FPGA-based Testbed for Timing Behavior Evaluation of the Controller Area Network (CAN)

Tobias Ziermann, Alexander Butiu, Jürgen Teich, and Daniel Ziener
Hardware/Software Co-Design
Department of Computer Science
University of Erlangen-Nuremberg
Germany

***Abstract*—The Controller Area Network (CAN) is one of the most popular networks for industrial distributed embedded systems, particularly automotive systems. In these systems the timely transmission of data messages is a key requirement. In this paper, a physical testbed is presented that allows the evaluation of the timing behavior of CAN message transmissions. The use of FPGAs as processing nodes allows to accurately perform measurements without influencing the original system due to the inherent parallelism of programmable hardware. As a case study, the improvement of response times by using dynamic scheduling of messages is shown with the testbed. Furthermore, the testbed helps to identify time critical simulation parameters.**

I. Introduction

Today, one of the most popular bus-based protocols for distributed embedded systems is the Controller Area Network (CAN) [1], which was originally designed for automotive applications. The target applications are distributed control embedded systems based on the use of multiple computing nodes connected by a communication bus. A common characteristic of such applications is that the communication over the bus is triggered periodically and the amount of data exchanged is relatively small. However, the time between sending data from the source node to receiving it on the destination node, called *message response time*, is crucial and subject to real-time constraints.

In CAN, a message-oriented approach is chosen in the data link layer. Each data frame has an identifier that is unique for each node and defines the message priority by which the bus access is granted. The disadvantage of this access method is that messages with low identifiers can possibly be delayed very long. This problem increases with increasing utilization of the bus. To cope with that problem analytical methods are used to calculate the maximum delay that could occur in the worst case scenario [2], [3]. However, these upper bounds are often too pessimistic, because on the one hand the worst case doesn't occur due to not modeled constraints such as dependencies between messages. On the other hand for many applications the average response time is more important than the maximum response time. A more practical approach is to simulate the behavior by software and measure the message response times [4], [5], [6]. The main question we want to answer in this paper is: *How good does a bit-accurate simulation represent the real behavior of the physical CAN bus?*

978-1-4673-2921-7/12/$31.00

To compare the simulated to the real behavior, we build a physical testbed connecting several FPGAs by CAN. Using the parallelism of the FPGAs and the CAN communication structure, the response times of the messages are measured and recorded for later evaluation without influencing the actual behavior of the system. Using this testbed with real components helps to reveal problems that are not directly visible in the simulation. We were able to identify that the clock drift of the individual nodes that is present in real systems has a strong influence on the response times. Furthermore, the worst-case assumptions of the simulations lead in some cases to overestimation of up to factor two. Another advantage of a testbed is that it could be connected to a working CAN to analyze the response times of certain components when mixed with real communication traffic. The main contribution of this paper is that to the best of our knowledge this is the first physical setup of CAN where response times are measured.

The rest of the paper is organized as follows. In Section II, we introduce CAN and define the model we are using for message generation. Section III gives the details of the testbed. Section IV describes the case study of dynamic offset adaptation. Section V presents the results we obtained using the testbed in comparison to simulation. Conclusions and future work are given in Section VI.

II. Problem and Model Description

In this section, first the necessary basics of the CAN protocol are introduced. Then, the model we used for message generation and the applied performance metric is described.

A. CAN Basics

The CAN specification [1] defines the data link layer and roughly describes the physical layer of the ISO/OSI-Model. The physical setup consists of a resistor terminated

2-wire bus. Bits are represented by the Non-Return-To-Zero method, where a logical "0" (dominant state) is represented by a voltage difference and a logical "1" (recessive state) by equal voltage level. This bit representation results in a dominance of the logical "0". The length of a bit depends on the bit rate used, e.g., 1μs at 1Mbit/s. The mechanism of bit-stuffing is applied that inserts an opposing bit after five consecutive similar bits to ensure correct synchronization.

