**DTU Library**

# Bus Access Optimisation for FlexRay-based Distributed Embedded Systems

**Pop, Traian; Pop, Paul; Eles, Petru; Peng, Zebo**

[Link back to DTU Orbit](#)

# Bus Access Optimisation for FlexRay-based Distributed Embedded Systems

Traian Pop[1], Paul Pop[2], Petru Eles[1], Zebo Peng[1]

[1]Department of Computer and Information Science, Linköping University, Sweden
[2]Informatics and Mathematical Modelling, Technical University of Denmark, Denmark
{trapo, petel, zebpe}@ida.liu.se, paul.pop@imm.dtu.dk

## Abstract

*FlexRay will very likely become the de-facto standard for in-vehicle communications. Its main advantage is the combination of high speed static and dynamic transmission of messages. In our previous work we have shown that not only the static but also the dynamic segment can be used for hard-real time communication in a deterministic manner. In this paper, we propose techniques for optimising the FlexRay bus access mechanism of a distributed system, so that the hard real-time deadlines are met for all the tasks and messages in the system. We have evaluated the proposed techniques using extensive experiments.*

## 1. Introduction

Currently, more and more real-time systems are implemented on distributed architectures in order to meet reliability, functional, and performance constraints. Communication in such systems can be triggered either dynamically, in response to an event (event-driven), or statically, at predetermined moments in time (time-driven). Therefore, on one hand, there are protocols that schedule the messages statically, based on the progression of time, such as the TTCAN [7], and Time-Triggered Protocol (TTP) [9]. A drawback of such protocols is their lack of flexibility. On the other hand, there are communication protocols where message scheduling is performed dynamically, such as CAN [2] or Byteflight [1].

In order to guarantee that real-time requirements are fulfilled, timing analysis for CAN [11] and TTP based buses [12] has been proposed.

A large consortium of automotive manufacturers and suppliers has recently proposed a hybrid type of protocol, namely the FlexRay [6]. FlexRay allows the sharing of the bus among event-driven (ET) and time-driven (TT) messages, thus offering the advantages of both worlds. FlexRay will very likely become the de-facto standard for in-vehicle communications. FlexRay is composed of static (ST) and dynamic (DYN) segments, which are arranged to form a bus cycle that is repeated periodically. The ST segment is similar to TTP, and employs a generalized time-division multiple-access (GTDMA) scheme. The DYN segment is similar to Byteflight and uses a flexible TDMA (FTD-MA) bus access scheme. While the importance of FlexRay has been quickly recognised, neither analysis nor optimisation approaches for the protocol have been available. In [5], the authors consider the case of a hard real-time application implemented on a FlexRay bus. However, in their discussion they restrict themselves exclusively to the static segment, which means that, in fact, only the classical problem of communication scheduling over a TDMA bus is considered. The per-

formance analysis of the Byteflight protocol, which is similar to the DYN segment of FlexRay, is analyzed in [3]. The authors assume a very restrictive "quasi-TDMA" transmission scheme for time-critical messages, which basically means that the DYN segment would behave as an ST (TDMA) segment in order to guarantee timeliness.

In [14] we have proposed timing analysis techniques for FlexRay, which are able to bound the message transmission times on both the ST and DYN segments. This was the first step towards enabling the use of this protocol in a systematic way for time critical applications. The second step towards an efficient use of FlexRay is taken in this paper. We propose an approach for determining a FlexRay bus configuration which is adapted to the particular features of an application and guarantees that all time constraints are satisfied. Heuristics for solving the bus access optimisation problem with FlexRay are proposed. While the optimisation techniques proposed by us can also be applied to other heterogeneous distributed applications, solving the particular problem of analysis and optimisation of Flexray-based systems is, today, of particular importance for the automotive industry.

The paper is organised in eight sections. Section 2 presents the system architecture, and Section 3 introduces the FlexRay media access control. In Section 4 we present the application model and in Section 5 we briefly introduce our timing analysis for systems with the FlexRay protocol. In Section 6 we describe the proposed optimisation techniques. Section 7 presents the experimental results followed by conclusions in Section 8.

## 2. System Model

We consider architectures consisting of nodes connected by a FlexRay communication channel (see Fig. 1). Each processing node is composed of a CPU and a communication controller that are interconnected through a two-way controller-host interface (CHI). The controller runs independently of the node's CPU and implements the FlexRay protocol services.

Each node runs a real-time kernel that contains two schedulers, for static cyclic scheduling (SCS) and fixed priority scheduling (FPS), respectively. When several tasks are ready on a node, the task with the highest priority is activated, and preempts the other tasks. SCS tasks are not preemptable and their start time is off-line fixed in the schedule table. FPS tasks can only be executed in the slack of the SCS schedule table. FPS tasks are scheduled based on priorities. SCS activities are triggered based on a local clock in each processing node. The synchronization of local clocks throughout the system is provided by the communication protocol [6].
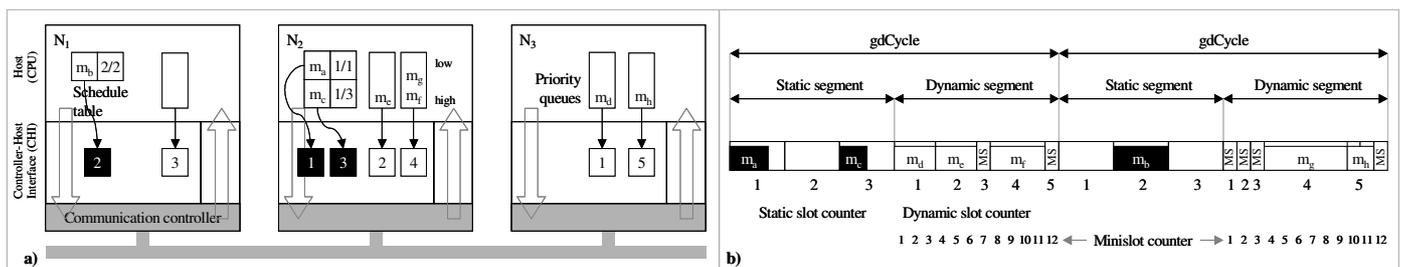


