Mälardalen University Press Licentiate Thesis
No.94

Hierarchical Real-Time Scheduling and Synchronization

Moris Behnam

October 2008

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 978-91-86135-09-6
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

The Hierarchical Scheduling Framework (HSF) has been introduced to enable compositional schedulability analysis and execution of embedded software systems with real-time constraints. In this thesis, we consider a system consisting of a number of semi-independent components called subsystems, and these subsystems are allowed to share logical resources. The HSF provides CPU-time to the subsystems and it guarantees that the individual subsystems respect their allocated CPU budgets. However, if subsystems are allowed to share logical resources, extra complexity with respect to analysis and run-time mechanisms is introduced.

In this thesis we address three issues related to hierarchical scheduling of semi-independent subsystems. In the first part, we investigate the feasibility of implementing the hierarchical scheduling framework in a commercial operating system, and we present the detailed figures of various key properties with respect to the overhead of the implementation.

In the second part, we studied the problem of supporting shared resources in a hierarchical scheduling framework and we propose two different solutions to support resource sharing. The first proposed solution is called SIRAP, a synchronization protocol for resource sharing in hierarchically scheduled open real-time systems, and the second solution is an *enhanced overrun mechanism*.

In the third part, we present a resource efficient approach to minimize system load (i.e., the collective CPU requirements to guarantee the schedulability of hierarchically scheduled subsystems). Our work is motivated from a trade-off between reducing resource locking times and reducing system load. We formulate an optimization problem that determines the resource locking times of each individual subsystem with the goal of minimizing the system load subject to system schedulability. We present linear complexity algorithms to find an optimal solution to the problem, and we prove their correctness.

To the memory of my mother

Acknowledgment

This thesis would not been possible without the help of my supervisors Prof. Mikael Sjödin and Dr. Thomas Nolte and the collaboration with Dr. Insik Shin. I would like to thank Mikael Sjödin for his advices and invaluable input to my research. Thomas, thank you very much for the supporting, encouraging, helping and always finding time to guide me.

A special thank goes to Insik for all the intensive discussions and fruitful cooperation. I would like to say how much I have appreciated working with Thomas, Insik and Mikael, and I have learned a lot from them.

I want to thank the PROGRESSers; Prof. Hans Hansson for his great leading of the PROGRESS center, and Prof. Ivica Crnkovic, Prof. Christer Norström, Prof. Sasikumar Punnekkat, Prof. Paul Pettersson, Dr. Jan Gustafsson, Dr. Andreas Ermedahl and Dr. Cristina Seceleanu.

Also, I would like to thank Prophs'ers (PROGRESS PhD students) Hüseyin Aysan, Andreas Hjertström, Séverine Sentilles, Farhang Nemati, Aneta Vulgarakis, Marcelo Santos, Stefan Bygde, Yue Lu and also the new PhD students Mikael Åsberg, Jagadish Suryadevara, Aida Causevic. We had a lot of fun especially when we arranged the social activities and student surprise for the PROGRESS trips and also when I participated with some of you in PhD schools and conferences.

Many thanks go to Dr. Damir Isovic for informing me about the PhD position and for the very nice recommendation letter that I received from him when I applied for that position.

I would also like to thank the my colleagues at the department for the nice time that I had in the department and special thank goes to the administrative staff, in particular Harriet Ekwall and Monica Wasell for their help in practical

issues.

I would like to express my special gratitude to Dr. Reinder J. Bril at Eindhoven University of Technology, for our collaboration and his constructive comments and discussions.

During my PhD studies, I have participated in 7 conferences, 3 PhD schools and 3 project trips in 7 different countries. Related to this, I would like to thank Dr. Johan Fredriksson and Dr. Daniel Sundmark for being great travel companions.

Finally, my deepest gratitude goes to my wife Rasha and my kids Dany and Hanna for all their support and love.

This work has been supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

Moris Behnam
Västerås, October, 2008

Contents

I

Thesis

Chapter 1

Introduction

Hierarchical scheduling has shown to be a useful approach in supporting modularity of real-time software [1] by providing temporal partitioning among applications. In hierarchical scheduling, a system can be hierarchically divided into a number of subsystems that are scheduled by a global (system-level) scheduler. Each subsystem contains a set of tasks that are scheduled by a local (subsystem-level) scheduler. The Hierarchical Scheduling Framework (HSF) allows for a subsystem to be developed and analyzed in isolation, with its own local scheduler. At a later stage, using a global scheduler such as Fixed Priority Scheduling (FPS), Earlier Deadline First (EDF) or Time Division Multiple Access (TDMA), it allows for the integration of multiple subsystems without violating the temporal properties of the individual subsystems. The subsystem integration involves a system-level schedulability test, verifying that all timing requirements are met. This approach by isolation of tasks within subsystems, and allowing for their own scheduler, has several advantages including [2]:

- It allows for the usage of the best scheduler (e.g., FPS, EDF or TDMA) that fit the requirements of each subsystem.

- By keeping a subsystem isolated from other subsystems, and keeping the subsystem local scheduler, it is possible to re-use a complete subsystem in a different application[1] from where it was originally developed.

[1] Assuming that the timing parameters of the internal tasks of the subsystem will not be changed when the subsystem is re-used in a different application.

- Hierarchical scheduling frameworks naturally support *concurrent development* of subsystems.

Over the years, there has been a growing attention to HSFs for real-time systems. Deng and Liu [3] proposed a two-level hierarchical scheduling framework for open systems, where subsystems may be developed and validated independently in different environments. Kuo and Li [4] presented schedulability analysis techniques for such a two-level framework with the fixed-priority global scheduler. Lipari and Baruah [5, 6] presented schedulability analysis techniques for the EDF-based global schedulers. Mok *et al.* [7, 8] proposed the bounded-delay virtual processor model to achieve a clean separation in a multi-level HSF. In addition, Shin and Lee [1] 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 [9, 10, 11] and under EDF scheduling [1, 12]. Being central to this thesis, the virtual periodic resource model is presented in detail in Chapter 3. More recently, Easwaran *et al.* [13] introduced Explicit Deadline Periodic (EDP) virtual processor model. However, a common assumption shared by all above studies is that tasks are independent.

In this thesis we address the challenges of enabling efficient compositional integration preserving temporal behavior for independently developed semi-independent subsystems (i.e., subsystems are allowed to synchronize by the sharing of logical resources) in open systems where subsystems can be developed independently. Efficient compositional integration means that the system should require as little CPU-resources as possible, allowing more subsystems to be integrated in a single processor. Achieving efficient compositional integration makes the HSF a cost-efficient approach applicable for a wide domain of applications, including, automotive, automation, aerospace and consumer electronics.

There have been studies on supporting resource sharing within subsystems [9, 4] and across subsystems [14, 15, 16] in HSFs. Davis and Burns [14] proposed the Hierarchical Stack Resource Policy (HSRP) supporting global resource sharing on the basis of an overrun mechanism. The schedulability analysis associated with the HSRP does not support independent subsystem development (i.e., when performing schedulability analysis for internal tasks of a subsystem using HSRP, information about other subsystems should be provided). Fisher *et al.* [16] proposed the BROE server in order to handle sharing of logical resources in a HSF. A detailed description of these protocols and a comparison between our proposed protocol and these protocols is

presented in Chapter 4.

Our overall goal of this thesis is to propose a scheduling framework and synchronization protocols that are able to fulfill the following requirements;

- With acceptable implementation overhead, it should be possible to implement the HSF in commercial real-time operating systems.
- The framework should support sharing of logical resources between subsystems while preserving the timing predictability and thereby allowing for temporal requirements of the system.
- No knowledge about the parameters of other subsystems is required when developing a subsystem, even in the case when there are dependencies between subsystems (semi-independent subsystems) inherent in the sharing of logical resources.
- The HSF should use the CPU-resources efficiently by minimizing the collective CPU requirement (i.e., system load) necessary to guarantee the schedulability of an entire framework.

1.1 Contributions

The contributions presented in this thesis can be divided into three parts:

Implementation Over the years, there has been a growing attention to HSFs for real-time systems. However, up until now, those studies have mainly worked on various aspects of HSFs from a theoretical point of view. To our knowledge, there are very few studies that focus on the implementation of HSF, especially looking at what can be done with commercial operating systems.

We present our work towards a full implementation of the hierarchical scheduling framework in the VxWorks commercial operating system without changing or modifying the kernel of the operating system. Moreover, to show the efficiency of the implementation, we measure the overheads imposed by the implementation as a function of number of subsystems and number of tasks for both FPS and EDF local and global schedulers.

Supporting shared resources Allowing tasks from different subsystems to share logical resources imposes more complexity for the scheduling of subsystems. A proper synchronization protocol should be used to prevent unpredictable timing behavior of the real-time system. Since there are dependencies

between subsystems though sharing of logical resources, using the protocol with the HSF should not require any information from other subsystems when developing a subsystem in order to not violate the requirement of developing subsystems independently (support open systems).

We present the SIRAP protocol, a novel approach to allow synchronization of semi-independent hierarchically scheduled subsystems. We present the deduction of bounds on the timing behaviour of SIRAP together with accompanying formal proofs and we evaluate the cost of using this protocol in terms of the extra CPU-resources that is required by the usage of the protocol.

In addition to SIRAP, we extend the schedulability analysis of HSRP [14] so that it allows for independent analysis of individual semi-independent subsystems. And also, we propose an enhanced overrun mechanism that gives two benefits (compared with the old version of overrun mechanism): (1) it may increase schedulability within a subsystem by providing CPU allocations more efficiently, and (2) it can even accept subsystems which developed their timing requirements without knowing that the proposed modified overrun mechanism would be employed in the system.

Efficient CPU-resources usage As mentioned previously, one of the requirements that the proposed framework should provide, is to minimize the system load. This can be achieved by finding optimal subsystem timing interfaces (specifies the collective temporal requirements of a subsystem) that minimize the system load. Supporting shared resources across subsystems produces interference among subsystems which imposes more CPU demands for each subsystem and makes the problem of minimizing the system load more complex.

We identify a tradeoff between reducing the time that a subsystem can block other subsystems when accessing a shared resource (locking time which is a part of subsystem timing interface) and decreasing the system load. Selecting the optimal subsystem interface for a subsystem requires information from other subsystems that the subsystem will interact with. However, the required information may not be available during the development stage of the subsystem and in this case we may not be able to select the optimal interface. To solve the problem of selecting an optimal interface for each subsystem, we propose a two-step approach towards the system load minimization problem. In the first step, a set of interface candidates, that have a potential to produce an optimal system load, is generated for each subsystem in isolation. In the second step, one interface will be selected for each subsystem from its own candidates to find the minimum resulting system load. We provide one algorithm for each step and we also prove the correctness and the optimality of the

provided algorithms formally.

1.2 Outline of thesis

The outline of this thesis is as follows: in Chapter 2 we explain and define the basic concepts for real-time systems and the terms that will be used throughout this thesis and in addition we present the system model. In Chpater 3 we describe the hierarchical scheduling framework and the associated schedulability analysis assuming that the subsystems are fully independent. In Chapter 4 we address the problem of allowing dependency through sharing logical resource between subsystem and we present some solutions for this problem. In Chapter 5 we present our conclusion and suggestions for future work. We present the technical overview of the papers that are included in this thesis in Chapter 6 and we present these papers in Chapters 7-10.

Chapter 2

Background

In this chapter we present some basic concepts concerning real-time systems, as well as some methods that will be used in the next chapters.

2.1 Real-time systems

A real-time system is a computing system whose correctness relies not only on the functionality, but also on timeliness, i.e., the system should produce correct results at correct instances of time. Real-time systems are usually constructed using concurrent programs called *tasks* and each task is supposed to perform a certain functionality (for example reading a sensor value, computing output values, sending output values to other tasks or devices, etc). A real-time task should complete its execution before a predefined time called *deadline*.

Real-time tasks can be classified according to their timing constraint to either *hard* real-time tasks or *soft* real-time tasks. For hard real-time tasks, all tasks should complete their execution before their deadlines otherwise a catastrophic consequence may occur. However, for soft real-time tasks, it is acceptable that deadlines are missed which may degrade the system performance, for example consider a mobile phone where missing some deadlines will decrease the quality of the sound. Many systems contain a mix of hard and soft real-time tasks.

A real-time task consists of an infinite sequence of activities called jobs, and depending on the way of task triggering, real-time tasks are modeled as either an *aperiodic task* or a *sporadic task* or a *periodic task*:

- Aperiodic tasks are triggered at arbitrary times, with no known minimum inter-arrival time.
- Sporadic tasks have known minimum inter-arrival time.
- Periodic tasks have a fixed inter-arrival time called period.

Depending on the task model, each task is characterized by timing parameters including task period (periodic task), worst case execution time, deadline, etc.

2.2 System model

In this thesis we focus on scheduling of a single node. Each node is modeled as a system $\mathcal{S}$ which consists of one or more subsystems $S_s \in \mathcal{S}$. The scheduling framework is a two-level hierarchical scheduling framework as shown in Fig 2.1. During run-time, the system level scheduler (Global scheduler) selects which subsystem that will access the CPU-resources.

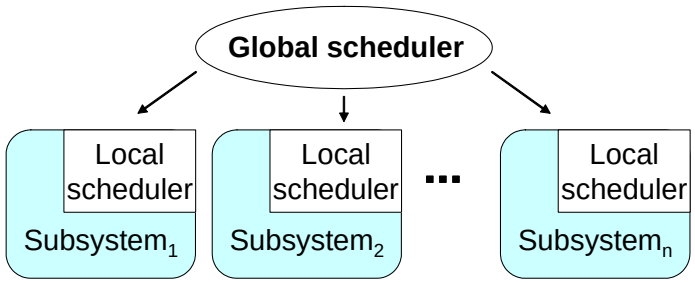


Figure 2.1: Two-level hierarchical scheduling framework with resource sharing.

2.2.1 Subsystem model

A subsystem S_s consists of a task set and a scheduler. Once a subsystem is assigned the processor, the corresponding local scheduler will select which

task that will be executed. Each subsystem S_s is associated with a periodic processor model (abstraction) $\Gamma_s(P_s, Q_s)$, where P_s and Q_s are the subsystem period and budget respectively. This abstraction $\Gamma_s(P_s, Q_s)$ specifies the collective temporal requirements of a subsystem and it is used as an interface between the subsystem and the global scheduler (we refer to this abstraction as *subsystem timing interface*).

2.2.2 Task model

In this thesis, we consider a deadline-constrained sporadic hard real-time 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 for one job, D_i is a relative deadline ($C_i \leq D_i \leq T_i$) by which each job must have finished its execution. Each task is allowed to access one or more logical resources 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.

2.2.3 Shared resources

The presented hierarchical scheduling framework allows sharing of logical resource between tasks in a mutually exclusive manner. To access a resource R_j, a task must first lock the resource, and when the task no longer needs the resource it is unlocked. The time during which a task holds a lock is called a critical section time. Only one task at a time may be inside a critical section corresponding to a specific resource. A resource that is used by tasks in more than one subsystem is denoted a *global shared resource*. A resource only used within a single subsystem is a *local shared resource*. We are concerned only with global shared resources and will simply denote them by shared resources.

2.3 Scheduling algorithms

In a single processor, the CPU can not be assigned to more than one task to be executed at the same time. If a set of tasks are ready to execute then a scheduling criterion should be used to define the execution order of these tasks. The scheduling criterion uses a set of rules defined by a scheduling algorithm to determine the execution order of the task set. If all tasks complete their execution before their deadlines then the schedule is called a feasible schedule and

the tasks are said to be schedulable. If the scheduler permit other tasks to interrupt the execution of the running task (task in execution) before completing of its execution then the scheduling algorithm is called a preemptive algorithm, otherwise it is called a non-preemptive scheduling algorithm.

Real-time scheduling algorithms fall in two basic categories; online schedule and off-line schedule [17].

2.3.1 Online scheduling

For online scheduling, the order of task execution is determined during run-time according to task priorities. The priorities of tasks can be static which means that the priorities of tasks will not change during run-time. This type of scheduling algorithm is called Fixed Priority Scheduling (FPS) and both Rate Monotonic (RM) scheduling [18] and Deadline Monotonic (DM) [19] use this type of scheduling. The task priorities can be dynamic which means that they can change during run-time, and Earlier Deadline First (EDF) [18] is an example of such scheduler.

RM and DM scheduling algorithms In RM, the priorities of the tasks are assigned according to their periods; the priority of a task is proportional to the inverse of the task period such that the task with shorter period will have higher priority than the tasks with longer period. The priority of a task is fixed during the run time. The RM scheduling algorithm assumes that tasks periods equals to tasks deadlines. Another FPS algorithm is DM which is similar to RM but the priority depends on the task relative deadlines instead of periods.

The schedulability analysis for each task using RM or DM is as follows [20];

$$\forall \tau_i \in \Gamma, 0 < \exists t \leq D_i \ \ \texttt{dbf}(i,t) \leq t. \tag{2.1}$$

where Γ is the set of tasks that will be scheduled and D_i is the relative deadline of the task τ_i and $\texttt{dbf}(i,t)$ is evaluated as follows;

$$\texttt{dbf}(i,t) = C_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k, \tag{2.2}$$

where C_i is the worst case execution time of the task τ_i and T_i is the task period and $\texttt{HP}(i)$ is the set of tasks with priority higher than that of τ_i.

EDF scheduling algorithm In this scheduling algorithm, the task that has earlier deadline among all tasks that are ready to execute, will execute first. The priority of the task is dynamic and can be changed during run-time depending on the deadline of the task instant and other released tasks ready for execution. The schedulability test for a set of tasks that use EDF is shown in Eq. (2.3) [21] which includes the case when task deadlines are allowed to be less than or equal to task periods.

$$\forall t > 0, \quad \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i \leq t \tag{2.3}$$

2.3.2 Offline scheduling

In offline scheduling, a schedule is created before run-time. The scheduling algorithm can take into consideration the timing constrains of real-time tasks such as execution time, deadline, precedence relation (if a task should execute always before another task), etc. The resulting execution sequence is stored in a table and then dispatched during run-time. Finding a feasible schedule using offline scheduling should be done up to the hyper-period (LCM) of task periods, and then, during the run-time, this hyper-period is repeated regularly.

2.4 Logical resource sharing

A *resource* is any software structure that can be used by a task to advance its execution [22]. For example a resource can be a data structure, flash memory, a memory map of a peripheral device. If more than one task use the same resource then that resource is called *shared resource*. The part of task's code that uses a shared resource is called critical section. When a job enters a critical section (starts accessing a shred resource) then no other jobs, including the jobs of higher priority tasks, can access the shared resource until the accessing job exits the critical section (mutual exclusion method). The reason is to guarantee the consistency of the data in the shared resource and this type of shared resource is called nonpreemptable resource. For preemptive scheduling algorithms, sharing logical resources cause a problem called *priority inversion*. The priority inversion problem happen when a job with high priority wants to access a shared resource that is currently accessed by another lower priority job, so the higher priority job will not be able to preempt the lower priority job. The higher priority job will be blocked until the lower priority job release

the shared resource. The time that the high priority job will be blocked can be unbounded since other jobs with intermediate priority that do not access the shared resource can preempt the low priority job while it is executing inside its critical section. As a result of the priority inversion problem, the higher priority job may miss its deadline. A proper protocol should be used to synchronize the access to the shared resource in order to bound the waiting time of the blocked tasks. Several synchronization protocols, such as the Priority Inheritance Protocol (PIP) [23], the Priority Ceiling Protocol (PCP) [24] and the Stack Resource Policy (SRP) [25], have been proposed to solve the problem of priority inversion. We will explain the SRP protocol in details, a protocol central for this thesis, suitable for RM, DM, and EDF scheduling algorithms.

2.4.1 Stack resource policy

To describe how SRP [25] works, we first define some terms that are used with SRP.

- *Preemption level*. Each task τ_i has a preemption level which is a static value and proportional to the inverse of task relative deadline $\pi_i = 1/D_i$, where D_i is a relative deadline of task τ_i.
- *Resource ceiling*. Each shared resource R_j is associated with a resource ceiling which equal to the highest preemption level of all tasks that use the resource R_j; $rc_j = \max\{\pi_i | \tau_i \text{ accesses } R_j\}$.
- *System ceiling*. System ceiling is a dynamic parameter that change during execution. The system ceiling is equal to the currently locked highest resource ceiling in the system. If at any time there is no accessed shared resource then the system ceiling would be equal to zero.

According to SRP, a job J_i generated by task τ_i can preempt the currently executing job J_k only if J_i is a higher-priority job of J_k and the preemption level of τ_i is greater than the current subsystem ceiling.

2.4.2 Resource holding time

For a set of tasks that uses the SRP protocol, the duration of time that a task τ_i locks a shared resource, is called *resource holding time* [26, 27] which equals to the maximum task execution time inside a critical section plus the interference (preemption inside the critical section) of higher priority tasks that have preemption level greater than the ceiling of locked resource. The resource holding

time can be computed depending on the scheduling algorithm in use, as shown below;

Under FPS scheduling the resource holding time h_j of a shared resource R_j is [26];

$$W_j^{FPS}(t) = cx_j + \sum_{k=rc_j+1}^{n} \lceil \frac{t}{T_k} \rceil \cdot C_k, \tag{2.4}$$

where cx_j is the maximum worst-case execution time inside the critical section of all tasks that access resource R_j and n is the number of tasks.

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

$$W_j^{FPS}(t^*) = t^*. \tag{2.5}$$

Under EDF scheduling the resource holding time h_j of a shared resource R_j is [27];

$$W_j^{EDF}(t) = cx_j + \sum_{k=rc_j+1}^{n} \Big(min\Big(\Big\lceil \frac{t}{T_k} \Big\rceil, \Big\lfloor \frac{D_i - D_k}{T_k} \Big\rfloor + 1 \Big) \Big) \cdot C_k, \tag{2.6}$$

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

$$W_j^{EDF}(t^*) = t^*. \tag{2.7}$$

An algorithm to decrease the resource holding time without violating the schedulability of the system under the same semantics as that of SRP, was presented in [26, 27]. The algorithm works as follows; it increases the resource ceiling of each shared resource to the next higher value (higher preemption level than the ceiling of the resource) in steps and in each step it checks if the schedule is still feasible or not. If the schedule is feasible then it continues increasing the ceiling of the resource until either the schedule becomes infeasible or the ceiling of the task equals to the maximum preemption level. The minimum resource holding time of a resource R_j is obtained when its resource ceiling equal to the maximum preemption level of the task set. Note that the resource holding time is a very important parameter for the hierarchical scheduling framework, as will be shown in Chapter 4.

Chapter 3

Real-Time Hierarchical Scheduling Framework

In this chapter, we will describe the HSF assuming that all tasks are fully independent, i.e., tasks are not allowed to share logical resources. While in the next chapter we will consider the problem of accessing global shared resources.

3.1 Hierarchical scheduling framework

One of the important properties that the HSF can provide is the isolation between subsystems during design time and run-time such that the subsystems are separated functionally for fault containment and for compositional verification, validation and certification. The HSF guarantees independent execution of the subsystems and it prevents one subsystem from causing a failure of another subsystem through providing the CPU-resources needed for each subsystem.

Each subsystem specifies the amount of CPU-resources that are required to schedule all internal tasks through its timing interface. And the global scheduler will provide the required CPU-resources for all subsystems as specified by the timing interfaces of the subsystems.

In the following sections, we will explain how to evaluate the subsystem timing interface and also show how to verify whether the global scheduler can supply the subsystems with required resources using global schedulability analysis.

Given a subsystem timing interface, it is required to check if the interface

can guarantee that all hard real-time tasks in the subsystem will meet their deadlines using this interface. This check is done by applying local schedulability analysis. But before presenting the local schedulability analysis, we will explain the virtual processor resource model which will be used in the local schedulability analysis.

3.2 Virtual processor model

The notion of real-time virtual processor (resource) model was first introduced by Mok *et al.* [7] to characterize the CPU allocations that a parent node provides to a child node in a hierarchical scheduling framework. The *CPU supply* of a virtual processor model refers to the amount of CPU allocations that the virtual processor model can provide. The *supply bound function* of a virtual processor model calculates the minimum possible CPU supply of the virtual processor model for a time interval length t.

Shin and Lee [1] proposed the periodic virtual processor model $\Gamma(P, Q)$, where P is a period ($P > 0$) and Q is a periodic allocation time ($0 < Q \leq P$). The capacity U_Γ of a periodic virtual processor model $\Gamma(P, Q)$ is defined as Q/P. The periodic virtual processor model $\Gamma(P, Q)$ is defined to characterize the following property:

$$\mathrm{supply}_\Gamma\big(kP, (k+1)P\big) = Q, \quad \text{where } k = 0, 1, 2, \ldots, \tag{3.1}$$

where the supply function $\mathrm{supply}_{R_s}(t_1, t_2)$ computes the amount of CPU allocations that the virtual processor model R_s provides during the interval $[t_1, t_2)$.

