Time Synchronization for an Emulated CAN Device on a Multi-Processor System on Chip

Gabriela Breaban*, Martijn Koedam*, Sander Stuijk*, Kees Goossens*†
*Eindhoven University of Technology, The Netherlands
{g.breaban,m.l.p.j.koedam,s.stuijk,k.g.w.goossens}@tue.nl
†Topic Embedded Products, The Netherlands

***Abstract*—The increasing number of applications implemented on modern vehicles leads to the use of multi-core platforms in the automotive field. As the number of I/O interfaces offered by these platforms is typically lower than the number of integrated applications, a solution is needed to provide access to the peripherals, such as the Controller Area Network (CAN), to all applications. Emulation and virtualization can be used to implement and share a CAN bus among multiple applications. Furthermore, cyber-physical automotive applications often require time synchronization. A time synchronization protocol on CAN has been recently introduced by AUTOSAR.**
In this article we present how multiple applications can share a CAN port, which can be on the local processor tile or on a remote tile. Each application can access a local time base, synchronized over CAN, using the AUTOSAR Application Programming Interface (API). We evaluate our approach with four emulation and virtualization examples, trading the number of applications per core with the speed of the software emulated CAN bus.

I. Introduction

The limited scalability of single-core ECUs in conjunction with the increasing number of functionalities being integrated in modern vehicles leads to a shift towards a domain controlled architecture in the automotive field. This consists of consolidating multiple software functionalities on the same hardware platform based on their domain [16] and it leads to increased computational requirements. To cope with this demand, the use of multi-core platforms has been proposed in literature [16]. Multi-core platforms can come as either Commercial-Off-The-Shelf (COTS) platforms or as Multi-Processor Systems on Chip (MPSoCs).

A COTS platform features a given number of cores and I/O interfaces. Since the number of I/O interfaces is typically lower than the number of applications requiring them, when integrating multiple software applications on such a platform, the given resources have to be shared between applications such that each one meets its requirements in terms of real-time capabilities, safety, and security.

The implementation of the protocol governing an I/O interface is usually done in hardware and therefore, sharing the I/O interface translates into sharing the hardware controller that drives the interface. When sharing a resource among applications with strict and diverse requirements, as in automotive, an important property of the sharing method is isolation. Isolated resource sharing is equivalent to *virtualization* and it means dividing the physical resource into multiple separate virtual resources that don't interfere and allocating each one to an application. On the other hand, when deciding the I/O interfaces for a Multi-Processor System on Chip (MPSoC), one can choose to include a hardware controller and search for virtualization solutions, or, as an alternative, a given communication service can be obtained by implementing it in software on top of an existing interface. We call the latter solution *software emulation*. The emulated interface can then be further shared through virtualization.

Since the automotive industry currently only uses COTS hardware platforms that typically include CAN controllers, a considerable amount of research focuses on virtualization solutions for such systems. To the best of our knowledge, the possibility of designing a CAN interface on a MPSoC platform that scales depending on the number of applications and cores has not been addressed in literature.

In terms of virtualization, the latest proposed methods in automotive systems are inspired by server environments where Virtual Machines (VMs) define an isolated set of resources [6]. Consequently, since the mostly used network in server environments is Ethernet, the virtualization methods for the CAN interface are derived from state-of-the-art techniques used for the Ethernet interface [9]. Virtual platforms have been introduced for isolating resources on a multi-processor platform and allocating them to individual applications [8].

In terms of software emulation, the CAN interface has been build on top of specific hardware architectures such as the Time Triggered Architecture (TTA) [11]. However, this solution targets non-critical non-real-time CAN applications and it does not address the problem of providing isolated CAN interfaces to multiple applications integrated on the same platform.

Time synchronization is used for distributed cyber-physical applications running on different processing nodes that require a global notion of time. Global time is needed either for synchronized actions (e.g. sensor reads, actuator triggers) or for accessing absolute time (e.g. Global Positioning System (GPS), Coordinated Universal Time (UTC), Temps Atomique International (TAI)) to perform sensor data fusion, event data recording, etc. Time synchronization can be obtained by exchanging messages between a predefined master and slave, after which the slave corrects its local clock. The most well known time synchronization protocols are Network Time Protocol (NTP) and Precision Time Protocol (PTP). In automotive, AUTOSAR recently introduced, as of release

4.2.2, simplified versions of the PTP protocol for the CAN, Flexray and automotive Ethernet networks.

In this paper we evaluate four different emulation and virtualization solutions as examples of a general method that provide a trade-off between the number of applications sharing a CAN port, which can be on the local or a remote processor tile, with the speed of the software emulated CAN bus. This offers to the user the possibility of choosing a different implementation depending on the number of applications being integrated on the platform and also the desired CAN bit rate. Our prototype enforces full temporal isolation and offers spatial isolation that is yet to be enforced in hardware. Hence, this impacts the degree of safety criticality that can be supported on our prototype. Our software CAN controller achieves bit rates between 1 and 100 kbit/s in the experiments done on our 5 Microblaze processor platform synthesized on the Xilinx ML605 Field-Programmable Gate Array (FPGA). The CAN user applications can also access a local time base, synchronized over CAN using the AUTOSAR CAN time synchronization protocol.

The paper is structured as follows: Section II presents the related work, Section III gives an overview of the proposed method, Section V and Section VI describe its implementation, and finally Section VIII concludes the paper.

II. Related work

Herber et al. propose software CAN controller virtualization methods inspired from server environments [9]. The software method consists of paravirtualization. However, the presented results show the performance of the method only in an interference-free scenario. Moreover, to avoid an increase of the performance overhead involved by scheduling, only one VM was mapped to each core, leading to a limited scalability. As a comparison, in one of our four solutions we also use a dedicated core as a CAN gateway. The main differences are that we use the CoMik microkernel [13] to schedule multiple applications on the CAN client cores and communicate the CAN message to the CAN gateway using C-HEAP FIFOs [14] via a contention-free Network on Chip (NoC). The C-HEAP protocol ensures a safe synchronous communication. On the CAN gateway core, the arbitration between the incoming messages is done using a round-robin schedule.

To reduce the performance overhead, Sander et al. offer the solution of hardware controller virtualization [17], based on Single Root I/O virtualization (SR-IOV). SR-IOV is an extension of the Peripheral Component Interconnect Express (PCIe) protocol and it is the state-of-the-art hardware I/O virtualization method for Ethernet. The implementation is done by extending a CAN controller to add virtualization support and connecting it to a multi-core processor via a PCIe interface. Unlike the software method, the hardware one has the downside that the PCIe interconnect affects the temporal isolation between the serviced VMs leading to a performance degradation. This is caused by the fact that all VMs share the same interconnect and the contention on the bus cannot be avoided. In comparison, our solution does not target the enhancement of existing COTS platforms. It rather proposes a combined software and hardware design method for a platform based on a template hardware architecture, whose instance could afterwards be taped out for a specific automotive system.

An orthogonal approach from Herber et al. introduces CAN network virtualization [10]. The method is implemented in hardware and it divides a physical network into multiple virtually isolated networks of different priorities. CAN nodes are then allocated to a certain network based on their criticality. Our method does not target the virtualization of a CAN network, but the emulation and virtualization of a CAN controller.

