JAMS: Jitter-Aware Message Scheduling for FlexRay Automotive Networks

Special Session Paper

Vipin Kumar Kukkala*, Sudeep Pasricha*, Thomas Bradley†
*Department of Electrical and Computer Engineering
†Department of Mechanical Engineering
Colorado State University, Fort Collins, CO, U.S.A.
{vipin.kukkala, sudeep, thomas.bradley}@colostate.edu

ABSTRACT

FlexRay is becoming a popular in-vehicle communication protocol for the next generation x-by-wire applications such as drive-by-wire and steer-by-wire. The protocol supports both time-triggered and event-triggered transmissions. One of the important challenges with time-triggered transmissions is jitter, which is the unpredictable delay-induced deviation from the actual periodicity of a message. Failure to account for jitter can be catastrophic for time critical automotive applications. In this paper we propose a novel scheduling framework (JAMS) that handles both jitter affected time-triggered messages and high priority event-triggered messages to ensure message delivery while satisfying timing constraints. At design time, JAMS handles packing of multiple signals from Electronic Control Units (ECUs) into messages, and synthesizes a schedule using intelligent heuristics. At runtime, a Multi-Level Feedback Queue handles jitter affected time-triggered messages, and high priority event-triggered messages. We also propose a runtime scheduler that packs these messages into the FlexRay static segment slots depending on available slack. Experimental analysis indicates that JAMS improves the response time by 25.3% on average and up to 41% compared to the best-known prior work in the area.

Categories and Subject Descriptors: C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; C.4 [**Performance of systems**]: Fault tolerance

Keywords: Flexray, scheduling, jitter, automotive networks

1. INTRODUCTION

In today's automobiles, Electronic Control Units (ECUs) are the basic processing units that control different components in the vehicle. ECUs run various types of automotive applications such as the anti-lock braking control, cruise control, transmission control, etc. Most of these automotive applications have strict deadlines and latency constraints and thus they are classified as hard real-time applications [1]. In general, ECUs are distributed across the vehicle and communicate with each other by exchanging signals. A signal can be any raw data value or a control pulse that is packed into a message and transmitted as a frame via the in-vehicle network. The most popular and widely used in-vehicle network is the Controller Area Network (CAN) bus. CAN is a serial protocol which supports a maximum payload of 8 bytes and a typical transmission rate of up to 1 Mbps [2]. Some of the key features of the CAN bus include low cost, a lightweight protocol, broadcast communication support, and support for message priorities and error handling [3], [4]. Other examples of current in-vehicle networks include Local Interconnect Network (LIN) and Media Oriented Systems Transport (MOST).

Recent advances in the automotive industry, such as the introduction of x-by-wire applications (throttle-by-wire, steer-by-wire, etc.), have led to an increase in the complexity of automotive applications, resulting in a huge demand for an efficient and reliable in-vehicle communication system to satisfy the timing constraints of all the critical applications. This goal is difficult to achieve using the existing CAN bus, which suffers from limited bandwidth and lack of timing determinism. As a result, researchers have been actively exploring other automotive communication solutions.

FlexRay has emerged in recent years as a potential solution that overcomes the limitations of the existing CAN protocol and offers added flexibility, higher data rates (at least 10× higher compared to CAN [7]), and better timing determinism. FlexRay supports both time-triggered and event-triggered transmissions. A major challenge with time-triggered transmissions is jitter, which is a delay-induced deviation from the actual periodicity of the message. There are various causes of jitter such as queuing of messages, delay in execution of tasks, network congestion, noise, and other external disturbances. In this work, we primarily focus on one of the more important sources of jitter: delay in execution of tasks in ECUs. Jitter must be addressed while designing schedules for time critical automotive applications as failure to do so can severely affect system performance and also be catastrophic in some cases (e.g., when the airbag deployment signal from the impact sensor to inflation module gets delayed due to jitter). This creates a requirement for an effective jitter handling mechanism that can be applied when designing and enforcing the schedules for time-critical automotive applications.

In this paper, we propose a novel scheduling framework called JAMS to handle both jitter effected time-triggered messages and high priority event-triggered messages in a FlexRay based automotive system. JAMS combines design time schedule optimization with a runtime jitter handling mechanism, to minimize the impact of jitter in the FlexRay network. Our novel contributions in this paper are:

- We propose a frame packing technique that packs the different signals from an ECU into messages;
- We develop a heuristic approach for the synthesis of design time schedules for the FlexRay-based system;
- We introduce a runtime scheduler that opportunistically packs the jitter affected time-triggered and high priority event-triggered messages in the FlexRay static segment slots;
- We compare our JAMS framework with the best-known prior works in the area and demonstrate its scalability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
NOCS '17, October 19–20, 2017, Seoul, Republic of Korea
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4984-0/17/10…$15.00
https://doi.org/10.1145/3130218.3130234

2. FLEXRAY OVERVIEW

FlexRay is an in-vehicle communication protocol designed for the next generation x-by-wire automotive applications. It supports both time-triggered and event-triggered transmissions. Figure 1 shows the structure of the FlexRay protocol. According to the FlexRay specification [5], a communication cycle is one complete instance of a communication structure that repeats periodically. Each communication cycle consists of a mandatory static segment, an optional dynamic segment, an optional symbol window, and a mandatory network idle time block.

