Towards Hierarchical Scheduling in VxWorks*

Moris Behnam†, Thomas Nolte, Insik Shin, Mikael Åsberg
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås
Sweden

Reinder J. Bril
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5612 AZ Eindhoven
The Netherlands

Abstract

Over the years, we have worked on hierarchical scheduling frameworks from a theoretical point of view. In this paper we present our initial results of the implementation of our hierarchical scheduling framework in a commercial operating system VxWorks. The purpose of the implementation is twofold: (1) we would like to demonstrate feasibility of its implementation in a commercial operating system, without having to modify the kernel source code, and (2) we would like to present detailed figures of various key properties with respect to the overhead of the implementation. During the implementation of the hierarchical scheduler, we have also developed a number of simple task schedulers. We present details of the implementation of Rate-Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Finally, we present the design of our hierarchical scheduling framework, and we discuss our current status in the project.

1 Introduction

Correctness of today's embedded software systems generally relies not only on functional correctness, but also on extra-functional correctness, such as satisfying timing constraints. System development (including software development) can be substantially facilitated if (1) the system can be decomposed into a number of parts such that parts are developed and validated in isolation and (2) the temporal correctness of the system can be established by composing the correctness of its individual parts. For large-scale embedded real-time systems, in particular, advanced methodologies and techniques are required for temporal and spatial isolation all through design, development, and analysis, simplifying the development and evolution of complex industrial embedded software systems.

Hierarchical scheduling has shown to be a useful mechanism in supporting modularity of real-time software by providing temporal partitioning among applications. In hierarchical scheduling, a system can be hierarchically divided into a number of subsystems that are scheduled by a global (system-level) scheduler. Each subsystem contains a set of tasks that are scheduled by a local (subsystem-level) scheduler. The Hierarchical Scheduling Framework (HSF) allows for a subsystem to be developed and analyzed in isolation, with its own local scheduler, and then at a later stage, using an arbitrary global scheduler, it allows for the integration of multiple subsystems without violating the temporal properties of the individual subsystems analyzed in isolation. The integration involves a system-level schedulability test, verifying that all timing requirements are met. Hence, hierarchical scheduling frameworks naturally support *concurrent development* of subsystems. Our overall goal is to make hierarchical scheduling a cost-efficient approach applicable for a wide domain of applications, including automotive, automation, aerospace and consumer electronics.

Over the years, there has been a growing attention to HSFs for real-time systems. Since a two-level HSF [9] has been introduced for open environments, many studies have been proposed for its schedulability analysis of HSFs [14, 17]. Various processor models, such as bounded-delay [20] and periodic [24], have been proposed for multi-level HSFs, and schedulability analysis techniques have been developed for the proposed processor models [1, 7, 11, 18, 23, 24, 25]. Recent studies have been introduced for supporting logical resource sharing in HSFs [3, 8, 12].

Up until now, those studies have worked on various aspects of HSFs from a theoretical point of view. This paper

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

†Contact author: moris.behnam@mdh.se

presents our work towards a full implementation of a hierarchical scheduling framework, and due to space limitations, interested readers are referred to [30] for more details. We have chosen to implement it in a commercial operating system already used by several of our industrial partners. We selected the VxWorks operating system, since there is plenty of industrial embedded software available, which can run in the hierarchical scheduling framework.

The outline of this paper is as follows: Section 2 presents related work on implementations of schedulers. Section 3 present our system model. Section 4 gives an overview of VxWorks, including how it supports the implementation of arbitrary schedulers. Section 5 presents our scheduler for VxWorks, including the implementation of Rate Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Section 6 presents the design, implementation and evaluation of the hierarchical scheduler, and finally Section 7 summarizes the paper.

2 Related work

Looking at related work, recently a few works have implemented different schedulers in commercial real-time operating systems, where it is not feasible to implement the scheduler directly inside the kernel (as the kernel source code is not available). Also, some work related to efficient implementations of schedulers are outlined.

Buttazzo and Gai [4] present an implementation of the EDF scheduler for the ERIKA Enterprise kernel [10]. The paper discusses the effect of time representation on the efficiency of the scheduler and the required storage. They use the Implicit Circular Timer's Overflow Handler (ICTOH) algorithm which allows for an efficient representation of absolute deadlines in a circular time model.

Diederichs and Margull [6] present an EDF scheduler plug-in for OSEK/VDX based real-time operating systems, widely used by automotive industry. The EDF scheduling algorithm is implemented by assigning priorities to tasks according to their relative deadlines. Then, during the execution, a task is released only if its absolute deadline is less than the one of the currently running task. Otherwise, the task will be delayed until the time when the running task finishes its execution.