At the data link layer, a message-oriented approach is chosen. Each data frame of one application has its unique identifier. This identifier defines the message priority by which the bus access is granted. Bus arbitration is done by Carrier Sense Multiple Access with Bitwise Arbitration (CSMA/BA). The method of bitwise arbitration can be described as follows: Each node that would like to have access to the bus, starts sending its message as soon as the bus is idle for the time of three bits. Every sent bit is also watched. When the sent bit differs from the watched one, a message with higher priority is also sending currently and nodes with lower priorities stop transmission. After sending the identifier, only the message with the highest priority is left and has exclusive bus access.

B. System Model

The CAN system we target can be described by a set of nodes, i.e. electronic control units (ECUs), communicating over the bus. One or more tasks on each node initiate the communication, typically periodically, i.e. they release messages. However, while there is a common time reference for tasks on a single node, there is no common time reference for the overall system, thus the system is asynchronous.

In our model, we abstract the tasks running on the nodes by considering only the mechanism used to release messages called a stream. A stream s_i is characterized by a transmission time, by a period (time between any two consecutive messages generated by stream s_i) and an offset. The offset is relative to a global time reference. It can therefore drift over time, because the local time reference differs from the global one. The hyper period P is the least common multiple of all periods. Assuming a synchronous system, the schedule is finally periodic with the period equal to the hyper period. A message is a single release or CAN frame of the stream. The time between a message release and the start of its uninterrupted transfer over the bus is the response time of the message. We do not add the constant non-preemptive time to transfer the message to the response time, thus a zero response time is always the best possible case. This simplifies comparisons between different schedules.

In this paper, we consider different example systems called *scenarios*. A scenario consists of a set of streams that are assigned to a set of nodes. The scenarios used for our experiments consist of synthetic scenarios automatically generated by Netcarbench [7] and are typical for the automotive domain with a bus load that can be freely adjusted. For all scenarios, a bus speed of 125 kbit/s is assumed. Typical for practical implementations is that there are not too many different periods, generally 5 - 10 that are mostly multiples of each other. This results in a small hyper period. In the example scenarios, it is always 2 seconds, which is also the largest period used.

To compare our results, we use the rating function called the *average weighted worst case response time*, or simply AWW, originally proposed in [8], which enables us to differentiate the quality of different message scheduling schemes:

$$AWW(t) = \frac{1}{k} \cdot \sum_{i=1}^{k} \frac{WCRT_i(t-P,t)}{T_i} \qquad (1)$$

where $WCRT_i(t-P,t)$ is the measured worst case response time of the stream i that has period T_i in the last hyper period P and k is the total number of streams generating messages. The function $AWW(t)$ takes into account that the streams with small periods are more sensitive to large response times, because each WCRT is weighted with its corresponding period. Finally, the sum is divided by the number of streams to get the average rating per stream. This also allows to compare different scenarios with different numbers of streams.

III. Testbed Setup

In this section, first we give an overview of the the physical setup of the proposed testbed. Next, the details of the FPGA architecture are provided.

A. Hardware Setup

Figure 1 shows the setup with four FPGA nodes we used for our case study. However, the proposed testbed is designed to cope with any number of FPGA boards, only limited by the CAN protocol to 2032. In our setup, we used the Virtex-5 OpenSPARC Evaluation Platform, but any other FPGA board with an RS232 connector and two I/O connectors could be used. Each FPGA is representing one CAN node. Even though the area requirement for each node allows to implement several nodes on one FPGA, test results show that the performance results are the same to scheduling the identical streams on one node. This is due to the same used clock, which keeps the nodes on the same FPGA synchronized. In future work, several nodes could be emulated in one FPGA by using different clocks.

For the physical connection between the FPGA boards and the CAN bus, we used a standard 8-pin CAN transceiver [9]. We created a simple PCB that is plugged on the I/O pins of the board. It is connected by two standard I/O pins for receiving and transmitting the CAN bus signal. Additionally, a 5V power supply to generate the CAN signal is required, which is provided on the I/O connectors on our evaluation platform. The only external connection required by the FPGA boards are the two wires of the CAN bus.

It has to be noted, that it is necessary to share a common ground which is most easily established by using a third wire within the CAN bus cable.