In terms of emulation, the CAN interface has been integrated in the TTA architecture by implementing it on top of the TTP/C interface [15]. Apart from providing the functionality of the CAN protocol, the emulated CAN adds new services such as membership information, global time, temporal composability and increased dependability. The reported implementation uses the embedded real-time Linux operating system to integrate CAN applications and real-time applications. However, the CAN applications are allocated to the non-real-time part of the kernel and are competing with standard Linux applications for resources. In our case, we do not implement the CAN protocol on top of another protocol, but we simply lift the implementation of the CAN Media Access Control (MAC) layer from the hardware to the software on top of a hardware module that realizes the CAN physical layer and use the CoMik microkernel to schedule real-time CAN applications.

In terms of time synchronization in in-car networks, Lim et al. offer an evaluation of IEEE 802.1AS standard for switched Ethernet [12]. The authors measured the peer propagation delay and the synchronization error in daisy-chain based topology using the OMNeT++ simulation environment. Our work evaluates the synchronization accuracy for the CAN bus using our MPSoC prototype.

III. Design Alternatives for CAN Emulation and Virtualization

A. Overview

In the context of automotive applications, we propose a method to design a CAN interface on a MPSoC that consists of defining different platform configurations that trade-off the number of supported applications and CAN ports with the bit rate of the CAN bus. The MPSoC platform consists of a set of processor tiles, each one embedding a processor, the local memories and the CAN modules. Each CAN module provides a CAN port. The main design parameters that we vary are:

1) the number of applications sharing each processor
2) the number of CAN ports per processor tile
3) the number of applications sharing a CAN port
4) the bit rate of the CAN bus

The CAN parameters (bit rate and number of ports) are used for hardware synthesis, while the others are part of the software design. Table I gives an overview of the exact values of the parameters for each of the four example configurations.

Each configuration ensures a complete temporal isolation between applications. Spatial isolation is logically ensured in the sense that each application gets assigned its own stack,

TABLE I
VIRTUALIZATION AND EMULATION PLATFORM CONFIGURATIONS

Configuration	E_1	E_2	V_1	V_2	
CAN Bus Baud Rate [kbit/s]	4	2	2	100	
# (applications + controllers) per core	Cores 1-4 1+1	Cores 1-4 2+2	Cores 1-4 2+1	Core 1-3 2+0	Core 4 0+1
# CAN ports per tile	Tiles 1-4 1	Tiles 1-4 2	Tiles 1-4 1	Tile 1-3 0	Tile 4 1

heap and data memory, but the proposed configurations do not include a memory protection unit to enforce this separation.

Each CAN port is connected to an individual hardware module that implements the physical layer of the CAN protocol. The MAC layer is implemented in software. We refer to this implementation as a *software emulated CAN device* since it achieves the functionality of a hardware CAN device in software. Further, if the CAN port is to be used by multiple applications such that the integrity of the data sent and received on CAN by each one of them is not affected, we say that the CAN device is *virtualized*.

Given the design parameters presented above, we defined four platform configurations: two configurations for which the CAN device is emulated but not virtualized, denoted E_1 and E_2 and two others for which the CAN device is emulated and virtualized, denoted V_1 and V_2. E_1 and E_2 differ on whether the processor is shared between multiple applications or not. V_1 and V_2 differ on whether the emulated CAN device shares the processor with other applications or not. As the CAN device is implemented in software, the maximum achievable bit rate in each case depends on whether the processor on which it runs is shared with other applications or not.

In the remainder of this section we will describe and evaluate each of the four configurations.

B. Platform Configuration E_1

This configuration is the simplest one, in the sense that the value of each of the design parameters mentioned above is equal to 1. We have one application on each processor using a local CAN port. The bit rate of the CAN bus is 4 kbit/s.

We will refer to Figure 1 to describe the system architecture of E_1 and E_2, as they have a similar structure. This configuration as well as the other ones, comprises four processor tiles. The figure shows the tile architecture for the case in which we have two applications and two controllers running on a processor. For E_1, the structure is the same, only that it has one application and one controller. On the software side, we can see that the sequence of function calls starts from the application layer, where the message is created. Then the AUTOSAR driver API [1] is called, that further calls a version of the C-Heap library to safely transfer the message into the controller's buffer. Finally the controller accesses the CAN hardware module to transmit the message. On the bottom software layer, the CoMik microkernel creates the TDM partitions in which the tasks (application and controller) can run without interference. Further details about the software implementation are given in Section V.

The main advantages of this configuration are the spatial isolation between applications, as they are mapped one-to-one to the processor cores and the use of the local data memory on the tile for the communication between the application and the CAN device, which implies a low timing overhead. The disadvantage is the low scalability in terms of number of supported applications.

C. Platform Configuration E_2

In this configuration, we increase both the number of applications and CAN ports per core to two, such that each application accesses its own emulated CAN device. Since the number of software entities running on the same processor is higher, the CAN bit rate decreases to 2 kbit/s.

The advantages of this configuration are the increased number of applications running on each core, the physical isolation between the CAN ports used by each application and, as in the previous case, the use of the local memory for the application to CAN device communication. The number of increased applications and CAN ports come at the expense of the reduced CAN bit rate, and, implicitly, extra area for the second CAN module.

D. Platform Configuration V_1

Configuration V_1 is similar to E_1, the main difference is that the number of applications running on each core is equal to two. This means that the emulated CAN device and the port that it drives is shared between the two applications. Each application has its own transmit and receive buffer and the arbitration between them is done in software based on the message ID. The bit rate of the CAN bus is 2 kbit/s. Figure 2 illustrates the system architecture for this case. The multiplexer inside the CAN Controller symbolizes the ID-based arbitration.

Compared to E_1, the main advantage of this configuration is the improved scalability of the CAN device. This comes at the price of using the same physical CAN port for all applications on the core.

E. Platform Configuration V_2

Configuration V_2 differs more from the previous ones. In this case, we use a dedicated core to implement a CAN device, which operates as a CAN gateway at 100 kbit/s bit rate. As this core is not shared with other applications, the CAN controller runs bare-metal. Each of the other cores runs two applications. To send and receive CAN messages, the cores use the NoC for the communication with the dedicated CAN core. Each CAN application has a separate transmit and receive FIFO. Moreover, the Daelite NoC [18] provides contention-free communication; therefore the message communication time

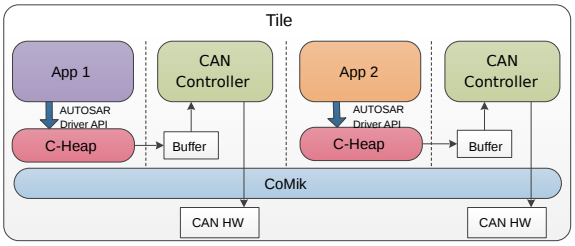

Fig. 1. CAN Configuration E2 - System Architecture of a tile

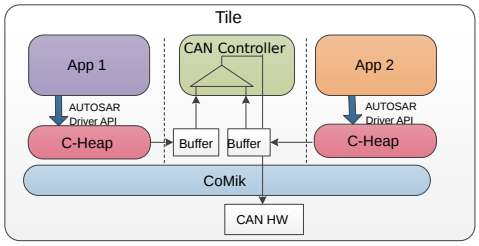

Fig. 2. CAN Configuration V1 - System Architecture of a tile

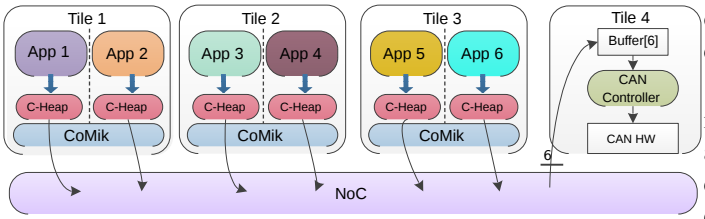

Fig. 3. CAN Configuration V2 - Using one tile as a CAN gateway

is predictable and bounded and it can be used to offer timing guarantees for the end-to-end transmission and reception of the messages to be sent over the CAN bus.