**Figure 1. FlexRay Communication Cycle Example**

## 3. The FlexRay Communication Protocol

Let us consider the example in Fig. 1 where we have three nodes, $N_1$ to $N_3$ sending messages $m_a$, $m_b$,... $m_h$ using a FlexRay bus.

In FlexRay, the communication takes place in periodic cycles (Fig. 1.b depicts two cycles of length $gdCycle$). Each cycle contains a static (ST) segment and a dynamic (DYN) segment. The ST and DYN segment lengths can differ, but are fixed over the cycles. We denote with $ST_{bus}$ and $DYN_{bus}$ the length of these segments. Both the ST and DYN segments are composed of several slots. In the ST segment, the slots number is fixed, and the slots have constant and equal length, regardless of whether ST messages are sent or not over the bus in that cycle. The length of an ST slot is specified by the FlexRay global configuration parameter $gdStaticSlot$ [6]. In Fig. 1 there are three static slots for the ST segment.

The length of the DYN segment is specified in number of "minislots", and is equal to $gNumberOfMinislots$. Thus, during the DYN segment, if no message is to be sent during a certain slot, then that slot will have a very small length (equal to the length $gdMinislot$ of a so called minislot), otherwise the DYN slot will have a length equal with the number of minislots needed for transmitting the whole message [6]. This can be seen in Fig. 1.b, where DYN slot 2 has 3 minislots (4, 5, and 6) in the first bus cycle, when message $m_e$ is transmitted, and one minislot (denoted with "MS" and corresponding to the minislot counter 2) in the second bus cycle when no message is sent.

During any slot, only one node is allowed to send on the bus, and that is the node which holds the message with the frame identifier ($FrameID$) equal to the current value of the slot counter. There are two slot counters, corresponding to the ST and DYN segments, respectively. The assignment of frame identifiers to nodes is static and decided offline, during the design phase. Each node that sends messages has one or more ST and/or DYN slots associated to it. The bus conflicts are solved by allocating offline one slot to at most one node, thus making it impossible for two nodes to send during the same ST or DYN slot.

In Fig. 1, node $N_1$ has been allocated ST slot 2 and DYN slot 3, $N_2$ transmits through ST slots 1 and 3 and DYN slots 2 and 4, while node $N_3$ has DYN slots 1 and 5. For each of these slots, the CHI reserves a buffer that can be written by the CPU and read by the communication controller (these buffers are read by the communication controller *at the beginning* of each bus slot, in order to prepare the transmission of frames).

For ST messages, we consider that the CPU in each node holds a schedule table with their transmission times. When the time comes for an ST message to be transmitted, the CPU will place that message in its associated ST buffer of the CHI. For example, ST message $m_b$ sent from node $N_1$ has an entry "2/2" in the schedule table, specifying that it should be sent in the second slot of the second ST cycle.

For the DYN messages, the designer specifies their $FrameID$. For example, message $m_e$ has the frame identifier "2", while messages $m_g$ and $m_f$ have both $FrameID$ 4. If two messages with the same frame identifier are ready to be sent in the same bus cycle, a priority scheme is used to decide which one will be sent first. Each DYN message $m_i$ has associated a priority, $priority_{m_i}$. Messages with the same $FrameID$ will be placed in an output queue ordered by their priorities.

At the beginning of each communication cycle, the communication controller of a node resets the slot and minislot counters. At the beginning of each communication slot, the controller verifies if there are messages ready for transmission (present in the CHI send buffers). In the example in Fig. 1 we assume that all messages are ready for transmission before the first bus cycle.

Messages packed into ST frames will be transmitted according to the schedule table. For example, in Fig. 1, messages $m_a$ and $m_c$ are placed into the associated ST buffers in the CHI in order to be transmit-

ted in the first bus cycle. However, messages selected and packed into DYN frames will be transmitted during the DYN segment of the bus cycle only if there is enough time until the end of the DYN segment. This is verified by comparing if, in the moment the DYN slot counter reaches the value of the $FrameID$ for that message, the value of the minislot counter is smaller than a given value $pLatestTx$. The value $pLatestTx$ is fixed for each node during the design phase, depending on the size of the largest DYN frame that node will have to send during run-time. For example, in Fig. 1, message $m_h$ is ready for transmission before the first bus cycle starts, but, after message $m_f$ is transmitted, there is not enough room left in the DYN segment. This will delay the transmission of $m_h$ for the next bus cycle.