Besides the FPGA boards, we are using a standard PC for two purposes: (1) A commercial CAN debugger is connected by USB to monitor the generated traffic during run-time. This allows to debug during the development phase, but also ensures correct and error free measurements. (2) One of the FPGA boards is connected to the PC using the RS232 protocol to gather the performance measurements. More details on that are provided in Section III-B3.

B. FPGA Modules

In the following, the design of the FPGA modules are described. As shown in Figure 2, a modular approach with simple interfaces is chosen to allow the easy replacement of any modules to research different behaviors in the future. The FPGA design is logical divided into three components: message generation, message transmission and performance evaluation. The first two components represent the system under test, while the purpose of the last component is to measure the performance of the system under test. The RS232 module is implemented on all FPGAs for simplicity reasons, but only board 4 is connected to a data logger. The two CAN controller are connected to one transceiver by a logical and for the output and a logical or for the input signal.

1) Message Generation: The *message generation* represents the heart of the system under test. In a real system, the main functions of the node would be performed here. In our testbed, its function is to generate the communication traffic. This could vary from a simple pattern generator up to execution of the real applications using dummy data. Modern FPGAs even allow to use soft core processors, so the message generation module could be a small processor. In that case, the scheduler depicted in Figure 2 could be a full operating system.

In the current setup, a static cyclic scheduler is used that issues dummy tasks which do nothing but transmitting messages. These tasks fulfill the properties of the streams described in Section II-B. According to the scenarios used for our tests the hyper period of the streams and therefore the period of our schedule is two seconds. A counter with a resolution of 8μs was chosen to carry out the schedule, which corresponds to the transmission time of one CAN bit. The embedded memory of the FPGA is used to store the information of the streams (identifier, next release time, period, length of the CAN message). Every counter tic, the stream memory is checked for a message to release. In case of a hit, the message is forwarded to the message transmission and the next release time is updated.

2) Message transmission: Two modules provide the functionality of the message transmission: The *message queue* and the *CAN controller*. The message queue stores the messages that are released and are waiting for transmission. Here again different queuing schemes could be tested. We chose a priority queue for our case study to ensure that the results of our dynamic scheduling algorithm are not distorted. The implemented message queue stores the waiting messages in the embedded memory. The priority schedule is enforced by scanning the whole memory with a maximum of 2032 entries every 8μs and picking the message with the highest priority.

The actual transmission of the message is initiated by feeding the message information (identifier, message length and data) to the CAN controller and enabling transmission. The CAN controller signals to the message queue whether it is ready for new transmissions and if messages were transmitted successfully, i.e., the arbitration was won. These transmitted messages can then be removed from the memory of the message queue. The function of the CAN controller module is to implement the CAN protocol. In our setup, we used our own developed CAN controller because it simplifies the response time measurement described in the following section. However, by adding some additional logic any CAN controller could be used. This could be a dedicated IC or an FPGA module like the CAN Protocol Controller from OpenCores [10].

3) Performance Evaluation: The performance evaluation component can be divided into one for measuring of the response time, and one for evaluation and transmission. The *response time measurement* module is closely connected to the unit under test. The release time of the message is signaled by the scheduler. This release time is stored in an embedded memory of the FPGA. The starting time of the message transmission needs to be known to calculate the response time. This is gathered by subtracting the transmission duration in bits, which is provided by our CAN controller, from the time of finishing the transmission. If the CAN controller doesn't provide the transmission duration, the start of transmission can be gathered by storing the start of transmission temporarily and validating it if the arbitration was successful.

The response times of all nodes are communicated by piggybacking the CAN messages. As soon as a response time calculation is finished the first four bytes of the next CAN message are used to transmit the response time. The 32 bits contain the corresponding identifier (11 bits) and the response time in μs with a maximum of two seconds (21 bits). So, the response time can be sent in all messages with a minimum length of four bytes. CAN messages that are smaller are extended to four bytes. The used scenarios and therefore typical automotive scenarios contain only very few messages (<20%) of smaller size, so this restriction of our approach is acceptable because it has a negligible influence on the measurements.

