Extending a HSF-enabled Open-Source Real-Time Operating System with Resource Sharing

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

Moris Behnam
Real-Time Systems Design Group
Mälardalen Real-Time Research Centre
P.O. Box 883, SE-721 23 Västerås, Sweden

***Abstract*—Hierarchical scheduling frameworks (HSFs) provide means for composing complex real-time systems from well-defined, independently analyzed subsystems. 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]. This paper describes an implementation to provide such HSFs with SRP-based synchronization protocols. We base our implementations on the commercially available real-time operating system μC/OS-II, extended with proprietary support for periodic tasks, idling periodic servers and two-level fixed priority preemptive scheduling. Specifically, we show the implementation of SRP as a local synchronization protocol, and present the implementation of both HSRP and SIRAP. Moreover, we investigate the system overhead induced by the synchronization primitives of each protocol. Our aim is that these protocols can be used side-by-side within the same HSF, so that 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 applications and (*ii*) integration of applications 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]. In such *open environments* [4], an application 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. Temporal isolation between applications is provided by allocating a *budget* to each subsystem. In this paper we assume a one-to-one relation between applications and subsystems.

To accommodate resource sharing between fixed-priority scheduled subsystems, two synchronization protocols have been proposed based on the *stack resource policy* (SRP) [5], i.e. HSRP [1] and SIRAP [2]. A HSF extended with such a protocol makes it possible to share logical resources between arbitrary tasks, which are located in arbitrary subsystems, in a mutually exclusive manner. A resource that is used in more than one subsystem is denoted as a *global shared resource*. A resource that is only shared within a single subsystem is a *local shared resource*. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of the corresponding subsystem's budget, excessive blocking periods can occur which may hamper the correct timeliness of other subsystems [6]. To prevent the depletion of a subsystem's budget during a global resource access SIRAP uses a skipping mechanism. Contrary, HSRP uses an overrun mechanism (with or without payback), i.e. the overrun mechanism reacts upon a budget depletion during a global resource access by temporarily increasing the budget with a statically determined amount for the duration of that resource access. The relative strengths of HSRP and SIRAP heavily depend on system characteristics [7], which makes it attractive to support both within the same HSF. In this paper we present the implementation of the synchronization primitives for both synchronization protocols to support global (i.e. inter-subsystem) resource sharing by extending a HSF-enabled μC/OS-II operating system. The choice of operating system is in line with our industrial and academic partners.

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

A. Problem Description

Most off-the-shelf real-time operating systems, including μC/OS-II, do not provide an implementation for SRP nor hierarchical scheduling. We have extended μC/OS-II with support for periodic tasks, idling periodic servers [8] and two-level fixed priority preemptive scheduling (FPPS). Although global resource sharing protocols within HSFs are extensively investigated for ideal system models, implementations are lacking within real-time operating systems. Moreover, the run-time overhead of these protocols is unknown and not included in these models. These overheads become relevant during deployment of such a resource sharing open environment.

B. Contributions

The contribution of this paper is fivefold. First, we present an implementation of SRP within μC/OS-II as a local (i.e. intra-subsystem) resource access protocol. We aim at a modular design, so that one can choose between the original priority inheritance implementation, or our SRP implementation. We show that our SRP implementation improves the existing μC/OS-II implementation for mutual exclusion by lifting several limitations and simplifying the implementation. We restrict this implementation to single unit resources and single processor platforms. Second, we present an implementation

for HSRP [1] and SIRAP [2] to support global (i.e. inter-subsystem) resource sharing within a two-level fixed priority hierarchically scheduled system. We aim at unified interfaces for both protocol implementations to ease the integration of HSRP and SIRAP within the same HSF. Third, we compare the system overhead of the primitives of both synchronization protocols. Fourth, we show how HSRP and SIRAP can be integrated side-by-side within the same HSF. Inline with μC/OS-II we allow to enable or disable each protocol extension during compile time. Finally, we show that our protocol implementations follow a generic design, i.e. the μC/OS-II specific code is limited.

C. Overview

The remainder of this paper is as follows. Section II describes the related work. Section III presents background information on μC/OS-II. Section IV summarizes our basic μC/OS-II extensions for periodic tasks, idling periodic servers and a two-level fixed priority scheduled HSF based on a timed event management system [9, 10]. Section V describes SRP at the level of local resource sharing, including its implementation. Section VI describes common terminology and implementation efforts for SRP at the level of global resource sharing. Section VII and VIII describe the implementation of global SRP-based synchronization protocols, i.e. SIRAP and subsequently HSRP. In Section IX we compare both protocol implementations. Section X discusses the challenges towards a system supporting both protocols side-by-side. Finally, Section XI concludes the paper.

II. Related work

In literature, several implementations are presented to provide temporal isolation between dependent applications. De Niz et al. [11] support resource sharing between reservations based on PCP in their fixed priority scheduled Linux/RK resource kernel. Buttazzo and Gai [12] implemented a reservation-based earliest deadline first (EDF) scheduler for the real-time ERIKA Enterprise kernel, including SRP-based synchronization support. However, both approaches are not applicable in open environments [4], because they lack a distinct support for local and global synchronization. Contrary, Behnam et al. [13] implemented a HSF on top of the real-time operating system VxWorks, but do not support synchronization between applications. Although resource sharing between applications within HSFs has been extensively investigated in literature, e.g. [1, 2, 7, 14, 15, 16, 17], none of the above implementations provide such synchronization primitives. In this paper we describe the implementation of such synchronization primitives by further extending our HSF-enabled μC/OS-II operating system. Looking at existing industrial real-time systems, FPPS is the de-facto standard of task scheduling. Having such support will simplify migration to and integration of existing legacy applications into the HSF, avoiding unbridgeable technology revolutions for engineers. In the remainder of this section we provide a brief overview of both local and global synchronization protocols.