## 4. Application Model

We model an application $\mathcal{A}$ as a set of directed, acyclic, polar graphs $G_i(\mathcal{V}_i, \mathcal{E}_i) \in \mathcal{A}$. A node $\tau_{ij} \in \mathcal{V}_i$ represents the $j$-th task or message in $G_i$. An edge $e_{ijk} \in \mathcal{E}_i$ from $\tau_{ij}$ to $\tau_{ik}$ indicates that the output of $\tau_{ij}$ is the input of $\tau_{ik}$. A task becomes ready after all its inputs have arrived and it issues its outputs when it terminates. The communication time between tasks mapped on the same processor is considered to be part of the task worst-case execution time and is not modeled explicitly. Communication between tasks mapped to different processors is by message passing over the bus and is modeled as a communication task inserted on the arc connecting the sender and the receiver task.

We consider that the scheduling policy for each task is known (either *SCS* or *FPS*), and we also know which messages are ST and which are DYN. For a task $\tau_{ij} \in \mathcal{V}_i$, $Node_{\tau_{ij}}$ is the node to which $\tau_{ij}$ is assigned for execution. When executed on $Node_{\tau_{ij}}$, a task $\tau_{ij}$ has a known worst-case execution time $C_{\tau_{ij}}$. We also consider that the size of each message $m$ is given, which can be directly converted into communication time $C_m$ on the particular bus, knowing the speed of the bus and the size of the frame that stores the message:

$$C_m = Frame\_size(m) \; / \; bus\_speed. \qquad (1)$$

Tasks and messages activated based on events also have a priority, $priority_{\tau_{ij}}$. All tasks and messages belonging to a task graph $G_i$ have the same period $T_{\tau_{ij}} = T_{G_i}$ which is the period of the task graph. A deadline $D_{G_i}$ is imposed on each task graph $G_i$. In addition, tasks can have associated individual release times and deadlines. If communicating tasks are of different periods, they are combined into a larger graph capturing all task activations for the hyper-period (LCM of periods).

## 5. Timing Analysis

Given a system as described in the previous sections, the tasks and messages have to be scheduled. For SCS tasks and ST messages, this means building the schedule tables, while for the FPS tasks and DYN messages we have to determine their worst case response times.

Two aspects have to be considered during such a timing analysis:

1. When performing the schedulability analysis for the FPS tasks and DYN messages, one has to take into consideration the interference from the SCS activities.
2. Among the possible correct schedules for SCS activities, it is important to build one which favours as much as possible the schedulability of FPS activities.

Fig. 2 presents the global scheduling and analysis algorithm, in which the main loop consists of a list-scheduling [4] based algorithm that iteratively builds the static schedule table with start times for SCS tasks and ST messages. A ready list (*TT_ready_list*) contains all SCS tasks and ST messages which are ready to be scheduled (they have no predecessors or all their predecessors have already been scheduled). From the ready list, tasks and messages are extracted one by one (line 2) to be scheduled on the processor they are mapped to (line 4), or into a static bus-slot associated to that processor on which the sender of the

**Figure 2. Global Scheduling Algorithm**

message is executed (line 6), respectively. The priority function which is used to select among ready tasks and messages is a modified critical path metric [12]. Let us consider a particular task $\tau_{ij}$ selected from the ready list to be scheduled. We consider that $ASAP_{\tau_{ij}}$ is the earliest time moment which satisfies the condition that all preceding activities (tasks or messages) of $\tau_{ij}$ are finished (line 10). With only the SCS tasks in the system, the straightforward solution would be to schedule $\tau_{ij}$ at the first time moment after $ASAP_{\tau_{ij}}$ when $Node_{\tau_{ij}}$ is free. Similarly, an ST message will be scheduled in the first available ST slot associated with the node that runs the sender task for that message.

As presented by us in [13], when scheduling SCS tasks, one has to take into account the interference they produce on FPS tasks. The function *schedule_TT_task* places a SCS task in the static schedule in such a way that the increase of worst-case response times for FPS tasks is minimised. This increase is determined by comparing the worst-case response times of FPS tasks obtained with our holistic schedulability analysis before and after inserting the new SCS task in the schedule.

In [14] we presented the holistic schedulability analysis for FlexRay-based systems (which is called in line 11 of the algorithm in Figure 2). In the next subsection we only outline a small part of the analysis which is concerned with the delay of DYN messages.

## 5.1 Schedulability Analysis of DYN Messages

The worst case response time $R_m$ of a message $m$ transmitted in the DYN segment of a FlexRay bus is given by the following equation:

$$R_m(t) = J_m + w_m(t) + C_m \qquad (2)$$

where $J_m$ is the message jitter inherited from the sender task, $C_m$ is the message communication time (see Section 4), and $w_m$ represents the worst case delay caused by the transmission of ST frames and higher priority DYN messages during a given time interval $t$.

For the calculation of $w_m$, we start from the observations that the transmission of a ready DYN message $m$ during the DYN slot $FrameID_m$ can be delayed because of the following causes:

- local messages with higher priority, that use the same frame identifier as $m$. We will denote this set of *higher priority local messages* with $hp(m)$. For example, in Fig. 1.a, messages $m_g$ and $m_f$ share $FrameID$ 4, thus $hp(m_g) = \{m_f\}$.
- any messages in the system that can use DYN slots with lower frame identifiers than the one used by $m$. We will denote this set of messages having *lower frame identifiers* with $lf(m)$. In Fig. 1.a, $lf(m_g) = \{m_d, m_e\}$.
- unused DYN slots with frame identifiers lower than the one used for sending $m$ (though such slots are unused, each of them still delays the transmission of $m$ for an interval of time equal with the length $gdMinislot$ of one minislot); we will denote the set of such minislots with $ms(m)$. Thus, in the example in Fig. 1.a, $ms(m_g) = \{1, 2, 3\}$, and $ms(m_f) = \{3\}$.

