Combining Dynamic and Static Scheduling in Hard Real-Time Systems

Jukka Mäki-Turja jukka.maki-turja@mdh.se

Mikael Sjödin mikael.sjodin@mdh.se

October 2002

MRTC Report no. 71

ISSN 1404-3041 ISRN MDH-MRTC--71--SE

Combining Dynamic and Static Scheduling in Hard Real-Time Systems

Jukka Mäki-Turja Mikael Sjödin
Mälardalen Real-Time Research Centre

Technical Report: MRTC–71

Contact author:

Mälardalen Real-Time Research Centre
Jukka Mäki-Turja
Box 883
SE–721 23 Västerås
Sweden

email: `jukka.maki-turja@mdh.se`
phone: +46 21 101 466

Abstact

We present a method to calculate response times for dynamically scheduled tasks that execute "in the background" of a static cyclic scheduled functions. The method will allow hard real-time systems designers to use both static cyclic scheduling and dynamic (fixed priority) scheduling within one system, thus significantly simplifying the design tradeoff of which scheduling paradigm to use.

In a case study we show how the method can be used to guarantee timeliness of functions moved from a static schedule to dynamic scheduled tasks, thus saving computing resources and allowing more functions to be added to the system.

[0]This work has been partially supported by Volvo Technical Development within the EAST/EEA project.

1 Introduction

This paper presents a method to calculate response times for dynamically scheduled tasks that are executed "in the background" of a static cyclic schedule. The method can be used to migrating functions from a static cyclic schedule (where functions sometimes consume excessive resources) to dynamically scheduled tasks, while still guaranteeing the timeliness of these functions.

In real-time systems, scheduling of CPUs and communications media can be classified into two categories:

- 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. 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 attributes (such as priority and deadline) are 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.

Both approaches have their virtues, and deciding which approach to use always includes a lot of tradeoffs. Some typical properties of static scheduling are:

- It is easy to verify that timing requirements are met.
- Synchronisation between tasks is resolved off-line, and hence poses no run-time overhead.
- It provides a high degree of determinism (which makes it easy to reproduce executions during testing).
- It is difficult/time-consuming to construct a schedule (especially for distributed systems where a set of synchronised schedules have to be constructed).
- Unless care is taken at design time, there is a high risk that the size of the schedules becomes unreasonable large.
- Once a schedule have been found, it may be very difficult to add new tasks to the schedule.
- Limited possibilities to respond to dynamic events in the environment.
- Resources may be wasted on polling and redundant computing due to limitations when constructing a schedule. (For instance, the schedule length,

often in the order of some 10-100ms, gives the longest possible time between two pollings of an event that may only occur once every few seconds.)

Some typical properties of dynamic scheduling are:

- Finding the scheduling attributes to ensure that timing requirements are met may be difficult.
- To asses that timing requirements will be met in all circumstances is not trivial. However, a large set of methods does exists, [ABD$^+$95, SSNB95].
- It is easy to add new tasks to a system without having to rebuild a schedule.
- Handling of dynamic events is easy.
- No resources have to be wasted on redundant polling of the environment.

Since both scheduling paradigms have both pros and cons, making the decision of which to use is non trivial. Indeed, within a system, some sub-systems may be best suited for implementation using one paradigm, while other sub-systems are best suited for implementation using the other paradigm.

This conflict is clearly illustrated by the last few years development of field busses for automotive applications. The Controller Area Network (CAN) [CAN92] has the last years 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 presented the Time Triggered Protocol (TTP) [KG94, TTT], which provides static scheduling, many automotive manufacturers and their sub-contractors embraced the new technology.

However, it was soon recognised that TTP was a bit *too* static. Hence, a consortium of automotive manufacturers and sub-contractors started the development of FlexRay [Flx], which provides both static and dynamic scheduling. On the other hand, efforts to make CAN less dynamic (and hence more predictable) has been made. For instance, Time-Triggered CAN (TT-CAN) [FMD$^+$00] gives a method to apply static cyclic scheduling on standard CAN equipment. Also, in the case of TT-CAN, the need to incorporate dynamic scheduled messages was recognised. Hence, the Flexible Time-Triggered CAN (FTT-CAN) [AFF99, PA00] has been proposed as a method to combine the both scheduling paradigms.

Also, on the operating-system side, products that support both static and dynamic scheduling have emerged. For instance, Arcticus Systems' [Arca] operating system Rubus [Arcb], and the free real-time operating system Asterix [Ast].

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. [Flx, Arcb]. That is, the dynamic scheduled tasks/messages are not given any response-time guarantees, only a best-effort service is provided. However, in order to fully utilise the potential of combining static and dynamic scheduling in real-time systems, both the dynamic and the static parts need to be able to provide response-time guarantees. The ability to guarantee responsiveness to dynamically scheduled tasks/messages is crucial to allow migration of functions from (the expensive, resource demanding) static scheduled subsystems into (the more flexible and light-weight) dynamically scheduled subsystems.

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.

The main contributions of this paper are:

- We develop a novel task model that is able to model static schedules and dynamically scheduled tasks, and
- we present a corresponding response-time formula for calculating response time guarantees for tasks in our model.
- In a case study, we illustrate the industrial motivation for, and the benefits of, combining static and dynamic scheduling strategies in the same system.

Paper Outline: Next, in section 2 we formally describe the problem studied in the paper. In section 3 we present our novel task model and response-time analysis, and in section 4 we illustrate how the analysis can be used to migrate functions from a static schedule, in order to free up system resources. Finally, in section 5 we present our conclusions and outline future work.

2 Problem Formulation

In this paper we address the problem of giving response-time guarantees to dynamically scheduled tasks when those tasks are scheduled "in the background" of statically scheduled tasks. That is, we assume that the statically scheduled task will take priority over dynamically scheduled tasks. However, we will allow preemption of the statically scheduled tasks by interrupts.

The system model contains:

- Interrupts. There may be multiple interrupt levels, so an interrupt may be interrupted by a higher level interrupt.
- A static cyclic schedule:

  - The schedule has a length (a duration) that is equal to the LCM (least common multiple) of all function periods that are statically scheduled. The schedule is constructed off-line by some scheduling tool.
  - A set of functions are scheduled in the schedule. Each function has a known worst case execution time (WCET).
  - Each function is scheduled at an offset relative to the start of the schedule.[1] This is also referred to as a function's *release time*.
  - The static cyclic scheduler executes 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 the static scheduled functions.

