# Temporal isolation in an HSF-enabled real-time kernel in the presence of shared resources

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. To support resource sharing within two-level, fixed priority scheduled HSFs, two synchronization protocols based on the stack resource policy (SRP) have recently been presented, i.e. HSRP [1] and SIRAP [2]. 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. As a solution we propose a SRP-based synchronization protocol for HSFs, named Basic Hierarchical Synchronization protocol with Temporal Protection (B-HSTP). The schedulability of those components that are independent of the unavailable resource is unaffected.**

**This paper describes an implementation to provide HSFs, accompanied by SRP-based synchronization protocols, with means for temporal isolation. We base our implementations on the commercially available real-time operating system $\mu$C/OS-II, extended with proprietary support for two-level fixed priority preemptive scheduling. We specifically show the implementation of B-HSTP and we investigate the system overhead induced by its synchronization primitives in combination with HSRP and SIRAP. By supporting both protocols in our HSF, their primitives can be selected based on the protocol's relative strengths[1].**

## I. Introduction

The increasing complexity of real-time systems demands a decoupling of (*i*) development and analysis of individual components and (*ii*) integration of components on a shared platform, including analysis at the system level. Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating this decoupling [3]. A component that is validated to meet its timing constraints when executing in isolation will continue meeting its timing constraints after integration (or admission) on a shared platform. The HSF therefore provides a promising solution for current industrial standards, e.g. the AUtomotive Open System ARchitecture (AUTOSAR) [4] which specifies that an underlying OSEK-based operating system should prevent timing faults in any component to propagate to different components on the same processor. The HSF provides temporal isolation between components by allocating a *budget* to each component, which gets mediated access to the processor by means of a *server*.

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 [5].

Looking at existing industrial real-time systems, fixed-priority preemptive scheduling (FPPS) is the de-facto standard of task scheduling, hence we focus on an HSF with support for FPPS within a component. Having such support will simplify migration to and integration of existing legacy applications into the HSF. Our current research efforts are directed towards the conception and realization of a two-level HSF that is based on (*i*) FPPS for both *global scheduling* of servers allocated to components and *local scheduling* of tasks within a component and (*ii*) the Stack Resource Policy (SRP) [6] for both local and global resource sharing.

To accommodate resource sharing between components, two synchronization protocols [1], [2] have been proposed based on SRP for two-level FPPS-based HSFs. 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* when the remaining budget is insufficient to complete a critical section [2] or (*ii*) *overrun* the budget until the critical section ends [1]. 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 [1], [2] therefore break the temporal encapsulation and fault-containment properties of an HSF without the presence of complementary protection.

### A. Problem description

Most off-the-shelf real-time operating systems, including $\mu$C/OS-II [7], do not provide an implementation for SRP nor hierarchical scheduling. We have extended $\mu$C/OS-II with support for idling periodic servers (IPS) [8] and two-level FPPS. However, existing implementations of the synchroniza-

[1]The work in this paper is supported by the Dutch HTAS-VERIFIED project, see http://www.htas.nl/index.php?pid=154. Our $\mu$C/OS-II extensions are available at http://www.win.tue.nl/~mholende/relteq/.

tion protocols in our framework [9], [10], as well as in the framework presented in [11], do not provide any temporal isolation during global resource access.

A solution to limit the propagation of temporal faults to those components that share global resources is considered in [12]. Each task is assigned a dedicated budget per global resource access and this budget is synchronous with the period of that task. However, in [12] they allow only a single task per component.

We consider the problem 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. Moreover, we present an efficient implementation and evaluation of our protocol in $\mu$C/OS-II. The choice of operating system is driven by its former OSEK compatibility[2].

### B. Contributions

The contributions of this paper are fourfold.

- To achieve temporal isolation between components, even when resource-sharing components misbehave, we propose a modified SRP-based synchronization protocol, named *Basic Hierarchical Synchronization protocol with Temporal Protection* (B-HSTP).
- We show its implementation in a real-time operating system, extended with support for two-level fixed-priority scheduling, and we efficiently achieve fault-containment by disabling preemptions of other tasks within the same component during global resource access.
- We show that B-HSTP complements existing synchronization protocols [1], [2] for HSFs.
- We evaluate the run-time overhead of our B-HSTP implementation in $\mu$C/OS-II on the OpenRISC platform [13]. These overheads become relevant during deployment of a resource-sharing HSF.

### C. Organization

The remainder of this paper is organized as follows. Section II describes related works. Section III presents our system model. Section IV presents our resource-sharing protocol, B-HSTP, which guarantees temporal isolation to independent components. Section V presents our existing extensions for $\mu$C/OS-II comprising two-level FPPS-based scheduling and SRP-based resource arbitration. Section VI presents B-HSTP's implementation using our existing framework. Section VII investigates the system overhead corresponding to our implementation. Section VIII discusses practical extensions to B-HSTP. Finally, Section IX concludes this paper.