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 offline 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 [21] 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 last two works in the sense that it implements a hierarchical scheduling framework in a commercial operating system without changing the OS kernel. Furthermore, the work differs from the above approaches in the sense that it implements a hierarchical scheduling framework intended for open environments [9], where real-time applications may be developed independently and unaware of each other and still there should be no problems in the integration of these applications into one environment. A key here is the use of well defined *interfaces* representing the collective resource requirements by an application, rich enough to allow for integration with an arbitrary set of other applications without having to redo any kind of application internal analysis.

3 System model

In this paper, we only consider a simple periodic task model $\tau_i(T_i, C_i, D_i)$ where T_i is the task period, C_i is a worst-case execution time requirement, and D_i is a relative deadline ($0 < C_i \leq D_i \leq T_i$). The set of all tasks is denoted by Γ ($\Gamma = \{\tau_i |$ `for all` $i = 1, .., n\}$ where n is the number of tasks).

We assume that all tasks are independent of each other, i.e., there is no sharing of logical resources between tasks and tasks do no suspend themselves.

The HSF schedules subsystems $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the set representing the whole system of subsystems. Each subsystem S_s consists of a set of tasks and a local scheduler (RM or EDF), and the global (system) scheduler (RM or EDF). The collective real-time requirements of S_s is referred to as a *timing-interface*. The subsystem interface is defined as (P_s, Q_s), where P_s is a subsystem period, and Q_s is a budget that represents an execution time requirement that will be provided to the subsystem S_s every period P_s.

4 VxWorks

VxWorks is a commercial real-time operating system developed by Wind River with a focus on performance, scalability and footprint. Many interesting features are provided with VxWorks, which make it widely used in industry, such as; Wind micro-kernel, efficient task management and multitasking, deterministic context switching, efficient interrupt and exception handling, POSIX pipes, counting

semaphores, message queues, signals, and scheduling, preemptive and round-robin scheduling etc. (see [29] for more details).

The VxWorks micro-kernel supports the priority preemptive scheduling policy with up to 256 different priority levels and a large number of tasks, and it also supports the round robin scheduling policy.

VxWorks offers two different modes for application-tasks to execute; either kernel mode or user mode. In kernel mode, application-tasks can access the hardware resources directly. In user mode, on the other hand, tasks can not directly access hardware resources, which provides greater protection (e.g., in user mode, tasks can not crash the kernel). Kernel mode is provided in all versions of VxWorks while user mode was provided as a part of the Real Time Process (RTP) model, and it has been introduced with VxWorks version 6.0 and beyond.

In this paper, we are considering kernel mode tasks since such a design would be compatible with all versions of VxWorks and our application domains include systems with a large legacy in terms of existing source codes. We are also considering fixed priority preemptive scheduling policy for the kernel scheduler (not the round robin scheduler). A task's priority should be set when the task is created, and the task's priority can be changed during the execution. Then, during runtime, the highest priority ready task will always execute. If a task with priority higher than that of the running task becomes ready to execute, then the scheduler stops the execution of the running task and instead executes the one with higher priority. When the running task finishes its execution, the task with the highest priority among the ready tasks will execute.

When a task is created, an associated Task Control Block (TCB) is created to save the task's context (e.g., CPU environment and system resources, during the context switch). Then, during the life-cycle of a task the task can be in one or a combination of the following states [28] (see Figure 1):

- **Ready state**, the task is waiting for CPU resources.
- **Suspended state**, the task is unavailable for execution but not delayed or pending.
- **Pending state**, the task is blocked waiting for some resource other than the CPU.
- **Delayed state**, the task is sleeping for some time.

Note that the kernel scheduler sorts all tasks that are ready to execute in a queue called the *ready queue*.

4.1 Scheduling of time-triggered periodic tasks

A periodic task is a task that becomes ready for execution periodically once every n-th time unit, i.e., a new instant of the task is executed every constant period of time.

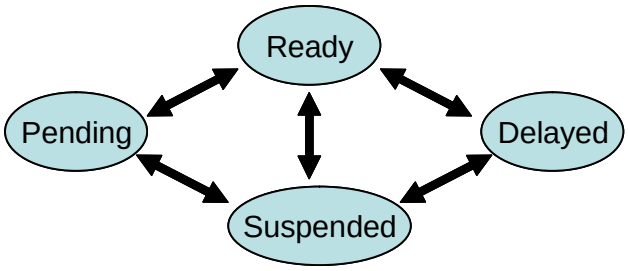

Figure 1. The application task state.