For the periodic model $\Gamma(P, Q)$, its supply bound function $\mathtt{sbf}_\Gamma(t)$ is defined to compute the minimum possible CPU supply for every interval length t as follows:

$$\mathtt{sbf}_\Gamma(t) = \begin{cases} t - (k+1)(P-Q) & \text{if } t \in [(k+1)P - 2Q, \\ & (k+1)P - Q], \\ (k-1)Q & \text{otherwise,} \end{cases} \tag{3.2}$$

where $k = \max\Big(\big\lceil \big(t - (P-Q)\big)/P \big\rceil, 1\Big)$. Here, we first note that an interval of length t may not begin synchronously with the beginning of period P. That is, as shown in Figure 3.1, the interval of length t can start in the middle of the period of a periodic model $\Gamma(P, Q)$. We also note that the intuition of k in Eq. (3.2) basically indicates how many periods of a periodic model can

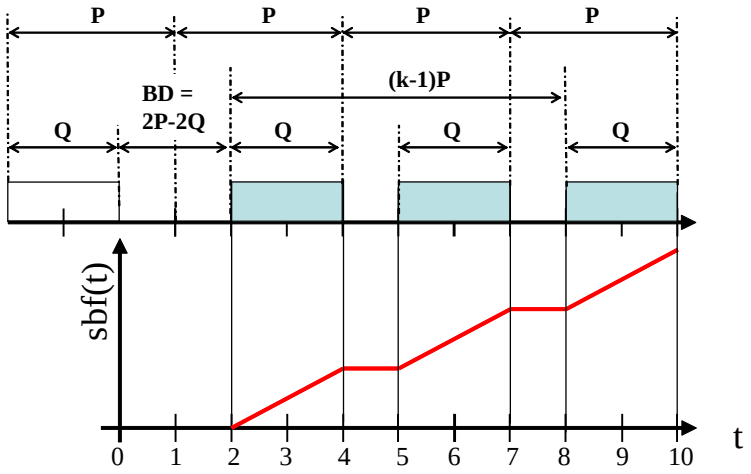


Figure 3.1: The supply bound function of a periodic virtual processor model $\Gamma(P, Q)$ for $k = 3$.

overlap the interval of length t, more precisely speaking, the interval of length $t-(P-Q)$. Figure 3.1 illustrates the intuition of k and how the supply bound function $\texttt{sbf}_\Gamma(t)$ is defined for $k = 3$.

3.3 Schedulability analysis

This section presents the schedulability analysis of the HSF, starting with local schedulability analysis needed to calculate subsystem interfaces, and finally, global schedulability analysis.

3.3.1 Local schedulability analysis

Let $\texttt{dbf}_{\texttt{EDF}}(i,t)$ denote the demand bound function of a task τ_i under EDF scheduling [28], i.e.,

$$\texttt{dbf}_{\texttt{EDF}}(i,t) = \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i. \tag{3.3}$$

The local schedulability condition under EDF scheduling is then ([1])

$$\forall t > 0 \quad \sum_{\tau_i \in \Gamma} \mathtt{dbf}_{\mathsf{EDF}}(i,t) \leq \mathtt{sbf}(t), \tag{3.4}$$

Let $\mathtt{dbf}_{\mathsf{FP}}(i,t)$ denote the demand bound function of a task τ_i under FPS [20], i.e.,

$$\mathtt{dbf}_{\mathsf{FP}}(i,t) = C_i + \sum_{\tau_k \in \mathtt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \tag{3.5}$$

where $\mathtt{HP}(i)$ is the set of tasks with higher priorities than that of τ_i. The local schedulability analysis under FPS can then easily be extended from the results of [25, 1] as follows:

$$\forall \tau_i, 0 < \exists t \leq D_i \;\; \mathtt{dbf}_{\mathsf{FP}}(i,t) \leq \mathtt{sbf}(t). \tag{3.6}$$

3.3.2 Global schedulability analysis

The global scheduler schedules subsystems in a similar way as scheduling simple real-time periodic tasks. The reason is that we are using the periodic resource model to abstract the collective timing temporal requirements of subsystems, so the subsystem can be modeled as a simple periodic task where the subsystem period is equivalent to the task period and the subsystem budget is equivalent to the task execution time. Depending on the global scheduler (if it is EDF, RM or DM), it is possible to use the schedulability analysis methods used for scheduling periodic tasks (presented in section 2.3) in order to check the global schedulability.

3.4 Subsystem interface calculation

Using HSF, a subsystem S_s is assigned fraction of CPU-resources which equals to Q_s/P_s. It is required to decrease the required CPU-resources fraction for each subsystem as much as possible without affecting the schedulability of its internal tasks. By decreasing the required CPU-resources for all subsystems, the overall CPU demand required to schedule the entire system (system load) will be decreased, and by doing this, more applications can be integrated in a single processor.

To evaluate the minimum CPU-resources fraction required for a subsystem S_s and given P_s, let $\mathsf{calculateBudget}(S_s, P_s)$ denote a function that calculates

the smallest subsystem budget Q_s that satisfies Eq. (3.4) and Eq. (3.6). Hence, $Q_s = \mathsf{calculateBudget}(S_s, P_s)$. The function is a searching function similar to the one presented in [1] and the resulting subsystem timing interface is (P_s, Q_s).

Chapter 4

Hierarchical Scheduling with Resource Sharing

In this chapter we extend the HSF that was presented in the previous chapter and allow tasks from different subsystems to share global resources. We are concerned only with global shared resources while managing of local shared resources can be done by using several existing synchronization protocols such as PIP, PCP, and SRP (see [9, 14, 4] for more details).

First, we explain the problem of supporting logical resources followed by discussing some solutions. Later, we show the effect of supporting sharing of global shared resources on the system load required to schedule the entire system.

4.1 Problem formulation

When a task access a shared resource, all other tasks that want to access the same resource will be blocked until the task that is accessing the resource releases it. To achieve a predictable real-time behaviour, the waiting time of other tasks that want to access a locked shared resource should be bounded. The traditional synchronization protocols such as PIP, PCP and SRP that are used with non-hierarchical scheduling, can not without modification, handle the problem of sharing global resources in hierarchical scheduling framework. To explain the reason, suppose a task τ_j that belongs to a subsystem S_I is holding a logical resource R_1, the execution of the task τ_j can be preempted while

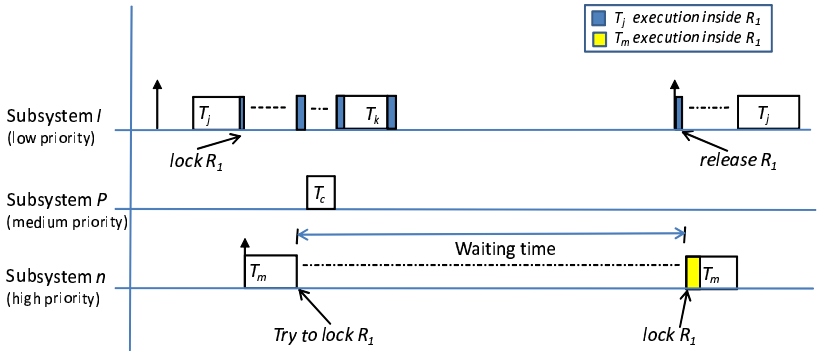


Figure 4.1: Task preemption while running inside a critical section.

τ_j is executing inside the critical section of the resource R_1 (see Fig 4.1) due to the following reasons:

1. **Inter subsystem preemption**, a higher priority task τ_k within the same subsystem preempts the task τ_j.

2. **Intra subsystem preemption**, a ready task τ_c that belongs to a subsystem S_P preempts τ_j when the priority of subsystem S_P is higher than the priority of subsystem S_I.

3. **Budget expiry inside a critical section**, if the budget of the subsystem S_I expires, the task τ_j will not be allowed to execute until the budget of its subsystem will be replenished at the beginning of the next subsystem period P_I.

The PIP, PCP and SRP protocols can only solve the problem caused by task preemption within a subsystem (case number 1) since there is a direct relationship between the priorities of tasks within the same subsystem. However, if tasks are from different subsystems (intra task preemption) then priorities of tasks belonging to different subsystems are independent of each other, which make these protocols not suitable to be used directly to solve this problem. One way to solve this problem is by using the protocols PIP, PCP and SRP between subsystems such that if a task that belongs to a subsystem lock a global resource, then this subsystem blocks all other subsystems where their internal tasks want to access the same global shared resource.

Another problem of directly applying PIP, PCP and SRP protocols is that of budget expiry inside critical section. The subsystem budget Q_I is said to *expire* at the point when one or more internal (to the subsystem) tasks have executed a total of Q_I time units within the subsystem period P_I. 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_I.

Budget expiration can cause a problem, if it happens while a task τ_j of a subsystem S_I is executing within the critical section of a global shared resource R_1. If another task τ_m, belonging to another subsystem, is waiting for the same resource R_1, this task must wait until S_I is replenished so τ_j can continue to execute and finally release the lock on resource R_1. This waiting time exposed to τ_m can be potentially very long, causing τ_m to miss its deadline.

4.2 Supporting logical resource sharing

Several mechanisms have been proposed to enable resource sharing in hierarchical scheduling framework. These mechanisms use different methods to handle the problem of bounding the waiting time of other tasks that are waiting for a shared resource. Most of them use the SRP protocol to synchronize access to a shared resource within a subsystem to solve the problem of inter subsystem preemption, and they also use SRP among subsystems to solve the problem of intra subsystem preemption. Note that the effect of using SRP with both local and global scheduling should be considered during the schedulability analysis.

In general, solving the problem of budget expiry inside a critical section is based on two approaches;

- Adding extra resources to the budget of each subsystem to prevent the budget expiration inside a critical section.
- Preventing a task from locking a shared resource if its subsystem does not have enough remaining budget.

The following sections explain these mechanisms in detail.

4.2.1 BWI

The BandWidth Inheritance protocol (BWI) [29] extends the resource reservation framework to systems where tasks can share resources. The BWI approach

uses (but is not limited to) the CBS algorithm together with a technique that is derived from the Priority Inheritance Protocol (PIP). According to BWI, each task is scheduled through a server, and when a task that executed inside lower priority server blocks another task executed in higher priority server, the blocking task will be added to the higher priority server. When the task releases the shared resource, then it will be discarded from the high priority server. For schedulability analysis, each server should be characterized by an interference time due to adding lower priority tasks in the server. This approach is suitable for systems where the execution time of a task inside critical section can not be evaluated. In addition, the scheduling algorithm does not require any prior knowledge about which shared resources that tasks will access nor the arrival time of tasks. However, BWI is not suitable for systems that consist of many hard real-time tasks. The reason is that the interference (that includes the summation of the execution times inside the critical section) from the lower priority tasks will be added to the budget of a hard real-time task server to guarantee that the task will not miss its deadline. Hence, BWI becomes pessimistic in terms of CPU-resources usage for hard real-time tasks.

4.2.2 HSRP

The Hierarchical Stack Resource Policy (HSRP) [14] extends the SRP protocol to be appropriate for hierarchical scheduling frameworks with tasks that access global shared resources. HSRP is based on the overrun mechanism which works as follows: when the budget of a subsystem expires and the subsystem has a job J_i that is still locking a global shared resource, the job J_i continues its execution until it releases the locked resource. When a job access a global shared resources its priority is increased to the highest local priority to prevent any preemption during the access of shared resource from other tasks that belong to the same subsystem. SRP is used in the global level to synchronize the execution of subsystems that have tasks accessing global shared resources. Each global shared resource has a ceiling equal to the maximum priority of subsystems that has a task accessing that resource. Two versions of the overrun mechanisms have been presented; 1) The overrun mechanism with payback which works as follows, whenever overrun happens in a subsystem S_s, the budget of the subsystem will be decreased by the amount of the overrun time in its next execution instant. 2) In the second version which is called overrun mechanism without payback, no further actions will be taken after the event of an overrun. Selecting which of these two mechanisms that can give better results in terms of task response times depends on the system param-

eters. The presented schedulability analysis does not support composability, disallowing independent analysis of individual subsystems since information about other subsystems is needed in order to apply the schedulability analysis for all tasks. In addition, HSRP does not provide a complete separation between the local and the global schedulers. The local scheduler should inform the global scheduler to let the server continue executing when a budget expiry inside a critical section problem happens and then the local scheduler should inform the global scheduler when its task releases the global shared resource.

4.2.3 BROE

The Bounded-delay Resource Open Environment (BROE) server [16] extends the Constant Bandwidth Server (CBS) [30] in order to handle the sharing of logical resources in a HSF. The BROE server is suitable for open systems since it allows for each application to be developed and validated independently. For each application, the maximum CPU-resources demand is characterized by server speed, delay tolerance (using the bounded-delay resource partition [7]) and resource holding time. These parameters will be used as an interface between the application and the system scheduler so that the system scheduler will schedule all servers according to their interface parameters. The interface parameters will also be used during the admission control of new applications to check if there is enough CPU-resources to run this new application on the processor. The BROE server uses the SRP protocol to arbitrate access to global shared resources and in order to prevent the budget expiration inside critical section problem, the application performs a budget check before accessing a global shared resource. If the application has sufficient remaining budget then it allows its task to lock the global resource otherwise it postpones its current deadline and replenishes its budget (according to certain rules that guarantee the correctness of the CBS servers execution) to be able to lock and release the global resource safely. Comparing the BROE server with HSRP, BROE does not need more resources to handle the problem of budget expiry in the global level while HSRP may require more resources since it uses an overrun mechanism and the overrun time should be taken into account in the global scheduling. However, the only scheduling algorithm that is suitable for the presented version of the BROE server is EDF which is one of the limitations of this approach. In addition, in [16], the authors didn't explain how to evaluate the value of the resource holding time for BROE server (the authors left this issue to a future submission) and how this value may affect the CPU-resources usage locally and globally.

4.2.4 SIRAP

The Subsystem Integration and Resource Allocation Policy (SIRAP) [15] protocol supports subsystem integration in the presence of shared logical resources. SIRAP can be used in an open systems. It uses a periodic resource model to abstract the timing requirements of each subsystem. Each subsystem is characterized by its period and budget and resource holding time and it is implemented as a simple periodic server. SIRAP uses the SRP protocol to synchronize the access to global shared resources in both local and global scheduling. SIRAP applies a skipping approach to prevent the budget expiration inside critical section which works as follows; when a job wants to enter a critical section, it enters the critical section at the earliest instant such that it can complete the critical section before the subsystem budget expires. This can be achieved by checking the remaining budget before granting the access to the global shared resources, if there is sufficient remaining budget then the job enters the critical section. If there is insufficient remaining budget, the local scheduler delays the critical section entering of the job until the next subsystem budget replenishment. Comparing SIRAP and BROE, both provide better isolation between the global and the local schedulers than HSRP since they solve the problem of budget expiry inside a critical section locally. However, using HSRP, it is not required to include the resource holding time in the interface of subsystems during run-time and its required only for schedulability analysis while the resource golding times are required during run-time for SIRAP and BROE. Both SIRAP and BROE do not need extra resources in the global scheduling level. The SIRAP protocol needs extra resources in the local level scheduling when it increases the resource demand of the subsystem and for BROE it is not clear since the way of evaluating resource holding time was not presented. Another difference between BROE and SIRAP is that the SIRAP protocol uses FPS as a global scheduling algorithm and can be easily adapted to include local and global EDF while BROE can only work with EDF as a global scheduler.

4.3 Subsystem interface and resource sharing

Supporting shared resources across subsystems produces interference among subsystems which imposes more CPU demands for each subsystem. In the local schedulability analysis and because of using SRP locally, the blocking times should be added to the maximum resources demand side in Eq. (3.4) and Eq. (3.6) and this will increase the minimum required subsystem budget Q_s. In the global level and because of using SRP between subsystems, the block-

ing time (resource holding time[1]) that a subsystem may block other subsystems should be added to the global schedulability analysis. So for the global schedulability analysis the subsystem interface should include in addition to the subsystem period and budget, the maximum resource holding time for each global shared resource that the internal tasks of the subsystem may access. One way to decrease the amount of information of subsystem interface needed for global schedulability analysis, can be by considering that the subsystem will access all global resources, then it is required to provide the maximum resource holding time of all internal tasks that access the global shared resources. The subsystem timing interface of a subsystem S_s for this case is (P_s, Q_s, H_s) where H_s is the maximum resource holding time of all internal tasks of S_s that access global shared resources. Finally the extra CPU demand that is required to solve the problem of budget expiry inside the critical section depends on the used mechanism.

As mentioned previously, 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 holding time, for example, using the mechanism presented in [26, 27] by increasing the resource ceiling of the global shared resources locally inside the subsystems, in order to potentially reduce the blocking of other subsystems towards decrease of the system load. However, we have found that decreasing the value of resource holding times may increase the required budget of the same subsystem Q_s and it may increase the system load.

[1] In paper D we use the term resource locking time instead of resource holding time to remove any confusion since the term resource holding time was firstly presented in the context of non-hierarchical scheduling.

Chapter 5

Conclusions

5.1 Summary

We have implemented a HSF in a commercial operating system (VxWorks) without changing the kernel of the operating system. Each subsystem has been implemented using periodic servers. As most commercial real-time operating system, VxWorks does not support the periodic activation of tasks. In order to enable periodic activations of tasks and servers, we have used a timer and an interrupt service rutin. We have measured the overhead of the implementation and the results shows that a hierarchical scheduling framework can effectively achieve the clean separation of subsystems in terms of timing interference (i.e., without requiring any temporal parameters of other subsystems) with reasonable implementation overheads.

We have also investigated the problem of supporting sharing of logical resources and we have presented a novel Subsystem Integration and Resource Allocation Policy (SIRAP), which is a synchronization protocol providing temporal isolation between subsystems that share logical resources. Furthermore, we have formally proven key features of SIRAP such as bounds on delays for accessing shared resources. Also we have provided schedulability analysis for tasks executing in the subsystems; allowing for use of hard real-time applications within the SIRAP framework. Naturally, the flexibility and predictability offered by SIRAP comes with some costs in terms of overhead. We have evaluated this overhead through a comprehensive simulation study.

In addition, we have proposed new overrun mechanisms based on the approach presented in [14], for hierarchical scheduling frameworks, that can be

used in the domain of open systems. We have presented both independent local schedulability analysis as well as global schedulability analysis for the proposed overrun mechanism as well as the existing basic overrun. We have presented analysis of when one overrun mechanism is better than the other and the results indicate that in the general case it is not trivial to evaluate which overrun mechanism that is better than the other.

We have focused on assigning the CPU-resources to subsystems in an efficient way such that the resulting system load will be as low as possible. We introduced a tradeoff between decreasing the resource locking time and the system load, and we 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.

5.2 Future work

The work presented in this thesis has left and opened some issues that would be interesting to be investigated in the future. Some of the issues that will be presented are general and some others are specific for each paper.

Starting from general issues, in this work we assume that a system is executed in a single processor while many real-time applications are distributed into several processors that communicate through some communication network. Also, complementing single processor systems, other systems are executed in a multi-processor or multi-core architecture. It will be interesting to extend the HSF include the distributed systems and multi-processor systems.

We would also like to include the subsystem context-switch in the schedulability analysis and check whether using non-preemptive global scheduling can be more efficient than preemptive scheduler in terms of CPU-resources usage. Note that a subsystem context-switch has more overhead than a task context-switch because if a subsystem gets preempted be another subsystem then the scheduler should remove the first subsystem and all its associated tasks and add the higher priority subsystem with all ready tasks that belong to the second subsystem, which takes longer time and could be expensive.

Another interesting work will be on supporting shared resources in multi-level hierarchical scheduling frameworks since we only consider a two-level hierarchical scheduling framework. Also we would like to consider other resource models such as the EDP resource model [13]. Finally it is important to test our framework with real applications by doing case studies.

Paper A In the next stage of the implementation of the HSF, we intend to implement synchronization protocols in hierarchical scheduling frameworks, e.g., using SIRAP [15] and HSRP [14]. In addition, our future work includes supporting sporadic tasks in response to specific events such as external interrupts. We also plan to support soft aperiodic tasks in an efficient way to increase the quality of service of the soft tasks. Moreover, we intend to extend the implementation to make it suitable for more advanced architectures including multi-core processors.

Paper B Future work includes investigating the effect of the context-switch overhead on subsystem utilization together with the subsystem period and the maximum value of h_i.

Paper C Future work includes finding the exact schedulability analysis for the enhanced overrun mechanism, since the presented analysis merely gives upper bound. We would like to include the development of local and global schedulability analysis for Fixed Priority Scheduling (FPS), as the current results only consider Earliest Deadline First (EDF). Another interesting issue is to compare the implementation of the enhanced overrun mechanism with other synchronization mechanisms such as BWI [29], BROE server [16] and SIRAP [15].

Paper C In this paper, we considered only Fixed Priority Scheduling (FPS), and we plan to extend our work to EDF scheduling. Furthermore, our future work includes generalizing our framework to other synchronization protocols such as BROE server [16] and SIRAP [15].

Chapter 6

Overview of Papers

6.1 Paper A

Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, Reinder J. Bril, *Towards Hierarchical Scheduling on top of VxWorks*, In Proceedings of the 4^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08), pages 63-72, Prague, Czech Republic, July, 2008.

Summary Over the years, we have worked on hierarchical scheduling frameworks from a theoretical point of view. In this paper we present our initial results of the implementation of our hierarchical scheduling framework in a commercial operating system VxWorks. The purpose of the implementation is twofold: (1) we would like to demonstrate feasibility of its implementation in a commercial operating system, without having to modify the kernel source code, and (2) we would like to present detailed figures of various key properties with respect to the overhead of the implementation. During the implementation of the hierarchical scheduler, we have also developed a number of simple task schedulers. We present details of the implementation of Rate-Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Finally, we present the design of our hierarchical scheduling framework, and we discuss our current status in the project.

My contribution The results of this paper was based on the results of a master project under the supervision of Moris Behnam.

6.2 Paper B

Moris Behnam, Insik Shin, Thomas Nolte, Mikael Nolin, *SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems*, In Proceedings of the 7^{th} ACM & IEEE International Conference on Embedded Software (EMSOFT'07), pages 279-288, Salzburg, Austria, October, 2007.

Summary This paper presents a protocol for resource sharing in a hierarchical real-time scheduling framework. Targeting real-time open systems, the protocol and the scheduling framework significantly reduce the efforts and errors associated with integrating multiple semi-independent subsystems on a single processor. Thus, our proposed techniques facilitate modern software development processes, where subsystems are developed by independent teams (or subcontractors) and at a later stage integrated into a single product. Using our solution, a subsystem need not know, and is not dependent on, the timing behaviour of other subsystems; even though they share mutually exclusive resources. In this paper we also prove the correctness of our approach and evaluate its efficiency.

My contribution The basic idea of this paper was suggested by Moris Behnam. The work was done in cooperation with Moris and Insik Shin, and Moris was responsible for the evaluation part of the paper and he was also involved in the schedulability analysis.

6.3 Paper C

Moris Behnam, Insik Shin, Thomas Nolte, Mikael Nolin, *Scheduling of Semi-Independent Real-Time Components: Overrun Methods and Resource Holding Times*, In Proceedings of the 13^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08), IEEE Industrial Electronics Society, Hamburg, Germany, September, 2008.

Summary The Hierarchical Scheduling Framework (HSF) has been introduced as a design-time framework enabling compositional schedulability analysis of embedded software systems with real-time properties. In this paper a system consists of a number of semi-independent components called subsystems. Subsystems are developed independently and later integrated to form a system. To support this design process, our proposed methods allow non-intrusive configuration and tuning of subsystem timing behaviour via subsystem interfaces for selecting scheduling parameters. This paper considers two methods to handle overruns due to resource sharing between subsystems in the HSF. We present the scheduling algorithms for overruns and their associated schedulability analysis, together with analysis that shows under what circumstances one or the other overrun method is preferred. Furthermore, we show how to calculate resource-holding times within our framework.

My contribution The paper is based on an idea of Insik Shin but Moris has done most of the work including the schedulability analysis for enhanced overrun mechanism and the comparison between the enhanced and the basic overrun mechanism, as well as the simplified equation to evaluate the resource holding times with the required proofs.

6.4 Paper D

Insik Shin, Moris Behnam, Thomas Nolte, Mikael Nolin, *Synthesis of Optimal Interfaces for Hierarchical Scheduling with Resources*, In Proceedings of the 29^{th} IEEE International Real-Time Systems Symposium (RTSS08), IEEE Press, Barcelona, Spain, December, 2008, (to be appear).

Summary 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 schedulability. 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.

My contribution The paper was based on ideas of Moris and Insik. Moris was responsible for developing the algorithms and prove their correctness and optimality formally. Moris was also involved in the discussions and witting of the other parts of the paper.

Bibliography

[1] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* 24^{th} *IEEE International Real-Time Systems Symposium(RTSS'03)*, pages 2–13, Cancun, Mexico, December 2003.

[2] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. In *Component-Based Software Engineering*, volume LNCS-3054/2004, pages 253–266. Springer Berlin / Heidelberg, May 2005.

[3] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the* 18^{th} *IEEE International Real-Time Systems Symposium (RTSS'97)*, pages 308–319, San Francisco, CA, USA, December 1997. IEEE Computer Society.

[4] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the* 20^{th} *IEEE International Real-Time Systems Symposium (RTSS'99)*, pages 256–267, Phoenix, AZ, USA, December 1999. IEEE Computer Society.

[5] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, Washington DC, USA, May-June 2000. IEEE Computer Society.

[6] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard-real-time environments. In *Proceedings of the* 21^{th} *IEEE International Real-Time Systems Symposium(RTSS'00)*, pages 217–226, Orlando, FL, USA, December 2000.

[7] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium(RTAS)*, pages 75–84, Taipei, Taiwan ROC, May 2001.

[8] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* 23^{th} *IEEE International Real-Time Systems Symposium (RTSS'02)*, pages 26–35, Austin, TX, USA, December 2002.

[9] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT '04)*, pages 95–103, Pisa, Italy, September 2004.

[10] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 151–158, Porto, Portugal, July 2003. IEEE Computer Society.

[11] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the* 26^{th} *IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 389–398, Miami Beach, FL, USA, December 2005.

[12] F. Zhang and A. Burns. Analysis of hierarchical EDF pre-emptive scheduling. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 423–434, Washington, DC, USA, December 2007. IEEE Computer Society.

[13] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium(RTSS'07)*, pages 129–138, Washington, DC, USA, 2007. IEEE Computer Society.

[14] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the* 27^{th} *IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, Rio de Janeiro, Brazil, December 2006.

[15] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, pages 279–288, Salzburg, Austria, October 2007.

[16] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, Washington, DC, USA, December 2007. IEEE Computer Society.

[17] J. A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for Real-Time Systems. Technical Report UM-CS-1993-023, University of Massachusetts, Amherst, June 1993.

[18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):40–61, January 1973.

[19] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, December 1982.

[20] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the* 20^{th} *IEEE International Real-Time Systems Symposium(RTSS'89)*, pages 166–171, Santa Monica, CA, USA, December 1989. IEEE Computer Society.