## II. Related work

Our basic idea is to use two-level SRP to arbitrate access to global resources, similar as [1], [2]. In literature several alternatives are presented to accommodate task communication in reservation-based systems. De Niz et al. [12] support resource sharing between reservations based on the immediate priority ceiling protocol (IPCP) [14] in their FPPS-based Linux/RK resource kernel and use a run-time mechanism based on resource containers [15] for temporal protection against misbehaving tasks. Steinberg et al. [16] 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 [17] 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 [14], i.e. when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. However, 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 [18]. A prerequisite to enable independent analysis of interacting components and their integration is the knowledge of which resources a task will access [2], [19]. 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 in FPPS-based HSFs, two synchronization protocols have been proposed based on SRP [6]: HSRP [1] and SIRAP [2]. Although HSRP [1] originally does not integrate into HSFs due to the lacking support for independent analysis of components, Behnam et al. [19] lifted this limitation. However, these two protocols, including their implementations in [9], [10], [11], assume that components respect their timing contract with respect to global resource sharing. In this paper we present an implementation of HSRP and SIRAP protocols that limits the unpredictable interferences caused by contract violations to the components that share the global resource.

## III. Real-time scheduling model

We consider a two-level FPPS-scheduled HSF, following the periodic resource model [3], to guarantee processor allocations to components. We use SRP-based synchronization to arbitrate mutually exclusive access to global shared resources.

### A. Component model

A system contains a set $\mathcal{R}$ of $M$ global logical resources $R_1, R_2, \ldots, R_M$, 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 [3], 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. A server implements a policy to distribute the available budget to the component's workload.

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

[2]Unfortunately, the supplier of $\mu$C/OS-II, Micrium, has discontinued the support for the OSEK-compatibility layer.

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 $X_{sl} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$.

### B. Task model

Each component $C_s$ contains a set $\mathcal{T}_s$ of $n_s$ sporadic tasks $\tau_{s1}$, $\tau_{s2}$, ..., $\tau_{sn_s}$. 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 $0 < E_{si} \leq D_{si} \leq T_{si}$. The worst-case execution time of task $\tau_{si}$ 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 $c_{sil} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$. All (critical-section) execution times are accounted in terms of processor cycles and allocated to the calling task's budget. For notational convenience we assume that tasks (and components) are given in priority order, i.e. $\tau_{s1}$ has the highest priority and $\tau_{sn_s}$ has the lowest priority.

### C. Synchronization protocol

Traditional synchronization protocols such as PCP [14] and SRP [6] can be used for *local* resource sharing in HSFs [20]. 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.

*1) 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 priority of any component or task that shares the resource. According to SRP, these ceilings are defined as:

$$RC_l = \min(N, \min\{s \mid R_l \in \mathcal{R}_s\}), \quad (1)$$
$$rc_{sl} = \min(n_s, \min\{i \mid c_{sil} > 0\}). \quad (2)$$

We use the outermost min in (1) and (2) to define $RC_l$ and $rc_{sl}$ in those situations where no component or task uses $R_l$.

*2) 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 priority is higher than its component ceiling. A similar condition for preemption holds for components.

*3) Prevent excessive blocking:* HSRP [1] uses an overrun mechanism [19] 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 $\tau_{si}$ releases the resource. These budget overruns cannot take place across replenishment boundaries, i.e. the analysis guarantees $Q_s + X_s$ processor time before the relative deadline $P_s$ [1], [19].

SIRAP [2] uses a self-blocking approach to prevent budget depletion inside a critical section. If a task $\tau_{si}$ 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 priority lower than the local ceiling $rc_{sl}$ to execute until the requested critical section has been finished.

The relative strengths of HSRP and SIRAP have been analytically investigated in [21] and heavily depend on the chosen system parameters. To enable the selection of a particular protocol based on its strengths, we presented an implementation supporting both protocols with transparent interfaces for the programmer [9]. In this paper we focus on mechanisms to extend these protocols with temporal protection and merely investigate their relative complexity with respect to our temporal-protection mechanisms.

## IV. SRP WITH TEMPORAL PROTECTION

Temporal faults may cause improper system alterations, e.g. due to unexpectedly long blocking or an inconsistent state of a resource. Without any protection a self-blocking approach [2] may miss its purpose under erroneous circumstances, i.e. when a task overruns its budget to complete its critical section. Even an overrun approach [1], [19] needs to guarantee a maximum duration of the overrun situation. Without such a guarantee, these situations 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 [22], where a task is allowed to indefinitely overrun its budget as long as it locks a resource, is therefore not *reliable*.

### A. 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 [20]. However, this might be too restrictive and lead 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 [12]. Otherwise such a misbehaving task may increase blocking to components with a higher priority, so that even independent components may suffer, as shown in Figure 1.

To prevent this effect we introduce a *resource-access budget* $q_s$ in addition to a component's budget $Q_s$, where budget $q_s$

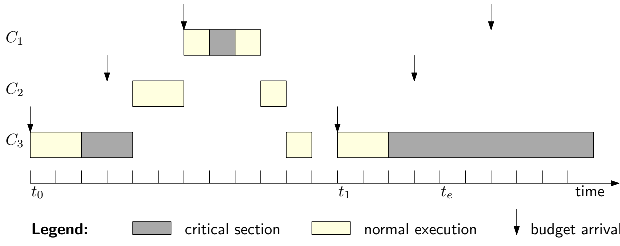


Fig. 1. Temporal isolation is unassured when a component, $C_3$, exceeds its specified critical-section length, i.e. at time instant $t_e$. The system ceiling blocks all other components.

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 [2] or HSRP [1].

