# Virtualization and Emulation of a CAN Device on a Multi-Processor System on Chip

Gabriela Breaban, Martijn Koedam, Sander Stuijk, Kees Goossens
Eindhoven University of Technology
Email:{g.breaban, m.l.p.j.koedam, s.stuijk, k.g.w.goossens}@tue.nl

***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. 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. 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 ECU's 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 [1] and it leads to increased computational requirements. To cope with this demand, the use of multi-core platforms has been proposed in literature [1]. Multi-core platforms can come as either Commercial-Off-The-Shelf (COTS) platforms or as Systems on Chip (SoCs).

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 [2]. 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 [3].

In terms of software emulation, the CAN interface has been build on top of specific hardware architectures such as the Time Triggered Architecture (TTA) [4]. 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.

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 Field-Programmable Gate Array (FPGA) platform.

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

## II. Related work

Herber et al. propose software CAN controller virtualization methods inspired from server environments [3]. 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 [5] to schedule multiple applications on the CAN client cores and communicate the CAN message to the CAN gateway using C-HEAP FIFOs [6] 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 [7], 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 [8]. 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 [9]. 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.

## 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, 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

TABLE I
VIRTUALIZATION AND EMULATION PLATFORM CONFIGURATIONS

| Configuration | $V_1$ | | | | $V_2$ | | | | $E_1$ | | | | $E_2$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CAN Bus Baud Rate [kbit/s] | 2 | | | | 100 | | | | 4 | | | | 2 | | | |
| Number of (applications + controllers) per core | Core 1 2+1 | Core 2 2+1 | Core 3 2+1 | Core 4 2+1 | Core 1 2+0 | Core 2 2+0 | Core 3 2+0 | Core 4 0+1 | Core 1 1+1 | Core 2 1+1 | Core 3 1+1 | Core 4 1+1 | Core 1 2+2 | Core 2 2+2 | Core 3 2+2 | Core 4 2+2 |
| Number of CAN ports per tile | Tile 1 1 | Tile 2 1 | Tile 3 1 | Tile 4 1 | Tile 1 0 | Tile 2 0 | Tile 3 0 | Tile 4 1 | Tile 1 1 | Tile 2 1 | Tile 3 1 | Tile 4 1 | Tile 1 2 | Tile 2 2 | Tile 3 2 | Tile 4 2 |
| Number of applications per CAN port | 2 | | | | 6 | | | | 1 | | | | 1 | | | |

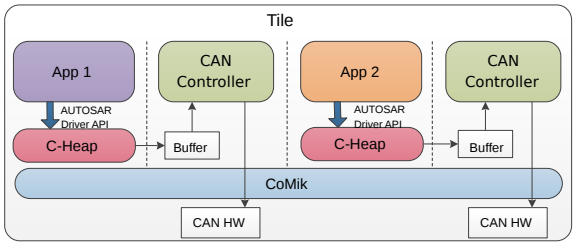


Fig. 1. CAN Configuration E2 - System Architecture of a tile

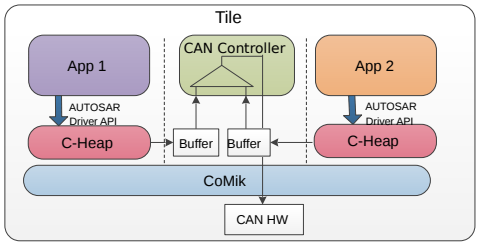


Fig. 2. CAN Configuration V1 - System Architecture of a tile

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 [10] 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 IV.

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.

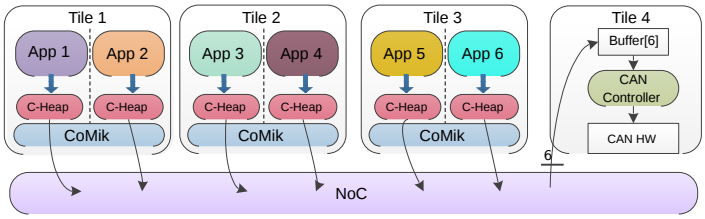


Fig. 3. CAN Configuration V2 - Using one tile as a CAN gateway

### D. Platform Configuration $V_1$

Configuration $V_1$ is similar to $E_1$, the main difference being 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, which 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 [11] provides contention-free communication; therefore the message communication time

