Resource Sharing Using the Rollback Mechanism in Hierarchically Scheduled Real-Time Open Systems

Mikael Åsberg, Thomas Nolte and Moris Behnam
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
{mikael.asberg, thomas.nolte, moris.behnam}@mdh.se

***Abstract*—In this paper we present a new synchronization protocol called RRP (Rollback Resource Policy) which is compatible with hierarchically scheduled open systems and specialized for resources that can be aborted and rolled back. We conduct an extensive event-based simulation and compare RRP against all equivalent existing protocols in hierarchical fixed priority preemptive scheduling; SIRAP (Subsystem Integration and Resource Allocation Policy), OPEN-HSRPnP (open systems version of Hierarchical Stack Resource Policy no Payback) and OPEN-HSRPwP (open systems version of Hierarchical Stack Resource Policy with Payback). Our simulation study shows that RRP has better average-case response-times than the state-of-the-art protocol in open systems, i.e., SIRAP, and that it performs better than OPEN-HSRPnP/OPEN-HSRPwP in terms of schedulability of randomly generated systems. The simulations consider both resources that are compatible with rollback as well as resources incompatible with rollback (only abort), such that the resource-rollback overhead can be evaluated. We also measure CPU overhead costs (in VxWorks) related to the rollback mechanism of tasks and resources. We use the eXtremeDB (embedded real-time) database to measure the resource-rollback overhead.**

***Index Terms*—real-time systems, hierarchical scheduling, open systems, resource sharing, synchronization protocol**

I. Introduction

The hierarchical scheduling technique [1], [2], [3] has been introduced in order to simplify parallel development of real-time systems. It simplifies the integration of complex real-time systems by providing a mechanism for temporal isolation between subsystems. A typical system (an end-product, a piece of software etc.) can consist of a number of smaller parts, i.e., subsystems, and these may constitute a function/feature of the system itself. When finalizing the development of a system, these subsystems will be integrated. Systems that are built upon smaller systems (compositional systems) are common in industries such as automotive [4], aerospace [5] and consumer electronics [6].

The idea with open systems is that it should be possible to independently develop and validate (using schedulability analysis) subsystems, and then execute them concurrently on a shared execution platform without violating any validated properties (e.g., requirements, behavior etc.).

Although the hierarchical scheduling technique solves much of the integration related problems, it can not manage isolation between subsystems when they share resources other than the CPU. The remedy for this shortcoming is to introduce a synchronization protocol which can handle resource sharing between subsystems. The existing protocols for fixed-priority hierarchical scheduling in open systems are called OPEN-HSRP [7] and SIRAP [8]. A note to the reader is that we use the term OPEN-HSRP when we mean both OPEN-HSRPwP and OPEN-HSRPnP, which are the two versions of HSRP with and without payback respectively.

The analysis of SIRAP is tighter than that of OPEN-HSRP since it uses more detailed information [9], hence, it gives better results and resource utilization. However, some of this information is also required during runtime, which makes SIRAP more difficult to use in practice. In addition, the response times of tasks under SIRAP are in general longer than the corresponding response times under OPEN-HSRP (see Section VI).

In this paper we present a new synchronization protocol, called RRP, that can handle resource sharing in a hierarchically scheduled open system. It can use the tight analysis of SIRAP but does not need the detailed information during runtime, i.e., the implementation of the protocol is easier. Hence, RRP can improve the average-case response-time significantly compared to SIRAP, and it can perform better than OPEN-HSRP (Section VI). RRP is primarily focused on (and limited to) resources that can be aborted and, if necessary, rolled back. In this paper we *focus on* real-time databases [10], [11] which is a good example of a resource with rollback support. Real-time database management systems (RTDBMS) have been extensively studied in many industrial systems such as telecommunication [10], [12] and heavy vehicle control-systems [11]. RTDBMS can also be found in avionics applications, for example, the Boeing AH-64D Apache Longbow Helicopter[1]. We also consider resources that can be aborted without the requirement of rollbacks. Hence, in this paper, we deal with both types of resources; the simulations in Section VI-B and Section VI-C include resources that do not require rollbacks while we in Section VI-D add the overhead of resource rollbacks in the simulations by using measured values of database rollbacks which are derived from experiments. The task-rollback overhead is evaluated in Section V-A and the database-rollback overhead is presented in Section V-B. The

[1]McObject, press release: http://www.mcobject.com/January19/2004

contributions of this paper are:

1) The main contribution is the introduction of a new synchronization protocol RRP (Rollback Resource Policy). We elaborate on its theoretical strengths and weaknesses.
2) The second contribution is a comparison of two protocol mechanisms, i.e., rollback (RRP) and skipping (SIRAP), implemented in the operating system VxWorks 6.6.
3) The third contribution is the overhead measurements of an industrial RTDBMS (eXtremeDB) in VxWorks 6.6.
4) The fourth contribution is an extensive event-based simulation evaluation of RRP, SIRAP, OPEN-HSRPnP and OPEN-HSRPwP. To the best of our knowledge, this is the first detailed study of the performance differences of the implemented synchronization protocols in the context of hierarchical fixed-priority scheduling (FPS). Previous studies [7], [8], [9], [13], [14], [15], [16], [17], [18] are either limited to analysis or they show limited evaluation results. Our simulation shows real execution values since it simulates a real execution platform while previous work, which is based on analysis, only show calculated worst-case values. We emphasize that we do **not** perform schedulability-analysis simulations (which has been done in previous work), but instead we execute the implementation of the protocols.

A. Organization of the paper

The outline of this paper is as follows: in Section II we outline the preliminaries on hierarchical scheduling, the system model, local resource sharing and global resource sharing. We present the related work in the area of hierarchical scheduling and resource sharing in Section III and in Section IV we introduce the protocol and elaborate on its theoretical properties. Section V presents overhead measurements of the mechanisms of RRP and SIRAP as well as measurements of database rollback-overhead. Section VI shows the results of the event-based simulations. Finally, in Section VII, we conclude our results.

II. Preliminaries

A. Hierarchical scheduling

The lower part of Figure 1 illustrates the scheduling mechanism in two-level hierarchical scheduling. The *Global scheduler* (in this case running FPS) is responsible for distributing the *CPU* resource to the servers (the schedulable entity of a subsystem). The servers have a defined time (*budget*) reserved at every time interval of length *period* [19] and the execution order is based on the servers *priority*. The servers are scheduled with respect to the scheduling policy of the global scheduler and its parameters (budget, period and priority), hence, they can be interpreted as ”virtual tasks”. A server can consist of a *Local scheduler* which schedules the content inside (its tasks) during the time when the server is selected to consume its budget (which is decided by the global scheduler). The tasks are described by their release period [20], execution time and priority.