  It should be noted that many scheduling tools can coalesce set of functions into function chains (e.g. the Rubus scheduler [Arcb]). In a function chain all functions are executed "back-to-back" and only the start of the function chain is assigned a release time in the schedule. For the purpose of this paper we will treat a function chain as a single function with WCET equal to the sum of the WCETs of each function in the function chain.

  In this paper we assume that a static cyclic schedule has been constructed prior to the analysis of dynamic tasks is performed. 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 [SEF98].

- A set of tasks that are dynamically dispatched. We call each such task a *dynamic task*. These tasks executes in the time slots available between interrupts and static scheduled functions. Dynamic tasks are scheduled by a fixed priority scheduler. Tasks are assumed to be periodic or, at least, to have a known minimum time between two invocations. (Section 3.1.1 explains the task model in greater detail.)

  In this paper we will present a method to calculate the worst case response-time for each dynamic task.

Traditionally, in this kind of system the dynamic tasks are assigned to non time-critical functions. The time-critical functions are all allocated in the static schedule. The reason for this partitioning has been that no method to calculate the response-time for dynamic tasks has existed.

[1]Some static cyclic schedulers allow the functions to overlap in time, i.e. one function may have a WCET that is longer than the time between its offset and its succeeding function. In those cases the scheduler may choose either to preempt the first function and run the second function at its scheduled time, or it may place the second function in a queue and execute it when the first function completes. The methods presented in this paper are applicable regardless if overlap is allowed and regardless if preemption or queueing is used.

3 Calculating the Response Time

In this section we will present our novel method to calculate response times for dynamic tasks. We will, for clarity, do this in three steps: first we will reformulate the classical response-time analysis. Second, we show how to calculate the response times when a particular form of static schedule is used (i.e. a schedule using the major/minor cycle paradigm). Finally, we present a response-time analysis for dynamical task running in the background of any static cyclic schedule.

3.1 Recapitulating Response-Time Analysis

Classical response-time analysis uses an extended version of the original Liu and Layland task model [LL73], that we describe below. Unlike Liu and Layland's rate monotonic approach, where a utilisation bound is used, response-time analysis provides a both necessary and sufficient schedulability test.

3.1.1 Task Model

In the classical response-time analysis (see e.g. [BW96, ABD+95]) we assume a fixed priority scheduler (i.e. a dynamic scheduler that always executes the highest priority eligible task).

Each task i is assumed to be a periodic task with the following attributes:

C_i The Worst-Case Execution-Time (WCET) of the task. Engblom gives an overview of methods to calculate the WCET [Eng02].

T_i The period of the task. Tasks are assumed to be periodic or, at least, to have a known minimum time between two consecutive invocations.

B_i The maximum blocking time (i.e. the maximum time to wait for a lower priority task that has locked a resource). In order to calculate the blocking time for a task, usually, a resource locking protocol like priority ceiling or immediate inheritance is needed. Buttazzo presents algorithms to calculate blocking times for different resource locking protocols [But97].

J_i The maximum jitter (i.e. max deviation from ideal periodicity).

D_i The deadline of the task.

Additional attributes for tasks that will be used in this paper are:

P_i The priority of the task. Priorities can be assigned with any method (e.g. rate monotonic or deadline monotonic). If a task i has higher priority than a task j, then $P_i > P_j$.

R_i The worst-case response-time (as derived by the response-time analysis, see section 3.1.2).

In this paper it assumed that:

- $C_i > 0$, $T_i > 0$, $D_i > 0$.
- $B_i \geq 0$, $J_i \geq 0$.
- $P_i \neq P_j$ if $i \neq j$ (i.e. unique task priorities).
- $D_i < T_i - J_i$ (i.e. the deadline is less that the time between two consecutive invocations of a task).

The two last assumption above could be removed using elsewhere published techniques (see e.g. [Tin94, AKA94, ABD+95]). The techniques presented later in this paper could be applied also without these assumptions. However, since these assumptions significantly simplifies the response-time equations we will, for pedagogical reasons, keep them throughout this paper.

A set of tasks is said to be schedulable if $R_i \leq D_i$ for each task i. Hence, the problem to decide if a set of tasks is schedulable is reduced to calculating the response-time, R_i, for each task.

3.1.2 Response-Time Equations

For the model above, the formula for calculating the response-time for a task i is [ABD+95]:

$$\begin{aligned} R_i &= B_i + C_i + \sum_{j \in \{x : P_x > P_i\}} execution\_demand(j, R_i) \\ execution\_demand(j,t) &= occurrences(j,t) * C_j \\ occurrences(j,t) &= \left\lceil \frac{t + J_j}{T_j} \right\rceil \end{aligned} \quad (1)$$

Where $execution\_demand(j,t)$ denotes the maximum execution demand task j can generate in an interval t, and $occurrences(j,t)$ denotes the maximum number of times a task j can arrive during an interval t.

However, since R_i cannot be isolated on one side of the equality the following iterative solution method is used:

$$R_i^{n+1} = B_i + C_i + \sum_{j \in \{x : P_x > P_i\}} execution\_demand(j, R_i^n)$$

where $R_i^0 = 0$ and $R_i = R_i^n$ when $R_i^n = R_i^{n+1}$.

3.2 Simple Static Cyclic Schedules

In this subsection we consider a restricted variant of static cyclic schedules. Here we assume that the whole schedule, called the *major cycle*, is divided into a number of equal sized sections, called *minor cycles*. Functions are only scheduled at the start of minor cycles. In the case where more than one function is to be executed within a minor cycle, these functions are coalesced into a single function chain, that is scheduled for the start of the minor cycle. Figure 1 shows a schedule with major cycle of 20 time units and a minor cycle of 5 time units.