A. Local synchronization

The priority inversion problem [18] can be prevented by the *priority inheritance protocol* (PIP). PIP makes a task inherit the highest priority of any other tasks that are waiting on a resource the task uses. A disadvantage of PIP is that it suffers from the chained blocking and deadlock problem [18]. As a solution, Sha et al. [18] proposed the *priority ceiling protocol* (PCP). An easier to implement alternative to PCP is the *highest locker protocol* (HLP). HLP uses an off-line computed ceiling to raise a task's priority when it accesses a resource. In case of HLP a task is already prevented from starting its execution when a resource is accessed by another task sharing this resource, whereas PCP postpones increasing of priorities until a blocking situation occurs. As an alternative to PCP, Baker [5] presented the *stack resource policy* (SRP). SRP has an easier implementation and induces less context switching overhead compared to PCP, and supports dynamic priority scheduling policies. Because of its wide applicability, ease of implementation and its apparent improvements of the existing synchronization protocol, we have decided to extend μC/OS-II with SRP, as presented in this paper.

B. Global synchronization

Recently, three SRP-based synchronization protocols for inter-subsystem resource sharing between tasks have been presented. Their relative strength depends on various system parameters [15]. BROE [14] considers resource sharing under EDF scheduling. Most commercial operating systems, including μC/OS-II, do not implement an EDF scheduler. Both HSRP [1] and SIRAP [2] assume FPPS. In order to deal with resource access while a subsystem's budget depletes, HSRP uses a run-time overrun mechanism [1]. The original analysis of HSRP [1] does not allow for integration in open environments due to the lacking support for independent analysis of subsystems. Behnam et al. [16] lifted this limitation, enabling the full integration of HSRP within HSFs. Alternatively, SIRAP uses a skipping approach to prevent budget depletion inside a critical section [2]. Recently, both synchronization protocols are analytically compared and their impact on the total system load for various system parameters is analyzed using simulations [7]. In this paper we implement both protocols within μC/OS-II, and compare the efficiency of the corresponding primitives.

III. μC/OS-II: A Brief Overview

The μC/OS-II operating system is maintained and supported by Micrium [19], and is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Micrium provides the full μC/OS-II source code with accompanying documentation [20]. The μC/OS-II kernel provides preemptive multitasking, and the kernel size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled.

A. μC/OS-II Task Scheduling

The number of tasks that can run within μC/OS-II is 64 by default, and can be extended to 256 by altering the configuration. Each group of 8 tasks is assigned a bit in the ready-mask, updated during run-time to indicate whether a task is ready to run within this group. The ready-mask allows for optimized fixed priority scheduling by performing efficient bit comparison operations to determine the highest priority ready task to run, as explained in more detail in [20, Section 3].

B. μC/OS-II Synchronization Protocol

The standard μC/OS-II distribution supports synchronization primitives by means of *mutexes* [20]. However, it is not clearly stated which synchronization protocol they implement. Lee and Kim [21] attempted to identify the protocol by analyzing the source code. The presence of a ceiling in the mutex interface suggested the implementation of HLP. However, we have analyzed the behavior in Figure 1 of two tasks that reserve two resources in opposite order[2]. Since we observe a deadlock, μC/OS-II does not seem to implement HLP or an other deadlock avoidance protocol.

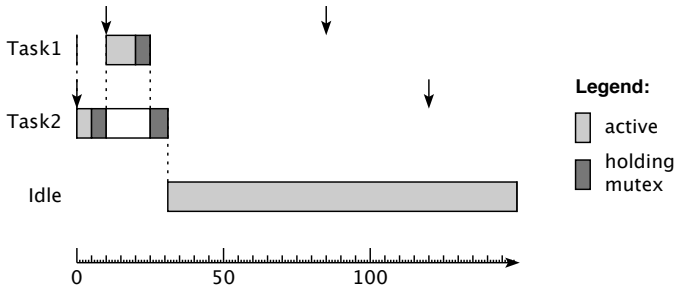

Fig. 1. The mutex implementation in μC/OS-II suffers from the deadlock problem, inherited from the PIP definition. The task-set $\mathcal{T}$ contains tasks τ_1 and τ_2, where τ_1 has the highest priority. τ_1 first locks resource R_1 and subsequently R_2, has a computation time[2] $C_1 = 10 + 5 + 5 + 5 + 5$, and a phasing $\varphi_1 = 10$. τ_2 first locks resource R_2 and subsequently R_1, has a computation time $C_2 = 5 + 10 + 25 + 10 + 5$, and no phasing (i.e. $\varphi_2 = 0$).

μC/OS-II only supports a *single task* on each priority-level, independent of its execution state, because a priority is also used as a task identifier. A priority-level is assigned to each resource on creation of the resource. The priority-level is used to raise the priority of a task when it blocks a higher priority task, which is named *priority calling* in the μC/OS-II terminology. Because of the assignment of a unique priority to each resource, the *transparent* character of PIP is lost. In the original PIP a priority is dynamically raised to the priority of the task that is pending on a resource, which does not require any off-line calculated information. Literature [21] outlines an implementation of PIP within μC/OS-II lifting the limitation on reserving a priority level.

[2]The fragmented computation time C_i denotes the task's consumed time units before/after locking/unlocking a resource, e.g. the scenario $C_{1,1} - Lock(R_1) - C_{1,2} - Lock(R_2) - C_{1,3} - Unlock(R_2) - C_{1,4} - Unlock(R_1) - C_{1,5}$ is denoted as $C_{1,1} + C_{1,2} + \ldots + C_{1,5}$.

IV. Basic μC/OS-II Extensions Recapitulated

In this paper, we consider a HSF with two-level FPPS, where a system $\mathcal{S}$ is composed of a set of *subsystems*, each of which is composed of a set of *tasks*. A *server* is allocated to each subsystem $S_s \in \mathcal{S}$. 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. Given such a HSF mapped on a single processor, we assume that a subsystem is implemented by means of an *idling periodic server* [8]. However, the proposed approach is expected to be easily adaptable to other server models. A server has a replenishment period P_s and a budget Q_s, which together define a *timing interface* $S_s(P_s, Q_s)$ associated with each subsystem S_s. We say that tasks assigned to a server consume processor time *relative to the server's budget* to signify that the consumed processor time is accounted to (and subtracted from) that budget.

Most real-time operating systems, including μC/OS-II, do not include a reservation-based scheduler, nor provide means for hierarchical scheduling. Although some real-time operating systems provide primitives to support periodic tasks, e.g. RTAI/Linux [22], μC/OS-II does not. In the remainder of this section we outline our realization of such extensions for μC/OS-II, which are required basic blocks to enable the integration of global synchronization.

