Hierarchical Scheduling of Complex Embedded Real-Time Systems*

Thomas Nolte, Moris Behnam, Mikael Åsberg
MRTC/Mälardalen University
P.O. Box 883, SE-721 23, Västerås, Sweden

Reinder J. Bril
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5612 AZ, Eindhoven, The Netherlands

Insik Shin
KAIST
Daejeon, South Korea 305-701

Abstract

For most of today's embedded software systems, correct operation requires not only correct function, they must additionally satisfy specific extra-functional properties, in particular related to timing. System development (including software development) is substantially facilitated if the system parts can be developed and verified in isolation, and if the correctness of the system can be inferred from the correctness of its parts. Such modular and compositional design of software system has for a long time been considered the holy-grail of system design, and is unfortunately only possible in selected scenarios. This paper deals with one such scenario: using hierarchical scheduling to provide predictable timing and temporal isolation of embedded software. During the past years we have worked on various issues on hierarchical scheduling, and this paper presents an overview of selected research results, focusing on issues related to synchronization among software modules.

1 Introduction

Component based software engineering is promoted as a key approach in providing structured software design and reuse for embedded (and other) software systems [3, 12]. Advanced operating system mechanisms such as hierarchical scheduling frameworks provide temporal and spatial isolation through virtual platforms, thereby providing means for extending component based software engineering towards component based systems engineering. A complex system can be divided into several modules, here denoted subsystems, each performing a specific well defined function. Development and verification of subsystems can ideally be performed independently (and concurrently) and their seamless and effortless integration results in a correctly functioning final product both from a functional as well as extra-functional point of view. This paper presents the hierarchical scheduling framework; a step towards concurrent development and reuse of complex embedded software systems with extra-functional requirements on timing and resource usage.

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

1.1 Background

The hierarchical scheduling framework is a modular approach for scheduling embedded real-time systems. A system is 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 division of a system into a number of subsystems naturally promotes *reuse* of subsystems from one system to another. Also, the hierarchical scheduling framework allows for a subsystem to be developed and analysed in isolation, with its own local scheduler, and then at a later stage, using an arbitrary global scheduler, integrated with other subsystems without violating the results of the analyses performed on the subsystem in isolation. The integration involves a system-level schedulability test, verifying that all extra-functional requirements are met. Hence, hierarchical scheduling frameworks naturally support *concurrent development* of subsystems.

The key enabler in allowing for concurrent development and reuse of subsystems is the subsystem *interface*. Subsystems are periodically scheduled in the hierarchical scheduling framework, and the subsystem interface contains information on the fraction of the CPU required by a subsystem in each subsystem period. As long as this fraction of CPU is always provided to the subsystem, it is guaranteed that the subsystem will function according to its specifications, e.g., that the extra-functional temporal requirements of the subsystem are met. Hence, an interface contains information representing the capacity Q to be provided to the subsystem each subsystem period P. If the hierarchical scheduling framework supports sharing of logical resources

among subsystems, the interface must also contain information on the length of the longest critical section in the subsystem [8]. Hence, interfaces for subsystems sharing logical resources contain, apart from Q and P, also the length of the longest critical section X.

During the past years, we have developed a hierarchical scheduling framework providing predictable timing. In particular, we have focused on the development of synchronization protocols for hierarchical scheduling together with associated analysis techniques. Our overall goal is to develop a cost efficient framework applicable for a wide range of applications, and this paper covers some of our recent work in hierarchical scheduling, synchronization, adaptation and implementation.

1.2 Outline

This paper presents a hierarchical scheduling framework based on the periodic resource model [34]. In Section 2, the paper covers related work in the area of hierarchical scheduling, and the related issue of synchronization among tasks executing in a hierarchical framework. The hierarchical scheduling framework is presented in detail in Section 3, along with its associated timing analysis in Section 4. Following, the paper covers synchronization techniques for hierarchical scheduling in Section 5, comparing several approaches, and going into detail into one of these approaches in Section 6; the approach of the Subsystem Integration and Resource Allocation Policy (SIRAP) [8]. The paper continues with discussing the role of a hierarchical scheduling framework in dynamic systems in Section 7, allowing for the system to change its configuration during runtime. Finally, Section 8 discusses implementation issues and experiences taken from an implementation of the hierarchical scheduling framework in the VxWorks operating system, and Section 9 concludes.

2 Related work

Before going into details describing our hierarchical scheduling framework, this section outlines related work in the area of hierarchical scheduling and protocols for synchronization when systems are scheduled hierarchically.

2.1 Hierarchical scheduling

Hierarchical real-time scheduling, originating in open systems [15] in the late 1990's, has been receiving an increasing research attention [2, 13, 15, 17, 20, 22, 23, 26, 30, 34, 35]. Since Deng and Liu [15] introduced a two-level hierarchical scheduling framework, its schedulability has been analyzed under fixed-priority global scheduling [20] and under EDF-based global scheduling [22, 25]. Mok *et al.* [27] proposed the bounded-delay resource model so as to achieve a clean separation in a multi-level hierarchical scheduling framework, and schedulability analysis techniques [17, 35] have been introduced for this resource model. In addition, Shin and Lee [34] introduced another so-called 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 [13, 23, 31] and under EDF scheduling [34]. More recently, Easwaran *et al.* [16] 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, i.e., no sharing of logical resources is allowed.

