Implementation of Overrun and Skipping in VxWorks

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

Reinder J. Bril
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5612 AZ Eindhoven
The Netherlands
r.j.bril@tue.nl

***Abstract*—In this paper we present our work towards allowing for dependence among partitions in the context of hierarchical scheduling of software systems with real-time requirements, and we present two techniques for cross-partition synchronization. We have earlier developed a Hierarchical Scheduling Framework (HSF) in VxWorks for independent real-time tasks, and in this paper we extend the HSF implementation with capabilities of synchronization between tasks resident in two different partitions. In particular, we have implemented the overrun and skipping mechanisms in our modular scheduling framework. Our framework has a key characteristic of being implemented on top of the operating system, i.e., no modifications are made to the kernel. Such a requirement enforce some restrictions on what can be made with respect to the implementation. The evaluation performed indicates that, under the restrictions of not modifying the kernel, the skipping mechanism has a much lower implementation overhead compared to the overrun mechanism**[1]**.**

I. Introduction

Advanced operating system mechanisms such as hierarchical scheduling frameworks provide temporal and spatial isolation through virtual platforms, thereby providing mechanisms simplifying development of complex embedded software systems. Such a system can now be divided into several modules, here denoted subsystems, each performing a specific well defined function. Development and verification of subsystems can ideally be performed independently (and concurrently) and their seamless and effortless integration results in a correctly functioning final product, both from a functional as well as extra-functional point of view.

In recent years, support for temporal partitioning has been developed for several operating systems. However, existing implementations typically assume independence among software applications executing in different partitions. We have developed such a modular scheduling framework for VxWorks without modifying any of its kernel source code. Our scheduling framework is implemented as a layer on top of the kernel. Up until now, this scheduling framework required that tasks executing in one subsystem must be independent of tasks executing in other subsystems, i.e., no task-level synchronization was allowed across subsystems. In this paper we present our work on implementing synchronization protocols for our hierarchical scheduling framework, allowing for task-level synchronization across subsystems. We implemented the synchronization protocols in VxWorks, however, they can naturally be extended to other operating systems as well. We are considering, in this paper, a two level hierarchical scheduling framework (as shown in Figure 1) where both the local and global schedulers schedule subsystems/tasks according to the fixed priority preemptive scheduling (FPS) policy.

The contributions of this paper are the descriptions of how the skipping and overrun mechanisms are implemented in the context of hierarchical scheduling without modifying the kernel. The gain in not altering the kernel is that it does not require any re-compilation, there is no need to maintain/apply kernel modifications when the kernel is updated/replaced and kernel stability is maintained. We have evaluated the two approaches and results indicate that, given the restriction of not being allowed to modify the kernel, the overhead of the skipping mechanism is much lower than the overhead of the overrun mechanism.

The outline of this paper is as follows: Section II gives an overview of preliminaries simplifying the understanding of this paper. Section III presents details concerning the implementation of the skipping and overrun mechanisms. Section IV presents an evaluation of the two methods, Section V presents related work, and finally Section VI concludes the paper together with outlining some future work.

II. Preliminaries

This section presents some preliminaries simplifying the presentation of the rest of the paper. Here we give an overview of our hierarchical scheduling framework (HSF) followed by details concerning the stack resource policy (SRP) protocol and the overrun and skipping mechanisms for synchronization, in the context of hierarchical scheduling.

A. HSF

The Hierarchical Scheduling Framework (HSF) enables hierarchical scheduling of tasks with real-time constraints. In [1] we assume that tasks are periodic and independent, and we use periodic servers to implement subsystems. The HSF is implemented as a two layered scheduling framework

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

as illustrated in Figure 1, where the schedulers support FPS and EDF scheduling.

