Dependable resource sharing for compositional real-time systems

17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications

Dependable resource sharing for compositional real-time systems

Martijn M. H. P. van den Heuvel, Reinder J. Bril and Johan J. Lukkien
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

Abstract

Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating temporal isolation between components that need to be integrated on a single shared processor. In the presence of shared resources, however, temporal isolation may break when one of the accessing components executes longer than specified during global resource access. The ability to confine such temporal faults makes the HSF more dependable. As a solution we propose a stack-resource-policy (SRP)-based synchronization protocol for HSFs, named Hierarchical Synchronization protocol with Temporal Protection (HSTP).

When a component exceeds its specified critical-section length, HSTP enforces a component to self-donate its own budget to accelerate the resource release. In addition, a component that blocks on a locked resource may donate budget. The schedulability of those components that are independent of the locked resource is unaffected. HSTP efficiently limits the propagation of temporal faults to resource-sharing components by disabling local preemptions in a component during resource access. We finally show that HSTP is SRP-compliant and applies to existing synchronization protocols for HSFs[1].

1. Introduction

Many real-time embedded systems implement increasingly complex and safety-critical functionality while their time to market and cost is continuously under pressure. This has resulted in standardized component-based software architectures, e.g. the AUTomotive Open System ARchitecture (AUTOSAR), where each component can be analysed and certified independently of its performance in an integrated system. Hierarchical scheduling frameworks (HSFs) have been investigated as a paradigm for facilitating such a decoupling [1] of development of individual components from their integration. HSFs provide temporal isolation between components by allocating a *budget* to each component. A component that is validated to meet its timing constraints when executing in isolation will therefore continue meeting its timing constraints after integration or admission on a shared uniprocessor platform.

1. The work in this paper is supported by the Dutch HTAS-VERIFIED project, see http://www.htas.nl/index.php?pid=154.

An HSF without further resource sharing is unrealistic, however, since components may for example use operating system services, memory mapped devices and shared communication devices which require mutually exclusive access. Extending an HSF with such support makes it possible to share logical resources between arbitrary tasks, which are located in arbitrary components, in a mutually exclusive manner. A resource that is used in more than one component is denoted as a *global shared resource*. A resource that is only shared by tasks within a single component is a *local shared resource*. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of its budget, excessive blocking periods can occur which may hamper the correct timeliness of other components [2].

To accommodate resource sharing between components, three synchronization protocols [3], [4], [5] have been proposed based on the *stack resource policy* (SRP) [6]. Each of these protocols describes a run-time mechanism to handle the depletion of a component's budget during global resource access. In short, two general approaches are proposed: (*i*) *self-blocking* before accessing a shared resource when the remaining budget is insufficient to complete a critical section [4], [5] or (*ii*) *overrun* the budget until the critical section ends [3]. However, when a task exceeds its specified worst-case critical-section length, i.e. it *misbehaves* during global resource access, temporal isolation between components is no longer guaranteed. The protocols in [3], [4], [5] therefore break the temporal encapsulation and fault-containment properties of an HSF without the presence of complementary protection.

A common practice to temporal protection is based on watchdog timers, which (*i*) trigger the termination of a misbehaving task, (*ii*) release all its locked resources and (*iii*) call an error handler to execute a *roll-back strategy* [7]. For some shared resources, e.g. network devices, it may be advantageous to continue a critical section, because interrupting and resetting a busy device can be time consuming. In addition, an eventual error handler needs to be constrained, so that it does not interfere with independent components.

A solution to confine temporal faults to those components that share global resources is considered in [8]. Each task is assigned a dedicated budget per global resource access and this budget is synchronous with the period of that task. After depletion of this budget, a critical section is discontinued until its budget replenishes and the task will therefore miss

1533-2306/11 $26.00 © 2011 IEEE
DOI 10.1109/RTCSA.2011.29

its deadline. To improve reliability, they propose a donation mechanism for tasks that encounter a locked resource, so that a critical section can continue and both the blocking as well as the resource-locking task may still meet their deadlines. However, in [8] they allow only a single task per component.

The first problem is to limit the propagation of temporal faults in HSFs, where multiple concurrent tasks are allocated a shared budget, to those components that share a global resource. However, when a critical-section exceeds its specified length, a component may nevertheless have remaining budget apart from the budget assigned to that critical section. The second problem is to allocate these unused budgets to accelerate a resource release before a resource-sharing component blocks on the locked resource. We aim to increase the reliability of resource-sharing components, while preserving the schedulability of independent components, by self-donations of the remaining budget of a resource-locking component itself and, next, by budget donations from others to critical sections.

Contributions. To achieve temporal isolation between components, even when resource-sharing components misbehave, we propose a modified SRP-based synchronization protocol, named *Hierarchical Synchronization protocol with Temporal Protection* (HSTP). It supports fixed-priority as well as earliest-deadline-first (EDF) scheduled systems and it complements existing synchronization protocols [3], [4], [5] for HSFs. We efficiently achieve fault-containment by disabling preemptions of other tasks within the same component during global resource access. Secondly, we allow a cascaded continuation of a critical section via self-donations as long as a component has remaining budget, or via budget donations from dependent components. Thirdly, we present HSTP's analysis and show that sufficiently short critical sections can execute with local preemptions disabled and preserve system schedulability. Finally, we evaluate HSTP's implementation in a real-time operating system.

Organization. The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 presents our system model. Section 4 presents our dependable resource-sharing protocol, HSTP, including a donation mechanism that enhances the reliability of inter-dependent components. Section 5 presents HSTP's corresponding schedulability analysis. Section 6 discusses HSTP's implementation and investigates its corresponding system overhead. Finally, Section 7 concludes this paper.

2. Related work

Our basic idea is to use two-level SRP to arbitrate access to global resources, similar as [3], [4], [5]. In literature several alternatives are presented to accommodate task communication in reservation-based systems. De Niz et al. [8] support resource sharing between reservations based on the immediate priority ceiling protocol (IPCP) [9] in their fixed-priority preemptively scheduled (FPPS) Linux/RK resource kernel and use a run-time mechanism based on resource containers [10] for temporal protection against misbehaving tasks. Steinberg et al. [11] showed that these resource containers are expensive and efficiently implemented a capacity-reserve donation protocol to solve the problem of priority inversion for tasks scheduled in a fixed-priority reservation-based system. A similar approach is described in [12] for EDF-based systems and termed bandwidth-inheritance (BWI). BWI regulates resource access between tasks that each have their dedicated budget. It works similar to the priority-inheritance protocol [9], i.e. when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. BWI does not require a-priori knowledge of tasks, i.e. precalculated ceilings are unnecessary. BWI has been extended with the *Clearing Fund Protocol* (CFP) [13], which makes a task pay back its inherited bandwidth, if necessary. All these approaches assume a one-to-one mapping from tasks to budgets, and inherently only have a single scheduling level.

In HSFs a group of concurrent tasks, forming a component, are allocated a budget [14]. A prerequisite to enable independent analysis of interacting components and their integration is the knowledge of which resources a task will access [4], [5], [15]. When a task accesses a global shared resource, one needs to consider the priority inversion between components as well as local priority inversion between tasks within the component. To *prevent budget depletion* during global resource access, three synchronization protocols have been proposed based on SRP [6], i.e. HSRP [3], SIRAP [4] and BROE [5]. Although HSRP [3] originally does not integrate into HSFs due to the lacking support for independent analysis of components, Behnam et al. [15] lifted this limitation. However, these three protocols, including their implementations in [16], [17], assume that components respect their timing contract with respect to global resource sharing. In this paper we smooth and limit the unpredictable interferences caused by contract violations to the components that share the global resource.