We next focus on determining the delay $w_m(t)$ in Eq. (2). The delay produced by the elements in $hp(m)$, $lf(m)$ and $ms(m)$ can extend to one or more bus cycles. Hence, the delay $w_m$ has the following expression:

$$w_m(t) = \sigma_m + BusCycles_m(t) \times gdCycle + w'_m(t) \qquad (3)$$

where $\sigma_m$ is the longest delay suffered during one bus cycle if the message is generated by its sender task after its slot has passed. $BusCycles_m(t)$ is the number of bus periods for which the transmission of $m$ is not possible because transmission of messages from $hp(m)$ and $lf(m)$ and because of minislots in $ms(m)$. The delay $w'_m(t)$ denotes the time, in the last bus cycle, until $m$ is sent, and is measured from the beginning of the bus cycle in which message $m$ is sent until the actual transmission of $m$ starts.

In order to solve the recurrence Eq. (3), we start from a value of $t=0$ and compute $w_m(t)$. If $w_m(t) > t$ then we assign $t = w_m(t)$ and compute iteratively the solution until $w_m(t) = t$.

To compute the value $BusCycles_m$ we start with the observation that a message $m$ with $FrameID_m$ cannot be sent by a node $N_p$ during a bus cycle $b$ if at least one of the following conditions is fulfilled:

1. There is too much interference from elements in $lf(m)$ and $ms(m)$, so that the minislot counter exceeds the value $pLatestTx_{N_p}$, making impossible for $N_p$ to start the transmission of $m$ during $b$.
2. The DYN slot $FrameID_m$ in $b$ is used by another local higher priority message from $hp(m)$.

Whenever a bus cycle satisfies at least one of these two conditions, it will be called "filled", since it is unusable for the transmission of the message $m$ under analysis. In the worst case, the value $BusCycles_m(t)$ is then the maximum number of bus cycles that can be filled using elements from $hp(m)$, $lf(m)$ and $ms(m)$.

In [14], we have proposed both exact approaches and polynomial complexity heuristics to compute all individual components of the delay $w_m(t)$ and, finally, the worst case response times $R_m$.

## 6. Bus Access Optimisation

The design of a FlexRay bus configuration for a given system consists of a collection of solutions for the following subproblems: (1) determine the length of an ST slot, (2) the number of STslots, and (3) their assignment to nodes; (4) determine the length of the DYN segment, (5) assign DYN slots to nodes, and (6) $FrameID$s to DYN messages.

The choice of a particular bus configuration is extremely important when designing a specific system, since its characteristics heavily influence the global timing properties of the application.

For example, notice in Fig. 3 how the structure of the ST segment influences the response time of message $m_3$ (for this example we ignored the DYN segment). The figure considers a system with two nodes, $N_1$ that sends message $m_1$ and $N_2$ that sends messages $m_2$ and $m_3$. The message sizes are depicted in the figure. In a first scenario, the ST segment consists of two slots, $slot_1$ used by $N_1$ and $slot_2$ used by $N_2$. In this situation, message $m_3$ can be scheduled only during the second bus cycle, with a response time of 16. If the ST segment consists of 3 slots (Fig. 3.b), with $N_2$ being allocated $slot_2$ and $slot_3$, then $N_2$ is able to send
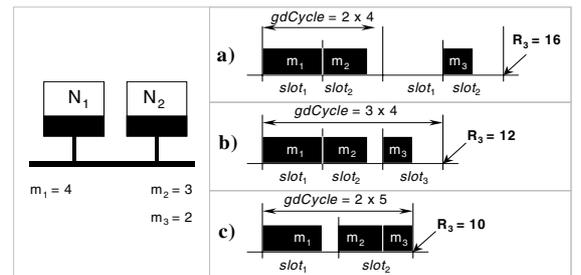


**Figure 3. Optimisation of the ST segment**
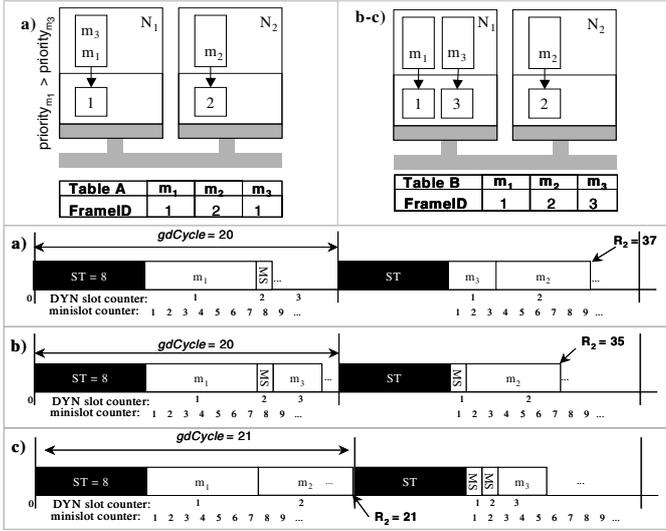
**Figure 4. Optimisation of the DYN Segment**

both its messages during the first bus cycle. The configuration in Fig. 3.c consists of only two slots, like in Fig. 3.a. However, in this case the slots are longer, such that several messages can be transmitted during the same frame, producing a faster response time for $m_3$ (one should notice, however, that, by extending the size of the ST slots, we delay the reception of message $m_1$ and $m_2$).