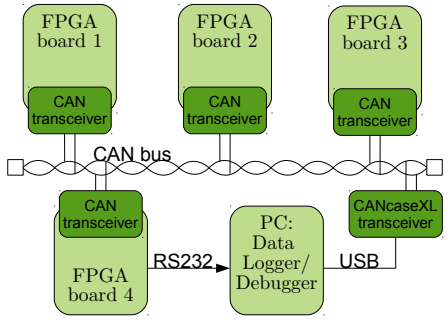

Figure 1. Testbed setup where four FPGA boards and one standard PC are connected by a CAN bus.

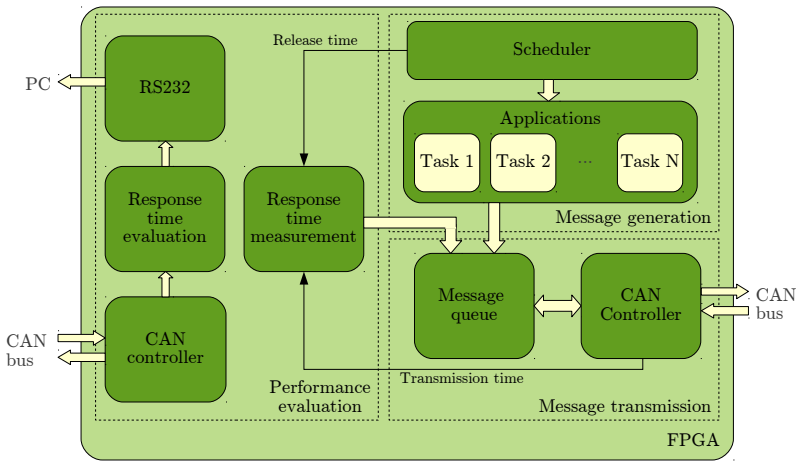

Figure 2. Schematic of the modules used in each testbed FPGA according to Figure 1.

Independent FPGA modules are used to evaluate the response times and transmit the results for offline evaluation. An additional CAN controller is used to receive the response times and transmit it to the *response time evaluation* module. We used the AWW function from Equation (1) to evaluate the response time. The maximum response time for each stream is stored in an embedded memory. Every hyper period P, i.e., two seconds, the AWW function is calculated and sent out by using the RS232 module to transmit it to a PC. Then, the response time memory is cleared to be ready for the next measurements. The PC stores the values in a file for visualization or later evaluation.

IV. Case Study

As an example application, the proposed testbed is used to test and validate the *dynamic offset adaptation algorithm* (DynOAA) introduced in [8]. It changes offsets over time following the change of the traffic on the CAN bus to reduce the AWW. In our testbed setup, the purely periodic scheduler is replaced by a module implementing the DynOAA that runs on each node independently according to Figure 1. An illustration of the operation of DynOAA for one stream is shown in Figure 3. In the upper part of the figure, on the top of the time line, the periodically released messages of the stream are indicated by small arrows. The larger arrows on the bottom of the time line indicate the instances when the points of adaptation. Each algorithm run consists of a traffic monitoring phase and a delay phase. During the monitoring phase, a list called *busy_idle_list* is created. An example of this list is shown in the lower part of Figure 3. It contains, for each time slot during the monitoring phase, an idle element if the bus is idle and a busy element if the bus is busy. A time slot has the length of the transmission time of one CAN bit. From the busy_idle_list, we can find the longest idle time $\alpha_{longest}$ and longest busy time $\beta_{longest}$, which are the maximum continuous intervals when the bus was idle or

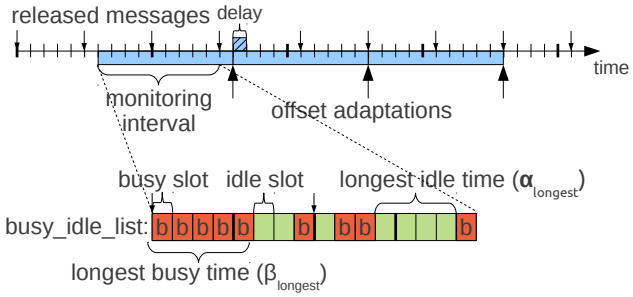

Figure 3. DynOAA illustration - timing diagram and *busy_idle_list* on a single node

busy, respectively. During the delay phase, the next message of the stream is then delayed, i.e., the offset is adjusted, so that a message in the next monitoring phase is released in the middle of $\alpha_{longest}$.