Figure 3 illustrates the system architecture for this configuration. For simplicity, the arrows illustrate the sequence of function calls only for the transmission of messages from the applications to the gateway through the NoC.

IV. Time Synchronization on the CAN network

Starting with the release 4.2.2, AUTOSAR introduced specifications for time synchronization on the CAN network [2], [3]. This section presents the time synchronization concepts according to the AUTOSAR specifications.

AUTOSAR defines 16 synchronized time bases and 16 offset time bases for the CAN [2]. A time base is a unique source of time that has its own progression rate, ownership and reference to the physical world. An offset time base is statically linked to a certain time base. Offset time bases were defined for large systems that require more than 16 time bases. A time base can be absolute (e.g. GPS) or relative. Relative time bases are used in automotive to track relative amounts of time, such as the operating time of the vehicle or of the ECU. A time hierarchy is formed by the distribution of a time base over different network segments via time gateways.

The time synchronization protocol for the CAN network is a simplified version of the PTP protocol [5]. The PTP protocol consists of four messages exchanged between the time master and the time slave. First, the master sends a SYNC message containing an estimate of the current time, then it sends a Follow Up (FUP) message containing a precise value of the current time, taken as close as possible to the physical network layer. In the second part, the slave sends a Delay Request message to which the master replies with a Delay Response message containing the receipt time of the Delay Request message. Based on these exchanged messages, the slave estimates the master-slave link delay and computes the offset. The computed offset is then used to correct the local clock.

For the CAN network, the time synchronization protocol is reduced to the first part of PTP, that is, only the SYNC and the FUP messages are being used. PTP can use several communication protocols such as Ethernet, PROFINET, UDP, etc. One fundamental difference between PTP and CAN Time Synchronization is that PTP does not rely on a MAC level means to detect the correct reception of a message at the slave side during its transmission. For Ethernet, when a collision happens on the bus, the sender backs off and retransmits the message later. Acknowledgement mechanisms can be added by using a high level protocol, such as TCP/IP, to indicate that the slave correctly received the message. Instead, a CAN message includes an acknowledgement field in the message, driven by the slave, by which the master can detect whether the message was correctly received. For PTP, the local time at the slave is computed from the transmitted timestamps and the link delay. The time synchronization on CAN, on the other hand, relies on the bit timing which is designed to compensate for the signal propagation time for the longest link in the network. Thus, the link delay doesn't have to be computed by the slave through bidirectional communication, as in the case of PTP.

Let us describe the CAN synchronization steps in more detail. Figure 4 shows the details of the protocol. The synchronization occurs periodically, with a predefined period. At the beginning of the synchronization period, the time master reads the current local time in both standard format (t_0, represented in seconds and nanoseconds) and raw format (t_{0r} in nanoseconds) and includes the seconds portion of the standard format (32 least significant bits, $s(t_0)$) in the SYNC message. When the SYNC message has been completely transmitted, the master records the difference in raw time between the current raw time (t_{1r}) and the SYNC message timestamp (t_{0r}): $t_{4r} = t_{1r} - t_{0r}$ and any seconds overflow (OVS) while the slave records the reception time in raw format (t_{2r}). Next, the master sends the recorded raw time difference in the FUP message. When receiving the FUP message, the slave records the difference in raw time between the reception time of the FUP message (t_{3r}), and the reception time of the previous SYNC message. Finally, the slave computes the synchronized local time as follows:

$$NewTime.nanoseconds = (t_{3r} - t_{2r} + t_{4r})\%10^9$$
$$NewTime.seconds = s(t_0) + OVS + (t_{3r} - t_{2r} + t_{4r})/10^9$$

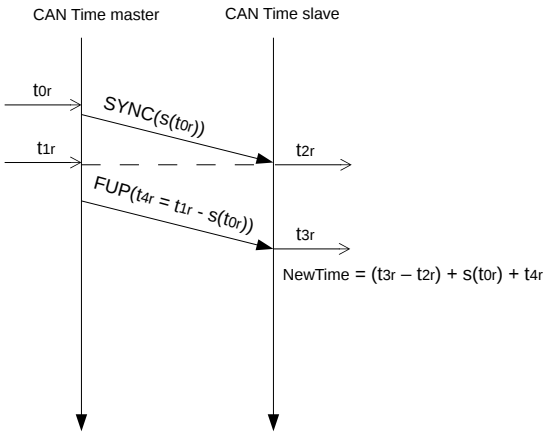

Fig. 4. Time Synchronization over CAN

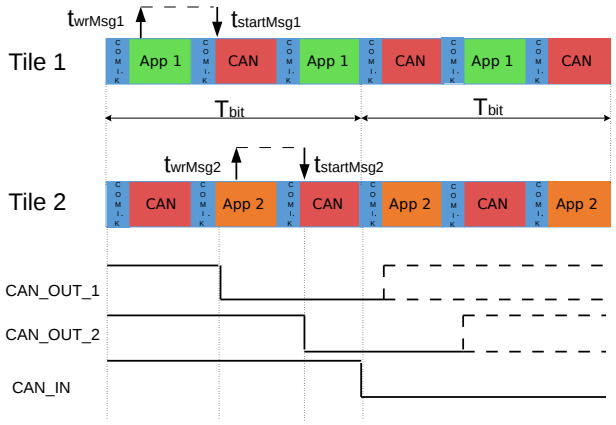

Fig. 5. Timing Diagram for configuration E_1 - Emulated CAN on top of CoMik

V. Implementation of the CAN Device

We have implemented the physical layer of the CAN interface as a hardware module. This module functions as a bidirectional bridge, receiving on one side the data to be transmitted on CAN from the Microblaze processor and on the other side putting it on the CAN port. The module can be instantiated multiple times on each processor tile and the resulting CAN line is a wired AND between all the CAN ports present on the platform. The CAN bit frequency is obtained by dividing the processor clock frequency with a constant value. All the tiles run synchronously at the same clock frequency. In the remainder of this article we use the term synchronous to refer to a platform that includes a single clock oscillator, which feeds all the hardware components instantiated on it.

A. Software Emulation of the CAN Controller

The CAN MAC layer was implemented in software in the C programming language and it consists of creating the CAN frame in the 2.0A format, as defined by the ISO 11898 standard [4], including bit stuffing, CRC computation and filtering of the received messages. We call the software implementation of the CAN MAC layer *emulation* since it acts as a CAN controller, which transmits the CAN frames sent by the application and returns back to it the received frames according to the configuration of the reception filter. To ensure a safe transfer of the data between the application and the controller, a simplified version of C-Heap is used. Further, we have implemented the driver API according to the AUTOSAR standard.

B. Implementing a CAN Controller on the Virtual Processor

We will first present the design concept for a synchronous CompSoC platform in which the clock skew and jitter are negligible. Then we will explain the modifications needed to tolerate these deviations.

To be able to run the software CAN controller together with other applications on the same processor, we use the CoMik microkernel. CoMik divides the physical processor into multiple virtual processors scheduled in TDM fashion. Each virtual processor gets a fraction of the processor capacity based on the number of allocated TDM slots and it is fully temporally isolated from the other virtual processors. The TDM table duration determines the maximum sustainable CAN bit rate, as the software controller has to be fast enough to write or read every CAN bit in its allocated slot.

