Efficient Development of Real-Time Systems Using Hybrid Scheduling

Jukka Mäki-Turja, Kaj Hänninen and Mikael Nolin
Mälardalen Research and Technology Centre (MRTC), Västerås Sweden

Abstract

This paper will show how advanced embedded real-time systems, with functionality ranging from time-triggered control functionality to event-triggered user interaction, can be made more efficient. Efficient with respect to development effort as well as run-time resource utilization. This is achieved by using a hybrid, static and dynamic, scheduling strategy. The approach is applicable even for hard real-time systems since tight response time guarantees can be given by the response time analysis method for tasks with offsets.

An industrial case study will demonstrate how this approach enables more efficient use of computational resources, resulting in a cheaper or more competitive product since more functionality can be fitted into legacy, resource constrained, hardware.

Keywords: real-time system design, efficient development, hybrid scheduled systems, response times.

1 Introduction

As the complexity of embedded real-time systems keeps growing, both by increases in size and in diversity, the developers are faced with the increasing challenge of modelling, analyzing, implementing and testing both the functional as well as the temporal behavior of these systems. This paper will present ways to simplify some of that complexity by introducing methods to verify the temporal correctness for a larger class of such systems.

Traditionally, one design parameter has been what execution model to choose. Two common and widespread execution models are the static and dynamic execution models:

- **Static scheduling**, where a schedule is produced off-line. The schedule contains all scheduling decisions, such as when to execute each task or to send each message. During run-time a simple dispatcher dispatches tasks according to the schedule. Static scheduling is sometimes referred to as time-triggered scheduling.
- **Dynamic scheduling**, where scheduling decisions are made on-line by a run-time scheduler. Typically some task attribute (such as priority or deadline) is used by the scheduler to decide what task to execute. The scheduler implements some queueing discipline, such as fixed priority scheduling or earliest deadline first. Dynamic scheduling is sometimes referred to as event-triggered scheduling.

Since both models have their pros and cons, the design decision of which one to use is not simple. A few trade-offs when choosing execution model are:

- **Overhead** – Since all scheduling and synchronization decision are made off-line in the static approach, the run-time overhead for scheduling is kept low. In dynamic scheduling these decisions are made on-line, often resulting in a larger overhead.
- **Responsiveness** – Statically scheduled systems are inflexible and have therefore limited possibility in responding to dynamic events, resulting in poor responsiveness. Dynamically scheduled systems, on the other hand, handles dynamic events naturally and can provide high degree of responsiveness.
- **Resource usage** – In order to provide some degree of responsiveness for dynamic events in the environment, statically scheduled systems tend to waste resources on redundant polling, whereas event-triggered dynamic schedulers only handle the actual events, enabling better service to soft or non-real time functionality when events do not occur at their maximum rate.
- **Overload** – In static scheduling the effects of overload are highly predictable. The exact capacity, e.g. in terms of number of inputs handled, is known and the effect of lost events, e.g. due to slow polling, can be predicted. In dynamic scheduling, no natural overload control is inherent. Instead, ad-hoc mechanisms are used to prevent, e.g., faulty sensors from flooding the systems with interrupts. A dynamically scheduled system which becomes overloaded is unpredictable, it is often difficult to assess which buffer will overflow and thus which tasks will miss their deadlines.

- **Determinism** – A statically scheduled system is highly deterministic, it executes according to the predefined schedule each time. A dynamically scheduled system, on the other hand, may exhibit different behavior each time the system is run, due to, e.g., race conditions on shared resources. This has a major impact on reproducibility, and thus also on the functional testability, of the system. Determinism also simplifies the verification process which is a major part when certifying safety critical applications.
- **Complex constraints** – Statically scheduled systems can handle more complicated inter-task relation constraints. For example, control systems, where control performance is important, need to have small (input and/or output) jitter, which is easier to accommodate in a static scheduler than with simpler dynamic scheduling parameters.
- **Adding new functionality** – Once a static schedule has been constructed it can be very hard to add new functionality in the system, a completely new schedule has to be constructed. For a dynamically scheduled system, new functions can be added with a minimum of impact on other parts of the system.

