Hardware Virtualization Support for Shared Resources in Mixed-Criticality Multicore Systems

Oliver Sander*, Timo Sandmann*, Viet Vu Duy*, Steffen Bähr*, Falco Bapp*, Jürgen Becker*,
Hans Ulrich Michel†, Dirk Kaule†, Daniel Adam†, Enno Lübbers‡, Jürgen Hairbucher‡,
Andre Richter§, Christian Herber§ and Andreas Herkersdorf§
*Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, firstname.lastname@kit.edu
†BMW F+T, München, Germany, firstname.lastname@bmw.de
‡Intel GmbH, München, Germany, firstname.lastname@intel.de
§Technical University Munich (TUM), München, Germany, firstname.lastname@tum.de

***Abstract*—Electric/Electronic architectures in modern automobiles evolve towards an hierachical approach where functionalities from several ECUs are consolidated into few domain computers. Performance requirements directly lead to multicore solutions but also to a combination of very different requirements on such ECUs. Using virtualization in addition is one promising way of achieving segregation in time and space of shared resources. Based on examples taken from the automotive domain several concepts for efficient hardware extensions of coprocessors and I/O devices are shown in this contribution. These provide mechanisms to ensure quality of service (QoS) levels in terms of execution time, throughput and latency. The resulting infotainment architecture is a feasibility study and is integrated into a vehicle demonstrator as centralized infotainment platform (VCT).**

I. Introduction

Information and communication systems play an ever increasing role in satisfying customer requirements in today's high-end automobiles. Especially in the last two decades the automobile industry had to face an increasing amount of functions, mainly realized through electronic systems and software. For this reason new electronic control unit (ECU) architectures and interfaces are needed [1].
One possible evolutionary step is a domain specific centralisation, namely high integration of functions into a small number of high performance ECUs (domain control units). Such electric/electronic network architecture has the potential of a harmonisation among different product lines. Higher energy efficiency at lower costs is just one advantage. However the huge amount of functions on one single domain computer imposes some additional challenges. Each software has its specific requirements, especially regarding safety (e.g. ISO 26262) or security (common criteria), thus demanding a strict separation between diverse functions. One possible solution is the use of virtualization techniques on these centralized computing platforms [1]. Due to the high integration of complex functionality on one single ECU also a huge amount of computing power must be provided.
Multicore is the most promising solution to provide this high level of performance [2]. But multicore also introduces additional side effects concerning real-time behaviour, access contention, and overall system complexity mostly due to shared resources like memories, coprocessors or I/O peripherals [3]. Accesses to shared functions need to be coordinated to allow efficient and predictable sharing of these dedicated resources. As soon as the traditional exclusive access paradigm of single-function platforms is lifted, a deeper analysis of the hardware resources to be shared is necessary.
The focus of our current work is to demonstrate the feasibility of multicore virtualization as a key enabling technology for the centralisation of functionality from different infotainment ECUs onto one single commercial off-the-shelf (COTS) hardware platform - the Virtualized Car Telematics (VCT) computer. This consolidation includes safety relevant instrument cluster information as well as customer specific applications that are not qualified by the vehicle manufacturer. Those functions share dedicated hardware resources among several virtualized partitions. Although sharing can be managed by the hypervisor, this solution may not be able to meet strict performance and latency requirements due to the software management overhead. The approaches presented in this paper overcome these limitations by means of virtualization hardware support for shared resources, namely coprocessors and I/O peripherals.

II. VCT Demonstrator Framework

The VCT demonstrator features an Intel-based multicore system, which is connected to various input/output devices like touch controller, steering wheel control buttons, several displays and different network types (CAN, Ethernet, Wi-Fi, UMTS/LTE). There are support computers simulating the environment and/or acting as proxy for the communication between the demonstrator computer and the actual vehicle. The complete hardware platform including support devices (power supply, input, display and network switches, etc.) are mounted on a metal rack which is built into the trunk of a BMW 320d touring car (see Fig. 1).