3. Real-time scheduling model

We consider a two-level HSF using the periodic resource model [1] to specify guaranteed processor allocations to components. The global scheduler and each individual component may apply a different scheduling algorithm. As scheduling algorithms we consider EDF, an optimal dynamic uniprocessor scheduling algorithm, and the deadline-monotonic (DM) algorithm, an optimal fixed-priority uniprocessor scheduling algorithm. We use an SRP-based synchronization protocol to arbitrate mutually exclusive access to global shared resources.

3.1. Compositional model

A system contains a set $\mathcal{R}$ of M global logical resources R_1, R_2, ..., R_M, a set $\mathcal{C}$ of N components $C_1, C_2, \ldots, C_N$, a set $\mathcal{B}$ of N budgets for which we assume a periodic resource model [1], and a single shared processor. Each component C_s has

a dedicated budget which specifies its periodically guaranteed fraction of the processor. The remainder of this paper leaves budgets implicit, i.e. the timing characteristics of budgets are taken care of in the description of components.

The timing characteristics of a component C_s are specified by means of a triple $< P_s, Q_s, \mathcal{X}_s >$, where $P_s \in \mathbb{R}^+$ denotes its period, $Q_s \in \mathbb{R}^+$ its budget, and $\mathcal{X}_s$ the set of maximum access times to global resources. The maximum value in $\mathcal{X}_s$ is denoted by X_s, where $0 < Q_s + X_s \leq P_s$. The set $\mathcal{R}_s$ denotes the subset of M_s global resources accessed by component C_s. The maximum time that a component C_s executes while accessing resource $R_l \in \mathcal{R}_s$ is denoted by X_{sl}, where $X_{sl} \in \mathbb{R}^+ \cup \{0\}$ and $X_{sl} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$.

3.1.1. Processor supply. The *processor supply* refers to the amount of processor allocation that a component C_s can provide to its workload. The supply bound function $\mathtt{sbf}_{\Gamma_s}(t)$ of the periodic resource model $\Gamma_s(P_s, Q_s)$, that computes the minimum supply for any interval of length t, is given by [1]:

$$\mathtt{sbf}_{\Gamma_s}(t) = \begin{cases} t - (k+1)(P_s - Q_s) & \text{if } t \in V^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \quad (1)$$

where $k = \max\left(\left\lceil \big(t - (P_s - Q_s)\big)/P_s \right\rceil, 1\right)$ and $V^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s, (k+1)P_s - Q_s]$. The longest interval a component may receive no processor supply is named the *blackout duration*, BD_s, i.e. $BD_s = 2(P_s - Q_s)$.

3.1.2. Task model. Each component C_s contains a set $\mathcal{T}_s$ of n_s sporadic tasks τ_{s1}, τ_{s2}, ..., τ_{sn_s}. Timing characteristics of a task $\tau_{si} \in \mathcal{T}_s$ are specified by means of a triple $< T_{si}, E_{si}, D_{si} >$, where $T_{si} \in \mathbb{R}^+$ denotes its minimum inter-arrival time, $E_{si} \in \mathbb{R}^+$ its worst-case computation time, $D_{si} \in \mathbb{R}^+$ its (relative) deadline, where $0 < E_{si} \leq D_{si} \leq T_{si}$. We assume that period P_s of component C_s is selected such that $2P_s \leq T_{si} (\forall \tau_{si} \in \mathcal{T}_s)$, because this efficiently assigns a budget to component C_s [1]. The worst-case execution time of task τ_{si} within a critical section accessing R_l is denoted c_{sil}, where $c_{sil} \in \mathbb{R}^+ \cup \{0\}$, $E_{si} \geq c_{sil}$ and $c_{sil} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$. For notational convenience we assume that tasks (and components) are given in deadline-monotonic order, i.e. τ_{s1} has the smallest deadline and τ_{sn_s} the largest.

3.2. Synchronization protocol

Traditional synchronization protocols such as PCP [9] and SRP [6] can be used for *local* resource sharing in HSFs [18]. This paper focuses on arbitrating *global* shared resources using SRP. To be able to use SRP in an HSF for synchronizing global resources, its associated ceiling terms need to be extended and excessive blocking must be prevented.

3.2.1. Preemption levels. Each task τ_{si} has a static preemption level equal to $\pi_{si} = 1/D_{si}$. Similarly, a component has a preemption level equal to $\Pi_s = 1/P_s$, where period P_s serves as a relative deadline. If components (or tasks) have the same calculated preemption level, then the smallest index determines the highest preemption level.

3.2.2. Resource ceilings. With every global resource R_l two types of resource ceilings are associated; a *global* resource ceiling RC_l for global scheduling and a *local* resource ceiling rc_{sl} for local scheduling. These ceilings are statically calculated values, which are defined as the highest preemption level of any component or task that shares the resource. According to SRP, these ceilings are defined as:

$$RC_l = \max(\Pi_N, \max\{\Pi_s \mid R_l \in \mathcal{R}_s\}), \quad (2)$$

$$rc_{sl} = \max(\pi_{sn_s}, \max\{\pi_{si} \mid c_{sil} > 0\}). \quad (3)$$

We use the outermost $\max$ in (2) and (3) to define RC_l and rc_{sl} in those situations where no component or task uses R_l.

3.2.3. System and component ceilings. The system and component ceilings are dynamic parameters that change during execution. The system ceiling is equal to the highest global resource ceiling of a currently locked resource in the system. Similarly, the component ceiling is equal to the highest local resource ceiling of a currently locked resource within a component. Under SRP a task can only preempt the currently executing task if its preemption level is higher than its component ceiling. A similar condition for preemption holds for components.

3.2.4. Prevent excessive blocking. HSRP [3] uses an overrun mechanism [15] when a budget depletes during a critical section. If a task $\tau_{si} \in \mathcal{T}_s$ has locked a global resource when its component's budget Q_s depletes, then component C_s can continue its execution until task τ_{si} releases the resource. To distinguish this additional amount of required budget from the *normal budget* Q_s, we refer to X_s as an *overrun budget*. These budget overruns cannot take place across replenishment boundaries, i.e. the analysis guarantees $Q_s + X_s$ processor time before the relative deadline P_s of component C_s [3], [15].

SIRAP [4] uses a self-blocking approach to prevent budget depletion inside a critical section. If a task τ_{si} wants to enter a critical section, it enters the critical section at the earliest time instant so that it can complete the critical section before the component's budget depletes. If the remaining budget is insufficient to lock and release a resource R_l before depletion, then (*i*) the task blocks itself until budget replenishment and (*ii*) the component ceiling is raised to prevent tasks $\tau_{sj} \in \mathcal{T}_s$ with a preemption level lower than the local ceiling rc_{sl} to execute until the requested critical section has been finished.

BROE [5] uses an other self-blocking variant than SIRAP uses. Contrary to SIRAP and HSRP, BROE only works with a global EDF scheduler. Its major advantage is that when the remaining budget of a component is insufficient to complete a critical section, it discards this remainder without violating the periodic resource supply [1]. BROE does therefore not waste processor resources during self-blocking and it is also unnecessary to allocate overrun budgets to components.

In this paper we ignore the relative strengths of the mechanisms presented by the protocols in [3], [4], [5], [15]. We focus on mechanisms to extend these existing protocols with

dependability attributes and merely investigate their relative complexity with respect to our mechanisms.