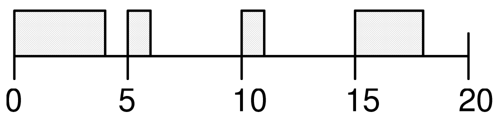


Figure 1: Example of major/minor cycle schedule

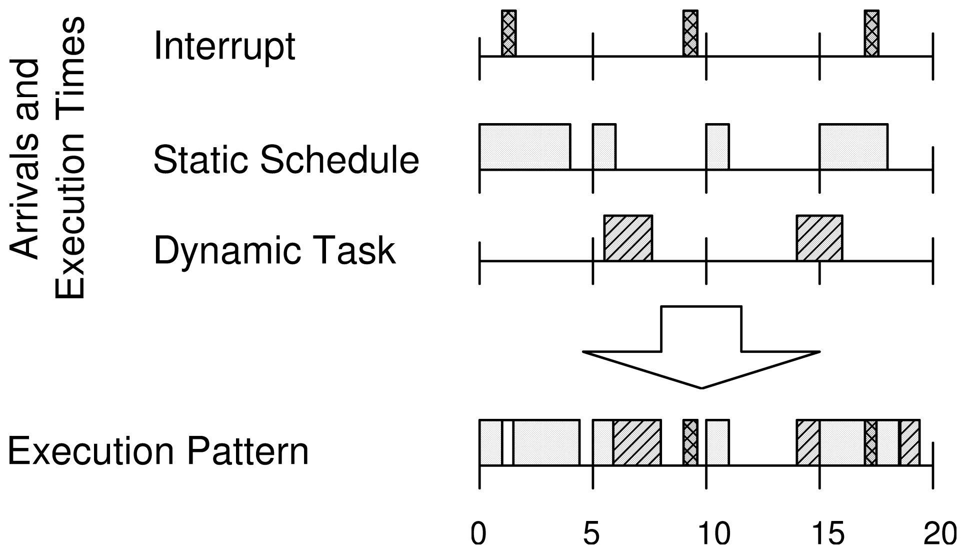


Figure 2: Example execution scenario

In figure 2 we see an example of how the execution could look like when executing the schedule from figure 1, with one interrupt source and one dynamic scheduled task. We make the observation that both interrupts and the static schedule act like higher priority tasks from the dynamic task's point of view.

We could use the response-time equation in section 3.1.2 to calculate the interference caused by the static schedule on the dynamic task. A safe, but overly pessimistic, way would be to make the static schedule a high priority task with attributes $C_i = 4$ (where 4 is the maximum time allocated for any function in the schedule) and $T_i = 5$ (where 5 is the minor cycle time). Then we could use the equation in section 3.1.2 to calculate response-times for each dynamic task. However, this would be overly pessimistic, and in essence it would allocate 4/5 (80%) of the CPU to the static schedule. In this example we can however see that only 9/20[2] (45%) of the CPU is allocated to the static schedule. Another, less pessimistic, approach would be to model the schedule with four tasks, with execution times of 4, 1, 1, and 3 respectively. All tasks having a period of 20. This approach is still too pessimistic since it assumes that all four task can be released for execution at the same time. In this example, a dynamic task with $C_i = 1$, would have a response time of 10. Looking at figure 1 one can see that the worst possible case would result in a response time of 5 (if the dynamic tasks is released at time 0).

Hence, our goal is to model static schedules so as to incur as little pessimism as possible. Thus, modelling both functions' WCETs as well as their release times as accurately as possible

3.2.1 Extending the Task Model

Our solution is to modify the model of section 3.1.1. We will use a technique that is similar to the technique used in [SH99]. Lets change the attribute C_i to denote a *vector* of execution times, such that:

C_i $= [C_i[0], C_i[1], \ldots, C_i[|C_i| - 1]]$
where a $C_i[k]$ denotes an execution time.

$|C_i|$ is the number of elements in C_i (i.e. the number of functions in the schedule).

T_i is the duration of the minor cycle.

A (traditional) dynamic task (that always has the same WCET) would have $|C_i| = 1$, and T_i would be set to the period. Thus, if $|C_i| = 1$ then this new model is equivalent to the classical model. Hence, we can use this model to represent both static cyclic schedules and dynamic tasks.

The elements of C_i denotes a cyclic pattern of execution times. Where each execution time would be the execution time of a function chain. In our example from figure 1 on the page before we would get:

- $C_i = [4, 1, 1, 3]$
- $|C_i| = 4$
- $T_i = 5$

Note that, we place no restrictions on where in the cyclic pattern the first element of C_i is. Consequently, $[1, 1, 3, 4]$ is (in this model) an equivalent definition of C_i.

[2] I.e., 4+1+1+3 units of execution in a schedule of length 20.

3.2.2 Extending the Response-Time Analysis

Now, when we have a task model that accurately captures the behaviour of a major/minor-cycle static schedule (with respect to its interference on dynamic tasks), how do we calculate the response times for our new task model?

We begin by introducing the infinite-length array $\hat{C}_i[k]$, where $\hat{C}_i[k]$ is the maximum total execution time of k successive invocations of task i. The formal definition is:

$$\hat{C}_i[k] = \begin{cases} 0 & \text{if } k = 0 \\ \max\limits_{t \in 0 \ldots |C_i|-1} \sum\limits_{l=0}^{k-1} C_i[(t+l) \mod |C_i|] & \text{if } k > 0 \end{cases} \tag{2}$$

Note specifically that $\hat{C}_i[1]$ is the maximum of the execution times in C_i, and $\hat{C}_i[|C_i|]$ is the sum of execution times of all elements in C_i. Also, if $|C_i| = 1$ then $\hat{C}_i[k] = C_i[0] * k$. For our example above, we would get $\hat{C}_i = [0, 4, 7, 8, 9, 13, 16, 17, \ldots]$.