Similar optimisations can be performed with regard to the DYN segment. Let us consider the example in Fig. 4, where we have two nodes $N_1$ and $N_2$. Node $N_1$ is transmitting messages $m_1$ and $m_3$, while $N_2$ sends $m_2$. Fig. 4 depicts three configuration scenarios, a-c. Table A depicts the frame identifiers for the scenario in Fig. 4.a, while Table B corresponds to Fig. 4.b-c. The length of the ST slot has been set to 8. In Fig. 4.a, the length of the DYN segment is not able to accommodate both $m_1$ and $m_2$, thus $m_2$ will be sent during the second bus cycle, after the transmission of $m_3$ ends. Fig. 4.b and 4.c depict the same system but with a different allocation of DYN slots to messages (Table B). In Fig. 4.b we notice that $m_3$, which now does not share the same frame identifier with $m_1$, can be sent during the first bus cycle, thus $m_2$ will be transmitted earlier during the second cycle. Moreover, if we enlarge the size of the DYN segment as in Fig. 4.c, then the worst-case response time of $m_2$ will considerably decrease since it will be sent during the first bus cycle (notice that in this case $m_3$, having a greater frame identifier than that of $m_2$, will be sent only during the second cycle).

## 6.1 The Basic Bus Configuration (BBC)

In this section we construct a basic bus configuration which results from analyzing the minimal bandwidth requirements of the application. The BBC algorithm is presented in Fig. 5 and it starts by assigning a *FrameID* to each of the DYN messages (implicitly DYN slots are assigned to the nodes that generate the message). This assignment (line 1) is performed under the following guidelines:

- Each DYN message receives an unique *FrameID*; this is recommended in order to avoid delays due to messages in the set $hp(m)$, as discussed in Section 5.1. For example, in Fig. 4, we notice that message $m_3$ has to wait for an entire *gdCycle* when it shares a frame identifier with the higher priority message $m_1$ (Fig. 4.a), which is not the case when it has its own *FrameID* (Fig. 4.b).
- DYN messages with a higher criticality receive smaller *FrameID*s.; this is required in order to reduce, for a given message, the delay produced by $lpf(m)$ and $ms(m)$ (see Section 5.1). We capture the criticality of a message $m$ as:

$$CP_m = D_m - LP_m, \qquad (4)$$

```
1   Assign FrameIDs to DYN messages
2   gdNumberOfStaticSlots = nodes_ST
3   gdStaticSlot = max (C_m), m is an ST message
4   assign one ST slot to each node (round robin)
5   for DYN_bus = DYN_bus^min to DYN_bus^max step gdMinislot do
6       gdCycle = ST_bus + DYN_bus
7       if gdCycle < 16000 μs then
8           GlobalSchedulingAlgorithm()
9           Compute cost function Cost
10          if Cost < BestCost then save current solution
11      endif
12  end for
```

**Figure 5. Basic Bus Configuration**

where $D_m$ is the deadline of the message and $LP_m$ is the longest path in the task graph from the root to the node representing the communication of message $m$. A small value of $CP_m$ (higher criticality) indicates that the message should be assigned a smaller *FrameID*.

In the next step, the algorithm sets the number of ST slots in a bus cycle (line 2). Since each node that generates ST messages needs at least one ST slot, the minimum number of ST slots is $nodes_{ST}$, the number of nodes that send ST messages. Next, the size of an ST slot is set so that it can accommodate the largest ST message in the system (line 3). In line 4, the configuration of the ST segment is completed by assigning in a round robin fashion one ST slot to each node that requires one (i.e. in a system with four nodes, the ST segment will contain four slots: node 1 will use slot 1, node 2 will use ST slot 2, etc.).

In order to determine the size of the DYN segment, we have to consider the fact that such a size is restricted by the protocol specifications (there can be at most 7994 minislots in a DYN segment) and by the application characteristics (the DYN segment should be large enough in order to accommodate the transmission of the largest DYN message; in addition, since we assumed that each DYN message has an unique *FrameID*, the DYN segment should have a number of minislots greater or equal than the number of DYN messages in the system). We denote with $DYN_{bus}^{min}$ and $DYN_{bus}^{max}$ the limits of this interval (line 5).

Since the sizes of the ST and DYN segments are now fixed, the bus period can be easily computed (line 6). Line 7 introduces a restriction imposed by the FlexRay specification, which limits the maximum bus cycle length to 16 ms.

Once we have defined the structure of the bus cycle, we can analyse the entire system (line 8) by performing the global static scheduling and schedulability analysis described in Section 5. The resulted system is then evaluated using a cost function that captures the schedulability degree of the system (line 9):

$$Cost = \begin{cases} f_1 = \sum_{\tau_{ij}} max(R_{ij} - D_{ij}, 0), \ if \ f_1 > 0 \\ f_2 = \sum_{\tau_{ij}} (R_{ij} - D_{ij}), \ if \ f_1 = 0 \end{cases} \qquad (5)$$

where $R_{ij}$ and $D_{ij}$ are the worst case response times and respectively the deadlines for all the activities $\tau_{ij}$ in the system. The function is strict positive if at least one task or message in the system misses its deadline, and negative if the whole system is schedulable. Its value is used in line 10, when deciding whether the current configuration is the best so far.