[21] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Journal of Real-Time Systems*, 2:301–324, 1990.

[22] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications,* 2^{nd} *ed.* Springer, 2005.

[23] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation IECON87*, pages 909–916, Cambridge, MA, USA, November 1987.

[24] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the* 9^{th} *IEEE International Real-Time Systems Symposium (RTSS'88)*, pages 259–269, Huntsville, AL, USA, December 1988. IEEE Computer Society.

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

[26] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *Proceedings of the 15th International Workshop on Parallel and Distributed Real-Time Systems(WPDRTS)*, pages 1–8, Long Beach, CA, USA, March 2007.

[27] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in EDF-scheduled systems. In *Proceedings of the* 13^th *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, Bellevue, WA, USA, 2007.

[28] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the* 11^th *IEEE International Real-Time Systems Symposium(RTSS'90)*, pages 182–190, Lake Buena Vista, Florida, USA, December 1990. IEEE Computer Society.

[29] G. Lipari, G. Lamastra, and L. Abeni. Task synchronization' in reservation-based real-time systems. *IEEE Transactions on Computers*, 53(12):1591–1601, December 2004.

[30] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the* 19^{th} *IEEE International Real-Time Systems Symposium (RTSS'98)*, pages 4–13, Madrid, Spain, December 1998. IEEE Computer Society.

II

Included Papers

Chapter 7

Paper A: Towards Hierarchical Scheduling in VxWorks

Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg and Reinder J. Bril

In Proceedings of the 4^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08), pages 63-72, Prague, Czech Republic, July, 2008.

Abstract

Over the years, we have worked on hierarchical scheduling frameworks from a theoretical point of view. In this paper we present our initial results of the implementation of our hierarchical scheduling framework in a commercial operating system VxWorks. The purpose of the implementation is twofold: (1) we would like to demonstrate feasibility of its implementation in a commercial operating system, without having to modify the kernel source code, and (2) we would like to present detailed figures of various key properties with respect to the overhead of the implementation. During the implementation of the hierarchical scheduler, we have also developed a number of simple task schedulers. We present details of the implementation of Rate-Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Finally, we present the design of our hierarchical scheduling framework, and we discuss our current status in the project.

7.1 Introduction

Correctness of today's embedded software systems generally relies not only on functional correctness, but also on extra-functional correctness, such as satisfying timing constraints. System development (including software development) can be substantially facilitated if (1) the system can be decomposed into a number of parts such that parts are developed and validated in isolation and (2) the temporal correctness of the system can be established by composing the correctness of its individual parts. For large-scale embedded real-time systems, in particular, advanced methodologies and techniques are required for temporal and spatial isolation all through design, development, and analysis, simplifying the development and evolution of complex industrial embedded software systems.

Hierarchical scheduling has shown to be a useful mechanism in supporting modularity of real-time software by providing temporal partitioning among applications. In hierarchical scheduling, a system can be hierarchically divided into a number of subsystems that are scheduled by a global (system-level) scheduler. Each subsystem contains a set of tasks that are scheduled by a local (subsystem-level) scheduler. The Hierarchical Scheduling Framework (HSF) allows for a subsystem to be developed and analyzed in isolation, with its own local scheduler, and then at a later stage, using an arbitrary global scheduler, it allows for the integration of multiple subsystems without violating the temporal properties of the individual subsystems analyzed in isolation. The integration involves a system-level schedulability test, verifying that all timing requirements are met. Hence, hierarchical scheduling frameworks naturally support *concurrent development* of subsystems. Our overall goal is to make hierarchical scheduling a cost-efficient approach applicable for a wide domain of applications, including automotive, automation, aerospace and consumer electronics.

Over the years, there has been a growing attention to HSFs for real-time systems. Since a two-level HSF [1] has been introduced for open environments, many studies have been proposed for its schedulability analysis of HSFs [2, 3]. Various processor models, such as bounded-delay [4] and periodic [5], have been proposed for multi-level HSFs, and schedulability analysis techniques have been developed for the proposed processor models [6, 7, 8, 9, 10, 5, 11]. Recent studies have been introduced for supporting logical resource sharing in HSFs [12, 13, 14].

Up until now, those studies have worked on various aspects of HSFs from a theoretical point of view. This paper presents our work towards a full im-

plementation of a hierarchical scheduling framework . We have chosen to implement it in a commercial operating system already used by several of our industrial partners. We selected the VxWorks operating system, since there is plenty of industrial embedded software available, which can run in the hierarchical scheduling framework.

The outline of this paper is as follows: Section 7.2 presents related work on implementations of schedulers. Section 7.3 present our system model. Section 7.4 gives an overview of VxWorks, including how it supports the implementation of arbitrary schedulers. Section 7.5 presents our scheduler for VxWorks, including the implementation of Rate Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Section 7.6 presents the design, implementation and evaluation of the hierarchical scheduler, and finally Section 7.7 summarizes the paper.

7.2 Related work

Looking at related work, recently a few works have implemented different schedulers in commercial real-time operating systems, where it is not feasible to implement the scheduler directly inside the kernel (as the kernel source code is not available). Also, some work related to efficient implementations of schedulers are outlined.

Buttazzo and Gai [15] present an implementation of the EDF scheduler for the ERIKA Enterprise kernel [16]. The paper discusses the effect of time representation on the efficiency of the scheduler and the required storage. They use the Implicit Circular Timer's Overflow Handler (ICTOH) algorithm which allows for an efficient representation of absolute deadlines in a circular time model.

Diederichs and Margull [17] present an EDF scheduler plug-in for OSEK/VDX based real-time operating systems, widely used by automotive industry. The EDF scheduling algorithm is implemented by assigning priorities to tasks according to their relative deadlines. Then, during the execution, a task is released only if its absolute deadline is less than the one of the currently running task. Otherwise, the task will be delayed until the time when the running task finishes its execution.

Kim *et al.* [18] propose the SPIRIT uKernel that is based on a two-level hierarchical scheduling framework simplifying integration of real-time applications. The SPIRIT uKernel provides a separation between real-time applications by using partitions. Each partition executes an application, and uses

the Fixed Priority Scheduling (FPS) policy as a local scheduler to schedule the application's tasks. An offline scheduler (timetable) is used to schedule the partitions (the applications) on a global level. Each partition provides kernel services for its application and the execution is in user mode to provide stronger protection.

Parkinson [19] uses the same principle and describes the VxWorks 653 operating system which was designed to support ARINC653. The architecture of VxWorks 653 is based on partitions, where a Module OS provides global resource and scheduling for partitions and a Partition OS implemented using VxWorks microkernel provides scheduling for application tasks.

The work presented in this paper differs from the last two works in the sense that it implements a hierarchical scheduling framework in a commercial operating system without changing the OS kernel. Furthermore, the work differs from the above approaches in the sense that it implements a hierarchical scheduling framework intended for open environments [1], where real-time applications may be developed independently and unaware of each other and still there should be no problems in the integration of these applications into one environment. A key here is the use of well defined *interfaces* representing the collective resource requirements by an application, rich enough to allow for integration with an arbitrary set of other applications without having to redo any kind of application internal analysis.

7.3 System model

In this paper, we only consider a simple periodic task model $\tau_i(T_i, C_i, D_i)$ where T_i is the task period, C_i is a worst-case execution time requirement, and D_i is a relative deadline ($0 < C_i \leq D_i \leq T_i$). The set of all tasks is denoted by Γ ($\Gamma = \{\tau_i | \texttt{for all}\ i = 1, .., n\}$ where n is the number of tasks).

We assume that all tasks are independent of each other, i.e., there is no sharing of logical resources between tasks and tasks do no suspend themselves.

The HSF schedules subsystems $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the set representing the whole system of subsystems. Each subsystem S_s consists of a set of tasks and a local scheduler (RM or EDF), and the global (system) scheduler (RM or EDF). The collective real-time requirements of S_s is referred to as a *timing-interface*. The subsystem interface is defined as (P_s, Q_s), where P_s is a subsystem period, and Q_s is a budget that represents an execution time requirement that will be provided to the subsystem S_s every period P_s.

7.4 VxWorks

VxWorks is a commercial real-time operating system developed by Wind River with a focus on performance, scalability and footprint. Many interesting features are provided with VxWorks, which make it widely used in industry, such as; Wind micro-kernel, efficient task management and multitasking, deterministic context switching, efficient interrupt and exception handling, POSIX pipes, counting semaphores, message queues, signals, and scheduling, preemptive and round-robin scheduling etc. (see [20] for more details).

The VxWorks micro-kernel supports the priority preemptive scheduling policy with up to 256 different priority levels and a large number of tasks, and it also supports the round robin scheduling policy.

VxWorks offers two different modes for application-tasks to execute; either kernel mode or user mode. In kernel mode, application-tasks can access the hardware resources directly. In user mode, on the other hand, tasks can not directly access hardware resources, which provides greater protection (e.g., in user mode, tasks can not crash the kernel). Kernel mode is provided in all versions of VxWorks while user mode was provided as a part of the Real Time Process (RTP) model, and it has been introduced with VxWorks version 6.0 and beyond.

In this paper, we are considering kernel mode tasks since such a design would be compatible with all versions of VxWorks and our application domains include systems with a large legacy in terms of existing source codes. We are also considering fixed priority preemptive scheduling policy for the kernel scheduler (not the round robin scheduler). A task's priority should be set when the task is created, and the task's priority can be changed during the execution. Then, during run-time, the highest priority ready task will always execute. If a task with priority higher than that of the running task becomes ready to execute, then the scheduler stops the execution of the running task and instead executes the one with higher priority. When the running task finishes its execution, the task with the highest priority among the ready tasks will execute.

When a task is created, an associated Task Control Block (TCB) is created to save the task's context (e.g., CPU environment and system resources, during the context switch). Then, during the life-cycle of a task the task can be in one or a combination of the following states [21] (see Figure 7.1):

- **Ready state**, the task is waiting for CPU resources.
- **Suspended state**, the task is unavailable for execution but not delayed

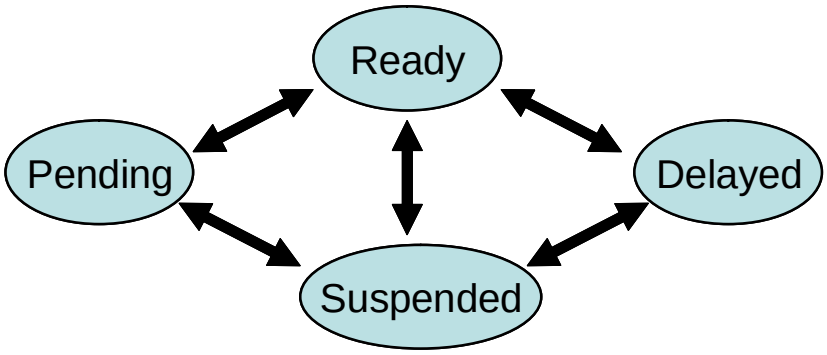


Figure 7.1: The application task state.

or pending.

- **Pending state**, the task is blocked waiting for some resource other than the CPU.
- **Delayed state**, the task is sleeping for some time.

Note that the kernel scheduler sorts all tasks that are ready to execute in a queue called the *ready queue*.

7.4.1 Scheduling of time-triggered periodic tasks

A periodic task is a task that becomes ready for execution periodically once every n-th time unit, i.e., a new instant of the task is executed every constant period of time. Most commercial operating systems, including VxWorks, do not directly support the periodic task model [22]. To implement a periodic task, when a task finishes its execution, it sleeps until the beginning of its next period. Such periodic behaviour can be implemented in the task by the usage of timers. Note that a task typically does not finish its execution at the same time always, as execution times and response times vary from one period to another. Hence, using timers may not be easy and accurate as the task needs to evaluate the time for next period relative to the current time, whenever it finishes its execution. This is because preemption may happen between the time measurement and calling the sleep function.

In this project we need to support periodic activation of *servers* in order to implement the hierarchical scheduling framework. The reason for this is that we base our hierarchical scheduling framework around the periodic resource model [5], and a suitable implementation of the periodic resource model is achieved by the usage of a server based approach similar to the periodic servers [23, 24] that replenish their budget every constant period, i.e., the servers behave like periodic tasks.

7.4.2 Supporting arbitrary schedulers

There are two ways to support arbitrary schedulers in VxWorks:

1. Using the VxWorks custom kernel scheduler [25].

2. Using the original kernel scheduler and manipulating the ready queue by changing the priority of tasks and/or activating and suspending tasks.

In this paper, we are using the second approach since implementing the custom kernel scheduler is a relatively complex task compared with manipulating the ready queue. However, it will be interesting to compare between the two methods in terms of CPU overhead, and we leave this as a future work.

In the implementation of the second solution, we have used an Interrupt Service Routine (ISR) to manipulate the tasks in the ready queue. The ISR is responsible for adding tasks in the ready queue as well as changing their priorities according to the hierarchical scheduling policy in use. In the remainder of this paper, we refer to the ISR as the User Scheduling Routine (USR). By using the USR, we can implement any desired scheduling policy, including common ones such as Rate Monotonic (RM) and Earliest Deadline First (EDF).

7.5 The USR custom VxWorks scheduler

This section presents how to schedule periodic tasks using our scheduler, the User Scheduling Routine (USR).

7.5.1 Scheduling periodic tasks

When a periodic task finishes its execution, it changes its state to suspended by explicitly calling the suspend function. Then, to implement a periodic task, a

timer could be used to trigger the USR once every new task activation time to release the task (to put it in the ready queue).

The solution to use a timer triggering the USR once every new period can be suitable for systems with a low number of periodic tasks. However, if we have a system with n periodic tasks such a solution would require the use of n timers, which could be very costly or not even possible. In this paper we have used a scalable way to solve the problem of having to use too many timers. By multiplexing a single timer, we have used a single timer to serve n periodic tasks.

The USR stores the next activation time of all tasks (absolute times) in a sorted (according to the closest time event) queue called Time Event Queue (TEQ). Then, it sets a timer to invoke the USR at the time equal to the shortest time among the activation times stored in the TEQ. Also, the USR checks if a task misses its deadline by inserting the deadline in the TEQ. When the USR is invoked, it checks all task states to see if any task has missed its deadline. Hence, an element in the TEQ contains (1) the absolute time, (2) the id of task that the time belongs to, and (3) the event type (task next activation time or absolute deadline). Note that the size of the TEQ will be $2 * n * B$ bytes (where B is the size in bytes of one element in the TEQ) since we need to save the task's next period time and deadline time.

When the USR is triggered, it checks the cause of the triggering. There are two causes for the USR to be triggered: (1) a task is released, and (2) the USR will check for deadline misses. For both cases, the USR will do the following:

- Update the next activation and/or the absolute deadline time associated with the task that caused triggering of the USR in the TEQ and re-insert it in the TEQ according to the updated times.
- Set the timer equal to the shortest time in the TEQ so that the USR will be triggered at that time.
- For task release, the USR changes the state of the task to Ready. Also, it changes priorities of tasks if required depending on the scheduler (EDF or RM). For deadline miss checking, the USR checks the state of the task to see if it is Ready. If so, the task missed its deadline, and the deadline miss function will be activated.

Updating the next activation time and absolute deadline of a task in the TEQ is done by adding the period of the task that caused the USR invocation to the current absolute time. The USR does not use the system time as a time

reference. Instead it uses a time variable as a time reference. The reason for using a time variable is that we can, in a flexible manner, select the size of variables that save absolute time in bits. The benefits of such an approach is that we can control the size of the TEQ since it saves the absolute times, and it also minimizes the overhead of implementing 64 bits operations on 32 bit microprocessor [15], as an example. The reference time variable t_s used to indicate the time of the next activation, is initialized (i.e., $t_s = 0$) at the first execution of the USR. The value of t_s is updated every time that the USR executes and it will be equal to the time given by the TEQ that triggered the USR.

When a task τ_i is released for the first time, the absolute next activation time is equal to $t_s + T_i$ and its absolute deadline is equal to $t_s + D_i$.

To avoid time consuming operations, e.g., multiplications and divisions, that increase the system overhead inherent in the execution of the USR, all absolute times (task periods and relative deadlines) are saved in system tick unit (system tick is the interval between two consecutive system timer interrupts). However, depending on the number of bits used to store the absolute times, there is a maximum value that can be saved safely. Hence, saving absolute times in the TEQ may cause problems related to overrun of time, i.e., the absolute times become too large such that the value can not be stored using the available number of bits. To avoid this problem, we apply a wrapping algorithm which wraps the absolute times at some point in time, so the time will restart again. Periods and deadlines should not exceed the wrap-around value.

The input of the timer should be in a relative time, so evaluating the time at which to trigger the USR again (next time) is done by $TEQ[1] - t_s$ where $TEQ[1]$ is the first element in the queue after updating the TEQ as well as sorting it, i.e., the closest time in the TEQ. The USR checks to see if there are more than one task that have the same current activation time and absolute deadline. If so, the USR serves all these tasks to minimize the unnecessary overhead of executing the USR several times.

7.5.2 RM scheduling policy

Each task will have a fixed priority during run-time when Rate Monotonic (RM) is used, and the priorities are assigned according to the RM scheduling policy. If only RM is used in the system, no additional operations are required to be added to the USR since the kernel scheduler schedules all tasks directly according to their priorities, and the higher priority tasks can preempt the execution of the lower priority task. Hence, the implementation overhead for RM

will be limited to the overhead of adding a task in the ready queue and managing the timer for the next period (saving the absolute time of the new period and finding the shortest next time in the TEQ) for periodic tasks.

The schedulability analysis for each task is as follows [26];

$$\forall \tau_i \in \Gamma, 0 < \exists t \leq T_i \;\; \texttt{dbf}(i,t) \leq t. \tag{7.1}$$

And $\texttt{dbf}(i,t)$ is evaluated as follows

$$\texttt{dbf}(i,t) = C_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k, \tag{7.2}$$

where $\texttt{HP}(i)$ is the set of tasks with priority higher than that of τ_i.

Eq. (7.2) can be easily modified to include the effect of using the USR on the schedulability analysis. Note that the USR will be triggered at the beginning of each task to release the task, so it behaves like a periodic task with priority equal to the maximum possible priority (the USR can preempt all application tasks). Checking the deadlines for tasks by using the USR will add more overhead, however, also this overhead has a periodic nature as the task release presented previously.

Eq. (7.3) includes the deadline and task release overhead caused by the USR in the response time analysis,

$$\begin{aligned} \texttt{dbf}(i,t) \;=\; C_i \;+ & \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k \;+ \sum_{\tau_j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil X_R \\ & + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t + T_j - D_j}{T_j} \right\rceil X_D \end{aligned} \tag{7.3}$$

where X_R is the worst-case execution time of the USR when a task is released and X_D is the worst-case execution time of the USR when it checks for deadline misses (currently, in case of deadline misses, the USR will only log this event into a log file).

7.5.3 EDF scheduling policy

For EDF, the priority of a task changes dynamically during run-time. At any time t, the task with shorter deadline will execute first, i.e., will have the highest priority. To implement EDF in the USR, the USR should update the priorities of all tasks that are in the Ready Queue when a task is added to the Ready

Queue, which can be costly in terms of overhead. Hence, on one hand, using EDF on top of commercial operating systems may not be efficient depending on the number of tasks, due to this sorting. However, the EDF scheduling policy provides, on other hand, better CPU utilization compared with RM, and it also has a lower number of context switches which minimizes context switch related overhead [27].

In the approach presented in this paper, tasks are already sorted in the TEQ according to their absolute times due to the timer multiplexing explained earlier. Hence, as the TEQ is already sorted according to the absolute deadlines, the USR can easily decide the priorities of the tasks according to EDF without causing too much extra overhead for evaluating the proper priority for each task.

The schedulability test for a set of tasks that use EDF is shown in Eq. (7.4) [28] which includes the case when task deadlines are allowed to be less than or equal to task periods.

$$\forall t > 0, \quad \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i \leq t \tag{7.4}$$

The overhead of implementing EDF can also be added to Eq. (7.4). Hence, Eq. (7.5) includes the overhead of releasing tasks as well as the overhead of checking for deadline misses.

$$\begin{aligned} \forall t > 0, \quad & \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil X_R \\ & + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t + T_j - D_j}{T_j} \right\rceil X_D \leq t \end{aligned} \tag{7.5}$$

7.5.4 Implementation and overheads of the USR

To implement the USR, we have used the following VxWorks service functions;

- Q_PUT - insert a node into a multi-way queue (ready queue).
- Q_REMOVE - remove a node from a multi-way queue (ready queue).
- taskCreat - create a task.
- taskPrioritySet - set a tasks priority.

We present our initial results inherent in the implementation of the USR, implementing both the Rate Monotonic (RM) scheduler as well as the Earliest Deadline First (EDF) scheduler. The implementations were performed on a ABB robot controller with a Pentium 200 MHz processor running the VxWorks operating system version 5.2. To trigger the USR for periodic tasks, we have used watchdog timers where the next expiration time is given in number of ticks. The watchdog uses the system clock interrupt routine to count the time to the next expiration. The platform provides system clock with resolution equal to $4500 ticks/s$. The measurement of the execution time of the USR is done by reading a timestamp value at the start as well as at the end of the USR's execution. Note that the timestamp is connected to a special hardware timer with resolution $12000000 ticks/s$.

Table 7.1 shows the execution time of the USR when it performs RM and EDF scheduling, as well as deadline miss checking, as a function of the number of tasks in the system. The worst case execution time for USR will happen when USR deletes and then inserts all tasks from and to TEQ and to capture this, we have selected a same period for all tasks. The table shows the minimum, maximum and average out of 50 measured values. Comparing between the results of the three cases (EDF, RM, deadline miss), we can see that there is no big difference in the execution time of the USR. The reason for this result is that the execution of the USR for EDF, RM and deadline miss checking all includes the overhead of deletion and re-inserting the tasks in the TEQ, which is the dominating part of the overhead. As expected, EDF causes the largest overhead because it changes the priority of all tasks in the ready queue during run-time. Figures 7.2-7.3 show that EDF imposes between $6 - 14\%$ extra overhead compared with RM.

7.6 Hierarchical scheduling

A Hierarchical Scheduling Framework (HSF) supports CPU sharing among subsystems under different scheduling policies. Here, we consider a two-level scheduling framework consisting of a global scheduler and a number of local schedulers. Under global scheduling, the operating system (global) scheduler allocates the CPU to subsystems. Under local scheduling, a local scheduler inside each subsystem allocates a share of the CPU (given to the subsystem by the global scheduler) to its own internal tasks (threads).

We consider that each subsystem is capable of exporting its own interface that specifies its collective real-time CPU requirements. We assume that such a

Number of tasks	X_R (RM)			X_R (EDF)			X_D (Deadline miss check)		
	Max	Average	Min	Max	Average	Min	Max	Average	Min
10	71	65	63	74	70	68	70	60	57
20	119	110	106	131	118	115	111	100	95
30	172	158	155	187	172	169	151	141	137
40	214	202	197	241	228	220	192	180	175
50	266	256	249	296	280	275	236	225	219
60	318	305	299	359	338	331	282	268	262
70	367	352	341	415	396	390	324	309	304
80	422	404	397	476	453	444	371	354	349
90	473	459	453	539	523	515	415	398	393
100	527	516	511	600	589	583	459	442	436

Table 7.1: USR execution time in μs, the maximum, average and minimum execution time of 45 measured values for each case.

subsystem interface is in the form of the periodic resource model (P_s, Q_s) [5]. Here, P_s represents a *period*, and Q_s represents a *budget*, or an execution time requirement within the period $(Q_s < P_s)$. By using the periodic resource model in hierarchical scheduling frameworks, it is guaranteed [5] that all timing constraints of internal tasks within a subsystem can be satisfied, if the global scheduler provides the subsystem with CPU resources according to the timing requirements imposed by its subsystem interface. We refer interested readers to [5] for how to derive an interface (P_s, Q_s) of a subsystem, when the subsystem contains a set of internal independent periodic tasks and the local scheduler follows the RM or EDF scheduling policy. Note that for the derivation of the subsystem interface (P_s, Q_s), we use the demand bound functions that take into account the overhead imposed by the execution of USR (see Eq. (7.3) and (7.5)).

7.6.1 Hierarchical scheduling implementation

Global scheduler: A subsystem is implemented as a periodic server, and periodic servers can be scheduled in a similar way as scheduling normal periodic tasks. We can use the same procedure described in Section 7.5 with some modifications in order to schedule servers. Each server should include the following information to be scheduled: (1) server period, (2) server budget, (3) remaining budget, (4) pointer to the tasks that belong to this server, and (5) the type of the local scheduler (RM or EDF) (6) local TEQ. Moreover, to schedule servers we need:

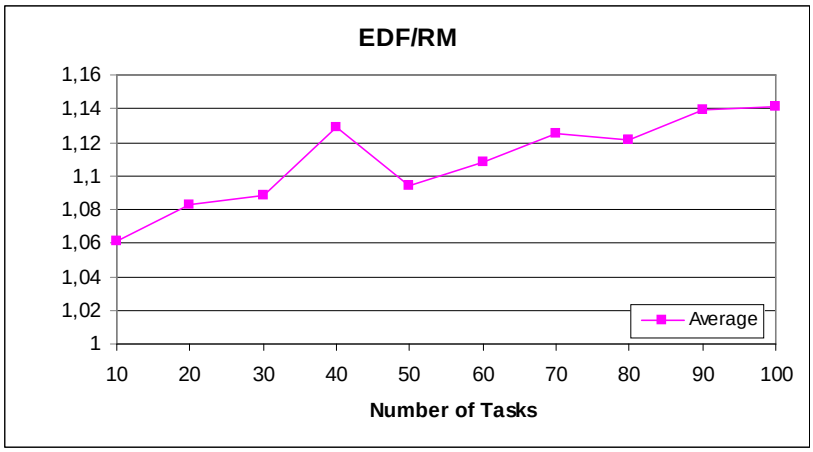


Figure 7.2: EDF normalized against RM, for average USR execution time.