4. Dependable resource sharing

Dependability encompasses many quality attributes [7], i.e. amongst others: safety, integrity, reliability, availability and robustness. Temporal faults can have catastrophic consequences, making the system *unsafe*. These temporal faults may cause improper system alterations, e.g. due to unexpectedly long blocking or an inconsistent state of a resource. Hence, it affects system *integrity*. Without any protection a self-blocking approach [4], [5] may miss its purpose under erroneous circumstances, i.e. a task still overruns its budget to complete its critical section. Even an overrun approach [3], [15] must guarantee a maximum duration of the overrun situation. Otherwise, overruns can hamper temporal isolation and resource *availability* to other components due to unpredictable blocking effects. A straightforward implementation of the overrun mechanism, e.g. as implemented in the ERIKA kernel [19], where a task is allowed to indefinitely overrun its budget as long as it locks a resource, is therefore *unreliable*. The extent to which a system tolerates such unforeseen interferences defines its *robustness*.

4.1. Resource monitoring and enforcement

A common approach to ensure temporal isolation and prevent propagation of temporal faults within the system is to group tasks that share resources into a single component [18]. However, this might be too restrictive and leading to large, incoherent component designs, which violates the principle of HSFs to independently develop components. Since a component defines a coherent piece of functionality, a task that accesses a global shared resource is critical with respect to all other tasks in the same component.

To guarantee temporal isolation between components, the system must *monitor* and *enforce* the length of a global critical section to prevent a malicious task to execute longer in a critical section than assumed during system analysis [8]. Otherwise such a misbehaving task may increase blocking to components with a higher preemption level, so that even independent components may suffer, as shown in Figure 1.

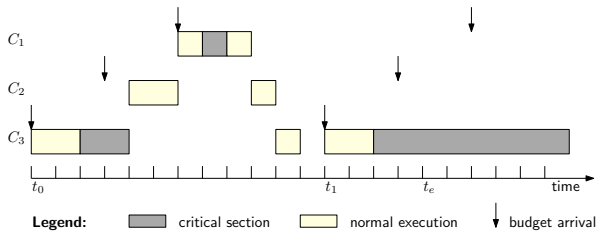

Fig. 1. Temporal isolation is unassured when a component, C_3, exceeds its specified critical-section length, i.e. at time instant t_e. The system ceiling, defined by resource ceiling $RC_l = \Pi_1$, blocks all other components.

To prevent this effect we introduce a *resource-access budget* q_s in addition to a component's budget Q_s, where budget q_s is used to enforce critical-section lengths. When a resource R_l gets locked, q_s replenishes to its full capacity, i.e. $q_s \leftarrow X_{sl}$. To monitor the available budget at any moment in time, we assume the availability of a function $Q_s^{\mathtt{rem}}(t)$ that returns the remaining budget of Q_s. Similarly, $q_s^{\mathtt{rem}}(t)$ returns the remainder of q_s at time t. If a component C_s executes in a critical section, then it consumes budget from Q_s and q_s in parallel, i.e. depletion of either Q_s or q_s forbids component C_s to continue its execution. We maintain the following invariant to prevent budget depletion during resource access:

Invariant 1: $Q_s^{\mathtt{rem}}(t) \geq q_s^{\mathtt{rem}}(t)$.

The way of maintaining this invariant depends on the chosen policy to prevent budget depletion during global resource access, e.g. by means of SIRAP [4], HSRP [3] or BROE [5]. For ease of presentation we will first complement HSRP's overrun mechanism with a mechanism for temporal protection. In Section 5.4 we show how our resulting protocol, HSTP, applies to both SIRAP's and BROE's self-blocking mechanisms.

4.1.1. Fault containment of critical sections. Existing SRP-based synchronization protocols in [4], [5], [15] make it possible to choose the local resource ceilings, rc_{sl}, according to SRP [6], see (3). In [20], [21] techniques are presented to trade-off preemptiveness against resource holding times. Given their common definition for local resource ceilings, a resource holding time, X_{sl}, may also include the interference of tasks with a preemption level higher than the resource ceiling. Task τ_{si} can therefore lock resource R_l longer than specified, because an interfering task τ_{sj} (where $\pi_{sj} > rc_{sl}$) exceeds its computation time, E_{sj}.

To prevent this effect we choose to disable preemptions for other tasks within the same component during critical sections, i.e. similar as HSRP [3] we choose local resource ceilings by $\forall R_l \in \mathcal{R}_s : rc_{sl} = \pi_{s_1}$. As a result X_{sl} only comprises task execution times within a critical section, i.e.

$$X_{sl} = \max_{1 \leq i \leq n_s} c_{sil}. \tag{4}$$

Since X_{sl} is enforced by budget q_s, temporal faults are contained within a subset of resource-sharing components.

4.1.2. Maintaining SRP ceilings. To enforce that a task τ_{si} resides no longer in a critical section than specified by X_{sl}, a resource $R_l \in \mathcal{R}$ maintains a state *locked* or *free*. We introduce an extra state *busy* to signify that R_l is locked by a misbehaving task. When a task τ_{si} tries to exceed its maximum critical-section length X_{sl}, we update SRP's *system ceiling* by mimicking a resource unlock and mark the resource *busy* until it is released. Since we decrease the system ceiling after τ_{si} executes for a duration of X_{sl} in a critical section to resource R_l, we can no longer guarantee absence of deadlocks. Nested critical sections to global shared resources are therefore unsupported[2]. One may alternatively aggregate

2. We allow nesting of a (sequence of) global resource access(es) inside local critical sections.

global resource accesses into a simultaneous lock and unlock of a single artificial resource [22]. Many protocols, or their implementations, lack deadlock avoidance [8], [11], [12], [17].

Although it seems attractive from a schedulability point of view to release the *component ceiling* when the critical-section length is exceeded, i.e. similar to the system ceiling, this would break SRP compliance, because a task may block on a busy resource instead of being prevented from starting its execution. Our approach therefore preserves the SRP property to share a single, consistent execution stack per component [6]. At the global level tasks can be blocked by a depleted budget, so that components cannot share an execution stack anyway.

4.1.3. Repetitive self-donation to resource-access budgets. A component that accesses a resource R_l maintains a dedicated resource-access budget q_s synchronous to its normal budget Q_s, i.e. similar to [8]. Since critical sections are often shorter than budget Q_s is, a component may still have remaining budget $Q_s^{\mathtt{rem}}(t) > 0$ when q_s depletes. Contrary to [8], when q_s depletes we repeatedly replenish it by a self-donation of X_{sl} until budget Q_s is exhausted. Although we need to *decrease the system ceiling before replenishing* q_s to avoid excessive blocking durations to other components, we may again increase the system ceiling for a duration of X_{sl} as soon as component C_s is selected by the global scheduler to continue its execution. This increases the likelihood that a resource-using component meets its deadline despite a misbehaving critical section and it reduces the likelihood that a resource-sharing component encounters a busy resource. Our resource-access budgets are therefore more reliable and robust than those in [8] are.

4.2. HSTP with self-donation

We specify four rules to manipulate a component's budget q_s when a resource is locked or unlocked and change the way budgets Q_s and q_s are replenished and depleted. This scheme defines HSTP, a protocol that maintains temporal protection between components during global resource access.

4.2.1. Lock resource. Upon an attempt of task $\tau_{si} \in \mathcal{T}_s$ to lock resource $R_l \in \mathcal{R}_s$ at time t_l, we raise the component ceiling independent of whether or not R_l is free.

If resource R_l is *free*, then τ_{si} locks the resource and q_s replenishes, i.e. $q_s \leftarrow X_{sl}$. Component C_s now runs in parallel on budget Q_s and q_s. We need to guarantee that C_s' normal budget Q_s does not deplete during global resource access. We therefore first save C_s' remaining budget as $Q_s^{\triangledown} \leftarrow Q_s^{\mathtt{rem}}(t_l)$ and then provide an overrun budget $Q_s^{\mathtt{rem}}(t_l) \leftarrow X_{sl}$.