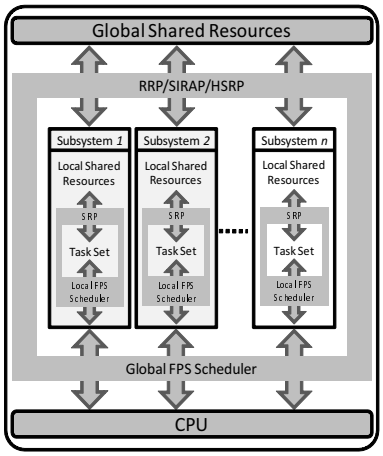

Fig. 1. HSF with shared resources

B. System model

A system consists of several subsystems that are scheduled periodically by a global scheduler on a single CPU. In this paper we assume that the global scheduler implements fixed-priority preemptive scheduling, i.e., FPPS. It is further assumed that tasks from different subsystems share mutually exclusive resources with each other, so there is a need for a synchronization protocol to resolve potential conflicts. The protocol should only be dependent on information from the subsystem under analysis and it may not alter parameters of other subsystems within the system. The resources are assumed to have the property that they can be locked and later aborted (and if necessary rolled back) while keeping a consistent state. The longest possible time between locking and unlocking a resource (not including task preemptions) is defined as the *critical section execution time* (CSET).

C. Resource sharing

Resource sharing in hierarchical scheduling can be divided in two parts: when resources are shared by tasks within the same subsystem (local resource sharing) and when tasks residing in different subsystems share the same resources (global resource sharing). Local resource sharing can be handled by using classical resource sharing protocols such as the Priority Inheritance Protocol (PIP) [21], the Priority Ceiling Protocol (PCP) [21] or the Stack Resource Policy (SRP) [22], within each subsystem. When it comes to globally shared resources (and we also assume that tasks in the same subsystem may share the same global resources) then there is a need to protect the resource access at the local level (i.e., using SRP or similar) as well as at the level of subsystems. In this sense, a task locking a global resource will cause its subsystem (server) to also lock a resource, hence, the same rules apply for servers when it comes to resource access (following the rules of the global resource protocol). The global resource access protocol may be similar as in the local level, e.g., SRP may also be used at the level of subsystems.

One additional challenge when sharing global resources is

what action to be taken in the case when a task is currently holding a global resource and its corresponding server budget is depleted. We will elaborate on this in Section II-C2.

1) Local resource access: When a task attempts to access a local resource then it performs a lock action, similarly, it unlocks the resource when it has finished its access.

Note that most of the global synchronization protocols (OPEN-HSRP, SIRAP, RRP, BROE etc.) are based on SRP (both locally and globally). Hence, we devote the rest of this subsection to describe the SRP protocol. SRP blocks a task if there is a potential risk that it might access a resource that is already in use by another task. The same mechanism is used for bounding the *priority inversion*. Priority inversion happens when higher priority tasks are blocked by lower priority tasks. Each resource has a defined *resource ceiling* which is the highest priority value of all tasks that will access the resource (assuming FPS). The mechanism checks at the time when releasing a task that it has higher priority than the resource ceiling of the currently (and most recently) locked resource (if any). A task is only allowed to be released if it has higher priority than the highest resource ceiling among all currently locked resources – this ceiling value is defined as the *system ceiling*. When a task locks a resource, the system ceiling is automatically updated with the resource ceiling of the locked resource. When the task unlocks, the system ceiling is reset to its previous value that it had before the resource was locked, which can be either another resource ceiling or empty. In this way, the system ceiling can be easily tracked using a stack.

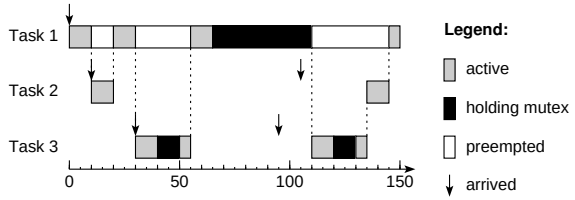

Fig. 2. SRP example

Figure 2 illustrates how SRP enables mutual exclusive access to resources (time 95) and prevents priority inversion (time 105). In this figure, the priority ordering is $Prio(Task3) > Prio(Task2) > Prio(Task1)$, and $Task1$ and $Task3$ share the same local resource.

2) Global resource access: When a task locks a global resource (which can be locked by tasks in the same or other subsystems) then there is a need for a protocol that can preserve mutual exclusive access to resources that are shared with tasks from the same or other subsystems. In this case, the protocol needs to control the execution (time reservations) of servers in the same way as resource access protocols controls the execution of tasks. Hence, each subsystem will have a local protocol controlling local resource accesses (as well as global accesses). In addition, a global resource access protocol will block servers in the case of global resources. Finally, there is a separate mechanism that prevents server budget-depletion in the case when a global resource is locked by a task. This is needed to prevent excessive access times of global resources. Figure 3 illustrates the problem with long blocking times (illustrated as waiting time) when resources are shared globally.

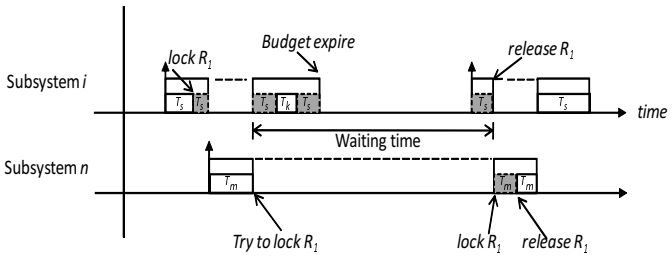

Fig. 3. Global resource locking problem

a) SIRAP: There exists several protocols that have all of these mentioned functionalities, and one of them is the SIRAP protocol. SIRAP uses the SRP protocol for sharing both local and global resources, hence, there is a global SRP that controls server execution and one local SRP for each subsystem which controls tasks. When a task wants to access a global resource, the SIRAP protocol checks if there is sufficient budget left for the task to execute its critical section and release it before budget depletion. The maximum time that a task can lock a global resource (including preemptions of other tasks in the same subsystem) is called *Resource Holding Time* [14] (RHT). If there is sufficient budget then the task locks the resource and continues. If not, the task raises the system ceiling level (within the subsystem) according to the priorities of the (subsystem) tasks accessing the global resource, and continues to execute (busy waiting) without actually locking the global resource. This is referred to as *self-blocking*. So locally, the task behaves like it is accessing a global resource. However, the global resource is actually free and the CPU budget will be idled away unless another task is released with higher priority than the system ceiling of the subsystem. The task will wait for the next subsystem period and lock the global resource at that time. Figure 4 illustrates the self-blocking scenario. The server priority ordering is $Prio(Server3) > Prio(Server2) > Prio(Server1)$ (left side), and their corresponding tasks (right side) are mapped by names, i.e., task $S3Task1$ belongs to $Server3$ etc. $Server1$ and $Server3$ share the same global resource. $Server3$ is released at time 30, and its task is released 10 time-units later. The task attempts to lock a global resource at time 45 (requiring 10 time-units for its RHT), but there is only 5 units remaining of the budget so the task self-blocks between time 45 and 50. Its server is released again at time 60 and due to the global system ceiling it is granted the CPU at time 85. At this time the task has sufficient budget to complete its RHT. Note that one task blocks (60) and another one preempts (65) when $S1Task1$ is holding the global resource (55-85). This is dependent on the system ceiling of the subsystem. HSRP [13] does not allow this type of preemption so it raises the system