- **Server Ready Queue** to store all servers that have non zero remaining budget. When a server is released at the beginning of its period, its budget will be charged to the maximum budget Q, and the server will be added to the Server Ready Queue. When a server executes its internal tasks for some time x, then the remaining budget of the server will be deceased with x, i.e., reduced by the time that the server execute. If the remaining budget becomes zero, then the server will hand over the control to the global scheduler to select and remove the highest priority server from Server Ready Queue.
- **Server TEQ** to release the server at its next absolute periodic time since we are using periodic servers and also track their remaining budgets.

Figures 7.4 illustrates the implementation of HSF in VxWorks. The Server Ready Queue is managed by the routine that is responsible for scheduling the servers. Tracking the remaining budget of a server is solved as follows; whenever a server starts running, it sets an absolute time at which the server budget expire and it equals to the current time plus its remaining budget. This time is added to the server event Queue to be used by the timer to trigger an event when the server budget expires. When a server is preempted by another server, it updates the remaining budget by subtracting the time that has passed since

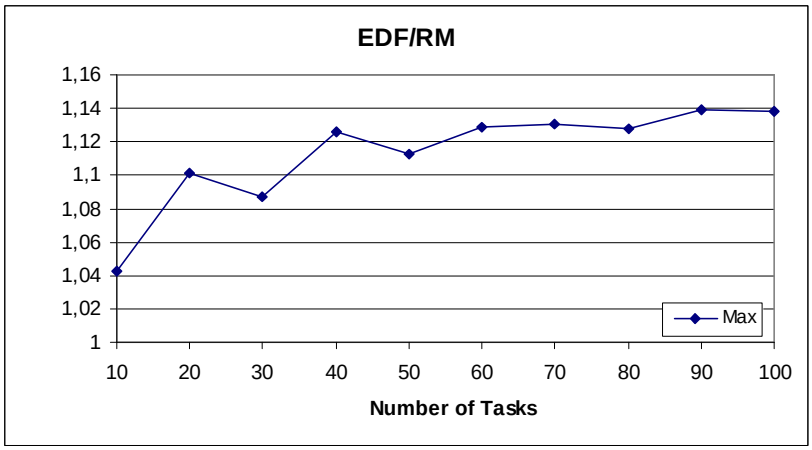


Figure 7.3: EDF normalized against RM, for maximum USR execution time.

the last release. When the server executes its internal tasks until the time when the server budget expiry event triggers, it will set its remaining budget to zero, and the scheduling routine removes the server from the Server Ready Queue.

Local scheduler: When a server is given the CPU resources, the ready tasks that belong to the server will be able to execute. We have investigated two approaches to deal with the tasks in the Ready Queue when a server is given CPU resources:

- All tasks that belong to the server that was previously running will be removed from the Ready Queue, and all ready tasks that belong to the new running server will be added to the Ready Queue, i.e., swapping of the servers' task sets. To remove tasks from the Ready Queue, the state of the tasks is changed to suspend state. However, this will cause a problem since the state of the tasks that finish their execution is also changed to suspend and when the server run again it will add non-ready tasks to the Ready Queue. To solve this problem, an additional flag is used in the task's TCB to denote whether the task was removed from Ready Queue and enter to suspend state due to budget expiration of its server or due to finishing its execution.

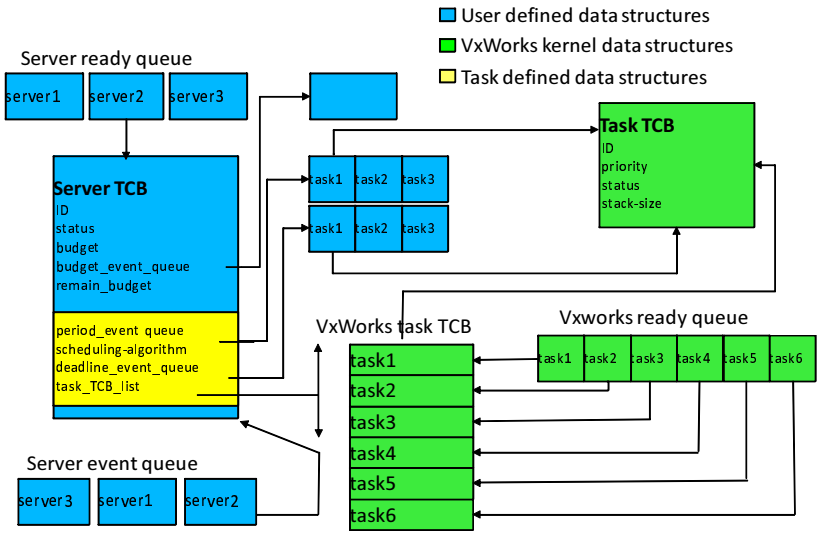


Figure 7.4: The implementation of HSF in VxWorks.

- The priority of all tasks that belong to the preempted server will be set to a lower (the lowest) priority, and the priority of all tasks that belong to the new running server will be raised as if they were executing exclusively on the CPU, scheduled according to the local scheduling policy in use by the subsystem.

The advantage of the second approach is that it can give the unused CPU resources to tasks that belong to other servers. However, the disadvantage of this approach is that the kernel scheduler always sorts the tasks in the Ready Queue and the number of tasks inside Ready Queue using the second approach will be higher which may impose more overhead for sorting tasks. In this paper, we consider the first approach since we support only periodic tasks. When a server is running, all interrupts that are caused by the local TEQ, e.g., releasing tasks and checking deadline misses, can be served without problem. However, if a task is released or its deadline occurs during the execution of another server, the server that includes the task, may miss this event. To solve this problem, when the server starts running after server preemption or when it finishes its budget, it will check for all past events (including task release and

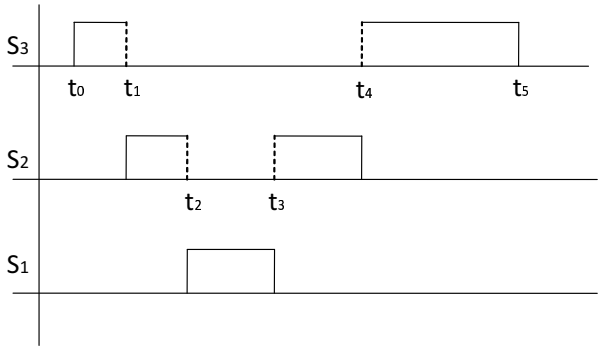


Figure 7.5: Simple servers execution example.

deadline miss check events) in the local TEQ that have absolute time less than the current time, and serve them.

Note that the time wrapping algorithm described in section 7.5.1 should take into account all local TEQ's for all servers and the server event queue, because all these event queues share the same absolute time.

Figure 7.5 illustrates the implementation of hierarchical scheduling framework which includes an example with three servers S_1, S_2, S_3 with global and local RM schedulers, the priority of S_1 is the highest and the priority of S_3 is the lowest. Suppose a new period of S_3 starts at time t_0 with a budget equal to Q_3. Then, the USR will change the state of S_3 to Ready, and since it is the only server that is ready to execute, the USR will;

- add the time at which the budget will expire, which equals to $t_0 + Q_3$, into the server event queue and also add the next period event in the server event queue.
- check all previous events that have occurred while the server was not active by checking if there are task releases or deadline checks in the time interval of $[t^*, t_0]$, where t^* is the latest time at which the budget of S_3 has been expired.
- start the local scheduler.

At time t_1 the server S_2 becomes Ready and it has higher priority than S_3. So S_2 will preempt S_3 and in addition to the previously explained action, the

USR will remove all tasks that belong to S_3 from the ready queue and save the remaining budget which equals to $Q_3 - (t_1 - t_0)$. Also the USR will remove the budget expiration event from the server event queue. Note that when S_3 executes next time it will use the remaining budget to calculate the budget expiration event.

Number of servers	Max	Average	Min
10	91	89	85
20	149	146	139
30	212	205	189
40	274	267	243
50	344	333	318
60	412	400	388
70	483	466	417
80	548	543	509
90	630	604	525
100	689	667	570

Table 7.2: Maximum, average and minimum execution time of the USR with 100 measured values as a function of the number of servers.

The USR execution time depends on the number of the servers, and the worst case happens when all servers are released at the same time. In addition, the execution time of the USR also depends on the number of ready tasks in both the currently running server to be preempted as well as the server to preempt. The USR removes all ready tasks that belong to the preempted server from ready queue and adds all ready tasks that belong to the preempting server with highest priority into the ready queue. Here, the worst case scenario is that all tasks of both servers are ready at that time. Table 7.2 shows the execution time of the USR (when a server is released) as a function of the number of servers using RM as a global scheduler at the worst case, where all the servers are released at the same time, just like the case shown in the previous section. Here, we consider that each server has a single task in order to purely investigate the effect of the number of servers on the execution time of the USR.

7.6.2 Example

In this section, we will show the overall effect of implementing the HSF using a simple example, however, the results from the following example are specific

for this example because, as we showed in the previous section, the overhead is a function of many parameters affect the number of preemptions such as number of servers, number of tasks, servers periods and budgets. In this example we use RM as both local and global scheduler, and the servers and associated tasks parameters are shown in Table 7.3. Note that $T_i = D_i$ for all tasks.

$S_1(P_1 = 5, Q_1 = 1)$			$S_2(P_2 = 6, Q_2 = 1)$			$S_3(P_3 = 70, Q_3 = 20)$		
τ_i	T_i	C_i	τ_i	T_i	C_i	τ_i	T_i	C_i
τ_1	20	1	τ_1	25	1	τ_1	140	7
τ_2	25	1	τ_2	35	1	τ_2	150	7
τ_3	30	1	τ_3	45	1	τ_3	300	30
τ_4	35	1	τ_4	50	1			
τ_5	40	7	τ_5	55	7			
-	-	-	τ_6	60	7			

Table 7.3: System parameters in μs.

The measured overhead utilization is about 2.85% and the measured release jitter for task τ_3 in server S_3 (which is the lowest priority task in the lowest priority server) is about $49ms$. The measured worst case response time is $208.5ms$ and the finishing time jitter is $60ms$. These results indicate that the overhead and performance of the implementation are acceptable for further development in future project.

7.7 Summary

This paper has presented our work on the implementation of our hierarchical scheduling framework in a commercial operating system, VxWorks. We have chosen to implement it in VxWorks so that it can easily be tested in an industrial setting, as we have a number of industrial partners with applications running on VxWorks and we intend to use them as case studies for an industrial deployment of the hierarchical scheduling framework.

This paper demonstrates the feasibility of implementing the hierarchical scheduling framework through its implementation over VxWorks. In particular, it presents several measurements of overheads that its implementation imposes. It shows that a hierarchical scheduling framework can effectively achieve the clean separation of subsystems in terms of timing interference (i.e.,

without requiring any temporal parameters of other subsystems) with reasonable implementation overheads.

In the next stage of this implementation project, we intend to implement synchronization protocols in hierarchical scheduling frameworks, e.g., [12]. In addition, our future work includes supporting sporadic tasks in response to specific events such as external interrupts. Instead of allowing them to directly add their tasks into the ready queue, we consider triggering the USR to take care of such additions. We also plan to support aperiodic tasks while bounding their interference to periodic tasks by the use of some server-based mechanisms. Moreover, we intend to extend the implementation to make it suitable for more advanced architectures including multicore processors.

Acknowledgements

The authors wish to express their gratitude to the anonymous reviewers for their helpful comments, as well as to Clara Maria Otero Pérez for detailed information regarding the implementation of hierarchical scheduling as a dedicated layer on top of pSoSystem, which is marketed by Wind River (see [29] for more details) and suggestions for improving our work.

Bibliography

[1] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the* 18^{th} *IEEE International Real-Time Systems Symposium (RTSS'97)*, pages 308–319, San Francisco, CA, USA, December 1997. IEEE Computer Society.

[2] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the* 20^{th} *IEEE International Real-Time Systems Symposium (RTSS'99)*, pages 256–267, Phoenix, AZ, USA, December 1999. IEEE Computer Society.

[3] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, Washington DC, USA, May-June 2000. IEEE Computer Society.

[4] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium(RTAS)*, pages 75–84, Taipei, Taiwan ROC, May 2001.

[5] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* 24^{th} *IEEE International Real-Time Systems Symposium(RTSS'03)*, pages 2–13, Cancun, Mexico, December 2003.

[6] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT '04)*, pages 95–103, Pisa, Italy, September 2004.

[7] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the* 26^{th} *IEEE International Real-Time Systems*

Symposium (RTSS'05), pages 389–398, Miami Beach, FL, USA, December 2005.

[8] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* 23^{th} *IEEE International Real-Time Systems Symposium (RTSS'02)*, pages 26–35, Austin, TX, USA, December 2002.

[9] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 151–158, Porto, Portugal, July 2003. IEEE Computer Society.

[10] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarhical fixed-priority scheduling. In *Proceedings of the* 14^{th} *Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 152–160, Vienna, Austria, June 2002. IEEE Computer Society.

[11] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the* 25^{th} *IEEE International Real-Time Systems Symposium(RTSS'04)*, pages 57–67, Lisbon, Portugal, December 2004. IEEE Computer Society.

[12] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, pages 279–288, Salzburg, Austria, October 2007.

[13] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the* 27^{th} *IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, Rio de Janeiro, Brazil, December 2006.

[14] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, Washington, DC, USA, December 2007. IEEE Computer Society.

[15] G. Buttazzo and P. Gai. Efficient implementation of an EDF scheduler for small embedded systems. In *Proceedings of the* 2^{nd} *International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'06) in conjunction with the* 18^{th} *Euromicro International*

Conference on Real-Time Systems (ECRTS'06), Dresden, Germany, July 2006.

[16] Evidence Srl. ERIKA Enterprise RTOS. URL: http://www.evidence.eu.com.

[17] F. Slomka G. Wirrer C. Diederichs, U. Margull. An application-based EDF scheduler for osek/vdx. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 87–88, 3001 Leuven, Belgium, Belgium, 2008. European Design and Automation Association.

[18] D. Kim, Y. Lee, and M. Younis. Spirit-ukernel for strongly partitioned real-time systems. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, page 73, Cheju Island, South Korea, December 2000. IEEE Computer Society.

[19] L. Kinnan P. Parkinson. Safety critical software development for integrated modular avionics. In *Wind River white paper. URL http://www.windriver.com/whitepapers/*, pages 87–88, 3001 Leuven, Belgium, Belgium, 2007. European Design and Automation Association.

[20] Wind River. Wind River VxWorks 5.x. http://www.windriver.com/.

[21] Wind River. VxWorks PROGRAMMERS GUIDE 5.5.

[22] J.W.S. Liu. Real-time systems. *Prentice Hall*, 2000.

[23] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of* 8^{th} *IEEE International Real-Time Systems Symposium (RTSS'87)*, pages 261–270, San Jose, California, USA, December 1987. IEEE Computer Society.

[24] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[25] Wind River. VxWorks KERNEL PROGRAMMERS GUIDE 6.2.

[26] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the* 20^{th} *IEEE International Real-Time Systems Symposium(RTSS'89)*, pages 166–171, Santa Monica, CA, USA, December 1989. IEEE Computer Society.

[27] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29(1):5–26, January 2005.

[28] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Journal of Real-Time Systems*, 2:301–324, 1990.

[29] C.M. Otero Perez and I. Nitescu. Quality of service resource management for consumer terminals: Demonstrating the concepts. *Work in Progress Session of the 14th Euromicro Conference on Real-Time Systems(ECRTS'02)*, pages 29–32, June 2002.

Chapter 8

Paper B: SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems

Moris Behnam, Insik Shin, Thomas Nolte, Mikael Nolin

In Proceedings of the 7^{th} ACM & IEEE International Conference on Embedded Software (EMSOFT'07), pages 279-288, Salzburg, Austria, October, 2007.

Abstract

This paper presents a protocol for resource sharing in a hierarchical real-time scheduling framework. Targeting real-time open systems, the protocol and the scheduling framework significantly reduce the efforts and errors associated with integrating multiple semi-independent subsystems on a single processor. Thus, our proposed techniques facilitate modern software development processes, where subsystems are developed by independent teams (or subcontractors) and at a later stage integrated into a single product. Using our solution, a subsystem need not know, and is not dependent on, the timing behaviour of other subsystems; even though they share mutually exclusive resources. In this paper we also prove the correctness of our approach and evaluate its efficiency.

8.1 Introduction

In many industrial sectors integration of electronic and software subsystems (to form an integrated hardware and software system), is one of the activities that is most difficult, time consuming, and error prone [1, 2]. Almost any system, with some level of complexity, is today developed as a set of semi-independent subsystems. For example, cars consist of multiple subsystems such as antilock braking systems, airbag systems and engine control systems. In the later development stages, these subsystems are integrated to produce the final product. Product domains where this approach is the norm include automotive, aerospace, automation and consumer electronics.

It is not uncommon that these subsystems are more or less dependent on each other, introducing complications when subsystems are to be integrated. This is especially apparent when integrating multiple software subsystems on a single processor. Due to these difficulties inherent in the integration process, many projects run over their estimated budget and deadlines during the integration phase. Here, a large source of problems when integrating real-time systems stems from subsystem interference in the time domain.

To provide remedy to these problems we propose the usage of a real-time scheduling framework that allows for an easier integration process. The framework will preserve the essential temporal properties of the subsystem both when the subsystem is executed in isolation (unit testing) and when it is integrated together with other subsystems (integration testing and deployment). Most importantly, the deviation in the temporal behaviour will be bounded, hence allowing for predictable integration of hard real-time subsystems. This is traditionally targeted by the philosophy of open systems [3], allowing for the independent development and validation of subsystems, preserving validated properties also after integration on a common platform.

In this paper we present the Subsystem Integration and Resource Allocation Policy (SIRAP), which makes it possible to develop subsystems individually without knowledge of the temporal behaviour of other subsystems. One key issue addressed by SIRAP is the resource sharing between subsystems that are only semi-independent, i.e., they use one or more shared logical resources.

Problem description A software system $\mathcal{S}$ consists of one or more subsystems to be executed on one single processor. Each subsystem $S_s \in \mathcal{S}$, in turn, consists of a number of tasks. These subsystems can be developed independently and they have their own local scheduler (scheduling the subsystem's tasks). This approach by isolation of tasks within subsystems, and allowing for

their own local scheduler, has several advantages [4]. For example, by keeping a subsystem isolated from other subsystems, and by keeping the subsystem local scheduler, it is possible to re-use a complete subsystem in a different application from where it was originally developed.

However, as subsystems are likely to share logical resources, an appropriate resource sharing protocol must be used. In order to facilitate independent subsystem development, this protocol should not require information from all other subsystems in the system. It should be enough with only the information of the subsystem under development in isolation.

Contributions The main contributions of this paper include the presentation of SIRAP, a novel approach to subsystem integration in the presence of shared resources. Moreover, the paper presents the deduction of bounds on the timing behaviour of SIRAP together with accompanying formal proofs. In addition, the cost of using this protocol is thoroughly evaluated. The cost is investigated as a function of various parameters including: cost as a function of the length of critical sections, cost depending on the priority of the task sharing a resource, and cost depending on the periodicity of the subsystem. Finally, the cost of having an independent subsystem abstraction, which is suitable for open systems, is investigated and compared with dependent abstractions.

Organization of the paper Firstly, related work on hierarchical scheduling and resource sharing is presented in Section 8.2. Then, the system model is presented in Section 8.3. SIRAP is presented in Section 8.4. In Section 8.5 schedulability analysis is presented, and SIRAP is evaluated in Section 8.6. Finally, the paper is summarized in Section 8.7.

8.2 Related work

Hierarchical scheduling For real-time systems, there has been a growing attention to hierarchical scheduling frameworks [5, 6, 3, 7, 8, 9, 10, 11, 12, 13, 14].

Deng and Liu [3] proposed a two-level hierarchical scheduling framework for open systems, where subsystems may be developed and validated independently in different environments. Kuo and Li [8] presented schedulability analysis techniques for such a two-level framework with the fixed-priority global scheduler. Lipari and Baruah [9, 15] presented schedulability analysis techniques for the EDF-based global schedulers.

Mok *et al.* [16] proposed the bounded-delay resource partition model for a hierarchical scheduling framework. Their model can specify the real-time guarantees that a parent component provides to its child components, where the parent and child components have different schedulers. Feng and Mok [7] and Shin and Lee [14] presented schedulability analysis techniques for the hierarchical scheduling framework that employs the bounded-delay resource partition model.

There have been studies on the schedulability analysis with the periodic resource model. This periodic resource model can specify the periodic resource allocation guarantees provided to a component from its parent component [13]. Saewong *et al.* [12] and Lipari and Bini [10] introduced schedulability conditions for fixed-priority local scheduling, and Shin and Lee [13] presented a schedulability condition for EDF local scheduling. Davis and Burns [6] evaluated different periodic servers (Polling, Deferrable, and Sporadic Servers) for fixed-priority local scheduling.

Resource sharing When several tasks are sharing a logical resource, typically only one task is allowed to use the resource at a time. Thus the logical resource requires mutual exclusion of tasks that uses it. To achieve this a *mutual exclusion protocol* is used. The protocol provides rules about how to gain access to the resource, and specifies which tasks should be blocked when trying to access the resource.

To achieve predictable real-time behaviour, several protocols have been proposed including the Priority Inheritance Protocol (PIP) [17], the Priority Ceiling Protocol (PCP) [18], and the Stack Resource Policy (SRP) [19].

When using SRP, a task may not preempt any other tasks until its priority is the highest among all tasks that are ready to run, and its preemption level is higher than the system ceiling. The preemption level of a task is a static parameter assigned to the task at its creation, and associated with all instances of that task. A task can only preempt another task if its preemption level is higher than the task that it is to preempt. Each resource in the system is associated with a resource ceiling and based on these resource ceilings, a system ceiling can be calculated. The system ceiling is a dynamic parameter that changes during system execution.

The duration of time that a task lock a resource, is called Resource Holding Time (RHT). Fisher *et al.* [20, 21] proposed algorithms to minimize RHT for fixed priority and EDF scheduling with SRP as a resource synchronization protocol. The basic idea of their proposed algorithms is to increase the ceiling of resources as much as possible without violating the schedulability of the

system under the same semantics of SRP.

Deng and Liu [3] proposed the usage of non-preemptive global resource access, which bounds the maximum blocking time that a task might be subject to. The work by Kuo and Li [8] used SRP and they showed that it is very suitable for sharing of local resources in a hierarchical scheduling framework. Almeida and Pedreiras [5] considered the issue of supporting mutually exclusive resource sharing within a subsystem. Matic and Henzinger [11] considered supporting interacting tasks with data dependency within a subsystem and between subsystems, respectively.

More recently, Davis and Burns [22] presented the Hierarchical Stack Resource Policy (HSRP), allowing their work on hierarchical scheduling [6] to be extended with sharing of logical resources. However, using HSRP, information on all tasks in the system must be available at the time of subsystem integration, which is not suitable for an open systems development environment, and this can be avoided by the SIRAP protocol presented in this paper.

8.3 System model

8.3.1 Hierarchical scheduling framework

A hierarchical scheduling framework is introduced to support CPU time sharing among applications (subsystems) under different scheduling services. Hence, a system $\mathcal{S}$ consists of one or more subsystems $S_s \in \mathcal{S}$. The hierarchical scheduling framework can be generally represented as a two-level tree of nodes, where each node represents a subsystem with its own scheduler for scheduling internal tasks (threads), and CPU time is allocated from a parent node to its children nodes, as illustrated in Figure 8.1.

The hierarchical scheduling framework provides *partitioning* of the CPU between different subsystems. Thus, subsystems can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification and unit testing.

The hierarchical scheduling framework is also useful in the domain of open systems [3], where subsystems may be developed and validated independently in different environments. For example, the hierarchical scheduling framework allows a subsystem to be developed with its own scheduling algorithm internal to the subsystem and then later included in a system that has a different global level scheduler for scheduling subsystems.

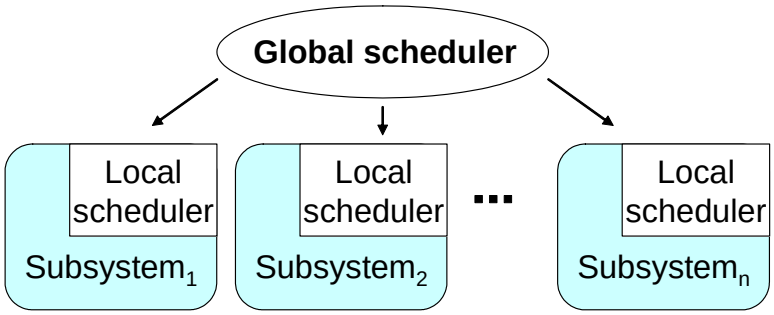


Figure 8.1: Two-level hierarchical scheduling framework.

8.3.2 Shared resources

For the purpose of this paper a shared (logical) resource, r_i, is a shared memory area to which only one task at a time may have access. To access the resource a task must first lock the resource, and when the task no longer needs the resource it is unlocked. The time during which a task holds a lock is called a *critical section*. Only one task at a time may lock each resource.

A resource that is used by tasks in more than one subsystem is denoted a *global shared resource*. A resource only used within a single subsystem is a *local shared resource*. In this paper we are concerned only with global shared resources and will simply denote them by shared resources. Management of local shared resources can be done by using any synchronization protocol such as PIP, PCP, and SRP.

8.3.3 Virtual processor model

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

Shin and Lee [13] proposed the periodic virtual processor model $\Gamma(\Pi, \Theta)$, where Π is a period ($\Pi > 0$) and Θ is a periodic allocation time ($0 < \Theta \leq \Pi$). The capacity U_Γ of a periodic virtual processor model $\Gamma(\Pi, \Theta)$ is defined as Θ/Π. The periodic virtual processor model $\Gamma(\Pi, \Theta)$ is defined to characterize the following property:

$$\mathrm{supply}_\Gamma\big(k\Pi, (k+1)\Pi\big) = \Theta, \quad \text{where } k = 0, 1, 2, \ldots, \tag{8.1}$$

where the supply function $\mathrm{supply}_{R_s}(t_1, t_2)$ computes the amount of CPU allocations that the virtual processor model R_s provides during the interval $[t_1, t_2)$.