*1) Fault containment of critical sections:* Existing SRP-based synchronization protocols in [2], [19] make it possible to choose the local resource ceilings, $rc_{sl}$, according to SRP [6]. In [23] 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 priority higher than the resource ceiling. Task $\tau_{si}$ can therefore lock resource $R_l$ longer than specified, because an interfering task $\tau_{sj}$ (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 [1]. As a result $X_{sl}$ only comprises task execution times within a critical section, i.e.

$$X_{sl} = \max_{1 \leq i \leq n_s} c_{sil}. \tag{3}$$

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

*2) Maintaining SRP ceilings:* To enforce that a task $\tau_{si}$ 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 $\tau_{si}$ 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 the system ceiling decreases after $\tau_{si}$ has executed 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 resources are therefore unsupported. One may alternatively aggregate global resource accesses into a simultaneous lock and unlock of a single artificial resource [24]. Many protocols, or their implementations, lack deadlock avoidance [11], [12], [16], [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.

### B. An overview of B-HSTP properties

This section presented a basic protocol to establish hierarchical synchronization with temporal protection (B-HSTP). Every lock operation to resource $R_l$ replenishes a corresponding resource-access budget $q_s$ with an amount $X_{sl}$. After this resource-access budget has been depleted, the component blocks until its normal budget $Q_s$ replenishes. We can derive the following convenient properties from our protocol:

1) as long as a component behaves according to its timing contract, we strictly follow SRP;
2) because local preemptions are disabled during global resource access and nested critical sections are prohibited, each component can only access a single global resource at a time;
3) similarly, each component can at most keep a single resource in the busy state at a time;
4) each access to resource $R_l$ by task $\tau_{si}$ may take at most $X_{sl}$ budget from budget $Q_s$, where $c_{sil} \leq X_{sl}$.
5) after depleting resource-access budget $q_s$, a task may continue in its component normal budget $Q_s$ with a decreased system ceiling. This guarantees that independent components are no longer blocked by the system ceiling;
6) when a component blocks on a busy resource, it discards all remaining budget until its next replenishment of $Q_s$. This avoids budget suspension, which can lead to scheduling anomalies [25].

As a consequence of property 2, we can use a simple non-preemptive locking mechanism at the local level rather than using SRP. We therefore only need to implement SRP at the global level and we can use a simplified infrastructure at the local level compared to the implementations in [9], [10], [11].

## V. $\mu$C/OS-II AND ITS EXTENSIONS RECAPITULATED

The $\mu$C/OS-II operating system is maintained and supported by Micrium [7], and is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Micrium provides the full $\mu$C/OS-II source code with accompanying documentation [26]. The $\mu$C/OS-II kernel provides preemptive multitasking for up to 256 tasks, and the kernel size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled.

Most real-time operating systems, including $\mu$C/OS-II, do not include a reservation-based scheduler, nor provide means for hierarchical scheduling. In the remainder of this section we outline our realization of such extensions for $\mu$C/OS-II, which are required basic blocks to enable the integration of global synchronization with temporal protection.

### A. Timed Event Management

Intrinsic to our reservation-based component scheduler is timed-event management. This comprises timers to accommodate (*i*) periodic timers at the global level for budget replenishment of periodic servers and at the component level to enforce minimal inter-arrivals of sporadic task activations and (*ii*) virtual timers to track a component's budget. The corresponding timer handlers are executed in the context of the timer interrupt service routine (ISR).

We have implemented a dedicated module to manage *relative timed event queues* (RELTEQs) [27]. The basic idea is to store events relative to each other, by expressing the expiration time of an event relative to the arrival time of the previous event. The arrival time of the head event is relative to the current time, see Figure 2

A *system queue* tracks all server events. Each server has its own *local queue* to track its tasks' events, e.g. task arrivals. When a server is suspended its local queues are deactivated to prevent that expiring events interfere with other servers. When a server resumes, its local queues are synchronized with global time. A mechanism to synchronize server queues with global time is implemented by means of a *stopwatch queue*, which keeps track of the time passed since the last server switch.

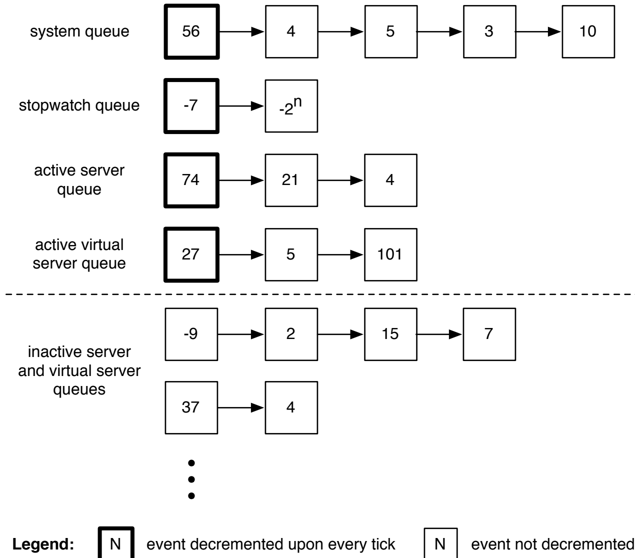


Fig. 2. RELTEQ-based timer management for two-level HSFs.

A dedicated server queue provides support for *virtual timers* to trigger timed events *relative to the consumed budget*. Since an inactive server does not consume any of its budget, a virtual timer queue is not synchronized when a server is resumed. We consider only budget depletion as a virtual event, so that a component can inspect its virtual-timer in constant time.