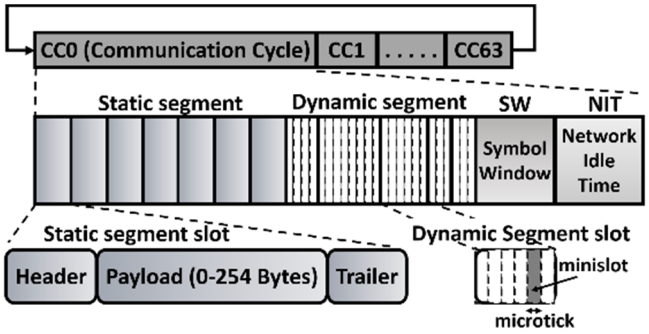

Figure 1. Structure of FlexRay protocol

The static segment in FlexRay consists of multiple equal sized slots called static segment slots that are used to transmit time-triggered and critical messages. The static segment enforces a Time Division Multiple Access (TDMA) media access scheme for the transmission of time-triggered messages, which results in a repetition of the schedule periodically. In this TDMA scheme, each ECU is assigned a particular static segment slot and cycle number to transmit its messages, which guarantees message delivery and time determinism. Each static segment slot incorporates one FlexRay frame, which has three segments: header, payload, and trailer. The *header* segment is 5-bytes long and consists of status bits, frame ID (FID), payload length, header cyclic redundancy check (CRC), and cycle count. The *payload* segment consists of actual data that has to be transmitted and is up to 127 words (254 bytes) long. The *trailer* segment consists of three 8-bit CRC fields to detect errors.

The dynamic segment consists of variable size slots called dynamic segment slots that are used to transmit event-triggered and low priority messages. A dynamic segment slot consists of a variable number of minislots (Figure 1), where each minislot is one microtick (usually 1 µs) long. The dynamic segment enforces a Flexible Time Division Multiple Access (FTDMA) media access scheme where ECUs are assigned minislots according to their priorities. If an ECU is selected to transmit a message, then it is assigned the required number of minislots depending on the size of the message, and hence the length of a dynamic segment slot can vary in the dynamic segment (Figure 1). During a transmission, all the other ECUs have to wait until the one that is transmitting finishes. If an ECU chooses not to transmit, then that ECU is assigned only one minislot and the next ECU is assigned the subsequent minislot. The symbol window (SW) is used for network maintenance and signaling for the starting of the first communication cycle, while the network idle time (NIT) is used to maintain synchronization between ECUs (Figure 1).

Inside a FlexRay ECU, messages are generated by a host processor and sent to the Communication Host Interface (CHI) where the message data is packed into FlexRay frames. Each frame has a unique frame ID (FID) that is equal to the slot ID in which the frame is transmitted [5]. CHI sends the qualified FlexRay frames to the Protocol Engine (PE), which transmits the frames on to a physical FlexRay bus, as shown in figure 2. A FlexRay frame is considered "qualified" when the message data is available at the CHI before the beginning of the allocated slot. Otherwise, a NULL frame is sent (by setting a bit in the header segment of the FlexRay frame and setting all the data bytes in the payload to zero). Jitter is one of the major reasons for delay in the availability of message data at the CHI. Hence in this paper we focus on a novel scheduling framework to overcome the delays and performance loss due to jitter.

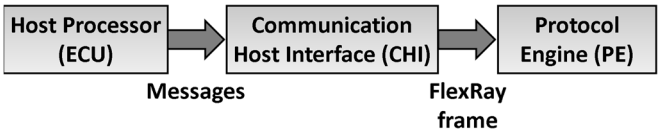

Figure 2. Message generation and transmission delay

3. BACKGROUND AND RELATED WORK

Many prior works in the area of FlexRay have focused on message scheduling of both time-triggered and event-triggered messages. The goal of these works is to generate message schedules by optimizing parameters such as bandwidth, number of static segment slots used, response time, etc., under strict timing constraints.

One of the popular techniques to maximize bandwidth utilization is using frame packing, where multiple signals are packed into messages. In [6] two frame packing algorithms were proposed to optimize bandwidth utilization. The authors in [7] proposed an Integer Linear Programming (ILP) formulation to solve the frame packing problem which requires multiple iterations with ILP to find the optimal solution. A Constraint Logic Programming (CLP) formulation and heuristic were presented for reliability-aware frame packing in [8], which may require multiple re-transmissions of the packed frames to meet reliability requirements. In [9], the frame packing problem is treated as a one-dimension allocation problem and an ILP formulation and a heuristic approach were proposed. The above-mentioned techniques have a tradeoff between optimizing bandwidth utilization and minimizing the time to generate optimal frame packing. Our proposed frame packing technique in this paper uses a fast heuristic approach to generate near optimal results.

In case of automotive applications, most of the parameters such as period, worst-case execution time, deadline etc., are known during the design time, which facilitates the synthesis of highly optimized design-time schedules. Several works explore the design-time scheduling of the static segment in FlexRay. One of the major objectives is to minimize the number of slots allocated to ensure future extensibility of the system while maximizing bandwidth utilization. An ILP formulation with an objective to minimize the number of slots allocated was proposed in [10], considering both message and task scheduling. This was later extended in [12] by including support for multiple real time operating systems and using ILP reduction techniques. In [11], the message scheduling problem is transformed into a two dimensional bin-packing problem and an ILP formulation and a heuristic approach were suggested for minimizing the number of allocated slots. The authors in [14] and [15] proposed a CLP and ILP formulation respectively for jointly solving the problem of task and message scheduling in FlexRay systems. A set of algorithms were proposed in [17] that enable scheduling of event-triggered messages in time-triggered communication slots using a virtual communication layer. A few other works solve the same problem with heuristics and variants of ILP and CLP [8], [13], [16], [18], [19]. *All of the above works lack jitter awareness, which makes them unreliable for use in real-time scenarios where jitter can significantly impact scheduling decisions.*

Jitter in FlexRay based systems has largely been ignored and there is limited literature on the topic. In [7] a jitter minimization technique using an ILP formulation is proposed. In [20], the frequency of message transmission is changed for those messages that are likely to be affected by jitter, to minimize message response time. However, in both [7] and [20], it is assumed that the jitter value and number of messages that are affected by jitter are known at design time, which is unlikely in real world scenarios. As random jitter can affect any message in the system, there is a need for a jitter handling mechanism that can handle jitter more comprehensively at runtime. *In this paper, we propose a hybrid framework that combines design time schedule*

optimization with runtime jitter handling, to minimize the impact of jitter in FlexRay.