This leads us to the final formulation of the extended response-time analysis. Using $\hat{C}_i$ we can now change *execution_ demand*(j,t) in equation 1 and get:

$$\begin{aligned} R_i &= B_i + \hat{C}_i[1] + \sum_{j \in \{x : P_x > P_i\}} \mathit{execution\_\ demand}(j, R_i) \\ \mathit{execution\_\ demand}(j,t) &= \hat{C}_j[\mathit{occurrences}(j, R_i)] \\ \mathit{occurrences}(j,t) &= \left\lceil \frac{t + J_j}{T_j} \right\rceil \end{aligned} \tag{3}$$

3.2.3 Calculating $\hat{C}_i$

An efficient implementation of equation 2 can be made by pre-computing and storing the first $|C_i| + 1$ elements of $\hat{C}_i$. The pre-computed array, denoted C_i^{pre}, is defined as:

$$C_i^{\text{pre}}[k] = \hat{C}_i[k] \qquad \forall k \in 0 \ldots |C_i|$$

Lets call a sequence of $|C_i|$ task executions a *full execution*. A full execution will contain one execution of each instance in C_i, i.e., a full execution will take $\sum_{k \in 0..|C_i|-1} C_i[k]$ time. As pointed out above, $\sum_{k \in 0..|C_i|-1} C_i[k] = \hat{C}_i[|C_i|]$ (which is stored in $C_i^{\text{pre}}[|C_i|]$). Now we can implement equation 2 as:

$\hat{C}_i[k] = no\_of\_full\_executions * C_i^{\text{pre}}[|C_i|] + C_i^{\text{pre}}[no\_of\_remaining\_executions]$
$no\_of\_full\_executions = k \text{ div } |C_i|$
$no\_of\_remaining\_executions = k \text{ rem } |C_i|$

Where a div b gives the integer part of the division a/b, and a rem b gives the reminder. Also, note that k will be $occurrences(j, t)$ in equation 3.

3.2.4 Related Work

The task model and response-time analysis presented in sections 3.2.1 and 3.2.2 are inspired from our previous work on scheduling analysis of compressed multimedia traffic in ATM networks [SH99], and is also presented in [Sjö02].

Also, Mok and Chen has presented an analysis technique that could be applied to static cyclic schedules with major and minor cycles [MC96]. However, their method would not be applicable to an arbitrary major/minor cycle schedule, since it requires a task to have a property called "accumulative monotonic". In essence, this means that the worst possible sum of consecutive execution times must always start with the same element (typically the first) in C_i.

Since that property is not considered by any (to our knowledge) known static scheduling algorithm it is not guaranteed that the generated schedules would exhibit the accumulative monotonic property. Hence, it is highly unlikely that Mok and Chen's method could be of any real use for the kind of systems considered in this paper. Also, Mok and Chen's method only consider the two first functions in a schedule, hence their method is highly pessimistic.

Furthermore Mok and Chen calculates a utilisation bound whereas the method presented here calculates response times for individual tasks. Thus, our work relates to Mok and Chen's in the same way classic RTA [ABD$^+$95] relates to Liu and Layland's original work [LL73].

In section 3.3.4 we will consider methods that has more general applicability than just to model major/minor-cycle schedules.

3.3 General Static Cyclic Schedules

In this section we will consider static cyclic schedules with arbitrary release times. Thus, we lift the restriction from section 3.2 that a schedule should be segmented into minor cycles. Previously tasks where scheduled for execution at the beginning of each minor cycle. In this section tasks can be released for execution at any point in time in the schedule.

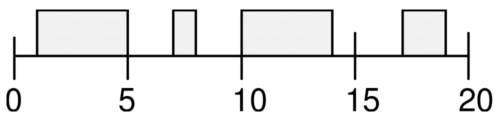


Figure 3: Example of static cyclic schedule

Figure 3 shows a static cyclic schedule of length 20, with 4 functions released at times 1, 7, 10 and 17, with WCETs 4, 1, 4 and 2 respectively.

3.3.1 Extending the Task Model

In this section we will extend the model of section 3.1.1 and model each task as a complete schedule. Dynamic tasks are viewed as a schedules with only one function, and the schedule's duration is equal to the task's period. Lets define a schedule using a list of pairs stored in an array, S_i, as follows:

$$\begin{aligned} S_i =& [\ldots,\ \{C_i[k],\ \tau_i[k]\},\ \ldots] \text{ for } k \in 0 \ldots |S_i| - 1 \\ & \text{where each } C_i[k] \text{ is the WCET of a function and} \\ & \tau_i[k] \text{ is the corresponding release time for that function.} \\ |S_i| =& \text{number of WCET+release time pairs in } S_i \\ T_i =& \text{the duration of schedule.} \end{aligned}$$

For the schedule in figure 3 we would get the following model:

$$\begin{aligned} S_i &= [\{4,1\},\ \{1,7\},\ \{4,10\},\ \{2,17\}] \\ |S_i| &= 4 \\ T_i &= 20 \end{aligned}$$

3.3.2 Extending the Response-Time Analysis

In order to calculate the response-times for tasks described by our new task model we need to redefine $execution\_demand(j,t)$ from equation 1 on page 6. $execution\_demand(j,t)$ should return the maximum execution demand of task j that can accumulate during and interval of time t.

The run-time situation that causes the highest possible execution demand is when one of the functions in the tasks schedule is released at the beginning of the interval (i.e. at time 0). This is analogous with Liu and Layland's critical instant definition [LL73]. Thus, we have a finite number of possible starting points for the worst case scenario; one possibility for each function in the schedule.

This means that the accumulated execution demand of a task i can only increase at limited number of points in time. That is, as the size of the time-interval t increases, the accumulated execution demand calculated by *execution_ demand*(i, t) can only increase at discrete points in time.

Now, we know that the worst case execution demand occurs when one of the functions is released at the start of the interval. In our example, the schedule has four functions (and thus four release times), hence we get four possible candidates for the worst case scenario. Figure 4 depicts how the execution demands accumulates over time for each of the four cases.