## 6.2 Heuristic for Optimised Bus Configuration (OBC)

The Basic Bus Configuration (BBC) generated as in the previous section can result in an unschedulable system (the cost function in Eq. (5) is positive). In this case, additional points in the solution space have to be explored. In Fig. 6 we present the OBC heuristic that further explores the design space in order to find a schedulable solution.

While for the BBC the number and size of ST slots has been set to the minimum ($gdNumberOfStaticSlots_{min} = nodes_{ST}$, $gdStaticSlot_{min} = max(C_m)$), the proposed heuristic explores different alternatives between these minimal values and the maxima imposed by the protocol specification (the for loops over lines 2 - 9 and 4 - 8). Thus, during a bus cycle there can be at most $gdNumberOfStaticSlots_{max} = 1023$ ST slots (line

```
1  Assign FrameIDs to DYN messages (similar to BBC, Fig. 5, line 1)
2  for gdNumberOfStaticSlots =
3    gdNumberOfStaticSlots_min to gdNumberOfStaticSlots_max do
4      for gdStaticSlot = gdStaticSlot_min to gdStaticSlot_max step 20 * gdBit do
5        Assign ST slots to nodes in round-robin fashion
6        DYN_bus = Determine_DYN_segment_length()
7        End optimisation if feasible DYN_bus and Cost ≤ 0
8      end for
9  end for
```

**Figure 6.  OBC Heuristic**

3), while the size of an ST slot can take at most $gdStaticSlot_{max} = 661$ macroticks. In addition, the payload for a FlexRay frame can increase only in 2-byte increments, which according to the FlexRay specification translates into 20 $gdBit$, where $gdBit$ is the time needed for transmitting one bit over the bus (line 4).

The assignment of ST slots (line 5) to nodes is performed, like for the BBC, in a round robin fashion, with the difference that each node can have not only one but a quota of ST slots, determined by the ratio of ST messages that it transmits (i.e. a node that sends more ST messages will be allocated more ST slots)

For each alternative configuration of the ST segment, the algorithm searches for that size of the DYN segment that allows the DYN messages to meet their deadlines and the cost function in Eq. (5) to be minimised (line 6). A straight forward alternative to perform this would be to evaluate all possible sizes of the DYN segment inside a for loop, like in the BBC algorithm (lines 5-12, Fig. 5). However, as opposed to the BBC, in the proposed heuristic the selection of the DYN segment length is nested inside two for loops (lines 2 and 4, Fig. 6). Moreover, the estimation of each individual solution alternative implies a complete scheduling and schedulability analysis of the systems (like in line 8, Fig. 5). Therefore, in the context of the heuristic in Fig. 6, such a straight forward approach is not affordable, due to excessively long run times. This is important, since, in the context of a system-level design framework, the bus access optimisation heuristic can be placed inside other optimisation loops, e.g. for task mapping. Thus, instead of evaluating the cost function in (Eq. (5)) for all possible lengths of the DYN segment, the evaluation should be performed for only a reduced number of points while, at the same time, obtaining a close to optimal result. The proposed solution is presented in the next subsection.

### 6.2.1 Curve-fitting Based Heuristic for DYN Segment Length

Let us go back to the schedulability analysis in Section 5.1. One can notice in Eq. (3) that the dominant part of the message delay is represented by the product between $BusCycles_m$ (number of bus cycles that the message under analysis has to wait) and $gdCycle$ (length of the bus cycle). If we consider a time interval $t$ on which a fixed set of messages $S$ is generated, then a shorter size for the bus cycle means that fewer messages will be served during such a cycle; consequently, several such bus cycles are needed to transmit all the messages in $S$ (a shorter $gdCycle$ results in larger $BusCycles_m$). A longer bus cycle generally means that more messages can be sent during the same bus period, resulting in a lower number of bus cycles required for the transmission of all messages (a larger $gdCycle$ results in smaller $BusCycles_m$). This trade-off is illustrated in Fig. 7, where we consider a system composed of 45 tasks which communicate through 10 static and 20 dynamic messages. We have performed the response time analysis for this system, assuming the length of the dynamic segment between 2285.4 and 13000 µs. The static segment size is fixed at 1286 and, consequently, the total size of the bus cycle is varying between 3571.4 and 14286µs. Fig. 7 shows the response time for several dynamic messages in this system. The curves confirm the trade-off outlined above. Large sizes of the bus cycle lead to increased response times. However, very short bus cycles will also lead to large response times due to the fact that the number of cycles to wait ($BusCycles_m$ in Eq. (3)) increases. This phe-
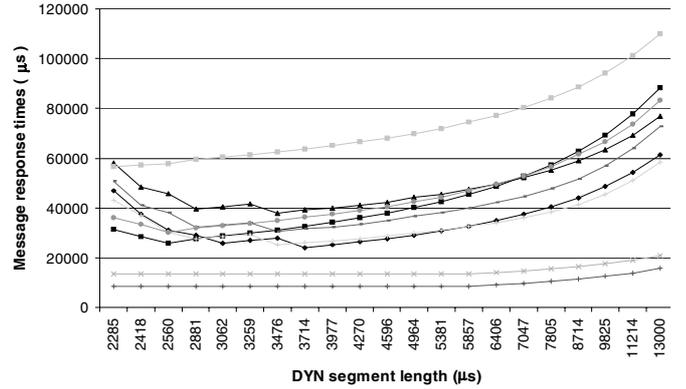