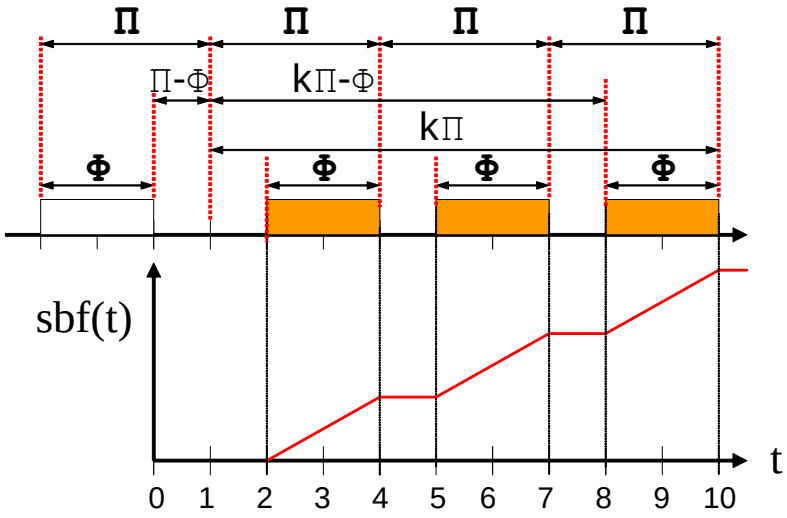


Figure 8.2: The supply bound function of a periodic virtual processor model $\Gamma(\Pi, \Theta)$ for $k = 3$.

For the periodic model $\Gamma(\Pi, \Theta)$, its supply bound function $\mathtt{sbf}_\Gamma(t)$ is defined to compute the minimum possible CPU supply for every interval length t as follows:

$$\mathtt{sbf}_\Gamma(t) = \begin{cases} t - (k+1)(\Pi - \Theta) & \text{if } t \in [(k+1)\Pi - 2\Theta, \\ & (k+1)\Pi - \Theta], \\ (k-1)\Theta & \text{otherwise,} \end{cases} \tag{8.2}$$

where $k = \max\left(\left\lceil \big(t - (\Pi - \Theta)\big)/\Pi \right\rceil, 1\right)$. Here, we first note that an interval of length t may not begin synchronously with the beginning of period Π. That is, as shown in Figure 8.2, the interval of length t can start in the middle of the period of a periodic model $\Gamma(\Pi, \Theta)$. We also note that the intuition of k in Eq. (8.2) basically indicates how many periods of a periodic model can overlap the interval of length t, more precisely speaking, the interval of length $t - (\Pi - \Theta)$. Figure 8.2 illustrates the intuition of k and how the supply bound function $\mathtt{sbf}_\Gamma(t)$ is defined for $k = 3$.

8.3.4 Subsystem model

A subsystem $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the whole system of subsystems, consists of a task set and a scheduler. Each subsystem S_s is associated with a periodic virtual processor model abstraction $\Gamma_s(\Pi_s, \Theta_s)$, where Π_s and Θ_s are the subsystem period and budget respectively. This abstraction $\Gamma_s(\Pi_s, \Theta_s)$ is supposed to specify the collective temporal requirements of a subsystem, in the presence of global logical resource sharing.

Task model We consider a periodic task model $\tau_i(T_i, C_i, \mathcal{X}_i)$, where T_i and C_i represent the task's period and worst-case execution time (WCET) respectively, and $\mathcal{X}_i$ is the set of WCETs within critical sections belonging to τ_i. Each element $x_{i,j}$ in $\mathcal{X}_i$ represents the WCET of a particular critical section $cx_{i,j}$ executed by τ_i. Note that C_i includes all $x_{i,j} \in \mathcal{X}_i$.

The set of critical sections cover for the following two cases of multiple critical sections within one job:

1. sequential critical sections, where $\mathcal{X}_i$ contains the WCETs of all sequential critical sections, i.e. $\mathcal{X}_i = \{x_{i,1}, ..., x_{i,o}\}$ where o is the number of sequential shared resources that task τ_i may lock during its execution.

2. nested critical sections, where $x_{i,j} \in \mathcal{X}$ being the length of the outer critical section.

Note that in the remaining paper, we use x_i rather than $x_{i,j}$ for simplicity when it is not necessary to indicate j.

Scheduler In this paper, we assume that each subsystem has a fixed-priority preemptive scheduler for scheduling its internal tasks.

8.4 SIRAP protocol

8.4.1 Terminology

Before describing the SIRAP protocol, we define the terminology (also depicted in Figure 8.3) that are related to hierarchical logical resource sharing.

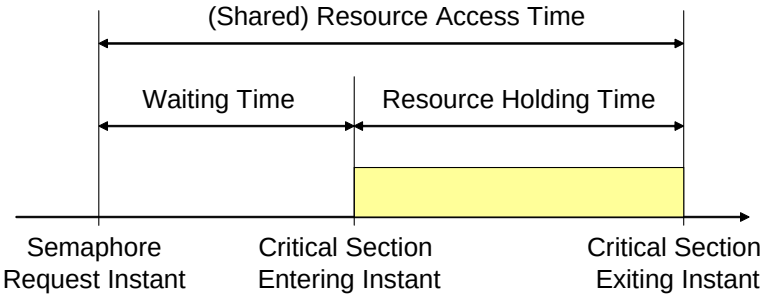


Figure 8.3: Shared resource access time.

- *Semaphore request instant:* an instant at which a job tries to enter a critical section guarded by a semaphore.
- *Critical section entering (exiting) instant:* an instant at which a job enters (exits) a critical section.
- *Waiting time:* a duration from a semaphore request time to a critical section entering time.
- *Resource holding time:* a duration from a critical section entering instant to a critical section exiting instant. Let $h_{i,j}$ denote the resource holding time of a critical section $cx_{i,j}$ of task τ_i.
- *(Shared) resource access time:* a duration from a semaphore request instant to a critical section exiting time.

In addition, a context switch is referred to as *task-level context switch* if it happens between tasks within a subsystem, or as *subsystem-level context switch* if it happens between subsystems.

8.4.2 SIRAP protocol description

The subject of this paper is to develop a synchronization protocol that can address global resource sharing in hierarchical real-time scheduling frameworks, while aiming at supporting independent subsystem development and validation. This section describes our proposed synchronization protocol, SIRAP (Subsystem Integration and Resource Allocation Policy).

Assumption SIRAP relies on the following assumption:

- The system's global scheduler schedules subsystems according to their periodic virtual processor abstractions $\Gamma_s(\Pi_s, \Theta_s)$. The subsystem budget is consumed every time when an internal task within a subsystem executes, and the budget is replenished to Θ_s every subsystem period Π_s. Similar to traditional server-based scheduling methods [23], the system provides a run-time mechanism such that each subsystem is able to figure out at any time t how much its remaining subsystem budget Θ_s is, which will be denoted as $\Theta'_s(t)$ in the remaining of this section.

The above assumption is necessary to allow run-time checking whether or not a job can potentially enter and execute a whole critical section before a subsystem-budget expire. This is useful particularly for supporting independent abstraction of subsystem's temporal behavior in the presence of global resource accesses.

In addition to supporting independent subsystem development, SIRAP also aims at minimizing the resource holding time and bounding the waiting time at the same time. To achieve this goal, the protocol has two key rules as follows:

R1 When a job enters a critical section, preemptions from other jobs within the same subsystem should be bounded to keep its resource holding time as small as possible.

R2 When a job wants to enter a critical section, it enters the critical section at the earliest instant such that it can complete the critical section before the subsystem-budget expires.

The first rule R1 aims at minimizing a resource holding time so that the waiting time of other jobs, which want to lock the same resource, can be minimized as well. The second rule R2 prevents a job J_i from entering a critical section $cx_{i,j}$ at any time t when $\Theta'(t) < h_{i,j}$. This rule guarantees that when the budget of a subsystem expires, no task within the subsystem locks a global shared resource.

SIRAP : preemption management The SRP [19] is used to enforce the first rule R1. Each subsystem will have its own system ceiling and resources ceiling according to its jobs that share global resources. According to SRP, whenever a job locks a resource, other jobs within the same subsystem can preempt it if the jobs have higher preemption levels than the locked resource ceiling, so as to bound the blocking time of higher-priority jobs. However, such task-level preemptions generally increase resource holding times and can potentially increase subsystem utilization. One approach to minimize $h_{i,j}$ is to allow no task-level preemptions, by assigning the ceiling of global resource equal to the maximum preemption level. However, increasing the resource ceiling to the maximum preemption level may affect the schedulability of a subsystem. A good approach is presented in [20], which increases the ceiling of shared global resources as much as possible while keeping the schedulability of the subsystem.

SIRAP : self-blocking When a job J_i tries to enter a critical section, SIRAP requires each local scheduler to perform the following action. Let t_0 denote the semaphore request instant of J_i and $\Theta'(t_0)$ denote the subsystem's budget at time t_0.

- If $h_{i,j} \leq \Theta'(t_0)$, the local scheduler executes the job J_i. The job J_i enters a critical section at time t_0.
- Otherwise, i.e., if $h_{i,j} > \Theta'(t_0)$, the local scheduler delays the critical section entering of the job J_i until the next subsystem budget replenishment. This is defined as *self-blocking*. Note that the system ceiling will be equal to resource ceiling at time t_0, which means that the jobs that have preemption level greater than system ceiling can only execute during the self blocking interval[1]. This guarantees that when the subsystem of J_i receives the next resource allocation, the subsystem-budget will be enough to execute job J_i inside the critical section[2].

[1] With simple modifications to the SRP protocol, the execution of tasks can be allowed within the self blocking interval if they do not access global resources even though their preemption levels are less than the system ceiling. However this is off the point of this paper.

[2] The idea of self-blocking has been also considered in different contexts, for example, in CBS-R [23] and zone based protocol (ZB) [24]. Our work is different from those in the sense that CBS-R used a similar idea for supporting soft real-time tasks, and ZB used it in a pfair-scheduling environment, while we use it for hard real-time tasks under hierarchical scheduling. This difference inherently requires the development of different schedulability analysis, including Eqs. (8.5), (8.6), and (8.7).

8.5 Schedulability analysis

8.5.1 Local schedulability analysis

Consider a subsystem S_s that consists of a periodic task set and a fixed-priority scheduler and receives CPU allocations from a virtual processor model $\Gamma_s(\Pi_s, \Theta_s)$. According to [13], this subsystem is schedulable if

$$\forall \tau_i, 0 < \exists t \leq T_i \ \ \mathtt{dbf}_{\mathtt{FP}}(i,t) \leq \mathtt{sbf}_{\Gamma}(t). \tag{8.3}$$

The goal of this section is to develop the demand bound function $\mathtt{dbf}_{\mathtt{FP}}(i,t)$ calculation for the SIRAP protocol. $\mathtt{dbf}_{\mathtt{FP}}(i,t)$ is computed as follows;

$$\mathtt{dbf}_{\mathtt{FP}}(i,t) = C_i + I_S(i) + I_H(i,t) + I_L(i), \tag{8.4}$$

where C_i is the WCET of τ_i, $I_S(i)$ is the maximum self blocking for τ_i, $I_H(i,t)$ is the maximum possible interference imposed by a set of higher-priority tasks to a task τ_i during an interval of length t, and $I_L(i)$ is the maximum possible interference imposed by a set of lower-priority tasks that share resources with preemption level (ceiling) greater than or equal to the priority of task τ_i.

The following lemmas shows how to compute $I_S(i)$, $I_H(i,t)$ and $I_L(i)$.

Lemma 1. *Self-blocking imposes to a job J_i an extra processor demand of at most $\sum_{j=1}^{o} h_{i,j}$ if a job access multiple shared resources.*

Proof. When the job J_i self-blocks itself, it consumes the processor of at most $h_{i,j}$ units being idle. If the job access shared resources then the worst case will happen when the job block itself whenever it tries to enter a critical section. □

Lemma 2. *A job J_i can be interfered by a higher-priority job J_j that access shared resources, at t time units for a duration of at most $\lceil \frac{t}{T_j} \rceil (C_j + \sum_{k=1}^{o} h_{j,k})$ time units.*

Proof. Similar to classical response time analysis [25], we add $\sum_{k=1}^{o} h_{j,k}$ to C_j which is the worst case self blocking from higher priority tasks, the lemma follows. □

Lemma 3. *A job J_i can be interfered by only one lower-priority job J_j by at most $2 \cdot max(h_{j,k})$, where k=1,...,o.*

Proof. A higher-priority job J_i can be interfered by a lower-priority job J_j. This occurs only if J_i is released after J_j tries to enter a critical section but before J_j exits the critical section. When J_i is released, only one job can try to enter or be inside a critical section. That is, a higher-priority job J_i can then be interfered by at most a single lower-priority job. The processor demand of J_j during a critical section period is bounded by $2 \cdot max(h_{j,k})$ for the worst case. The lemma follows. □

From Lemma 1, the self-blocking $I_S(i)$ is given by;

$$I_S(i) = \sum_{k=1}^{o} h_{i,k} \tag{8.5}$$

According to Lemma 2 and taking into account the interference from higher priority tasks, $I_H(i,t)$ is computed as follows;

$$I_H(i,t) = \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil (C_j + \sum_{k=1}^{o} h_{j,k}). \tag{8.6}$$

The maximum interference from lower priority tasks can be evaluated according to Lemma 3 according to;

$$I_L(i) = \max_{j=i+1,\ldots,n} (2 \cdot \max_{k=1,\ldots,o} (h_{j,k})). \tag{8.7}$$

Based on Eq. (8.5) and (8.6) and (8.7), the processor demand bound function is given by Eq. (8.4).

The resource holding time $h_{i,j}$ of a job J_i that access a global resource is evaluated as the maximum critical section execution time $x_{i,j}+$ the maximum interference from the tasks that have preemption level greater than the ceiling of the logical resource during the execution $x_{i,j}$. $h_{i,j}$ is computed [20] using $W_{i,j}(t)$ as follows;

$$W_{i,j}(t) = x_{i,j} + \sum_{l=ceil(x_{i,j})+1}^{u} \lceil \frac{t}{T_l} \rceil C_l, \tag{8.8}$$

where $ceil(x_{i,j})$ is the ceiling of the logical resource accessed within the critical section $x_{i,j}$, and C_l, T_l are the worst case execution time and the period of job that have higher preemption level than $ceil(x_{i,j})$, and u is the maximum ceiling within the subsystem.

The resource holding time $h_{i,j}$ is the smallest time t_i^* such that $W_{i,j}(t_i^*) = t_i^*$.

8.5.2 Global schedulability analysis

Here, issues for global scheduling of multiple subsystems are dealt with. For a subsystem S_s, it is possible to derive a periodic virtual processor model $\Gamma_s(\Pi_s, \Theta_s)$ that guarantees the schedulability of the subsystem S_s according to Eq. (8.3).

The local schedulability analysis presented for subsystems is not dependent on any specific global scheduling policy. The requirements for the global scheduler, are as follows: i) it should schedule all subsystems according to their virtual processor model $\Gamma_s(\Pi_s, \Theta_s)$, ii) it should be able to bound the waiting time of a task in any subsystem that wants to access global resource.

To achieve those global scheduling requirements, preemptive schedulers such as EDF and RM together with the SRP [19] synchronization protocol can be used. So when a subsystem locks a global resource, it will not be preempted by other subsystems that have preemption level less than or equal to the locked resource ceiling. Each subsystem, for all global resources accessed by tasks within a subsystem, should specify a list of pairs of all those global resources and their maximum resource holding times $\{(r_1, H_{r_1}), ..., (r_p, H_{r_p})\}$. However it is possible to minimize the required information that should be provided for each subsystem by assuming that all global resources have the same ceiling equal to the maximum preemption level $\hat{\pi}_s$ among all subsystems. Then for the global scheduling, it is enough to provide virtual processor model $\Gamma_s(\Pi_s, \Theta_s)$ and the maximum resource holding times among all global resources $\hat{H}_s = max(H_{R_1}, ..., H_{R_p})$ for each subsystem S_s. On the other hand, assigning the ceiling of all global resources to the maximum preemption level of the subsystem that access these resources is not as efficient as using the original SRP protocol, this since we may have resources with lower ceiling which permit more preemptions from the higher preemption level subsystems.

Under EDF global scheduling, a set of n subsystems is schedulable [19] if

$$\forall k_{k=1,...,n}(\sum_{i=1}^{k} \frac{\Theta_i}{\Pi_i}) + \frac{B_k}{\Pi_k} \leq 1, \tag{8.9}$$

where B_k of subsystem S_k is the duration of the longest resource holding time among those belonging to subsystems with preemption level lower than π_k.

For RM global scheduling, the schedulability test based on tasks' response time is

$$W_i = \Theta_i + B_k + \sum_{j=1}^{i-1} \left\lceil \frac{W_i}{\Pi_j} \right\rceil (C_j). \tag{8.10}$$

It is also possible to use a non-preemptive global scheduler together with the SIRAP protocol. In this case, no subsystem-level context switch happens when there is a task inside a critical section. That is, whenever a task tries to lock a global resource, it is guaranteed that the global resource is not locked by another task from other subsystems. This way provides a clean separation between subsystems in accessing global shared resources. Then, we can achieve a more subsystem abstraction, i.e., subsystems do not have to export information about their global shared resource accesses, for example, which global shared resources they access and the maximum resource holding time. In fact, it will require more system resources to schedule subsystems under non-preemptive global scheduling rather than under preemptive global scheduling. Hence, we can see a tradeoff between abstraction and efficiency. Exploring this tradeoff is a topic of our future work.

8.5.3 Local resource sharing

So far, only the problem of sharing global resource between subsystems has been considered. However, many real time applications may have local resource sharing within subsystem as well. Almeida and Pedreiras [5] showed that some traditional synchronization protocols such as PCP and SRP can be used for supporting local resource sharing in a hierarchical scheduling framework by including the effect of local resource sharing in the calculation of $\texttt{dbf}_{\texttt{FP}}$. That is, to combine SRP/PCP and the SIRAP protocol for synchronizing both local and global resources sharing, Eq. (8.7) should be modified to

$$I_L(i) = \max(\max(2 \cdot x_{j,k}), b_i), \quad \text{where } j = i+1, \ldots, n. \tag{8.11}$$

where b_i is the maximum duration for which a task i can be blocked by its lower-priority tasks in critical sections from local resource sharing.

8.6 Protocol evaluation

In this section, the cost of using SIRAP is investigated in terms of extra CPU utilization (U_Γ) required for subsystem schedulability guarantees. We assume

that all global resource ceilings can be equal to the maximum preemption level, which means that no tasks within a subsystem preempt a task inside a critical section, and therefore $h_{i,j} = x_{i,j}$. Supporting logical resource sharing is expected to increase subsystem utilizations U_Γ. This increment in U_Γ depends on many factors such as the maximum WCET within a critical section $x_{i,j}$, the priority of the task sharing a global resource, and the subsystem period Π_s.

Sections 8.6.1, 8.6.2, and 8.6.3 investigate the effect of those factors under the assumption that task i accesses a single critical section. In Section 8.6.4, this assumption is relaxed so as to investigate the effect of the number of critical sections. Section 6.5 compares independent and dependent abstractions in terms of subsystem utilization.

8.6.1 WCET within critical section

One of the main factors that affect the cost of using SIRAP is the value of $x_{i,j}$. It is clear from Eqs. (8.4), (8.6), and (8.7) that whenever $x_{i,j}$ (which equals to $h_{i,j}$) increases, $\texttt{dbf}_{\texttt{FP}}$ will increase as well, potentially causing U_Γ to increase in order to satisfy the condition in Eq. (8.3). Figure 8.4 shows the effect of increasing x_i on two different task sets. Task set 1 is sensitive for small changes in x_i whilst task set 2 can tolerate the given range of x_i without showing a big change in U_Γ. The reason behind the difference is that task set 1 has a task with period very close to Π_s while the smallest task period in task set 2 is greater than Π_s by more than 4 times. Hence, SIRAP can be more or less sensitive to x_i depending on the ratio between task and subsystem period.

For the remaining figures (Figure 8.5 and 8.6), simulations are performed as follows. We randomly generated 100 task sets, each containing 5 tasks. Each task set has a utilization of 25%, and the period of the generated tasks range from 40 to 1000. For each task set, a single task accesses a global shared resource; the task is the highest priority task, the middle priority task, or the lowest priority task. For each task set, we use 11 different values of x_i ranging from 10% to 50% of the subsystem period.

8.6.2 Task priority

From Eqs. (8.4), (8.6) and (8.7), looking how tasks sharing global logical resources affect the calculations of $\texttt{dbf}_{\texttt{FP}}$, it is clear that task priority for these tasks is of importance. The contribution of low priority tasks on $\texttt{dbf}_{\texttt{FP}}$ is fixed to a specific value of x_i (see Eq. (8.7)), while the increase in $\texttt{dbf}_{\texttt{FP}}$ by higher priority tasks depends on many terms such as higher priority task period T_k and

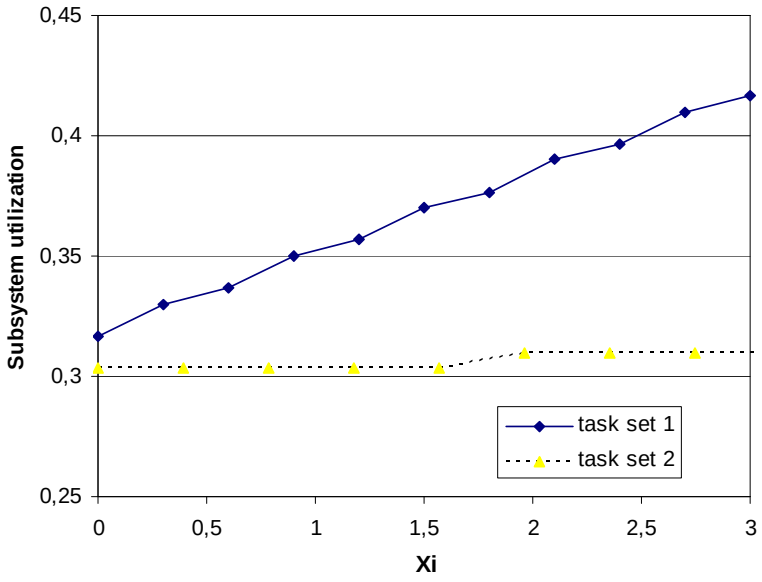


Figure 8.4: U_Γ as a function of x_i for two task sets where only the lowest priority task share a resource.

execution time C_k (see Eq. (8.6)). It is fairly easy to estimate the behaviour of a subsystem when lower priority tasks share global resources; on one hand, if the smallest task period in a subsystem is close to Π_s, U_Γ will be significantly increased even for small values of x_i. As the value of `sbf` is small for time intervals close to Π_s, the subsystem needs a lot of extra resources in order to fulfil subsystem schedulability. On the other hand, if the smallest task period is much larger than Π_s then U_Γ will only be affected for large values of x_i, as shown in Figure 8.4.

Figure 8.5 shows U_Γ as a function of x_i for when the highest, middle and lowest priority task are sharing global resources, respectively, where $\Pi_s = 15$. The figure shows that the highest priority task accessing a global shared resource needs in average more utilization than other tasks with lower priority. This observation is expected as the interference from higher priority task is larger than the interference from lower priority tasks (see Eq. (8.6) and (8.7)). However, note that in the figure this is true for x_i within the range of [0,5]. If

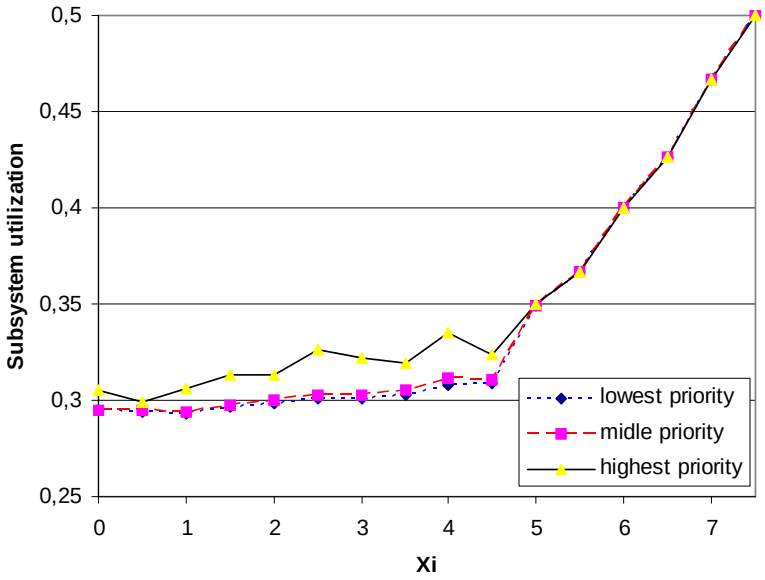


Figure 8.5: Average utilization for 100 task sets as a function of x_i, when low, medium and high priority task share a resource respectively, $\Pi_s = 15$.

the value of x_i is larger than 5, then U_Γ keeps increasing rapidly without any difference among the priorities of tasks accessing the global shared resource. This can be explained as follows. When using SIRAP, the subsystem budget Θ_s should be no smaller than x_i to enforce the second rule R2 in Section 8.4.2. Therefore, when $x_i \geq 5$, Θ_s should also become greater than 5 even though subsystem period is fixed to 15. This essentially results in a rapid increase of U_Γ with the speed of $x_i/15$.

8.6.3 Subsystem period

The subsystem period is one of the most important parameters, both in the context of global scheduling and `sbf` calculations for a subsystem. As Π_s is used in the `sbf` calculations, Π_s will have significant effect on U_Γ (see Eq. (8.3)).