Most commercial operating systems, including VxWorks, do not directly support the periodic task model [19]. To implement a periodic task, when a task finishes its execution, it sleeps until the beginning of its next period. Such periodic behaviour can be implemented in the task by the usage of timers. Note that a task typically does not finish its execution at the same time always, as execution times and response times vary from one period to another. Hence, using timers may not be easy and accurate as the task needs to evaluate the time for next period relative to the current time, whenever it finishes its execution. This is because preemption may happen between the time measurement and calling the sleep function.

In this project we need to support periodic activation of *servers* in order to implement the hierarchical scheduling framework. The reason for this is that we base our hierarchical scheduling framework around the periodic resource model [24], and a suitable implementation of the periodic resource model is achieved by the usage of a server based approach similar to the periodic servers [16, 26] that replenish their budget every constant period, i.e., the servers behave like periodic tasks.

4.2 Supporting arbitrary schedulers

There are two ways to support arbitrary schedulers in VxWorks:

1. Using the VxWorks custom kernel scheduler [27].
2. Using the original kernel scheduler and manipulating the ready queue by changing the priority of tasks and/or activating and suspending tasks.

In this paper, we are using the second approach since implementing the custom kernel scheduler is a relatively complex task compared with manipulating the ready queue. However, it will be interesting to compare between the two methods in terms of CPU overhead, and we leave this as a future work.

In the implementation of the second solution, we have used an Interrupt Service Routine (ISR) to manipulate the

tasks in the ready queue. The ISR is responsible for adding tasks in the ready queue as well as changing their priorities according to the hierarchical scheduling policy in use. In the remainder of this paper, we refer to the ISR as the User Scheduling Routine (USR). By using the USR, we can implement any desired scheduling policy, including common ones such as Rate Monotonic (RM) and Earliest Deadline First (EDF).

5 The USR custom VxWorks scheduler

This section presents how to schedule periodic tasks using our scheduler, the User Scheduling Routine (USR).

5.1 Scheduling periodic tasks

When a periodic task finishes its execution, it changes its state to suspended by explicitly calling the suspend function. Then, to implement a periodic task, a timer could be used to trigger the USR once every new task activation time to release the task (to put it in the ready queue).

The solution to use a timer triggering the USR once every new period can be suitable for systems with a low number of periodic tasks. However, if we have a system with n periodic tasks such a solution would require the use of n timers, which could be very costly or not even possible. In this paper we have used a scalable way to solve the problem of having to use too many timers. By multiplexing a single timer, we have used a single timer to serve n periodic tasks.

The USR stores the next activation time of all tasks (absolute times) in a sorted (according to the closest time event) queue called Time Event Queue (TEQ). Then, it sets a timer to invoke the USR at the time equal to the shortest time among the activation times stored in the TEQ. Also, the USR checks if a task misses its deadline by inserting the deadline in the TEQ. When the USR is invoked, it checks all task states to see if any task has missed its deadline. Hence, an element in the TEQ contains (1) the absolute time, (2) the id of task that the time belongs to, and (3) the event type (task next activation time or absolute deadline). Note that the size of the TEQ will be $2 * n * B$ bytes (where B is the size in bytes of one element in the TEQ) since we need to save the task's next period time and deadline time.

When the USR is triggered, it checks the cause of the triggering. There are two causes for the USR to be triggered: (1) a task is released, and (2) the USR will check for deadline misses. For both cases, the USR will do the following:

- Update the next activation and/or the absolute deadline time associated with the task that caused triggering of the USR in the TEQ and re-insert it in the TEQ according to the updated times.
- Set the timer equal to the shortest time in the TEQ so that the USR will be triggered at that time.
- For task release, the USR changes the state of the task to Ready. Also, it changes priorities of tasks if required depending on the scheduler (EDF or RM). For deadline miss checking, the USR checks the state of the task to see if it is Ready. If so, the task missed its deadline, and the deadline miss function will be activated.

Updating the next activation time and absolute deadline of a task in the TEQ is done by adding the period of the task that caused the USR invocation to the current absolute time. The USR does not use the system time as a time reference. Instead it uses a time variable as a time reference. The reason for using a time variable is that we can, in a flexible manner, select the size of variables that save absolute time in bits. The benefits of such an approach is that we can control the size of the TEQ since it saves the absolute times, and it also minimizes the overhead of implementing 64 bits operations on 32 bit microprocessor [4], as an example. The reference time variable t_s used to indicate the time of the next activation, is initialized (i.e., $t_s = 0$) at the first execution of the USR. The value of t_s is updated every time that the USR executes and it will be equal to the time given by the TEQ that triggered the USR.

When a task τ_i is released for the first time, the absolute next activation time is equal to $t_s + T_i$ and its absolute deadline is equal to $t_s + D_i$.