A. Timed Event Management

Real-time systems need to schedule many different timed events (e.g. programmed delays, arrival of periodic tasks and budget replenishment) [9]. On contemporary computer platforms, however, the number of hardware timers is usually limited, meaning that events need to be multiplexed on the available timers. Facing these challenges led to the invention of RELTEQ [9].

The basic idea behind RELTEQ is to store the arrival times of events relative to each other, by expressing the arrival 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, as shown in Figure 2. While RELTEQ is not restricted to any specific hardware timer, in this paper we assume a periodic timer. At every tick of the periodic timer the time of the head event in the queue is decremented.

Two operations can be performed on an event queue: new events can be inserted and the head event can be popped. When a new event e_i with absolute time t_i is inserted, the event queue has to be traversed, accumulating the relative times of the events until a later event e_j is found, with $t_i < t_j$, where t_i and t_j are both absolute times. When such an event is found, then *(i)* e_i is inserted before e_j, *(ii)* its time is set relative to the previous event, and *(iii)* the arrival time of e_j is set relative to e_i. If no later event was found, then e_i is appended at the end of the queue, and its time is set relative to the previous event.

In [10] we proposed a technique to extend RELTEQ with the aim to minimize the overhead of handling events belonging

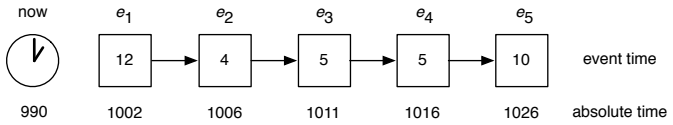

Fig. 2. Example of the RELTEQ event queue.

to inactive servers. To support hierarchical scheduling, we introduced a *server queue* for *each* server to keep track of the events local to the server. At any time at most one server can be active; all other servers are inactive. A *stopwatch queue* keeps track of the time passed since the last server switch, which provides a mechanism to synchronize the server queues with the global time upon a server context switch. Finally, we introduced the notion of a *virtual event*, which are timed events *relative to the consumed budget*, e.g. budget depletion. An additional server event queue that is not synchronized with global time upon a server context-switch implements the infrastructure to support virtual events.

B. Periodic Task Scheduling

The implemented RELTEQ extensions within μC/OS-II easily allow the support for periodic tasks. Because different μC/OS-II services can influence the state of a task, we do not directly alter the task's state. Instead, a periodic task is characterized by an infinite loop which executes its periodic workload and subsequently pends on a semaphore. The event handler corresponding to RELTEQ's activation event releases the pending task and inserts a new event for the next period.

C. Simplified Server Scheduling

Extending the standard μC/OS-II scheduler with basic HSF support requires the identification and realization of the following concepts:

1) Applications: An application can be modeled as a set of tasks. Since μC/OS-II tasks are bundled in groups of eight to accommodate efficient fixed priority scheduling, as explained in Section III-A, a server can naturally be represented by a multiple of eight tasks.

2) Idling Periodic Servers: A realization of the idling periodic server is very similar to the implementation of a periodic task using our RELTEQ extensions [10], with the difference that the server structures do not require additional semaphores. An idling task is contained in all servers at the lowest local priority.

3) Two-level FPPS-based HSF: Similar to the existing μC/OS-II task scheduling approach, we introduce an additional bit-mask to represent whether a server has capacity left. When the scheduler is called it determines the highest priority server with remaining capacity, and hides all tasks from other servers for the local scheduler. Subsequently, the local scheduler determines the highest priority ready task within the server.

V. Stack Resource Policy Implementation

As a supportive step towards global synchronization, first the SRP protocol is summarized, followed by the implementation description of the SRP primitives. Note that in its original formulation SRP introduces the notion of preemption-levels. In this paper we consider FPPS, which allows to unify preemption-levels with task priorities.

A. SRP Recapitulated

The key idea of SRP is that when a task needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. Therefore a preemption test is performed during runtime by the scheduler: A task is not permitted to preempt until its priority is the highest among those of all ready tasks *and* its priority is higher than the *system ceiling*.

1) Resource Ceiling: Each resource is assigned a static, off-line calculated ceiling, which is defined as the maximum priority of any task that shares the resource.

2) System Ceiling: The system ceiling is defined as the maximum of the resource ceilings of all currently locked resources. 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. A run-time mechanism for tracking the system ceiling can be implemented by means of a stack.

B. 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 the next subsection. The corresponding mutex interfaces are defined as follows:

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

C. SRP Primitive and Data-structure Implementation

Nice properties of the SRP are its simple locking and unlocking operations. Moreover, SRP allows to share a single stack between all tasks within an application. In turn, during run-time we need to keep track of the system ceiling and the scheduler needs to compare the highest ready task priority with the system ceiling.

1) 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. 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.

2) Resource Locking: When a task 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.

3) Resource Unlocking: When unlocking a resource, the value on top of the system ceiling stack is popped if the corresponding resource holds the current system ceiling. The scheduler is called to allow for scheduling ready tasks that might have arrived during the execution of the critical section.

4) Scheduling: When the μC/OS-II scheduler is called it calls a function which returns the highest priority ready task. Accordingly to SRP we extend this function with the following rule: when the highest ready task 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.

D. Evaluation

To show that our SRP-based implementation improves on the standard mutex implementation we have simulated the same task set as in Figure 1. The resulting trace in Figure 3 shows that our SRP implementation successfully handles nested critical sections, whereas the priority inheritance implementation causes a deadlock of the involved tasks.

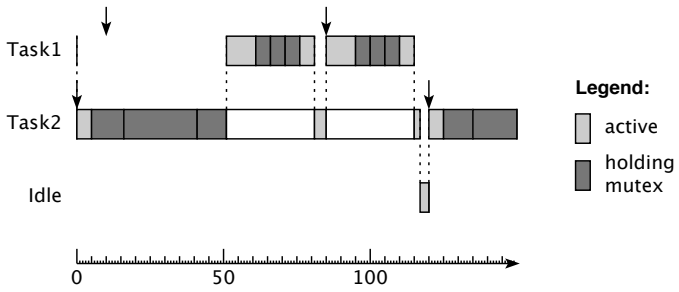

Fig. 3. Using the SRP mutexes the deadlock problem for nested resources is resolved. The task parameters are equal to the example in Section III-B.