By virtue of SRP, a resource R_l can never be *locked* when a task τ_{si} attempts to lock it. If there exists a task $\tau_{uj} \in \mathcal{T}_u$ which currently holds R_l *busy*, then we immediately deplete the remaining budget of C_s, i.e. $Q_s^{\mathtt{rem}}(t_l) \leftarrow 0$. After the budget Q_s of component C_s has replenished, the blocked task τ_{si} again tests whether or not resource R_l is free.

4.2.2. Unlock resource. If task $\tau_{si} \in \mathcal{T}_s$ unlocks a resource $R_l \in \mathcal{R}_s$ at time t_f, then the component and system ceilings are decreased according to the rules of SRP and resource R_l is marked *free*. Moreover, component C_s no longer consumes budget q_s and we need to restore C_s' budget. When a budget overrun has occurred, i.e. $Q_s^{\triangledown} < X_{sl} - q_s^{\mathtt{rem}}(t_f)$, then component C_s is suspended on its depleted budget Q_s. Otherwise, component C_s continues within its remaining budget, i.e. we restore Q_s with $\max(0, Q_s^{\triangledown} - (X_{sl} - q_s^{\mathtt{rem}}(t_f)))$.

4.2.3. Budget depletion. If component C_s' budget Q_s or q_s depletes and all resources $R_l \in \mathcal{R}_s$ are free, then nothing changes compared to default budget-depletion policies. However, if a task τ_{si} holds a resource $R_l \in \mathcal{R}_s$ so that q_s has been depleted, then (*i*) resource R_l is marked *busy*; (*ii*) the *system ceiling* is decreased according to the rules of unlocking an SRP resource and (*iii*) the budget Q_s of component C_s is restored with $\max(0, Q_s^{\triangledown} - X_{sl})$.

These actions guarantee that τ_{si} can overrun at most an amount X_{sl} and a component C_s can therefore at most request $Q_s + X_s$ processor time during each period P_s. If no other component C_t can preempt based on its preemption level $\Pi_t > \Pi_s$ after decreasing the system ceiling, the system ceiling can be raised again for a duration of X_{sl} without increasing the blocking times that component C_t may experience. Since the component ceiling is persistently raised during global resource access, other tasks in $\mathcal{T}_s$ than the resource-accessing one cannot execute. Hence, as long as a resource R_l is kept locked or busy by component C_s, task τ_{si} may entirely consume the remaining budget $Q_s^{\mathtt{rem}}(t)$ with a raised component and system ceiling via self-donations, *if* it decreases the system ceiling after every X_{sl} and performs a global preemption test.

4.2.4. Budget replenishment. If component C_s' budget Q_s replenishes, it is guaranteed that at most one resource $R_l \in \mathcal{R}_s$ is kept *busy* via component C_s and the corresponding resource-access budget $q_s^{\mathtt{rem}}(t) = 0$. If component C_s does not keep any resource busy, then nothing changes compared to default budget-replenishment policies. Otherwise, we restrict the replenishment of component C_s, so that it may continue its critical section, i.e (*i*) the budget to be restored upon unlocking, $Q_s^{\triangledown}$, replenishes to Q_s; (*ii*) the budgets q_s and Q_s that allow continuing resource access to R_l replenish with X_{sl} and (*iii*) a resource lock is mimicked by raising the system ceiling according to SRP. Hence, component C_s continues its execution with a raised system ceiling.

4.2.5. Final remarks. A component C_u may block on a busy resource R_l at a time t_b and subsequently R_l can become free at a time t_f before the replenishment of C_u's budget, Q_u. If we signal component C_u to continue its execution within $Q_u^{\mathtt{rem}}(t_b)$ at time t_f, we can no longer guarantee the provisioning of budget Q_u within its period boundaries P_u due to the self-suspension interval $[t_b, t_f]$ of budget Q_u. This introduces unpredictable interferences and scheduling anomalies to other components in the system [23]. We can therefore either let a

blocking task spinlock on a busy resource, so that it consumes its component's budget, or suspend a blocking component until its replenishment before allowing this component to resume its execution, similar to [8].

4.3. HSTP extended with third-party donations

The basic HSTP specification in Section 4.2 immediately depletes the budget of a component C_u upon an attempt at time t_b to lock a busy resource R_l. As a result, all remaining budget, $Q_u^{\texttt{rem}}(t_b)$, of the blocking component C_u is discarded until its next replenishment. Since C_u is unable to consume its budget due to the raised component ceiling, C_u may alternatively donate its budget to the misbehaving component C_s with the aim to reduce the waiting time on the busy resource R_l. In this section we complement HSTP with a donation mechanism.

4.3.1. Attempt to lock a busy resource. When component C_u encounters a busy resource at time t_b, we can repeatedly donate X_{ul} to misbehaving component C_s until Q_u is depleted with the aim to reduce its waiting time on resource R_l. Such a donation from donor C_u to donatee C_s (*i*) mimics a resource lock by raising the system ceiling, (*ii*) saves donor C_u's remaining budget as $Q_u^{\triangledown} \leftarrow Q_u^{\texttt{rem}}(t_b)$, (*iii*) saves donatee C_s' remaining budget as $Q_s^{\triangledown} \leftarrow Q_s^{\texttt{rem}}(t_b)$ and (*iv*) allocates budget $Q_s^{\texttt{rem}}(t_b), q_s \leftarrow X_{ul}$ to the critical section of component C_s. Due to the raised system ceiling, component C_s effectively runs at the resource-ceiling's preemption level RC_l. It is therefore unnecessary to change the deadline or priority of the donatee, because all components that may use R_l are blocked by the system ceiling.

4.3.2. Unlock resource. If an erroneous component C_s unlocks resource R_l at time t_f while consuming a donation, then any remaining resource-access budget $q_s^{\texttt{rem}}(t_f)$ is donated back to donor C_u, i.e. $Q_u^{\texttt{rem}}(t_f) \leftarrow \max(0, Q_u^{\triangledown} - (X_{ul} - q_s^{\texttt{rem}}(t_f)))$. Donatee C_s has consumed a part of the donated budget, $X_{ul} - q_s^{\texttt{rem}}(t_f)$, in the place of donor C_u. This part is accounted to donor C_u rather than to donatee C_s, so that the budget of donatee C_s is restored to $Q_s^{\texttt{rem}}(t_f) \leftarrow Q_s^{\triangledown}$. As a result both resource-sharing components C_s and C_u may resume execution within their restored budgets.

4.3.3. Budget depletion. When a donated budget X_{ul} is depleted by donatee C_s, the system ceiling is decreased and X_{ul} is subtracted from donor C_u's budget, i.e. $Q_u^{\texttt{rem}}(t) \leftarrow \max(0, Q_u^{\triangledown} - X_{ul})$. The budget of donatee C_s is restored with its original value $Q_s^{\texttt{rem}}(t) \leftarrow Q_s^{\triangledown}$. If donor C_u gets re-selected by the global scheduler for execution and resource R_l is still busy, it may again donate X_{ul} to component C_s.

4.3.4. Budget replenishment. During the consumption of donated budget X_{ul} from donor C_u, the budget of donatee C_s itself can get replenished. The replenishment of budget Q_s remains unchanged compared to Section 4.2.4. However, we can only replenish a depleted resource-access budget, i.e. $q_s^{\texttt{rem}}(t) = 0$. Otherwise, component C_s may cause double blocking to other components, i.e. $X_{ul} + X_{sl}$ instead of X_{sl}, see Figure 2. How to avoid this effect is unclearly described in [8], however. Because it is unattractive to account for double blocking in the system analysis, we avoid this blocking.