For further discussions on these trade-offs see [13] which advocates cyclic scheduling), and [25] which advocates dynamic, fixed priority, scheduling.

As can be seen, both approaches have their virtues and one often wishes to have both approaches available when developing embedded real-time applications. This desire is clearly illustrated by the last few years of development in the area of field busses for automotive applications. The Controller Area Network (CAN) [7] has been predominant in the automotive industry. CAN provides dynamic scheduling (using fixed priority scheduling). However, the automotive industry felt a need for a more dependable and predictable bus architecture. So when Kopetz brought attention to his Time Triggered Protocol (TTP) [12], which provides static scheduling, many automotive manufacturers and their sub-contractors embraced the new technology. It was soon recognized that TTP was a bit *too* static. Hence, a consortium of automotive manufacturers and sub-contractors started the development of FlexRay [9], which provides both static and dynamic scheduling. Also, on the operating-system side, products that support both static and dynamic scheduling have emerged. For instance, Arcticus Systems' operating system Rubus [1], and the open source real-time operating system Asterix [2]. In fact, most priority driven operating systems can implement hybrid static and dynamic scheduling by letting a dispatcher (a time-table) execute at highest priority.

Thus, we see that the need to combine static and dynamic scheduling have led to some practical solutions available today. However, one problem with systems that tries to combine static and dynamic scheduling is that they often consider the dynamic part as non real-time, e.g. [1, 9]. That is, dynamic scheduled tasks/messages are not given any response-time guarantees, only best-effort service is provided. However, in order to fully utilize the potential of combining static and dynamic scheduling in hard real-time systems, both the dynamic and the static parts need to be able to provide response-time guarantees. A recent study of industrial needs recognizes that one of the key issues for embedded systems is analyzability [15].

This paper presents a method to model hybrid, statically and dynamically, scheduled systems with the task model with offsets [14]. With this model, and the corresponding response time analysis, tight response time guarantees can be given also for dynamically scheduled tasks. The modelled system can be realized with commercially available operating systems support. Furthermore, in a case study we show how a legacy system at Volvo Construction Equipment could benefit from this approach by migrating functionality from the resource demanding statically scheduled part to the dynamically scheduled part, freeing system resources while still fulfilling original temporal constraints.

Paper Outline: Next, section 2 describes the type of systems studied in this paper. Section 3 shows how these systems can be modelled using the task model with offsets. Section 4 discusses related work. Section 5 illustrates, through a case study, how this approach can be applied to a legacy system, migrating functions from a static schedule, freeing system resources. Finally, section 6 presents our conclusions.

2 System description

In this paper, we address the issue of providing tight response-time guarantees to dynamically scheduled tasks running "in the background" of a static schedule. The system model contains:

- **Interrupts**. There may be multiple interrupt levels, i.e., an interrupt may be preempted by higher level interrupts.
- **A static cyclic schedule**.
  - A set of periodic static tasks (functions) are scheduled in the schedule. Each task has a known worst case execution time (WCET).
  - The schedule has a length (a duration) that is equal to the LCM (least common multiple) of all statically scheduled function periods. The schedule is constructed off-line by a scheduling tool.

- Each function is scheduled at an offset relative to the start of the schedule. This is also referred to as a function's *release time*.
- The static cyclic scheduler activates each function in the schedule at its release time. When the whole schedule has been executed the schedule is restarted from the beginning.

Interrupts may preempt the execution of statically scheduled functions.

- **A set dynamically dispatched tasks**. We call each such task a *dynamic task*. These tasks executes in the time slots available between interrupts and statically scheduled functions. Dynamic tasks are scheduled by a fixed priority preemptive scheduler. They are assumed to be periodic or, at least, to have a known minimum time between two invocations.