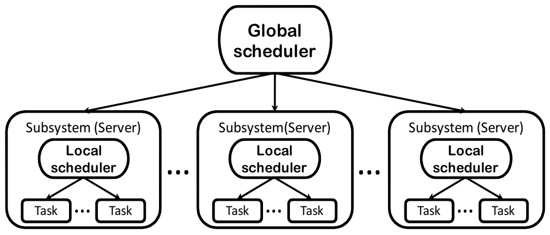

Fig. 1. HSF structure

Both schedulers (local and global) are activated periodically according to task/server parameters and a one-shot timer is used to trigger the schedulers. The next triggering (absolute) time of the tasks/servers are stored in a *Time Event Queue* (TEQ). The TEQ is essentially a priority queue, storing the release times (in absolute values) of tasks/servers. The input to the one-shot timer is a value derived by subtracting the shortest time in the TEQs from the current absolute time (since the timer input should be in relative time). Three TEQs can be active at once, the TEQ holding server release times, the current active servers TEQ for task release times and a TEQ (with one node) holding the current active servers budget expiration time. The current absolute time is updated only at a scheduler invocation, i.e., when the one-shot timer is set, we also set the absolute time equal to the next triggering time. When the next event arrives, the current absolute time will match the $real$ time. It is important to note that if we would like to invoke our scheduler $before$ the event arrives, then the current absolute time will not be correct. This fact needs to be taken into account when implementing synchronization protocols in our framework. The triggering of the global and local schedulers are illustrated in Figure 2. The $Handler$ is responsible for deriving the next triggering event (could be task or server related). Depending on which kind of event, i.e., task activation, server activation or budget expiration, the $Handler$ will either call the $Global\ scheduler$ or the $Local\ scheduler$. The $Global\ scheduler$ will call $Local\ scheduler$ in case of server activation (there might be task activations that have not been handled when the server was inactive). The VxWorks scheduler is responsible for switching tasks in the case when a task has finished its execution. The VxWorks scheduler will be invoked after an interrupt handler has executed (i.e., after $Handler$ has finished), but only if there has been any change to the ready queue that will affect the task scheduling.

All servers, that are ready, are added to a server ready queue and the global scheduler always selects the highest priority server to execute (depends also on the chosen global scheduling algorithm). When a server is selected, all tasks that are ready, and that belong to that subsystem, are added to the VxWorks task ready queue and the highest priority ready task is selected to execute.

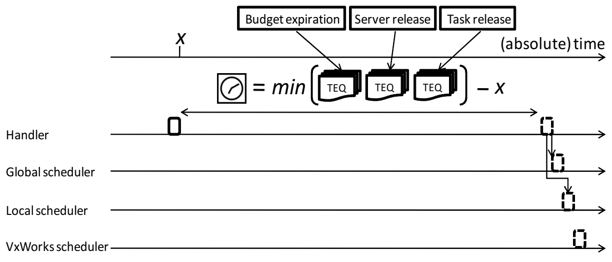

Fig. 2. Scheduler triggering

B. Shared resources in HSF

The presented HSF allows for sharing of logical resources between arbitrary tasks, located in arbitrary subsystems, in a mutually exclusive manner. To access a resource, a task must first lock the resource, and when the task no longer needs the resource, it is unlocked. The time during which a task holds a lock is called a critical section. For each logical resource, at any time, only a single task may hold its lock. A resource that is used by tasks in more than one subsystem is denoted a *global shared resource*. A resource only used within a single subsystem is denoted a *local shared resource*. In this paper, both local and global shared resources are managed by the SRP protocol. This protocol has the strength that it can be used with different scheduling algorithms such as FPS and EDF scheduling, which are supported by HSF at both global and local scheduling level.

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

- **Resource ceiling:** Each global shared resource is associated with two types of resource ceilings; an *internal* resource ceiling for local scheduling and an *external* resource ceiling for global scheduling. They are defined as the priority of the highest priority task/subsystem that access this resource.
- **System/subsystem ceiling:** The system/subsystem ceilings are dynamic parameters that change during execution. The system/subsystem ceiling is equal to the highest external/internal resource ceiling of a currently locked resource in the system/subsystem.

Under SRP, a task τ_k can preempt the currently executing task τ_i (even inside a critical section) within the same subsystem, only if the priority of τ_k is greater than its corresponding subsystem ceiling. The same reasoning can be made for subsystems from a global scheduling point of view. The problem that SRP solves (synchronization of access to shared resources without deadlock) can arise at two completely different levels, due to that subsystems share resources and because tasks (within a subsystem) share resources. That is why SRP is needed at both local and global level, and also the reason why a global resource has a local and global ceiling.