4. PROBLEM DEFINITION

We consider a general scenario with multiple ECUs running various time critical automotive applications that are connected using a FlexRay bus. Executing an application may result in the generation of signals at an ECU, which may be required for another ECU executing a different application. These signals are packed into messages and transmitted as FlexRay frames on the bus. In a typical automotive system, messages can be categorized as: (a) time-triggered or periodic messages, and (b) event-triggered or aperiodic messages. Time-triggered messages are transmitted in the static segment of the FlexRay cycle while the dynamic segment is used for transmitting event-triggered messages. Every ECU or node in the system is capable of transmitting both types of messages in addition to the dedicated connection to different sensors and actuators (henceforth the terms ECU and node are used interchangeably). In this work, we primarily focus on the challenging problem of scheduling time-triggered messages and high priority event-triggered messages in the FlexRay static segment and ignore the scheduling of low-priority event-triggered messages in the FlexRay dynamic segment.

We consider an automotive system with the following inputs:

- $\mathcal{N}$ represents the set of nodes, where $\mathcal{N} = \{1, 2, 3, \ldots, N\}$;
- For each node n $\in \mathcal{N}$, $S^n = \{s_1^n, s_2^n, \ldots, s_{K_n}^n\}$ denotes the set of signals transmitted from that node and K_n represents the maximum number of signals in node n;
- Every signal $s_i^n \in S^n$, (i = 1, 2 …, K_n) is characterized by the tuple $\{\overline{p}_i^n, \overline{d}_i^n, \overline{b}_i^n\}$, where $\overline{p}_i^n, \overline{d}_i^n, \overline{b}_i^n$ denote the period, deadline, and data size (in bytes) of the signal s_i^n respectively;
- After frame packing, every node maintains a set of messages $M^n = \{m_1^n, m_2^n, \ldots, m_{R_n}^n\}$ in which every message $m_j^n \in M^n$, (j = 1, 2, …, R_n) is characterized by the tuple $\{p_j^n, d_j^n, b_j^n\}$, where p_j^n, d_j^n, b_j^n denotes the period, deadline and data size (in bytes) of the message m_j^n respectively;

We assume the following definitions:

- *Slot number or Slot identifier (slot ID)*: A number used to identify a specific slot within a communication cycle;
- *Cycle number*: A number used to identify a specific communication cycle in the schedule;
- To transmit a message m_j^n on the FlexRay bus, it needs to be allocated a slot ID $sl \in \{1, 2, \ldots, N_{ss}\}$ and a cycle number bc $\in \{0, 1, \ldots, C_{fx}\}$ where N_{ss} and C_{fx} are the total number of static segment slots in a cycle and total number of cycles, respectively. This is referred to as *message-to-slot assignment*.
- If a message m_j^n is assigned to a particular slot and a cycle then the source node n of that message should be allocated ownership of that slot. This is known as *ECU-to-slot assignment*.

Problem Objective: For the above inputs, the goal of our work is to: *(i)* maximize the bandwidth utilization on the Flexray bus with frame packing, *(ii)* find an optimal design-time schedule (message-to-slot assignment, ECU-to-slot assignment) for the time-triggered messages without violating timing constraints, *(iii)* minimize the effect of jitter on time-triggered messages and high priority event-triggered messages.

5. JAMS FRAMEWORK OVERVIEW

We propose the novel JAMS framework that enables jitter-aware scheduling of time-triggered messages and collocation of high priority event-triggered messages in the static segment in a FlexRay-based automotive system. Figure 3 shows an overview of the proposed framework. At design time, JAMS packs different signals into messages during frame packing, and then synthesizes a schedule using a design time scheduler. These frame packing and scheduling phases at design time only consider time-triggered messages, as the arrival pattern of event-triggered messages is not known at design time. At runtime, JAMS facilitates the handling of both jitter affected time-triggered messages and high priority event-triggered messages using a multi-level feedback queue (MLFQ). The runtime scheduler in JAMS packs these messages into the FlexRay static segment slots depending on the available slack using the design time schedule and the output of MLFQ. Each of these stages are discussed in detail in the next four subsections.

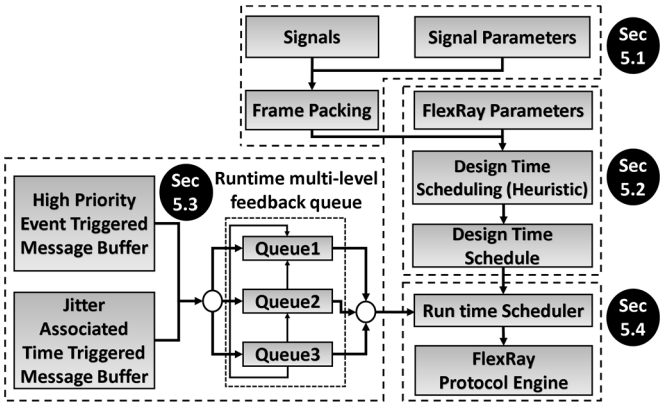

Figure 3. Overview of JAMS framework

5.1 Design Time Frame Packing

Frame packing is the first step in the JAMS framework where multiple periodic signals in a node are grouped together into messages. This step helps in maximizing bandwidth utilization, which not only improves system performance but also enhances the extensibility of the system by utilizing fewer slots than without frame packing. As discussed in section III, several works have proposed techniques for solving the frame-packing problem. However, many of these efforts lack scalability. In order to overcome these shortcomings, we propose a scalable frame packing heuristic to maximize bandwidth utilization in a greedy manner.