We assume that a static cyclic schedule has been constructed prior to the analysis of dynamic tasks. Furthermore, we assume that the schedule is valid even if its functions are preempted by interrupts. How a scheduler can generate a feasible schedule, with interfering interrupts, is described in [22].

2.1 Example system

Figure 1 shows a static cyclic schedule of length 20, with 4 functions released at times 0, 5, 10 and 15, with WCETs 4, 1, 1 and 3 respectively.

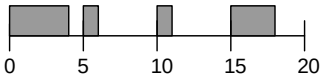

Figure 1. Example of static cyclic schedule

In figure 2 we see an example execution scenario when executing the schedule from figure 1, with one interfering interrupt source and one dynamically scheduled task (two instances of that task are activated). We make the observation that both interrupts and the static schedule act like higher priority tasks from the dynamic tasks' point of view.

One of the main objectives of this paper is to enable response-time calculations for dynamic tasks. The goal is to model static schedules (and interrupts) so as to incur as little interference on dynamic tasks execution as possible. Thus, modelling both functions' WCETs as well as their release times as accurately as possible.

3 Modelling the system

Classical response-time analysis (see e.g. [3, 5, 11]), assumes that a critical instant[1] occurs when all tasks

[1]Point in time, where the task under analysis is released for execution, resulting in the longest possible response-time.

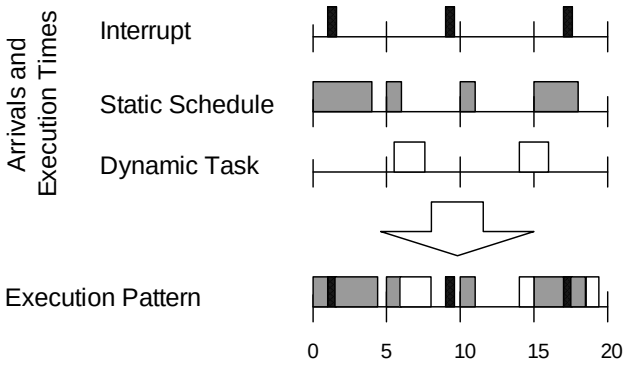

Figure 2. Example execution scenario

are released simultaneously. Using this model, the static schedule described in section 2, can be modelled as 4 tasks. These tasks would have a period of 20 and WCETs of 4, 1, 1, and 3 respectively. However, this approach is overly pessimistic since it assumes that all four static tasks can be released for execution at the same time. In our example, assuming no interrupt interference, a dynamic task with a WCET of 1, would have a response time of 10 (4+1+1+3+1). However, looking at figure 1 one can see that the actual worst possible response-time is 5 (if the dynamic tasks coincides with the static function scheduled at time 0).

In static schedules it is impossible for all static tasks to start at the same time. The task model with offset introduced by [18, 23] is able to capture the time separation in static schedules, and thus reduce the pessimism. In [14] we further reduced the pessimism in the corresponding response time formulae.

3.1 Task model with offsets

The task set, Γ, in [14] consists of a set of k transactions, $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by a periodic sequence of events with period T_i. A transaction Γ_i, contains $|\Gamma_i|$ number of tasks, and each task is activated when a relative time, *offset*, elapses after the arrival of the event.

τ_{ij} is used to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the number of the task within that transaction. A task τ_{ij} is defined by a worst case execution time (C_{ij}), an offset (O_{ij}), a deadline (D_{ij}), maximum jitter (J_{ij}), maximum blocking from lower priority tasks (B_{ij}), and a priority (P_{ij}). The task set Γ is formally expressed as follows:

$$\begin{aligned}\Gamma &:= \{\Gamma_1, \ldots, \Gamma_k\} \\ \Gamma_i &:= \langle \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\}, T_i \rangle \\ \tau_{ij} &:= \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij} \rangle\end{aligned}$$