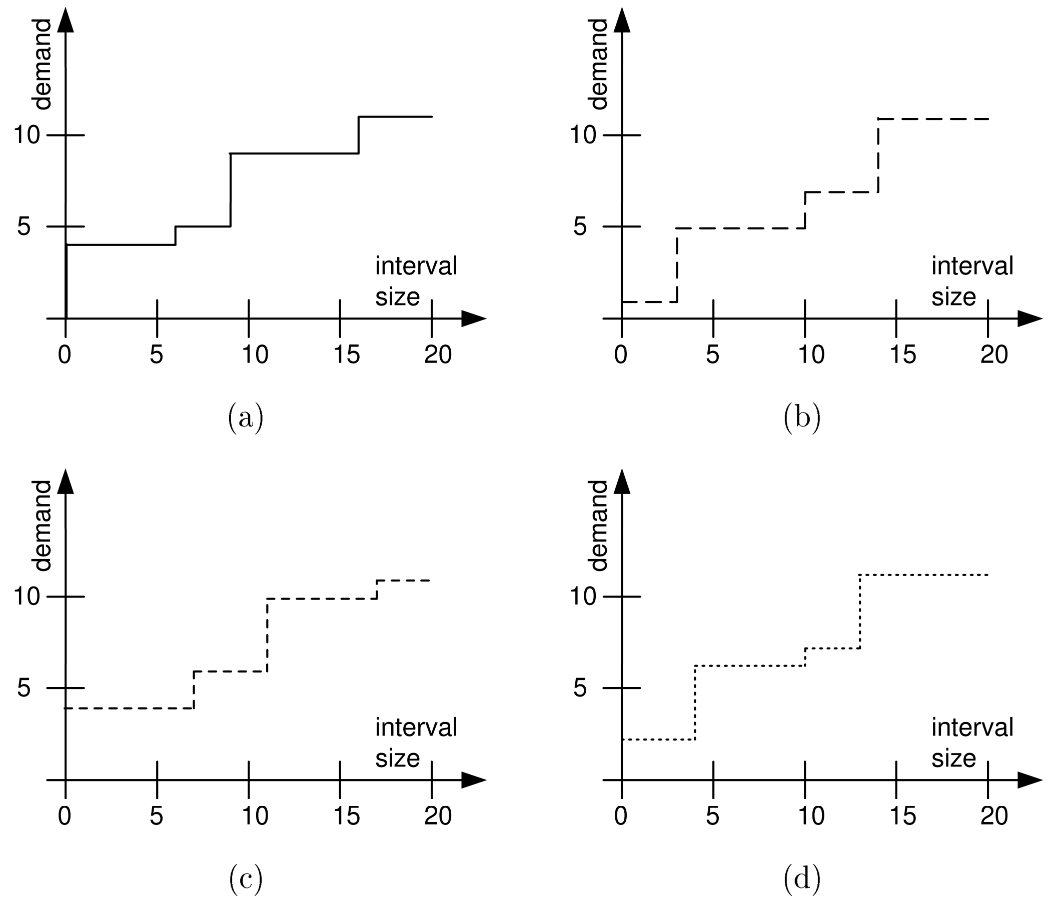
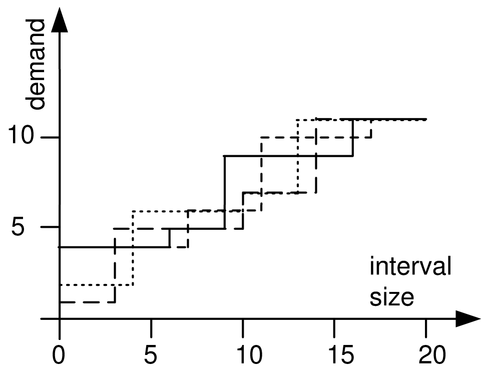


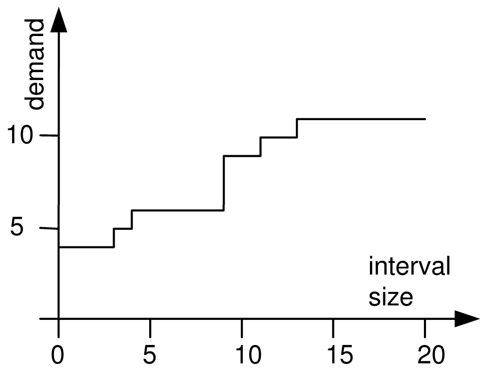
Figure 4: The four candidate worst case scenarios

Figure 5 on the following page shows figures 4(a)–4(d) combined in a single graph. Here it can been seen that each one of the four cases of figure 4 will, for some interval length, have the highest accumulated execution demand. Hence, we draw the conclusion that we cannot settle for considering any single one case. Note that, at the end of the schedule (i.e. at time 20) all four cases has converged to the same execution demand, and that after time 20 the increase in execution demand will follow the exact same curve as it did the first 20 time units.

Here we will reuse the trick of section 3.2.3 and precompute an array that



(a) All scenarios



(b) Max of the four curves in (a)

Figure 5: The scenarios of figure 4 combined

will help us to calculate *execution_ demand*(j, t). Lets define S_i^{pre} as an array representing the maximum of the execution demands for each of the candidates. S_i^{pre} stores the accumulated execution demand for the first T_i time units. (Visually, S_i^{pre} will represent figure 5(b)). S_i^{pre} is an array of pairs:

$$S_i^{\text{pre}} = [\ldots, \{C_i^{\text{pre}}[k], \tau_i^{\text{pre}}[k]\}, \ldots] \text{ for } k \in 0 \ldots |S_i^{\text{pre}}| - 1$$
$$|S_i^{\text{pre}}| \text{ is the number of pairs in } S_i^{\text{pre}}$$

Where $\tau_i^{\text{pre}}[k]$ represents a length of an interval and the corresponding $C_i^{\text{pre}}[k]$ represents the total execution released for execution within that interval. In section 3.3.3 we will return to how to calculate the elements of S_i^{pre} and its size $|S_i^{\text{pre}}|$. For now we will settle to show how S_i^{pre} will look for our example from figure 3:

$$S_i^{\text{pre}} = [\{4,0\}, \{5,3\}, \{6,4\}, \{9,9\}, \{10,11\}, \{11,13\}]$$
$$|S_i^{\text{pre}}| = 6$$

The interpretation of the above is that for an interval, t, of size $0 < t \leq 3$ (upper limit given by $\tau_i^{\text{pre}}[1]$) a maximum of 4 time units of execution has been released. For interval sizes $3 < t \leq 4$, the execution demand is 5, and so on. Note that our example S_i^{pre} corresponds to the execution demand shown in figure 5(b).

