LBF_s(t)}$ and thereby load_s (see Eq. (8)). That is, a redundant candidate only has a potential to increase $\mathsf{load}_{\mathsf{sys}}$ (see Eq. (9)).
- both interfaces are equivalent then system load for both is the same.

□

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_i) imposed to a task τ_i can vary depending on which resource τ_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 $\mathcal{R}^i$ denote a set of resources whose resource ceilings are i. Suppose a resource $R_k \in \mathcal{R}^i$ yields the greatest blocking among all the elements of $\mathcal{R}^i$. 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 $\mathcal{R}^i$, i.e.,*

$$\Big(b_{i,k} = \max_{\forall R_j \in \mathcal{R}^i}\{b_{i,j}\}\Big) \rightarrow \Big(w_{k,i} = \max_{\forall R_j \in \mathcal{R}^i}\{w_{j,i}\}\Big). \quad (10)$$

Proof The $w_{j,i}$ depends on two parameters (see Eq. (1)); 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 $R_j \in \mathcal{R}^i$, 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 τ_i. Therefore, it does not change the request 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 2 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 [4]). The algorithm then generates an interface candidate (Q^*, X^*) based on the current resource ceilings (line 4 and 5). This new interface candidate is added into a list (line 6). Such addition can make some candidate redundant according to Lemma 1, and those redundant candidates are removed (line 7). Let R^* denote the resource that determines X^* in line 5, 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.

- calculateBudget($S_s, P_s, \mathcal{RC}_s$) returns the smallest subsystem budget that satisfies Eq. (2).
- increaseCeiling$X^*(\mathcal{RC}_s)$ 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, $Q^*, X^*, \mathcal{RC}_s$) adds new interface in the interface list array.
- removeRedundant(Interface) removes all redundant interfaces from the interface list.

```
1: RC_s = {rc_1, ..., rc_m} // rc_j=initial ceiling of R_j using SRP
2: num = 0
3: do
4:       Q* = calculateBudget(S_s, P_s, RC_s)
5:       X* = max{w_{1,rc_1}, ..., 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 2. The ICG algorithm.

$\mathcal{T}$	C_i	T_i	R_j	$c_{i,j}$	$\mathcal{T}$	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	35	500	R_1	10
τ_5	1	160	-	0	τ_6	2	150	-	0

Table 1. Example task set parameters

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 $(Q_{s,1} = 51, X_{s,1} = 102)$, 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 $X_{s,i} = w_1$ but we increase the ceiling of rc_2 according to Lemma 3. The algorithm selects the interface candidates from steps $3, 4$ and 5.

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

$Step$	rc_1	rc_2	w_1	w_2	$Q_{s,i}$	$X_{s,i}$
1	4	1	13	102	51	102
2	4	2	13	52	51	52
3	4	4	13	7	51	13
4	5	5	12	6	52.5	12
5	6	6	10	4	56	10

Table 2. Example algorithm

Lemma 4 *Let $\mathcal{IC}$ denote a set of up to n interface candidates that are generated by the ICG algorithm of Figure 2. 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 $Q'_s + X'_s <= Q^*_s + X^*_s$ and $X'_s <= X^*_s$. Then, (Q'_s, X'_s) will never produce not only a larger $\mathtt{RBF}_s(t)$ 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. (4)-(6). Then, the following lemma directly follows:

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

Proof Looking at Eq. (4), we can decrease $\mathtt{LBF}_s(t)$ to decrease the system load by decreasing the blocking B_s and/or $\mathtt{RBF}_s(t)$. For the blocking, using the interface $Q_{s,i}, X_{s,i}$ may increase the blocking on the higher priority subsystems because $X_{s,i} > X_{s,j}$. For $\mathtt{RBF}_s(t)$, it will be increased if we use $Q_{s,i}, X_{s,i}$ because $(Q_{s,i} + X_{s,i}) > (Q_{s,j} + X_{s,j})$ see Eq. (5). For this we can conclude that we can remove the interface $(Q_{s,i}, X_{s,i})$ since it will not reduce the system load compared with the other interfaces. □

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

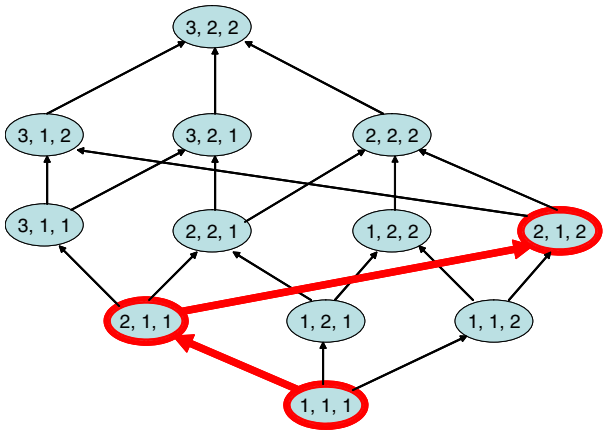

Figure 3. Search space for a system consisting of 3 subsystems.