2) Mechanisms to handle budget expiry while executing within a critical section: To bound the waiting time of tasks from different subsystems that want to access the same shared resource, subsystem budget expiration should be prevented while locking a global shared resource. The following two mechanisms can be used to solve this problem:

- **The overrun mechanism:** The problem of subsystem budget expiry inside a critical section is handled by adding extra resources to the budget of each subsystem to prevent the budget expiration inside a critical section. Hierarchical Stack Resource Policy (HSRP) [2] is based on an overrun mechanism. HSRP stops task preemption within the subsystem whenever a task is accessing a global shared resource. SRP is used at the global level to synchronize the execution of subsystems that have tasks accessing global shared resources. Two versions of overrun mechanisms have been presented; 1) *with payback*; whenever overrun happens in a subsystem S_s, the budget of the subsystem will, in its next execution instant, be decreased by the amount of the overrun time. 2) *without payback*; no further actions will be taken after the event of an overrun.
- **The skipping mechanism:** Skipping is another mechanism that prevent a task from locking a shared resource by skipping (postpone the locking of the resource) its execution if its subsystem does not have enough remaining budget at the time when the task tries to lock the resource. Subsystem Integration and Resource Allocation Policy (SIRAP) [3] is based on the skipping mechanism. SIRAP uses the SRP protocol to synchronize the access to global shared resources in both local and global scheduling. SIRAP checks the remaining budget before granting the access to the globally shared resources; if there is sufficient remaining budget then the task enters the critical section, and if there is insufficient remaining budget, the local scheduler delays the critical section entering of the job until the next subsystem budget replenishment (assuming that the subsystem budget in the next subsystem budget replenishment is enough to access the global shared resource by the task). The delay is done by blocking that task that want to access the resource (self blocking) during the current server period and setting the local ceiling equal to the value of internal resource ceiling of the resource that that task wanted to access.

Scheduling analysis of both of these two mechanisms can be found in [2] respectively [3].

III. Implementation

This section compares and discusses some issues related to the implementation of the skipping and overrun mechanisms. These implementations are based on our previous implementation of the Hierarchical Scheduling Framework (HSF) [1] in the VxWorks operating system. To support synchronization between tasks (or subsystems) when accessing global shared resources, advances in the implementation of VxWorks made since [1] does not include the implementation of the SRP protocol, and SRP is used by both both skipping and overrun mechanisms. Therefore, our implementation of the SRP protocol is outlined below.

A. Local synchronization mechanism

Since both skipping and overrun depend on the synchronization protocol SRP, which is not implemented in VxWorks, we have implemented this protocol ourselves. The implementation of SRP is part of our previous VxWorks implementation (HSF), hence, this SRP implementation is adjusted to fit with hierarchical scheduling. We added two queues to the server TCB, see Figure 3. Whenever a task wants to access a locally shared resource (within a subsystem), it calls a corresponding `SrpLock` function (Figure 4). When the resource access is finished, it must call `SrpUnlock` (Figure 5).

```
struct SERVER_TCB {
        // Resource queue, sorted by ceiling
        queue SRP_RESOURCES;
        // Blocked tasks, sorted by priority/preempt. level
        queue SRP_TASK_BLOCKED_QUEUE;
        /* The rest of the server TCB */
```

Fig. 3. Data-structures used by SRP

```
1: void SrpLock (int local_res_id) {
2:      InterruptDisable( );
3:      LocalResourceStackInsert(local_res_id); // Ceiling is updated
4:      InterruptEnable( );
5: }
```

Fig. 4. Lock function for SRP

```
1: void SrpUnlock (int local_res_id) {
2:      InterruptDisable( );
3:      LocalResourceStackRemove(local_res_id); // Ceiling is updated
4:      if (LocalCeilingHasChanged( ))
5:         MoveTasksFromBlockedToReady(RunningServer);
6:      NewTask = GetHighestPrioReadyTask( );
7:      if (RunningTask.ID ≠ NewTask.ID)
8:         RunningServer.LocalScheduler( );
9:      InterruptEnable( );
10: }
```

Fig. 5. Unlock function for SRP

Lines (3, 5, 8) in Figure 5 are specific to each server, since they have their own task ready-, blocked- and resource-queue (stack), and a local scheduler. The same goes for line (3) in Figure 4. Note that `SrpUnlock` is executed at task-level (user-mode). Hence, we start the local scheduler by generating an interrupt that is connected to it. When our local scheduler (which is part of an interrupt handler) has finished, the VxWorks scheduler will be triggered if a context switch should occur. This is illustrated in Figure 6, where we use the VxWorks system call `sysBusIntGen` to generate an interrupt which will trigger the corresponding connected handler, which in this case is our local scheduler.