Algorithm 1: Greedy frame packing

Inputs: Set of nodes (N), Set of time-triggered (periodic) signals in each node (S^n), static segment slot size (b_{slot}), framing overhead (Of)

```
for each node n in N do
    Sort S^n in decreasing order of data size
    M^n = { }
    for each signal s_i^n in S^n do
        current_message = { }
        utilization = 0
        if signal s_i^n is not packed then
            Push s_i^n → current_message list
            utilization += b̄_i^n
            for each signal s_j^n in {S^n − s_i^n} do
                if p̄_j^n == p̄_i^n then
                    if signal s_j^n is not packed then
                        if utilization + b̄_j^n + O_f < b_slot then
                            Push s_j^n → current_message list
                            utilization += b̄_j^n
                        end if
                    end if
                end if
            end for
        end if
        if current_message ≠ { } then
            Add current_message list to M^n
        end if
    end for
    Add M^n to M
end for
```

Output: Set of messages in each node

Algorithm 1 shows the pseudo code of the proposed heuristic where the inputs are: a set of time-triggered (periodic) signals for each node (S^n), static segment slot size (b_{slot}), and the framing overhead (O_f; calculated from [7]). According to the FlexRay protocol specification [5], any slot in a given FlexRay cycle can be assigned to at most one node, which restricts the packing of signals from different nodes into the same message. This makes the frame packing problem solvable independently for different nodes as shown in step **1**. In addition to this, signals with the same periods and from the same node are prioritized for being grouped together, to minimize re-transmissions, which in turn minimizes the number of slots used. For any give node n ∈ ℕ, the algorithm starts with sorting the set of signals S^n in the decreasing order of bandwidth utilization (step **2**). In step **3,** an empty set is created for storing the set of messages for that node and all the signals in S^n are considered for packing as shown in step **4**. In steps **5-19**, every unpacked signal is packed with the other signals of the same period and the same node until utilization constraints in step **13** are violated. The signal that caused the violation of utilization constraints is added to a new message after adding the previously computed message to M^n. This is repeated until all the messages in that node are packed. At the end, M^n computed for each node is added to another set M, where M is a superset consisting of the set of messages in each node. The resulting messages will have their period be the same as the signal period and their deadline will be equal to the lowest signal deadline packed in that message. In this work, all the messages have deadlines equal to their periods. The output of the algorithm is an optimal packing of signals into messages for each node.

5.2 Design Time Scheduling

In this subsection, we present a design time scheduling approach for the previously generated time-triggered messages. As discussed in section IV, for the transmission of any message over FlexRay, a slot ID and cycle number needs to be assigned for both the message and the source node of that message. A design time schedule consists of such an assignment of slot IDs and cycle numbers to the messages in the system. In creating this schedule, we aim to minimize the number of assigned static segment slots for efficient bandwidth utilization. In addition, we also take advantage of the cycle multiplexing in FlexRay 3.0.1 in which multiple nodes can be assigned slots with the same slot ID in different communication cycles, which is not possible in FlexRay 2.1A. This helps to maximize the static segment utilization while using only minimal number of slots [11].

The fundamental basis for our proposed heuristic are the concepts of message repetition and the slot ID utilization in a FlexRay based system. As the time-triggered messages are periodic, their schedules also repeat after a certain period. For any time-triggered message in FlexRay, message repetition is the ratio of message period to the cycle time of the FlexRay as shown in equation (1):

$$rm_j^n = \frac{p_j^n}{C_{fx}} \quad (1)$$

This number is usually an integer value as the FlexRay cycle time is chosen to be the greatest common divisor of all the message periods in the system. In addition, from [20] it is evident that any time-triggered message that is assigned a particular slot ID will use up $(1/rm_j^n)$ of the available slots having that slot ID. Algorithm 2 shows how this metric is used in JAMS to assign slots to the messages. After initializing variables in step **1**, all the time-triggered messages are sorted in the increasing order of message periods. For each message in the list, we check if adding the message to the current slot ID does not violate the utilization constraint (step **4**). If the maximum utilization is exceeded, then a new slot ID is allocated to that message and the utilizations and slot variables are updated (steps **4-9**). After a slot ID has been assigned to a message, cycle numbers are determined using steps **10** and, **11**. The lowest cycle number c is chosen as the first communication cycle for that message and all the $c + (i * rm_j^n)$ cycles (for i ∈ N_0) away from the chosen cycle are added to the list of cycles for that message. Moreover, all the cycle numbers that were added to the message are removed from the *cycleList*. This results in a slot and cycle assignment to all the time-triggered messages in the system and their corresponding source nodes.