Each software controller accesses a unique physical CAN port. In order to provide CAN access to multiple applications, we need to either instantiate in hardware the same number of CAN ports as the number of applications, or share a lower number of CAN ports. Both options imply creating a TDM table that accommodates all the applications and their software CAN controllers, and defining the maximum CAN bit rate based on the maximum delay between two successive allocated TDM slots allocated to the same controller, among all controllers. Thus, in this case, the minimum CAN bit duration, $T_{bit_{min}}$ is:

$$T_{bit_{min}} = \max_{0<i\leq N}\{\max_{0<j<2\cdot M_i}(t_{i_{j+1}} - t_{i_j})\} \quad (1)$$

where N refers to the total number of CAN controllers running on the platform, M_i represents the number of TDM slots allocated to the controller i and t_{i_j},$t_{i_{j+1}}$ denote the start time of slots j and $j+1$ of controller i. To detect the maximum delay between any two successive slots of controller i, we need to consider two successive TDM frames, which is why the upper bound for the second max operator is $2 \cdot M_i$. Hence, the maximum CAN bit rate, R_{max} for this case is:

$$R_{max} = \frac{1}{T_{bit_{min}}} \quad (2)$$

Figure 5 shows the TDM schedule for configuration E_1 and the CAN signals. A TDM frame consists of two slots, one allocated to the application and one to the CAN controller. Each TDM slot contains a CoMik sub-slot and a partition sub-slot. In the CoMik sub-slot the context switch operations are performed. The application and CAN driver each run in partition sub-slot. In the figure, the maximum delay between any two consecutive CAN slots is two slots and the chosen CAN bit period, T_{bit} is higher than the minimum (two slots) and it is equal to three slots. We can see that applications 1 and 2 write a transmit message in corresponding buffers

at times t_{wrMsg1} and t_{wrMsg2} respectively. The C-Heap library is not shown in the figure for the sake of simplicity. Each CAN controller detects the message in the following slot, at times $t_{startMsg1}$ and $t_{startMsg2}$ respectively and it starts to drive the allocated CAN output port immediately. The resulting CAN line, CAN_IN changes at the start of every CAN bit period and it reflects the result of all the CAN output lines on the platform. All CAN controllers synchronize with the CAN bus at the beginning of each bit period, T_{bit}. When the controller is shared, as in configuration V_1, separate buffers are allocated to each client application and the incoming messages are arbitrated based on their IDs.

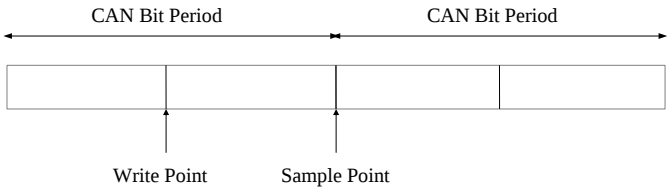

Fig. 6. CAN Bit Timing on CompSoC

Driving the CAN bus at any point within the bit period works properly in a synchronous platform in which the skew and jitter of the different processor tiles are low enough to be ignored. However, substantial skew and jitter can lead to incorrect functioning of the bus since the writing and reading of the bus values within a bit period would not be always synchronized among the tiles. To tolerate clock substantial jitter, we set the writing point and the sampling point as far as possible from each other, that is half CAN bit period apart, as seen in Figure 6. For this the CoMik TDM table must be aligned with the CAN bit period on each processor tile and the slot for the CAN controller is allocated such that the controller is running when the middle of the CAN bit period is reached. Thus, we will only have one CAN controller slot per TDM table. This will impact the design space choices, as we can no longer make use of more than one CAN port per tile. We return to clock synchronization below.

C. Bare-metal Implementation of the CAN Controller

Configuration V_2 illustrates the possibility of allocating the entire processor to the CAN controller. Figure 7 shows the stages of sending a CAN message from the moment the application creates it, t_{wrMsg1} until its transmission starts on the CAN output line, CAN_OUT. As mentioned before, we use the C-HEAP library to send the CAN messages across the NoC. Each sending application has its own FIFO transmit buffer in the local memory of the CAN gateway tile. A FIFO contains a number of predefined data tokens. In our case, a token is a CAN message. When writing a token into a remote FIFO, the sender first sends the token and then the value of the updated write counter via the NoC. A NoC path between 2 tiles includes a number of routers. In the figure, the tokens traveling from the sender tile to the CAN gateway go through four routers. The NoC is scheduled using a pipelined TDM table. This means that across the path, each router forwards the data from one of its inputs to one of its outputs in a given TDM slot,

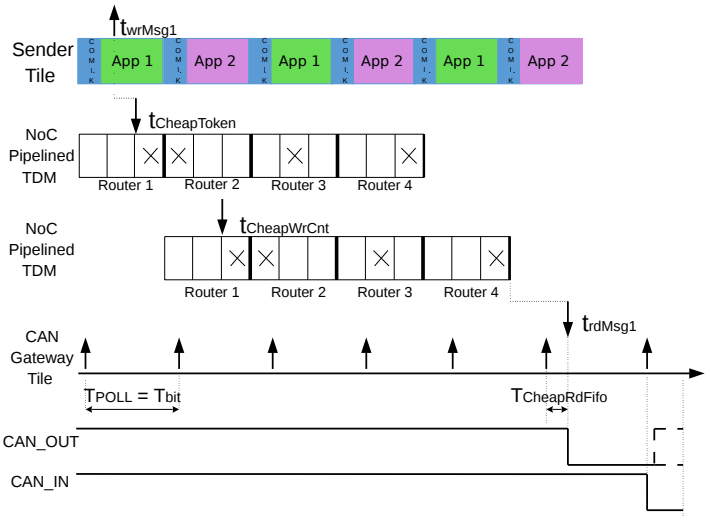

Fig. 7. Timing Diagram for configuration V_2 - Bare-metal implementation of the CAN Controller and the Communication of CAN messages via NoC

such that for a TDM frame having *n* slots, router *i* forwards the data during slot *j* and router *i+1* forwards the same data in the following slot, *(j+1) mod n*. In the figure, the NoC TDM table has 3 slots and the connection between the sender tile and the gateway tile uses slot 3 in the first router and it increases with 1 in every upcoming router. After the write counter has left the last router, it reaches the gateway tile. Here, when the CAN bus is idle, at the start of every CAN bit period, T_{bit}, the transmit FIFO of each CAN client is polled. If a new token is found, it is read during $T_{CheapRdFifo}$ and the transmission of the message starts right away on the CAN_OUT line. Since in this case the processor is not virtualized, the performance bottleneck determining the CAN bit rate is no longer given by the TDM table, but by the worst case execution time needed to send one CAN bit, which is determined by accessing the communication FIFOs.

VI. Implementation of the Time Synchronization over CAN

This section describes how the configurations presented above can be extended to include the time synchronization protocol over CAN. We distinguish between two main configuration types: one that uses the local emulated/virtualized CAN device (such as E_1, E_2 and V_1) and one that uses the remote CAN device (such as V_2).

A. CAN Bit Timing and Clock Signal Deviation Concepts

CAN is an event-triggered communication protocol. The nodes connected to the bus synchronize with each other via the edges of the CAN signal. For this, the Non Return To Zero (NRZ) signal encoding enforces a signal change (and thus, an edge) after every 5 consecutive bits having the same value. The CAN bit synchronization happens at the start of frame (on the Recessive to Dominant edge) and during the frame via the stuffed bits.

The CAN bit period consists of four segments: SYNC, PROP, PHASE_1 and PHASE_2, as can be seen in Figure 8.