Figure 8.6 compares average subsystem utilization for different values of subsystem period, i.e., for $\Pi_s = 20$ and $\Pi_s = 40$ for the same task sets. Here,

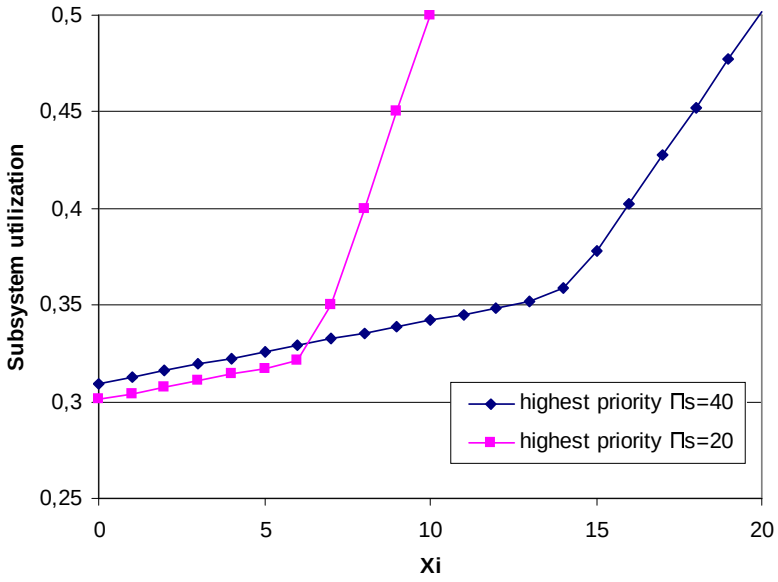


Figure 8.6: Average utilization for 100 task sets as a function of x_i, when only the highest priority tasks share a resource and the subsystem period is $\Pi_s = 20$ and $\Pi_s = 40$.

only the highest priority task accesses a global shared resource. It is interesting to see that the lower value of Π_s, i.e, $\Pi_s = 20$, results in a lower subsystem utilization when x_i is small, i.e., $x_i \leq 6$, and then a higher subsystem utilization when x_i gets larger from $x_i = 6$. That is, x_i and Π_s are not dominating factors one to another, but they collectively affect subsystem utilization. It is also interesting to see in Figure 8.6 that the subsystem utilization of $\Pi_s = 40$ behaves in a similar way by increasing rapidly from $x_i = 14$.

Hence, in general, Π_s should be less than the smallest task period in a subsystem, as in hierarchical scheduling without resource sharing, the lower value of Π_s gives better results (needs less utilization). However, in the presence of global resources sharing, the selection of the subsystem period depends also on the maximum value of x_i in the subsystem.

8.6.4 Multiple critical sections

We compare the case when a task i accesses multiple critical sections (MCS) with the case when a task j accesses a single critical section (SCS) within duration $x_j = \sum_{k=1}^{o} x_{i,k}$ according to the demand bound function calculations in Eq. (8.4). The following shows the effect of accessing MCS by a task on itself and on higher and lower priority tasks;

- Self blocking, Eq. (8.5) shows that both accessing MCS and SCS by a task gives the same result.
- Higher priority task, the effect from higher priority task accessing MCS or SCS can be evaluated by Eq. (8.6). I_H will be the same for both cases also.
- Lower priority task, Eq. (8.7) shows that I_L for MCS is less than SCS case because in MCS the maximum of $x_{i,j}$ will be less than x_i for SCS.

We can conclude that the required subsystem utilization for MCS case will be always less than or equal to the case of SCS having $x_j = \sum_{k=1}^{o} x_{i,k}$, which means that our proposed protocol is scalable in terms of the number of critical sections.

8.6.5 Independent abstraction

In this paper, we have proposed a synchronization protocol that supports independent abstraction of a subsystem, particularly, for open systems. Independent abstraction is desirable since it allows subsystems to be developed and validated without knowledge about temporal behavior of other subsystems. In some cases, subsystems can be abstracted *dependently* of others when some necessary information about all the other subsystems is available. However, dependent abstraction has a clear limitation to open systems where such information is assumed to be unavailable. In addition, dependent abstraction is not good for dynamically changing systems, since it may be no longer valid when a new subsystem is added. Despite of the advantages of independent abstraction vs. dependent abstraction, however, one may wonder what costs look like in using independent abstraction in comparison with using dependent abstraction. In this section, we discuss this issue in terms of resource efficiency (subsystem resource utilization).

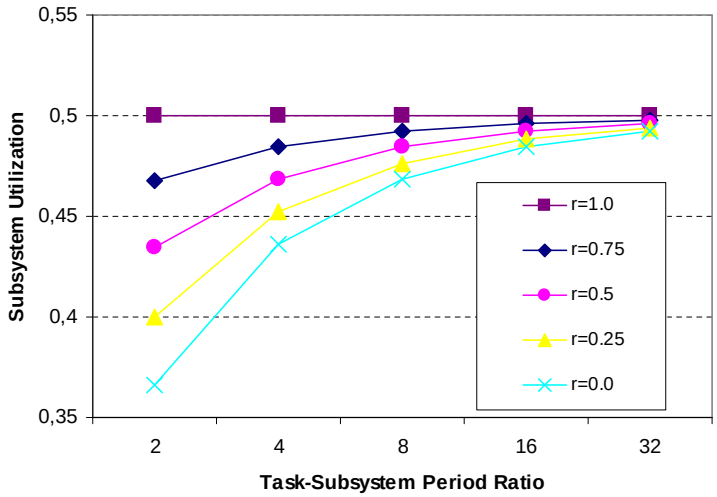


Figure 8.7: Comparison between independent and dependent abstractions in terms of subsystem utilization.

One of the key differences between independent and dependent abstractions is how to model a resource supply provided to a subsystem, more specifically, how to characterize the longest *blackout duration* during which no resource supply is provided. Under independent abstraction, the longest blackout duration is assumed to be the worst-case (maximum) one. Whereas, it can be exactly identified by some techniques [6, 26] under dependent abstraction. This difference inherently yields different subsystem resource utilizations, as illustrated in Figure 8.7. Before explaining this figure, we need to establish some notions and explain how to obtain this figure.

We first extend the periodic resource model $\Gamma(\Pi, \Theta)$ by introducing an additional parameter, *blackout duration ratio (r)*. We define r as follows. Let L_{min} and L_{max} denote the minimum and maximum possible blackout duration, and

$$L_{min} = \Pi - \Theta \text{ and } L_{max} = 2(\Pi - \Theta).$$

When exactly computed, the longest blackout duration can then be represented as $r \cdot (L_{max} - L_{min}) + L_{min}$. We generalize the supply bound function of Eq.

(8.2) with the blackout duration ratio r as follows:

$$\mathtt{sbf}_{\Gamma}(t) = \begin{cases} t-(k+1)(\Pi-\Theta) & \text{if } t \in [k\Pi-\Theta \\ & +r(\Pi-\Theta), \\ & k\Pi+r(\Pi-\Theta)], \\ (k-1)\Theta & \text{otherwise}, \end{cases} \tag{8.12}$$

where $k = \max\Big(\big\lceil\big(t-(\Pi-\Theta)\big)/\Pi\big\rceil, 1\Big)$.

We here explain the notion of *task-subsystem period ratio*, which is the x-axis of the figure. Suppose a periodic resource model $\Gamma_1(\Pi_1, \Theta_1, r_1)$ is an abstraction that guarantees the schedulability of a subsystem S. According to Eq. (8.3), there then exists a time instant t_i^*, where $0 < t_i^* \leq T_i$, for each task τ_i within the subsystem S such that

$$\forall \tau_i, \;\; \mathtt{dbf}_{\mathtt{FP}}(i, t_i^*) \leq \mathtt{sbf}_{\Gamma_1}(t_i^*). \tag{8.13}$$

In fact, given the values of subsystem period Π and blackout duration ratio r, we can find a smallest value of Θ, denoted as Θ_i^*, that can satisfy Eq. (8.13) at t_i^* for each task τ_i. The value of budget Θ is then finally determined as the maximum value among all Θ_i^*. This way makes sure that Θ is large enough to guarantee the timing requirements of all tasks. Let T^* denote a time instant t_k^* such that Θ_k^* is the maximum among the ones. We can see that $T^* \in [T_{min}, T_{max}]$, where T_{min} and T_{max} denote the minimum and maximum task periods within subsystem, respectively. We define the *task-subsystem period ratio* as T^*/Π.

Given a periodic abstraction Γ_1 of the subsystem S, another periodic resource model $\Gamma_2(\Pi_2, \Theta_2, r_2)$ can be also an abstraction of S, if

$$\forall \tau_i, \;\; \mathtt{sbf}_{\Gamma_1}(t_i^*) \leq \mathtt{sbf}_{\Gamma_2}(t_i^*), \tag{8.14}$$

since Eq. (8.3) can be satisfied with S and Γ_2 as well. More specifically, $\Gamma_2(\Pi_2, \Theta_2, r_2)$ can be an abstraction of S, if

$$\mathtt{sbf}_{\Gamma_1}(T^*) \leq \mathtt{sbf}_{\Gamma_2}(T^*). \tag{8.15}$$

That is, given Γ_1 and the values of Π_2 and r_2, we can find the minimum value of Θ_2 that satisfies Eq. (8.15).

Figure 8.7 shows subsystem utilizations of periodic abstractions under different values of blackout duration ratio r, when they have the same subsystem period in abstracting the same subsystem. In general, it shows that dependent abstraction, which can exactly identify the value of r, would pro-

duce more resource-efficient subsystem abstractions. Specifically, for example, when $r = 0$, i.e., when the subsystem has the highest priority under fixed-priority global scheduling, a subsystem can be abstracted with 15% less subsystem utilization than in the case of independent abstraction ($r = 1$). The figure also shows that differences in subsystem utilization generally decrease when the task-subsystem period ratio increases and/or the blackout duration ratio increases. For example, when $r = 0.5$, i.e., when the system has a moderately high utilization and subsystems have medium or low priorities under fixed-priority global scheduling or subsystems are scheduled under global EDF scheduling, differences are shown to be smaller than 8%.

8.7 Conclusion

In this paper we have presented the novel Subsystem Integration and Resource Allocation Policy (SIRAP), which provides temporal isolation between subsystems that share logical resources. Each subsystem can be developed, tested and analyzed without knowledge of the temporal behaviour of other subsystems. Hence, integration of subsystems, in later phases of product development, will be smooth and seamless.

We have formally proven key features of SIRAP such as bounds on delays for accessing shared resources. Further, we have provided schedulability analysis for tasks executing in the subsystems; allowing for use of hard real-time application within the SIRAP framework.

Naturally, the flexibility and predictability offered by SIRAP comes with some costs in terms of overhead. We have evaluated this overhead through a comprehensive simulation study. From the study we can see that the subsystem period should be chosen as much smaller than the smallest task period in a subsystem and take into account the maximum value of h_i in the subsystem to prevent having high subsystem utilization. Future work includes investigating the effect of context switch overhead on subsystem utilization together with the subsystem period and the maximum value of h_i.

Bibliography

[1] D. Andrews, I. Bate, T. Nolte, C. M. Otero Pérez, and S. M. Petters. Impact of embedded systems evolution on RTOS use and design. In Giuseppe Lipari, editor, *Proceedings of the* 1^{st} *International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'05) in conjunction with the* 17^{th} *Euromicro International Conference on Real-Time Systems (ECRTS'05)*, pages 13–19, Palma de Mallorca, Balearic Islands, Spain, July 2005.

[2] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri. From a federated to an integrated architecture for dependable embedded real-time systems. Technical Report 22, Technische Universität at Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[3] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the* 18^{th} *IEEE International Real-Time Systems Symposium (RTSS'97)*, pages 308–319, San Francisco, CA, USA, December 1997. IEEE Computer Society.

[4] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. In *Component-Based Software Engineering*, volume LNCS-3054/2004, pages 253–266. Springer Berlin / Heidelberg, May 2005.

[5] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT '04)*, pages 95–103, Pisa, Italy, September 2004.

[6] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the* 26^{th} *IEEE International Real-Time Systems*

Symposium (RTSS'05), pages 389–398, Miami Beach, FL, USA, December 2005.

[7] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* 23^{th} *IEEE International Real-Time Systems Symposium (RTSS'02)*, pages 26–35, Austin, TX, USA, December 2002.

[8] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the* 20^{th} *IEEE International Real-Time Systems Symposium (RTSS'99)*, pages 256–267, Phoenix, AZ, USA, December 1999. IEEE Computer Society.

[9] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, Washington DC, USA, May-June 2000. IEEE Computer Society.

[10] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 151–158, Porto, Portugal, July 2003. IEEE Computer Society.

[11] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *Proceedings of the* 26^{th} *IEEE International Real-Time Systems Symposium(RTSS'05)*, pages 99–110, Washington, DC, USA, December 2005. IEEE Computer Society.

[12] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarhical fixed-priority scheduling. In *Proceedings of the* 14^{th} *Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 152–160, Vienna, Austria, June 2002. IEEE Computer Society.

[13] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* 24^{th} *IEEE International Real-Time Systems Symposium(RTSS'03)*, pages 2–13, Cancun, Mexico, December 2003.

[14] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the* 25^{th} *IEEE International Real-Time Systems Symposium(RTSS'04)*, pages 57–67, Lisbon, Portugal, December 2004. IEEE Computer Society.

[15] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard-real-time environments. In *Proceedings of the* 21^{th} *IEEE International Real-Time Systems Symposium(RTSS'00)*, pages 217–226, Orlando, FL, USA, December 2000.

[16] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium(RTAS)*, pages 75–84, Taipei, Taiwan ROC, May 2001.

[17] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation IECON87*, pages 909–916, Cambridge, MA, USA, November 1987.

[18] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the* 9^{th} *IEEE International Real-Time Systems Symposium (RTSS'88)*, pages 259–269, Huntsville, AL, USA, December 1988. IEEE Computer Society.

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

[20] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *Proceedings of the 15th International Workshop on Parallel and Distributed Real-Time Systems(WPDRTS)*, pages 1–8, Long Beach, CA, USA, March 2007.

[21] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in EDF-scheduled systems. In *Proceedings of the* 13^th *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, Bellevue, WA, USA, 2007.

[22] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the* 27^{th} *IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, Rio de Janeiro, Brazil, December 2006.

[23] Marco Caccamo and Lui Sha. Aperiodic servers with resource constraints. In *IEEE Real-Time Systems Symposium*, pages 161–170, 2001.

[24] Philip Holman and James H. Anderson. Locking under pfair scheduling. *ACM Trans. Comput. Syst.*, 24(2):140–174, 2006.

[25] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, 29(5):390–395, October 1986.

[26] Reinder J. Bril, Wim F. J. Verhaegh, and Clemens C. Wust. A cognac-glass algorithm for conditionally guaranteed budgets. In *RTSS '06: Proceedings of the* 27^th *IEEE International Real-Time Systems Symposium(RTSS'06)*, pages 388–400, Rio de Janeiro, Brazil, December 2006.

Chapter 9

Paper C: Scheduling of Semi-Independent Real-Time Components: Overrun Methods and Resource Holding Times

Moris Behnam, Insik Shin, Thomas Nolte, Mikael Nolin

In Proceedings of the 13^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08), IEEE Industrial Electronics Society, Hamburg, Germany, September, 2008.

Abstract

The Hierarchical Scheduling Framework (HSF) has been introduced as a design-time framework enabling compositional schedulability analysis of embedded software systems with real-time properties. In this paper a system consists of a number of semi-independent components called subsystems. Subsystems are developed independently and later integrated to form a system. To support this design process, our proposed methods allow non-intrusive configuration and tuning of subsystem timing-behaviour via subsystem interfaces for selecting scheduling parameters.

This paper considers two methods to handle overruns due to resource sharing between subsystems in the HSF. We present the scheduling algorithms for overruns and their associated schedulability analysis, together with analysis that shows under what circumstances one or the other overrun method is preferred. Furthermore, we show how to calculate resource-holding times within our framework.

9.1 Introduction

The Hierarchical Scheduling Framework (HSF) has been introduced to support hierarchical resource sharing among applications under different scheduling services. The hierarchical scheduling framework can be generally represented as a tree of nodes, where each node represents an application with its own scheduler for scheduling internal workloads (e.g., threads), and resources are allocated from a parent node to its children nodes.

The HSF provides means for decomposing a complex system into well-defined parts. In essence, the HSF provides a mechanism for timing-predictable *composition* of course-grained components or *subsystems*. In the HSF a subsystem provides an introspective *interface* that specifies the timing properties of the subsystem precisely [1]. This means that subsystems can be independently developed and tested, and later assembled without introducing unwanted temporal behaviour. Also, the HSF facilitates *reusability* of subsystems in timing-critical and resource constrained environments, since the well defined interfaces characterize their computational requirements.

Earlier efforts have been made in supporting compositional subsystem integration in the HSFs, preserving the independently analyzed schedulability of individual subsystems. One of the common assumptions shared by earlier studies is that subsystems are independent. This paper relaxes this assumption by addressing the challenge of enabling efficient compositional integration for independently developed *semi-independent* subsystems interacting through sharing of logical resources. Here, semi-independence means that subsystems are allowed to synchronize by the sharing of logical resources.

To enable sharing of logical resources in HSFs, Davis and Burns proposed the *overrun* mechanism that allows the subsystem to overrun (its budget) to complete the execution of the critical section [2]. The study provided schedulability analysis for this mechanism; however, it does not allow independent analysis of individual subsystems. Hence, this schedulability analysis does not support composability of subsystems. For Davis and Burns' overrun mechanism, we have presented schedulability analysis supporting composability in [3] and in addition, we also presented another overrun mechanism (enhanced overrun mechanism) that potentially increases schedulability within a subsystem by providing CPU allocations more efficiently.

The main contributions of this paper are twofold. The first contribution of this paper is the comparison analysis showing under what circumstances one [2] or the other overrun method [3] is preferred. The second contribution of this paper is the presentation of, for the periodic virtual processor model that

is used throughout the paper, how to calculate bounds on the *resource holding time*. The resource holding time represents the time during which a subsystem may lock a shared resource, and it plays a key role in defining the interface of a subsystem in this paper.

The outline of the paper is as follows: Section 9.2 presents related work, while Section 9.3 presents our system model. In Section 9.4 we present the schedulability analysis for the model. Section 9.5 presents the two overrun mechanisms and Section 9.6 presents their analytical comparison. In Section 9.7 we show how to calculate the resource holding times, and finally, Section 9.8 concludes.

9.2 Related work

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

9.2.1 Hierarchical scheduling

The HSF for real-time systems, originating in open systems [4] in the late 1990's, has been receiving an increasing research attention. Since Deng and Liu [4] introduced a two-level HSF, its schedulability has been analyzed under fixed-priority global scheduling [5] and under EDF-based global scheduling [6, 7]. Mok *et al.* [8] proposed the bounded-delay resource model so as to achieve a clean separation in a multi-level HSF, and schedulability analysis techniques [9, 10] have been introduced for this resource model. In addition, Shin and Lee [1, 11] introduced another periodic resource model (to characterize the periodic resource allocation behaviour), and many studies have been proposed on schedulability analysis with this resource model under fixed-priority scheduling [12, 13, 14] and under EDF scheduling [1]. More recently, Easwaran *et al.* [15] introduced Explicit Deadline Periodic (EDP) resource model. However, a common assumption shared by all the studies in this paragraph is that tasks are required to be independent.

9.2.2 Resource sharing

In many real systems, tasks are semi-independent, interacting with each other through mutually exclusive resource sharing. Many protocols have been introduced to address the priority inversion problem for semi-independent tasks,

including the Priority Inheritance Protocol (PIP) [16], the Priority Ceiling Protocol (PCP) [17], and Stack Resource Policy (SRP) [18]. Recently, Fisher *et al.* addressed the problem of minimizing the resource holding time [19] under SRP. There have been studies on extending SRP for HSFs, for sharing of logical resources within a subsystem [20, 5] and across subsystems [2, 21, 22]. Davis and Burns [2] proposed the Hierarchical Stack Resource Policy (HSRP) supporting sharing of logical resources on the basis of an overrun mechanism. Behnam *et al.* [21] proposed the Subsystem Integration and Resource Allocation Policy (SIRAP) protocol that supports subsystem integration in the presence of shared logical resources, on the basis of skipping. Fisher *et al.* [22] proposed the BROE server that extends the Constant Bandwidth Server (CBS) [23] in order to handle sharing of logical resources in a HSF. Lipari *et al.* proposed the BandWidth Inheritance protocol (BWI) [24] which extends the resource reservation framework to systems where tasks can share resources. The BWI approach is based on using the CBS algorithm and a technique that is derived from the Priority Inheritance Protocol (PIP). Particularly, BWI is suitable for systems where the execution time of a task inside a critical section can not be evaluated.

9.3 System model and background

9.3.1 Resource sharing in the HSF

The Hierarchical Scheduling Framework (HSF) has been introduced to support CPU time sharing among applications (subsystems) under different scheduling policies. In this paper, we consider a two level-hierarchical scheduling framework which works as follows: a global (system-level) scheduler allocates CPU time to subsystems, and a local (subsystem-level) scheduler subsequently allocates CPU time to its internal tasks.

Having such a HSF also allows for the sharing of logical resources among tasks in a mutually exclusive manner (see Figure 9.1). Specifically, tasks can share *local* logical resources within a subsystem as well as *global* logical resources across (in-between) subsystems. However, note that this paper focuses around mechanisms for sharing of global logical resources in a HSF while local logical resources easily can be supported by traditional synchronization protocols such as SRP (see, e.g., [20, 2, 5]).

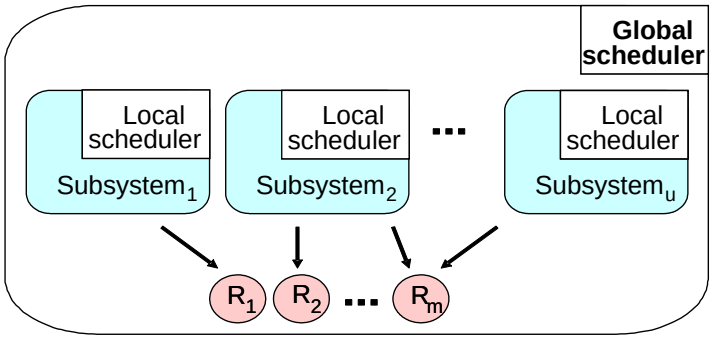


Figure 9.1: Two-level HSF with resource sharing.

9.3.2 Virtual processor models

The notion of real-time virtual processor (resource) model was first introduced by Mok *et al.* [8] to characterize the CPU allocations that a parent node provides to a child node in a HSF. The *CPU supply* of a virtual processor model refers to the amounts of CPU allocations that the virtual processor model can provide. The *supply bound function* of a virtual processor model calculates its minimum possible CPU supply for any given time interval of length t.

The periodic virtual processor model $\Gamma(P, Q)$ was proposed by Shin and Lee [1] to characterize periodic resource allocations, where P is a period ($P > 0$) and Q is a periodic allocation time ($0 < Q \leq P$). The capacity U_Γ of a periodic virtual processor model $\Gamma(P, Q)$ is defined as Q/P.

The *supply bound function* $\mathtt{sbf}_\Gamma(t)$ of the periodic virtual processor model $\Gamma(P, Q)$ was given in [1] to compute the minimum resource supply during an interval of length t. Further, in this paper, we rephrase it with an additional parameter of BD, where BD represents its longest possible *blackout duration* during which the periodic virtual processor model may provide no resource allocation at all.

$$\mathtt{sbf}_\Gamma(t, \mathsf{BD}) = \begin{cases} t - (k-1)(P-Q) - BD & \text{if } t \in W^{(k)} \\ (k-1)Q & \text{otherwise,} \end{cases} \tag{9.1}$$

where $k = \max\left(\lceil (t + (P-Q) - BD)/P \rceil, 1\right)$ and $W^{(k)}$ denotes an interval $[(k-1)P+\mathsf{BD}, (k-1)P+\mathsf{BD}+Q]$. Here, we first note that the original $\mathtt{sbf}_\Gamma(t)$

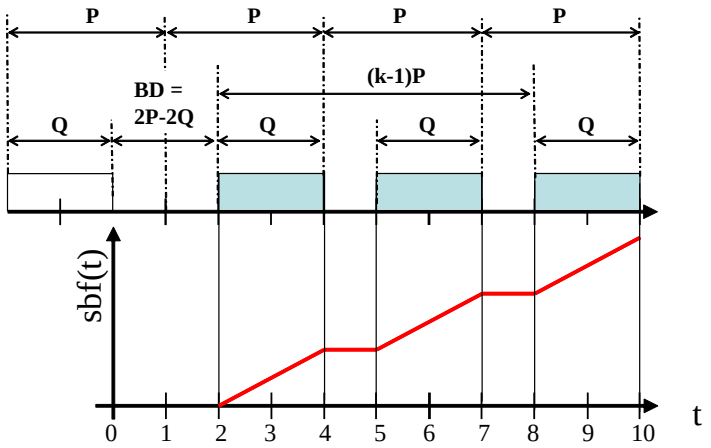


Figure 9.2: The supply bound function of a periodic virtual processor model $\Gamma(3, 2)$.

in [1] is equivalent to $\mathtt{sbf}_\Gamma(t, \mathsf{BD})$ when $\mathsf{BD} = 2(P-Q)$. We also note that an interval of length t may not begin synchronously with the beginning of period P; as shown in Figure 9.2, the interval of length t can start in the middle of the period of a periodic virtual processor model $\Gamma(P, Q)$. Figure 9.2 illustrates the supply bound function $\mathtt{sbf}_\Gamma(t)$ of the periodic virtual processor model.

9.3.3 Stack resource policy (SRP)

To be able to use SRP [18] in the HSF, its associated terms are extended as follows:

- *Preemption level*. Each task τ_i has a preemption level equal to $\pi_i = 1/D_i$, where D_i is the relative deadline of the task. Similarly, each subsystem S_s has an associated preemption level equal to $\Pi_s = 1/P_s$, where P_s is the subsystem's per-period deadline.

- *Resource ceiling*. Each globally shared resource R_j is associated with two types of resource ceilings; one *internal* resource ceiling for local scheduling $rc_j = \max\{\pi_i | \tau_i \text{ accesses } R_j\}$ and one *external* resource ceiling for global scheduling.