Moreover, our implementation reduces the amount of source code: μC/OS-II's PIP implementation consists of 442 lines of code (excluding comments) versus 172 lines of code for our SRP implementation. Additionally, SRP avoids keeping track of the waiting tasks, i.e. it is more processor time and memory space efficient, and lifts the limitation to reserve a priority for each resource.

VI. Global SRP-based Synchronization

Both HSRP and SIRAP can be used for independent development of subsystems and support subsystem integration in the presence of globally shared resources [2, 16]. Besides, both protocols use SRP to synchronize global resource access, and therefore parts of their implementations are common, as described in this section.

A. Definitions

Lifting SRP to a two-level HSF requires to extend our notion of a ceiling compared to the original SRP.

1) Resource ceiling: With every global resource two types of resource ceilings are associated; a *global* resource ceiling for global scheduling and a *local* resource ceiling for local scheduling. These ceilings are defined according the SRP.

2) System/subsystem ceiling: The system/subsystem ceilings are dynamic parameters that change during execution. The system/subsystem ceiling is equal to the highest global/local resource ceiling of a currently locked resource in the system/subsystem. Under SRP, a task can only preempt the currently executing task (even when accessing a global resource) if its priority is higher than its subsystem ceiling. A similar condition for preemption holds for subsystems.

B. Extending the SRP Data Structures

Each global resource accessed using an SRP-based mutex is represented by a `Resource` structure. Additionally, the resource is represented by a `localResource` structure defined as follows:

```
typedef struct {
  struct resource* globalResource;
  INT8U    localCeiling;
  INT8U    localLockingTask;
  void*    previous;
} localResource;
```

The `localResource` structure stores properties which are used to track the subsystem ceiling, as explained in the next subsection.

C. Tracking the Subsystem/System Ceiling

Similar to the SRP implementation we need to maintain a stack for the global and local resource ceilings. The global stack is represented by the stack implementation described in Section V-C. A global mutex creates a normal SRP mutex and passes the system ceiling as a ceiling, i.e.

Pseudo-code 1 $Resource*$ GlobalMutexCreate($INT8U$ globalCeiling);

```
1: InitializeLocalResourceData();
2: return SRPMutexCreate(globalCeiling, 0);
```

To keep track of local subsystem ceilings, we need to maintain a separate *subsystem ceiling stack* for each subsystem. We use the `localResource` data-structure to implement a *subsystem ceiling stack*. The `globalResource` points to the corresponding resource block at the global level. `localCeiling` stores the local resource ceiling and `localLockingTask` stores the identifier of the task currently holding the resource. The `previous` pointer is used to maintain the stack structure, i.e. it points to the previous `localResource` structure on the stack. The `localCeiling` field of the `localResource` on top of the stack represents the current subsystem ceiling.

D. Scheduling

Extending the scheduler with a preemption rule is similar to the SRP implementation. When the scheduler selects the next server to be activated, its associated subsystem priority must exceed the current system ceiling. Similarly, the priority of the selected task must exceed the subsystem ceiling.

VII. SIRAP IMPLEMENTATION

This section presents the SIRAP implementation using the SRP infrastructure described in Section VI. First, we summarize SIRAP, followed by its realization within μC/OS-II.

A. SIRAP Recapitulated

SIRAP uses SRP to synchronize access to globally shared resources [2], and uses a skipping approach to prevent budget depletion inside a critical section. If a task 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 subsystem budget expires. If the remaining budget is not sufficient to lock and release a resource before expiration, (*i*) the task blocks itself, and (*ii*) the subsystem ceiling is raised to prevent other tasks in the subsystem to execute until the resource is released.

B. SIRAP Data and Interface Description

The SIRAP interfaces for locking and unlocking globally shared resources are defined as follows:

1) Lock SIRAP mutex:
```
void SIRAP_Lock(Resource* r, INT16U holdTime);
```
2) Unlock SIRAP mutex:
```
void SIRAP_Unlock(Resource* r);
```

The lock operation contains a parameter *holdTime*, which is accounted in terms of processor cycles and allocated to the calling task's budget. Efficiently filling in this parameter in terms of system load requires the programmer to correctly obtain the resource holding time [2, 23]. Since this provides an error-prone way of programming, we discuss an alternative approach in Section X.

C. SIRAP Primitive Implementation

SIRAP's locking and unlocking are building on the SRP implementation. Note that kernel primitives are assumed to execute non-preemptively, unless denoted differently (i.e. in SIRAP's lock operation).

1) Resource Locking: The lock operation first updates the subsystem's local ceiling according to SRP to prevent other tasks within the subsystem from interfering during the execution of the critical section. In order to successfully lock a resource there must be sufficient remaining budget within the server's current period. The remaining budget $Q_{remaining}$ is returned by a function that depends on the virtual timers mechanism, see Section IV-A. SIRAP's skipping approach requires the knowledge of the resource holding times ($holdTime$) [23] when accessing a resource. If $Q_{remaining}$ is not sufficient, 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 $Q_{remaining}$ is strictly larger than $holdTime$ before granting access to a resource. The lock operation in pseudo-code is shown in Source 2.

When the server's budget is replenished, all tasks spinlocking on a resource are unlocked as soon as the task is rescheduled. Although after budget replenishment a repeated test on the remaining budget is superfluous [2], we claim that spinlocking efficiently implements the skipping mechanism. A disadvantage of this implementation is that it relies on the assumption of a idling periodic server[3]. For any budget-preserving server, e.g. the deferrable server [25], the skipping mechanism by means of a spinlock is unacceptable, because a task consumes server budget during spinlocking.

Pseudo-code 2 void SIRAP_lock($Resource*$ r, $INT16U$ holdTime);

```
1: updateSubsystemCeiling();
2: while holdTime >= Q_remaining do
3:     enableInterrups;
4:     disableInterrups;
5: end while
6: SRPMutexLock(r, 0);
```

An alternative implementation would be to suspend a task when the budget is insufficient and resume a task when the budget is replenished. Firstly, this alternative approach induces additional overhead within the budget replenishment event due to the resumption of the blocked task. Secondly, μC/OS-II requires at any time a schedulable ready task, which is optionally a special idle task at the lowest priority. However, the system/subsystem ceilings prevent the idle task to be switched in. We could choose to make an exception for the idle task, but this breaks the property of SRP allowing to share stack space among tasks [5]. We consider further elaboration on these issues out of the scope of this paper.