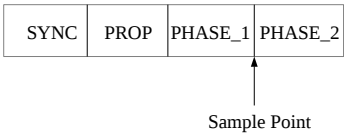

Fig. 8. CAN Bit Timing

The SYNC segment is used for synchronization and it is where the signal edge is expected, while the other segments are used to compensate for the signal propagation times and phase differences across the network. The Sampling Point is the moment when the current value of the bit is sampled by all the connected nodes. Via synchronization, the CAN controllers shorten or lengthen the bit period to align with each other on the bit period start. This shortening or lengthening is realized either by restarting the CAN bit timing, at the Start of Frame, or by adjusting the PHASE_1 and PHASE_2 segments, on the stuff bits.

In an ideal platform, all HW modules have synchronous clocks, that have the same phase and period. Although one particular instance of the CompSoC platform manifests these ideal properties, in general it is a GALS platform, that deviates from this ideal case. To characterize the behavior of the clocks on GALS platforms, we introduce three concepts: *clock skew*, *clock jitter* and *clock drift*. The clock skew or phase shift is a constant time difference between a clock transition and a reference. It is constant from a cycle to another and is equivalent to a phase shift [7]. The concept is illustrated in Figure 9. Clock jitter represents a deviation from periodicity, which can vary from cycle to cycle, as shown in Figure 10. Finally, clock drift refers to the variation of the clock signal frequency with respect to a reference frequency. Clock drift is illustrated in Figure 11. Out of these deviations, clock drift is the main contributor to time desynchronization. Clocks that drift away from each other will cause arbitrarily different time values, making time synchronization necessary across devices that require a common notion of time.

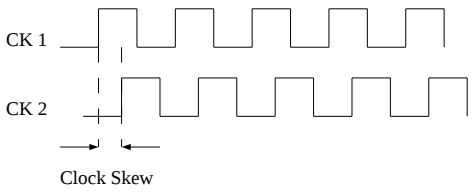

Fig. 9. Clock Skew

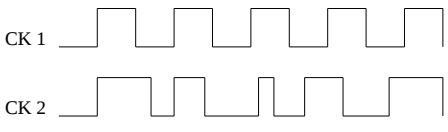

Fig. 10. Clock Jitter

Our implementation is 100% synchronous and therefore only exhibits skew and jitter at processor frequency. This frequency is much higher than the CAN frequency and hence can be ignored. Drift is not present. In a GALS version of CompSoC and embedded CAN, skew, jitter and drift will be present. Because in this case the processor clocks run much faster than the CAN clock, we propose to use a software adjustment to clocking issues, i.e. by adding or removing processor cycles in the TDM slot to stay in sync with the CAN bus.

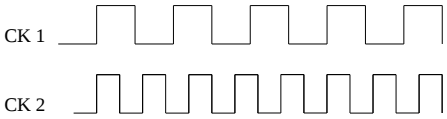

Fig. 11. Clock Drift

B. AUTOSAR Time Synchronization Concepts

According to the AUTOSAR specification, there are three software modules involved in the time synchronization: the CAN Time Synchronization module, the Synchronized Time Base Manager and the CAN driver. The CAN Time Synchronization module is responsible for starting periodically the time synchronization and creating the corresponding CAN messages, on the master side, and for processing the contents of the time synchronization messages, on the slave side. For this, it interacts with the Synchronized Time Base Manager for either reading the current time (for the master) or setting it (for the slave). The Synchronized Time Base Manager keeps the synchronized time base(s) in both raw format (as given by the local timer) and standard format. The raw format uses the nanosecond as a unit and is represented on 32 bits. The standard format data type, shown in Figure 12, is used to express time in seconds (on 48 bits) and nanoseconds on 32 bits. Finally, the CAN driver is responsible for interacting with the CAN hardware.

The SYNC and FUP CAN messages share the same CAN ID. The value of the ID is to be decided by the user.

```
typedef struct {
  StbM_TimeBaseStatusType timeBaseStatus;
  uint32 nanoseconds;
  uint32 seconds;
  uint16 secondsHi;
} StbM_TimeStampType;
```

Fig. 12. AUTOSAR StbM_TimeStampType

C. Time Synchronization using a local CAN Device

To illustrate the implementation of the time synchronization for this type of configuration, we chose configuration E_1, the simplest of the three configurations of this type.

Figure 13 shows the software architecture for configuration E_1. The changes consist of adding an extra TDM time slot that corresponds to the CAN Time Synchronization module and a new library that implements the Synchronized Time Base Manager. The application, denoted as App 1, can get the current time value for a certain time base by using the Synchronized Time Base Manager API, while the CAN Time Synchronization either starts the time synchronization

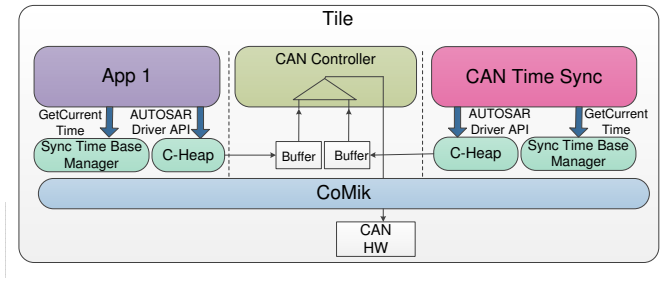

Fig. 13. Time Synchronization Implementation for configuration E_1

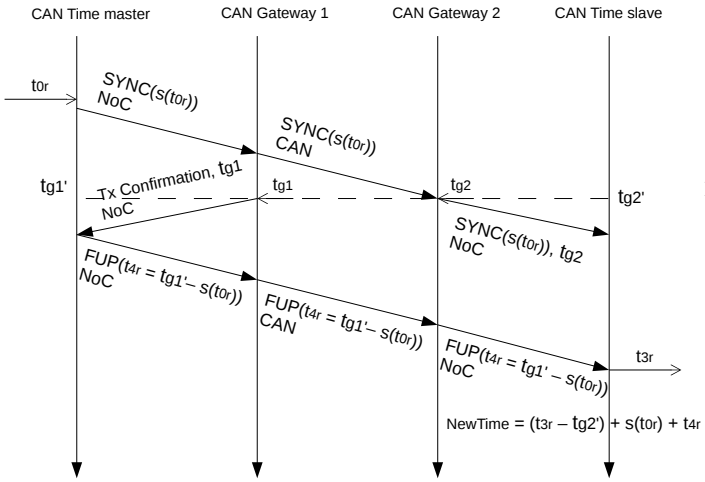

Fig. 14. Time Synchronization over CAN gateways

periodically or it updates the local time base using the data received in the time synchronization messages.

An important observation is that since the implementation of time synchronization requires the addition of an extra TDM slot, as a result, the achievable bit rate for the CAN bus scales down, as explained in Section V-B.

For this type of configuration, the clock deviation concepts presented in Subchapter VI-A apply when the CAN communication takes place between two asynchronous tiles.

D. Time Synchronization using a remote CAN Device

When using a CAN configuration in which the CAN device is implemented on a remote processor, such as V_2, the CAN Timemaster and Timeslave need to send and receive, respectively, the corresponding CAN messages via the NoC to/from the CAN gateway.

We distinguish two subcases here: the case in which the CAN Timemaster and the CAN Timeslave are connected to different gateways (shown in Figure 14) and the case in which they are connected to the same gateway (as in Figure 15). Remember that for both cases, we reserve one processor per MPSOC to act as CAN gateway.