ceiling such that it is equal to the highest priority task. In SIRAP and RRP (as well as in the newer version of HSRP, i.e., OPEN-HSRP [7]) however, this is optional. The final observation is that $Server2$ is blocked at time 75 due to that the global system ceiling is equal to the priority of $Server3$ (this prevents priority inversion at the level of subsystems).

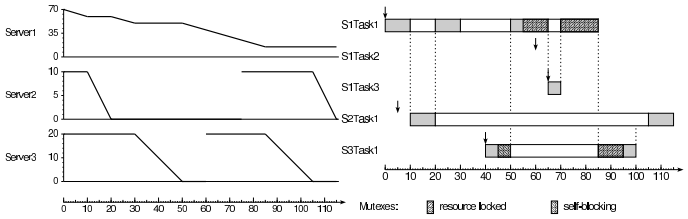

Fig. 4. SIRAP example

b) OPEN-HSRP: OPEN-HSRP [7] (as well as the original HSRP [13]) uses SRP at both levels. The main difference from SIRAP is the mechanism that is used in case of a budget depletion during the CSET of a task. Instead of checking and preventing a task from locking a resource, OPEN-HSRP will prolong the budget (overrun) of a server until the task unlocks the resource. The difference between OPEN-HSRPnP and OPEN-HSRPwP is that OPEN-HSRPwP will decrease the next upcoming budget with the same amount of time that was spent in the overrun section.

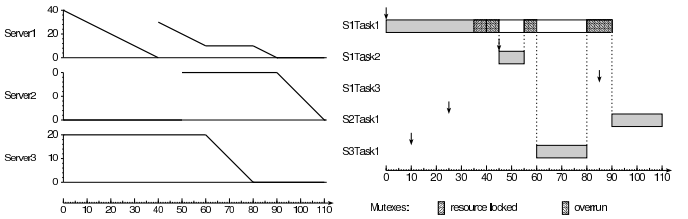

Fig. 5. OPEN-HSRP example

This overrun section is animated in Figure 5. $S1Task1$ locks a resource at time 35 and the budget depletes at time 40. $Server1$ has an overrun from time 40-60 and 80-90 (not including the suspended time due to server preemption between time 60 and 80). We can also see local interference between time 45 and 55 which can happen if the preempting task has higher priority than the local system ceiling (both SRP levels will behave exactly the same as in SIRAP). If the example in Figure 5 was illustrated with OPEN-HSRPwP, then the budget of $Server1$ would decrease with 30 time units at its next budget replenishment (the local task interference at time 45-55 is included in this payback).

III. Related work

Hierarchical scheduling theory A lot of work on hierarchical scheduling and its schedulability analysis [23], [24], [25], [26], [27], [28] has been presented previously, which has originated from open systems [2] in the late 1990's. [25] proposed analysis which support open systems, i.e., it does not require knowledge of other subsystems in order to analyze a particular subsystem. SIRAP [8] and OPEN-HSRP [7] extended the analysis to support resource sharing.

Hierarchical scheduling with resource sharing Starting with the HSRP [13] protocol in 2006, which was later extended in 2010: OPEN-HSRP [7], the problem of resource sharing in a hierarchically scheduled environment was solved by overrunning the allocated CPU time for subsystems, when a subsystem was unable to finish its resource access before its budget depleted. We call this mechanism overrun [29], [30]. However, since HSRP is not suitable for open systems, SIRAP [8] was introduced in 2007, as well as BROE [14] which was developed for dynamic-priority scheduling.

Recent published work include a preliminary comparison of SIRAP, HSRP and BROE [15], as well as implementation comparisons [16] in the OS μC/OS-II. In [31], the authors present a similar mechanism as RRP named HSTP (Hierarchical Synchronization with Temporal Protection). The difference from RRP is that HSTP continues to execute with the help of donations while RRP instead executes rollbacks. Another difference is that the authors focus on the analysis while we devote our work to simulations. Other studies [9] have also compared SIRAP and HSRP. [17] shows the differences in the amount of interactions that SIRAP and HSRP have with semaphores and the scheduler in VxWorks. Another implementation of SIRAP and HSRP in μC/OS-II [18] presents overhead measurements.

Rollback This is a well studied technique, within the research of fault tolerance, and can be traced back to the early 1970's [32]. The idea is to save the state of a program (known as checkpointing) such that it can later be restarted (rollback) from the most recent checkpoint in case of program error. There has also been more recent studies on the impact of schedulability when rollback is used [33]. Rollback can also be used to resolve concurrent access at the time of unlocking a shared resource instead of blocking a task when a resource is locked. This is presented in [34] but it is limited to 1-level scheduling. Two protocols called rollback (RP) and skip (SP) [35] has been developed for Pfair scheduling on multi-core platforms. In Pfair, tasks execute in so called *quanta* time slots. In order to avoid tasks from having locked resources at the end of a quanta, a *freeze* signal is issued which disables tasks from locking a resource. This initiates a *frozen* interval which does not allow any resources to be locked by any tasks (thus it resembles the skipping technique used in SIRAP). Each lock has its own FIFO queue which is used to grant tasks the resource in first-in-first-out (FIFO) fashion in case the resource is already locked. The main difference between the RP and SP protocol is how they handle these FIFO queues during the frozen interval. The SP protocol leaves pending requests in the queue and ignores them up until the requesting tasks are scheduled again. RP however discards any pending request, forcing the requesting tasks to re-issue the lock later when they are scheduled again. Hence, the requesting tasks will be inserted into the FIFO queue later in their next quanta.

Compared to RRP and SIRAP, both RP and SP resemble the SIRAP protocol since all of them implement some form of skip mechanism. RRP does not have any type of check at the end of a time slot (unlike RP, SP and SIRAP), it simply grants the lock no matter if the time slot expires during the tasks CSET. If concurrency is detected then RRP will simply rollback (the resource) and restart the task execution. Just to clarify, RRP does not resemble RP since RP resembles SIRAP due to that the frozen interval prohibit tasks from locking resources (SP has the same resemblance to SIRAP).