Algorithm 2: JAMS design time scheduling

Inputs: Set of all time-triggered messages in the system (M = {M_1, M_2, …, M_N})

1: **Initialize**: $util_{cur}$, $util_{prev}$= 0; $slot_{init}$, $slot_{cur}$ = 1; cycleList= {0,1,…, 63}
2: **Sort** M in the increasing order of message periods
3: **for** each message m_j^n in M **do**
4: **if** ($util_{prev}$ + $1/rm_j^n$) > 1 **then**
5: $util_{cur}$ = $util_{prev}$ = 0; $slot_{cur}$ + = 1; cycleList = {0,1,…., 63}
6: **end if**
7: $util_{cur}$ + = $1/rm_j^n$
8: $util_{prev}$ = $util_{cur}$
9: $slot_{m_j^n} \leftarrow slot_{cur}$
10: **Push** c , c +$(i * rm_j^n)$ → list of cycles of m_j^n for i ∈ N_0
11: **Remove** c , c +$(i * rm_j^n)$ from cycleList for i ∈ N_0
12: **Assign** slot ownership($slot_{m_j^n}$, list of cycles of m_j^n) to node n
13: **end for**

Output: slot ID ($slot_{m_j^n}$) & set of communication cycles (list of cycles) for each time-triggered message m_j^n, slot ownership of each node n.

5.3 Runtime Multi-level Feedback Queue

The schedule generated by the design time scheduler will only guarantee latencies for time-triggered messages under ideal conditions. In realistic scenarios, various disturbances may interfere with the regular operation on the FlexRay bus. One of the major disturbances that can severely impact the performance of schedules is jitter, which is a serious threat for safety critical systems. Hence, we focus on handling jitter at runtime using a runtime scheduler that re-schedules jitter affected time-triggered messages using the input from the design time schedule and the output of a Multi-Level Feedback Queue (MLFQ). Similarly, the high priority event-triggered messages are also treated as jitter affected time-triggered messages during the runtime scheduling, so that they can be scheduled in a timely manner as part of the FlexRay static segment.

The MLFQ consists of two or more queues that have different priorities and are capable of exchanging messages between different levels using feedback connections (figure 3). The number of MLFQ queues defines the number of levels; each level queue can have a different prioritization scheme and scheduling policy than other queues.

We considered an MLFQ consisting of three level queues as shown in figure 3, with queue 1 (Q1) having the highest priority followed by queue 2 (Q2) and queue 3 (Q3) with lower priorities. In addition to prioritization between different level queues, we set up priorities between different types of messages and within the messages of the same type. We prioritize time-triggered messages over event-triggered ones and all other types of messages. Moreover, within the time-triggered messages, we adapt a Rate Monotonic (RM) policy to prioritize messages with high frequency of occurrence. In case of a tie, priorities are assigned using a First Come First Serve (FCFS) strategy. Event-triggered messages inherit the priority of their generating node. In cases of multiple event-triggered messages from the same node, an Earliest Deadline First (EDF) scheme is employed to prioritize messages. These static priorities of the messages are used to reorder the queues and promote messages to upper level queues. In addition, there are two separate buffers that are used to handle jitter affected time-triggered messages and high priority event-triggered messages, which are later fed to the MLFQ (as shown in figure 3).

The operation of the MLFQ is depicted in figure 4. We begin by checking the time-triggered message buffer for jitter-affected messages. If a time-triggered (TT) message is available, the 'load TT message' function is executed which checks for a vacancy in queues in the order Q1, Q2, and Q3 and, stores the TT message in the first

available queue. If the TT message buffer is empty and an event-triggered (ET) message is available in the ET message buffer, the 'load ET message' function is executed which checks for the vacancy in queues in the order Q2, Q3 and Q1 and, stores the ET message in the first available queue. In either case, when all the three queues are full, the message is held in the corresponding buffer and the same function is executed in the next clock cycle. Whenever there are no messages available in both the buffers and, the reorder queue function is not executed in the preceding clock cycle, messages in the queues are reordered in the order of their priorities, by executing the 'reorder queues' function. Otherwise, the queues are checked in the order Q1, Q2 and Q3 by executing the 'POP queue' function. The conditions for popping a queue are discussed in detail in the next subsection.

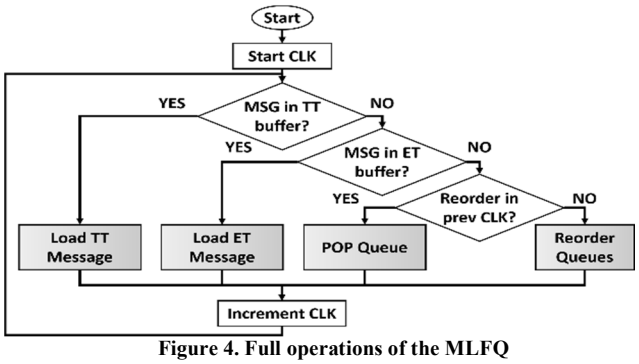

Figure 4. Full operations of the MLFQ

5.4 Runtime Scheduler

After handling the jitter-affected messages in the MLFQ as mentioned in the previous subsection, the next task is to re-schedule them and meet their deadline constraints. To accomplish this, we introduce a runtime scheduler that handles multiple inputs coming from the output of the MLFQ, and the design time generated schedule. In addition to these, the runtime scheduler also has information on the available slack in each of the static segment slots from the design time generated schedule. Thus, whenever there is a jitter-affected message available at the MLFQ, the runtime scheduler checks for two different conditions in the next incoming slot: *(i)* whether there is sufficient slack in the incoming slot to accommodate the available jitter-affected message and, *(ii)* whether the incoming slot is owned by the sender node of the jitter-affected message. When these two conditions are satisfied, the jitter-affected message from MLFQ is collocated with the jitter unaffected message in the incoming slot, as shown in figure 5. Similarly, when a high priority event-triggered message is available at the MLFQ, the abovementioned conditions are checked and the high priority event-triggered message is treated as the jitter affected time-triggered message. This results in packing of two different message data in to the payload segment of one frame, which unfortunately leads to two major challenges. Firstly, there is a requirement for a mechanism at the receiver to decode the payload and distinguish between the two messages. Secondly, the implicit addressing scheme of TDMA is lost because of combining two different messages, as the receiving nodes will not be able to identify to which specific node the message is meant for.