The CAN gateway is responsible for the CAN communication and can take a timestamp when the transmission of the SYNC message is completed and acknowledged by the slave. This timestamp can further be used by the time master or the time slave to proceed with the time synchronization protocol. However, since the CAN gateway is using a different

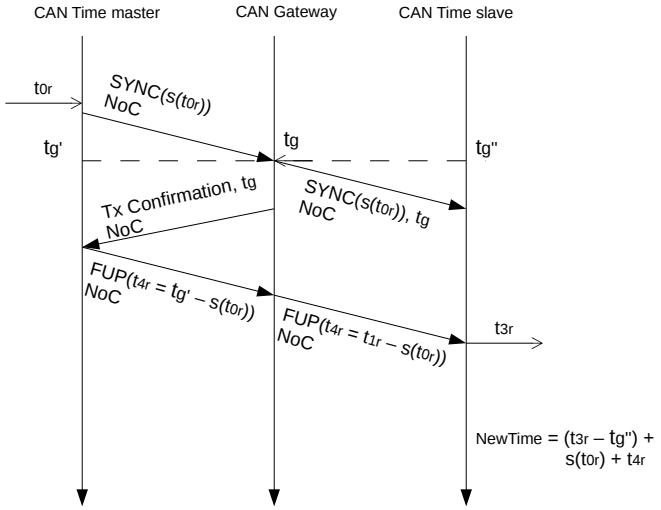

Fig. 15. Time Synchronization for configuration V_2

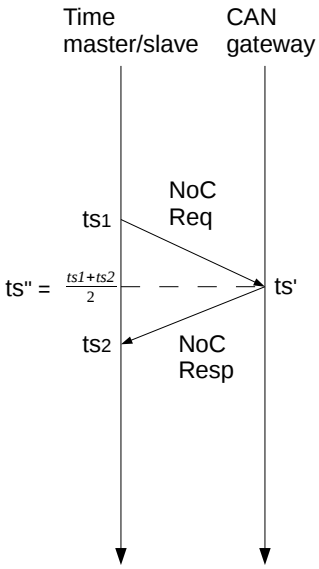

Fig. 16. CAN Gateway to master/slave offset computation

clock than the time master/slave, we need to keep track of the offset between the two clocks in order to transpose the CAN gateway timestamp into the corresponding timestamp at the master/slave side. Therefore, the time master/slave has to read regularly (with a predefined period) the CAN gateway clock value and detect the offset. This process is shown in Figure 16: the master/slave takes a timestamp, ts_1, then requests a timestamp from the gateway, ts', and takes another timestamp ts_2 after receiving the response. Assuming that this process is not interrupted (as explained in the previous section) and that the communication on both ways is symmetric, we can consider that the gateway timestamp ts' corresponds to the midtime between ts_1 and ts_2 and compute the offset as ts''-ts'. This can be visualized in Figure 16. Note that the communication between the tiles is realized via the NoC using DMAs. The DMAs do not introduce communication jitter since the DMAs are not shared between applications. Although the NoC is synchronous with the processor tile, its

TDM schedule (16 slots of 3 words each) is different from that of the processors (TDM slots of 16,000 cycles) and it therefore introduces a small jitter. In general, the jitter on the paths between master and slave is asymmetric. However, the jitters are small enough to be ignored, and do not significantly impact the time synchronization accuracy.

If the CAN Timemaster is connected to a different CAN gateway than the slave, the communication on CAN happens between the two gateways. Each time the CAN message is transmitted on the CAN bus, each gateway first takes a timestamp after the message is completely sent/received, and then sends a transmit or receive confirmation to the Timemaster and the Timeslave, respectively, together with the recorded timestamp. The Timemaster/Timeslave then transposes the timestamp into its own time and uses it according to the protocol. In other words, the transposed gateway timestamp corresponds to t_{1r} and t_{2r} in Figure 4. The process works similarly in both subcases and is illustrated in Figure 15 and Figure 14.

VII. Experiments

A. CAN Emulation and Virtualization

We synthesized the four platforms according to the configurations described in the previous sections on a ML605 Xilinx FPGA platform. Each of the four configurations includes five processor tiles, out of which four are used for running CAN applications and the fifth tile is used as a CAN monitor, which prints the value of every CAN bit. Table II shows the FPGA resource utilization and the clock generation timing results for each configuration.

The applications within all configurations are synthetic, meaning that their only purpose is to send and receive CAN messages periodically.

Figure 17 shows the message latencies and software cost for each of the proposed configurations using a logarithmic scale. In configuration E_1 three applications send messages periodically with a dynamic offset and a fourth application is receiving them. The sending period is 0.1 s and it was chosen to fit three worst-case CAN messages coming from the three applications. The offset is varying between 0 and 40.9 μs (the TDM slot duration) with a step of 0.1 μs. The message offset was set in the same manner in all four configurations and the messages are created simultaneously in all applications. The plots show the global minimum, maximum and average software cost and the maximum message latency among all sending applications for all possible CAN message payloads. The software cost is the sum of the sending cost on the sending tile and the receiving cost on the receiving tile. The sending cost comprises the duration between the moment the sending application has created the CAN message and the moment when the controller sends the first message bit on the bus. Analogously, the receiving cost comprises the duration between the moment the last message bit was received on the other side by the controller and the moment when the receiving application gets the message. The sending cost is illustrated in Figure 5 as the time between t_{wrMsg1} and $t_{startMsg1}$ for Tile 1. The maximum message latency is determined by the software cost plus the transmission time on the bus. The large values obtained for Payload = 2,3,6,7 bytes come from sporadic cases in which one application creates a message just after the controller enters the reception mode. The minimum overhead is given by the added duration of the CoMik slots on the sending and receiving side that run between the application and controller slots. Thus, the software cost reflects the execution time of the controller, the communication time between the application and the controller and the TDM schedule in CoMik, but it can occasionally include the blocking time caused by the reception of CAN messages.

In configuration E_2, the number of sending applications and CAN controllers are doubled on each core. The minimum cost scales consequently from 100 to 200 μs. The maximum cost, on the other hand, is given by the alignment between the CAN bit period, the start time of each CAN controller slot and the CAN message offset. In the worst case, the controllers running in the earlier TDM slots detect the new messages and start sending them and the ones running in the later slots enter directly into reception mode before detecting the new messages.

For configuration V_1, the obtained results are almost the same as for E_2, the only difference is in the average cost. In this case it is much higher due to the fact that there is only one controller on each core that arbitrates between two senders. Therefore, the sender with the lower priority will always experience the worst case delay, while in the previous configuration, the varying offset determined this delay only when the messages were created later in the CAN bit period. Hence, using a separate controller for each application leads to a better average performance.

In configuration V_2 we have six sending applications sending messages with a period of 8.35 ms. As we have no external CAN device connected, the results shown characterize only the sending software cost and the corresponding maximum message latency. Here, the minimum cost is around 12 μs and is basically given by the message communication time on the NoC. We implemented a time-based round robin schedule which iterates between the six senders based on the order of their CAN message ID and each time slot is equal to the CAN bit duration (10 μs). Thus the maximum cost is obtained when the sending application has just missed its time slot in the CAN gateway and has to wait until the messages coming from all the other applications have been sent.