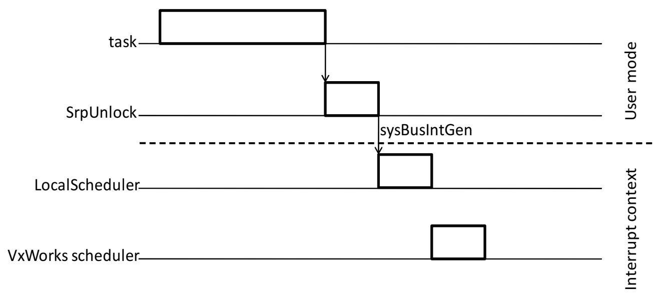

Fig. 6. Local scheduler invocation

The only modification made in our local scheduler is that it compares the local system ceiling against the task priority, before releasing a task (putting it in the task ready queue).

B. Global synchronization mechanisms

It is important to note that from the user perspective, there is no difference when locking a local or global resource, since all global resources are mapped to one corresponding local resource. When calling a `lock` function that implements a global synchronization protocol (i.e., overrun or skipping), the only information needed is the local resource ID. From this, we can derive the global resource ID. Hence, the global synchronization protocol calls the local synchronization protocol (i.e., SRP in this case), and, it also implements the global synchronization strategy, i.e., skipping or overrun in this case. Both of them need to use a local synchronization protocol, other than that, skipping is the only protocol of the two that need direct access to the local system, i.e., the local scheduler. The reason for this is covered in section III-E.

To support the synchronization mechanisms, additional queues are required in the system level (resource queue and blocked queue) to save all global resources that are in use, and to save the blocked servers. Similar queues are required for each subsystem (covered in section III-A) to save the local resources that are in use within the subsystem, and to save the blocked tasks. The resource queues are sorted, by the resource ceilings, hence, the first node represents the system ceiling (there is one (local) system ceiling per server and one (global) system ceiling). The resource queues are mapped as outlined in Figure 7.

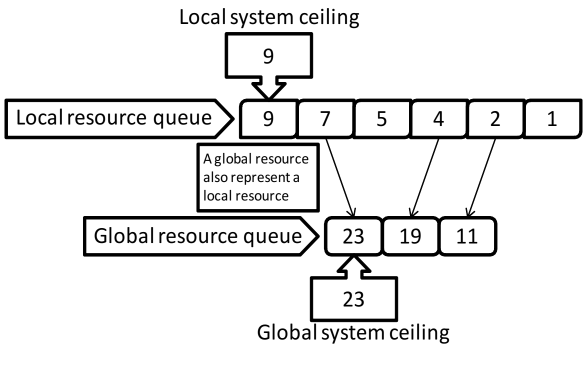

Fig. 7. Resource queue mapping

When a task wants to access a global shared resource, it uses the `lock` function, and when the task wants to release the resource, it calls the function `unlock`. The implementation of `lock` and `unlock` depends on the type of synchronization approach (overrun or skipping). In general, `lock` and `unlock` change some parameters that are used by the scheduler, e.g., system/subsystem ceiling, server/task ready queue, and server/task blocked queue. When a server/task is activated, the local/global schedulers checks weather the server/task has a higher priority than the current system/subsystem ceiling. If yes, then the server/task is added to the ready queue, otherwise the server/task will be added to the blocked queue. When the `unlock` function is called, all tasks and servers that were blocked, by the currently released shared resource, should be moved from the server/task blocked queue to the ready queue, and then the scheduler should be called to reschedule the subsystems and tasks. For this reason, it is very important that the `lock`/`unlock` functions should have mutual exclusion with respect to the scheduler, to protect the shared data-structures. In this implementation, interrupt disable in the `lock`/`unlock` function has been used to protect shared data-structures, noting that the interrupt disable time should be very short.