IV. Rollback resource policy (RRP)

Comparing SIRAP and HSRP, looking from a global perspective of the CPU resource, SIRAP punishes the subsystem itself by not using CPU cycles of other subsystems, and wasting the subsystems own CPU cycles, in benefit of having independent subsystems. HSRP does the opposite, it punishes other subsystems, i.e., delaying them while overrunning. Hence, HSRP is utilizing the slack time in the system (if any) and giving it to the overrunning subsystem. RRP represents the next step in this evolvement: to make more efficient use of the CPU cycles, i.e., more than SIRAP and less than HSRP. This is done without the risk of temporarily overloading the CPU (which is one of the drawbacks with HSRP) while keeping the property of allowing independently developed subsystems. We do so by identifying resources that can be aborted during its access (and rolled back to a consistent state if necessary), and taking advantage of this property in order to save CPU cycles. The complete description of RRP is presented in the following section.

A. Protocol description

The RRP protocol is based on a 2-level SRP mechanism, similar to SIRAP [8] and OPEN-HSRP [7]. The main difference with RRP is how it handles the situation when the subsystem budget depletes during a tasks CSET, this is illustrated in Figure 6. A task is always granted access to a global resource, no matter if the CSET length is less than the remaining subsystem budget or not. If the CSET is larger than the remaining budget then the system ceiling will be decreased at the time of budget depletion (while the task is still accessing the global resource). If no other subsystem (i.e., task in a subsystem) wants to access the same global resource up until the next subsystem instance, then, once the subsystem is scheduled again, the system ceiling is raised and the task can simply continue to execute its critical section. In the case when another subsystem wants to access the same global resource, while the previous task is still accessing this resource, then it performs a rollback on the resource and accesses it. This will cause the previous task (T_s in the figure) to be rolled back to a state, just before it locked the resource. The task will roll back to the checkpoint at its next subsystem instance and restart its critical section execution.

We will compare RRP with SIRAP in this section since their mechanisms resemble eachother most. The limitation of RRP is that it can only handle resources that can be either

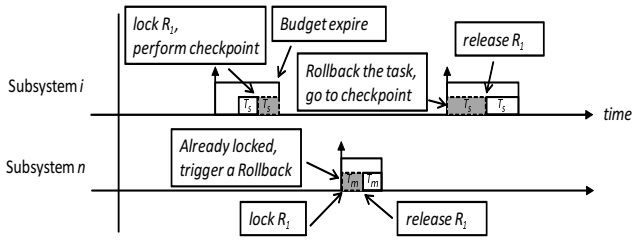

Fig. 6. The rollback mechanism

aborted, or aborted and rolled back (during its use). However, for this subset of resources, RRP shows good performance as we will see in Section VI.

In the worst case, a rollback can happen at each subsystem period which is equivalent to selfblocking in SIRAP. Hence, the analysis in SIRAP [8] is also applicable for RRP. However, RRP has the potential to improve the average case performance as shown in Figure 7. RRP is more robust since it always conducts a rollback if there is a concurrent access while SIRAP does not have a similar safety mechanism. SIRAP must rely on that the worst-case RHT is a safe upper estimate, hence, it typically has a safety margin added to every RHT. This means that there will be a higher risk that SIRAP will perform selfblockings even though the tasks could have entered their critical sections and finished before budget depletion. RRP will perform better in these cases since it always allows a task to access a global resource irrespective of how much remaining budget that is left. Hence, RRP will waste less CPU cycles.

In theory, and in practice as our event-based simulations will show, RRP has good performance even in the case when tasks in the same subsystem are allowed to preempt other tasks during their global resource accesses. SIRAP tends to degrade its performance excessively when global resource accesses are preemptive (locally). The checking of RHT against the remaining budget (when a task wants to access a global resource) is at greater risk of failing since the RHT must include the WCET of tasks that might preempt.

Looking from the perspective of the user of the SIRAP protocol, the RHTs are both technically hard to estimate and it puts an extra load on the user, making it more complicated to use (implementation wise, not the analysis). RRP is easier to use since it does not require this information during runtime, i.e., this makes the implementation easier and more safe.

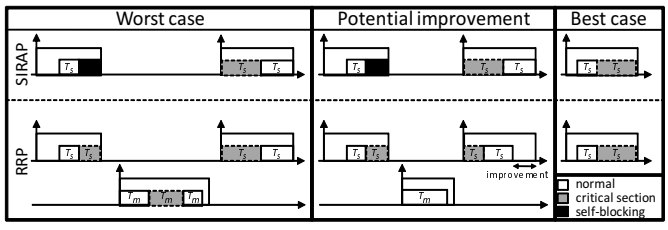

Fig. 7. RRP and SIRAP have the same worst and best case behavior, however, RRP can potentially improve the performance in the average case

Note that it is unlikely that systems have resources that are

all compatible with abortion and rollback. In these cases, a mixture of RRP and SIRAP/OPEN-HSRP could be used, i.e., use SIRAP/OPEN-HSRP for resources that cannot be aborted. This could improve the average-case performance.

V. Rollback overhead

In this section we present the overhead measurements of task rollback (Section V-A) and database rollback (Section V-B). The experiments were conducted on the real-time OS VxWorks 6.6 running on an Intel P4 1.6 GHz platform.

A. Task-rollback overhead

Central to comparing protocols is the overhead inherent in their implementations. Assessing this we ran experiments to estimate the overhead of the task-rollback and skipping operations. In order to get checkpoint and task-rollback functionality we used the primitives taskRegsGet/taskRegsSet to store and load the register (program counter) of a task. The taskRegsSet operation will restart a task job from the beginning of its task body. In order to create a checkpoint just prior to the lock operation, we use the setjmp function. Having setjmp prior to the lock and longjmp at the start of the task enables the restart of a task to jump directly to the lock function. SIRAP requires the tickGet function in order to estimate the remaining budget. Table I shows the best- and worst-case execution time of the mentioned primitives as well as context-switch time and the total overhead of a rollback. The rollback includes the context switch to the start of a task (body) and jump (using longjmp) to the code section prior to the lock primitive (the cost of the primitive taskRegsSet is not included). The execution time of setjmp is very small so we consider this overhead to be negligible.