A. Hardware

The Intel-based computer features an i7-3770T multicore CPU (3rd Generation Intel Core i7), mounted on an Intel DQ77MK board. The CPU features four physical cores and supports up to eight parallel threads via simultaneous multithreading. It also features hardware support for processor virtualization (VT-x), extended page tables (EPT), and I/O virtualization (VT-d). The PCI Express (PCIe) 3.0 x16 bus add-in card connector, driven directly by the CPU, is used for connecting a FPGA board as described below.

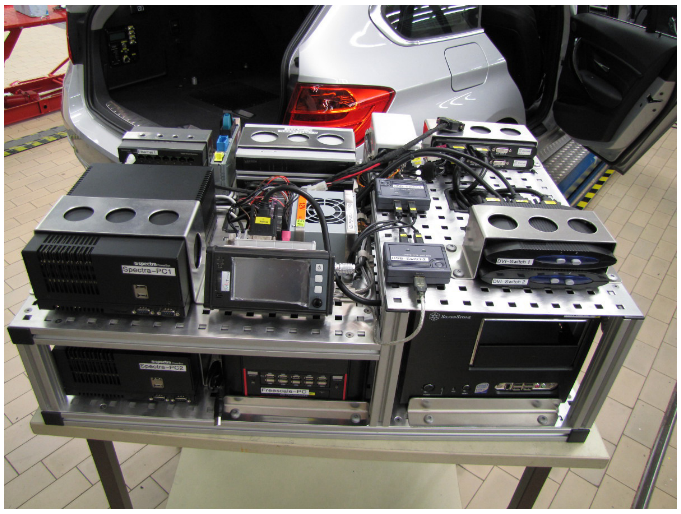

Fig. 1. Hardware Setup Mounted on the Rack

The FPGA is a Xilinx Virtex 7 that is mounted to a Xilinx VC709 board [4]. The FPGA is used for the hardware implementation of the shared devices - one self-virtualized I/O controller for the controller area network (CAN) and one cryptographic coprocessor for vehicle-to-X (Car2X) communication - and their virtualization support based on the SR-IOV standard [5]. Furthermore, a custom extension board is connected via the FMC HPC connector of the FPGA board and delivers two CAN interfaces, which are needed by the CAN controller, as well as several debug interfaces.
PCIe is one of the most popular bus standards used in microprocessor systems and is now emerging in high performance embedded systems. Its attractiveness stems from the capability to execute high-speed data transfers. Recently, due to the trend towards virtualization, the demand for using a single PCIe device in several virtual machines (VMs) arose. This was typically dealt with by contacting the governing hypervisor for each access. However this has an undesired impact on the resulting latencies [6]. Motivated by this drawback the SR-IOV standard arose. It allows a device to facilitate access to its resources across several VMs by making use of PCIe functions. Those are partitioned in two types: physical functions (PFs), typically used by hypervisor for configuration, and virtual functions (VFs), used by the VMs for I/O. The Xilinx Virtex 7 FPGA featured on the VC709 board supports the SR-IOV specification with up to 2 PFs and 6 VFs, which are fully utilized and divided evenly between the two shared devices of the demonstrator.

B. Software

On the software side, a virtualized architecture with 3 VMs running on top of the Wind River (WR) hypervisor are deployed (see Fig. 2). While the trusted VM based on Ubuntu contains only software from the vehicle manufacturer (OEM) which is supposed to be safety critical and free from malicious code, an untrusted VM running Android is available for the integration of arbitrary user applications. Furthermore, there is a server VM running a Linux distribution which provides services used by the remaining VMs, most of them with respect to other shared resources which are not the focus of this work