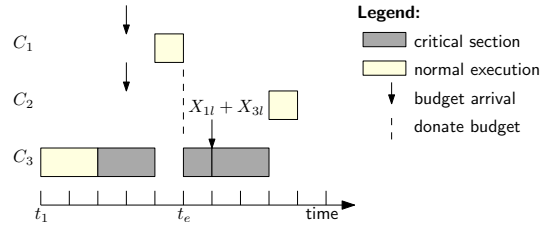

Fig. 2. If resource-access budget q_3 gets replenished and a task $\tau_{3i} \in \mathcal{T}_3$ executes in a critical section, then component C_3 may double block other components.

The key to the solution to avoid the double-blocking problem is to allow preemption *before* replenishing q_s. We can either choose to immediately deplete $q_s^{\texttt{rem}}(t) > 0$ or we can defer replenishment until q_s depletes. Note that the first alternative is less reliable, because the donor will not receive any of its donation back. However, in both cases the system ceiling must be decreased according to SRP's rules and the global scheduler must be called *prior to* replenishing q_s.

4.4. Donation policies

When a component depletes its resource-access budget or blocks on a *busy* resource, it cannot continue its execution. We allow two alternative budget-donation policies, symmetrically for a donatee C_s via self-donation or a third-party donor C_u:

4.4.1. At-once donation. A component C_u may (self-)donate its remaining budget $Q_u^{\texttt{rem}}(t)$ at once to a donatee C_s using BWI [8], [12]. However, component C_s *must* consume such donations *in the place of* donor C_u [24], i.e. at the donor's preemption-level rather than at the resource-ceiling level. This requires multiple budgets per component and migration of tasks to enable the execution of tasks over several budgets [8], which is costly [11] and breaks SRP's stack-sharing property. Hence, at-once donation causes additional run-time penalties compared to a regular two-level HSF implementation.

4.4.2. Repetitive donation. Similar to the misbehaving component C_s, the blocking component C_u may entirely donate its remaining budget $Q_u^{\texttt{rem}}(t)$ with a raised system ceiling *if* it decreases the system ceiling after every X_{sl} and performs a preemption test to avoid double blocking. As a result, after a preemption test the component with the highest preemption level, which is ready to execute, resumes its execution, so that a donatee always receives donated budget from the resource-dependent component with the highest preemption level.

Similar to *at-once donation*, however, a component C_u may (repetitively) donate its budget to a component C_s with a

depleted resource-access budget q_s, although donatee C_s has remaining budget $Q_s^{\texttt{rem}}(t) > 0$. An advantage of *repetitive self-donations* is that it reduces the number of such unnecessary donations to third-parties, because a replenished component can be prevented by the system ceiling from starting its execution and donation for a duration of X_{sl}; see Figure 3.

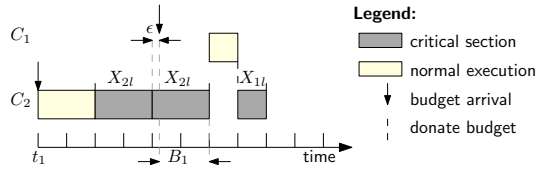

Fig. 3. Repetitive consumption of remaining budget can reduce unnecessary donations from blocking components. In this example component C_1 arrives an infinitesimal amount of time, ϵ, after the first depletion and preemption test of C_2, so that donation is deferred until component C_2 depletes its replenished budget $q_2 = X_{2l}$.

Given the repetitive donation policy, the following lemma follows from our HSTP specification:

Lemma 1: At each time instant t there is at most one donor component C_u per donatee C_s and each component C_s can only execute a single global critical section.

Proof: As long as a resource R_l is kept locked or busy by component C_s, its component ceiling prevents all other tasks within the component from starting their execution. By absence of nested critical sections, no other resource R_k can be kept locked or busy by the same component C_s.

A component C_u which attempts to lock R_l may donate a budget of length X_{ul} and immediately raises the system ceiling. Hence, other R_l-sharing components C_t cannot preempt while donatee C_s consumes the donated budget X_{ul}. □

In the remainder of this paper we use the repetitive-donation policy, because it eliminates donations at arbitrary moments in time, which increases the reliability of a system. Secondly, it eliminates transitive donations, so that we need to track at most a single donor component within the context of a donatee.

5. Dependable and compositional analysis

We presented a synchronization protocol, HSTP, which enables a dependable execution of components that exceed their specified resource holding times. Contrary to [8], a mechanism to monitor blocking times is unnecessary, because we will show that HSTP complies to SRP's blocking times. To make HSTP applicable in HSFs we reuse the analysis results presented in [4], [15], so that we obtain independent analysis for individual components and their integration. To summarize, the HSTP specification implies the following *system invariant*:

Invariant 2: If component C_s executes on the processor and holds resource R_l (locked or busy), then either (*i*) the system ceiling is raised to RC_l or (*ii*) C_s has the highest preemption level $\Pi_s > \Pi_u$ among all components C_u that share R_l and are ready to execute (i.e. $q_u^{\texttt{rem}}(t) > 0$).

Lemma 2: The HSTP specification in Section 4.2 and Section 4.3 implies Invariant 2.

Proof: If component C_s executes and keeps resource R_l *locked* or *busy*, then $q_s^{\texttt{rem}}(t) > 0$ and the system ceiling equals RC_l. Hence, no R_l-sharing component can preempt.

Similarly, if component C_s received X_{ul} from a donor C_u, then C_s executes with a raised system ceiling (RC_l) and keeps resource R_l *busy*. For donor C_u holds: $q_u = 0$, so that it is suspended and other R_l-sharing components cannot preempt.

Hence, component C_s has the highest preemption level among all ready components that share R_l. □

5.1. SRP-compliant blocking

When a task exceeds its maximum duration inside a critical section, only components that are involved in the interaction are affected. Although we decrease SRP's system ceiling when a task exceeds its specified critical-section length, HSTP has the same calculation of the global blocking terms as SRP.

Theorem 1: Invariant 2 guarantees that a component C_s does not suffer more blocking or interference on unused resources R_l, i.e. $R_l \notin \mathcal{R}_s$, compared to SRP.

Proof: As long as the *busy* state of a resource R_l is unreached, i.e. R_l is either *free* or *locked*, our protocol strictly follows SRP. Given Lemma 1, we need to consider only a single busy resource R_l for each component in the system.

Consider an erroneous component C_s that exceeds its worst-case critical-section length X_{sl} for resource R_l. Component C_s may receive a donation from C_w with a lower preemption level $\Pi_w < \Pi_s$ or component C_h with a higher preemption level $\Pi_h > \Pi_s$. We can therefore have independent components C_t with $R_l \notin \mathcal{R}_t$ at four different preemption levels:

1) C_t with $\Pi_t > \Pi_h > \Pi_s > \Pi_w$;
2) C_t with $\Pi_h > \Pi_t > \Pi_s > \Pi_w$ blocked by C_w or C_s and preempted by C_h;
3) C_t with $\Pi_h > \Pi_s > \Pi_t > \Pi_w$ blocked by C_w and preempted by C_s and C_h;
4) C_t with $\Pi_h > \Pi_s > \Pi_w > \Pi_t$ preempted by C_w, C_s and C_h.