Operation	Description	Exec. time µs
taskRegsSet	Load program counter (start of task body)	≤ 1
taskRegsGet	Save program counter (start of task body)	2-3
Rollback	Context switch to saved program counter	2-3
tickGet	Read system tick	1-2
Context switch	Regular context switch	2-3
setjmp	Save exec. context (from within a task)	-
longjmp	Load exec. context (from within a task)	-

TABLE I
Operation overheads in VxWorks

taskRegsGet is the most costly operation in a task rollback. However, it is sufficient if it is executed once prior to the first job of a task and saved. Executing taskRegsGet prior to the first job of each task saves the current registers, i.e., the start of the task, and it is not required to execute this primitive again during the life-span of a task since we assume that the program-counter value of the beginning of a task will not change. A task rollback (not including the overhead of the operation taskRegsSet) has a similar cost as a regular task context switch and in this overhead we also include the cost of longjmp.

Figure 8 shows the sources of overhead in the two protocols. taskRegsGet is required once prior to the first job so we do not include it. setjmp is negligible as mentioned previously and a rollback (which includes a task context-switch and a longjmp call) has equal overhead as a context switch so none of these two values affect our comparison. The difference between the two protocols is tickGet for SIRAP, and taskRegsSet for RRP. Looking at only skip/rollback overhead, then the difference between the protocols (in case that both protocols fail to lock/unlock) is 1 µs (taskRegsSet) for RRP and 1-2 µs (tickGet) for SIRAP. In case the protocols succeed to lock/unlock before the budget deplete then RRP has 0 µs overhead and SIRAP 1-2 µs (tickGet). Hence, in either case, RRP does not cost more than SIRAP in terms of CPU. However, rollback may of course be more costly in terms of memory management compared to SIRAP.

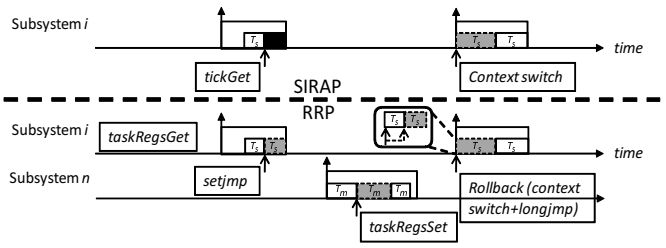

Fig. 8. Overhead sources for SIRAP and RRP

These experiments focus only on the rollback mechanism and overhead for tasks. There might of course also be overhead when rolling back a resource, i.e., re-balancing a data structure such as a red-black tree for example. We compensate for this by emulating resource-rollback overhead in our simulations for RRP (Section VI-D).

In this paper we skip the overhead comparison of RRP and OPEN-HSRP since previous work [17] has already studied the overhead differences between overrun and skipping (OPEN-HSRP and SIRAP). This study concluded that SIRAP had less overhead than OPEN-HSRP in terms of expensive calls to the global scheduler.

B. Resource-rollback overhead

The resource-rollback overhead presented in this section relates to the context of heavy vehicle control-systems. The targeted resource (which we use in our experiments) is an industrial RTDBMS called eXtremeDB[2] version 4.1 for VxWorks 6.6/6.7 PENTIUMdiab. According to a case study at Volvo Construction Equipment [11], hard real-time tasks in their systems access one, or at most, a few data elements. Hence, we assume that database transactions have a maximum of 20 data elements. Figure 9 shows the measured overhead of 20 different database transactions operating on $1-20$ data elements. We can observe that the overhead of Rollback and Commit operations are low compared to the actual manipulation of database items (Insert). A rollback of a transaction with 1 to 20 data elements take $20-40$ µs. An overestimation of this

[2]eXtremeDB: http://www.mcobject.com/

parameter will be used in Section VI-D when the performance of RRP (including this overhead) is evaluated.

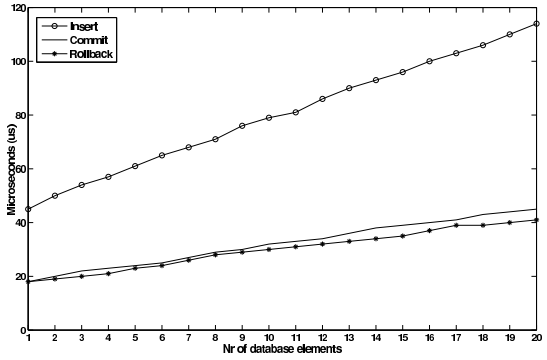

Fig. 9. Database-operation overhead for transaction sizes of $1-20$ elements

VI. Simulation results

The simulations were conducted using a timed-automata (with tasks) model of a two-level FPPS scheduling framework [36] of periodic servers [19] and tasks [20]. The model was implemented using the TIMES tool and we extended this model with an implementation of RRP, SIRAP, OPEN-HSRPnP and OPEN-HSRPwP. The model (schedulers and protocols) was verified using model-checking. We carefully selected example systems that generated all possible protocol scenarios in order to verify certain properties, similar techniques were used to verify the two-level scheduler [36]. These systems were used to verify that the model was free from deadlock/livelock, that a resource was never locked by more than one user at a time, and that two consecutive unlocks were never performed on a resource. Having our simulator model-checked and verified makes our simulation results solid, at the cost of longer simulation time since automatically generated code of timed automata in general has slow performance. Thus, this massive simulation study was calculated to take about 2672 days (a bit more than 7 years) on a single core platform. However, thanks to the Ericsson Research Laboratory (at Mälardalen University) we were privileged to use their experimental 128 core platform (AMD Opteron) utilizing 64 of the cores for our simulations. This decreased the simulation time substantially. In order to perform an efficient simulation, we generated C-code from our extended model, and executed the simulation framework without the use of the modeling tool.

A. Simulation settings

We ran in total 256500 simulation runs for approximately 15 minutes each (the time limit is required since the simulations can take several years to complete otherwise). The parameters we use are randomly selected, although their characteristics are inspired by a vehicle (wheel-loader) application from Volvo Construction Equipment [37]. We define a system as a set of servers where each server has a set of tasks. Each task will access either one global resource or it will not access any resource at all, i.e., there is no nested resource accesses. The number of global resources per system is randomly selected between $1/3$ of the number of tasks and the number of tasks in a subsystem (the number of tasks per subsystem is equal among all subsystems in a system). Server periods are randomly selected in the range from 500 to 2000 time units and task periods are randomly selected in the range 5000 to 15000 time units. Half of the systems were generated with server budgets calculated using the analysis from SIRAP [8] and the other half had budgets which were calculated using OPEN-HSRPwP [7]. The main difference between these two budget calculations is that SIRAP requires a larger budget than OPEN-HSRP in order for the RHT to fit in the local analysis. The reason for running half of the systems with each approach is because we want to be fair to all protocols since the OPEN-HSRPwP budget calculation is optimized for the OPEN-HSRP protocols and the SIRAP budget calculation is optimized for SIRAP and RRP.