Since the scheduler can be triggered by the `unlock` functions (unlike the implementation in [1]), the current absolute time for this event should be calculated by subtracting a current timestamp value with the timestamp from the latest scheduler invocation and adding this value to the latest evaluated absolute time. The difference in time between the *real* current absolute time and the calculated one is the drift caused by both the skipping and overrun mechanisms. More of this is discussed in the next section.

C. Time drift

Budgets and time-triggered periodic tasks are implemented using a one-shot timer [1], which may give rise to relative jitter [4] due to inaccuracies caused by time calculations, setting the timer, and activities that temporarily disable interrupts. Relative jitter (or drift) may give rise to severe problems whenever the behavior of the system needs to remain synchronized with its environment. In the implementation used in this paper, such explicit synchronization requirements is not assumed, however. Implementation induced relative jitter can therefore be accommodated in the analysis as long as the jitter can be bound. By assuming a maximum relative jitter for every time the timer is set, and a maximum number of times the timer is set for a given interval, the relative jitter can be bound for periods of both budgets and time-triggered tasks and for capacities of budgets. Now the worst-case analysis can be adapted by making worst-case assumptions, i.e., by using (a) maximal inter-arrival times for periods and minimal capacities for budgets and (b) minimal-inter-arrival times (and worst-case computation times) of tasks. For the two types of synchronization protocols discussed in this paper, i.e., overrun with (or without) payback and skipping, the impact of relative jitter is similar.

D. Overrun mechanism implementation

Besides the data-structures needed for keeping track of global system ceiling, line (1) in Figure 8, and the queue of blocked servers, line (2) in Figure 8, overrun also need data-structures to keep track of when an overrun has occurred, line (5,7) in Figure 8.

```
1: queue GLOBAL_RESOURCES; // Used by Overrun
2: queue SERVER_BLOCKED_QUEUE; // Used by Overrun
3: struct SERVER_TCB {
4:         // Nr of global resources that are locked
5:         char nr_global_resources_locked;
6:         // Flag for keeping track if an overrun has occurred
7:         char overrun;
8:         /* The rest of the server TCB */
```

Fig. 8. Data-structures used by Overrun

Figure 9 shows the `OverrunLock` function for the overrun mechanism. The resource that is accessed is inserted in both the global and local resource queue which are sorted by the node's resource ceilings.

```
1: void OverrunLock (int local_res_id) {
2:         SrpLock(local_res_id);
3:         InterruptDisable( );
4:         GlobalResourceStackInsert(local_res_id); // Ceiling is updated
5:         RunningServer.nr_global_resources_locked++;
6:         InterruptEnable( );
7: }
```

Fig. 9. Lock function for Overrun

In line (5) in Figure 9, the function increment `RunningServer.nr_global_resources_locked` by one, which indicate the number of shared resources that are in use. This is important for the scheduler so it does not terminate the server execution at the budget expiration. When the budget of a server expires, the scheduler checks this value. If it is greater than 0 then it does not remove the server from the server ready queue and it sets the budget expiration event equal to X_s, which means that the server is overrunning its budget (i.e., there will not be a scheduler event until `OverrunUnlock` is called). Also, the scheduler indicates that the server is in overrun state by setting the `overrun` flag, line (7) in Figure 8, to true. Otherwise, the scheduler removes the server from the server ready queue.

Figure 10 shows the `OverrunUnlock` function. In this function, the released resource is removed from both the local and global resource queues and the system and subsystem ceilings are updated, which may decrease them. If the system/subsystem ceiling is decreased, the function checks if there are servers/tasks in the blocked queue that are blocked by this shared resource. It will move them to the server/task ready queues, depending on their preemption levels and the system/subsystem ceilings. In line (9) in Figure 10, the function checks if it should call the global scheduler, and there are two cases to do this. The first case is when the server was in overrun state, then it should be removed from the ready queue. The second case is if the server, after releasing the resource, is not the highest priority server, then it will be preempted by another server. The global scheduler will be invoked through the `sysBusIntGen` system call, similar to the local scheduler in the SRP implementation. The reason is that there will be a task switch (so the VxWorks scheduler needs to be invoked), and of course also a server switch, but this can be handled without the help of the VxWorks scheduler. The global/local scheduler (in HSF) must have knowledge about the current absolute time in order to set the next scheduling event, so this time must be derived before calling the scheduler.

```
1: void OverrunUnlock (int local_res_id) {
2:         SrpUnlock(local_res_id);
3:         InterruptDisable( );
4:         GlobalResourceStackRemove(local_res_id); // Ceiling is updated
5:         if (GlobalCeilingHasChanged( ))
6:            MoveServersFromBlockedToReady( );
7:         RunningServer.nr_global_resources_locked–;
8:         NewServer = GetHighestPrioReadyServer( );
9:         if ((RunningServer.overrun == TRUE &&
10:             RunningServer.nr_global_resources_locked == 0) ||
11:             RunningServer.ID ≠ NewServer.ID)
12:                 GlobalScheduler( );
13:         InterruptEnable( );
14: }
```