Without loss of generality, we may restrict ourselves to an artificial system comprising each of these cases, i.e. a system $C_1, \ldots, C_7 \in \mathcal{C}$ with a single shared resource $R_l \in \mathcal{R}_2 \cap \mathcal{R}_4 \cap \mathcal{R}_6$ and $R_l \notin \mathcal{R}_1 \cup \mathcal{R}_3 \cup \mathcal{R}_5 \cup \mathcal{R}_7$ and resource ceiling $RC_l = \Pi_2$. Furthermore, C_4 keeps R_l busy after it has executed for X_{4l} with a raised system ceiling. We now prove the theorem by contradiction, i.e. assume there exists an independent component C_1, C_3, C_5 or C_7 that can experience more than one blocking occurrence and additional interference in the presence of HSTP while this cannot happen with SRP.

Because C_1 has $\Pi_1 > RC_l$, it cannot be blocked by (or suffer interference from) any R_l-sharing component (case 1).

After the system ceiling is decreased, component C_2 can preempt according to its preemption level, $\Pi_2 > \Pi_4$. If component C_2 tries to access R_l, it may donate at most X_{2l} to component C_4. This is already accounted as interference for

$C_3 \ldots C_7$. Hence, C_3, C_5 and C_7 do not suffer more blocking or more interference from C_2 (case 2).

If component C_6 tries to access R_l it may donate at most X_{6l} to C_4, which will block C_3 and C_5 no longer than X_{6l}. This donation, X_{6l}, is accounted as interference for C_7 and blocking for $C_2 \ldots C_5$. Hence, C_3, C_5 and C_7 do not suffer more blocking or more interference from C_6 (case 3).

Although component C_4 may consume the remainder of its budget Q_4^{rem} while it keeps R_l busy, the interference for component C_7 remains the same compared to SRP (case 4).

By contradiction and using Lemma 2 we conclude that HSTP preserves SRP's blocking properties. □

Since Theorem 1 has shown that HSTP complies to SRP's blocking term, we can *safely* reuse existing global analysis for component integration [1], [4], [15] without the implicit assumption that tasks behave according to their contract.

5.2. Global analysis under temporal protection

In line with our specification, we first first present the global analysis of components based on HSTP and an overrun mechanism. In Section 5.4 we adapt this analysis for SIRAP and BROE which each apply a self-blocking mechanism.

The following sufficient schedulability condition holds for global EDF-based systems [25]:

$$\forall t > 0 \quad : \quad B(t) + \texttt{dbf}_{\texttt{EDF}}(t) \leq t. \tag{5}$$

The blocking term, $B(t)$, is defined in [25] according to SRP. The demand bound function $\texttt{dbf}_{\texttt{EDF}}(t)$ computes the total processor demand of all components in the system for every time interval of length t, i.e.

$$\texttt{dbf}_{\texttt{EDF}}(t) \quad = \quad \sum_{C_s \in \mathcal{C}} \left\lfloor \frac{t}{P_s} \right\rfloor (Q_s + O_s(t)) \tag{6}$$

A component C_s, using an overrun mechanism to prevent budget depletion during global resource access, demands $O_s(t)$ more resources in its worst-case scenario [15], where

$$O_s(t) = \begin{cases} X_s & \text{if } t \geq P_s \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

For global FPPS the following sufficient condition holds:

$$\forall 1 \leq s \leq N : \quad \exists t \in [0, P_s] : \quad B_s + \texttt{dbf}_{\texttt{DM}}(t, s) \leq t, \tag{8}$$

where the blocking term, B_s, is defined as in [6] and $\texttt{dbf}_{\texttt{DM}}(t, s)$ denotes the worst-case cumulative processor request of C_s for a time interval of length t [1], i.e.

$$\texttt{dbf}_{\texttt{DM}}(t, s) \quad = \quad Q_s + O_s + \sum_{1 \leq r < s} \left\lceil \frac{t}{P_r} \right\rceil (Q_r + O_r). \tag{9}$$

A component C_s, using an overrun mechanism, demands $O_s = X_s$ more resources in its worst-case scenario [15].

5.3. Local analysis for donating components

By filling in task characteristics in the dbf of (5) and (8) and replacing their right-hand sides by (1), i.e. replace t for $\texttt{sbf}_{\Gamma_s}(t)$, the same schedulability analysis holds for tasks within a component as for components at the global level. The processor supply to a component, as specified by (1), is only affected when the component blocks on a busy resource.

Inherent to HSFs, however, the local schedulability of tasks in $\mathcal{T}_s$ is only guaranteed when all tasks $\tau_{si} \in \mathcal{T}_s$ respect their timing characteristics. For example, when a task τ_{si} exceeds its worst-case execution time, all other tasks in $\mathcal{T}_s$ may miss their deadline, even when tasks are independent. A similar argument applies when a task shares mutually non-preemptive resources, i.e. a task may unpredictably block for an unbounded duration on a busy resource. It may be useful for a task to test whether a resource is *busy*, so that a task can *compensate* by providing a reduced functionality [7].

If a task τ_{si} in component C_s exceeds its specified critical-section length with an amount c'_{sil}, then its finishing time is potentially delayed. This can be modeled as extra interference of c'_{sil} processor time for task τ_{si} as well as all other tasks in the same component, $\tau_{sj} \in \mathcal{T}_s$. For a donor C_u the situation is symmetrical, i.e. all tasks $\tau_{uj} \in \mathcal{T}_u$ of donor C_u experience c'_{sil} extra interference. Since the local cost of consuming a donation is the same as the local cost of a donation itself, we can consider the longest duration of c'_{sil} for any component $C_s \in \mathcal{C}$ independently, so that all tasks $\tau_{si} \in \mathcal{T}_s$ can still make their deadline. However, a corresponding allocation of budgets to components while maximizing the system robustness is left as a future work.

5.4. HSTP and self-blocking

HSTP also applies to SIRAP and BROE, which cancels the allocation of overrun budgets in (6) and (9), i.e. $O_s(t) = O_s = 0$. When the budget is insufficient to complete a critical section, both protocols postpone the execution of the critical section until sufficient budget $Q_s^{\texttt{rem}}(t) \geq X_{sl}$ is guaranteed. This self-blocking condition also applies for donations to third-parties, i.e. we can only *donate budget to others* when there is sufficient budget to donate. This gives the misbehaving component C_s the opportunity to resolve its own malicious behaviour until component C_u has sufficient resources and gets selected for execution.

Consider a misbehaving component C_s that accesses resource R_l. When component C_s needs more processor time to complete its critical section than specified by X_{sl}, we can repetitively *self-donate* any remaining budget $Q_s^{\texttt{rem}}(t) > 0$ to resource-access budget q_s. It is unnecessary to check for sufficient budget before a self-donation. To prevent budget overruns, however, we constrain a self-donation by only providing $Q_s^{\texttt{rem}}(t) \leq X_{sl}$, i.e. $q_s \leftarrow \min(Q_s^{\texttt{rem}}(t), X_{sl})$.

Assume that a resource-sharing component C_u attempts to lock the *busy* resource R_l at time t_b. Before donating a budget X_{ul} to component C_s, a self-blocking mechanism applies

(either SIRAP or BROE). The first time instant t_l at which component C_u resumes its execution after budget Q_u has been replenished, is the actual time that a task tries to lock resource R_l. Within time interval $[t_b, t_l]$ the resource may get released. If resource R_l is *busy* at time t_l, then, similar to self-donations, component C_u can repetitively donate budget without further self-blocking to accelerate the resource release. When a component C_u starts donating its budget, however, its tasks will miss deadlines, unless it donates slack time.