### B. Server Scheduling

A *server* is assigned to each component to distribute its allocated budget to the component's tasks. A global scheduler is used to determine which server should be allocated the processor at any given time. A local scheduler determines which of the chosen server's tasks should actually execute. Although B-HSTP is also applicable to other server models, we assume that a component is implemented by means of an *idling periodic server* (IPS) [8]. Extending $\mu$C/OS-II with basic HSF support requires a realization of the following concepts:

*1) Global Scheduling:* At the system level a RELTEQ queue is introduced to keep track of server periods. We use a bit-mask to represent whether a server has capacity left. When the scheduler is called, it traverses the RELTEQ and activates the *ready* server with the earliest deadline in the queue. Subsequently, the $\mu$C/OS-II fixed-priority scheduler determines the highest priority ready task within the server.

*2) Periodic Servers:* Since $\mu$C/OS-II tasks are bundled in groups of sixteen to accommodate efficient fixed-priority scheduling, a server can naturally be represented by such a group. The implementation of periodic servers is very similar to implementing periodic tasks using our RELTEQ extensions [27]. An idling server contains an idling task at the lowest, local priority, which is always ready to execute.

*3) Greedy Idle Server:* In our HSF, we reserve the lowest priority level for a idle server, which contains $\mu$C/OS-II's idle task at the lowest local priority. Only if no other server is eligible to execute, then the idle server is switched in.

### C. Global SRP implementation

The key idea of SRP is that when a component needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. Nice properties of SRP are its simple locking and unlocking operations. In turn, during run-time we need to keep track of the system ceiling and the scheduler needs to compare the highest ready component priority with the system ceiling. Hence, a preemption test is performed during run time by the scheduler: A component cannot preempt until its priority is the highest among those of all ready components *and* its priority is higher than the *system ceiling*. In the original formulation of SRP [6], it introduces the notion of preemption-levels. This paper considers FPPS, which makes it possible to unify preemption-levels with priorities.

The system ceiling is a dynamic parameter that changes during execution. Under SRP, a component can only preempt the currently executing component if its priority is higher than the system ceiling. When no resources are locked the system ceiling is zero, meaning that it does not block any tasks from preempting. When a resource is locked, the system ceiling is adjusted dynamically using the resource ceiling, so that the system ceiling represents the highest resource ceiling of a currently locked resource in the system. A run-time

mechanism for tracking the system ceiling can be implemented by means of a stack data structure.

*1) SRP data and interface description:* Each resource accessed using an SRP-based mutex is represented by a `Resource` structure. This structure is defined as follows:

```
typedef struct resource{
    INT8U  ceiling;
    INT8U  lockingTask;
    void*  previous;
} Resource;
```

The `Resource` structure stores properties which are used to track the system ceiling, as explained in below. The corresponding mutex interfaces are defined as follows:

- Create a SRP mutex:
  ```
  Resource* SRPMutexCreate(INT8U ceiling,
                           INT8U *err);
  ```
- Lock a SRP mutex:
  ```
  void SRPMutexLock(Resource* r, INT8U *err);
  ```
- Unlock a SRP mutex:
  ```
  void SRPMutexUnlock(Resource* r);
  ```

The lock and unlock operations only perform bookkeeping actions by increasing and decreasing the system ceiling.

*2) SRP operations and scheduling:* We extended $\mu$C/OS-II with the following SRP rules at the server level:

*a) Tracking the system ceiling:* We use the `Resource` data-structure to implement a *system ceiling stack*. `ceiling` stores the resource ceiling and `lockingTask` stores the identifier of the task currently holding the resource. From the task identifier we can deduct to which server it is attached. The `previous` pointer is used to maintain the stack structure, i.e. it points to the previous `Resource` structure on the stack. The `ceiling` field of the `Resource` on top of the stack represents the current system ceiling.

*b) Resource locking:* When a component tries to lock a resource with a resource ceiling higher than the current system ceiling, the corresponding resource ceiling is pushed on top of the system ceiling stack.

*c) Resource unlocking:* When unlocking a resource, the value on top of the system ceiling stack is popped. The absence of nested critical sections guarantees that the system ceiling represents the resource to be unlocked. The scheduler is called to allow for scheduling ready components that might have arrived during the execution of the critical section.

*d) Global scheduling:* When the $\mu$C/OS-II scheduler is called it calls a function which returns the highest priority ready component. Accordingly to SRP we extend this function with the following rule: when the highest ready component has a priority lower than or equal to the current system ceiling, the priority of the *task* on top of the resource stack is returned. The returned priority serves as a task identifier, which makes easily allows to deduct the corresponding component.

*e) Local scheduling:* The implementations of two-level SRP protocols in [9], [10], [11] also keep track of component ceilings. We only have a binary local ceiling to indicate whether preemptions are enabled or disabled, because we explicitly chose local resource ceilings equal to the highest local priority. During global resource access, the local scheduler can only select the resource-accessing task for execution.

## VI. B-HSTP IMPLEMENTATION

In this section we extend the framework presented in Section V with our proposed protocol, B-HSTP. In many microkernels, including $\mu$C/OS-II, the only way for tasks to share data structures with ISRs is by means of disabling interrupts. We therefore assume that our primitives execute non-preemptively with interrupts disabled.