2.2 Synchronization

In many real systems, tasks are required to interact with each other through mutually exclusive resource sharing. Many protocols have been introduced to address the priority inversion problem for tasks sharing logical resources, including the Priority Inheritance Protocol (PIP) [32], the Priority Ceiling Protocol (PCP) [29], and Stack Resource Policy (SRP) [4]. Recently, Fisher *et al.* addressed the problem of minimizing the resource holding time [19] under SRP. There have been studies on extending SRP in a hierarchical scheduling framework, for sharing of logical resources within a subsystem [2, 20] and across subsystems [8, 14, 18]. Davis and Burns [14] proposed the Hierarchical Stack Resource Policy (HSRP) supporting sharing of logical resources on the basis of an overrun mechanism. Behnam *et al.* [8] 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.* [18] proposed the BROE server that extends the Constant Bandwidth Server (CBS) [1] in order to handle sharing of logical resources in a hierarchical scheduling framework. The work in this paper focuses on HSRP and SIRAP, targeting systems based on FPS schedulers. Note that FPS is the defacto standard used (for local scheduling) in industry.

3 The Hierarchical Scheduling Framework

This paper focuses on scheduling of a single node, where each node is modeled as a system $\mathcal{S}$ consisting of one or more subsystems $S_s \in \mathcal{S}$. The system is scheduled by a two-level Hierarchical Scheduling Framework (HSF) as shown in Figure 1. During runtime, the system level scheduler (global scheduler) selects, at all times, which subsystem that will access the common (shared) CPU resource. The synchronization protocols, SRP mediates access to local shared logical resources, and HSRP and SIRAP will mediate access to global shared logical resources.

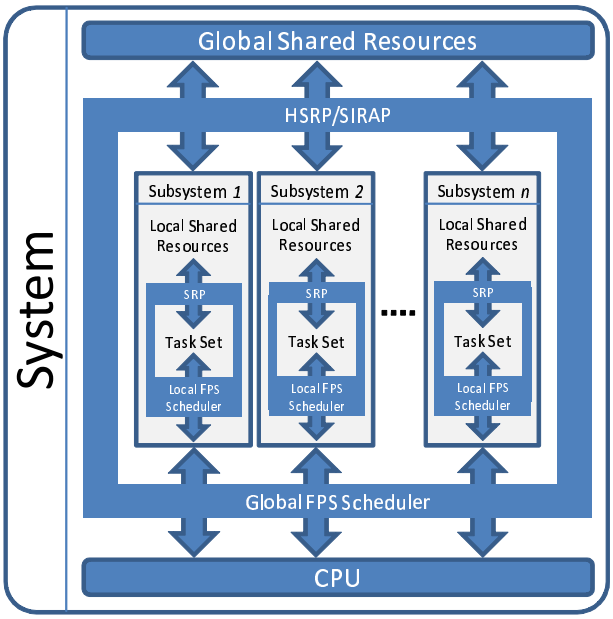

Figure 1. HSF with resource sharing.

3.1 Subsystem model

A subsystem S_s consists of a set $\mathcal{T}_s$ of n_s tasks and a local scheduler. Once a subsystem is assigned the processor (CPU), its scheduler will select which of its tasks will be executed. With each subsystem S_s, a subsystem timing interface $S_s(P_s, Q_s, X_s)$ is associated, where Q_s is the subsystem budget that the subsystem S_s will receive every subsystem period P_s, and X_s is the maximum time that a subsystem internal task may lock a globally shared resource. Finally, both the local scheduler of a subsystem S_s as well as the global scheduler of the system $\mathcal{S}$ is assumed to implement the FPS scheduling policy. Let $\mathcal{R}_s$ be the set of global shared resources accessed by S_s, and let m_s be the cardinality of $\mathcal{R}_s$.

3.2 Task model

The task model considered in this paper is the 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 arrival of successive jobs of τ_i, C_i is their worst-case execution-time, and D_i is an arrival-relative deadline ($0 < C_i \leq D_i \leq T_i$) before which the execution of a job must be completed. Each task is allowed to access one or more shared 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. It is assumed that all tasks belonging to the same subsystem are assigned unique static priorities and are sorted according to their priorities in the order of increasing priority. Without loss of generality, it is assumed 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. The same assumption is made for the subsystems. The set of shared resources accessed by τ_i is denoted $\{R^i\}$. Let $\mathtt{hp}(i)$ return the set of tasks with priorities higher than that of τ_i and $\mathtt{lp}(i)$ return the set of tasks with priorities lower than that of task τ_i. For each subsystem, we assume that the subsystem period is selected such that $2P_s \leq T_{min}$, where τ_{min} is the task with the shortest period. The motivation for this assumption is that higher P_s will require more CPU resources [36]. In addition, this assumption simplifies the presentation of the paper (evaluating X_s).

3.3 Shared resources

The presented HSF allows for sharing of logical resources between arbitrary tasks, located in arbitrary subsystems, 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. A resource that is used by tasks in more than one subsystem is denoted a *global shared resource*.

To be able to use SRP in a HSF for synchronizing global shared resources, its associated terms resource, system and subsystem ceilings are extended as follows:

- *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_j) for global scheduling. Lower bounds for rc_j and RX_j are defined as $rc_j^{\text{LWB}} = \max\{i | \tau_i \in \mathcal{T}_s \text{ accesses } R_j\}$ and $RX_j^{\text{LWB}} = \max\{s | S_s \text{ accesses } R_j\}$, respectively.

- *System/subsystem ceiling*: The system/subsystem ceilings (SC/sc_s) 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 applies for subsystems from a global scheduling point of view. An attractive property of SRP is that it allows tasks within a subsystem to share a common stack.

4 HSF 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. However, before jumping into the detailed analysis, the periodic processor model is presented, being instrumental in the following analyses.

4.1 The periodic processor model

The notion of real-time virtual processor (resource) model was first introduced by Mok *et al.* [27] 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 [34] 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 periodic virtual processor model $\Gamma(P, Q)$ is defined to characterize the following property:

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

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

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

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

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 2, 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. (2) basically indicates how many periods of a periodic model can overlap the interval of length t, more precisely speaking, the interval of length $t - (P - Q)$. Figure 2 illustrates the intuition of k and how the supply bound function $\texttt{sbf}_\Gamma(t)$ is defined for $k = 3$.