5.4.1. An example. Consider a component C_1 serviced by BROE [5], see Figure 4, which self-blocks on its budget Q_1 upon an attempt to lock a *busy* resource R_l at time t_b. Within component C_1 virtual time advances with a rate $\frac{Q_1}{P_1}$. When the absolute time of the current budget $Q_1^{\text{rem}}(t_b)$ reaches the virtual time, i.e. component C_1 was running ahead of absolute time, a replenished budget becomes available at time $t_a = t_{d_k} - \frac{P_1}{Q_1} Q_1^{\text{rem}}(t_b)$ with a deadline $t_{d_{k+1}} = t_a + P_1$. A replenishment can happen even without self-blocking if the component was lagging behind. After we would have donated $Q_1^{\text{rem}}(t_b)$, a replenishment happens at t_{d_k}, but with a larger absolute deadline $t'_{d_{k+1}} = t_{d_k} + P_1$ where $t_{d_k} < t_{d_{k+1}} \leq t'_{d_{k+1}}$.

By refraining a donation at time t_b, potential deadline misses of tasks in $\mathcal{T}_1$ can be prevented. If the busy resource R_l is released before component C_1 continues its execution, it is unnecessary for component C_1 to donate budget.

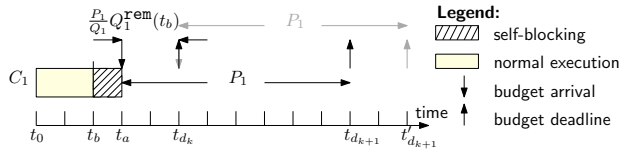

Fig. 4. Donating $Q_1^{\text{rem}}(t_b)$ (in grey) instead of self-blocking (in black) can cause unnecessary deadline misses.

5.4.2. Gain-time provisioning. Contrary to BROE, SIRAP needs to account for an additional self-blocking term in its local demand bound function of DM-scheduled tasks [4], i.e.[3]

$$I_s(i,t) = b_{si} + \sum_{R_l \in \mathcal{R}_s} (c_{sil} + \sum_{1 \leq j < i} \left\lceil \frac{t}{T_{sj}} \right\rceil c_{sjl}), \qquad (10)$$

where the blocking term, b_{si}, for tasks is defined as in [6]. We can similarly adapt (6) for local EDF-scheduling of tasks that share resources arbitrated by SIRAP. Since we disable local preemptions during global resource access, the allocated processor time to a component for self-blocking is unused. We can make this unused processor time available as *gain time* by also donating $Q_s^{\text{rem}}(t_b) < X_{sl}$, i.e. independent of the self-blocking condition.

3. For the ease of presentation, Equation 10 assumes that each instant of a task (i.e. a job) accesses a shared resource at most one time.

5.5. Locally non-preemptive critical sections

Local preemptions during global resource access can decrease the required budget to make a task set $\mathcal{T}_s$ schedulable, but this may on its turn adverse the global schedulability of components due to increased resource holding times. In this paper we are not interested in this schedulability trade-off [20], [21], however, but merely in the containment of temporal faults in critical sections. For this purpose we can introduce an intermediate reservation level assigned and allocated to critical sections in order to enforce that blocking times to other components are not exceeded due to locally preempting tasks [8]. Since this approach causes performance penalties [11], HSTP disables local preemptions during global resource access. In this section we propose an efficient algorithm to check whether off-the-shelf components can be integrated in our dependable resource-sharing framework, which enables a fast design-space exploration during system composition.

We consider a component C_s with a given interface description $< P_s, Q_s, \mathcal{X}_s >$, where resource ceilings are locally and globally configured according to SRP, i.e. see (2) and (3). We want to check whether executing the global critical sections of component C_s with local preemptions disabled hampers the schedulability of a task set $\mathcal{T}_s$ for the given periodic resource model $\Gamma_s(P_s, Q_s)$ of component C_s. In many practical cases local preemptions can be disabled temporary, because the budget Q_s assigned to a component is typically pessimistic due to abstraction overheads in its calculation [1]. Each task $\tau_{si} \in \mathcal{T}_s$ may therefore finish its execution before its deadline, so that small finalization delays do not cause a deadline miss.

5.5.1. Delay tolerance. From our assumption $2P_s \leq T_{si} (\forall \tau_{si} \in \mathcal{T}_s)$ we can deduct the following lemma:

Lemma 3: Given $2P_s \leq T_{si} (\forall \tau_{si} \in \mathcal{T}_s)$, all tasks τ_{sj} that are allowed to preempt, can preempt at most once during an access to a global shared resource R_l by task τ_{si}.

Proof: The proof for SIRAP and overrun is presented in [26]. The proof for BROE is presented in [27]. □

Using Lemma 3 we can efficiently calculate how long local preemptions can be disabled without affecting the local schedulability of tasks.

We define the *laxity* δ_{si} of a task τ_{si}, such that τ_{si} finishes its execution at least δ_{si} time units prior to its deadline D_{si} if it has the entire processor at its disposal. For each resource $R_l \in \mathcal{R}_s$ with a local ceiling $rc_{sl} < \pi_{s1}$, we compute the least laxity of all tasks with a preemption level higher than rc_{sl}. The largest common *delay tolerance* of these preempting tasks is:

$$\Delta'_{sl} = (\min\ i : \pi_{si} > rc_{sl} : \delta_{si} = D_{si} - \sum_{1 \leq j \leq i} E_{sj}). \qquad (11)$$

Note that each task τ_{sj} can only preempt once and all tasks are ordered according to their preemption level, i.e. the lowest index gives the highest preemption level.

The laxity of each task must at least exceed a single blackout duration of its component, i.e. $\delta_{si} \geq BD_s (\forall \tau_{si} \in \mathcal{T}_s)$, in order to be schedulable with a periodic processor supply

of $\Gamma_s(P_s, Q_s)$. An additional blackout is impossible in the processor supply of preempting tasks, because a critical section of length X_{sl} is guaranteed to fit within a single budget provisioning by virtue of the protocols in [3], [4], [5]. Tasks may nevertheless have more delay tolerance than required to bridge the blackout duration. We now need to subtract BD_s from the calculated delay tolerance, i.e.

$$\Delta_{sl} \leftarrow \Delta'_{sl} - BD_s, \tag{12}$$

so that the result is $\Delta_{sl} \geq 0$. This final value Δ_{sl} defines the longest time that a task $\tau_{si} \in \mathcal{T}_s$ that may preempt during resource access to R_l can be deferred without missing any deadline. By construction the following theorem follows:

Theorem 2: Given that a component C_s accessing resource $R_l \in \mathcal{R}_s$ with resource ceiling rc_{sl} is schedulable on a component with parameters $(P_s, Q_s, \mathcal{X}_s)$ using the periodic resource model $\Gamma_s(P_s, Q_s)$: if $c_{sil} \leq \Delta_{sl}$, then component C_s can be scheduled on the same component using $\Gamma_s(P_s, Q_s)$ and can execute each critical section of τ_{si} to R_l with local preemptions disabled, where Δ_{sl} is defined in (12).

Proof: Given Lemma 3, all tasks that may preempt a critical section with resource ceiling rc_{sl} finish within a budget of $Q_s + X_s$ and preempt only once. After increasing the resource ceiling, the total required budget to meet the resource demands of c_{sil} and the execution times of these tasks that now suffer blocking does not increase. We are given two facts: (*i*) each task instance may experience only one blocking occurrence under SRP-based resource arbitration [6] and (*ii*) using the original resource ceiling, each preempting tasks completes its execution at least Δ_{sl} time units prior to its deadline on periodic resource $\Gamma_s(P_s, Q_s)$. Hence, a blocking duration of $c_{sil} \leq \Delta_{sl}$ cannot cause a deadline miss. □

5.5.2. An algorithm. Theorem 2 makes it possible to *exactly determine* whether or not all critical sections within a component C_s can be executed with local preemptions disabled by (*i*) calculating Δ_{sl} using (11) and (12) for each $R_l \in \mathcal{R}_s$ and subsequently (*ii*) apply Theorem 2 on each task to verify $c_{sil} \leq \Delta_{sl}$. This algorithm has a time complexity of $\mathcal{O}(M_s \times n_s)$ for a single component C_s with M_s global shared resources. Note that it only needs to inspect the laxity for interfering tasks, which makes our algorithm much more efficient than the algorithms in [20], [21].