In distributed systems, all streams are considered independent of each other. If more than one stream starts to execute the adaptation simultaneously, there is a high probability that the value that the next position will be identical at more than one stream. Instead of spreading, the message release times would in that case be clustered around the same time instant. Therefore, only one stream is adapting its offsets at the same time needs to be ensured. This is accomplished by introducing a unique criterion for all streams, which ensures that the stream that is the first in the longest busy time $\beta_{longest}$ will adapt its offset.

V. Experimental Results

In this section, first the area utilization of the FPGA modules is evaluated. Then, the results of the response time measurements from the testbed are compared with simulation results. Finally, the testbed is used to show the achievable performance in terms of the AWW of the DynOAA.

Module	Slice Register	Slice LUTs
Message generation	608 (0.88 %)	1234 (1.79 %)
Message queue	236 (0.34 %)	3852 (5.57 %)
CAN controller	540 (0.78 %)	1034 (1.40 %)
Response time measurement	249 (0.36 %)	359 (0.52 %)
Full response time evaluation	924 (1.34 %)	1875 (2.71 %)
Sum	2557 (3.7 %)	8354 (12.09 %)

Table I
RESOURCE UTILIZATION OF THE TESTBED MODULES ON THE TARGET DEVICE XILINX VIRTEX-5 XC5VLX110T

A. Resource Utilization

Table I shows the resources the FPGA modules need to run on a Xilinx Virtex-5 XC5VLX110T. However, it is not at all necessary to use such a big FPGA. The whole design including the evaluation unit that needs to be included only on one FPGA can easily fit on a mid-size Xilinx Spartan 6 device. The main logic resources are required by the priority message queue, because it is build to handle all 2047 CAN priorities. This usage could be easily reduced by optimizing the queue to the number of used streams. The use of embedded memory is not shown, because it is in sum less than 400 Bytes, which is available on any device having embedded memory.

B. Performance Measurements

In the following, we will evaluate the testbed by means of the AWW according to Equation (1) and scenarios as described in Section II-B. The results of the testbed are compared to two different simulation engines: our own and RTaW-sim [11]. Our simulator is discrete-event driven with the simulation step equal to one CAN bit. The main target of the simulation is to be highly configurable such that it is easy to integrate different stream models including run-time scheduling algorithms and that the desired logging information can be easily stored and analyzed. RTaW-sim is a freely available discrete-event simulation of the CAN bus, providing distributions and statistics of the average and maximum message response time from the start of the simulation. Due to these limited logging capabilities, we could only measure the AWW of the first hyper period and not use the advanced clock-drift simulations. The simulation model of both simulations are based on worst-case assumptions of the CAN protocol. For example, the transmission length is calculated by assuming worst-case bit-stuffing (all bits are equal).

Figure 4 and 5 show the results of measuring the AWW over time for a scenario with a bus load of 60% and 90%, respectively. The offsets are chosen uniformly distributed as provided by Netcarbench [7]. The measured results on the testbed are consistent with the lower values obtained through the simulations. Interestingly, the simulation results deviate up to a factor of two. This shows again how important physical setups are. For all runs, the simulations

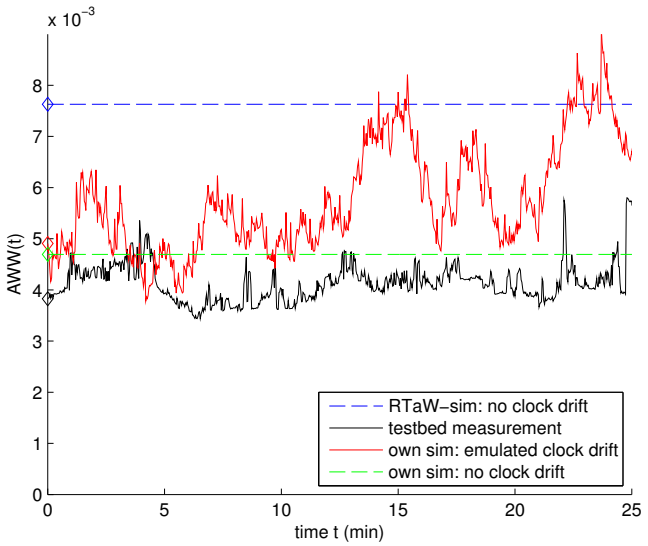

Figure 4. AWW over time for a scenario with a bus load of 60% using random offsets.