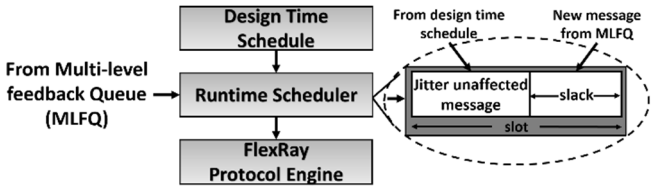

Figure 5. Packing of jitter affected messages during runtime

To overcome the abovementioned challenges, we propose a segmentation and addressing scheme to differentiate multiple messages packed into the same frame. The scheme introduces two additional segments in the payload segment of the FlexRay frame (figure 6). The first is called AI or Append Indicator segment, which is the first segment in the payload. It is a 1-bit field indicating whether a message is appended in the frame. The second is a custom header segment, which consists of two fields: a type field and a length field. The type is a 15-bit field used to specify different message types. It consists of one bit each for payload preamble indicator, null frame indicator, sync frame indicator and startup frame indicator which are defined in [5] and an 11 bit frame ID (FID) field for specifying the FID of the jitter affected message. The length field is 8-bit long and specifies the data length of the jitter affected message in bytes. The length field in the custom header along with the payload length field in the frame header is used to find the start byte of the jitter affected message in the payload segment. The regular operation of the FlexRay protocol is not altered in any way by implementing these changes.

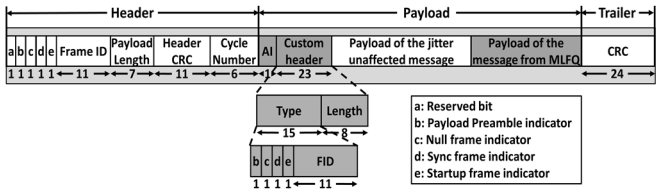

Figure 6. Updated frame format of the FlexRay frame using the proposed segmentation and addressing scheme (sizes of individual segments are shown in bits). The parts of the frame highlighted in gray represent our modifications.

6. EXPERIMENTS

6.1 Experimental Setup

In this section, we evaluate the performance of the JAMS framework by comparing it with Grouping based Message Scheduling (GMSC [7]), Optimal Message Scheduling with Jitter minimization (OMSC-JM [20]), Optimal Message Scheduling with FID minimization, (OMSC-FM [20]), and Policy based Message Scheduling (PMSC [17]). GMSC [7] begins with frame packing of signals using an ILP approach and tries to group these messages into selected slots. After generating the corresponding groups, the allocation algorithm proposed in [20] is used to generate the slot IDs and communication cycles for which the response times are calculated. Based on the available knowledge of worst-case jitter, slot allocations are modified. PMSC [17] uses a priority based runtime scheduler, which supports preemption based on the message arrival time and priority. Messages are scheduled based on the slot assignment generated using heuristics. OMSC-JM [20] and OMSC-FM [20] use the frame packing technique in [7] and alter the message repetition to minimize the effect of jitter. OMSC-JM and OMSC-FM differ by the weights associated with the objective function in the optimization problem. OMSC-JM tries to minimize the effect of jitter by allocating more slots and frequent message transmissions while OMSC-FM aims at minimizing the number of allocated slots.

To evaluate JAMS and compare against the above-mentioned prior works, a set of test cases was created using different combinations of the number of nodes, number of signals in the system, and the periods of the signals. For our initial set of experiments, we selected a test case configuration representing a FlexRay 3.0.1 based system consisting of 4 nodes. The nodes exchanged 33 different signals with periods varying from 5 ms to 45 ms with data sizes ranging from 8 to 32 bytes. The FlexRay system considered for all the experiments has a cycle duration of 5ms (t_{dc}) with 62 static segment slots (N_{ss}), with a slot size of 42 bytes (b_{slot}) and 64 communication cycles (C_{fx}). We randomly sample jitter values from the range of 1–8 ms to modify the arrival times of the messages originating from a randomly selected jitter-affected node. The messages with a deadline under 25ms are considered as high priority messages and the others as low priority

messages. All the simulations are run on an Intel Core i7 3.6GHz server with 16 GB RAM.

6.2 Response Time Analysis

Figures 7(a)-(b) illustrate the average response times of the high priority and lower priority messages, respectively. The confidence interval on each bar represents the minimum and maximum average response time of the message (with the deadline on x-axis) achieved using the corresponding technique. The dashed horizontal lines show the four different message deadlines in each figure. From figure 7(a), it is evident that under many cases, GMSC exceeds the deadline of the messages in the presence of jitter. This is because of its lack of ability to handle random jitter. OMSC-FM results in high response times in the presence of jitter as this technique allocates a minimal number of slots resulting in long delays between any two instances of a message transmission (e.g., when there is a delay in the message arrival, the jitter-affected message has to wait for a longer duration to be transmitted in the next allocated slot). Moreover, OMSC-FM does not consider deadline constraints. OMSC-JM has relatively better response times (except in one case, where it misses a deadline because of high jitter) as it assigns additional slots to the nodes and thus has more number of available slots for the transmission of jitter-affected messages. In PMSC, jitter has a strong impact on the high priority messages (figure 7(a)), because of the type of frame packing involved. PMSC aims to use the entire static segment slot by packing the signals that are larger than the slot size and uses EDF-based preemption at the beginning of each slot. In the presence of jitter for high priority messages, the arrival of these messages are delayed, causing the node to wait for the next transmitting slot to preempt existing transmissions of low priority messages. This delay along with jitter can sometimes exceed the message deadline and may lead to missed deadlines, as shown in figure 7(a).