To avoid time consuming operations, e.g., multiplications and divisions, that increase the system overhead inherent in the execution of the USR, all absolute times (task periods and relative deadlines) are saved in system tick unit (system tick is the interval between two consecutive system timer interrupts). However, depending on the number of bits used to store the absolute times, there is a maximum value that can be saved safely. Hence, saving absolute times in the TEQ may cause problems related to overrun of time, i.e., the absolute times become too large such that the value can not be stored using the available number of bits. To avoid this problem, we apply a wrapping algorithm which wraps the absolute times at some point in time, so the time will restart again. Periods and deadlines should not exceed the wrap-around value.

The input of the timer should be in a relative time, so evaluating the time at which to trigger the USR again (next time) is done by $TEQ[1] - t_s$ where $TEQ[1]$ is the first element in the queue after updating the TEQ as well as sorting it, i.e., the closest time in the TEQ. The USR checks to see if there are more than one task that have the same current activation time and absolute deadline. If so, the USR serves all these tasks to minimize the unnecessary overhead of executing the USR several times.

5.2 RM scheduling policy

Each task will have a fixed priority during run-time when Rate Monotonic (RM) is used, and the priorities are assigned according to the RM scheduling policy. If only RM is used in the system, no additional operations are required to be added to the USR since the kernel scheduler schedules all tasks directly according to their priorities, and the higher priority tasks can preempt the execution of the lower priority task. Hence, the implementation overhead for RM will be limited to the overhead of adding a task in the ready queue and managing the timer for the next period (saving the absolute time of the new period and finding the shortest next time in the TEQ) for periodic tasks.

The schedulability analysis for each task is as follows [15];

$$\forall \tau_i \in \Gamma, 0 < \exists t \leq T_i \ \ \texttt{dbf}(i,t) \leq t. \tag{1}$$

And $\texttt{dbf}(i,t)$ is evaluated as follows

$$\texttt{dbf}(i,t) = C_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k, \tag{2}$$

where $\texttt{HP}(i)$ is the set of tasks with priority higher than that of τ_i.

Eq. (2) can be easily modified to include the effect of using the USR on the schedulability analysis. Note that the USR will be triggered at the beginning of each task to release the task, so it behaves like a periodic task with priority equal to the maximum possible priority (the USR can preempt all application tasks). Checking the deadlines for tasks by using the USR will add more overhead, however, also this overhead has a periodic nature as the task release presented previously.

Eq. (3) includes the deadline and task release overhead caused by the USR in the response time analysis,

$$\begin{aligned}\texttt{dbf}(i,t) \ = \ \ C_i \ + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k \ + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil X_R \\ + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t + T_j - D_j}{T_j} \right\rceil X_D\end{aligned} \tag{3}$$

where X_R is the worst-case execution time of the USR when a task is released and X_D is the worst-case execution time of the USR when it checks for deadline misses (currently, in case of deadline misses, the USR will only log this event into a log file).

5.3 EDF scheduling policy

For EDF, the priority of a task changes dynamically during run-time. At any time t, the task with shorter deadline will execute first, i.e., will have the highest priority. To implement EDF in the USR, the USR should update the priorities of all tasks that are in the Ready Queue when a task is added to the Ready Queue, which can be costly in terms of overhead. Hence, on one hand, using EDF on top of commercial operating systems may not be efficient depending on the number of tasks, due to this sorting. However, the EDF scheduling policy provides, on other hand, better CPU utilization compared with RM, and it also has a lower number of context switches which minimizes context switch related overhead [5].

In the approach presented in this paper, tasks are already sorted in the TEQ according to their absolute times due to the timer multiplexing explained earlier. Hence, as the TEQ is already sorted according to the absolute deadlines, the USR can easily decide the priorities of the tasks according to EDF without causing too much extra overhead for evaluating the proper priority for each task.

The schedulability test for a set of tasks that use EDF is shown in Eq. (4) [2] which includes the case when task deadlines are allowed to be less than or equal to task periods.

$$\forall t > 0, \quad \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i \leq t \tag{4}$$

The overhead of implementing EDF can also be added to Eq. (4). Hence, Eq. (5) includes the overhead of releasing tasks as well as the overhead of checking for deadline misses.

$$\begin{aligned}\forall t > 0, \quad \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil X_R \\ + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t + T_j - D_j}{T_j} \right\rceil X_D \leq t\end{aligned} \tag{5}$$

5.4 Implementation and overheads of the USR

To implement the USR, we have used the following VxWorks service functions;

- Q_PUT - insert a node into a multi-way queue (ready queue).
- Q_REMOVE - remove a node from a multi-way queue (ready queue).
- taskCreat - create a task.
- taskPrioritySet - set a tasks priority.