Half of systems have a CSET of 3-9% (3, 6 and 9%) of the task execution time and the other half has 5-25% (5, 10, 15, 20 and 25%). All systems have a range of different settings wrt the number of servers, number of tasks, taskset utilization and resource ceilings. We have chosen to have systems with 2, 4, 6, 8 or 10 servers, 3, 5 or 7 tasks per subsystem, 30, 50 or 70% total task-set utilization and highest priority setting for resource ceilings on half of the systems, while the other half has ceilings calculated using the SRP protocol. For each combination of the number of servers and tasks, taskset utilization, CSET and ceiling we generated 25 different systems which differ in period, budget, execution-time, priority and the number of resources. In total, all systems are equally divided with 6 different settings (which we just described), i.e., the systems have all possible combinations of settings. We do this with the intention that none of the protocols (RRP, SIRAP or OPEN-HSRP) should benefit from a particular setting.

The RHT values do not include task interference, hence, they only represent the CSET (i.e., RHT = CSET). Half of our simulation systems (128250) have SRP calculated ceilings which allow task preemptions during the CSET (thus increasing the actual runtime value of the RHT). We do not compensate for these high RHT values in the calculated budgets. Hence, we intentionally want to stress the protocols to see how sensitive they are to larger RHT values at runtime which are not present in the analysis.

The simulations presented in Section VI-B and VI-C assume that resources do not require rollbacks [10]. The simulation results presented in Section VI-D assumes that there is an overhead of 1 time unit every time a resource is rolled back, i.e., RRP will pay a penalty every time a rollback occurs.

B. Schedulability

Table II shows the percentage of systems (out of the 36000 systems) that each protocol managed to schedule without any task or server deadline-violation. As can be seen, SIRAP suffers a lot from the SRP-based ceilings as we mentioned previously (this is even more clear in the right figure of Figure 10).

Protocol	Schedulable systems (%)
RRP	67.4
OPEN-HSRPwP	66.9
OPEN-HSRPnP	65.9
SIRAP	32.8

TABLE II
SCHEDULABLE SYSTEMS BY EACH PROTOCOL OUT OF 36000 SYSTEMS.

RRP has the largest amount of schedulable systems of all protocols. This is not surprising since SIRAP and OPEN-HSRP both have an Achilles' heel, while RRP can avoid them both. SIRAP tends to reduce the CPU resource to a subsystem when a resource cannot be locked (due to a too small remaining budget), hence, this can cause a task deadline violation when the CPU time is decreased. OPEN-HSRP on the other hand has a different problem. It tends to overload the CPU when a resource access exceeds the budget. During overruns, there is not always enough CPU time for both the overrunning subsystem and the subsystem that was originally scheduled to run. If both subsystems want more than 100% CPU at this point in time then this will cause a server deadline violation. In the best case, RRP does not reduce the CPU share of a subsystem (like SIRAP), nor does it ever overload the CPU (like OPEN-HSRP). Hence, RRP utilizes the CPU better in most cases compared to SIRAP and OPEN-HSRP and that is why it has the highest rate of schedulable systems.

Protocol	rollback/selfblock/overrun
RRP	5840456
SIRAP	6078920
OPEN-HSRPnP	17068697
OPEN-HSRPwP	17528265

TABLE III
NR OF MECHANISM OPERATIONS, I.E., ROLLBACKS, SELFBLOCKINGS OR OVERRUNS IN THE SCHEDULABLE SYSTEMS.

Table III shows the number of times that each protocol had to activate its protection mechanism that avoids budget depletion before a resource gets unlocked. The mechanism activations are only counted from the systems that were schedulable. Take into consideration that SIRAP only has half as many schedulable systems as the others, hence, its number should at least be the double. RRP has the lowest number of mechanism activations (rollback in the case of RRP) which is natural since the number of rollbacks are dependent on the risk of resource conflicts in between server activations rather than the number of resource accesses that exceeds the budget (which is the case for the other protocols). The low number of rollbacks is the reason why RRP has a high success rate of schedulable systems (Table II).

Figure 10 shows the number of local and global deadline violations per protocol wrt the ceiling settings. RRP and SIRAP have more local scheduling errors than OPEN-HSRP while OPEN-HSRP has more global scheduling errors than RRP and SIRAP. The reason for this is because RRP and SIRAP decreases CPU time for subsystem tasks when resources cannot be locked, while OPEN-HSRP delays other servers in

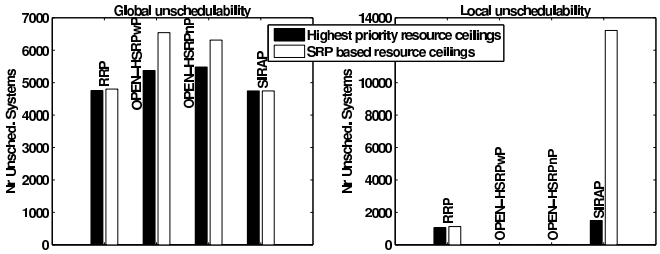

Fig. 10. Nr of unschedulable systems wrt the ceiling settings.

this case. SIRAP has a high number of local scheduling errors since it selfblocks a lot due to the SRP-based ceilings. What is also interesting to note (but which is difficult to observe in Figure 10) is that SIRAP has about 1% less global scheduling errors than RRP in total. The reason is because rollbacks are in general a little more costly in terms of server level blocking since tasks lock global resources prior to the rollback while SIRAP avoids it. Looking at the ceiling setting, SIRAP is most sensitive to the increased RHT (which is caused by the SRP-based ceiling setting) but we can also note that OPEN-HSRP increases the number of scheduling errors substantially with the SRP-based ceiling setting. RRP is the most robust protocol when RHT is larger at runtime than what was assumed in the analysis.

C. Response time

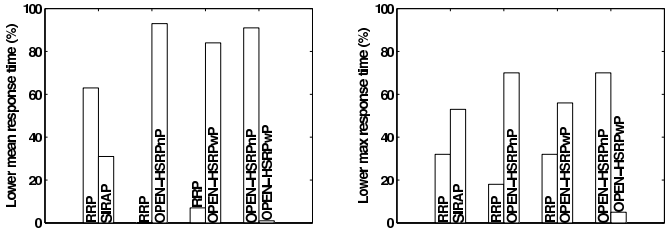

Fig. 11. Pair-wise comparison of response times: percentage of commonly schedulable systems with lower response-time.