Because critical sections are non-nested and local preemptions are disabled, at most one task $\tau_{si}$ at a time in each component may use a global resource. This convenient system property makes it possible to multiplex both resource-access budget $q_s$ and budget $Q_s$ on a single budget timer by using our virtual timer mechanism. The remaining budget $Q_s^{\mathtt{rem}}(t)$ is returned by a function that depends on the virtual timers mechanism, see Section V-A. A task therefore merely blocks on its component's budget, which we implement by adjusting the single available budget timer $Q_s^{\mathtt{rem}}(t)$.

*1) Resource locking:* The lock operation updates the local ceiling to prevent other tasks within the component from interfering during the execution of the critical section. Its pseudo-code is presented in Algorithm 1.

In case we have enabled SIRAP, rather than HSRP's overrun, there must be sufficient remaining budget within the server's current period in order to successfully lock a resource. If the currently available budget $Q_s^{\mathtt{rem}}(t)$ is insufficient, the task will spinlock until the next replenishment event expires. To avoid a race-condition between a resource unlock and budget depletion, we require that $Q_s^{\mathtt{rem}}(t)$ is strictly larger than $X_{sr}$ before granting access to a resource $R_r$.

**Algorithm 1** void HSF_lock(`Resource*` r);

1: $updateComponentCeiling(r)$;
2: **if** HSF_MUTEX_PROTOCOL == SIRAP **then**
3:     **while** $X_{sr} >= Q_s^{\mathtt{rem}}(t)$ **do** {apply SIRAP's self-blocking}
4:         $enableInterrups$;
5:         $disableInterrups$;
6:     **end while**
7: **end if**
8: **while** $r.status = busy$ **do** {self-suspend on a busy resource}
9:     $setComponentBudget(0)$;
10:     $enableInterrups$;
11:     $Schedule()$;
12:     $disableInterrups$;
13: **end while**
14: $Q_s^{\nabla} \leftarrow Q_s^{\mathtt{rem}}(t)$;
15: $setComponentBudget(X_{sr})$;
16: $C_s.lockedResource \leftarrow r$;
17: $r.status \leftarrow locked$
18: $SRPMutexLock(r)$;

A task may subsequently block on a busy resource, until it becomes free. When it encounters a busy resource, it suspends the component and discards all remaining budget. When the resource becomes free and the task which attempted to lock the resource continues its execution, it is guaranteed that there is sufficient budget to complete the critical section (assuming that it does not exceed its specified length, $X_{sr}$). The reason for this is that a component discards its budget when it blocks on a busy resource and can only continue with a fully replenished budget. This resource holding time $X_{sr}$ defines

the resource-access budget of the locking task and component. The component's remaining budget is saved as $Q_s^{\triangledown}$ and reset to $X_{sl}$ before the lock is established.

$setComponentBudget(0)$, see line 9, performs two actions: (*i*) the server is blocked to prevent the scheduler from rescheduling the server before the next replenishment, and (*ii*) the budget-depletion timer is canceled.

*2) Resource unlocking:* Unlocking a resource means that the system and component ceilings must be decreased. Moreover, the amount of consumed budget is deducted from the components stored budget, $Q_s^{\triangledown}$. We do not need to restore the component's budget, if the system ceiling is already decreased at the depletion of its resource-access budget, i.e. when a component has exceeded its specified critical-section length. The unlock operation in pseudo-code is as follows:

**Algorithm 2** void HSF_unlock(`Resource*` r);

```
1: updateComponentCeiling(r);
2: r.status ← free;
3: C_s.lockedResource ← 0;
4: if System_ceiling == RC_r then
5:     setComponentBudget(max(0, Q_s^▽ − (X_sr − Q_s^rem(t))));
6: else
7:     ; {we already accounted the critical section upon depletion of X_sr}
8: end if
9: SRPMutexUnlock(r);
```

*3) Budget depletion:* We extend the budget-depletion event handler with the following rule: if any task within the component holds a resource, then the global resource ceiling is decreased according to SRP and the resource is marked busy. A component $C_s$ may continue in its restored budget with a decreased system ceiling. The pseudo-code of the budget-depletion event handler is as follows:

**Algorithm 3** on budget depletion:

```
1: if C_s.lockedResource ≠ 0 then
2:     r ← C_s.lockedResource;
3:     r.status ← busy;
4:     SRPMutexUnlock(r);
5:     setComponentBudget(max(0, Q_s^▽ − X_sr));
6: else
7:     ; {apply default budget-depletion strategy}
8: end if
```

*4) Budget replenishment:* For each periodic server an event handler is periodically executed to recharge its budget. We extend the budget-replenishment handler with the following rule: if any task within the component holds a resource busy, then the global resource ceiling is increased according to SRP and the resource-access budget is replenished with $X_{sr}$ of resource $R_r$. A component $C_s$ may continue in its restored budget with an increased system ceiling for that duration, before the remainder of its normal budget $Q_s$ becomes available. The pseudo-code of this event handler is shown in Algorithm 4.