**Figure 7.  Influence of DYN Segment Length on Message Response Times**

nomenon has been confirmed by a large number of experiments similar to those illustrated in Fig. 7. This regularity of the dependence *response time* vs. *size of the dynamic segment* is at the foundation of our heuristic presented in Fig. 8 (which is invoked in line 6 of the OBC algorithm in Fig. 6). Instead of exhaustively perform the scheduling and schedulability analysis for all possible values of the DYN segment length, we will evaluate response times for only a small number of points and use a curve fitting approach to extrapolate the response time corresponding to all other points.

The algorithm in Fig. 8 stores, in the set *Points,* the characteristics (DYN segment length, message response times, cost function) for a reduced set of bus configurations, The set initially contains only a small number (in our experiments we used five) of DYN segment sizes in the interval $[DYN_{bus}^{min}, DYN_{bus}^{max}]$ (line 1). For each alternative configuration in this initial set, our heuristic will run the global scheduling and analysis algorithm, in order to determine the worst-case response times of all tasks and messages and the corresponding values of the cost function (line 3). For all possible values of the size of the DYN segment (line 6), the response times of messages in the system are then interpolated[1], based on the response times computed for the configurations stored in the set *Points* (lines 8-10). Next, the point $DYN_{bus}$ with the best schedulability degree $Cost_{min}$ (i.e. minimum cost function) is investigated (line 11). The following alternatives have to be considered:

```
Determine_DYN_segment_length()
1  Points = { DYN_bus^min , DYN_bus^1 , DYN_bus^2 ,..., DYN_bus^max }
2  for DYN_bus in Points do
3      GlobalSchedulingAlgorithm()
4      Store message response times R_m(DYN_bus) and cost function Cost
5  end for
6  for DYN_bus = DYN_bus^min to DYN_bus^max step gdMinislot do
7      if DYN_bus is in Points then continue;
8      foreach DYN msg m interpolate R_m(DYN_bus) based on R_m(Points)
9      Compute and store Cost(interpolated response times)
10 end for
11 select DYN_bus with minimum stored Cost_min
12 if Cost_min ≤ 0 and DYN_bus ∈ Points then return DYN_bus
13 if Cost_min ≤ 0 and DYN_bus ∉ Points then
14     GlobalSchedulingAlgorithm(); if schedulable then return DYN_bus;
15     if not termination condition then Add DYN_bus to Points; goto 6
           else return infeasible DYN_bus
16 end if
17 if DYN_bus ∉ Points then GlobalSchedulingAlgorithm(); Add DYN_bus to Points
18 else select (DYN_bus', Cost'_min) for which response times are interpolated;
19     GlobalSchedulingAlgorithm(); Add DYN_bus' to Points
20 endif
21 if not termination condition then goto 6 else return infeasible DYN_bus
```

**Figure 8.  Determining the Size of the DYN segment**

---

1.  For curve fitting, we use a Newton polynomial, which is extremely fast, in particular when recalculating the values after a new point has been added to the set *Points*.

- $Cost_{min} \le 0$ and $Dyn_{bus} \in Points$ (the system is schedulable and the cost function is based on exact schedulability analysis): we have found the result (line 12).
- $Cost_{min} \le 0$ and $Dyn_{bus} \notin Points$ (the systems is schedulable but the cost function is based on interpolated values): schedulability analysis is performed for the point $DYN_{bus}$; if the system is schedulable, we have found the result (line 14), otherwise $DYN_{bus}$ is included into the set *Points* and the process is continued with a more exact interpolation (line 15).
- $Cost_{min} > 0$ (the system is not schedulable): the process is continued with a more exact interpolation, by adding one more point into the set *Points*. If $Dyn_{bus} \notin Points$, $DYN_{bus}$ will be added to *Points* for the next iteration (line 17); if this is not the case, then the point $DYN'_{bus}$ will added, which is the interpolated point corresponding to the smallest cost function (lines 18-19).

The algorithm stops if a schedulable solution is found or if a certain *termination condition* is met. This condition is that a certain number $N_{max}$ of iterations have been performed without finding a schedulable solution and without any improvement of the cost function (in our experiments we had $N_{max}$=10).

## 7. Experimental Results

In order to evaluate our optimisation algorithms we generated 7 sets of 25 applications representing systems of 2 to 7 nodes respectively. We considered 10 tasks mapped on each node, leading to applications with a number of 20 to 70 tasks. Depending on the mapping of tasks, each such system had up to 60 additional nodes in the application task graph due to the communication tasks. The tasks were grouped in task graphs of 5 tasks each. Half of the tasks in each system were time triggered and half were event triggered. The execution times were generated in such a way that the utilisation on each node was between 30% and 60% (similarly, the message transmission times were generated so that the bus utilisation was between 10% and 70%). All experiments were run on an AMD Athlon 2400+ PC.