5.5.3. A special case. BROE has a *sufficient test*, i.e. $\Delta_{sl} = \frac{1}{2}BD_s = P_s - Q_s$, for disabling local preemptions of EDF-scheduled task sets [5]. Our algorithm determines *exactly* which critical sections can execute without local preemptions and it applies to any periodic resource model, independent of the local scheduler, with SRP-based resource arbitration.

6. Evaluation

We have recently extended a commercial real-time microkernel, μC/OS-II [28], with an HSF and synchronization support by means of SIRAP, HSRP and BROE [16]. In this section we investigate the complexity and overheads of the synchronization primitives of HSTP within our framework.

Because we need to set a budget-expiration timer to track the component's resource-access budget in every lock operation and we need to cancel the same timer in every unlock operation, HSTP primitives are more expensive than a straightforward two-level SRP implementation. If this budget-depletion timer expires during global resource access, i.e. it is not canceled before expiration, then it executes a handler which implements HSTP's unlock policy and marks the resource busy. Based on our measurements, the timer manipulations triple the execution times of the lock and unlock operations compared to the implementations of these primitives in [16]. However, this is the price for preventing the propagation of temporal faults to resource-independent components.

The self-donation mechanism can be easily implemented by extending the global scheduler. Upon a context switch the global scheduler must check whether or not the selected component keeps a resource busy and at the same time has a depleted resource-access budget. This if-statement only takes a few instructions in a reservation-based framework and is therefore relatively cheap compared to the cost of context switching itself. Only when a task within the selected component misbehaves by exceeding its critical-section length, the scheduler executes the expensive operations corresponding to locking a resource, i.e. reset a budget-expiration timer for the selected component, update its normal budget and raise the system ceiling.

Third-party donations can be implemented similarly, but in the lock operation rather than in the global scheduler. If a task attempts to lock a busy resource, the lock operation disables all local preemptions, donates budget by setting a new budget-timer for the donatee and raises the system ceiling. As a result the donor component itself is blocked from continuing its execution. These actions are repeated every time the donor task gets selected by the local scheduler to continue its execution, until the requested resource becomes free.

Since Lemma 1 tells that we only need to keep track of at most a single donor at each time instant, each component maintains a single variable to track its current donor. Upon donation, the donor writes its component identifier into this variable. When it contains a valid identifier when a busy resource is unlocked, a donation back to the donor is executed. The variable is cleared when either the resource is freed or the donated budget is depleted.

7. Conclusion

This paper presented HSTP, an SRP-based synchronization protocol, which provides temporal protection between components in which multiple tasks share a budget, even when interacting components exceed their specified critical-section lengths. Prerequisites to dependable resource sharing in HSFs are mechanisms to enforce and monitor critical-section lengths. We followed the choice in [3] to make critical sections non-preemptive for tasks within the same component, because

this makes containment of temporal faults within critical sections efficient. Moreover, sufficiently short critical sections can execute with local preemptions disabled and preserve system schedulability. A reservation-based mechanism to monitor and enforce blocking times is unnecessary, see [8], because HSTP complies to existing SRP-based global analysis for HSFs.

We proposed an SRP-compliant budget donation mechanism. Our repetitive self-donation mechanism, complemented with donations from other components, limits preemptions during global resource access as long as possible and preserves the schedulability of independent components. Moreover, it enhances the reliability of resource-sharing components when one of the components misbehaves by accelerating the resource release. HSTP expands across existing protocols in the context of HSFs [3], [4], [5] and integrating its primitives into a reservation-based kernel is straightforward. Our protocol therefore promises a dependable solution to resource sharing in future safety-critical industrial applications for which temporal and functional correctness is essential.

References

[1] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symp.*, Dec. 2003, pp. 2–13.

[2] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Syst.*, vol. 9, no. 1, pp. 31–67, 1995.

[3] R. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Real-Time Systems Symp.*, 2006, pp. 257–267.

[4] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in *Conf. on Embedded Software*, Oct. 2007, pp. 279–288.

[5] M. Bertogna, N. Fisher, and S. Baruah, "Resource-sharing servers for open environments," *IEEE Trans. on Industrial Informatics*, vol. 5, no. 3, pp. 202–219, Aug. 2009.

[6] T. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, March 1991.

[7] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[8] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource sharing in reservation-based systems," in *Real-Time Systems Symp.*, Dec. 2001, pp. 171–180.

[9] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronisation," *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[10] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Symp. on Operating Systems Design and Implementation*, 1999, pp. 45–58.

[11] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction for real-time systems," in *Euromicro Conf. on Real-Time Systems*, July 2005, pp. 89–97.

[12] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Trans. on Computers*, vol. 53, no. 12, pp. 1591–1601, Dec. 2004.

[13] R. Santos, G. Lipari, and J. Santos, "Improving the schedulability of soft real-time open dynamic systems: The inheritor is actually a debtor," *Journal of Systems and Software*, vol. 81, pp. 1093–1104, July 2008.

[14] Z. Deng and J.-S. Liu, "Scheduling real-time applications in open environment," in *Real-Time Systems Symp.*, Dec. 1997, pp. 308–319.

[15] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, "Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 1, pp. 93 –104, Feb. 2010.

[16] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Protocol-transparent resource sharing in hierarchically scheduled real-time systems," in *Conf. Emerging Technologies and Factory Automation*, 2010.

[17] M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril, "Implementation of overrun and skipping in VxWorks," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010.

[18] L. Almeida and P. Peidreiras, "Scheduling with temporal partitions: response-time analysis and server design," in *Conf. on Embedded Software*, Sept. 2004, pp. 95–103.

[19] G. Buttazzo and P. Gai, "Efficient implementation of an EDF scheduler for small embedded systems," in *Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2006.

[20] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *Parallel and Distrib. Processing Symp.*, 2007.

[21] N. Fisher, M. Bertogna, and S. Baruah, "Resource-locking durations in EDF-scheduled systems," in *Real-Time and Embedded Technology and Applications Symp.*, April 2007, pp. 91–100.

[22] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symp.*, Dec. 1988, pp. 259–269.

[23] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *Real-Time Systems Symp.*, Dec. 2004, pp. 47–56.

[24] R. J. Bril, W. F. J. Verhaegh, and C. C. Wüst, "A cognac-glass algorithm for conditionally guaranteed budgets," in *Real-Time Systems Symp.*, Dec. 2006, pp. 388–397.

[25] S. K. Baruah, "Resource sharing in EDF-scheduled systems: A closer look," in *Real-Time Systems Symp.*, 2006, pp. 379–387.

[26] M. Behnam, T. Nolte, M. Åsberg, and R. J. Bril, "Overrun and skipping in hierarchically scheduled real-time systems," in *Conf. on Embedded and Real-Time Computing Systems and Applications*, 2009, pp. 519–526.

[27] M. Behnam, T. Nolte, and N. Fisher, "On optimal real-time subsystem-interface generation in the presence of shared resources," in *Conf. on Emerging Technologies and Factory Automation*, Sept. 2010.

[28] Micrium, "RTOS and tools," March 2010. [Online]. Available: http://micrium.com/