B. CAN Time Synchronization

We have extended configuration E_1 with the concepts presented in section VI-C. We allocated a TDM slot to the Synchronized Time Base Manager on each processor. This did not modify the bus speed of 4 kbps due to the fact that the original implementation was designed with a margin of one TDM slot. In other words, the CAN bit period was designed to be equal to 2 + 1 TDM slots, 2 for the difference between two successive CAN driver slots and 1 extra. We have one Timemaster on processor 1 and three Timeslaves on the other processors. The Timemaster sends synchronization messages to the slaves every second. The CAN time messages (SYNC

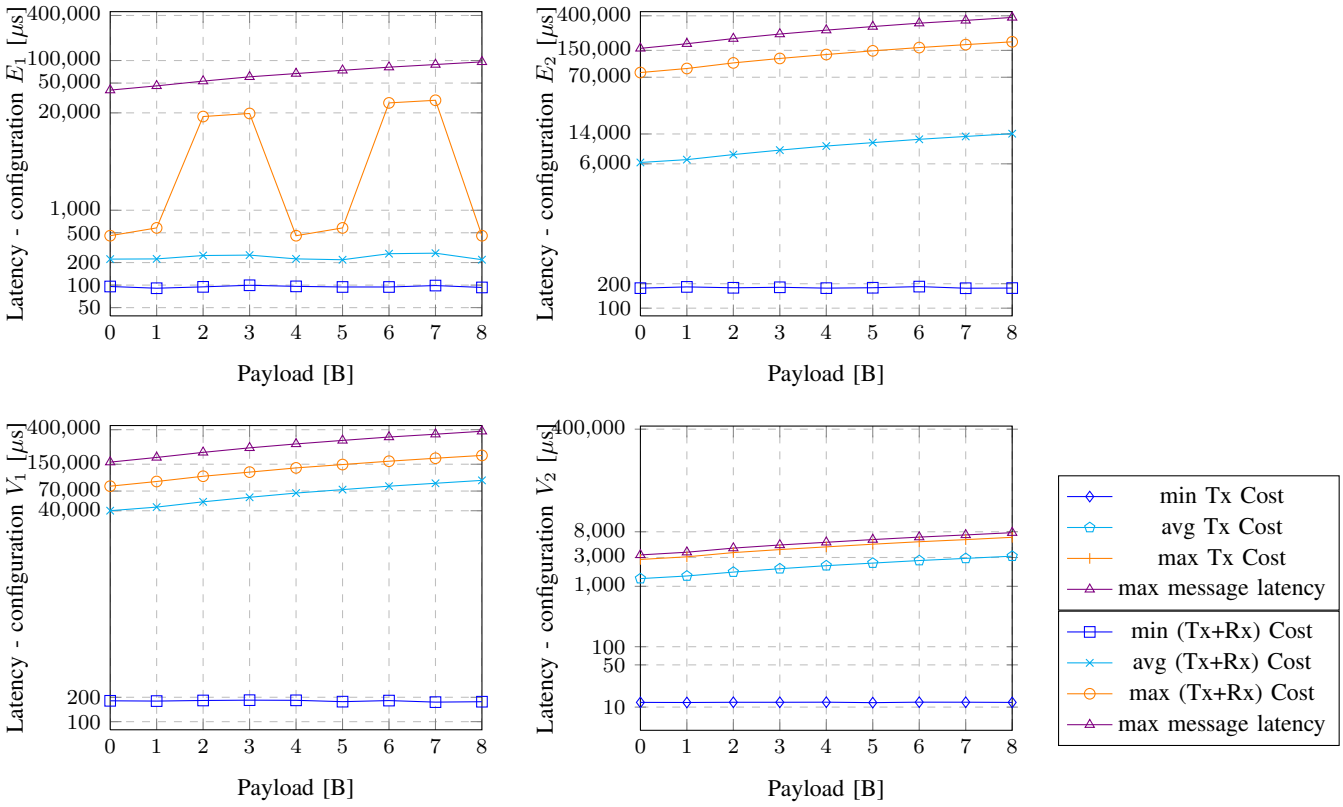

Fig. 17. CAN message and software overhead latency for the four platform configurations

TABLE II
FPGA SYNTHESIS RESULTS

Configuration	Device Utilization		Clock Timing Report	
	# Slice Registers	# Slice LUTs	Net Skew [ns]	Net Delay [ns]
E_1	7%	24%	0.372	1.952
E_2/V_1	7%	24%	0.344	1.924
V_2	12%	42%	0.466	2.048

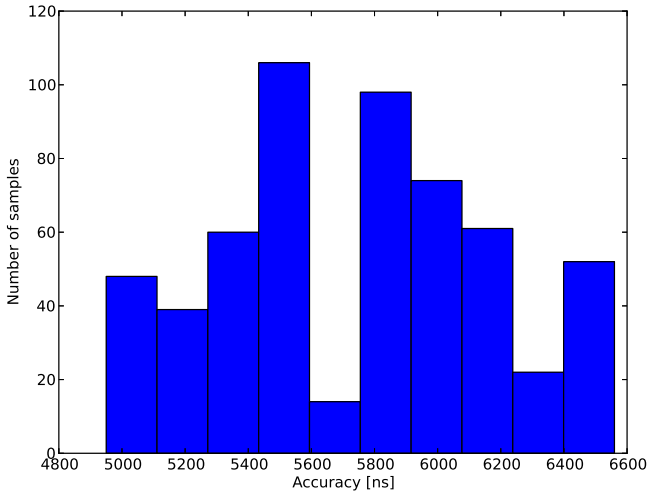

Fig. 18. Time Synchronization Accuracy between Tile 1 and Tile 2

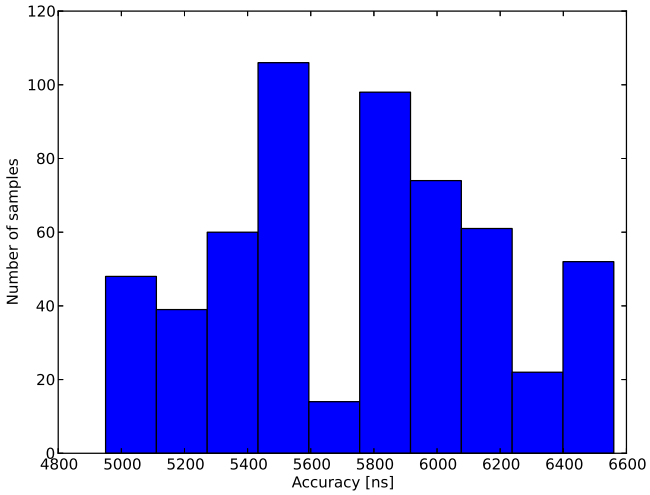

Fig. 19. Time Synchronization Accuracy between Tile 1 and Tile 3

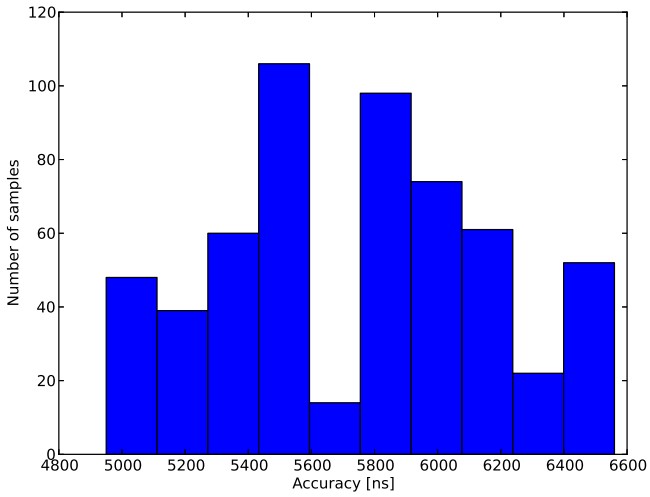

Fig. 20. Time Synchronization Accuracy between Tile 1 and Tile 4

and FUP) have the highest priority, and the priority of the original CAN messages used in the previous experiments for E_1 was incremented with 1.