Fig. 10. Unlock function for Overrun

At every new subsystem activation, the server checks if there has been an overrun in its previous instance. If so, this overrun time length is subtracted from the servers budget, in the case of using overrun with payback mechanism. The global scheduler measures the overrun time when it is called, in response to budget expiration, and when it is called in response to the unlock function. If the other version of overrun is used (ONP), then the budget of the subsystem does not change.

On all server activations, the preemption level of each server is checked against current system ceiling. If the preemption level is lower than ceiling, then the server is inserted in the blocked queue.

E. Skipping mechanism implementation

The skipping implementation uses the same data-structures as overrun for keeping track of system ceiling and blocked servers. What is further needed, in order to implement skipping, is a simple FIFO (First In First Out) queue for tasks, line (8) in Figure 11. Also, a post in the VxWorks task TCB is needed, line (2) in Figure 11. According to the SIRAP protocol, the time length of the critical section must be known (and therefore also stored) so that it can be compared against the remaining budget, in order to prevent the budget from overrunning. One disadvantage with our current implementation is that we only allow maximum one shared global resource per task. This implementation can easily be extended to support more than one global resource per task, by adding more data-structures to store the locking times of the resources.

```
/* The rest of struct WIND_TCB (VxWorks TCB) */
        int spare4; // We keep resource locking time here
};
queue GLOBAL_RESOURCES; // Used by Skipping
queue SERVER_BLOCKED_QUEUE; // Used by Skipping
struct SERVER_TCB {
        // Used by Skipping to queue tasks during self-blocking
        queue TASK_FIFO_QUEUE;
        /* The rest of the server TCB */
```

Fig. 11. Data-structures used by Skipping

When calling the `SkippingLock` function (Figure (12), it checks if the remaining budget is enough to lock and release the shared resource before the budget expires (line (4) in Figure (12)). If the remaining budget is sufficient, then the resource will be inserted in both the global and local resource queue, similar to the overrun mechanism mentioned earlier. If the remaining budget is not sufficient, then the resource will be inserted in the local resource queue and the local system ceiling is updated, finally, the task is suspended in line (12) in Figure (12). Note that the rest of the function, lines (13-17), will not be executed until this task is moved to the ready queue. When the task is executed next time, it will continue from line (13) and insert the shared resource (line (14)) in the global resource queue, then update the global system ceiling and finally start executing in the critical section. Whenever a server starts to execute, after it has been released, its local scheduler checks if there are tasks that are suspended (by checking the `TASK_FIFO_QUEUE`), if any, it moves them (in FIFO order) to the ready queue. In this way, skipping affects the local scheduler while overrun does not.

```
1: void SkippingLock (local_res_id) {
2:      InterruptDisable( );
3:      RemainBudget = CalcRemainBudget(RunningServer);
4:      if (RemainBudget ≥ RunningTask.spare4) {
5:         GlobalResourceStackInsert(local_res_id); // Ceiling is updated
6:         SrpLock(local_res_id);
7:      }
8:      else { // Budget is not enough, block the task
9:         SrpLock(local_res_id);
10:        BlockedQueueInsert(RunningTask);
11:        InterruptEnable( );
12:        TaskSuspend(RunningTask); // This call will block...
13:        InterruptDisable( ); // ...cont. here when task is awakened
14:        GlobalResourceStackInsert(local_res_id); // Ceiling is updated
15:     }
16:     InterruptEnable( );
17: }
```