4.2 Local schedulability analysis

Let $\texttt{dbf}_{\texttt{EDF}}(i, t)$ denote the demand bound function of a task τ_i under EDF scheduling [5], i.e.,

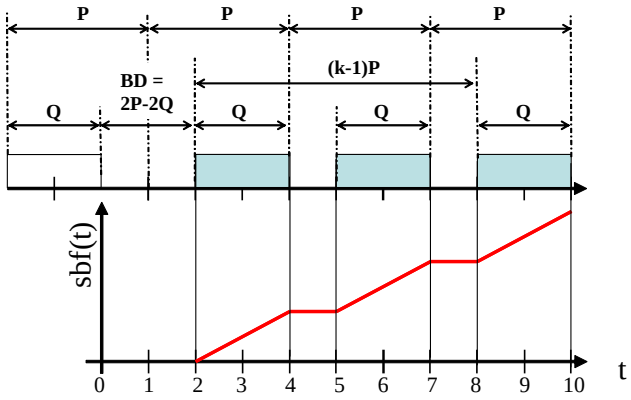

Figure 2. Supply bound function of a periodic virtual processor model $\Gamma(P, Q)$ for $k = 3$.

$$\texttt{dbf}_{\texttt{EDF}}(i, t) = \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i. \tag{3}$$

The local schedulability condition under EDF scheduling is then ([34])

$$\forall t > 0 \quad \sum_{\tau_i \in \Gamma} \texttt{dbf}_{\texttt{EDF}}(i, t) \leq \texttt{sbf}_\Gamma(t), \tag{4}$$

Let $\texttt{dbf}_{\texttt{FP}}(i, t)$ denote the demand bound function of a task τ_i under FPS [21], i.e.,

$$\texttt{dbf}_{\texttt{FP}}(i, t) = C_i + \sum_{\tau_k \in \texttt{hp}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k. \tag{5}$$

The local schedulability analysis under FPS can then easily be extended from the results of [4, 34] as follows:

$$\forall \tau_i, 0 < \exists t \leq D_i \;\; \texttt{dbf}_{\texttt{FP}}(i, t) \leq \texttt{sbf}_\Gamma(t). \tag{6}$$

4.3 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 in order to check the global schedulability.

4.4 Subsystem interface calculation

Using HSF, a subsystem S_s is assigned a 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. (4) and Eq. (6). Hence, $Q_s = \mathsf{calculateBudget}(S_s, P_s)$. The function is a searching function similar to the one presented in [34] and the resulting subsystem timing interface is (P_s, Q_s).

5 Hierarchical scheduling with resource sharing

In this section we take a closer look into the problem of allowing for sharing of logical resources in a HSF, and we discuss and compare a number of possible solutions.

5.1 Detailing the problem

When a task accesses a shared logical resource, all other tasks that want to access the same resource will be blocked until the task that is currently accessing the resource releases it. To achieve a predictable real-time behavior, the waiting time of other tasks that want to access a locked shared resource should be bounded. The traditional synchronization protocols such as SRP and PCP, protocols that are often used with non-hierarchical scheduling, cannot without modification handle the problem of sharing global resources in a HSF. 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 τ_j is executing inside the critical section of the resource R_1 (see Figure 3) due to the following reasons:

1. *Intra subsystem preemption*; a higher priority task τ_k within the same subsystem preempts the task τ_j.
2. *Inter subsystem preemption*; a ready task τ_c that belong 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 SRP and PCP 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 (inter subsystem 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 the problem of synchronization in a HSF. One way to solve this problem is using the protocols SRP and PCP between subsystems such that if a task that belongs to a subsystem locks 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 the SRP and PCP protocols in a HSF is that of budget expiry inside a 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 current 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.

5.2 Supporting sharing of logical resources

Several protocols have been proposed to enable sharing of logical resources in a HSF. These protocols 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 rely on the SRP protocol to synchronize access to a shared resource within a subsystem to solve the problem of intra subsystem preemption, and they also use SRP among subsystems to solve the problem of inter 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 one of, or a combination of, the two approaches of

- adding extra resources to the budget of each subsystem

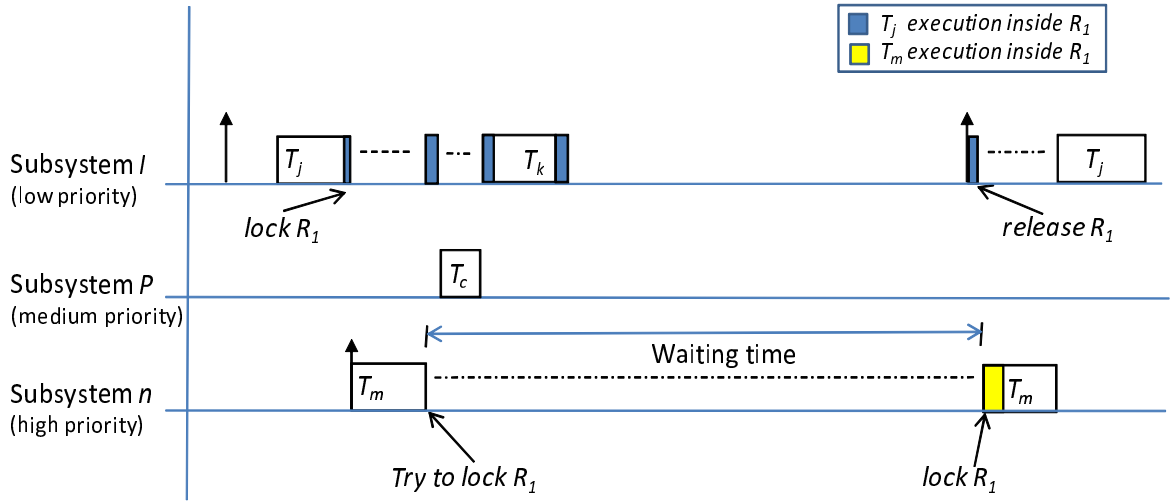

Figure 3. Task preemption while running inside a critical section.

to prevent the budget expiration inside a critical section (applied by HSRP [14]), and/or,

- preventing a task from locking a shared resource if its subsystem does not have enough remaining budget at the time when the task tries to lock the resource (applied by BROE [18] and SIRAP [8]).