2) Resource Unlocking: Unlocking a resource simply means that the system/subsystem ceiling must be updated and the SRP mutex must be released. Note that the latter command will also cause rescheduling.

Pseudo-code 3 void SIRAP_unlock($Resource*$ r);

```
1: updateSubsystemCeiling();
2: SRPMutexUnlock(r);
```

VIII. HSRP IMPLEMENTATION

This section presents the HSRP implementation. First, we summarize HSRP, followed by its realization within μC/OS-II.

A. HSRP Recapitulated

HSRP uses SRP to synchronize access to globally shared resources [1], and uses an overrun mechanism to prevent excessive blocking times due to budget depletion inside a critical section. When the budget of a subsystem expires and the subsystem has a task τ_i that is still locking a globally shared resource, this task τ_i continues its execution until it releases the locked resource. When a task accesses a global resource the local subsystem ceiling is raised to the highest local priority, i.e. for the duration of the critical section the task executes non-preemptively with respect to other tasks within the same subsystem. Two alternatives of the overrun mechanism are presented: (*i*) overrun with payback, and (*ii*) overrun without

[3] A polling server [24] also works under this assumption, but does not adhere to the periodic resource model [3], and therefore increases the complexity to integrate SIRAP and HSRP within a single HSF [7]. We leave the implementation for alternative server models as future work.

payback. The payback mechanism requires that when a overrun happens in a subsystem S_s, the budget of this subsystem is decreased with the consumed amount of overrun in its next execution instant. Without payback no further actions are taken after an overrun has occurred. We do not further investigate the relative strengths of both alternatives, since these heavily depend on the chosen system parameters [7]. In this section we show an implementation supporting both HSRP versions.

B. HSRP Data and Interface Description

The HSRP interfaces for locking and unlocking globally shared resources are defined as follows:

1) Lock HSRP mutex:
```
void HSRP_Lock(Resource* r);
```
2) Unlock HSRP mutex:
```
void HSRP_Unlock(Resource* r);
```

Contrary to SIRAP, the lock operation lacks the $holdTime$ parameter. Instead, HSRP uses a static amount of overrun budget, X_s, assigned to each server within the system.

C. HSRP Primitive Implementation

HSRP's locking and unlocking are building on the SRP implementation. Additionally, we need to adapt the budget depletion event handler to cope with overrun. This requires to keep track of the number of resources locked ($lockedResourceCounter$) within subsystem S_s. The server data-structure is extended with four additional fields for bookkeeping purposes, i.e. $lockedResourceCounter$, $inOverrun$, $consumedOverrun$ and $paybackEnabled$. The consumption of overrun budget ends when the normal budget is replenished [17], which requires an adaption of the budget replenishment event. Optionally, we implement an payback mechanism in the budget replenishment event. These event handlers are provided by RELTEQ as presented in Section IV-A.

1) Resource Locking: The lock operation first updates the subsystem's local ceiling to the highest local priority to prevent other tasks within the subsystem from interfering during the execution of the critical section. The lock operation in pseudo-code can be denoted as follows:

Pseudo-code 4 void HSRP_lock($Resource*$ r);

```
1: updateSubsystemCeiling();
2: S_s.lockedResourceCounter++;
3: SRPMutexLock(r, 0);
```

2) Resource Unlocking: Unlocking a resource means that the system/subsystem ceiling must be updated and the SRP mutex must be released. Additionally, in case that overrun budget, X_s, is consumed and no other global resource is locked within the same subsystem, we need to inform the scheduler that overrun has ended. Optionally, the amount of consumed overrun budget is stored to support payback upon the next replenishment. The unlock operation in pseudo-code is is shown in Pseudo-code 5.

The command $setSubsystemBudget(0)$ performs two actions: (*i*) the server is blocked to prevent the scheduler from rescheduling the server, and (*ii*) the budget depletion event is removed from RELTEQ's virtual event queue.

Pseudo-code 5 void HSRP_unlock($Resource*$ r);

```
1: updateSubsystemCeiling();
2: S_s.lockedResourceCounter--;
3: if S_s.lockedResourceCounter == 0 and S_s.inOverrun then
4:     if S_s.paybackEnabled then
5:         S_s.consumedOverrun = X_s − Q_remaining;
6:     end if
7:     setSubsystemBudget(0);
8: end if
9: SRPMutexUnlock(r);
```

3) Budget Depletion: We extend the event handler corresponding to a budget depletion by a conditional enlargement of the budget of the size X_s, with $X_s > 0$, i.e. in pseudo code:

Pseudo-code 6 on budget depletion:

```
1: if S_s.lockedResourceCounter > 0 then
2:     setSubsystemBudget(X_s);
3:     S_s.inOverrun = true;
4: end if
```

Note that $setSubsystemBudget(X_s)$ inserts a new event in RELTEQ's virtual event queue. Furthermore, we postpone server inactivation.

4) Budget Replenishment: When a server is still consuming overrun budget while its normal budget is replenished, the overrun state of this server is reset. Additionally, to support the optionally enabled payback mechanism, we replace the budget replenishment line in the corresponding event handler. The replenished budget is decreased with the consumed overrun budget in the previous period, i.e. in pseudo code:

Pseudo-code 7 on budget replenishment:

```
1: if S_s.inOverrun then
2:     if S_s.paybackEnabled then
3:         S_s.consumedOverrun = X_s − Q_remaining;
4:     end if
5:     S_s.inOverrun = false
6: end if
7: setSubsystemBudget(Q_s − S_s.consumedOverrun);
8: S_s.consumedOverrun = 0;
```

IX. Comparing SIRAP and HSRP

In this section we compare both implementations for HSRP and SIRAP. First, we present a brief overview of our test platform. Next, we compare the implementations of HSRP and SIRAP and demonstrate their effectiveness by means of an example system. Finally, we investigate the system overhead of the synchronization protocol's corresponding primitives.