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

Following the rules of SRP, a job J_i that is generated by a task τ_i can preempt the currently executing job J_k within a subsystem only if J_i has a priority higher than that of job J_k and, at the same time, the preemption level of τ_i is greater than the current subsystem ceiling. A similar reasoning is made for subsystems from a global scheduling point of view.

9.3.4 System model

In this paper a periodic task model $\tau_i(T_i, C_i, D_i, \{c_{i,j}\})$ is considered, where T_i, C_i and D_i represent the task's period, worst-case execution time (WCET) and relative deadline, respectively, where $D_i \leq T_i$, and $\{c_{i,j}\}$ is the set of WCETs within critical sections associated with task τ_i. Each element $c_{i,j}$ in $\{c_{i,j}\}$ represents the WCET of the task τ_i inside a critical section of the global shared resource R_j.

Looking at a shared resource R_j, the *resource holding time* $h_{j,i}$ of a task τ_i is defined as the time given by the task's maximum execution time inside a critical section plus the interference (inside the critical section) of higher priority tasks having preemption level greater than the internal ceiling of the locked resource.

A subsystem $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the whole system of subsystems, is characterized by a task set $\mathcal{T}_s$ and a set of internal resource ceilings $\mathcal{RC}_s$ inherent from internal tasks using the globally shared resources. Each subsystem S_s is assumed to have an EDF local scheduler, and the subsystems are scheduled according to EDF on a global level. The collective resource requirements by each subsystem S_s is characterised by its *interface* (the subsystem interface) defined as (P_s, Q_s, H_s), where P_s is the subsystem's period, Q_s is it's execution requirement budget, and H_s is the subsystem's maximum resource holding time, i.e., $H_s = \max\{h_{j,i} | \tau_i \in \mathcal{T}_s \text{ accesses } R_j\}$.

9.4 Schedulability analysis

This section presents the schedulability analysis of the HSF, starting with local schedulability analysis needed to calculate subsystem interfaces, and finally,

global schedulability analysis. The analysis presented assumes that SRP is used for synchronization on the local (within subsystems) level.

9.4.1 Local schedulability analysis

Let $\mathtt{dbf}_{\mathsf{EDF}}(i,t)$ denote the demand bound function of a task τ_i under EDF scheduling [25], i.e.,

$$\mathtt{dbf}_{\mathsf{EDF}}(i,t) = \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i. \tag{9.2}$$

The local schedulability condition under EDF scheduling is then (by combining the results of [26] and [1])

$$\forall t > 0 \quad \sum_{i=1}^{n} \mathtt{dbf}_{\mathsf{EDF}}(i,t) + b(t) \leq \mathtt{sbf}(t), \tag{9.3}$$

where $b(t)$ is the blocking function [26] that represents the longest blocking time during which a job J_i with $D_i \leq t$ may be blocked by a job J_k with $D_k > t$ when both jobs access the same resource.

9.4.2 Subsystem interface calculation

Given a subsystem 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 Q_s that satisfies Eq. (9.3). Hence, $Q_s = \mathsf{calculateBudget}(S_s, P_s, \mathcal{RC}_s)$. The function is similar to the one presented in [1], however, due to space limitations, its details are left out of this paper.

9.4.3 Global schedulability analysis

Following Theorem 1 of [26], global schedulability analysis under EDF scheduling is given using the system load bound function $\mathtt{LBF}(t)$ as follows:

$$\forall t > 0 \quad \mathtt{LBF}(t) = B(t) + \sum_{S_s \in \mathcal{S}} \mathtt{DBF}_s(t) \leq t, \tag{9.4}$$

where

$$\mathtt{DBF}_s(t) = \left\lfloor \frac{t}{P_s} \right\rfloor \cdot Q_s, \tag{9.5}$$

and the system-level blocking function $B(t)$ represents the maximum blocking time during which a subsystem S_s may be blocked by another subsystem S_k, where $P_s \leq t$ and $P_k > t$. $B(t)$ is defined as

$$B(t) = \max\{H_k \mid P_k > t\}. \tag{9.6}$$

9.5 Overrun mechanisms

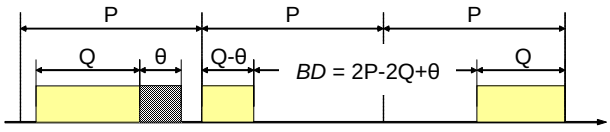
This section explains two overrun mechanisms that can be used to handle budget expiry during a critical section in the HSF. Consider a global scheduler that schedules subsystems according to their periodic interfaces (P_s, Q_s, H_s). The subsystem budget Q_s is said to *expire* at the point when one or more internal (to the subsystem) tasks have executed a total of Q_s time units within the subsystem period P_s. Once the budget is expired, no new task within the same subsystem can initiate its 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 may cause a problem if it happens while a job J_i of a subsystem S_s is executing within a critical section of a global shared resource R_j. If another job J_k, belonging to another subsystem, is waiting for the same resource R_j, this job must wait until S_s is replenished again so J_i can continue to execute and finally release the lock on resource R_j. This waiting time exposed to J_k can be potentially very long, causing J_k to miss its deadline.

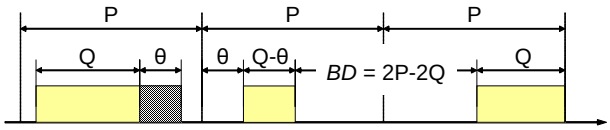
In this paper, we consider an overrun mechanism as follows; when the budget of subsystem S_s expires and S_s has a job J_i that is still locking a globally shared resource, job J_i continues its execution until it releases the locked resource. The extra time that J_i needs to execute after the budget of S_s expires is denoted as *overrun time* θ. The maximum θ occurs when J_i locks a resource that gives the longest resource holding time just before the budget of S_s expires. Here, we consider the payback overrun mechanism [2]. Whenever overrun happens, the subsystem S_s pays back θ in its next execution instant, i.e., the subsystem budget Q_s will be decreased by θ for the subsystem's execution instant following the overrun (note that only the instant following the overrun is affected). Hereinafter, we call this payback overrun mechanism *basic overrun*.

9.5.1 Basic overrun

Davis *et al.* [2] presented schedulability analysis for basic overrun, however, it is not suitable for open environments [4] as it requires detailed information of



(a) basic overrun mechanism



(b) enhanced overrun mechanism

Figure 9.3: Basic and enhanced overrun mechanisms.

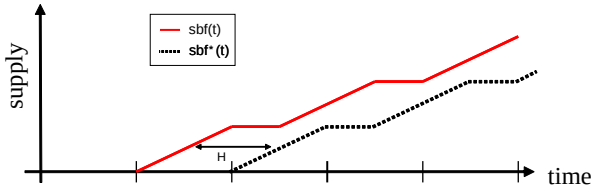


Figure 9.4: Comparing $\mathtt{sbf}(\mathtt{t})$ with $\mathtt{sbf}^{\circ}(\mathtt{t})$.

all tasks in the system in order to calculate global schedulability. This section discusses how to extend the existing schedulability analysis for basic overrun, making it suitable for open environments.

Independent analysis with basic overrun

The supply bound function in [1] was developed under the assumption that the greatest blackout duration is $2(P-Q)$. Basic overrun cannot employ this existing supply bound function for schedulability analysis because its greatest Blackout Duration (BD) is $2(P-Q)+H$ (as shown in Figure 9.3a). Taking this into account, below is the presentation of a modified supply bound function $\mathtt{sbf}^{\circ}_{\Gamma}(t)$, that can be used with basic overrun (using Eq. (9.1)), as follows:

$$\mathtt{sbf}^{\circ}_{\Gamma}(t) = \mathtt{sbf}_{\Gamma}(t, \mathsf{BD}^{\circ}), \text{ where } \mathsf{BD}^{\circ} = 2(P - Q) + H. \tag{9.7}$$

The existing schedulability conditions of Eq. (9.3) can then be extended by substituting $\mathtt{sbf}_{\Gamma}(t)$ with $\mathtt{sbf}^{\circ}_{\Gamma}(t)$.

Global analysis with basic overrun

We first discuss how to extend the demand bound function of a subsystem with the basic overrun mechanism. Looking at basic overrun with payback in a subsystem S_s, the maximum contribution on $\mathtt{DBF}_s(t)$ is H_s. When S_s overruns with its maximum, which is H_s, the subsystem's resource demand within the subsystem period P_s will be increased to $Q_s + H_s$. Following this, the budget of the next period will be decreased to $Q_s - H_s$ due to the payback mechanism. Then, suppose that the subsystem overruns again. Now, during the next subsystem period, the subsystem's resource demand will be $Q_s - H_s + H_s = Q_s$. Here, one can easily see that the subsystem's resource demand will be at most $kQ_s + H_s$ during k subsystem periods. Hence, the demand bound function $\mathtt{DBF}^{\circ}_s(t)$ of a subsystem S_s with the basic overrun mechanism is

$$\mathtt{DBF}^{\circ}_s(t) = \mathtt{DBF}_s(t) + O_s(t), \tag{9.8}$$

where

$$O_s(t) = \begin{cases} H_s & \text{if } t \geq P_s, \\ 0 & \text{otherwise.} \end{cases} \tag{9.9}$$

The schedulability condition of Eq. (9.4) can then be extended by substituting $\mathtt{DBF}_s(t)$ with $\mathtt{DBF}^{\circ}_s(t)$.

9.5.2 Enhanced overrun

As seen in Section 9.5.1, the basic overrun mechanism works with a modified supply bound function $\mathtt{sbf}^{\circ}(t)$ that is less efficient in terms of CPU resource usage compared with the original $\mathtt{sbf(t)}$, as illustrated in Figure 9.4. Now we propose an enhanced overrun mechanism that makes it possible to use $\mathtt{sbf(t)}$ and overrun to improve the efficiency of CPU resource utilization.

The enhanced overrun mechanism is based on imposing an offset (delaying the budget replenishment of subsystem) equal to the amount of an overrun θ_s to the execution instant that follows a subsystem overrun. As shown in Figure 9.3b, the execution of the subsystem will be delayed by θ_s after a new

period followed by overrun even if that subsystem has the highest priority at that time. By this the maximum BD will be decreased to $2(P-Q)$ compared with basic overrun shown in Figure 9.3a and therefore it is possible to use the same supply bound function presented in Section 9.3.2. One of the important features that the enhanced overrun mechanism provides is that it moves the effect of overrun from the local to the global schedulability analysis, so the subsystem development will not depend on if there is an overrun mechanism or not. This feature is very important in an open environment. We can then use the existing local EDF schedulability condition of Eq. (9.3) without any modification.

Global analysis with enhanced overrun

The effect of overrun is now moved to global schedulability analysis in the enhanced overrun mechanism. Here, we present a demand bound function $\texttt{DBF}^*_s(t)$ of a subsystem S_s that upper-bounds the demand requested by S_s under the enhanced overrun mechanism. Now, $\texttt{DBF}^*_s(t)$ includes the offset $\theta_s = H_s$ as follows:

$$\texttt{DBF}^*_s(t) = \left\lfloor \frac{t + H_s}{P_s} \right\rfloor \cdot Q^*_s + O^*_s(t), \tag{9.10}$$

where

$$O^*_s(t) = \begin{cases} H_s & \text{if } t \geq P_s - H_s, \\ 0 & \text{otherwise.} \end{cases} \tag{9.11}$$

The schedulability condition of Eq. (9.4) can then be extended by substituting $\texttt{DBF}_s(t)$ with $\texttt{DBF}^*_s(t)$.

9.6 Comparison between basic and enhanced overrun mechanisms

In this section, we will compare the efficiency of the two overrun mechanisms. First, we will show the effect of using each of them locally, i.e., on a subsystem level. Then, we will show their effect globally, i.e., on a system level.

9.6.1 Subsystem-level comparison

The following lemma shows that the minimum required subsystem budget when using enhanced overrun will be lower than or equal to the minimum required budget when using basic overrun.

Lemma 4. *Assuming that the minimum required budget to schedule all tasks in a subsystem S_s using basic overrun is Q_s°, and that the corresponding budget using enhanced overrun is Q_s^*, then $Q_s^* \leq Q_s^\circ$.*

Proof. A subsystem S_s is exactly schedulable iff in addition to Eq. (9.3), $\sum_i^n \texttt{dbf}_{\texttt{EDF}}(i,t) + b(t) = \texttt{sbf}(t)$ for $\exists t$ s.t. $min_i^n D_i \leq t \leq LCM_{S_s} + max_i^n D_i$ (see theorem 2.2 in [15]). This means that if the budget Q_s is the minimum required to guarantee the schedulability of tasks in S_s, then there is a set of times t^e at which $\sum_i^n \texttt{dbf}_{\texttt{EDF}}(i,t) + b(t) = \texttt{sbf}(t)$. Without loss of generality, we assume that t^e includes one element. If we use same subsystem budget Q_s for both basic and enhanced overrun then

$$\texttt{sbf}^\circ(t) = \texttt{sbf}(t - H_s) \tag{9.12}$$

where $\texttt{sbf}(t)$ is used with enhanced overrun and the shift in time "$-H_s$" comes from the difference in BD when using enhanced and basic overrun. From Eq. (9.1) and Eq. (9.12), we have two cases:

case 1: $\texttt{sbf}^\circ(t) = \texttt{sbf}(t)$ for $t \in [kP_s - Q_s + H_s, (k+1)P_s - 2Q_s]$ where k is an integer number $k > 1$.

case 2: $\texttt{sbf}^\circ(t) < \texttt{sbf}(t)$ for t out of the range in case 1.

If $t^e \in [kP_s - Q_s + H_s, (k+1)P_s - 2Q_s]$ then $\texttt{sbf}^\circ(t^e) = \texttt{sbf}(t^e)$. And then $\sum_i^n \texttt{dbf}_{\texttt{EDF}}(i,t^e) + b(t^e) = \texttt{sbf}^\circ(t^e)$, which means that Q_s may be enough to schedule all tasks in a subsystem S_s using basic overrun, so $Q_s^* = Q_s^\circ$ at time $t = t^e$. However, Eq. (9.3) must be checked if it holds for all other times t, to be sure that the subsystem S_s is still scheudlable.

If t^e is not in the range given for case 1, then $\texttt{sbf}^\circ(t^e) < \texttt{sbf}(t^e)$. And then $\texttt{sbf}^\circ(t^e) < \sum_i^n \texttt{dbf}_{\texttt{EDF}}(i,t^e) + b(t^e)$ which means that the budget Q_s will not satisfy the condition in Eq. (9.3) using basic overrun and we should provide higher budget. In this case $Q_s^* < Q_s^\circ$.

□

9.6.2 System-level comparison

As shown in the previous section, the minimum required budget when using enhanced overrun is lower than or equal to the minimum budget when using basic overrun. However, at system level, it is not easy to see which of the two approaches that will require minimum overall system CPU resources.

Let us define *system load* as a quantitative measure to represent the minimum amount of CPU allocations necessary to guarantee the schedulability of the system S. Then, we will investigate the impact of each overrun mechanism on the system load, respectively. The system load $\mathsf{load}_{\mathsf{sys}}$ is computed as follows:

$$\mathsf{load}_{\mathsf{sys}} = \max_{t} \frac{\mathtt{LBF}(t)}{t} \ . \tag{9.13}$$

Note that $\alpha = \mathsf{load}_{\mathsf{sys}}$ is the smallest fraction of the CPU that is required to schedule all the subsystems in the system S (satisfying Eq. (9.4)) assuming that the resource supply function (at system level) is αt.

Looking at Eq. (9.13), we can decrease $\mathsf{load}_{\mathsf{sys}}$ by lowering $\mathtt{LBF}(t)$. Enhanced overrun will make $\mathtt{LBF}(t)$ lower compared with basic overrun since $\mathtt{LBF}(t)$ depends on the subsystem budget. However, because of the offset imposed in the global scheduling when using enhanced overrun, the resource demand should be provide earlier (see Eq. (9.10)). Comparing Eq. (9.8), which computes $\mathtt{DBF}^{\circ}_{s}(t)$ using basic overrun, and Eq. (9.10), which computes $\mathtt{DBF}^{*}_{s}(t)$ using enhanced overrun, we can see that the $\mathtt{DBF}^{\circ}_{s}(t)$ is changed when $t = a \times P_s$ for $s = 1, .., m$ where m is the number of subsystems and a is an integer number such that $a > 0$. While $\mathtt{DBF}^{*}_{s}(t)$ is changed when $t = a \times P_s - H_s$ for $s = 1, .., m$. Note that $\mathsf{load}_{\mathsf{sys}}$ will be evaluated at times when the demand bound function changes, so it is not possible to decide if using enhanced overrun will require less $\mathsf{load}_{\mathsf{sys}}$ compared with basic overrun. However, in some special cases depending on the parameters of each subsystem, we can decide which of the two overrun mechanisms that will produce lower $\mathsf{load}_{\mathsf{sys}}$. For a subsystem S_s, we have two cases:

1. If $Q^{\circ}_{s}/P_S < Q^{*}_{s}/(P_s - H_s)$, then the $\mathsf{load}_{\mathsf{sys}}$ using enhanced overrun will be greater than or equal to the $\mathsf{load}_{\mathsf{sys}}$ with basic overrun. The reason for this is that $max(\mathtt{DBF}^{\circ}_{s}(t_{ba})/t_{ba}) < max(\mathtt{DBF}^{*}_{s}(t_{en})/t_{en})$ where $t_{ba} = a \times P_s$ is the time when $\mathtt{DBF}^{\circ}_{s}(t)$ is changed, and $t_{en} = a \times P_s - H_s$ is the time when $\mathtt{DBF}^{*}_{s}(t)$ is changed.

2. Otherwise, it will depend on the parameters of the other subsystems to

decide which of the two overrun mechanisms that will be better in terms of $\mathsf{load}_{\mathsf{sys}}$.

We will explain the previous two cases by the following three examples:

Example 1: For the first case ($Q_s^\circ/P_S < Q_s^*/(P_s - H_s)$), suppose that a system S consists of two subsystems S_1 with parameters $P_1 = 50, Q_1^\circ = 10, Q_1^* = 10, H_1 = 4$ and S_2 with parameters $P_2 = 150, Q_2^\circ = 15, Q_2^* = 14.9, H_2 = 8$. Then using enhanced overrun $\mathsf{load}_{\mathsf{sys}} = 0,478$ and using basic overrun $\mathsf{load}_{\mathsf{sys}} = 0,44$. The basic overrun is better than the enhanced overrun by about 3.8%.

Example 2: For the second case ($Q_s^\circ/P_S \geq Q_s^*/(P_s - H_s)$), suppose that a system S consists of two subsystems S_1 with parameters $P_1 = 50, Q_1^\circ = 10, Q_1^* = 9, H_1 = 4$ and S_2 with parameters $P_2 = 150, Q_2^\circ = 15, Q_2^* = 12, H_2 = 8$. Then using enhanced overrun $\mathsf{load}_{\mathsf{sys}} = 0,456$ and using basic overrun $\mathsf{load}_{\mathsf{sys}} = 0,44$. The basic overrun is better than the enhanced overrun by about 1.2%.

Example 3: For the second case ($Q_s^\circ/P_S \geq Q_s^*/(P_s - H_s)$), suppose that a system S consists of two subsystems S_1 with parameters $P_1 = 20, Q_1^\circ = 5, Q_1^* = 4, H_1 = 2$ and S_2 with parameters $P_2 = 150, Q_2^\circ = 10, Q_2^* = 9, H_2 = 2$. Then using enhanced overrun $\mathsf{load}_{\mathsf{sys}} = 0,285$ and using basic overrun $\mathsf{load}_{\mathsf{sys}} = 0,3$. The enhanced overrun is better than the basic overrun by about 11.5%.

9.7 Computing resource holding time

In this section we explain how to compute the resource holding time $h_{j,i}$. Using the periodic virtual processor model, each subsystem S_s receives CPU resources with allocation time Q_s every period P_s. During Q_s, the CPU allocation is 100 % of the CPU capacity (see Figure 9.2 where the slope in the supply curve during Q is one). The mechanism presented in Section 9.5 guarantees that locking and releasing a critical section of a globally shared resource R_j will happen within the allocated CPU resource $Q_s + \theta$. Then $h_{j,i}$ will include the execution time of the task τ_i that locks R_j inside the critical section as well as the interference from all tasks within the same subsystem that can preempt

the execution inside the critical section. The worst case scenario happens when all tasks that can preempt the execution of the critical section will be released just after task τ_i has entered the critical section of resource R_j. Then, the $h_{j,i}$ is computed [19] using $W_j(t)$ as follows:

$$W_j(t) = cx_j + \sum_{\tau_k \in U} \Big(min\Big(\Big\lceil \frac{t}{T_k} \Big\rceil, \Big\lfloor \frac{D_i - D_k}{T_k} \Big\rfloor + 1 \Big) \Big) \cdot C_k, \qquad (9.14)$$

where $cx_j = \max\{c_{i,j}\}$ is the maximum execution time of task τ_i inside the critical section of the resource R_j and U is the set of tasks such that $U = \{\tau_k | \pi_k > rc_j\}$.

The resource holding time $h_{j,i}$ is the smallest positive time t^* such that

$$W_j(t^*) = t^*. \qquad (9.15)$$

Note that we have not counted the preemption inside the critical section from other subsystems when calculating the resource holding time. However, we are taking the interference from higher preemption level subsystems into account in the global schedulability analysis. Looking at Eq. (9.4), $h_{j,i}$ may act as a blocking to other subsystems. Moreover, it is also the extra capacity required to prevent budget expiry inside critical section. When $h_{j,i}$ is considered as a blocking time for other subsystems, the effect of interference from higher preemption level subsystems inside the critical section will be included in the global schedulability of the blocked subsystem (in the summation part of Eq. (9.4)). When $h_{j,i}$ is used to evaluate the overrun, interference from other subsystems inside the critical section will not be important, as the only important part here is that the locked resource should be released before the end of the period.

Eq. (9.14) can be simplified to evaluate $h_{j,i}$ as shown below:

$$h_{j,i} = cx_j + \sum_{\tau_k \in U} C_k \qquad (9.16)$$

The difference between Eq. (9.16) and Eq. (9.14) is that in Eq. (9.14) we assume that all tasks that can preempt inside the critical section can execute only once (we remove the min function from the summation of Eq. (9.14)). The reason for why it is safe to assume only one execution of each preempting task inside the critical section is given in the following lemma, showing that if a task executes more than one time inside the critical section, the subsystem will become unschedulable.

Lemma 5. *For a subsystem that uses an overrun mechanism to arbitrate access to global shared resource under the periodic virtual processor model, each task that is allowed to preempt the execution of anther task currently inside the critical section of a globally shared resource can, in the worst case, only execute (cause interference) once.*

Proof. We prove this lemma by considering two cases:

(1) $P_s < T_m$ (where $T_m = min(T_i)$ for all $i = 1, ..n$), if the task having period T_m executes 2 or more times inside the critical section, this means that the resource will be locked during this period, i.e., $h_{j,i} > T_m$ then $h_{j,i} > P_S$, which in turn means that the CPU utilization required by the subsystem S_s will be $U_s = (Q_s + h_{j,i})/P_s > 1$.

(2) If $P_s \geq T_m$, $\texttt{sbf}_\Gamma(t)$ should provide at least C_m at time $t = T_m$ to ensure the schedulability test in Eq. (9.3). Note that $\texttt{sbf}_\Gamma(t) = 0$ during $t \in [0, 2P_s - 2Q_s]$ so, $2P_s - 2Q_s < T_m$ which means $Q_s > P_s - T_m/2$.

If the task that has period T_m execute 2 times inside the critical section then $h_{j,i} > T_m$. Hence, $Q_s + h_{j,i} > P_s + T_m/2$ which means $U_s = (Q_s + h_{j,i})/p > 1$. □

From Lemma (5), we can conclude that if $t^* > T_m$ then the required CPU utilization U_s will be greater than one. This means that, in turn, all tasks that can preempt the execution of a critical section should do so maximum one time in order to keep the utilization of a subsystem less than one. This proves the correctness of Eq. (9.16) which is based on the assumption that all tasks can interfere only once as a worst case while a task is in the critical section of the resource R_j. If the value of $h_{j,i}$ becomes greater than $min(T_m, P_s)$ then we can conclude that the subsystem will not be schedulable and we do not have to continue the calculation towards finding a exact value of $h_{j,i}$.

9.8 Summary

In this paper we have considered a new overrun mechanism, for hierarchical scheduling frameworks, that can be used in the domain of open environments [3]. The main contributions of this paper are twofold: (1) we have presented analysis of when one overrun mechanism is better than the other, and (2) we have presented how to calculate resource holding times when using the periodic virtual processor model.

The results indicate that in the general case it is not trivial to evaluate which overrun method that is better than the other, as their impact on the CPU utiliza-

tion is highly dependent on global system parameters such as subsystem periods and budgets. However, for open systems, enhanced overrun is generally better than basic overrun, as it moves the effect of overrun from the local to the global schedulability analysis.

Future work includes the development of local and global schedulability analysis for Fixed Priority Scheduling (FPS), as the current results consider Earliest Deadline First (EDF). Another interesting issue is to compare the enhanced overrun mechanism with other synchronization mechanisms such as BWI [24], BROE server [22] and SIRAP [21].

Bibliography

[1] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* 24^{th} *IEEE International Real-Time Systems Symposium(RTSS'03)*, pages 2–13, Cancun, Mexico, December 2003.

[2] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the* 27^{th} *IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, Rio de Janeiro, Brazil, December 2006.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin. An overrun method to support composition of semi-independent real-time components. *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 1347–1352, 28 2008-Aug. 1 2008.

[4] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the* 18^{th} *IEEE International Real-Time Systems Symposium (RTSS'97)*, pages 308–319, San Francisco, CA, USA, December 1997. IEEE Computer Society.

[5] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the* 20^{th} *IEEE International Real-Time Systems Symposium (RTSS'99)*, pages 256–267, Phoenix, AZ, USA, December 1999. IEEE Computer Society.