We have performed the bus optimisation using four approaches: (1) BBC (Section 6.1), (2) OBCCF - the OBC heuristic with the curve fitting procedure (Section 6.2), (3) OBCEE - the OBC heuristic with an exhaustive exploration of the sizes for the DYN segment, and (4) SA - a Simulated Annealing [8] based design space exploration, which we implemented with the goal to provide a base-line for evaluation of the proposed heuristics. In order to produce close to optimal results we have performed extremely long runs (several hours) using the SA-based algorithm with moves concerning not only the number and size of static slots and size of the DYN segment, but also the assignment of slots to nodes and *FrameID*s to messages.

Fig. 9 shows the results obtained after running our algorithms on the generated applications. On the left side of the figure one can see the average percentage deviation for the cost function obtained with BBC, OBCCF and OBCEE respectively, relative to the cost function obtained with SA. On the right side we present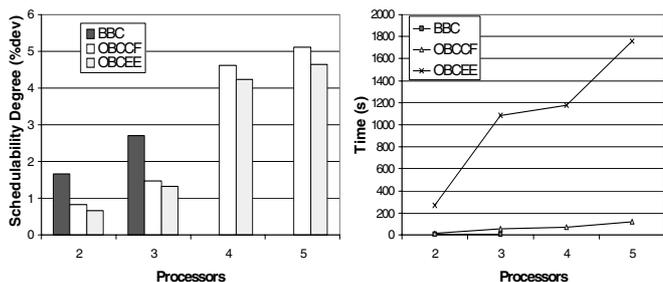 the computation times required by each algorithm. One can notice that the BBC approach runs in almost zero time, but it fails to find any schedulable configurations for systems with more than 3 processors. On the other hand, the other approaches continue to find schedulable solutions even for larger systems. Comparing OCCF and OBCEE, we can observe that both produce results which are very close to the reference values produced by SA (max. 4-5% deviation). OBCCF generates results which are less than 0.5% away from those produced by OBCEE, but with a run time that is up to 2 orders of magnitude smaller. This confirms the high efficiency of our curve fitting based optimisation.

Finally, we considered a real-life example implementing a vehicle cruise controller that consists of 54 tasks and 26 messages grouped in 4 task graphs that are mapped over 5 nodes. Two of the task graphs were time triggered and the other two were event triggered. Configuring the system using the BBC approach took less than 5 seconds but resulted in an unschedulable system. Using the OBCCF approach took 137 seconds, while the OBCEE required 29 minutes. The cost function obtained by OBCCF was 1.2% larger in the solution obtained with OBCEE. In both cases the selected bus configuration resulted in a schedulable system.

## 8. Conclusions

FlexRay is rapidly becoming the de-facto standard for automotive electronics. The performance of systems based on such a protocol can be improved by carefully adapting the bus cycle to the particular requirements of the application. In this paper, we have presented bus optimisation approaches for systems based on FlexRay and we evaluated them through extensive experiments. Together with the timing analysis proposed by us in [14], this constitutes an important step towards a systematic and efficient use of FlexRay for time critical applications.

## References

[1] J. Berwanger, M. Peller, R. Griessbach, "A New High Performance Data Bus System for Safety-Related Applications", http://www.byteflight.de, 2000.
[2] R. Bosch GmbH, *CAN Specification Version 2.0*, 1991.
[3] G. Cena, A. Valenzano, "Performance analysis of Byteflight networks", Proceedings of the IEEE International Workshop on Factory Communication Systems, 157-166, 2004.
[4] E.G. Coffman Jr., R.L. Graham, "Optimal Scheduling for two Processor Systems", *Acta Informatica*, 1, 1972.
[5] S. Ding, N. Murakami, H. Tomiyama, H. Takada, "A GA-Based Scheduling Method for FlexRay Systems", *Proceedings of EMSOFT*, 2005.
[6] FlexRay homepage: http://www.flexray-group.com, 2006.
[7] ISO, "Road Vehicles - Controller Area Network (CAN) - Part 4: Time Triggered Communication", Standard ISO/CD 11898-4, International Organization for Standardization, 2001.
[8] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, "Optimisation by simulated annealing", *Science*, 220, 671-680, 1983.
[9] H. Kopetz, G. Bauer, "The time-triggered architecture", *Proceedings of the IEEE*, 91(1), 112-126, 2003.
[10] J. C. Palencia, M. Gonzaléz Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets", *Proceedings of the Real-Time Systems Symposium*, 26–38, 1998.
[11] K. Tindell, A. Burns, A. Wellings, "Calculating CAN Message Response Times", Control Engineering Practice, 3(8), 1163-1169, 1995.
[12] P. Pop, P. Eles, Z. Peng, "Schedulability-Driven Communication Synthesis for Time-Triggered Embedded Systems", Real-Time Systems Journal, 24, pp. 297-325, 2004.
[13] T. Pop, P. Pop, P. Eles, Z. Peng, "Optimization of Hierarchically Scheduled Heterogeneous Embedded Systems", 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), Hong Kong, pp. 67-71, 2005
[14] T. Pop, P. Pop, P. Eles, Z. Peng, A. Andrei, "Timing Analysis of the FlexRay Communication Protocol", Proceedings of 18th EuroMicro Conference on Real-Time Systems, Dresden, pp. 203-213, 2006.

**Figure 9. Evaluation of Bus Optimisation Algorithms**