We ran the code for 10 minutes and measured the accuracy of the synchronization. The accuracy is measured by printing the local synchronized time at the master and slaves at the beginning of the next CAN bit period right after the synchronization process and computing the difference between the master and each slave. It is worth mentioning that for this experiments we used a synchronous platform, hence the beginning of the CAN bits periods are aligned and the printing of the local time is done simultaneously by all the applications. There are two possible factors that affect the synchronization accuracy which can be captured in these experiments and they are both software related. The first one is the time elapsed between capturing the initial timestamp t_{0r} and the corresponding raw time value t_{1r} (as seen in Figure 4) at the slave side. The second one is the time spent to compute the new synchronized time based on the received timestamps at the slave side. In our implementation we optimized the first factor by taking a snapshot of the raw time right after reading the local time, at the master. This eliminates the delay between the completion of the first function call, that returns the local time at the master, and the subsequent call that returns the current raw time. Hence, the only factor that is effectively measured in the experiments is the computation time at the slave side. The obtained values range between 4.95 and 6.56 μs. Figure 18, Figure 19 and Figure 20 show the probability distributions for the obtained accuracies between tile 1 (time master) and tile 2 (time slave), tile 1 and tile 3 and between tile 1 and tile 4, respectively. The distributions are identical since all the slaves run at the same frequency and perform the same operations to compute the new synchronized time. The shown distribution of the accuracy values is caused by the variation in the CAN buffer access time at the slave side, where a polling loop is used to get the latest value received on the bus.

VIII. Conclusions

In this paper we proposed how multiple applications can share a CAN port in a MPSoC platform. The shared CAN port can be on the local processor tile, or on a remote one. As part of our hardware and software design process, we tune the number of applications per CAN port, we explore the possibility of using local and remote CAN ports and we dimension the bit rate of the CAN bus accordingly. Our experimental evaluation shows that configuration V_2 is suitable for applications that require a high performance (bandwidth and latency) while E_1 offers the best average software cost. Configurations E_2 and V_1 offer similar cost and performance, the only difference being that E_2 has a much lower average software cost. Further, the evaluation of our time synchronization for configuration E_1 shows that we can achieve accuracies in the range of several microseconds.

Acknowledgment

This work was partially funded by projects CATRENE ARTEMIS 621429 EMC2, 621353 DEWI, 621439 ALMARVI, SCOTT, IMECH.

References

[1] "AUTOSAR release 4.2 - SWS CANDriver," Tech. Rep.
[2] "AUTOSAR release 4.2.2 - Specification of Synchronized Time Base Manager," Tech. Rep.
[3] "AUTOSAR release 4.2.2 - Specification of Time Synchronization over CAN," Tech. Rep.
[4] "ISO11989-1:2015 road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling," Tech. Rep.
[5] "IEEE standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pp. 1–269, July 2008.
[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*, vol. 37, no. 5, 2003.
[7] E. G. Friedman, "Clock distribution networks in synchronous digital integrated circuits," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 665–692, May 2001.
[8] K. Goossens, M. Koedam, A. Nelson, S. Sinha, S. Goossens, Y. Li, G. Breaban, R. van Kampenhout, R. Tavakoli, J. Valencia, H. Ahmadi Balef, B. Akesson, S. Stuijk, M. Geilen, D. Goswami, and M. Nabi, "NOC-based Multi-Processor Architecture for Mixed Time-Criticality Applications," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds. Springer, 2017.
[9] C. Herber, D. Reinhardt, A. Richter, and A. Herkersdorf, "HW/SW trade-offs in I/O virtualization for controller area network," in *Design Automation Conference (DAC)*, 2015.
[10] C. Herber, A. Richter, T. Wild, and A. Herkersdorf, "A network virtualization approach for performance isolation in controller area network (CAN)," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
[11] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, 2003.
[12] H. T. Lim, D. Herrscher, L. Völker, and M. J. Waltl, "IEEE 802.1AS Time Synchronization in a Switched Ethernet based In-Car Network," in *2011 IEEE Vehicular Networking Conference (VNC)*, Nov 2011, pp. 147–154.
[13] A. Nelson, A. B. Nejad, A. Molnos, M. Koedam, and K. Goossens, "CoMik: A predictable and cycle-accurately composable real-time microkernel," in *Design Automation and Test in Europe Conference (DATE)*, 2014.
[14] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens, "C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, 2002.

[15] R. Obermaisser, "CAN emulation in a time-triggered environment," in *International Symposium on Industrial Electronics (ISIE)*, vol. 1, 2002.
[16] D. Reinhardt and M. Kucera, "Domain Controlled Architecture - A new approach for large scale software integrated automotive systems," in *International Conference on Pervasive Embedded Computing and Communication Systems (PECCS)*, 2013.
[17] O. Sander, T. Sandmann, V. V. Duy, S. Bähr, F. Bapp, J. Becker, H. U. Michel, D. Kaule, D. Adam, E. Lübbers, J. Hairbucher, A. Richter, C. Herber, and A. Herkersdorf, "Hardware virtualization support for shared resources in mixed-criticality multicore systems," in *Design Automation and Test in Europe Conference (DATE)*, 2014.
[18] R. Stefan, A. Molnos, A. Ambrose, and K. Goossens, "dAElite: A TDM NoC supporting QoS, multicast, and fast connection set-up," *Computers, IEEE Transactions on*, vol. 63, no. 3, 2014.

Gabriela Breaban is a PhD student in the Electrical Engineering department at the Technical University of Eindhoven. She obtained her Bachelor degree in Electronics and Telecommunications at the Technical University of Iasi, Romania in 2009. Afterwards, she completed her Master studies in Digital Radio Communications in 2011 at the same university. Her work experience includes 2 years as an Embedded Software Developer in the automotive industry and another 2 years as a Digital Design Verification Engineer in the semiconductor industry. Her research interests are in the areas of formal models of computation, time synchronization and embedded systems architecture.

Martijn Koedam received his master degree in Electrical Engineering at the Technical University of Eindhoven. His work experience includes software development in the audio industry, design and implementation of a regression test framework for POS systems, proof of concept security hack for payment systems and evaluating wireless ticketing systems. Since 2011 he works as a researcher and developer at the same university. His research interests include design, modeling, and simulation of embedded Systems-on-Chip, composable, predictable, real-time and mixed-criticality systems and execution models.

Sander Stuijk received his M.Sc. (with honors) in 2002 and his Ph.D. in 2007 from the Eindhoven University of Technology. He is currently an assistant professor in the Department of Electrical Engineering at Eindhoven University of Technology. He is also a visiting researcher at Philips Research Eindhoven working on bio-signal processing algorithms and their embedded implementations. His research focuses on modelling methods and mapping techniques for the design and synthesis of predictable systems.

Kees Goossens has a PhD from the University of Edinburgh in 1993 on hardware verification using embeddings of formal semantics of hardware description languages in proof systems. He worked for Philips/NXP from 1995 to 2010 on real-time networks on chip for consumer electronics. He was part-time full professor at Delft university from 2007 to 2010, and is now full professor at the Eindhoven University of Technology, researching composable, predictable, low-power embedded systems, supporting multiple models of computation. He is also system architect at Topic Products. He published 4 books, 170+ papers, and 24 patents.