[6] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, Washington DC, USA, May-June 2000. IEEE Computer Society.

[7] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard-real-time environments. In *Proceedings of the* 21^{th} *IEEE International Real-Time Systems Symposium(RTSS'00)*, pages 217–226, Orlando, FL, USA, December 2000.

[8] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium(RTAS)*, pages 75–84, Taipei, Taiwan ROC, May 2001.

[9] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* 23^{th} *IEEE International Real-Time Systems Symposium (RTSS'02)*, pages 26–35, Austin, TX, USA, December 2002.

[10] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the* 25^{th} *IEEE International Real-Time Systems Symposium(RTSS'04)*, pages 57–67, Lisbon, Portugal, December 2004. IEEE Computer Society.

[11] 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.

[12] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarhical fixed-priority scheduling. In *Proceedings of the* 14^{th} *Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 152–160, Vienna, Austria, June 2002. IEEE Computer Society.

[13] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 151–158, Porto, Portugal, July 2003. IEEE Computer Society.

[14] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the* 26^{th} *IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 389–398, Miami Beach, FL, USA, December 2005.

[15] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium(RTSS'07)*, pages 129–138, Washington, DC, USA, 2007. IEEE Computer Society.

[16] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation IECON87*, pages 909–916, Cambridge, MA, USA, November 1987.

[17] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the* 9^{th} *IEEE International Real-Time Systems Symposium (RTSS'88)*, pages 259–269, Huntsville, AL, USA, December 1988. IEEE Computer Society.

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

[19] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in EDF-scheduled systems. In *Proceedings of the* 13^th *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, Bellevue, WA, USA, 2007.

[20] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT '04)*, pages 95–103, Pisa, Italy, September 2004.

[21] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, pages 279–288, Salzburg, Austria, October 2007.

[22] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, Washington, DC, USA, December 2007. IEEE Computer Society.

[23] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the* 19^{th} *IEEE International Real-Time Systems Symposium (RTSS'98)*, pages 4–13, Madrid, Spain, December 1998. IEEE Computer Society.

[24] G. Lipari, G. Lamastra, and L. Abeni. Task synchronization' in reservation-based real-time systems. *IEEE Transactions on Computers*, 53(12):1591–1601, December 2004.

[25] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the* 11^th *IEEE International Real-Time Systems Symposium(RTSS'90)*, pages 182–190, Lake Buena Vista, Florida, USA, December 1990. IEEE Computer Society.

[26] Sanjoy K. Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of the* 27^{th} *IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 379–387, Rio de Janeiro, Brazil, December 2006.

Chapter 10

Paper D: Synthesis of Optimal Interfaces for Hierarchical Scheduling with Resources

Insik Shin, Moris Behnam, Thomas Nolte, Mikael Nolin

In Proceedings of the 29^{th} IEEE International Real-Time Systems Symposium (RTSS08), IEEE Press, Barcelona, Spain, December, 2008, (to be appear).

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.

10.1 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, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] have been introduced for the schedulability analysis of HSFs, where subsystems are independent. For dependent subsystems, synchronization protocols [14, 15, 16] have been proposed for arbitrating accesses to logical resources (i.e., semaphore) across subsystems in HSFs. There have been a few studies [11, 5] 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* [11, 17], 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 [18], 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 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 [3], 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 10.2 presents related work, followed by system model and background in Section 10.3. Section 10.4 presents schedulability analysis in our HSF, followed by problem formulation and solution outline in Section 10.5. Section 10.6 addresses the first step of the two-step approach; efficiently generating interface candidates, and Section 10.7 resolves the second step finding an optimal solution out of the candidates. Section 10.8 discuss the use of another overrun mechanism which called overrun with payback mechanism and finally, Section 10.9 concludes.

10.2 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 [3] in the late 1990's, has been receiving an increasing research attention. Since Deng and Liu [3] introduced a two-level HSF, its schedulability has been analyzed under fixed-priority global scheduling [7] and under Earliest Deadline First (EDF) based global scheduling [8]. Mok *et al.* [10] proposed the bounded-delay virtual processor model to achieve a clean separation in a multi-level HSF, and schedulability analysis techniques [6, 12] have been

introduced for this resource model. In addition, Shin and Lee [11, 17] 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, 9, 2] and under EDF scheduling [11, 13]. More recently, Easwaran *et al.* [4] 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) [20], and Stack Resource Policy (SRP) [21]. There have been studies on supporting resource sharing within subsystems [1, 7] 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 [15] and skipping [14], and another protocol [16] for bounded-delay virtual processor model based HSFs. Bertogna *et al.* [18] 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.

10.3 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.

10.3.1 Virtual processor models

The notion of real-time virtual processor model was first introduced by Mok *et al.* [10] 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 [11] 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 $\texttt{sbf}_\Gamma(t)$ of $\Gamma(P, Q)$ was given in [11] that computes the minimum possible CPU supply for every interval length t as follows:

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

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

10.3.2 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 10.3.3. 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 [18], however, developed for a different virtual-processor model. RHT in [18] is developed for a dedicated

processor model[1] (or a fractional processor model [10]), where subsystems do not preempt each other. However, our HSF is based on a time-shared (partitioned) processor model [11], 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.

10.3.3 Stack Resource Policy (SRP)

In this paper, we consider the SRP protocol [21] 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 [18]):

[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 [22] for its computation in our HSF.

$$w_j^{(m+1)} = cx_j + \sum_{k=rc_j+1}^{n} \lceil \frac{w_j^{(m)}}{T_k} \rceil \cdot C_k, \tag{10.1}$$

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. (10.1) 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, $X_s = \max\{w_j |$ for all $R_j \in \mathcal{R}_s\}$, where $\mathcal{R}_s$ is a set of global shared resources accessed by S_s.

10.4 Resource sharing in the HSF

10.4.1 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 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 [15] 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.

10.4.2 Schedulability analysis

In this paper, we use HSRP [15] for resource synchronization in HSF. Schedulability analysis under global and local FPS with the overrun mechanism is presented in [15]. 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 $\mathtt{rbf}_{\mathtt{FP}}(i,t)$ denote the request bound function of a task τ_i under FPS [23], i.e.,

$$\mathtt{rbf}_{\mathtt{FP}}(i,t) = C_i + \sum_{\tau_k \in \mathtt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \tag{10.2}$$

The local schedulability analysis under FPS can be then easily extended from the results of [21, 11] as follows:

$$\forall \tau_i, 0 < \exists t \leq D_i \;\; \mathtt{rbf}_{\mathtt{FP}}(i,t) + b_i \leq \mathtt{sbf}(t), \tag{10.3}$$

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 can be selected within a finite set of scheduling points [24].

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. (10.3) depending on the local scheduler of S_s. Such a function is similar to the one in [11]. 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. (10.2)):

$$\mathtt{LBF}_s(t) = \mathtt{RBF}_s(t) + B_s \ , \text{ where} \tag{10.4}$$

$$\texttt{RBF}_s(t) = (Q_s + O_s(t)) + \sum_{S_k \in \texttt{HPS}(s)} \left\lceil \frac{t}{P_k} \right\rceil (Q_k + O_k(t)), \tag{10.5}$$

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,

$$B_s = \max\{X_j |\ S_j \in \texttt{LPS}(S_s)\}, \tag{10.6}$$

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

A global schedulability condition under FPS is then

$$\forall S_s, 0 < \exists t \leq P_s\ \texttt{LBF}_s(t) \leq \texttt{t} \tag{10.7}$$

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

$$\alpha_s = \min_{0<t\leq P_s} \{\frac{\texttt{LBF}_s(t)}{t} \mid \texttt{LBF}_s(t) \leq t\}. \tag{10.8}$$

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

$$\mathsf{load}_{\mathsf{sys}} = \max_{\forall S_s \in \mathcal{S}} \{\alpha_s\}. \tag{10.9}$$

Note that α_s is the smallest fraction of the CPU resources that is required to schedule a subsystem S_s (satisfying Eq. (10.7)) 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.

10.5 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 subject to the schedulability of all subsystems. It is suggested from Eqs. (10.4) and (10.9) that $\mathsf{load}_{\mathsf{sys}}$ can be minimized by reducing Q_s and X_s for all subsystem S_s.

A recent study [18] introduced a method to reduce X_i. According to Eq. (10.1), the value of X_s can decrease, when it has less interference (i.e., the summation part of Eq. (10.1)) 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 [18], 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. (10.3), this may require more CPU supply (i.e., Q_s). Figure 10.1 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. (10.6), 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

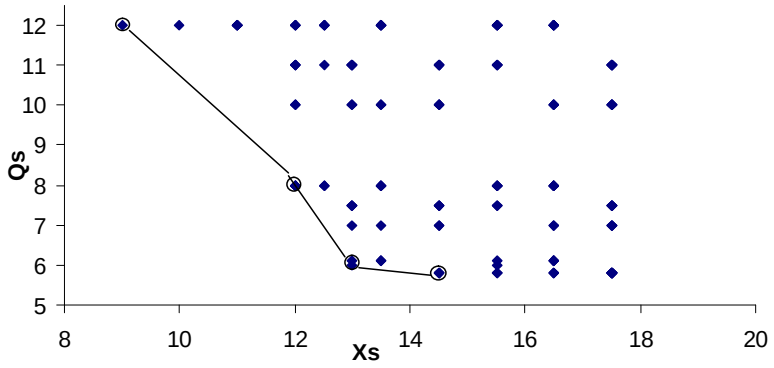


Figure 10.1: Tradeoff between Q_s and X_s.

system load can be minimized.

10.6 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 10.5, 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. (10.4) and (10.5). 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 6. *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. (10.5)) 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. (10.8)). That is, a redundant candidate only has a potential to increase $\mathsf{load}_{\mathsf{sys}}$ (see Eq. (10.9)).
- both interfaces are equivalent then system load for both is the same.

□

Lemma 6 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 7. *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.,*

$$\Big(b_{i,k} = \max_{\forall R_j \in \mathcal{R}^i}\{b_{i,j}\}\Big) \rightarrow \Big(w_{k,i} = \max_{\forall R_j \in \mathcal{R}^i}\{w_{j,i}\}\Big). \tag{10.10}$$

Proof. The $w_{j,i}$ depends on two parameters (see Eq. (10.1)); 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 7, the following lemma particularly shows how we can effectively exclude redundant candidates.

Lemma 8. *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 7 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, $X^* \leq X'$. □

- calculateBudget$(S_s, P_s, \mathcal{RC}_s)$ returns the smallest subsystem budget that satisfies Eq. (10.2).
- increaseCeilingX$^*(\mathcal{RC}_s)$ 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 8).
- 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: $\mathcal{RC}_s = \{rc_1, \cdots, rc_m\}$ // rc_j=initial ceiling of R_j using SRP
2: num $= 0$
3: do
4: $\quad Q^* =$ calculateBudget$(S_s, P_s, \mathcal{RC}_s)$
5: $\quad X^* = \max\{w_{1,rc_1}, \cdots, w_{m,rc_m})\}$
6: $\quad$ addInterface(Interface, $Q^*, X^*, \mathcal{RC}_s$)
7: $\quad$ num=removeRedundant(Interface)
8: while (increaseCeilingX$^*(\mathcal{RC}_s)$)
9: return (Interface, num)

Figure 10.2: 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 10.1: Example task set parameters

10.6.1 ICG algorithm

Description. Using Lemmas 6, 7, and 8, we can reduce the complexity of a search algorithm. The algorithm shown in Figure 10.2 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 [18]). 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 8 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 10.1. 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 10.2. 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 6. 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 8 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 8. The algorithm selects the interface candidates from steps $3, 4$ and 5.

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

Lemma 9. *Let $\mathcal{IC}$ denote a set of up to n interface candidates that are generated by the ICG algorithm of Figure 10.2. There exists no non-redundant*

$Step$	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 10.2: Example algorithm

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 8, 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 7 and 8).

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 7, $b_{f,k}$ (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 6. 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 10.4.1 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 $\texttt{RBF}_s(t)$ 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. (10.4)-(10.6). Then, the following lemma directly follows:

Lemma 10. *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. (10.4), we can decrease $\texttt{LBF}_s(t)$ to decrease the system load by decreasing the blocking B_s and/or $\texttt{RBF}_s(t)$. 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 $\texttt{RBF}_s(t)$, 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. (10.5). 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 10, 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.

10.7 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.

10.7.1 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

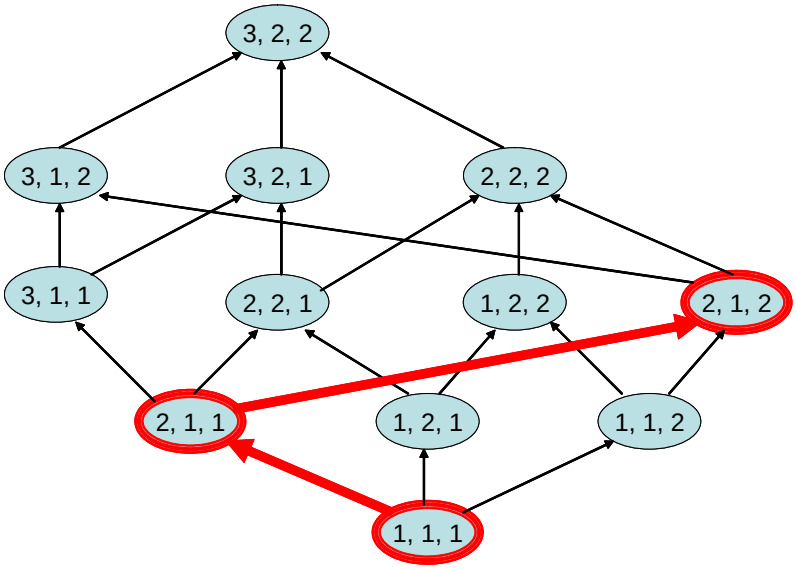


Figure 10.3: Search space for a system consisting of 3 subsystems.

increasing order of collective demands ($Q_s + X_s$) (see Lemma 10). 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_k^i to denote an element of CF_i, i.e., $\mathsf{CF}_i = \{f_1^i, \ldots, f_N^i\}$. The variable f_k^i represents the interface candidate index of a subsystem S_k, indicating that the configuration in the i-th step includes $(Q_{k,f_k^i}, X_{k,f_k^i})$.

Figure 10.3 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_k^i 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 10.3, $\mathsf{CF}_0 = \{1, 1, 1\}$ (see line 2 of Figure 10.4).

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 10.3, 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.,

$$f_k^i = \begin{cases} f_k^{i-1} + 1 & \text{if } k = \delta_i, \\ f_k^{i-1} & \text{otherwise.} \end{cases} \tag{10.11}$$

In the example shown in Figure 10.3, $\delta_1 = 1$.

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. (10.9)). We can find such S_{s^*} by evaluating the *processor request bound*'s of all subsystems (in line 5 of Figure 10.4).

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. (10.4), 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 10.4).

When such S_{k^*} is found, it then checks whether the X_{k^*} can be further reduced (in line 7 of Figure 10.4). 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.

10.7.2 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.,

$$\mathcal{AS} = IC_1 \times \cdots \times IC_n. \tag{10.12}$$

In the example shown in Figure 10.3, 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.

[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. (10.9)
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 10.4: The ICS algorithm.

• $\widehat{IC}_k^i$ 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 during iteration. For notation convenience, we introduce a variable ($\widehat{IC}_k^i$) to indicate the remaining interface candidates of a subsystem

S_k to explore. By definition, 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

$$\widehat{IC}_k^{\,i} = \{f_k^i, \ldots, max_k\} \text{ for all } k = 1, \ldots, n, \tag{10.13}$$

where max_k is the number of interface. In the example shown in Figure 10.3, $\widehat{IC}_1^{\,1} = \{2, 3\}$.

• 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

$$\mathsf{XP}_i = (\widehat{IC}_1^{\,i} \times \cdots \times \widehat{IC}_n^{\,i}) \setminus \mathsf{CF}_i. \tag{10.14}$$

• 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

$$\mathsf{RM}_i = (\widehat{IC}_1^{(i-1)*} \times \cdots \times \widehat{IC}_n^{(i-1)*}) \setminus \{\mathsf{CF}_{i-1}\}, \text{ where} \tag{10.15}$$

$$\widehat{IC}_k^{(i-1)*} = \begin{cases} \{f_k^{i-1}\} & \text{if } k = \delta_i, \\ \widehat{IC}_k^{\,i} & \text{otherwise.} \end{cases} \tag{10.16}$$

In the example shown in Figure 10.3, $\mathsf{RM}_1 = \{\{1, 2, 1\}, \{1, 2, 2\}, \{1, 1, 2\}\}$.

• 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.,

$$\mathsf{AH}_i = \{\mathsf{CF}_1, \ldots, \mathsf{CF}_i\}. \tag{10.17}$$

• 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.,

$$\mathsf{AR}_i = \mathsf{RM}_{(i-1)} \cup \mathsf{RM}_i, \text{ where } \mathsf{AR}_0 = \phi. \tag{10.18}$$

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 11. *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 10.7.1, 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 10.4, 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 12. *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 11, 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. (10.5) . 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 13. *At an arbitrary i-th iteration step,*

$$\mathsf{XP}_i = \mathsf{XP}_{i-1} \setminus (\mathsf{RM}_i \cup \{\mathsf{CF}_i\}). \tag{10.19}$$

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

$$\widehat{IC}_k^i = \begin{cases} \widehat{IC}_k^{i-1} \setminus \{f_k^{i-1}\} & \text{if } k = \delta_i, \\ \widehat{IC}_k^{i-1} & \text{otherwise.} \end{cases} \tag{10.20}$$

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

$$\begin{aligned} \mathsf{XP}_i^* &= \widehat{IC}_1^i \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i \\ &= \left(\widehat{IC}_1^{i-1} \setminus \{f_1^{i-1}\}\right) \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i \\ &= \left(\widehat{IC}_1^{i-1} \times \widehat{IC}_2^{i-1} \times \cdots \times \widehat{IC}_n^{i-1}\right) \setminus \\ &\quad \left(\{f_1^{i-1}\} \times \widehat{IC}_2^i \times \cdots \times \widehat{IC}_n^i\right) \\ &= \mathsf{XP}_{i-1}^* \setminus \mathsf{RM}_i^* \\ &= \left(XP_{i-1} \cup \{\mathsf{CF}_{i-1}\}\right) \setminus \left(\mathsf{RM}_i \cup \{\mathsf{CF}_{i-1}\}\right) \\ &= XP_{i-1} \setminus \mathsf{RM}_i \,. \end{aligned} \tag{10.21}$$

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

$$XP_i = XP_{i-1} \setminus (\mathsf{RM}_i \cup \{\mathsf{CF}_i\}) \,. \tag{10.22}$$

□

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

$$\mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i = \mathcal{AS} \tag{10.23}$$

Proof. We will prove this lemma by using mathematical induction. As a base step, we wish to show Eq. (10.23) 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. (10.14). It follows that $\mathsf{AR}_0 \cup \mathsf{AH}_0 \cup \mathsf{XP}_0 = AP$.

We assume that Eq. (10.23) 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.,

$$\mathsf{AR}_i \cup \mathsf{AH}_i \cup \mathsf{XP}_i = \mathsf{AR}_{i+1} \cup \mathsf{AH}_{i+1} \cup \mathsf{XP}_{i+1}. \tag{10.24}$$

According to the definitions AH_{i+1}, AR_{i+1}, and XP_{i+1} (see Eq. (10.17), (10.18) and (10.19)), we can rewrite the right-hand side of Eq. (10.24) as follows:

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

□

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

Theorem 15. *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 11, $\mathsf{load}_{\mathsf{sys}}(\mathsf{CF}_{i-1}) < \mathsf{load}_{\mathsf{sys}}(opt)$, which contradicts the definition of opt. Hence, $opt \notin \mathsf{AR}_f$.

Suppose $opt \in \mathsf{XP}_f$. Then, according to Lemma 12, 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 14, it follows that $opt \in \mathsf{CF}_f$. □

10.8 Overrun mechanism with payback

David and Burns [15] presented another overrun mechanism called overrun with payback. 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 one affected by 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 the overrun with payback mechanism and we will discuss how this will effect on system load. First, we will briefly explain how to analyze the local and global schedulability with this type of overrun mechanism.

Local schedulability analysis. We can still use Eq. (10.3) for the payback version of overrun where $\mathtt{dbf}_{\mathsf{FP}}(i,t)$ is the same as in the overrun without payback (presented in Section 10.4.1). However, the $\mathtt{sbf}(t)$ 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 [25] 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 required by the other version of overrun.

Global schedulability analysis. Eq. (10.7) is valid only if we change $O_k(t)$ in Eq. (10.5) such that $O_k(t) = X_k$ for $0 \leq t \leq P_k$. Then Eq. (10.5) using the overrun with payback mechanism can be rewritten as follows,

$$\mathtt{DBF}_s(t) = Q_s + X_s + \sum_{S_k \in \mathtt{HPS}(S_s)} \left\lceil \frac{t}{P_k} \right\rceil \cdot (Q_k) + X_k \tag{10.25}$$

The ICG algorithm presented in Section 10.6.1 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 7-9 are based on the local scheduling. Lemma 6 is based on the global scheduling, and it is valid also with the payback version of overrun.

For the ICS algorithm, the possibilities to minimize $\texttt{LBF}_s(t)$ using overrun with payback are as follows; looking at Eq. (10.25), and depending on the values of P_s, P_k, Q_k, X_k, the value of $DBF_s(t)$ can be minimized in some cases by minimizing $Q_k + X_k$ and in other cases by minimizing only Q_k. The second factor that has effect on $LBF_s(t)$ is $Q_s + X_s$ of the subsystem, and the third factor is X_s. So there is an additional factor that affect on $LBF_s(t)$ using the payback version compared with the other version of the overrun mechanism, which is minimizing Q_s. Hence, Lemma 10 may not be correct for all cases when using the overrun mechanism with payback. We can conclude that the optimization problem when using overrun with payback is more complex and the ICS algorithm may not be able to find an optimal solution.

Comparing the two versions of the overrun mechanisms, the overrun mechanism without payback is better than the other version in the local schedulability, and it will require lower subsystem budget. While in the global schedulability, the payback version will be better than the other version because the interference from higher priority subsystems on S_s is increased by Q_k every period P_k (see Eq. (10.25)). On the other hand, when using overrun without payback, the interference from higher priority subsystems increases by $Q_k + X_k$ every period P_k (see Eq. (10.5)). Another difference between the two versions of overrun mechanisms is that overrun with payback has a restriction in $Q_s \geq X_s$ while there is no such restriction when using overrun without payback.

10.9 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 in-

cludes generalizing our framework to other synchronization protocols. For example, this paper considered only the overrun mechanism without payback [15], and we are extending towards another overrun mechanism (with-payback version) [15]. 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. (10.5)), and this requires appropriate changes to the post-processing part of the ICG algorithm. We are investigating how to make changes to the post-processing part in ways that require less subsequent changes to the ICS algorithm.

Bibliography

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT '04)*, pages 95–103, Pisa, Italy, September 2004.

[2] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the* 26^{th} *IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 389–398, Miami Beach, FL, USA, December 2005.

[3] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the* 18^{th} *IEEE International Real-Time Systems Symposium (RTSS'97)*, pages 308–319, San Francisco, CA, USA, December 1997. IEEE Computer Society.

[4] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium(RTSS'07)*, pages 129–138, Washington, DC, USA, 2007. IEEE Computer Society.

[5] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hieararchical real-time components. In *Proceedings of the 6th ACM Conference on Embedded Software*, September 2006.

[6] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* 23^{th} *IEEE International Real-Time Systems Symposium (RTSS'02)*, pages 26–35, Austin, TX, USA, December 2002.

[7] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the* 20^{th} *IEEE International*

Real-Time Systems Symposium (RTSS'99), pages 256–267, Phoenix, AZ, USA, December 1999. IEEE Computer Society.

[8] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, Washington DC, USA, May-June 2000. IEEE Computer Society.

[9] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 151–158, Porto, Portugal, July 2003. IEEE Computer Society.

[10] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium(RTAS)*, pages 75–84, Taipei, Taiwan ROC, May 2001.

[11] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* 24^{th} *IEEE International Real-Time Systems Symposium(RTSS'03)*, pages 2–13, Cancun, Mexico, December 2003.

[12] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the* 25^{th} *IEEE International Real-Time Systems Symposium(RTSS'04)*, pages 57–67, Lisbon, Portugal, December 2004. IEEE Computer Society.

[13] F. Zhang and A. Burns. Analysis of hierarchical EDF pre-emptive scheduling. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 423–434, Washington, DC, USA, December 2007. IEEE Computer Society.

[14] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, pages 279–288, Salzburg, Austria, October 2007.

[15] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the* 27^{th} *IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, Rio de Janeiro, Brazil, December 2006.

[16] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. In *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, Washington, DC, USA, December 2007. IEEE Computer Society.

[17] 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.

[18] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *Proceedings of the 15th International Workshop on Parallel and Distributed Real-Time Systems(WPDRTS)*, pages 1–8, Long Beach, CA, USA, March 2007.

[19] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation IECON87*, pages 909–916, Cambridge, MA, USA, November 1987.

[20] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the* 9^{th} *IEEE International Real-Time Systems Symposium (RTSS'88)*, pages 259–269, Huntsville, AL, USA, December 1988. IEEE Computer Society.

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

[22] Insik Shin, Moris Behnam, Thomas Nolte, and Mikael Nolin. On optimal hierarchical resource sharing in open environments. Technical report, 2008. Available at http://www.idt.mdh.se/∼tnt/rtss08long.pdf.

[23] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the* 20^{th} *IEEE International Real-Time Systems Symposium(RTSS'89)*, pages 166–171, Santa Monica, CA, USA, December 1989. IEEE Computer Society.

[24] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269, 2005.

[25] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. Scheduling of semi-independent real-time components: Overrun methods and resource holding times. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08)*. IEEE Industrial Electronics Society, September 2008.