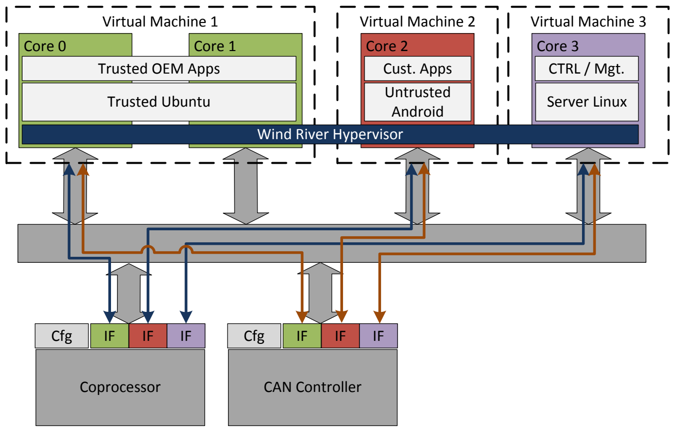

Fig. 2. Software Architecture of the VCT Intel-based Demonstrator Computer

e.g. audio, graphics and mass storages.
The WR hypervisor takes care of all software aspects of the virtualization implementation on the target platform. The core hypervisor includes the control of the cores, the memory management unit, the interrupt logic, and the management of VMs. Based on the core function of WR hypervisor, physical devices can be exposed to the operating system in a VM in different ways. In particular these cases can be distinguished:

1) Passthrough: a device is used exclusively by one VM. The guest operating system's driver controls the device completely.

2) Shared: a device is used by more than one VM. The primary driver for the device is located inside the hypervisor; this is called an actual device driver (ADD). This ADD controls the device and also decides on the actual sharing. For each VM there is a virtual device in the hypervisor which moderates the communication between the ADD and the driver in the VM's operating system.

3) Combined: an SR-IOV capable device is a good example for this case. SR-IOV has a management interface (PF) and VM interfaces (VFs) realized in hardware. Such a device can be treated as passthrough for the VFs and the management function is realized by an ADD in the hypervisor. In the VCT computer platform the self-virtualized CAN controller and the Car2X coprocesor fall into this category of device sharing, with the PFs assigned to the hypervisor and each VM having exclusive access to one VF of the CAN controller and one VF of the Car2X coprocessor. However SR-IOV is currently not directly supported by the WR hypervisor so driver extensions have to be implemented.

III. PCIe SR-IOV Hardware Adapter

Usage of PCIe on the FPGA is provided by Xilinx's PCIe Integrated Endpoint IP core [7]. This PCIe Endpoint offers solely one single shared interface for all transfers of all VFs and PFs. Thus in order to enable usage for all participants i.e. coprocessors and I/O modules with their functions, a communication infrastructure was established. Its main goals are to allow participants to maintain an interface independent of the endpoint, decoupling participants from its specifics, and allow high performance communication. Our infrastructure developed to achieve this is shown as a SysML model in Fig. 3.

A. PCIe AXI Adapter

All slave transfers to all VF/PF devices are passed through the PCIe Endpoint's completer interface. In the infrastructure it's entirely handled by the PCIe AXI Adapter module. It mainly acts as an facilitator of transfers. For this purpose one interface per VF is provided to all participants. To broaden the number of components applicable, we chose to use AXI [8] for the interconnect between PCIe AXI Adapter and participants.

B. DMA Control

Outgoing transfers intending direct memory access (DMA) and their corresponding data in response are managed through the requester interface. This is managed in the infrastructure by the DMA control module. For every participant in need of DMA, for example the Car2X coprocessor, a separate interface is provided, empowering it of signaling DMA requests. In case such a request is signaled, the DMA control module is reading the specifics of the transfer e.g. target memory address, directly from the requesting participant's memory. This is done by usage of AXI for every function interface present, similarly to the PCIe AXI Adapter. In case of returning data following read requests, the DMA control module is writing the data into the respective participant's memory on its own and is going to signal its arrival.
The capability of accessing participants as a master, is additionally used to perform communication between them. By using the same interface as for ordinary DMA transactions, a participant is able to directly address neighboring ones without PCIe communication. To make the DMA control module configurable by the hypervisor it is connected to the PCIe AXI Adapter. This way configuration parameters like the PCIe packet size can be defined.

C. Interrupt Generator

In PCIe interrupts are asserted by memory write requests. The address and data sent are used for identification of the concrete interrupt line to be asserted. Additionally a device is capable of sending different interrupts with Message Signaled Interrupts (MSI)/MSI-X. Using that variant requires the selection of the right address and data pair for each interrupt request issued by a participant.
The selection and transmission of interrupts is handled by the interrupt generator module. This allows the separation of the specific interrupt mechanisms of PCIe from the coprocessor. As a result a more generic interface for indicating interrupt requests is possible.