A. Experimental Setup

In our experiments we use the cycle-accurate OpenRISC simulator provided by the OpenCores project [26]. Within this project an open-source hardware platform is developed. The hardware architecture comprises a scalar processor and basic peripherals to provide basic functionality [27]. The OpenRISC simulator allows simple code analysis and system performance evaluation. Recently, we created a port for μC/OS-II to the

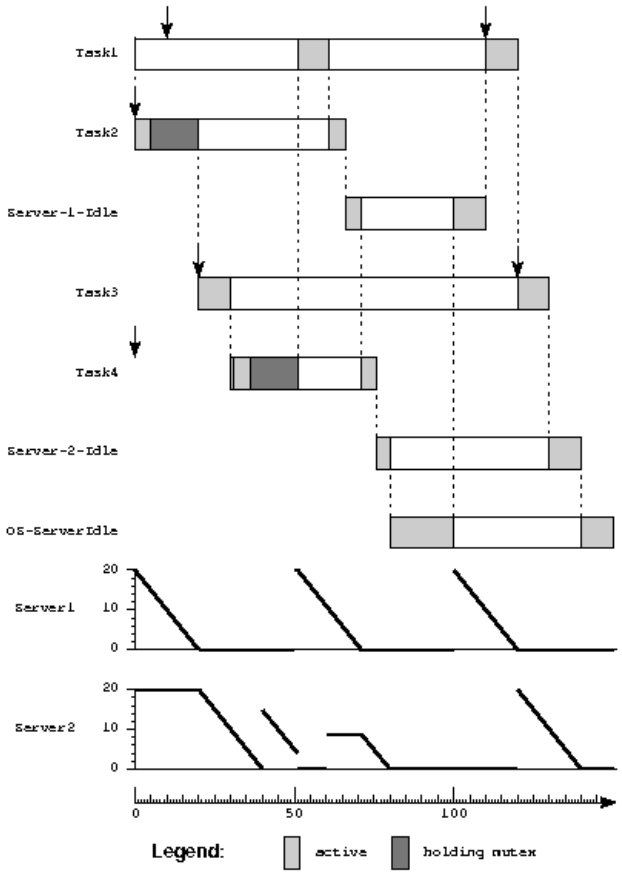

Fig. 4. Example trace for HSRP using the overrun and payback mechanisms.

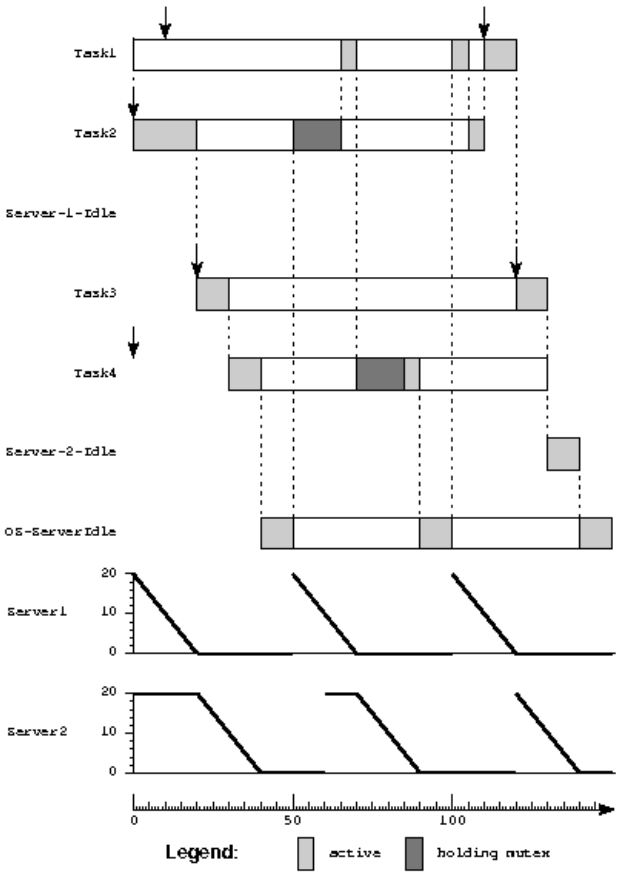

Fig. 5. Example trace for SIRAP using the skipping mechanism. Note that skipping occurs as normal task activation in the execution behaviour of a task.

OpenRISC platform, and extended the toolchain with visualization tools, which make it possible to plot a real-time system's behaviour [28, 29].

B. Protocol Comparison

To demonstrate the behavioural difference between HSRP and SIRAP, consider an example system comprised of two subsystems (see Table I) each with two tasks (see Table II) sharing a single global resource R_1. Note that the subsystem/task with the lowest number has the highest priority and that the computation times of tasks are denoted similarly as in Section III-B[2]. The local resource ceilings of R_1 are chosen to be equal to the highest local priority for SIRAP, while for HSRP this is the default setup.

TABLE I
EXAMPLE SYSTEM: SUBSYSTEM PARAMETERS

Subsystem	Period (P_s)	Budget (Q_s)	Max. blocking (X_s)
Server 1	50	20	15
Server 2	60	20	15

Inherent to the protocol, HSRP immediately grants access to a shared resource and allows the task to overrun its server's budget for the duration of the critical section, see Figure 4. Server 2 replenishes its budget with X_s at time 40. At time 50 task 4 releases R_1 and the remainder of X_s is discarded.

TABLE II
EXAMPLE SYSTEM: TASK PARAMETERS

Server	Task	Period	Computation time	Phasing
Server 1	Task 1	100	10	10
Server 1	Task 2	150	5+15+5	-
Server 2	Task 3	100	10	10
Server 2	Task 4	200	5+15+5	-

The normal budget of server 2 is reduced with its consumed overrun at the next replenishment (time 60).

Contrary, SIRAP postpones resource access when the budget is insufficient, as illustrated in Figure 5. SIRAP's spinlocking implementation is visualized as a longer normal execution time compared to HSRP, e.g. see the execution of task 2 in time interval $(10, 20]$. Figure 6 shows the behaviour of the example system when server 1 selects the SIRAP protocol and server 2 selects the HSRP (with payback).