The following sections explain details of these two approaches as they are applied in the HSRP, BROE and SIRAP protocols, respectively.

5.2.1 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 an *overrun* mechanism working 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 accesses a global shared resource its priority is increased to the highest local priority, preventing any preemption during the access of a shared resource from other tasks that belong to the same subsystem. SRP is used at 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 have a task accessing that resource. Two versions of overrun mechanisms have been presented; 1) *with payback*, which works as follows: whenever overrun happens in a subsystem S_s, the budget of the subsystem will, in its next execution instant, be decreased by the amount of the overrun time. 2) *without payback*, no further actions will be taken after the event of an overrun. Selecting which of these two mechanisms that gives better results in terms of task response times in the general case is not stated, as it depends on the particular system parameters.

5.2.2 BROE

The Bounded-delay Resource Open Environment (BROE) server [18] extends the Constant Bandwidth Server (CBS) [1] in order to handle the sharing of logical resources in a HSF. The BROE server is suitable for open environments since it allows for each application to be developed and validated independently. For each application, the maximum CPU resource demand is characterized by server speed, delay tolerance (using the bounded-delay resource partition [27]) and resource holding time [10, 19]. 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 are 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 execution of CBS servers) to be able to lock and release the global resource safely.

5.2.3 SIRAP

The Subsystem Integration and Resource Allocation Policy (SIRAP) [8] protocol supports subsystem integration in the presence of globally shared logical resources, and SIRAP can as BROE be used in open environment systems. It uses a periodic resource model to abstract the timing requirements of each subsystem. Each subsystem is characterized by its period, budget and resource holding time, and it is implemented as a 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 problem. The mechanism 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, and if there is insufficient remaining budget, the local scheduler delays the critical section entering of the job until the next subsystem budget replenishment (i.e., the task that wants to enter the critical section will be blocked internally inside its subsystem until the next subsystem budget replenishment).

5.3 Comparing the protocols

This section compares HSRP, BROE, and SIRAP, looking at independency, universality, abstraction, efficiency, and implementation complexity, respectively.

- *Independency*; the local schedulability analysis can be performed independently and therefore the HSF is suitable for use in open environment systems, where applications may be developed and validated independently in different environments.
- *Universality*; means that the scheduling algorithms should not be specific to (a) certain algorithm(s), i.e., it should be possible to employ any scheduling algorithm in the HSF.
- *Timing abstraction*; each subsystem specifies the amount of CPU demand required to schedule all internal tasks through their respective timing interface. The global scheduler schedules all subsystems according to their timing interfaces. Hence, it is required to evaluate the minimum collective CPU demand requirement, which will appear in the subsystem interface, guaranteeing feasibility of the local schedulers of a subsystem.
- *Efficiency*; looking at CPU resource usage, the HSF should use the CPU-resource efficiently by minimizing the collective CPU requirement (i.e., system load) necessary to guarantee the schedulability of an entire framework. By minimizing the system load, more subsystems can be integrated in a single processor, which makes the framework cost-efficient and applicable for a wide domain of applications, e.g., automotive, automation, aerospace and consumer electronics.
- *Implementation complexity*; the implementation of the protocols is a very important issue that should be carefully dealt with, as many protocols lose their efficiency when they are implemented, i.e., the overhead of implementing a protocol may be higher than the resources that the protocol can save.

5.3.1 Independency

The schedulability analysis associated with both SIRAP and BROE support independent development of subsystems, i.e., the schedulability of a subsystem can be analyzed independently, making both protocols suitable for open environments. However, HSRP does not support this feature in the sense that information about other subsystems is needed in order to apply the schedulability analysis for tasks. In [9, 33], we presented schedulability analysis that enables independent development of subsystems using HSRP, by assuming that each subsystem will be supplied with the minimum amount of CPU resources (i.e., considering a worst-case scenario) from the global scheduler.

5.3.2 Universality

The schedulability analysis presented for HSRP in [14] is based on Fixed Priority Scheduling (FPS) for both local and global schedulers, and in [9], Earliest Deadline First (EDF) scheduling is used for both local and global scheduling. For SIRAP the local scheduler is FPS (with some modifications the local scheduler can be EDF as well) and the global scheduler can be either EDF or FPS. The BROE server uses EDF both locally and globally. Note that BROE uses CBS globally, which means that the global scheduler is restricted to EDF.

5.3.3 Timing abstraction

For comparison purposes, for HSRP, we use the local schedulability analysis presented in [9, 33] as the work presented in [14] did not show how to evaluate a subsystem's corresponding subsystem interface; it assumes that the timing interface is given. To specify the subsystem timing interface for both HSRP and SIRAP we use the periodic resource model $\Gamma_s(P_s, Q_s)$ [34], where P_s and Q_s are the subsystem period and budget respectively. Since all protocols use SRP locally[1] then the effect of this should be included in the local schedulability analysis, which can be

[1]The HSRP version presented in [14] did not use SRP locally but it prevents any preemption while a task is accessing a global shared resource.

considered as a blocking time that a task may block other tasks while accessing global shared resources. In addition, SIRAP includes the effect of self blocking, thereby solving the problem of budget expiry inside a critical section, in the local schedulability analysis.