Note particularly that $C_i^{\text{pre}}[0]$ (in the first pair of S_i^{pre}) is the max of all $C_i[k]$-s and that $C_i^{\text{pre}}[|S_i^{\text{pre}}| - 1]$ (in the last pair of S_i^{pre}) is the sum of all $C_i[k]$-s. Also, note that each pair in S_i^{pre} does not correspond to the same starting point in S_i. In this example, the first element in S_i^{pre} corresponds either to the scenario shown in figure 4(a) or in figure 4(c), whereas the second element corresponds the scenario shown in figure 4(b).

Given S_i^{pre} we can calculate the accumulated execution demand for a task i in any interval t, and hence we now have the following response-time equation:

$$
\begin{aligned}
&R_i = B_i + \underbrace{C_i^{\text{pre}}[0]}_{\text{Max of } C_i} + \sum_{j \in \{x : P_x > P_i\}} \mathit{execution\_demand}(j, R_i) \\
&\quad \mathit{execution\_demand}(j,t) = \\
&\quad\quad \mathit{no\_of\_full\_schedules} * \underbrace{C_j^{\text{pre}}\left[|S_j^{\text{pre}}| - 1\right]}_{\text{Sum of } C_i} + \mathit{partial\_schedule} \\
&\quad \mathit{no\_of\_full\_schedules} = (t + J_j) \text{ div } T_j \\
&\quad \mathit{remaining\_time} = (t + J_j) \text{ rem } T_j \\
&\quad \mathit{partial\_schedule} = \begin{cases} 0 & \text{if } \mathit{remaining\_time} = 0 \\ C_j^{\text{pre}}[k] & \text{if } \mathit{remaining\_time} > 0 \end{cases} \\
&\quad k = \max\{x \ : \ \mathit{remaining\_time} > \tau_j^{\text{pre}}[x]\}
\end{aligned}
\tag{4}
$$

Where *no_ of_full_ schedules* denotes the number of complete executions of the schedule (accounting for jitter) and *remaining_ time* is the remaining time of t after *no_ of_full_ schedules* schedule executions. k gives the index into C_i^{pre} that tells how much execution demand that can have accumulated during the last *remaining_ time* time units.

3.3.3 Calculating S^{pre}

The array S_i^{pre} is calculated by considering each possible start of the schedule. This means that we will consider $|S_i|$ cases. We will calculate how the execution demand will accumulate for each candidate worst-case scenario. We then take the maximum of these execution demands. Intuitively, we will calculate the graph shown in figure 5(b) based on the graphs in figures 4(a)–4(d).

In order to do this we will, for each index $n \in 0 \ldots |S_i| - 1$, calculate an array $S_{i,n}^{\text{pre}}$ of size $|S_i|$. $S_{i,n}^{\text{pre}}$ will contain the accumulated execution demand for intervals up to length T_i, assuming that function n is released at start of the interval. Each element, k, in $S_{i,n}^{\text{pre}}$ is a pair of the form $\{C_{i,n}^{\text{pre}}[k],\ \tau_{i,n}^{\text{pre}}[k]\}$, where $\tau_{i,n}^{\text{pre}}[k]$ represents the length of an interval and the corresponding $C_{i,n}^{\text{pre}}[k]$ represents the accumulated execution demand within that interval. $S_{i,n}^{\text{pre}}$ is defined as follows:

1. S_i^{pre} = Append all $S_{i,n}^{\text{pre}}$ arrays.
2. Sort S_i^{pre} on ascending τ_i^{pre}-values. (Now, we have all possible execution demands listed in the order in which they may occur.)
3. If multiple pairs in S_i^{pre} have the same τ_i^{pre}-value discard all duplicate pairs except the pair with the highest C_i^{pre}. (The intuition for this is that if a time can have more than one execution demand, only the highest demand corresponds to the worst case situation.)
4. Discard each pair k (for $k \geq 1$) in S_i^{pre} where $C_i^{\text{pre}}[k-1] \geq C_i^{\text{pre}}[k]$. (The intuition for this is that if we have a pair where the execution demand decreases, that demand does not correspond to the worst case situation.)

Figure 6: Algorithm to calculate S_i^{pre} from set of $S_{i,n}^{\text{pre}}$

$$S_{i,n}^{\text{pre}} = \left[\ldots, \left\{C_{i,n}^{\text{pre}}[k],\ \tau_{i,n}^{\text{pre}}[k]\right\},\ \ldots\right] \quad \text{for } k \in 0 \ldots |S_i| - 1$$

$$C_{i,n}^{\text{pre}}[k] = \sum_{l=0}^{k} C_i\left[(l+n) \mod |S_i|\right]$$
$$\tau_{i,n}^{\text{pre}}[k] = (\tau_i[k] - \tau_i[n]) \mod T_i$$

For our example from figure 3 we would get the following four $S_{i,n}^{\text{pre}}$ (each $S_{i,n}^{\text{pre}}$ corresponds to one graph in figure 4):

$$\begin{array}{lll}
S_{i,0}^{\text{pre}} & = [\{4,0\}, \{5,6\}, \{9,9\}, \{11,16\}] & \text{(Figure 4(a))} \\
S_{i,1}^{\text{pre}} & = [\{1,0\}, \{5,3\}, \{7,10\}, \{11,14\}] & \text{(Figure 4(b))} \\
S_{i,2}^{\text{pre}} & = [\{4,0\}, \{6,7\}, \{10,11\}, \{11,17\}] & \text{(Figure 4(c))} \\
S_{i,3}^{\text{pre}} & = [\{2,0\}, \{6,4\}, \{7,10\}, \{11,13\}] & \text{(Figure 4(d))}
\end{array}$$

From the set of $S_{i,n}^{\text{pre}}$-s we now have to select the pairs that corresponds to the worst possible case in each situation. The algorithm to perform this selection is shown in figure 6. Intuitively, the algorithm merges all $S_{i,n}^{\text{pre}}$-s to get an array that corresponds to figure 5(a). From that array redundant information is thrown away to get an array that represents figure 5(b). The result of the algorithm is the S_i^{pre} array to use in equation 4 on the preceding page.