C. Measurements and Results

In this section we investigate the overhead of the synchronization primitives of HSRP and SIRAP. Current analysis techniques do not account for overhead of the corresponding synchronization primitives, although these overheads become of relevance upon deployment of such a system. All our measurements are independent of the number of subsystems

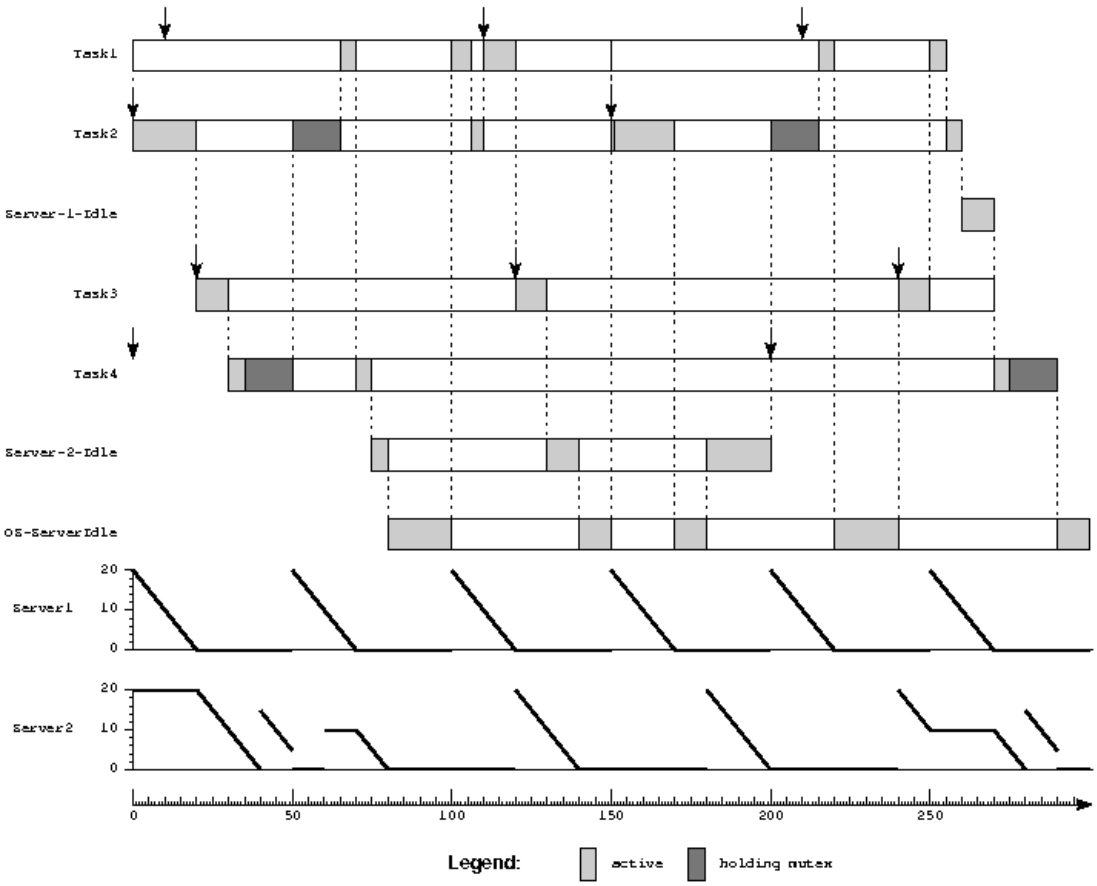

Fig. 6. Example trace combining SIRAP (server 1) and HSRP with payback (server 2) to access a single shared resource.

and tasks within a system.

As we can observe in our implementation, SIRAP induces overhead locally within a subsystem, i.e. the spinlock adds to the budget consumption of the particular task that locks the resource. HSRP introduces overhead that interferes at the global system level, i.e. the overrun mechanism requires the manipulation of event queues. The overheads introduced by the implementation of these protocols is summarized in Table III. A nice analogy of the implementation with respect to the schedulability analysis [7] is that HSRP has an overrun term at the global analysis level, while SIRAP accounts for self-blocking at the local analysis level.

TABLE III
OVERVIEW OF THE SYNCHRONIZATION PRIMITIVE'S IMPLEMENTATION COMPLEXITY FOR HSRP'S AND SIRAP'S RUN-TIME MECHANISMS.

Event	HSRP	SIRAP
Lock resource	-	spinlock
Unlock resource	overrun completion	-
Budget depletion	overrun	-
Budget replenishment	overrun completion, payback (optionally)	spinlock-completion

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 [7]. The number of CPU instructions executed for a single test is 10 instructions on our test platform.

The best-case HSRP overhead is null in addition to the normal number of CPU instructions that are spent to increase and decrease the subsystem and system ceilings. The worst-case HSRP overhead occurs at overrun. When the budget depletes, it is replenished with the maximum allowed overrun budget, which takes 383 instructions[4]. Overrun completion can occur due to two alternative scenarios: (*i*) a task unlocks a resource while consuming overrun budget, or (*ii*) the normal budget is replenished while the subsystem consumes overrun budget. The system overhead for both cases is 966 CPU instructions. When the payback mechanism is enabled, one additional computation is done to calculate the number that needs to be paid back at the next server replenishment, i.e. a system overhead of 5 instructions. As expected, we can conclude that especially ending overrun in HSRP's unlock operation is expensive.

[4]Our current setup only uses a dedicated virtual event queue for each server to keep track of a subsystem's budget, and the queue manipulations therefore have constant system overhead. In case multiple virtual events are stored in this queue, the system overhead for inserting and removing events becomes linear in its length [10].

D. Evaluation

The synchronization protocol implementations are composed of (*i*) variable assignments, (*ii*) (sub)system ceiling stack manipulations, (*iii*) RELTEQ operations [9, 10], (*iv*) a mechanism to allow non-preemptive execution (enable/disable interrupts) and (*v*) scheduler extensions. The first three building blocks are not specifically bound to μC/OS-II. The latter two are μC/OS-II specific. Especially, the extension of the scheduler by SRP's preemption rules is eased by μC/OS-II's open-source character.

X. Discussion

In the previous section we compared the system overhead of the HSRP and SIRAP primitives. Complementary, earlier results have shown that these protocols induce different system loads depending on the chosen (sub)system parameters [7]. To optimize the overall resource requirements of a system, we would like to enable both protocols side-by-side within the same HSF, as demonstrated in Figure 6. Enabling this integration puts demands on the implementation and the schedulability analysis. From the implementation perspective, an unification of the primitive interfaces is requires. However, the choice for a particular protocol at different levels in the system impacts the complexity of the analysis.