**Algorithm 4** on budget replenishment:

```
1: Q_s^▽ ← Q_s;
2: if C_s.lockedResource ≠ 0 then {C_s keeps a resource busy}
3:     r ← C_s.lockedResource;
4:     setComponentBudget(X_sr);
5:     SRPMutexLock(r);
6: else
7:     ; {Apply default replenishment strategy}
8: end if
```

## VII. Experiments and results

This section evaluates the implementation costs of B-HSTP. First, we present a brief overview of our test platform. Next, we experimentally investigate the system overhead of the synchronization primitives and compare these to our earlier protocol implementations. Finally, we illustrate B-HSTP by means of an example system.

### A. Experimental setup

We recently created a port for $\mu$C/OS-II to the OpenRISC platform [13] to experiment with the accompanying cycle-accurate simulator. The OpenRISC simulator allows software-performance evaluation via a cycle-count register. This profiling method may result in either longer or shorter measurements between two matching calls due to the pipelined OpenRISC architecture. Some instructions in the profiling method interleave better with the profiled code than others. The measurement accuracy is approximately 5 instructions.

### B. Synchronization overheads

In this section we investigate the overhead of the synchronization primitives of B-HSTP. By default, current analysis techniques do not account for overheads of the corresponding synchronization primitives, although these overheads become of relevance upon deployment of such a system. Using more advanced analysis methods, for example as proposed in [28], these measures can be included in the existing system analysis. The overheads introduced by the implementation of our protocol are summarized in Table I and compared to our earlier implementation of HSRP and SIRAP in [9], [10].

*1) Time complexity:* Since it is important to know whether a real-time operating system behaves in a time-wise predictable manner, we investigate the disabled interrupt regions caused by the execution of B-HSTP primitives. Our synchronization primitives are independent of the number of servers and tasks in a system, but introduce overheads that interfere at the system level due to their required timer manipulations.

Manipulation of timers makes our primitives more expensive than a straightforward two-level SRP implementation. However, this is the price for obtaining temporal isolation. The worst-case execution time of the lock operation increases with 380 instructions in every server period in which a component blocks, so that the total cost depends on the misbehaving critical-section length which causes the blocking. The budget replenishment handler only needs to change the amount to be replenished, so that B-HSTP itself does not contribute much to the total execution time of the handler. These execution times of the primitives must be included in the system analysis by adding these to the critical section execution times, $X_{sl}$. At the local scheduling level B-HSTP is more efficient, however, because we use a simple non-preemptive locking mechanism.

TABLE I
BEST-CASE (BC) AND WORST-CASE (WC) EXECUTION TIMES FOR SRP-BASED PROTOCOLS, INCLUDING B-HSTP, MEASURED ON THE OPENRISC PLATFORM IN NUMBER OF PROCESSOR INSTRUCTIONS.

| **Event** | **single-level SRP** [10] | | **HSRP** (see [9], [10]) | | **SIRAP** (see [9], [10]) | | **B-HSTP** | |
|---|---|---|---|---|---|---|---|---|
| | BC | WC | BC | WC | BC | WC | BC | WC |
| Resource lock | 124 | 124 | 196 | 196 | 214 | 224 | 763 | - |
| Resource unlock | 106 | 106 | 196 | 725 | 192 | 192 | 688 | 697 |
| Budget depletion | - | - | 0 | 383 | - | - | 59 | 382 |
| Budget replenishment | - | - | 0 | 15 | - | - | 65 | 76 |

*2) Memory complexity:* The code sizes in bytes of B-HSTP's lock and unlock operations, i.e. 1228 and 612 bytes, is higher than the size of plain SRP, i.e. 196 and 192 bytes. This includes the transparently implemented self-blocking and overrun mechanisms and timer management. $\mu$C/OS-II's priority-inheritance protocol has similar sized lock and unlock primitives, i.e. 924 and 400 bytes.

Each SRP resource has a data structure at the global level, i.e. we have $M$ shared resources. Each component only needs to keep track of a single globally shared resource, because local preemptions are disabled during global resource access. However, each component $C_s$ needs to store its resource-access durations for all its resources $R_l \in \mathcal{R}_s$.

### *C. SIRAP and HSRP re-evaluated*

From our first implementation of SIRAP and HSRP, we observed that SIRAP induces overhead locally within a component, i.e. the spin-lock, which checks for sufficient budget to complete the critical section, adds to the budget consumption of the particular task that locks the resource. SIRAP's overhead consists at least of a single test for sufficient budget in case the test is passed. The overhead is at most two of such tests in case the initial test fails, i.e. one additional test is done after budget replenishment before resource access is granted. All remaining tests during spinlocking are already included as self-blocking terms in the local analysis [2]. The number of processor instructions executed for a single test is approximately 10 instructions on our test platform.

HSRP introduces overhead that interferes at the global system level, i.e. the overrun mechanism requires to manipulate event timers to replenish an overrun budget when the normal budget of a component depletes. This resulted in a relatively large overhead for HSRP's unlock operation compared to SIRAP, see Table I. Since similar timer manipulations are required for B-HSTP, the difference in overhead for HSRP and SIRAP becomes negligible when these protocols are complemented with means for temporal isolation. Furthermore, the absolute overheads are in the same order of magnitude.