A final remark concerning subsystem interfaces is the level of timing abstraction achieved. For all presented protocols (HSRP, SIRAP and BROE) the values of resource holding times [19], which is the maximum time that a subsystem may lock a resource, should be included in the subsystem interface, to be used in the global schedulability analysis. However, during runtime only SIRAP and BROE require the explicit values of resource holding times in order to check if there is enough remaining budget before locking global shared resources; this is not necessary for HSRP.

5.3.4 Efficiency (in terms of CPU resource usage)

When the SRP protocol is used globally, its effect should be included in the global schedulability. Similar to local schedulability analysis, its effect in the global analysis will be visible as blocking times that a subsystem may block each other subsystem. Note that, the maximum blocking time that a subsystem may block other subsystems will be equal to its maximum resource holding time. For HSRP the effect of allowing a subsystem to overrun, in case of budget expiration inside a critical section, is also added to the global schedulability analysis.

For comparison purposes, 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. Eq. (13) in [9] shows how to evaluate the system load when the global scheduler is EDF, and Eq. (9) in [33] yields the same information for an FPS global scheduler. An efficient protocol is a protocol that produces the lowest system load once used. Comparing between SIRAP and HSRP, it is not possible to prove that one of the protocols is more efficient than the other [6], as such a statement depends on the subsystem parameters as well as on the parameters of the shared resources (even between the two types of overrun mechanisms presented in [14] is not easy to find which of them that requires less system load). BROE seems to be more efficient than the other two, however it is not easy to prove such a property as in [18] the authors did not explain how to evaluate the resource holding times when using a BROE server (the authors left this issue to a future submission), and these values have great effect on the system load (since they are used in the global schedulability analysis as a blocking times).

5.3.5 Implementation complexity

Both SIRAP and HSRP rely on using the periodic server[2] to implement each subsystem that is allowed to execute budget Q_s every period P_s. Implementing the BROE server is done relying on a EDF global scheduler together with a modified version of CBS. Comparing the two types of servers, the implementation of the periodic server is easier than the implementation of the CBS server (CBS has more states and the server change its state when it passes certain instances in time). One more thing in the comparison is that the CBS server should update its parameters (deadline, virtual time and reactivation time etc.) and it continuously changes the state of the server (contending, non-contending, suspended, inactive, blocked). On the other hand, using periodic server, the global scheduler can update the server parameters (remaining budget) and change the state of the server (ready, non-ready, blocked).

Comparing SIRAP and HSRP using the periodic server, the SIRAP protocol provides better isolation between the local and the global scheduler. The reason for this is that when using HSRP, (1) the local scheduler should inform the global scheduler about events such as overrun, in order to keep the server executing even when its budget is consumed, and (2) the local scheduler should inform the global scheduler when its task release a global resource in order to remove the server from the execution when executing out of budget. While for SIRAP, no such communication between the global and local schedulers is required since the problem of the budget expiration inside the critical section of a global shared resource is solved locally.

6 Detailed analysis of SIRAP

This section present a detailed analysis of the SIRAP protocol, starting with the local schedulability analysis, followed by how to derive various parameters, and finishing with the global schedulability analysis.

6.1 Local schedulability analysis

The local schedulability analysis under FPS is as follows [4, 34]:

$$\forall \tau_i \; \exists t : 0 < t \leq D_i, \;\; \texttt{rbf}_{\texttt{FP}}(i,t) \leq \texttt{sbf}_s(t), \qquad (7)$$

where $\texttt{sbf}_s(t)$ is the *supply bound function* based on the periodic resource model presented in [34] that computes the minimum possible CPU supply to S_s for every interval length t (see Section 4 for details), and $\texttt{rbf}_{\texttt{FP}}(i,t)$ denotes the *request bound function* of a task τ_i. Note that, for

[2] A periodic server is a server that works/behaves similar as a periodic task.

Eq. (7), t can be selected within a finite set of scheduling points [24].

The request bound function $\texttt{rbf}_{\texttt{FP}}(i,t)$ of a task τ_i is given by:

$$\texttt{rbf}_{\texttt{FP}}(i,t) = C_i + I_S(i) + I_H(i,t) + I_L(i), \quad (8)$$

$$I_S(i) = \sum_{R_k \in \{R^i\}} X_{i,k}, \quad (9)$$

$$I_H(i,t) = \sum_{\tau_j \in \texttt{hp}(i)} \left\lceil \frac{t}{T_j} \right\rceil (C_j + \sum_{R_k \in \{R^j\}} X_{j,k}), \quad (10)$$

$$I_L(i) = \max_{\tau_f \in \texttt{lp}(i)} (2 \cdot \max_{\forall R_j | rc_j \geq i} (X_{f,j})), \quad (11)$$

where $I_S(i)$ is the self blocking of task τ_i, $I_H(i,t)$ is the interference from tasks with priority higher than that of τ_i, and $I_L(i)$ is the interference from tasks, with priority lower than that of τ_i, that access shared resources.

6.2 Calculating X_s

Given a subsystem S_s, its critical section execution time X_s represents a worst-case CPU demand that internal tasks of S_s may collectively request while executing inside any critical section. Note that any task τ_i accessing a resource R_j can be preempted by tasks with priority higher than rc_j. Note that SIRAP prevents subsystem budget expiration inside a critical section of a global shared resource. When a task experiences self-blocking during a subsystem budget period it is guaranteed access to the resource during the next period. A sufficient condition to provide this guarantee is

$$Q_s \geq X_s. \quad (12)$$

We now derive $X_s \leq Q_s < P_s$ and since we assume that $2P_s \leq T_{min}$ then all tasks that are allowed to preempt while τ_i accesses R_j will be activated at most one time from the time that self blocking happens until the end of the next subsystem period. Then $X_{i,j}$ can be computed as follows,

$$X_{i,j} = c_{i,j} + \sum_{k=rc_j+1}^{n_s} C_k. \quad (13)$$

Let $X_j = \max\{X_{i,j} | \text{ for all } \tau_i \in \mathcal{T}_s \text{ accessing } R_j\}$, then $X_s = \max\{X_j | \text{ for all } R_j \in \mathcal{R}_s\}$.