introduce a variable f^i_k to denote an element of CF_i, i.e., $\mathsf{CF}_i = \{f^i_1, \ldots, f^i_N\}$. 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 3 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 3, $\mathsf{CF}_0 = \{1, 1, 1\}$ (see line 2 of Figure 4).

Iteration step. The ICS algorithm transits from $(i-1)$-th step to i-th step, increasing only one element of CF_{i-1} in value by one. In Figure 3, the arrows with bold lines illustrate the path that ICS can take. For instance, ICS moves from the initialization step ($\mathsf{CF}_0 = \{1, 1, 1\}$) to the first step ($\mathsf{CF}_1 = \{2, 1, 1\}$). Then, the ICS algorithm excludes the two sibling nodes of 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 δ_i denote the only single element whose value increases by one between CF_{i-1} and CF_i, i.e.,

$$f^i_k = \begin{cases} f^{i-1}_k + 1 & \text{if } k = \delta_i, \\ f^{i-1}_k & \text{otherwise.} \end{cases} \quad (11)$$

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 CF_{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 CF_i can cause the system load to be minimized.

Let $\mathsf{load}_{\mathsf{sys}}(i)$ denote the value of $\mathsf{load}_{\mathsf{sys}}$ when a configuration CF_i is used as a *system interface*. We are now interested in reducing the value of $\mathsf{load}_{\mathsf{sys}}(i-1)$. Let s^* denote the subsystem S_{s^*} that has the largest *processor request bound* among all subsystems. That is, $\mathsf{load}_{\mathsf{sys}}(i-1) = \alpha_{s^*}$ (see Eq. (9)). We can find such S_{s^*} by evaluating the *processor request bound*'s of all subsystems (in line 5 of Figure 4).

By the definition of s^*, we can reduce the value of $\mathsf{load}_{\mathsf{sys}}(i-1)$ by reducing the value of $\mathtt{LBF}_{\mathtt{s}^*}(\mathtt{t})$. There are two potential ways to reduce the value of $\mathtt{LBF}_{\mathtt{s}^*}(\mathtt{t})$. From the definition of $\mathtt{LBF}_{\mathtt{s}}(\mathtt{t})$ in Eq. (4), one is to reduce its maximum blocking B_{s^*} and the other is to reduce the subsystem CPU demands ($\mathtt{RBF}_{s^*}(t)$). A key aspect of this algorithm is that it always reduces the blocking part, but does not reduce the request bound function 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., $B_{s^*} = X_{k^*} = \max\{X_j \mid \text{for all} X_s \in LPS(s^*)\}$[3], 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 4).

When such S_{k^*} is found, it then checks whether the X_{k^*} can be further reduced (in line 7 of Figure 4). 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 $\mathsf{load}_{\mathsf{sys}}$ 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 $O(N * m')$ iterations, where N is the number of subsystems and m' is the greatest number of interface candidates of a subsystem among all.

[3]If more than one lower priority subsystem impose the same maximum blocking on Ss^*, 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 X_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.
- $\mathsf{subsystemWithMaxLoad}()$ returns the subsystem S_{s^*} that has the greatest *processor request bound* among all subsystems, i.e., $\mathsf{load}_{\mathsf{sys}} = \alpha_{s^*}$.
- $\mathsf{maxBlockingSubsystemToSysload}(\mathsf{s}^*)$ 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 ∈ 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. (9)
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 ICS algorithm.

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.,

$$\mathcal{AS} = IC_1 \times \cdots \times IC_n. \tag{12}$$

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 ICS algorithm at an arbitrary i-th iteration step.