Again, looking at our example, we would get the following result from each step in the algorithm in figure 6:

1. S_i^{pre} =[{4, 0}, {5, 6}, {9, 9}, {11, 16}, {1, 0}, {5, 3}, {7, 10}, {11, 14}, {4, 0}, {6, 7}, {10, 11}, {11, 17}, {2, 0}, {6, 4}, {7, 10}, {11, 13}]
2. S_i^{pre} =[{4, 0}, {1, 0}, {4, 0}, {2, 0}, {5, 3}, {6, 4}, {5, 6}, {6, 7}, {9, 9}, {7, 10}, {7, 10}, {10, 11}, {11, 13}, {11, 14}, {11, 16}, {11, 17}]
3. S_i^{pre} =[{4, 0}, {5, 3}, {6, 4}, {5, 6}, {6, 7}, {9, 9}, {7, 10}, {10, 11}, {11, 13}, {11, 14}, {11, 16}, {11, 17}]
4. S_i^{pre} =[{4, 0}, {5, 3}, {6, 4}, {9, 9}, {10, 11}, {11, 13}]

It should be noted the by coalescing each individual candidate worst case scenario into a single worst case description (i.e. by merging the $S_{i,n}^{\text{pre}}$-s into S_i^{pre}) we create an overestimation of the worst case execution demand. This, can be seen in our example: In figure 5(b) it looks like execution demand increases from 4 to 5 at time 3. However, as can be seen in figures 4(a) and 4(b) it is either the case that execution demand remains at 4 until time 6, *or* execution demand reaches 5 from being 1. Thus, regardless which of the two scenarios would occur in a real world situation our model would incur a pessimistic estimate of the execution demand.

3.3.4 Related Work

For schedules without the major/minor cycles the simple model of Mok and Chen [MC96] mentioned before does not suffice. However, the response-time analysis for tasks with transactions using offsets [Tin92, GH98] would be able to model static schedules. In this case, the static cyclic schedule would be modelled as a transaction with high priority tasks. Each task in the transaction would correspond to one function in the schedule, and the task offset would be the function's release time.

The main disadvantage of using the offset analysis is its high computational complexity. At its core, the offset analysis uses the same fix-point iteration method as does our analysis. However, the offset analysis have to consider a prohibingly large set of cases (i.e. calculate a large set of candidate response-times). In fact, the number of cases to consider grows exponentially with the number of transactions. Hence a conservative approximative method is used. The approximation limits the number of cases that need to be considered, making the analysis technique computationally tractable. Still, a large number of cases need to be considered. Also, using the approximation will introduce pessimism into the offset analysis. Our analysis also inhibits some degree of pessimism; to study which of the two methods that introduce the least pessimism is an interesting future study.

As a side note, it should be mentioned that the response-time analysis presented in this paper could be extended to become an analysis technique for tasks with transactions using offsets. However, to further investigate this is outside the scope of this paper.

4 Case Study

The company Volvo Construction Equipment (VCE) [Vol] 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. VCE uses the Rubus OS [Arcb], that has a Red part (using static cyclic scheduling) and a Blue part (using a FPS dynamic scheduler that runs in the background of the static schedule) [HLS96]. Currently at VCE, all safety critical functionality is implemented in the Red part and only soft real-time or non real-time activity resides in the Blue part. The demand on more functionality in next generation machinery is growing. However, the static schedule is getting close to full utilisation, leaving little or no room for added functionality. This can either be addressed with new and more expensive hardware or to find a better way of utilising the current hardware resources.

Demand on responsiveness (i.e. deadlines) for functionality in the red part ranges from a few milliseconds up to several seconds. This could potentially result in a 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.

Here we will present an example system that can be viewed as a highly simplified version on one of the systems constructed by VCE. We will show how functions currently residing in the Red part can be moved to the Blue part and, by using our proposed response-time analysis, we can still guarantee that the function deadlines will be met. For our example the task specification in table 1 on the next page will be used. (For simplicity we will in this example ignore interrupts.)

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

Task i	T_i	C_i	D_i
A	10	2	10
B	20	2	5
C	50	1	2
D	50	6	50
E	100	8	100
F	2000	7	100
G	2000	8	100
H	2000	8	2000

Table 1: The set of tasks in the Red system

of its period (2000ms), however, the resulting schedule would become too large and memory consuming (it would have to extend for 2000ms). Setting the schedule length to 100ms would be adequate for all tasks except task H. Hence, the schedule length is set to 100ms, and the resulting schedule can be seen in figure 7.

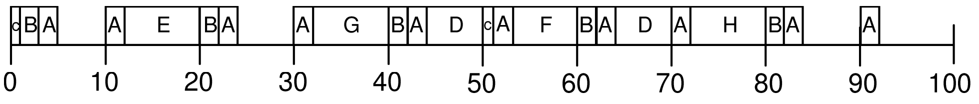


Figure 7: Static cyclic schedule for task in table 1

The total utilisation of the tasks is 75%. Adding new functionality, requiring some kind of temporal guarantee, to this system is hard, there are not many free time-slots in the schedule.

However tasks F, G, and H could be placed in the dynamic part instead, and making them event triggered, thus freeing some resources. The resulting static schedule can be seen in figure 8. The utilisation for the static schedule now becomes 52%. The utilisation for the three dynamic tasks are 1,15%, resulting in a total utilisation of just above 53%. Thus, by moving these three tasks from the static schedule we free nearly 23% of the CPU resources.

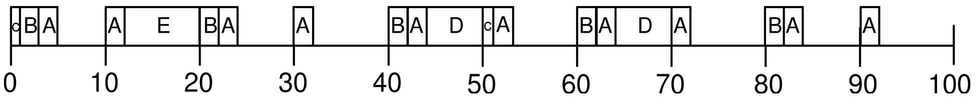


Figure 8: Static cyclic schedule with tasks F, G and H removed

Now, it remains to see whether the three tasks will meet their deadlines when running as dynamic tasks. To be able to calculate the response times for tasks F, G, and H we model the static schedule as a task. In this case we can use the simpler major/minor cycle model described in section 3.2.1, with a minor cycle of 10ms, and the model of the schedule, S, becomes:

$$C_S = [5,\ 10,\ 4,\ 2,\ 10,\ 3,\ 10,\ 2,\ 4,\ 2]$$
$$T_S = 10$$

From C_S we calculate C_S^{pre} according to section 3.2.3 to be the following:

$$C_S^{\text{pre}} = [0,\ 10,\ 15,\ 23,\ 26,\ 31,\ 39,\ 44,\ 46,\ 50,\ 52]$$

Assuming that F, G, and H have priorities high, medium, and low respectively we can now calculate the response times for the three tasks according to equation 3 in section 3.2.2. And the result is:

$$R_F = 30$$
$$R_G = 46$$
$$R_H = 67$$

And from the deadlines in table 1 we can see that all three tasks will meet their deadline.