There are no restrictions placed on offset, deadline or jitter. The maximum blocking time for a task, τ_{ij}, is the maximum time it has to wait for a resource which is locked by a lower priority task. In order to calculate the blocking time for a task, usually, a resource locking protocol like priority ceiling or immediate inheritance is needed. Algorithms to calculate blocking times for different resource locking protocols are presented in [6]. Priorities can be assigned with any method (e.g. rate monotonic, deadline monotonic, or user defined priorities). One must assume that the load of the task set is less than 100%.[2]

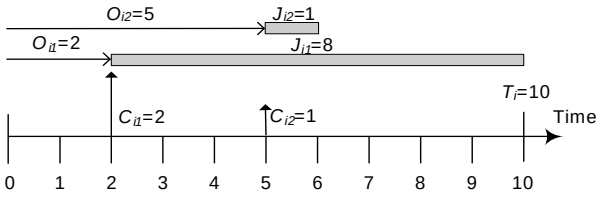

Figure 3. Example transaction

Parameters for an example transaction (Γ_i) with two tasks (τ_{i1} and τ_{i2}) is depicted in figure 3. The offset denotes the earliest possible release time of a task relative to the start of its transaction and jitter (illustrated by the shaded region) denotes maximum possible variability in the actual release of a task. The upward arrows denotes earliest possible release of a task and the height of the arrow corresponds to the amount of execution released. The end of the shaded region represents the latest possible release of a task.

3.2 System model

The system in section 2 can be modelled, and dynamic tasks subsequently analyzed for response times, with the above task model as follows (subscripts i, s, and d denote a generic interrupt, static, and dynamic transaction respectively):

- **Each interrupt** will be modelled as a transaction, Γ_i, containing one single task (i.e., $|\Gamma_i| = 1$) with T_i set to minimum inter-arrival time of the corresponding interrupt. These interrupt tasks will have the highest priorities in the system. If there are several interrupt levels, priorities are assigned accordingly, i.e., highest priority to highest interrupt level.
- **The static schedule** is modelled as one transaction, Γ_s, where each release time in the schedule is modelled as one task, τ_{sj}, where the offset ,O_{sj}, is set to the corresponding release time. The worst case execution time, C_{sj}, is set to the corresponding functions WCET. The priority, one suffices, for static tasks must be lower than for any interrupt, but higher than those for dynamic tasks.

  Our example schedule of figure 1 will be modelled as a transaction ($T_s = 20$) with 4 tasks, with offsets 0, 5, 10, 15 and worst case execution time of 4, 1, 1, 3 respectively.
- **Dynamic tasks** will have the most variability on how they are modelled. In the simplest case they are modelled exactly the same way as interrupts but with lower priorities. This situation corresponds to simple periodic (or sporadic) dynamic tasks with no jitter, no time separation (offsets), and no blocking. However depending on the nature of the dynamic tasks their corresponding transaction can be extended by:
  - jitter if there is variability in their periodicity,
  - by blocking if they share resources and providing the run-time system supports an analyzable resource sharing protocol, and
  - offsets if there are temporal dependencies, such as precedence, among dynamic tasks.

  Note that dynamic tasks cannot communicate with static tasks, via locked resources, since they must not affect their temporal behavior. However, there exist methods to communicate between these two systems that will not affect the temporal behavior of static tasks, see e.g. [17].

  Assuming the dynamic task of figure 2 is a sporadic task with minimum inter-arrival time of 10 time units and a release jitter of 3 time units, it is modelled as a transaction with $T_d = 10$ containing one task with $J_{dj} = 3$. The execution time is 2 and since it is the lowest priority task the blocking is zero ($C_{dj} = 2$ and $B_{dj} = 0$).

The formulae to calculate the response times rely on a relaxed critical instant assumption stating that only one task out of every transaction has to coincide with the critical instant. The complete formulae can be found in [14], and would, for our example system of figure 2, result in a response time of 5 time units for a dynamic task with $C_{dj} = 1$, assuming no interrupt interference.