• $\widehat{IC}_k^i$ 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 dur-

ing 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, f_k^i 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_k^i from the end of an i-th step. Then, $\widehat{IC}_k^i$ is defined as

$$\widehat{IC}_k^i = \{f_k^i, \ldots, max_k\} \text{ for all } k = 1, \ldots, n, \quad (13)$$

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 CF_i chosen at the i-th step. Then, XP_i is defined as

$$\mathsf{XP}_i = (\widehat{IC}_1^i \times \cdots \times \widehat{IC}_n^i) \setminus \mathsf{CF}_i. \quad (14)$$

• 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 $\mathsf{XP}_i \subset \mathsf{XP}_{i-1}$. 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 $f_{\delta_i}^{i-1}$ to $f_{\delta_i}^i$. Then, a subset of XP_i that contains the value of $f_{\delta_i}^{i-1}$, is excluded at the i-th step. RM_i is defined as

$$\mathsf{RM}_i = (\widehat{IC}_1^{(i-1)*} \times \cdots \times \widehat{IC}_n^{(i-1)*}) \setminus \{\mathsf{CF}_{i-1}\}, \text{ where} \quad (15)$$

$$\widehat{IC}_k^{(i-1)*} = \begin{cases} \{f_k^{i-1}\} & \text{if } k = \delta_i, \\ \widehat{IC}_k^i & \text{otherwise.} \end{cases} \quad (16)$$

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

• 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.,

$$\mathsf{AH}_i = \{\mathsf{CF}_1, \ldots, \mathsf{CF}_i\}. \quad (17)$$

• AR_i Let 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.,

$$\mathsf{AR}_i = \mathsf{RM}_{(i-1)} \cup \mathsf{RM}_i, \text{ where } \mathsf{AR}_0 = \phi. \quad (18)$$

We define partial ordering between interface candidates as follows:

Definition 2 *A interface candidate* $\mathsf{sc} = \{c_1, \ldots, c_n\}$ *is said to be* strictly precedent *of another interface 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 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 (*RM_i*), and any excluded solution candidate* $r \in \mathsf{RM}_i$ *does not yield a smaller system load than that by* CF_{i-1}.

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

Firstly, we wish to show that $\mathtt{RBF}_{\mathtt{s}_i^*}(\mathtt{t})$ does not decrease when we transform CF_{i-1} to any interface 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 $\mathsf{CF}_{i-1} \prec r$. Then, it follows that $\mathtt{RBF}_{\mathtt{s}_i^*}(\mathtt{t})$ never decreases when CF_{i-1} changes to r.

Secondly, we wish to show that when we change CF_{i-1} to any interface candidate $r \in \mathsf{RM}_i$, $B_{s_i^*}$ does not decrease. As shown in line 6 in Figure 4, the ICS algorithm finds the subsystem S_{δ_i} that generates the maximum blocking to for subsystem $S_{s_i^*}$. Then, the algorithm increases $f_{\delta_i}^{i-1}$ 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 $f_{\delta_i}^{i-1}$, rather than the value of $f_{\delta_i}^i$. This means that $B_{s_i^*}$ never decreases when we change CF_{i-1} to r. □

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 (*$\mathsf{xp} \in \mathsf{XP}_f$*) does not yield a smaller system load than* CF_f *does.*

Proof As explained in the proof of lemma 6, there are two ways to reduce $\mathsf{load}_{\mathsf{sys}}$ (i.e., $\mathtt{LBF}_{\mathtt{s}_f^*}(\mathtt{t})$).

One is to reduce $\mathtt{RBF}_{\mathtt{s}_f^*}(\mathtt{t})$ in Eq. (5) . However, it does not decrease, since $\mathsf{CF}_f \prec \mathsf{xp}$ for all $\mathsf{xp} \in \mathsf{XP}_f$.