Figure 11 shows comparisons of measured task response-times (RT). We have compared the protocols two-by-two by identifying the subset of systems that they both managed to schedule successfully. We compare them two-by-two instead of all four since we get more common schedulable systems. We check how many of these common schedulable systems that each protocol had the lowest mean or max RT in. For each system we calculate the mean measured RT for each task and normalize them with the task period, then we take the mean value of all tasks mean values. Similarly, we take the maximum RT of each task and normalize it to the period, then we calculate one mean value for all of these values. On the left hand side of Figure 11 we show the mean RTs and on the right side are the max RT values. We see that RRP has more systems with lower mean RT than SIRAP, while SIRAP has more systems with lower max RTs. Also, RRP has a higher rate of systems with lower max RT when compared

to OPEN-HSRPnP and OPEN-HSRPwP (as opposed to mean RTs). OPEN-HSRPwP clearly increases RT due to its payback mechanism, while OPEN-HSRPnP gains a lot on taking global slack-time (and not paying it back) which we can see on the mean and max RT values. RRP has low average RT values, but the occasional rollbacks generates a few but very long RTs for some tasks. The same reasoning can be applied when comparing RRP with OPEN-HSRPnP and OPEN-HSRPwP. The reason why SIRAP is better than RRP when it comes to max RTs is because RRP keeps a global resource lock (in the cases where SIRAP selfblocks instead) and this causes blocking of other subsystems. This could potentially cause a higher rate of max RTs. Note that SIRAP has better max RTs than RRP in less than 32.8% of all the systems (since SIRAP only managed to schedule 32.8%). RRP has 34.6% more schedulable systems compared to SIRAP which means that RRP had in total much better max and mean RTs.

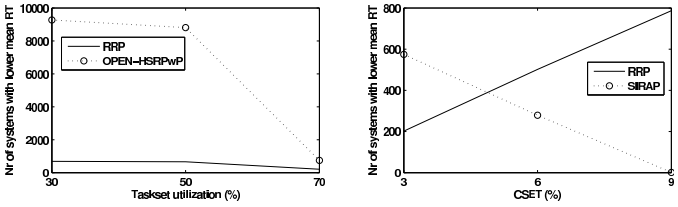

Fig. 12. LEFT: Mean response time difference between RRP and OPEN-HSRPwP wrt the taskset utilization. RIGHT: Mean response time difference between RRP and SIRAP wrt CSET.

Left sub-figure in Figure 12 shows that OPEN-HSRPwP has an almost equal amount of systems with better mean RT when taskset utilization is high, when compared to RRP. The reason for this is because the overrun time increases (more task interference) which causes larger paybacks which in turn affects the budgets a lot and thereby increases task RTs. It is also interesting to note that SIRAP has a larger amount of systems with lower mean RTs when CSET is low (3%), this can be seen in Figure 12 (right sub-figure). This is natural since RRP and SIRAP should have similar performance when CSET is low and RRP should be better when it is high. When CSET is high, more scenarios happen where CSET exceeds budget, thus causing more selfblocks but a smaller amount of rollbacks.

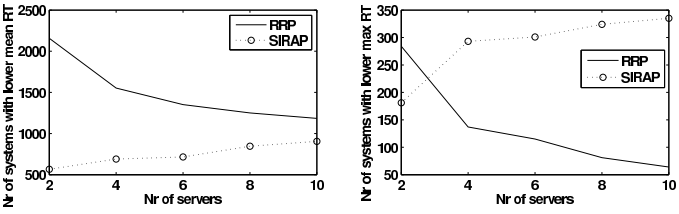

Fig. 13. LEFT: Mean response time difference between RRP and SIRAP wrt the number of servers. RIGHT: Max response time difference between RRP and SIRAP wrt the number of servers.

Figure 13 shows that an increasing number of servers cause more conflicts in between server activations for RRP which increases the amount of rollbacks. What can also be noted is that RRP has higher max RT values than SIRAP. This relates to RRPs global resource locking prior to a rollback, which does not happen in the case of SIRAP during selfblocking.

D. Resource rollback overhead

We conducted 22500 simulation runs (for each protocol) with a CSET length of 10, 20, 30, 40 and 50% of the corresponding server budget (the rest of the parameters were the same as presented in section VI-A). In this setting we also simulated RRP with a penalty overhead of 1 time unit for each rollback, denoted RRPwRO (RRP with resource overhead). This overhead corresponds to the extra time it takes for a resource to rollback to a consistent state. We have measured this overhead and the result is at most 40 µs for a RTDBMS in a heavy vehicle application context (Section V-B). Hence, a scheduling tick of 1 should be sufficient to cover this overhead if we assume 40 µs between ticks. This is immoderate since 100 µs is the minimum time between ticks that we could achieve in our platform (VxWorks/P4) without saturating the system with interrupts. Table IV shows that RRP (with/without resource overhead) manages to schedule most systems successfully. The overhead penalty (RRPwRO) decreases the number of schedulable systems marginally (only 0.2%). This result is of course dependent on the number of rollbacks etc.

Protocol	Schedulable systems (%)
RRP	71.2
RRPwRO	71.0
OPEN-HSRPwP	64.0
OPEN-HSRPnP	62.9
SIRAP	32.0

TABLE IV
SCHEDULABLE SYSTEMS BY EACH PROTOCOL OUT OF 22500 SYSTEMS.

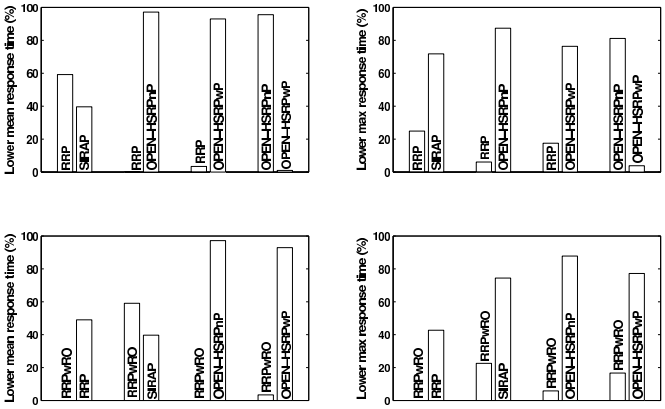

Fig. 14. Pair-wise comparison of response times: percentage of commonly schedulable systems with lower response-time.

Figure 14 shows the RT differences between the protocols. The two upper sub-figures show similar results as in Figure 11. The two lower sub-figures show the results with RRPwRO. The overhead penalty (RRPwRO) does not increase the mean or max RT (i.e., they are the same), compared to RRP without overhead, in approximately half of all the systems.

For example, the lower left sub-figure shows that RRP has lower mean RT in 49% of all of the systems that both RRP and RRPwRO managed to schedule successfully. Hence, since RRPwRO has 0% systems with lower mean RTs, the rest of the systems (51%) had the same mean RT values. The lower right sub-figure shows that RRP has lower max RT in 42.7% of the systems. The reason for this lower percentage value (compared to 49%) is because the resource overhead does not always affect the maximum RT. For example, it might increase the second highest measured RT, thereby not affecting the maximum measured value.