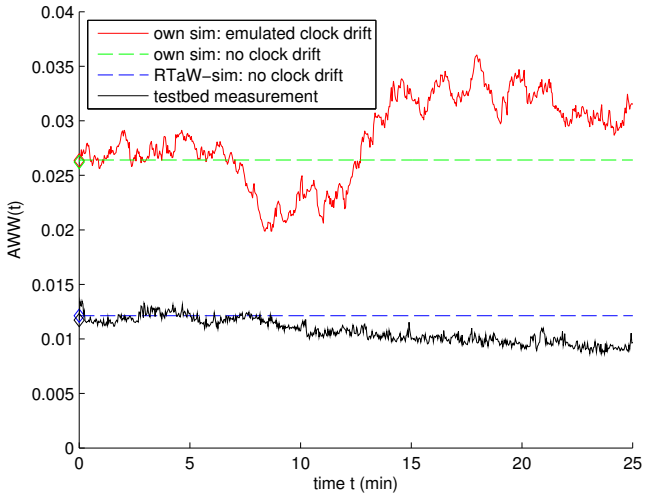

Figure 5. AWW over time for a scenario with a bus load of 90% using random offsets.

tend to be more pessimistic. One of the reasons could be that the simulations assume worst-case bit-stuffing, so the transmissions length is always larger than the real one.

Another interesting observation is that with a bus load of 60%, there are larger variations than with a 90% bus load. As the testbed was designed to have a constant time from release to the first try of transmission on the bus, we could assure that the variations are caused by the traffic patterns on the bus.

The results also show, that the clock drift between the nodes has large influence on the AWW. We simulated this behavior in our own simulator by measuring the clock drift between the FPGA boards and change the periods accordingly. However, using fixed random offsets doesn't result in the same behavior, which can particularly be seen on the measurements with 90% bus load. It has to be noted

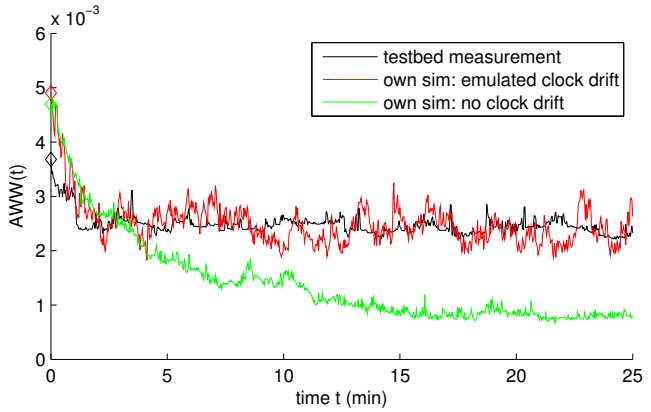

Figure 6. AWW over time for a scenario with a bus load of 60% using DynOAA.

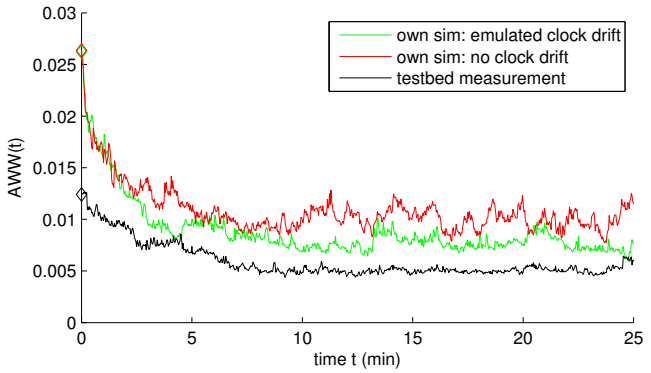

Figure 7. AWW over time for a scenario with a bus load of 90% using DynOAA.

that we only investigated on the clock drift after gathering the results on the testbed.