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 CF_f changes to any xp. □

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

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

$$\mathsf{XP}_i = \mathsf{XP}_{i-1} \setminus (\mathsf{RM}_i \cup \{\mathsf{CF}_i\}). \quad (19)$$

Proof The ICS algorithm transforms CF_{i-1} to CF_i at an i-th step by increasing the value of its δ_i-th element. Then, we have

$$\widehat{IC}_k^i = \begin{cases} \widehat{IC}_k^{i-1} \setminus \{f_k^{i-1}\} & \text{if } k = \delta_i, \\ \widehat{IC}_k^{i-1} & \text{otherwise.} \end{cases} \quad (20)$$

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

$$\begin{aligned} \mathsf{XP}_i^* &= \widehat{IC}_1^i \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i \\ &= \left(\widehat{IC}_1^{i-1} \setminus \{f_1^{i-1}\}\right) \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i \\ &= \left(\widehat{IC}_1^{i-1} \times \widehat{IC}_2^{i-1} \times \cdots \times \widehat{IC}_n^{i-1}\right) \setminus \\ &\quad \left(\{f_1^{i-1}\} \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i\right) \\ &= \mathsf{XP}_{i-1}^* \setminus \mathsf{RM}_i^* \\ &= \left(XP_{i-1} \cup \{\mathsf{CF}_{i-1}\}\right) \setminus \left(\mathsf{RM}_i \cup \{\mathsf{CF}_{i-1}\}\right) \\ &= XP_{i-1} \setminus \mathsf{RM}_i \,. \end{aligned} \quad (21)$$

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

$$XP_i = XP_{i-1} \setminus (\mathsf{RM}_i \cup \{\mathsf{CF}_i\}) \,. \quad (22)$$

□

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 9 *At an arbitrary i-th step, the sets of AR_i, AH_i, and XP_i include all possible interface candidates.*

$$\mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i = \mathcal{AS} \quad (23)$$

Proof We will prove this lemma by using mathematical induction. As a base step, we wish to show Eq. (23) is true, when $i = 1$. Note that $\mathsf{AR}_0 = \phi$ and $\mathsf{AH}_0 = \{\mathsf{CF}_0\}$. In addition, $\mathsf{XP}_0 = \mathcal{AS} \setminus \mathsf{CF}_0$, according to Eq. (14). It follows that $\mathsf{AR}_0 \cup \mathsf{AH}_0 \cup \mathsf{XP}_0 = AP$.

We assume that Eq. (23) 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.,

$$\mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i = \mathsf{AR}_{i+1} \cup \mathsf{AH}_{i+1} \cup \mathsf{XP}_{i+1}. \quad (24)$$

According to the definitions AH_{i+1}, AR_{i+1}, and XP_{i+1} (see Eq. (17), (18) and (19)), we can rewrite the right-hand side of Eq. (24) as follows:

$$\begin{aligned} &\mathsf{AR}_{i+1} \cup \mathsf{AH}_{i+1} \cup \mathsf{XP}_{i+1} \\ = \;& \left(\mathsf{AR}_i \cup \mathsf{RM}_{i+1}\right) \cup \left(\mathsf{AH}_i \cup \{\mathsf{CF}_{i+1}\}\right) \cup \\ & \left(\mathsf{XP}_i \setminus (\mathsf{RM}_{i+1} \cup \{\mathsf{CF}_{i+1}\})\right) \\ = \;& \mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i \,. \square \end{aligned}$$

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 $opt \notin \mathsf{AR}_f$ 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 6, $\mathsf{load}_{\mathsf{sys}}(\mathsf{CF}_{i-1}) < \mathsf{load}_{\mathsf{sys}}(opt)$, which contradicts the definition of *opt*. Hence, $opt \notin \mathsf{AR}_f$.

Suppose $opt \in \mathsf{XP}_f$. Then, according to Lemma 7, it should be $\mathsf{load}_{\mathsf{sys}}(\mathsf{CF}_f) < \mathsf{load}_{\mathsf{sys}}(opt)$, which contradicts the definition of *opt* as well. Hence, $opt \notin \mathsf{AR}_f$.

According to Lemma 9, it follows that $opt \in \mathsf{CF}_f$. □

8 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 [6], and we are extending towards another overrun mechanism (with-payback version) [6]. Unlike with the former overrun mechanism, the intra- and inter-subsystem aspects of the tradeoff are not clearly separated with the latter mechanism. The latter mechanism changes the way of a subsystem's own contributing to the system load (i.e., Eq. (5)), and this requires appropriate changes to the post-processing part of the ICG algorithm. We refer interested readers to our technical report [20] for more details. We are investigating how to make changes to the post-processing part in ways that require less subsequent changes to the ICS algorithm.

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 *WPDRTS*, 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] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *RTSS*, 2007.

[9] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hieararchical real-time components. In *EMSOFT'06*.

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

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

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

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

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

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

[16] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 2005.

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

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

[19] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *IECON87*, 1987.

[20] I. Shin, M. Behnam, T. Nolte, and M. Nolin. On optimal hierarchical resource sharing in open environments. Technical report, 2008. Available at http://www.idt.mdh.se/∼tnt/rtss08long.pdf.

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

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

[23] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems*, 7(3):(30)1–39, April 2008.

[24] F. Zhang and A. Burns. Analysis of hierarchical edf pre-emptive scheduling. In *RTSS*, 2007.