We present our initial results inherent in the implementation of the USR, implementing both the Rate Monotonic

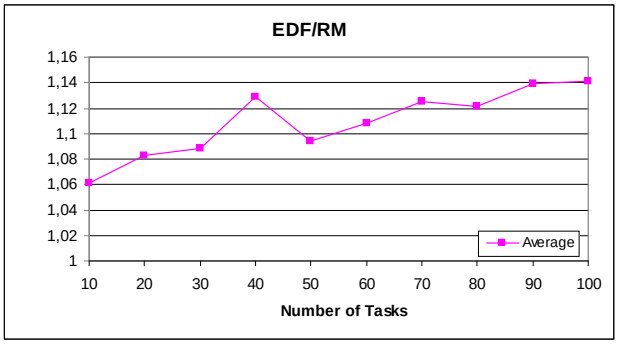

Figure 2. EDF normalized against RM, for average USR execution time.

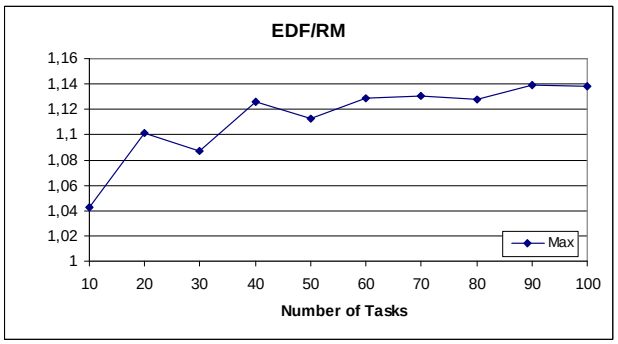

Figure 3. EDF normalized against RM, for maximum USR execution time.

(RM) scheduler as well as the Earliest Deadline First (EDF) scheduler. The implementations were performed on a ABB robot controller with a Pentium 200 MHz processor running the VxWorks operating system version 5.2. To trigger the USR for periodic tasks, we have used watchdog timers where the next expiration time is given in number of ticks. The watchdog uses the system clock interrupt routine to count the time to the next expiration. The platform provides system clock with resolution equal to $4500 ticks/s$. The measurement of the execution time of the USR is done by reading a timestamp value at the start as well as at the end of the USR's execution. Note that the timestamp is connected to a special hardware timer with resolution $12000000 ticks/s$.

Table 1 shows the execution time of the USR when it performs RM and EDF scheduling, as well as deadline miss checking, as a function of the number of tasks in the system. The worst case execution time for USR will happen when USR deletes and then inserts all tasks from and to TEQ and to capture this, we have selected a same period for all tasks. The table shows the minimum, maximum and average out of 50 measured values. Comparing between the results of the three cases (EDF, RM, deadline miss), we can see that there is no big difference in the execution time of the USR. The reason for this result is that the execution of the USR for EDF, RM and deadline miss checking all includes the overhead of deletion and re-inserting the tasks in the TEQ, which is the dominating part of the overhead. As expected, EDF causes the largest overhead because it changes the priority of all tasks in the ready queue during run-time. Figures 2-3 show that EDF imposes between $6 - 14\%$ extra overhead compared with RM.

6 Hierarchical scheduling

A Hierarchical Scheduling Framework (HSF) supports CPU sharing among subsystems under different scheduling policies. Here, we consider a two-level scheduling framework consisting of a global scheduler and a number of local schedulers. Under global scheduling, the operating system (global) scheduler allocates the CPU to subsystems. Under local scheduling, a local scheduler inside each subsystem allocates a share of the CPU (given to the subsystem by the global scheduler) to its own internal tasks (threads).

We consider that each subsystem is capable of exporting its own interface that specifies its collective real-time CPU requirements. We assume that such a subsystem interface is in the form of the periodic resource model (P_s, Q_s) [24]. Here, P_s represents a *period*, and Q_s represents a *budget*, or an execution time requirement within the period $(Q_s < P_s)$. By using the periodic resource model in hierarchical scheduling frameworks, it is guaranteed [24] that all timing constraints of internal tasks within a subsystem can be satisfied, if the global scheduler provides the subsystem with CPU resources according to the timing requirements imposed by its subsystem interface. We refer interested readers to [24] for how to derive an interface (P_s, Q_s) of a subsystem, when the subsystem contains a set of internal independent periodic tasks and the local scheduler follows the RM or EDF scheduling policy. Note that for the derivation of the subsystem interface (P_s, Q_s), we use the demand bound functions that take into account the overhead imposed by the execution of USR (see Eq. (3) and (5)).