As a side note it could be mentioned that by removing tasks F, G and H from the schedule we have enabled shorter response times for any other dynamic tasks that might have existed in the system. The schedule in figure 7 has a longest busy period of 54ms, whereas the new schedule in figure 8 has a longest busy period of 14ms. Since any dynamic task (in the worst case) will have to wait for the longest busy period, we now have significantly reduced that time.

A requirement for moving functions from the static subsystem to the dynamic subsystem is that any requirements on function precedences or function intercommunication is still met when a function is moved.

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

This solution does not require an increase in resources at any end. We save utilisation 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. Note however, that all tasks in the static schedule share a common stack, whereas, moving a task from the schedule to the dynamic part may require it to have a separate stack, hence increasing the memory consumption. However, using a resource locking protocol such as the immediate inheritance mechanism [But97]) can allow also dynamic tasks to share a single stack, hence reducing the overhead for dynamic tasks.

There is an ongoing master thesis project at VCE focusing on how much of the functionality can be shifted from the static part to the dynamic FPS part, without jeopardising any of the original timing constraints. Preliminary results suggests that there is much to gain.

5 Conclusion and Future Work

In this paper we have presented a task model with corresponding response-time analysis (i.e. a method to calculate the worst case response-times) for tasks scheduled by a fixed priority scheduler, when those tasks are executed "in the background" of a static cyclic schedule and interrupt service routines.

The response-time analysis makes it possible to develop hard real-time systems using a hybrid scheduling policy with both static cyclic scheduling and fixed priority scheduling. This, in turn, can significantly simplify the design tradeoff of whether to use static or dynamic scheduling. Using our method, both types of scheduling can be used in the same system.

We present a case study that illustrates the usefulness of our new analysis technique. The case study mainly shows two things:

1. A hybrid task model is of real industrial use, especially if one can guarantee real-time properties for all tasks.
2. The task model and the corresponding response-time analysis are applicable tools for guaranteeing real-time behaviour in hybrid scheduled systems.

As pointed out in sections 3.2.4 and 3.3.4 there are other methods that could potentially be used to guarantee timeliness for dynamic tasks in the presence of a static cyclic schedule. In the future we would like to evaluate the usefulness of these methods, and also perform a quantitative study where properties such as execution time and the degree of pessimism is studied.

Also, the response-time analysis proposed in section 3.3 could be a base for an alternative analysis method for tasks with offsets. To further extend our method to a general offset analysis method and to evaluate its performance relative to existing offset analysis methods [Tin92, GH98] would be interesting.

References

[ABD$^+$95] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell, and A.J. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2/3):129–154, 1995.

[AFF99] L. Almeida, J. A. Fonseca, and P. Fonseca. A Flexible Time-Triggered Communication System Based on the Controller Area Network: Experimental Results. In *Int. Conf. on Filedbus Technology (FeT'99)*, 1999.

[AKA94] A.Burns, K.Tindell, and A.J.Wellings. Fixed Priority Scheduling with Deadlines Prior to Completion. In *Proc. of the 6th Euromicro Workshop of Real-Time Systems*, pages 128–142, June 1994.

[Arca] Arcticus Systems Home-Page. http://www.arcticus.se.

[Arcb] Arcticus Systems. The Rubus Operating System. http://www.-arcticus.se.

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

[But97] G.C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997. ISBN 0-7923-9994-3.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[CAN92] Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communications, February 1992. ISO/DIS 11898.

[Eng02] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Uppsala University, Department of Information Technology, 2002. http://user.it.uu.se/~jakob/-phdthesis.html.

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

[FMD$^+$00] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time Triggered Communication on CAN. In *7th International CAN Conference*, 2000.

[GH98] P. C. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS)*, December 1998.

[HLS96] H. Hansson, H. Lawson, and M. Strömberg. BASEMENT a Distributed Real-Time Architecture for Vehicle Applications. *Journal of Real-Time Systems*, 3(11):223–244, November 1996.

[KG94] H. Kopetz and G. Grünsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, pages 14–23, January 1994.

[LL73] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[MC96] A. Mok and D. Chen. A Multiframe Model for Real-Time Tasks. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS)*, pages 22–29, December 1996.

[NGS+01] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, and N. E. Bånkestad. Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry. In *Eigth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. IEEE Computer Society, April 2001.

[PA00] P. Pedreiras and L. Almeida. Combining Event-triggered and Time-triggered Traffic in FTT-CAN: Analysis of the Asynchronous Messaging System. In *Proc. 3rd IEEE International Workshop on Factory Communication Systems (WFCS2000)*, September 2000.

[SEF98] K. Sandström, C. Eriksson, and G. Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In *Proc. of the 5th International conference on Real-Time Computing Systems and Applications (RTCSA'98)*, 1998.

[SH99] M. Sjödin and H. Hansson. Analysing Multimedia Traffic in Real-Time ATM Networks. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 203–212, June 1999.

[Sjö02] M. Sjödin. Response-Time Analysis for Dynamically and Statically Scheduled Systems. Technical report, Mälardalen Real-Time Research Centre (MRTC), April 2002. http://www.mrtc.mdh.-se/showPublications.phtml.

[SSNB95] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.

[Tin92] 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. Available at ftp://ftp.cs.york.ac.uk/pub/realtime/papers/YCS182_[12].ps.Z.

[Tin94] K. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, February 1994. Available at ftp://ftp.cs.york.ac.uk/pub/realtime/papers/thesis/ken/.

[TTT] Time Triggered Technologies Home Page. http://www.tttech.-com/.

[Vol] Volvo Construction Equipment Home-Page. http://www.-volvoce.com.