6.3 Internal resource ceiling

Looking at Eq. (13), assigning internal resource ceilings according to SRP may make the value of X_s very high which causes the subsystem to require more CPU resources. One way to handle this problem is by preventing the preemption inside the subsystem when a task is accessing a shared resource as proposed in [14] so $X_{i,j} = c_{i,j}$. It can be implemented using SRP by assigning the resource ceiling of all resources equal to the maximum task priority $rc_j = n_s$ where n_s is the task ID number of the highest priority task. However, Bertogna *et al.* [10] showed that preventing preemption while accessing a global shared resource may violate the local schedulability of the subsystem and proposed an algorithm based on increasing the ceiling of all resources in steps as much as possible without violating the local schedulability. Finally, Shin *et al.* [33] showed that there is a tradeoff between decreasing the value of X_s and the minimum subsystem budget required to guarantee the schedulability of the subsystem.

The result of this paper does not depend on any of the discussed methods to set the internal resource ceiling. So we assume that the internal ceiling of resource R_j can be selected within the following range $n_s \geq rc_j \geq rc_j^{\text{LWB}}$.

6.4 Global schedulability analysis

The general condition for global schedulability is

$$\forall S_s \; \exists t : 0 < t \leq P_s, \;\; \texttt{RBF}_s(t) + B_s \leq t \quad (14)$$

where B_s is the maximum blocking imposed to a subsystem S_s, when it is blocked by lower-priority subsystems (suppose that S_j imposes the maximum blocking on S_s then $B_s = X_j$). Eq. (15) is used to evaluate $\texttt{RBF}_s(t)$ for the SIRAP protocol:

$$\texttt{RBF}_s(t) = Q_s + \sum_{S_k \in \texttt{HP}(s)} \left\lceil \frac{t}{P_k} \right\rceil \cdot Q_k \quad (15)$$

where $\texttt{HP(s)}$ is the set of subsystems with priority higher than than of subsystem S_s. Note that the way of calculating $\texttt{RBF}_s(t)$ depends on the synchronization protocol.

7 Adaptive and reconfigurable systems

The HSF is very useful when it comes to the implementation of operating system support for adaptability and reconfigurability needed in dynamic open systems, where applications (one or more subsystems) may be allowed to join and/or leave the system during runtime. In allowing such functionality, a proper *admission control* (AC) must be provided. Also, the HSF allows for a convenient implementation of quality of service management policies, allowing for a dynamic allocation of resources to subsystems.

The admission control (AC) applies one or more algorithms to determine if a new application (consisting of one or multiple subsystems) can be allowed to join the system and start execution (admission) without violating the requirements of the already existing applications (or the requirements of the whole system). The decision of the AC

depends on the state of the system resources and the resources required by the new application asking for admission. If there are enough resources available in the system, the application will be admitted; otherwise the application will be rejected.

In general, since the AC uses online algorithms the complexity and overhead of implementing these algorithms should be very low for several reasons, such as maintaining scalability of the AC and minimizing its interference on the system. Hence, one objective in designing the AC concerns keeping the input to these algorithms as simple as possible, e.g., the resource requirement for each individual task could be abstracted to the subsystem level. Another objective concerns minimizing interference between the AC and the system online, making it desirable to perform as much work as possible offline.

7.1 Admission of resources

The *resources* considered by the AC may include, but are not limited to, *CPU* resources, *memory* resources, *network* resource and *energy* resources. Initially, we have been focusing on CPU and network resources, and are now also looking at memory resources.

7.1.1 CPU resources

When using the HSF, traditional schedulability algorithms can be used in order to check the CPU resources, e.g., by using the global schedulability test in the HSF [34, 35]. This algorithm depends on the type of system level scheduler used, e.g., EDF, FPS, etc. The AC checks the schedulability condition of the system including the new subsystem. If the system is still schedulable, the new subsystem will pass this test; otherwise the new application will be rejected. In using this test, it is guaranteed that all hard real time requirements will be met. The input to the algorithm is the subsystem interface (subsystem budget and period) of each running subsystem together with the interface of the new subsystem. Note that these parameters are evaluated and determined during the development of the subsystem (offline).

7.1.2 Memory resources

When allowing for a new application to enter the system, the AC should guarantee that there is sufficient memory space to be used by all subsystems. Otherwise, unexpected problems may happen during run time. In a similar way as for CPU resources, the maximum memory space required by each subsystem is evaluated during its development. In the AC test, a simple algorithm can be used to check if there is enough memory space available in the system, by checking if the summation of the maximum memory space for all subsystems is less than or equal to the memory space provided by the platform. Such an algorithm is very simple; however, the accuracy of the result is not high as all applications will not likely need their specified maximum memory space at the same time. Higher efficiency can be achieved by the usage of algorithms such as the approximated algorithm presented in [11].

7.1.3 Energy resources

Most of the modern processors support changing the frequency and voltage of the CPU during runtime, in controlling the CPU's power consumption. The HSF can use this feature to select the lowest frequency/voltage that guarantees the hard real time requirements of the system. Decreasing the frequency of a processor will increase the worst-case execution time (WCET) of its tasks. In doing this, more CPU resources should be allocated to subsystems in order to ensure that all hard real time tasks will meet their deadlines. Looking at the HSF, if predefined levels of frequencies are used, we can find a subsystem interface for each frequency level for all subsystems. Then, during runtime, the AC will make sure that the processor is working with the lowest frequency keeping the schedulability of the current set of subsystems. When it is required to add a new subsystem, the AC will check the schedulability condition with the current processor frequency; if the system is deemed not schedulable, then the AC will try with higher frequencies. When a subsystem is removed from the system, the AC will try to reduce the frequency of the CPU in order to reduce its power consumption.