Fig. 12. Lock function for Skipping

The `SkippingUnlock` function is similar to the `OverrunUnlock` function (Figure 10), but with two differences. The first one is that skipping does not need to keep count of the number of locked global resources, and second, skipping will call the scheduler only if there is a server in the ready queue that has higher priority than the currently running server. In case of nested critical sections, the task call `SkippingLock/SkippingUnlock` functions only when it access and release the outermost shared resource, and the ceiling of the outermost shared resource equals to the highest ceiling of the nested shared resources.

```
void SkippingUnlock (int local_res_id) {
        SrpUnlock(local_res_id);
        InterruptDisable( );
        GlobalResourceStackRemove(local_res_id); // Ceiling is updated
        if (GlobalCeilingHasChanged( ))
           MoveServersFromBlockedToReady( );
        NewServer = GetHighestPrioReadyServer( );
        if (RunningServer.ID != NewServer.ID)
           GlobalScheduler( );
        InterruptEnable( );
}
```

Fig. 13. Unlock function for Skipping

If the global system ceiling has changed then the servers, for which preemption level is higher than global system ceiling, are put in the server ready queue. If the new global system ceiling causes a higher priority server to be inserted in the ready queue, then current running server is removed, and the global scheduler is called.

IV. Evaluation

In order to compare the runtime overhead of both synchronization mechanisms, we generated 8 systems according to the setup illustrated in Figure 14. In this setup, a system S^i contains 5 servers with 8 tasks each, and each system has 2 global resources (2-6 tasks will access the global resources). We monitored both skipping and overrun with payback.

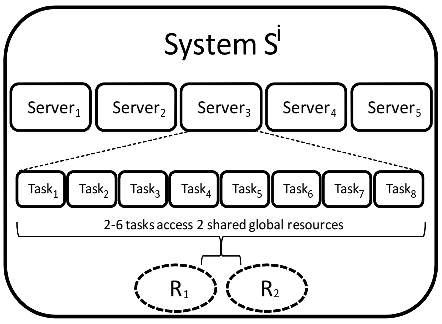

Fig. 14. Experimental setup

The metrics we used are the number of calls to the corresponding lock and unlock functions as well as the number of calls to the scheduler. The measurements were recorded in 600 time units (tu), and the range of tasks periods were scaled from 40 to 100 tu and the range of subsystem periods were 5-20 tu (we scaled the periods of subsystem and tasks in order to remove the effect of scheduling overhead). The task utilization was set to 15% per system.

Protocol	System															
	S^1	S^2	S^3	S^4	S^5	S^6	S^7	S^8	S^1	S^2	S^3	S^4	S^5	S^6	S^7	S^8
	# calls to `lock/unlock`								# calls to `Scheduler`							
Skipping	306	335	248	275	181	224	202	236	8	5	7	4	5	5	10	6
Overrun	304	335	247	275	181	225	203	236	47	13	40	16	36	17	30	25

TABLE I
EXPERIMENTAL RESULTS

Table I shows the results of running systems S^1 to S^8. Each of these systems (S^i) had different task/server parameters, different amount of resources and different resource users (depending on the generation of the systems). It is clear that the number of scheduler calls under the skipping mechanism is lower compared to using the overrun mechanism, which makes the runtime overhead for the skipping mechanism lower than the corresponding overhead when using the overrun mechanism. The difference between the corresponding `unlock` functions under skipping and under overrun is also the reason why the number of calls to the scheduler differs. For the overrun mechanism, the unlock function calls to the scheduler when the server unlocks the shared resource after overrun, while there is no such case in skipping, i.e., there is a higher risk that the scheduler is called in overrun, than in skipping (since there is two cases in overrun and one case in skipping). This explains the recorded results with respect to the number of scheduler calls.

V. Related work

Related work in the area of hierarchical scheduling originated in open systems [5] in the late 1990's, and it has been receiving an increasing research attention [6], [5], [7], [8], [9], [10], [11], [12]. However, the main focus of the research was on the schedulability analysis of independent tasks, and not much work has been conducted on the implementation of the proposed theories.