A. Uniform Analysis

The synchronization protocols guarantee a maximal blocking time with respect to other subsystems under the assumptions that (*i*) the analysis at the local and global level is correctly performed; (*ii*) the obtained parameters are filled in correctly; and (*iii*) the subsystem behaves according to its parameters. In order to allow SIRAP and HSRP to be integrated side-by-side at the level of subsystems within a single HSF, we need to unify the (global) schedulability analysis of both protocols. Initial results based on our implementation suggests that this integration step is fairly straightforward. The analysis of this integration is left as future work.

B. Uniform Interfaces

Assuming the system analysis supports integration of HSRP and SIRAP within the same HSF at the level of subsystems, one might choose a different synchronization protocol per subsystem depending on its characteristics. Enabling this integration requires that an application programmer (*i*) can *ignore* which synchronization protocol is selected by the system, and (*ii*) cannot *exploit* the knowledge of the selected protocol. The primitive interfaces therefore need to be unified.

The interface description of SIRAP differs from HSRP, because it requires to explicitly check the remaining budget before granting access to a resource, hence the occurrence of the $holdTime$ parameter in its lock interface (see Section VII). However, the integration of HSRP and SIRAP at the level of subsystems only requires that the maximum critical sections length, X_s, within subsystem S_s is known. Assuming X_s is available from the analysis, we can easily store this information within the server-structure. This relaxes the amount of run-time information and allows to remove the $holdTime$ parameter from SIRAP's lock operation, although at the cost of budget over-provisioning due to larger self-blocking times.

XI. Conclusion

This paper describes the implementation of two alternative SRP-based synchronization protocols within a two-level fixed priority scheduled HSF to support inter-application synchronization. In such systems, several subsystems execute on a shared processor where each subsystem is given a virtual share of the processor and is responsible for local scheduling of tasks within itself. We specifically demonstrated a feasible implementation of these synchronization protocols within μC/OS-II.

First, we presented an implementation of SRP within μC/OS-II that optimizes its existing synchronization primitives by a reduced amount of source code, a simplified implementation, and optimized run-time behaviour. Next, we presented the implementation of SIRAP using a run-time skipping mechanism, and HSRP using a run-time overrun mechanism (with or without payback). We discussed the system overhead of the accompanying synchronization primitives, and how these primitives can be integrated within a single HSF.

We aim at using these protocols side-by-side within the same HSF, so that their primitives can be selected based on the relative strengths of the protocol, which depend on system characteristics [7]. We showed that enabling the full integration of both synchronization protocols at the level of subsystems is relatively straightforward.

Our current research focuses on an appropriate selection criterium that minimizes the system overhead based on the subsystem parameters. In the future we would like to investigate less restrictive ways of combining synchronization protocols within HSFs in a predictable manner, e.g. per task, resource or resource access, by extending the existing analysis techniques and corresponding tooling. Finally, we would like to further investigate (*i*) trade-offs between different design and implementation alternatives of HSFs with appropriate synchronization protocols, and (*ii*) their applicability to a wider range of server models.

References

[1] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proc. RTSS*, Dec. 2006, pp. 257–270.

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

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

[4] Z. Deng and J.-S. Liu, "Scheduling real-time applications in an open environment," in *Proc. RTSS*, Dec. 1997, pp. 308–319.

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

[6] 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.

[7] M. Behnam, T. Nolte, M. Åsberg, and R. Bril, "Overrun and skipping in hierarchically scheduled real-time systems," in *Proc. RTCSA*, Aug. 2009, pp. 519–526.

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

[9] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Multiplexing real-time timed events," in *Proc. ETFA*, July 2009.

[10] M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Virtual timers in hierarchical real-time systems," *Proc. WiP session of the RTSS*, pp. 37–40, Dec. 2009.
[11] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource sharing in reservation-based systems," in *Proc. RTSS*, Dec. 2001, pp. 171–180.
[12] G. Buttazzo and P. Gai, "Efficient implementation of an EDF scheduler for small embedded systems," in *Proc. OSPERT*, July 2006.
[13] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proc. OSPERT*, July 2008, pp. 63–72.
[14] N. Fisher, M. Bertogna, and S. Baruah, "The design of an EDF-scheduled resource-sharing open environment," in *Proc. RTSS*, Dec. 2007, pp. 83–92.
[15] M. Behnam, T. Nolte, M. Åsberg, and I. Shin, "Synchronization protocols for hierarchical real-time scheduling frameworks," in *Proc. CRTS*, Nov. 2008, pp. 53–60.
[16] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Scheduling of semi-independent real-time components: Overrun methods and resource holding times." in *Proc. ETFA*, Sep. 2008, pp. 575–582.
[17] R. J. Bril, U. Keskin, M. Behnam, and T. Nolte, "Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited," in *Proc. CRTS*, 2009.
[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.
[19] Micrium, "RTOS and tools," March 2010. [Online]. Available: http://micrium.com/
[20] J. J. Labrosse, *Microc/OS-II*. R & D Books, 1998.
[21] J.-H. Lee and H.-N. Kim, "Implementing priority inheritance semaphore on uC/OS real-time kernel," in *Proc. WSTFES*, May 2003, pp. 83–86.
[22] M. Bergsma, M. Holenderski, R. J. Bril, and J. J. Lukkien, "Extending RTAI/Linux with fixed-priority scheduling with deferred preemption," in *Proc. OSPERT*, June 2009, pp. 5–14.
[23] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *Proc. WPDRTS*, March 2007, pp. 1–8.
[24] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Syst.*, vol. 1, no. 1, pp. 27–60, 1989.
[25] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.
[26] OpenCores. (2009) OpenRISC overview. [Online]. Available: http://www.opencores.org/project,or1k
[27] M. Bolado, H. Posadas, J. Castillo, P. Huerta, P. Sánchez, C. Sánchez, H. Fouren, and F. Blasco, "Platform based on open-source cores for industrial applications," in *Proc. DATE*, 2004, p. 21014.
[28] 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 *Proc. WATERS*, July 2010.
[29] "Simulating uC/OS-II inside the OpenRISC simulator," March 2010. [Online]. Available: http://www.win.tue.nl/~mholende/ucos/