6.1 Hierarchical scheduling implementation

Global scheduler: A subsystem is implemented as a periodic server, and periodic servers can be scheduled in a similar way as scheduling normal periodic tasks. We can use the

Number of tasks	X_R (RM)			X_R (EDF)			X_D (Deadline miss check)		
	Max	Average	Min	Max	Average	Min	Max	Average	Min
10	71	65	63	74	70	68	70	60	57
20	119	110	106	131	118	115	111	100	95
30	172	158	155	187	172	169	151	141	137
40	214	202	197	241	228	220	192	180	175
50	266	256	249	296	280	275	236	225	219
60	318	305	299	359	338	331	282	268	262
70	367	352	341	415	396	390	324	309	304
80	422	404	397	476	453	444	371	354	349
90	473	459	453	539	523	515	415	398	393
100	527	516	511	600	589	583	459	442	436

Table 1. USR execution time in μs, the maximum, average and minimum execution time of 45 measured values for each case.

same procedure described in Section 5 with some modifications in order to schedule servers. Each server should include the following information to be scheduled: (1) server period, (2) server budget, (3) remaining budget, (4) pointer to the tasks that belong to this server, and (5) the type of the local scheduler (RM or EDF) (6) local TEQ. Moreover, to schedule servers we need:

- **Server Ready Queue** to store all servers that have non zero remaining budget. When a server is released at the beginning of its period, its budget will be charged to the maximum budget Q, and the server will be added to the Server Ready Queue. When a server executes its internal tasks for some time x, then the remaining budget of the server will be deceased with x, i.e., reduced by the time that the server execute. If the remaining budget becomes zero, then the server will hand over the control to the global scheduler to select and remove the highest priority server from Server Ready Queue.

- **Server TEQ** to release the server at its next absolute periodic time since we are using periodic servers and also track their remaining budgets.

Figures 4 illustrates the implementation of HSF in VxWorks. The Server Ready Queue is managed by the routine that is responsible for scheduling the servers. Tracking the remaining budget of a server is solved as follows; whenever a server starts running, it sets an absolute time at which the server budget expire and it equals to the current time plus its remaining budget. This time is added to the server event Queue to be used by the timer to trigger an event when the server budget expires. When a server is preempted by another server, it updates the remaining budget by subtracting the time that has passed since the last release. When the server executes its internal tasks until the time when the server budget expiry event triggers, it will set its remaining budget to zero, and the scheduling routine removes the server from the Server Ready Queue.

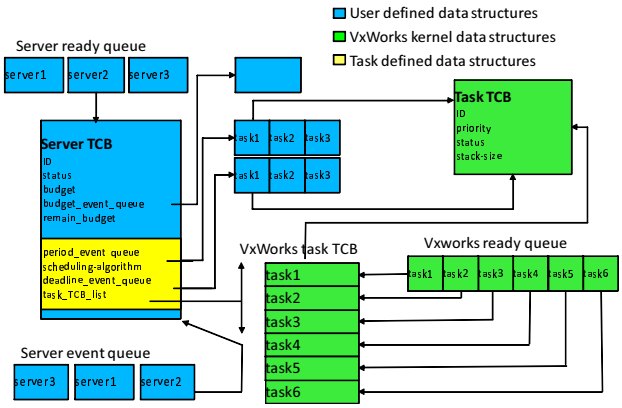

Figure 4. The implementation of HSF in VxWorks.

Local scheduler: When a server is given the CPU resources, the ready tasks that belong to the server will be able to execute. We have investigated two approaches to deal with the tasks in the Ready Queue when a server is given CPU resources:

- All tasks that belong to the server that was previously running will be removed from the Ready Queue, and all ready tasks that belong to the new running server will be added to the Ready Queue, i.e., swapping of the servers' task sets. To remove tasks from the Ready Queue, the state of the tasks is changed to suspend state. However, this will cause a problem since the state of the tasks that finish their execution is also changed to suspend and when the server run again it

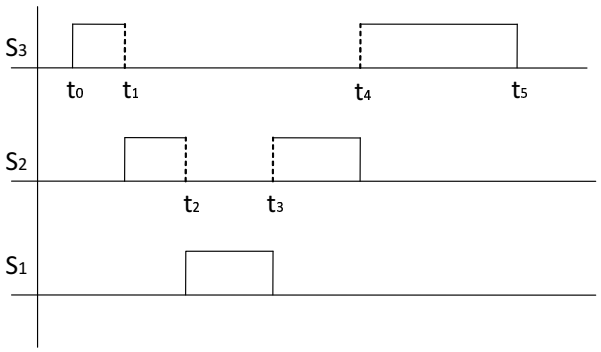

Figure 5. Simple servers execution example.

will add non-ready tasks to the Ready Queue. To solve this problem, an additional flag is used in the task's TCB to denote whether the task was removed from Ready Queue and enter to suspend state due to budget expiration of its server or due to finishing its execution.