Among the few implementation work, Kim *et al.* [13] propose the SPIRIT uKernel that is based on a two-level hierarchical scheduling framework simplifying integration of real-time applications. The SPIRIT uKernel provides a separation between real-time applications by using partitions. Each partition executes an application, and uses the Fixed Priority Scheduling (FPS) policy as a local scheduler to schedule the application's tasks. An off-line scheduler (timetable) is used to schedule the partitions (the applications) on a global level. Each partition provides kernel services for its application and the execution is in user mode to provide stronger protection. Parkinson [14] uses the same principle and describes the VxWorks 653 operating system which was designed to support ARINC653. The architecture of VxWorks 653 is based on partitions, where a Module OS provides global resource and scheduling for partitions and a Partition OS implemented using VxWorks microkernel provides scheduling for application tasks.

The work presented in this paper differs from the above last two works in the sense that it implements a hierarchical scheduling framework in a commercial operating system without changing the OS kernel.

The implementation of a HSF in VxWorks without changing the kernel has been presented in [1] assuming the tasks are independent. In this paper, we extend this implementation by enabling sharing of logical resources among tasks located in the same and/or different subsystem(s). More recently, [15] implemented a two-level fixed priority scheduled HSF based on a timed event management system in the commercial real-time operating system μC/OS-II, however, the implementation is based on changing the kernel of the operating system, unlike the implementation in this paper.

In order to allow for dependencies among tasks, many theoretical works on synchronization protocols have been introduced for arbitrating accesses to shared logical resources, addressing the priority inversion problem, e.g., the Stack Resource Policy (SRP) [16]. For usage in a HSF, additional protocols have been proposed, e.g., the Hierarchical Stack Resource Policy (HSRP) [2], the Subsystem Integration and Resource Allocation Policy (SIRAP) [3], and the Bounded-delay Resource Open Environment (BROE) [17] protocols. The work in this paper concerns the former two, targeting systems implementing FPPS schedulers.

VI. Conclusion

In this paper we have presented our work on implementing synchronization protocols for hierarchical scheduling of tasks without doing any modification to the operating system kernel. We have presented two techniques for synchronization; overrun and skipping, and we have implemented the two techniques in our hierarchical scheduling framework for VxWorks [1]. The evaluation of these two techniques indicates that, when the synchronization protocol is implemented, skipping requires far less overhead when compared to the overrun mechanism.

Future work includes management of memory and interrupts towards a complete operating system virtualizer implemented as a layer on top of an arbitrary operating system kernel.

References

[1] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, July 2008, pp. 63–72.

[2] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS'06)*, December 2006, pp. 257–267.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in *Proceedings of the ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, October 2007, pp. 278–288.

[4] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[5] Z. Deng and J.-S. Liu, "Scheduling real-time applications in an open environment," in 18^{th} *IEEE Int. Real-Time Systems Symposium (RTSS'97)*, Dec. 1997.

[6] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *RTSS'05*, December 2005, pp. 389–398.

[7] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in 23^{th} *IEEE Int. Real-Time Systems Symposium (RTSS'02)*, Dec. 2002.

[8] T.-W. Kuo and C.-H. Li, "A fixed-priority-driven open environment for real-time applications," in 20^{th} *IEEE International Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.

[9] G. Lipari and S. K. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, May-Jun. 2000.

[10] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in 15^{th} *Euromicro Conference on Real-Time Systems (ECRTS'03)*, Jul. 2003.

[11] S. Matic and T. A. Henzinger, "Trading end-to-end latency for composability," in 26^{th} *IEEE International Real-Time Systems Symposium(RTSS'05)*, December 2005, pp. 99–110.

[12] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in 24^{th} *IEEE International Real-Time Systems Symposium (RTSS'03)*, Dec. 2003.

[13] D. Kim, Y. Lee, and M. Younis, "Spirit-ukernel for strongly partitioned real-time systems," in *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, 2000.

[14] L. K. P. Parkinson, "Safety critical software development for integrated modular avionics," in *Wind River white paper. URL http://www.windriver.com/whitepapers/*, 2007.

[15] 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.

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

[17] N. Fisher, M. Bertogna, and S. Baruah, "The design of an edf-scheduled resource-sharing open environment," in *Proceedings of the* 28^{th} *IEEE International Real-Time Systems Symposium (RTSS'07)*, December 2007, pp. 83–92.