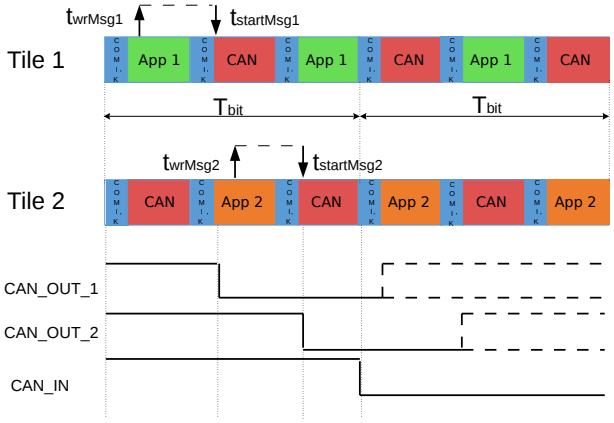


Fig. 4. Timing Diagram for configuration $E_1$ - Emulated CAN on top of CoMik

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. Implementation

We have implemented the physical layer of the CAN interface as a hardware module that 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 rate is obtained by dividing the processor clock frequency. All the tiles run synchronously on the same clock domain.

### 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 [12], 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 Application Programming Interface (API) according to the AUTOSAR standard.

### B. Implementing a CAN Controller on the Virtual Processor

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 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:

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

Figure 4 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 an application/CAN sub-slot. In the CoMik sub-slot the context switch operations are performed. 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 and it is equal to three slots. We can see that each application writes a transmit message in its corresponding buffer 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 and the start of every CAN bit period and it reflects the result of all the CAN output lines on the platform. The CAN controller synchronizes 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.

### C. Bare-metal Implementation of the CAN Controller

Configuration $V_2$ illustrates the possibility of allocating the entire processor to the CAN controller. Figure 5 shows the

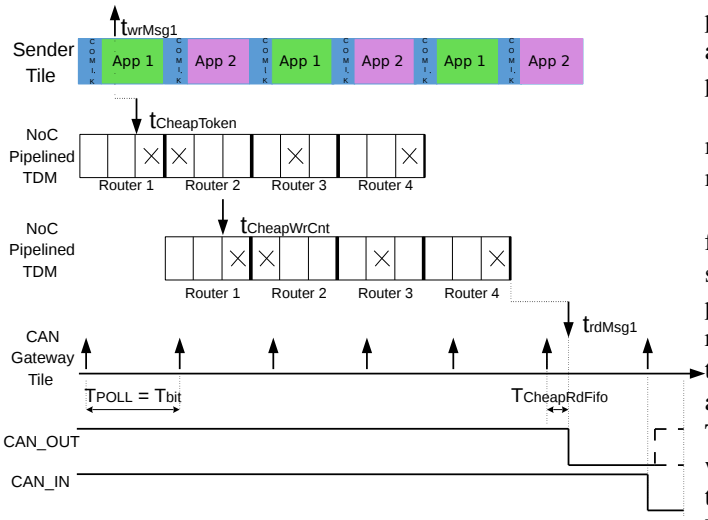


Fig. 5. Timing Diagram for configuration $V_2$ - Bare-metal implementation of the CAN Controller and the Communication of CAN messages via NoC

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, 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 worst case execution time needed to send one CAN bit, which can be determined by accessing the communication FIFOs or computing the CRC.

## V. Experiments

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.

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

Figure 6 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$\mu$s (the TDM slot duration) with a step of 0.1$\mu$s. 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 4 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$\mu$s. 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