### *D. B-HSTP: an example*

We have recently extended our development environment with a visualization tool, which makes it possible to plot a HSF's behaviour [29] by instrumenting the code, executed on the OpenRISC platform, of our $\mu$C/OS-II extensions. To demonstrate the behavior of B-HSTP, consider an example system comprised of three components (see Table II) each with two tasks (see Table III) sharing a single global resource $R_1$. We use the following conventions:

1) the component or task with the lowest number has the highest priority;
2) the computation time of a task is denoted by the consumed time units after locking and before unlocking a resource. For example, the scenario – $E_{s1,1}$; $Lock(R_1)$; $E_{s1,2}$; $Unlock(R_1)$; $E_{s1,3}$ – is denoted as $E_{s1,1} + E_{s1,2} + E_{s1,3}$ and the term $E_{s1,2}$ represents the critical-section length, $c_{s1l}$;
3) the resource holding time is longest critical-section length within a component, see Equation 3.
4) the component ceilings of the shared resource, $R_1$, are equal to the highest local priority, as dictated by B-HSTP.

The example presented in Figure 3 complements HSRP's overrun mechanism with B-HSTP.

TABLE II
EXAMPLE SYSTEM: COMPONENT PARAMETERS

| **Server** | **Period** ($P_s$) | **Budget** ($Q_s$) | **Res. holding time** ($X_s$) |
|---|---|---|---|
| IPS 1 | 110 | 12 | 4.0 |
| IPS 2 | 55 | 8 | 0.0 |
| IPS 3 | 50 | 23 | 7.4 |

TABLE III
EXAMPLE SYSTEM: TASK PARAMETERS

| **Server** | **Task** | **Period** | **Computation time** |
|---|---|---|---|
| IPS 1 | Task 11 | 220 | 6.5 +4.0+6.5 |
| IPS 1 | Task 12 | 610 | 0.0+0.17+0.0 |
| IPS 2 | Task 21 | 110 | 5.0 |
| IPS 2 | Task 22 | 300 | 7.0 |
| IPS 3 | Task 31 | 100 | 12+7.4+12 |
| IPS 3 | Task 32 | 260 | 0.0+0.095+0.0 |

At every time instant that a task locks a resource, the budget of the attached server is manipulated according to the rules of our B-HSTP protocol, e.g. see time 11 where task 31 locks the global resource and the budget of IPS 3 is changed. After task 31 has executed two resource accesses within its specified length, in the third access it gets stuck in an infinite loop, see time instant 211. Within IPS 3, the lower priority task is indefinitely blocked, since B-HSTP does not concern the local schedulability of components. IPS 1 blocks on the busy resource at time instant 227 and cannot continue further until the resource is released. However, the activations of the independent component, implemented by IPS 2, are unaffected, because IPS 3 can only execute 7.4

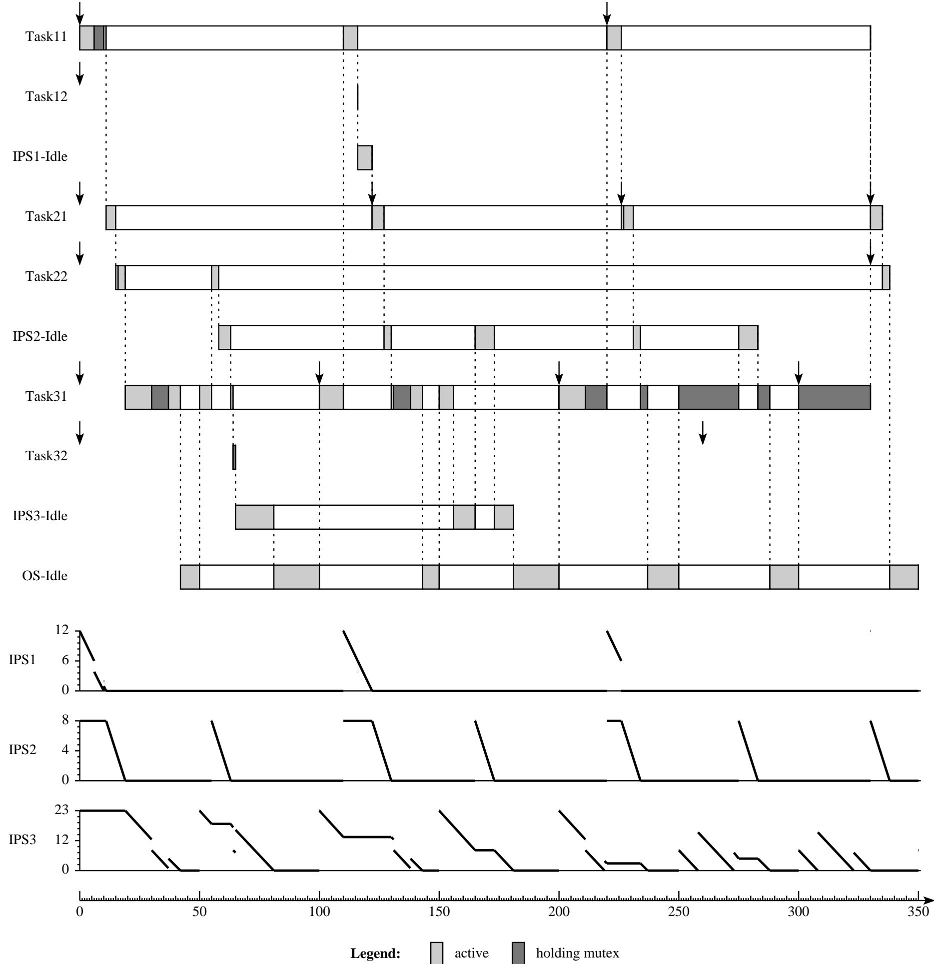


Fig. 3. Example trace, generated from instrumented code [29], combining HSRP and B-HSTP to arbitrate access between IPS 1 and IPS 3 to a single shared resource. IPS 2 is independent and continues its normal execution even when task 31 exceeds its specified critical-section length, i.e. starting at time 219. IPS 1 blocks on a busy resource and looses its remaining budget at time 227.