7.1.4 Network resources

This type of resource is important in distributed systems where there typically exist communications between nodes in the network. The network resource is different from the other resources previously described in the sense that the network resource is shared by all nodes, while the other resources are local to each node. When the AC is faced with a request for adding a subsystem, it should check if the communications requirements will be met, i.e., check if all important messages will be delivered in proper time [28]. Selecting an algorithm that checks this resource is more complex as there are many different requirements, communication protocols, network types, etc. Covering all these aspects might not be necessary but as an illustration consider a simple algorithm which relies on the communication bandwidth. During the development of each subsystem, their maximum communication bandwidth requirements should be evaluated such that the AC can use it in order to check if the summation of required bandwidth for all subsystems is less than 100%.

7.2 Approaches to admission control in distributed systems

Implementing the AC in distributed systems is more complex than doing so for a single CPU. The main reason for this is that the information needed by the AC algorithms must be consistent. For example, when using the network resources, awareness of all network users must be maintained by the AC, and these users are typically located on many nodes throughout the distributed system. Commonly, information on the current state is kept at one place, managing the information needed by the AC. Also, when an application consists of more than one subsystem, and these subsystems are located at different nodes, all these subsystems should pass the AC tests before admitting the application.

In designing the AC we have identified 3 different approaches based on where the AC test will be implemented.

- A special **Master Node (MA)** will implement the AC tests of all resources in the system. Only the MA will have information about resources in the system. Hence, consistency is not a problem, it is easy to determine the order between AC requests, and the AC does not have to contact multiple nodes in getting the current system state as only a single AC request to the MA is needed. On the downside the MA is a single system level point of failure.

- **All Nodes (AN)** will implement the AC tests of all resources in the system. Each node should have the consistent information of all resources that are used by the system. Hence, in this fully distributed approach the AC test can be performed without having to communicate with other nodes. Also, the approach is tolerant to failures. On the downside, consistency must be maintained between all nodes, and ordering is more complex compared with MA. Also, more memory is needed in maintaining all resource state replicas.

- There will be **One Node (ON)** implementing the AC test of each resource in the system. Each node will maintain information about the resources that is responsible for. Hence, there will be no issues with respect to data consistencies between replicas of the same information, but a single AC request might have to communicate with a number of nodes in order to get a valid system state. Also, ordering among AC requests must be solved, and each resource owner will be a single resource level point of failure.

The characteristics of the above three approaches are outlined in Table 1, where L is the number of resources and n is the number of nodes.

	MA	**AN**	**ON**
No. messages	Single	None	Multiple
Consistency	No problem	Complex	No problem
Ordering	Simple	Complex	Complex
Fault tolerance	Single point of failure (system level)	Fault tolerant	Single point of failure (resource level)
Memory	$L * n$	$L * n * n$	$L * n$

Table 1. Properties of the 3 AC strategies.

8 Implementation

In this section we compare and discuss some issues related to the implementation of both SIRAP and HSRP. The corresponding implementation is based on our previous implementation of the Hierarchical Scheduling Framework (HSF) [7] in the VxWorks operating system. In the implementation presented in [7], we have used periodic servers to implement subsystems assuming that tasks are independent, i.e., no sharing of logical resources is allowed.

The implementation of HSF supports both FPS and EDF at the local and global scheduler levels. To support synchronization between tasks (or subsystems) when accessing global shared resources, advances in the implementation made since [7] include the implementation of the SRP protocol. Both schedulers (local and global) are activated periodically according to task/server parameters and we have used a timer to trigger the schedulers. The schedulers use a queue to save the time events called the *Time Event Queue* (TEQ) (one TEQ to schedule the servers and one for each subsystem to schedule its internal tasks). The TEQ is a priority queue that consists of sorted events (period, deadline), according to absolute times, which are updated at each scheduler invocation. At each global/local scheduler activation, the following is done:

- Handle the scheduling event (period or deadline event).
- Update the occurred event in the TEQ.
- Fetch the nearest event (from TEQ) and set next scheduler expiration to timer.
- Update the SYSTEM TIME.

Figure 4 illustrates how the scheduler is triggered. After the scheduler has taken care of the current scheduling event (first node in the TEQ), this event is updated and put in the TEQ. Triggering the scheduler to run at time 200 is done by setting a timer to the value of the next event's absolute time subtracted with the current SYSTEM TIME since the timer input should be in relative time. The schedulers SYSTEM TIME are discrete clocks used only by the schedulers (it has nothing to do with the OS system clock). The SYSTEM TIME is set to 200 at scheduling event 100, so the SYSTEM

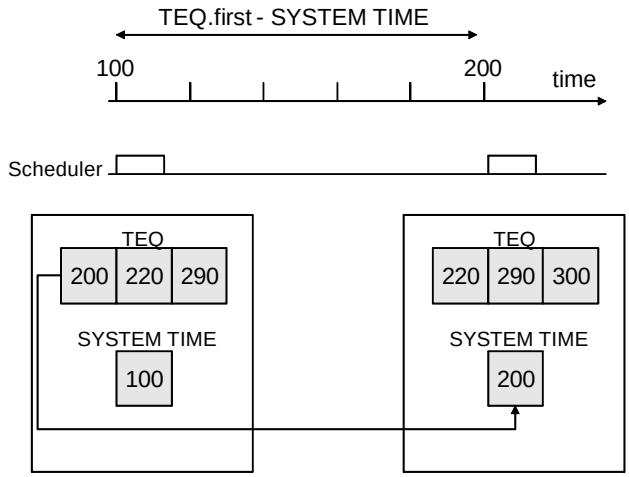

Figure 4. Scheduler execution.

TIME will equal to 200 until next event (at absolute time 200).

As mentioned before, SRP is used by both local and global schedulers when accessing a global shared resource, so both HSRP and SIRAP require an SRP implementation.