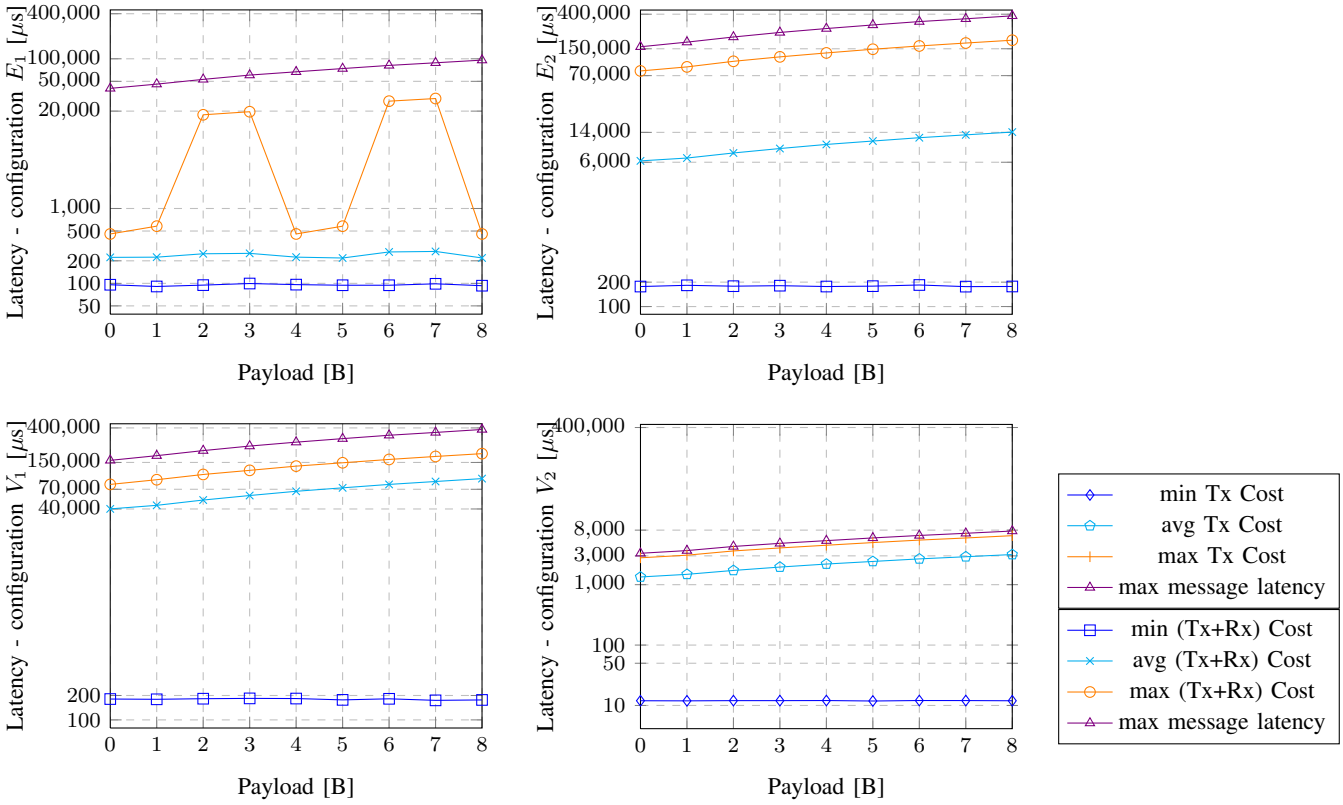


Fig. 6. CAN message and software overhead latency for the four platform configurations

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$\mu$s 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$\mu$s). 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.

## VI. 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. We evaluate each solution and we show the obtained software cost and end-to-end latency for the CAN messages.


## Acknowledgment

This work was partially funded by projects CATRENE CA505 BENEFIC, CA703 OpenES, CT217 RESIST; ARTEMIS 621429 EMC2, 621353 DEWI, and 621439 ALMARVI.


## References


[1] D. Reinhardt *et al.*, "Domain Controlled Architecture - A new approach for large scale software integrated automotive systems," in *PECCS*, 2013.

[2] P. Barham *et al.*, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, 2003.

[3] C. Herber *et al.*, "HW/SW trade-offs in I/O virtualization for controller area network," in *DAC*, 2015.

[4] H. Kopetz *et al.*, "The time-triggered architecture," *Proceedings of the IEEE*, 2003.

[5] A. Nelson *et al.*, "CoMik: A predictable and cycle-accurately composable real-time microkernel," in *DATE*, 2014.

[6] A. Nieuwland *et al.*, "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.

[7] O. Sander *et al.*, "Hardware virtualization support for shared resources in mixed-criticality multicore systems." in *DATE*, 2014.

[8] C. Herber *et al.*, "A network virtualization approach for performance isolation in controller area network (CAN)," in *RTAS*, 2014.

[9] R. Obermaisser, "CAN emulation in a time-triggered environment," in *ISIE*, vol. 1, 2002.

[10] "AUTOSAR release 4.2 - SWS CANDriver," Tech. Rep.

[11] R. Stefan *et al.*, "dAElite: A TDM NoC supporting QoS, multicast, and fast connection set-up," *Computers, IEEE Transactions on*, vol. 63, no. 3, 2014.

[12] "ISO11989-1:2015 road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling," Tech. Rep.