Since all type of tasks, interrupt, static, and dynamic, can be analyzed for responsiveness, the inability of providing response time guarantees will no longer be a basis for rejecting an execution model for a function, thus making hybrid static and dynamic scheduling suitable even for hard real-time systems.

4 Related work

There has been number of research projects addressing the issue of combining several execution models

[2] This can easily be tested, and if not fulfilled some response-times may be infinite; rendering the task set unschedulable.

[20, 21, 4]. These provide reservation-based guarantees where task characteristics are not fully known in advance. Furthermore, no commercially available real-time operating system support exist for them. Our approach is to model existing systems, supported by commercial RTOSes, where task attributes are fully known at design time. However, [19] aims at modelling real situations through hierarchically modelling different schedulers. They cover preemptive and non-preemptive priority schedulers and do not model static schedulers. In fact, the work presented in this paper could extend their more general framework with the ability to model also static schedulers.

5 Case Study

A case study [10] conducted at Volvo Construction Equipment (VCE) [24], with the objective of finding a way to use available resources in a more efficient way has studied the design trade-offs between static and dynamic scheduling.

VCE has a tradition in statically scheduled systems. This is mainly due to the safety critical nature of their control systems in their heavy machinery, e.g., articulated haulers, trucks, wheel loaders and excavators. Rubus OS by Arcticus [1], used by VCE, has run-time support for the system model described in section 2.

Currently at VCE, all safety critical functionality is implemented in the static part and only soft real-time or non real-time activity resides in the dynamic part. In recent interviews (in an ongoing research project) they state that about 20-25% of their applications are considered safety critical, mainly residing in transmission and engine control. However, some operational modes, have static schedule utilization as high as 74%.

The demand on more functionality in next generation machinery is growing. However, the static schedule is getting close to full utilization, leaving little or no room for new functionality. This can either be addressed with new and more expensive hardware or to find a better way of utilizing the current hardware resources.

Demand on responsiveness (i.e. deadlines) for functionality in the static part ranges from a few milliseconds up to several seconds. This could potentially result in very large schedules (with corresponding high memory consumption). VCE's solution to this has been to fix the schedule length at 100ms, which result in waste of computing resources due to redundant polling for any function with a responsiveness demand higher than 100ms (even functions with responsiveness demand within 100ms but associated with events that occur seldom will in this case waste computing resources). A solution that could get rid of this redundant polling, while still guaranteeing the responsiveness and without increasing the schedule length, would be highly desirable.

5.1 An example system

Here we will present an example system that can be viewed as a simplified version of one of the systems constructed by VCE. A complete system would consists of several hundreds of tasks [10] and would be too complex to present in this paper. We will show how functions currently residing in the static part can be moved to the dynamic part and, by using the response-time analysis of [14], still guarantee that the function deadlines will be met. Type of functionality that could be moved, according to [10], consists of events that by nature are event-triggered, visual interaction with driver, and logging of operational statistics. Another example of functionality that may be moved to the dynamic part is control functionality that is not part of sampling or actuation. Control performance is often sensitive to jitter in sampling and actuation and therefore often placed in a static schedule [8]. However, the control calculation and updating of control state do not have these strict requirements on jitter and their responsiveness requirement is only restricted by the corresponding output action and sampling in the next period respectively. Therefore control and updating control state functionality could be moved to the dynamic part.