When a task wants to access a global shared resource, it first calls the function `lock`, and the task releases the shared resource when it calls the function `unlock`. The `lock/unlock` function has a similar implementation in both SIRAP and HSRP. In order to manipulate or read the VxWorks ready queue, resource queue or task blocking queue (which are necessary to implement the SRP protocol) in a safe manner, the `lock/unlock` must use interrupt disable, in very short time, during these operations. Locking out interrupts is done in the `lock/unlock` function to create mutual exclusion between the synchronization protocol and the schedulers (which are the only ones who use these data structures).

The main difference in the implementations of SIRAP and HSRP is related to how they avoid budget expiration inside a critical section. In SIRAP, we postpone the execution of the critical section to the next server period if there is not enough remaining budget to execute and release the critical section before the budget expiration. Implementing this is done by blocking the task that tried to enter the critical section and set the subsystem ceiling equal to the internal ceiling of the resources that the task tried to access. Then letting the server continue to execute and queue all the blocked tasks (which are blocked by the use of SRP protocol) that have release times during the self-blocking phase (from the time that the task try to lock a resource and becomes blocked until the time at which the budget of the subsystem expires). Hence, only the tasks that have priority higher than the subsystem ceiling are allowed to execute during the self-blocking phase.

HSRP will prolong the server budget, when the budget expires inside a critical section, by letting the protocol manipulate (increase) the remaining budget in the server. The budget will reset (equals to zero) when the critical section execution is finished, which means that the global scheduler must be informed (called) about this from task level in order to remove the server from the execution. This approach is not good because of two reasons:

Firstly, the scheduler needs the current absolute time in order to set a correct next expiration time and the current SYSTEM TIME is not correct since the activation of the global scheduler is not in response to time event. Time stamping must be used to get the correct current absolute time. The problem is that using timestamp to evaluate the absolute SYSTEM TIME may add drift depending on the resolution of this facility. If the timestamp value has a error margin of 1 μs, then the scheduler could set its next expiration 1 μs too early or too late which will have a big impact on the scheduling and might cause a drift. Normally, the interrupt facility will start the scheduler in the amount of time equal to the expiration time that was set in the last scheduler invocation.

Secondly, we break the isolation between the local and global schedulers by letting the local level directly call the global entity. Subsystems must not be aware of the global scheduling if subsystem development should be done in isolation.

To enable the payback approach in HSRP (payback means that the overrun time will be paid back in the next execution instant), the overrun time is measured (by using timestamp) and the global scheduler decrease the budget of the server by the amount of overrun in the next execution instant. Note that a server can be preempted during the execution in overrun phase which should be taken into account while measuring the overrun time.

9 Summary

In this paper we have presented the Hierarchical Scheduling Framework (HSF) together with a brief overview of related approaches. Particular focus have been put on techniques to allow for synchronization in HSFs, and a number of synchronization protocols have been compared. We have discussed some considerations regarding using HSFs in dynamic, adaptable and reconfigurable systems, highlighting various approaches for implementing admission control in distributed systems. Finally, we have presented experiences from our implementation of a HSF in VxWorks, comparing issues related to the implementation of several synchronization protocols.

Some pointers to further reading on related topics include:

- Our original paper on the SIRAP synchronization protocol allowing for mutually exclusive sharing of log-

ical resources in two-level hierarchically scheduled real-time systems for single processors [8]. In this paper, we have formally proven key features of the protocol, such as bounds on delays for accessing shared logical resources, and we have developed schedulability analysis for tasks executing in the subsystems; allowing for hard real-time applications to use the protocol.

- Techniques to handle budget expiration in hierarchical scheduling [9]. One problem that becomes apparent when allowing for synchronization in hierarchical scheduling is how potential overruns shall be coped with. We have analyzed three methods to handle overruns due to mutual exclusive resource sharing between subsystems. For each one of these three overrun methods corresponding scheduling algorithms and associated schedulability analysis have been presented together with analysis that shows under what circumstances one or the other is preferred.

- We have developed algorithms for (1) system-independent synthesis of timing-interfaces for subsystems and (2) system-level selection of interfaces to minimize CPU load. We show that the use of shared mutually exclusive logical 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. We have presented a methodology where such a tradeoff can be effectively explored [33]. We have implemented a hierarchical scheduling framework in a commercial operating system [7], together with several synchronization protocols for mutual exclusion of shared logical resources.

Currently we are working on issues related to multiprocessors and implementation in Linux.

References

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

[2] 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.

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

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

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

[6] M. Behnam, T. Nolte, M. Åsberg, and R. Bril. Overrun and skipping in hierarchically scheduled real-time systems. In *Proceedings of the 15th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'09)*, Beijing, China, August 2009.

[7] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril. Towards hierarchical scheduling on top of VxWorks. In *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, Prague, Czech Republic, July 2008.

[8] 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.

[9] M. Behnam, I. Shin, T. Nolte, and M. 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)*, Hamburg, Germany, September 2008.

[10] 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'07)*, pages 1–8, Long Beach, CA, USA, March 2007.

[11] M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin. Safe shared stack bounds in systems with offsets and precedences. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-221/2008-1-SE, Mälardalen University, January 2008.

[12] B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume LNCS-3436. Springer, 2005.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[27] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, Taipei, Taiwan ROC, May 2001.

[28] T. Nolte. *Share-Driven Scheduling of Embedded Networks*. PhD thesis, Department of Computer and Science and Electronics, Mälardalen University, Sweden, May 2006.

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

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

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

[32] 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.

[33] I. Shin, M. Behnam, T. Nolte, and M. Nolin. Synthesis of optimal interfaces for hierarchical scheduling with resources. In *Proceedings of the 29th IEEE International Real-Time Systems Symposium (RTSS'08)*, Barcelona, Spain, December 2008.

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

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

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