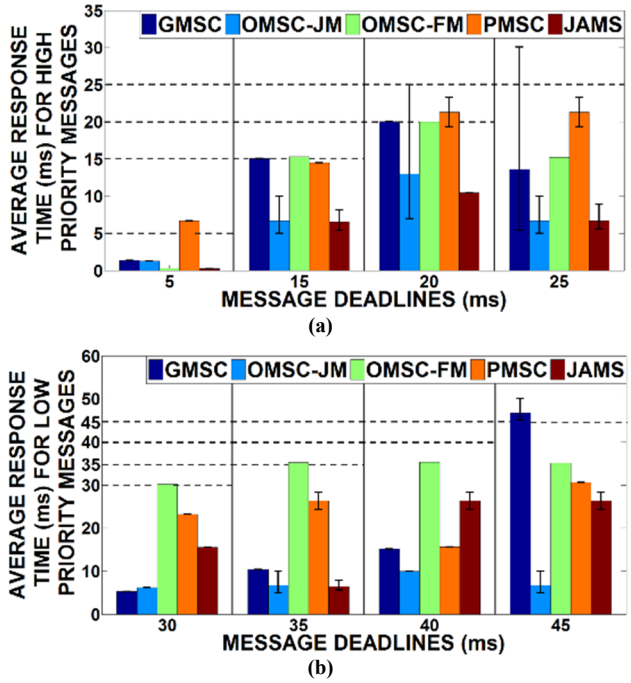

Figure 7. Message deadlines vs average response time for (a) high priority messages and (b) low priority messages; for GMSC [7], OMSC-JM [20], OMSC-FM [20], PMSC [17], and our JAMS framework.

It is evident that our JAMS framework outperforms all the other techniques and achieves lower response time for all the high priority messages (figure 7(a)). In addition, JAMS is also able to minimize the average response time of the low priority messages while meeting their deadline constraints, as shown in figure 7(b).

6.3 Scalability Analysis

To evaluate the effectiveness of JAMS for system configurations with varying complexities, we analyzed it against the prior works by selecting different combinations of number of nodes and number of signals. Figures 8(a)-(c) plot the average response times of all the signals (y-axis) of the jitter-affected node under low, medium and high jitter respectively (jitter values varied from 1–3 ms for low, 3 – 5 ms for medium, and 6 – 8 ms for high), for different system configurations {p, q} where p denotes the number of nodes and q is the number of signals (x-axis). The number on the top of each bar indicates the number of signals that missed the deadline in that configuration. It can be observed in figures 8(a)-(c) that the average response times of all the signals in the jitter-affected node is less in almost all the cases when using JAMS. JAMS does have an overhead in the average response time compared to the best average response time (using OMSC-JM [20]) for the first configuration with 3 nodes and 21 signals, especially for messages with very low jitter.

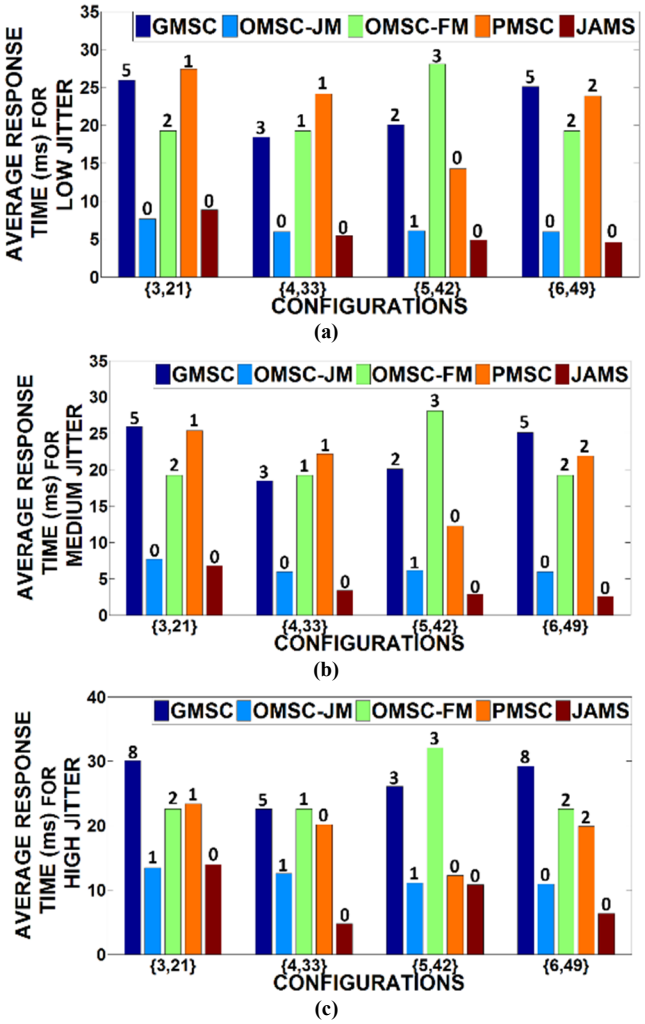

Figure 8. Average response time of all signals for different configurations (with number of missed deadlines shown on top of the bars) under (a) low, (b) medium and, (c) high jitter conditions, for GMSC [7], OMSC-JM [20], OMSC-FM [20], PMSC [17], and our proposed JAMS framework.