Task i	T_i	C_i	D_i	U^{100}	U^T
A	10	2	10	20%	20%
B	20	2	5	10%	10%
C	50	1	2	2%	2%
D	50	6	50	12%	12%
E	100	8	100	8%	8%
F	2000	7	100	7%	0.35%
G	2000	8	100	8%	0.4%
H	2000	8	2000	8%	0.4%

Table 1. The set of tasks in the Static system

For our example, the task specification in table 1 will be used. (For simplicity we will in this example ignore interrupt interference.) Tasks F and G handle events that may occur once every 2000ms, and with a response time requirement of 100ms. Placing tasks F and G in a static schedule, means that they would have to be polled at the rate of their deadline (100ms) instead of their period (2000ms) (since we do not know exactly when the events are going to occur). Task H, however, could be polled at the rate of its period (2000ms), how-

ever, the resulting schedule would become too large and memory consuming (it would have to extend for 2000ms and thus consume over 20kb of ROM). Setting the schedule length to 100ms would be adequate for all tasks except task H. Hence, the schedule length is set to 100ms, and a resulting schedule can be seen in figure 4.

In table 1 on the previous page, U^{100} represents the task utilization when scheduled in a static schedule with a period of 100ms, and U^T represents the utilization when tasks are scheduled with their period.

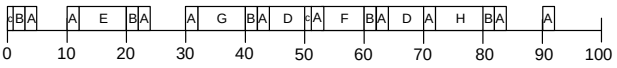

Figure 4. Static schedule for table 1 task set

The total utilization of the static schedule is 75%. Adding new functionality, requiring some kind of temporal guarantee, to this system can be difficult, there are not many free time-slots in the schedule, especially if there has to be room also for interrupts and non-real-time functionality.

Improving the system

However if tasks F, G, and H could be made event triggered, by placing them in the dynamic part of the Rubus OS, some resources could be freed. The resulting static schedule can be seen in figure 5. The utilization for the static schedule now becomes 52%. The utilization for the three dynamic tasks are 1,15%, resulting in a total utilization of just above 53%. Thus, by moving these three tasks from the static schedule we free nearly 22%[3] of the CPU resources.

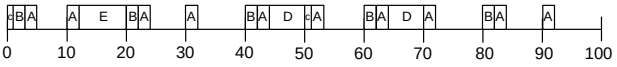

Figure 5. Schedule without tasks F, G and H

Now, it remains to see whether the three tasks will meet their deadlines when running as dynamic tasks. To be able to calculate response times for tasks F, G, and H we model the static schedule as a transaction with $T_s = 100$. WCETs and offsets are set as follows:

$$C_{sj} = (5,\ 10,\ \ 4,\ \ \ 2,\ 10,\ \ 3,\ 10,\ \ \ 2,\ \ 4,\ \ 2)$$
$$O_{sj} = (0,\ 10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90)$$

Assuming that F, G, and H have priorities high, medium, and low respectively, we can calculate the response times for the three tasks according to [14]. And the result is:

$$R_F = 26 \quad R_G = 44 \quad R_H = 64$$

We see that all three tasks will meet their deadlines of table 1. In fact, their responsiveness is considerably increased compared to being statically scheduled every 100ms. It could be mentioned that by removing tasks F, G and H from the schedule we have enabled shorter response times for other dynamic tasks, that might have existed in the system, as well. The schedule in figure 4 has a longest busy period of 54ms (between 30–84), whereas the new schedule in figure 5 has a longest busy period of 14ms (between 10–24). Since any dynamic task (in the worst case) will have to wait for the longest busy period, we now have significantly reduced that time.

With the approach presented in this paper the static schedule could be kept small (with respect to memory consumption as well as utilization). By modelling the static schedule as one transaction, response time analysis for task with offsets can be used to evaluate timeliness for the dynamic part.

Our solution reduce utilization by moving functionality, previously polled excessively, from the static schedule to the dynamic part. Our method also gives a possibility to shrink the static schedule since functions with long periods can be moved from the static schedule. It should be mentioned however, that all tasks in the static schedule share a common stack, whereas moving tasks from the schedule to the dynamic part may require them to have separate stacks, hence increasing the memory consumption for dynamic tasks. However, using a resource locking protocol such as the immediate inheritance allows also dynamic tasks to share a single stack [6, 16].