- The priority of all tasks that belong to the preempted server will be set to a lower (the lowest) priority, and the priority of all tasks that belong to the new running server will be raised as if they were executing exclusively on the CPU, scheduled according to the local scheduling policy in use by the subsystem.

The advantage of the second approach is that it can give the unused CPU resources to tasks that belong to other servers. However, the disadvantage of this approach is that the kernel scheduler always sorts the tasks in the Ready Queue and the number of tasks inside Ready Queue using the second approach will be higher which may impose more overhead for sorting tasks. In this paper, we consider the first approach since we support only periodic tasks. When a server is running, all interrupts that are caused by the local TEQ, e.g., releasing tasks and checking deadline misses, can be served without problem. However, if a task is released or its deadline occurs during the execution of another server, the server that includes the task, may miss this event. To solve this problem, when the server starts running after server preemption or when it finishes its budget, it will check for all past events (including task release and deadline miss check events) in the local TEQ that have absolute time less than the current time, and serve them.

Note that the time wrapping algorithm described in section 5.1 should take into account all local TEQ's for all servers and the server event queue, because all these event queues share the same absolute time.

Figure 5 illustrates the implementation of hierarchical scheduling framework which includes an example with three servers S_1, S_2, S_3 with global and local RM schedulers, the priority of S_1 is the highest and the priority of S_3 is the lowest. Suppose a new period of S_3 starts at time t_0 with a budget equal to Q_3. Then, the USR will change the state of S_3 to Ready, and since it is the only server that is ready to execute, the USR will;

- add the time at which the budget will expire, which equals to $t_0 + Q_3$, into the server event queue and also add the next period event in the server event queue.
- check all previous events that have occurred while the server was not active by checking if there are task releases or deadline checks in the time interval of $[t^*, t_0]$, where t^* is the latest time at which the budget of S_3 has been expired.
- start the local scheduler.

At time t_1 the server S_2 becomes Ready and it has higher priority than S_3. So S_2 will preempt S_3 and in addition to the previously explained action, the USR will remove all tasks that belong to S_3 from the ready queue and save the remaining budget which equals to $Q_3 - (t_1 - t_0)$. Also the USR will remove the budget expiration event from the server event queue. Note that when S_3 executes next time it will use the remaining budget to calculate the budget expiration event.

Number of servers	Max	Average	Min
10	91	89	85
20	149	146	139
30	212	205	189
40	274	267	243
50	344	333	318
60	412	400	388
70	483	466	417
80	548	543	509
90	630	604	525
100	689	667	570

Table 2. Maximum, average and minimum execution time of the USR with 100 measured values as a function of the number of servers.

The USR execution time depends on the number of the servers, and the worst case happens when all servers are released at the same time. In addition, the execution time of the USR also depends on the number of ready tasks in both the currently running server to be preempted as well as the server to preempt. The USR removes all ready tasks that belong to the preempted server from ready queue and adds all ready tasks that belong to the preempting server with highest priority into the ready queue. Here, the worst case scenario is that all tasks of both servers are ready at that time. Table 2 shows the execution time of the USR (when a server is released) as a function of the number of servers

using RM as a global scheduler at the worst case, where all the servers are released at the same time, just like the case shown in the previous section. Here, we consider that each server has a single task in order to purely investigate the effect of the number of servers on the execution time of the USR.

6.2 Example

In this section, we will show the overall effect of implementing the HSF using a simple example, however, the results from the following example are specific for this example because, as we showed in the previous section, the overhead is a function of many parameters affect the number of preemptions such as number of servers, number of tasks, servers periods and budgets. In this example we use RM as both local and global scheduler, and the servers and associated tasks parameters are shown in Table 3. Note that $T_i = D_i$ for all tasks.

The measured overhead utilization is about 2.85% and the measured release jitter for task τ_3 in server S_3 (which is the lowest priority task in the lowest priority server) is about $49ms$. The measured worst case response time is $208.5ms$ and the finishing time jitter is $60ms$. These results indicate that the overhead and performance of the implementation are acceptable for further development in future project.

7 Summary

This paper has presented our work on the implementation of our hierarchical scheduling framework in a commercial operating system, VxWorks. We have chosen to implement it in VxWorks so that it can easily be tested in an industrial setting, as we have a number of industrial partners with applications running on VxWorks and we intend to use them as case studies for an industrial deployment of the hierarchical scheduling framework.