6.4 Slot Allocation Analysis

Figure 9 illustrates the number of slots allocated for each configuration evaluated in the previous subsection, across the various frameworks. It can be seen that a higher number of slots are allocated with OMSC-JM [20] than JAMS. This high allocation facilitates the frequent transmission of jitter-effected messages, which can lower response times. However, even with a slight advantage in response

time, OMSC-JM [20] still misses the deadlines for high jitter-affected messages, unlike JAMS which has no deadline misses across all the configurations. Also from figure 9, it is evident that OMSC-FM [20] has a lower number of slots allocated in all the configurations. But figures 8(a)-(c) illustrate that the average response time of the signals impacted by jitter is higher using OMSC-FM [20] as compared to JAMS because of the fewer availability of slots. In addition, OMSC-FM [20] suffers from multiple deadline misses in comparison to JAMS. From the results related to OMSC-JM and OMSC-FM in figures 8(a)-8(c) and 9, it is important to observe that there exists a tradeoff between the number of allocated slots and the average response time. JAMS balances this tradeoff quite well. Thus with a minimal overhead in the number of allocated slots, JAMS surpasses all the other techniques with no deadline misses and low response times.

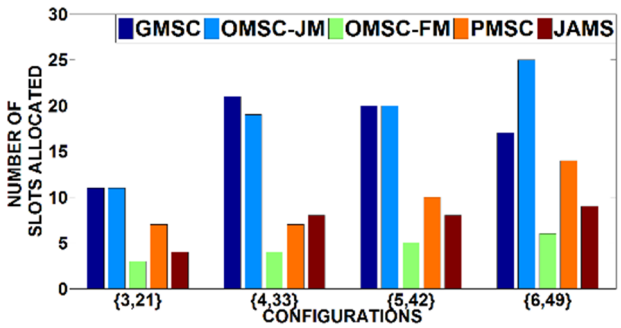

Figure 9. Number of allocated slots for each configuration

7. CONCLUSION

We presented a novel scheduling framework for effectively handling jitter-affected time triggered and high-priority event-triggered messages in a FlexRay based automotive system. Our framework utilizes both design time and run time scheduling to mitigate the effect of jitter on the system. Our proposed JAMS framework has several advantages compared to other scheduling approaches from prior work, in terms of both response times and slot allocation. Our experimental analysis with the proposed framework indicates that it is highly scalable and outperforms the best-known prior works in the area. JAMS improves the response time by 25.3% on average and up to 41% compared to the best-known prior work (OMSC-JM). Our approach can also be extended to other time-triggered protocols with minimal changes.

REFERENCES

[1] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan and A. Sangiovanni-Vincentelli, "Period Optimization for Hard Real-time Distributed Automotive Systems", in Proc. DAC, 2007.

[2] CAN Specifications version 2.0, Robert Bosch gmbh, 1991.

[3] SAE Automotive Engineering International, July 2016.

[4] K. Tindell, A. Burns and A.J. Wellings, "Calculating Controller Area Network (CAN) Message Response Time", in CEP, 1995.

[5] FlexRay. FlexRay Communications System Protocol Specification, ver.3.0.1. [Online]. Available: http://www.flexray.com

[6] R. Saket, N. Navet, "Frame Packing Algorithms for Automotive Applications", in JEC, 2006.

[7] K. Schmidt, E.G. Schmidt, "Message Scheduling for the FlexRay Protocol: The Static Segment", in IEEE TVT, 2009.

[8] B. Tanasa, U.D. Bordoloi, P. Eles and Z. Peng, "Reliability-Aware Frame Packing for the static segment of FlexRay", in Proc. EMSOFT, 2011.

[9] M. Kang, K. Park and M.K. Jeong, "Frame Packing for Minimizing the Bandwidth Consumption of the FlexRay Static Segment", in IEEE TIE, 2013.

[10] H. Zeng, W. Zheng, M. Di Natale, A. Ghosal, P. Giusto and A. Sangiovanni-Vincentelli, "Scheduling the FlexRay bus using optimization techniques", in Proc. DAC, 2009.

[11] M. Lukasiewycz, M. Glaß, J. Teich and P. Milbredt, "Flexray Schedule Optimization of the Static Segment", in Proc. CODES+ISSS, 2009.

[12] H.Zeng, M. Di Natale, A. Ghosal and A. Sangiovanni-Vincentelli, "Schedule Optimization of Time-Triggered Systems Communicating Over the FlexRay Static Segment", in IEEE TII, 2011.

[13] M. Grenier, L. Havet and N. Navet, "Configuring the communication on FlexRay- the case of the static segment", in Proc. ERTS, 2008.

[14] Z. Sun, H. Li, M. Yao and N. Li, "Scheduling Optimization Techniques for FlexRay Using Constraint-Programming", in Proc. CPSCom, 2010.

[15] M. Lukasiewycz, R. Schneider, D. Goswami and S. Chakraborty, "Modular Scheduling of Distributed Heterogeneous Time-Triggered Automotive Systems", in ASP-DAC, 2012.

[16] D. Goswami, M. Lukasiewycz, R. Schneider and S. Chakraborty, "Time-triggered implementations of mixed- criticality automotive software", in Proc. DATE, 2012.

[17] P. Mundhenk, F. Sagstetter, S. Steinhorst, M. Lukasiewycz and S. Chakraborty, "Policy-based Message Scheduling Using FlexRay", in Proc. CODES+ISSS, 2014.

[18] B. Tanasa, U.D. Bordoloi, P. Eles and Z. Peng, "Scheduling for Fault-Tolerant Communication on the Static Segment of FlexRay", in Proc. RTSS, 2010.

[19] Lange, F. Vasques, P. Portugal and R.S. de Oliveira, "Guaranteeing Real-Time Message Deadlines In The FlexRay Static Segment Using a On-line Scheduling Approach", in WFCS, 2014.

[20] K. Schmidt, E.G. Schmidt, "Optimal Message Scheduling for the Static Segment of FlexRay", in Proc. VTC, 2010.