The possibility to selectively migrate functions from static scheduled legacy systems to dynamic scheduled systems will substantially facilitate for companies to gradually move into the area of dynamic scheduling, and thus, in the long run, help companies to use cheaper hardware for, or fit more functions into, their products. Also the development process becomes easier because event triggered functionality does not have to be force-fitted into a static model.

6 Conclusions

As stated in [15] analyzability is one of the major concern for embedded systems development. We have in this paper shown how a hybrid, static and dynamic, scheduling model can be modelled and dynamic tasks analyzed for responsiveness. The type of system presented can be realized by commercially available OS

[3] Increase in overhead for tasks F, G, and H as dynamic tasks will be marginal, hence not considered here.

support, e.g., Rubus OS by Arcticus [1]. In fact, any fixed priority OS complemented with an external static scheduler can implement this type of system with the static schedule as a task at highest priority.

A hybrid, static and dynamic, scheduling model simplifies the design trade-offs of which scheduling model to choose. Appropriate scheduling model can be chosen on function level instead of system level. Since temporal guarantees can be provided, this approach will also be applicable for hard real-time systems. Choosing the most appropriate model for each function, instead of force-fitting it to an overall model, not only simplifies the design choices but also gives the possibility to save system resources and improve responsiveness. This is demonstrated in a case study [10] at Volvo Construction Equipment using the commercial real-time operating system Rubus by Arcticus [1].

References

[1] Arcticus Systems Web-Page. http://www.arcticus.se.

[2] The Asterix Real-Time Kernel. http://www.mrtc.-mdh.se/projects/asterix/.

[3] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2/3):173–198, 1995.

[4] S. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time, and Non-Real-Time Processes. In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2003.

[5] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[6] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9994-3.

[7] Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communications, February 1992. ISO/DIS 11898.

[8] A. Cervin. Improved scheduling of control tasks. In *Proc. of the 11^{th} Euromicro Workshop of Real-Time Systems*, pages 4 – 10, June 1999.

[9] FlexRay Home Page. http://www.flexray-group.org/.

[10] K. Hänninen and T. Riutta. Optimal Design. Master's thesis, Mälardalens Högskola, Dept of Computer Science and Engineering, 2003.

[11] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[12] H. Kopetz and G. Grünsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, pages 14–23, January 1994.

[13] C. Locke. Software Architecture For Hard Real-Time Applications - Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems*, 4:37–53, 1992.

[14] J. Mäki-Turja and M. Nolin. Tighter Response-Times for Tasks with Offsets. In *Proc. of the 10^{th} International conference on Real-Time Computing Systems and Applications (RTCSA'04)*, August 2004.

[15] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *7th International Symposium on Component-based Software Engineering (CBSE7)*. IEEE Computer Society, May 2004.

[16] Northern Real-Time Applications. SSX5 True RTOS, 1999.

[17] D. Nyström, M. Nolin, A. Tesanovic, C. Norström, and J. Hansson. Pessimistic Concurrency-Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proc. of the 16^{th} Euromicro Conference on Real-Time Systems*, June 2004.

[18] J. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS)*, December 1998.

[19] J. Regher, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2003.

[20] J. Regher and J. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. 22^{th} IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2001.

[21] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein. Analysis of hierarchical fixed priority scheduling. In *Proc. of the 14^{th} Euromicro Conference on Real-Time Systems*. IEEE Computer Society, June 2002.

[22] K. Sandström, C. Eriksson, and G. Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In *Proc. of the 5^{th} International conference on Real-Time Computing Systems and Applications (RTCSA'98)*, 1998.

[23] K. Tindell. Using Offset Information to Analyse Static Priority Pre-emptively Scheduled Task Sets. Technical Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[24] Volvo Construction Equipment. http://www.volvoce.-com.

[25] J. Xu and D. Parnas. Priority Scheduling Versus Pre-Run-Time Scheduling. *The Journal of Real-Time Systems*, 18(1):7–23, January 2000.