This paper demonstrates the feasibility of implementing the hierarchical scheduling framework through its implementation over VxWorks. In particular, it presents several measurements of overheads that its implementation imposes. It shows that a hierarchical scheduling framework can effectively achieve the clean separation of subsystems in terms of timing interference (i.e., without requiring any temporal parameters of other subsystems) with reasonable implementation overheads.

In the next stage of this implementation project, we intend to implement synchronization protocols in hierarchical scheduling frameworks, e.g., [3]. In addition, our future work includes supporting sporadic tasks in response to specific events such as external interrupts. Instead of allowing them to directly add their tasks into the ready queue, we consider triggering the USR to take care of such additions. We also plan to support aperiodic tasks while bounding their interference to periodic tasks by the use of some server-based mechanisms. Moreover, we intend to extend the implementation to make it suitable for more advanced architectures including multicore processors.

Acknowledgements

The authors wish to express their gratitude to the anonymous reviewers for their helpful comments, as well as to Clara Maria Otero Pérez for detailed information regarding the implementation of hierarchical scheduling as a dedicated layer on top of pSoSystem, which is marketed by Wind River (see [22] for more details) and suggestions for improving our work.

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proc. of the Fourth ACM International Conference on Embedded Software*, September 2004.

[2] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Journal of Real-Time Systems*, 2:301–324, 1990.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proc. of the 7th ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, 2007.

[4] G. Buttazzo and P. Gai. Efficient implementation of an EDF scheduler for small embedded systems. In *Proc. of the* 2^{nd} *International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'06) in conjunction with the* 18^{th} *Euromicro International Conference on Real-Time Systems (ECRTS'06)*.

[5] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgement day. *Real-Time Systems*, 29(1):5–26, January 2005.

[6] C. Diederichs, U. Margull, F. Slomka, G. Wirrer. An application-based EDF scheduler for osek/vdx. In *DATE '08: Proc. of the conference on Design, automation and test in Europe*, 2008.

[7] R. I. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. In *Proc. of the* 26^{th} *IEEE International Real-Time Systems Symposium (RTSS'05)*, December 2005.

[8] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proc. of the* 27^{th} *IEEE International Real-Time Systems Symposium (RTSS'06)*, December 2005.

[9] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. of IEEE Real-Time Systems Symposium*, pages 308–319, December 1997.

[10] Evidence Srl. ERIKA Enterprise RTOS. URL: http://www.evidence.eu.com.

[11] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proc. of IEEE Real-Time Systems Symposium*, pages 26–35, December 2002.

$S_1(P_1 = 5, Q_1 = 1)$			$S_2(P_2 = 6, Q_2 = 1)$			$S_3(P_3 = 70, Q_3 = 20)$		
τ_i	T_i	C_i	τ_i	T_i	C_i	τ_i	T_i	C_i
τ_1	20	1	τ_1	25	1	τ_1	140	7
τ_2	25	1	τ_2	35	1	τ_2	150	7
τ_3	30	1	τ_3	45	1	τ_3	300	30
τ_4	35	1	τ_4	50	1			
τ_5	40	7	τ_5	55	7			
-	-	-	τ_6	60	7			

Table 3. System parameters in μs.

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

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

[14] T.-W. Kuo and C. Li. A fixed-priority-driven open environment for real-time applications. In *Proc. of IEEE Real-Time Systems Symposium*, pages 256–267, December 1999.

[15] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proc. of IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[16] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. of 8^{th} IEEE International Real-Time Systems Symposium (RTSS'87)*, pages 261–270.

[17] G. Lipari and S. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 166–175, May 2000.

[18] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proc. of Euromicro Conference on Real-Time Systems*, July 2003.

[19] J. Liu. Real-time systems. *Prentice Hall*, 2000.

[20] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 75–84, May 2001.

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

[22] C.M. Otero Perez and I. Nitescu. Quality of Service Resource Management for Consumer Terminals: Demonstrating the Concepts. In *Proc. Work in Progress Session of the 14th Euromicro Conference on Real-Time Systems, ,* June 2002.

[23] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proc. of Euromicro Conference on Real-Time Systems*, June 2002.

[24] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of IEEE Real-Time Systems Symposium*, pages 2–13, December 2003.

[25] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proc. of IEEE Real-Time Systems Symposium*, December 2004.

[26] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[27] Wind River. VxWorks KERNEL PROGRAMMERS GUIDE 6.2.

[28] Wind River. VxWorks PROGRAMMERS GUIDE 5.5.

[29] Wind River. Wind River VxWorks 5.x. http://www.windriver.com/.

[30] M. Åsberg. On Hierarchical Scheduling in VxWorks. Master thesis, Department of Computer Science and Electronics, Mälardalen University, Sweden, 2008.