VII. Conclusion

In this paper we have presented a new synchronization protocol called Rollback Resource Policy (RRP). Our event-based simulations compared RRP to SIRAP [8], OPEN-HSRPnP [7] and OPEN-HSRPwP [7]. These results show that RRP improves the average case response times of tasks compared to SIRAP. Also, RRP managed to schedule more systems than OPEN-HSRPnP and OPEN-HSRPwP and it handles highly stressed systems best compared to the other protocols. Our results indicate that neither task- nor resource-rollback overhead have a significant effect on the performance of RRP. It has deadline misses in 0.2% more systems when emulating overhead for resource rollbacks. The good performance is not the only positive aspect of RRP. It supports the tight analysis of SIRAP and it uses less analysis information at runtime making it easier and more safe to use than SIRAP.

Future work includes investigating the combination of RRP with SIRAP and/or OPEN-HSRP. This is interesting since RRP can improve the performance of SIRAP and OPEN-HSRP by using it for resources that can be rolled back. We will also look into the implementation of RRP in VxWorks.

Acknowledgments

The authors wish to express their gratitude to the lab manager Daniel Flemström at the Ericsson Research Laboratory for letting us use their equipment. It would not have been possible to conduct a simulation study of this magnitude without their hardware.

References

[1] P. Goyal, X. Guo, and H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *OSDI'96*.

[2] Z. Deng and J. W.-S. Liu, "Scheduling Real-time Applications in an Open Environment," in *RTSS'97*.

[3] J. Regehr and J. A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *RTSS'01*.

[4] T. Scharnhorst, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.-L. Mate, and K. Nishikawa, "AUTOSAR - Challenges and Achievements," in *12^{th} International VDI Congress Electronic Systems for Vehicles*, 2005.

[5] ARINC/RTCA-SC-182/EUROCAE-WG-48, "Minimal operational performance standard for avionics computer resources." RTCA, Incorporated, 1828 L Street, NW, Suite 805, Washington D.C. 20036, 1999.

[6] D. Andrews, I. Bate, T. Nolte, C. M. O. Pérez, and S. M. Petters, "Impact of Embedded Systems Evolution on RTOS Use and Design," in *OSPERT'05*.

[7] M. Behnam, T. Nolte, M. Sjödin, and I. Shin, "Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems," *IEEE Transactions on Industrial Informatics*, vol. 6, pp. 93–104, 2010.

[8] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems," in *EMSOFT'07*.

[9] M. Heuvel, M. Behnam, R. J. Bril, J. J. Lukkien, and T. Nolte, "Opaque Analysis for Resource Sharing in Compositional Real-Time Systems," in *CRTS'11*.

[10] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring, "DeeDS Towards a Distributed And Active Real-Time Database System," *SIGMOD Record*, vol. 25, pp. 38–51, 1996.

[11] D. Nyström, "Data Management in Vehicle Control-Systems," Ph.D. dissertation, Mälardalen University, 2005. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1040

[12] J. Taina and K. Raatikainen, "RODAIN: A Real-Time Object-Oriented Database System for Telecommunications," in *CIKM'96*.

[13] R. I. Davis and A. Burns, "Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems," in *RTSS'06*.

[14] N. Fisher, M. Bertogna, and S. Baruah, "The Design of an EDF-Scheduled Resource-Sharing Open Environment," in *RTSS'07*.

[15] M. Behnam, T. Nolte, M. Åsberg, and I. Shin, "Synchronization Protocols for Hierarchical Real-Time Scheduling Frameworks," in *CRTS'08*.

[16] M. Heuvel, R. J. Bril, and J. J. Lukkien, "Protocol-Transparent Resource Sharing in Hierarchically Scheduled Real-time Systems," in *ETFA'10*.

[17] M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril, "Implementation of Overrun and Skipping in VxWorks," in *OSPERT'10*.

[18] M. Heuvel, J. J. Lukkien, R. J. Bril, and M. Behnam, "Extending a HSF-enabled Open-Source Real-Time Operating System with Resource Sharing," in *OSPERT'10*.

[19] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for some Practical Problems in Prioritized Preemptive Scheduling," in *RTSS'86*.

[20] C. Liu and J. Layland, "Scheduling Algorithms for Multi-programming in a Hard Real-Time Environment," *ACM*, vol. 20, pp. 46–61, 1973.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Comput.*, vol. 39, pp. 1175–1185, 1990.

[22] T. P. Baker, "Stack-Based Scheduling of Real-time Processes," *Real-Time Systems*, vol. 3, pp. 67–99, 1991.

[23] R. I. Davis and A. Burns, "Hierarchical Fixed Priority Pre-emptive Scheduling," in *RTSS'05*.

[24] T.-W. Kuo and C.-H. Li, "A Fixed-Priority-Driven Open Environment for Real-Time Applications," in *RTSS'99*.

[25] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *RTSS'03*.

[26] X. Feng and A. Mok, "A Model of Hierarchical Real-Time Virtual Resources," in *RTSS'02*.

[27] G. Lipari and S. K. Baruah, "Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems," in *RTAS'00*.

[28] G. Lipari and E. Bini, "Resource Partitioning Among Real-Time Applications," in *ECRTS'03*.

[29] T. M. Ghazalie and T. P. Baker, "Aperiodic Servers in a Deadline Scheduling Environment," *Real-Time Systems*, vol. 9, pp. 31–67, 1995.

[30] L. Abeni and G. Buttazzo, "Resource Reservation in Dynamic Real-Time Systems," *Real-Time Systems*, vol. 27, pp. 123–167, 2004.

[31] M. Heuvel, R. J. Bril, and J. J. Lukkien, "Dependable Resource Sharing for Compositional Real-Time Systems," in *RTCSA'11*, 2011.

[32] K. M. Chandy and C. V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," *IEEE Transactions on Computers*, vol. C-21, pp. 546 –556, 1972.

[33] S. Punnekkat, A. Burns, and R. Davis, "Analysis of Checkpointing for Real-Time Systems," *Real-Time Systems*, vol. 20, pp. 83–102, 2001.

[34] T. Johnson, "Interruptible Critical Sections for Real-Time Systems," Department of Computer and Information Science, University of Florida, Technical Report, 1993.

[35] P. Holman and J. H. Anderson, "Locking in Pfair-Scheduled Multiprocessor Systems," in *RTSS'02*.

[36] M. Åsberg, P. Pettersson, and T. Nolte, "Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling," in *ECRTS'11*.

[37] T. Nolte, I. Shin, M. Behnam, and M. Sjödin, "A Synchronization Protocol for Temporal Isolation of Software Components in Vehicular Systems," *IEEE Transactions on Industrial Informatics*, vol. 5, pp. 375–387, 2009.