The results in Figure 6 and 7 were obtained using the same setup as the previous two figures except that the offsets are adapted during run-time by the DynOAA as described in Section IV. The RTaW-sim could not be used for these experiments, because there are no possibilities to adapt the offsets during run-time. The figures show an improvement compared to the random offsets in all setups. On the results of the 60% bus load scenario the testbed could approve the results from the simulation, if the clock drift is simulated. Using the scenario with 90% bus load the results of the simulation are too pessimistic.

A typical performance measure for simulation and emulation approaches is the ratio between simulated and simulation time. The FPGA testbed falls into the category of real-time testing as the ratio is one, which is in the nature of the approach. The simulations should perform better because they are at a higher level of abstraction. However, our own simulation also has a ratio of about one, because the focus was laid on simple implementation rather than performance. In comparison, the RTaW-sim has a ratio of up to three orders of magnitude faster mainly depending on the amount of logging that is done during the simulation.

VI. Conclusions and Future Work

In this paper, we propose a testbed consisting of several FPGAs connected by a CAN bus to analyze the response time of messages, e.g., for debugging, optimization, or for testing of scheduling options as shown here. Even though the response time measurements can only be obtained in real-time, the value of results on an actual physical testbed are indispensable, because using only simulation might ignore important features of the real system such as, e.g., asynchronous nodes or scheduling calculation overhead on nodes. The approach allows on the one hand to validate simulation results and on the other hand to test the design in its physical environment that cannot be easily modeled and emulated by software-based simulators. For example, only the testbed revealed that the clock drift has a larger influence on the schedule than expected. Using the testbed, we were able to show the feasibility and the benefits of our dynamic scheduling algorithms under close to real conditions. Our future work includes extending the evaluation unit to retrieve more timing information such as for example the response times of individual streams over time. Additionally, testing of different nodes is planned as for example using a soft core processor for message generation.

Acknowledgment

This work was supported in part by the German Research Foundation (DFG) under contract TE 163/15-1.

References

[1] "CAN specification 2.0 b." *Robert Bosch GmbH, Stuttgart, Germany*, 1991.

[2] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[3] M. Grenier, L. Havet, and N. Navet, "Pushing the limits of CAN-scheduling frames with offsets provides a major performance boost," in *Proc. of the 4th European Congress Embedded Real Time Software (ERTS 2008), Toulouse, France*, 2008.

[4] F. Zhou, S. Li, and X. Hou, "Development method of simulation and test system for vehicle body CAN bus based on CANoe," in *7th World Congress on Intelligent Control and Automation, WCICA*. IEEE, 2008, pp. 7515–7519.

[5] J. Gamiz, J. Samitier, J. Fuertes, and O. Rubies, "Practical evaluation of messages latencies in CAN," in *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA'03. IEEE Conference*, 2003, pp. 185–192.

[6] S. Samii, S. Rafiliu, P. Eles, and Z. Peng, "A simulation methodology for worst-case response time estimation of distributed real-time systems," in *Proceedings of the conference on Design, automation and test in Europe, DATE*. ACM, 2008, pp. 556–561.

[7] C. Braun, L. Havet, and N. Navet, "NETCARBENCH: A benchmark for techniques and tools used in the design of automotive communication systems," in *7th IFAC International Conference on Fieldbuses and Networks in Industrial and Embedded Systems*, 2007, pp. 321–328.

[8] T. Ziermann, Z. Salcic, and J. Teich, "DynOAA - dynamic offset adaptation algorithm for improving response times of CAN systems," in *Proc. of Design, Automation, and Test in Europe (DATE)*, 2011, pp. 269–272.

[9] *LT1796 - Overvoltage Fault Protected CAN Transceiver*, 2001.

[10] I. Mohor, "Can protocol controller," http://opencores.org/project,can.

[11] RTaW-Sim, "Real-time at Work CAN Simulator," http://www.realtimeatwork.com/software/rtaw-sim/, 2009.