time units with a raised system ceiling, e.g. see time interval $[220, 235]$ where IPS 3 gets preempted after by IPS 1 that blocks on the busy resource and IPS 2 that continues its execution normally. Moreover, IPS 3 may even use its overrun budget to continue its critical section with a decreased system ceiling, see time interval $[273, 280]$, where IPS 3 is preempted by IPS 2. This is possible due to the inherent pessimism in the overrun analysis [1], [19] which allocates an overrun budget at the global level without taking into account that in normal situations the system ceiling is raised for that duration.

## VIII. Discussion

### A. Component ceilings revisited

We assume locally non-preemptive critical sections, which may reduce the component's schedulability. Suppose we allow preemptions of tasks that are not blocked by an SRP-based component ceiling, see (2). The blocking times of all tasks with a lower preemption level than the component ceiling do not change, providing no advantage compared to the case where critical sections are non-preemptive. Moreover, enforcement of critical-section lengths is a prerequisite to guarantee temporal isolation in HSFs, see Section IV.

As a solution we could introduce an intermediate reservation level assigned and allocated to critical sections. In addition, we need to enforce that blocking times to other components are not exceeded due to local preemptions [12]. This requires an extension to our two-level HSF and therefore complicates an implementation. It also affects performance, because switching between multiple budgets for each component (or task) is costly [16] and breaks SRP's stack-sharing property.

### B. Reliable resource sharing

To increase the reliability of the system, one may artificially increase the resource-access budgets, $X_{sl}$, to give more slack to an access of length $c_{sil}$ to resource $R_l$. Although this alleviates small increases in critical-section lengths, it comes at the cost of a global schedulability penalty. Moreover, an increased execution time of a critical section of length $c_{sil}$ up to $X_{sl}$ should be compensated with additional budget to

guarantee that the other tasks within the same component make their deadlines. Without this additional global schedulability penalty, we may consume the entire overrun budget $X_s$ when we choose HSRP to arbitrate resource access, see Figure 3, because the analysis in [1], [19] allocate an overrun budget to each server period at the server's priority level. In line with [1], [19], an overrun budget is merely used to complete a critical section. However, we leave budget allocations while maximizing the system reliability as a future work.

### C. Watchdog timers revisited

If we reconsider Figure 1 and Figure 3 we observe that the time-instant at which a maximum critical-section length is exceeded can be easily detected using B-HSTP, i.e. when a resource-access budget depletes. We could choose to execute an error-handler to terminate the task and release the resource at that time instant, similar to the approach proposed in AUTOSAR. However, instead of using expensive timers, we can defer the execution of such an error handler until component $C_s$ is allowed to continue its execution. This means that the error handler's execution is accounted to $C_s$' budget of length $Q_s$. A nice result is that an eventual user call-back function can no longer hamper temporal isolation of other components than those involved in resource sharing.

## IX. Conclusion

This paper presented B-HSTP, an SRP-based synchronization protocol, which achieves temporal isolation between independent components, even when resource-sharing components misbehave. We showed that it generalizes and extends existing protocols in the context of HSFs [1], [2]. Prerequisites to dependable resource sharing in HSFs are mechanisms to enforce and monitor maximum critical-section lengths. We followed the choice in [1] to make critical sections non-preemptive for tasks within the same component, because this makes an implementation of our protocol efficient. The memory requirements of B-HSTP are lower than priority-inheritance-based protocols where tasks may pend in a waiting queue. Furthermore, B-HSTP primitives have bounded execution times and jitter. Both HSRP [1] and SIRAP [2], which each provide a run-time mechanism to prevent budget depletion during global resource access, have a negligible difference in implementation complexity when complemented with B-HSTP. Our protocol therefore promises a reliable solution to future safety-critical industrial applications that may share resources.

## References


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

[2] 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.

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

[4] AUTOSAR GbR, "Technical overview," 2008. [Online]. Available: http://www.autosar.org/

[5] 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.

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

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

[8] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Real-Time Systems Symp.*, Dec. 2005, pp. 389–398.

[9] 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.

[10] M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien, and M. Behnam, "Extending a HSF-enabled open-source real-time operating system with resource sharing," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010, pp. 71–81.

[11] 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.

[12] 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.

[13] OpenCores. (2009) OpenRISC overview. [Online]. Available: http://www.opencores.org/project,or1k

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] J. J. Labrosse, *Microc/OS-II*. R & D Books, 1998.

[27] M. M. H. P. van den Heuvel, M. Holenderski, R. J. Bril, and J. J. Lukkien, "Constant-bandwidth supply for priority processing," *IEEE Trans. on Consumer Electronics*, vol. 57, no. 2, May 2011.

[28] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *Real-Time Systems Symp.*, Dec. 2003, pp. 25–36.

[29] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, July 2010, pp. 37–42.