D. Temporal Isolation and PCIe Passthrough I/O

The previously stated advantages of PCIe Passthrough solutions, however, have the side-effect of introducing problems regarding temporal isolation. This is because different VMs share the same interconnect for accessing their passthrough device or VF, while at the same time, there is no means of supervising the access patterns. In [9], we show the possible impact of disregarding this lack of temporal isolation for a PCIe Passthrough setup composed of COTS hardware. It is presented how a single malfunctioning or malintent VM, which is assigned to a passthrough device or VF, can exploit the lack of temporal isolation in such a way that the performance of

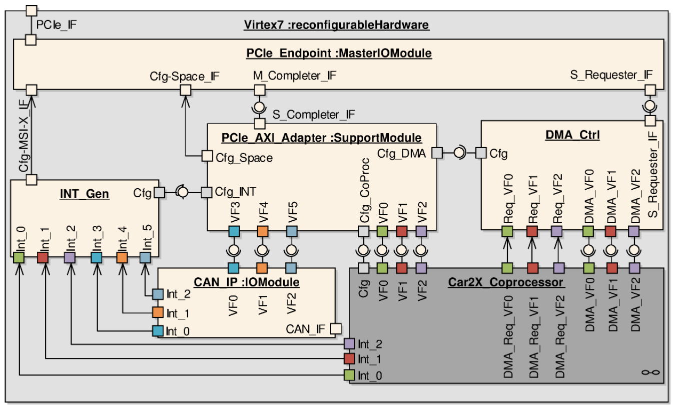

Fig. 3. SysML Model of the Resource Sharing Layer for Coprocessors and I/O Peripherals

every other I/O device in the system suffers severely. Naturally, the problem also applies to scenarios without malfunctioning or malintent VMs, because it must be ensured that standard communications of multiple VMs do not interfere on the shared interconnect. Within the scope of the presented architecture, we identified three approaches for a possible enablement of temporal isolation for PCIe Passthrough:

1) Modern CPUs have built-in performance counters which allow monitoring of CPUs at core granularity. These monitoring facilities can be leveraged by the hypervisor to enforce I/O-access policing for VMs which run on different cores. For related problems, performance counters have already been proven useful [3].
2) The PCIe standard defines a full set of QoS features, including Virtual Channel (VC) usage, combinable with multiple arbitration schemes, which could be utilized to enforce temporal isolation. Unfortunately, VC usage is defined optional, which is why there is currently no COTS hardware available providing these features.
3) Future I/O devices which are built for sharing (SR-IOV) or passthrough usage could employ facilites to detect abusive or harmful access patterns and launch appropriate countermeasures, e.g. discarding packets which are considered harmful and reporting the packet source to the hypervisor (which then engages appropriate countermeasures).

IV. Hardware Virtualization of Coprocessors

As an example for a coprocessor that is shared among diverse partitions we choose a cryptographic coprocessor for Car2X communication. It is important to note that only coprocessors can deal with the huge amount of Car2X messages and the complexity of the cryptographic operations needed by the necessary signature generation and verification. Furthermore, due to resource constraints, there may be only one coprocessor available in the system, making it a shared resource. The VCT demonstrator makes use of the virtualization concept and its implementation to share the cryptographic coprocessor between the different software partitions in a safe, secure and efficient manner.

A. *Temporal Segregation and QoS*

In current multicore architectures parallel access requests for shared resources from different partitions will typically be executed in a first-come-first-served manner based on their arrival time at the shared resource. Besides the disadvantage of no prioritization of accesses from the partitions, this also eliminates the determinism due to the fact that in a multicore system accesses from different partitions, e.g. cores, can occur in real parallelism. To overcome these drawbacks, we integrated a quality of service aware hardware based scheduling mechanism. Because of the low-latency demands, the scheduling is done completely by a hardware module, which is integrated in the coprocessor interface layer. Another benefit of implementing the complete scheduling subsystem in hardware is motivated by safety requirements; a hardware implementation can be more robust against segregation failures introduced by erroneous software components like kernel-drivers at operating system or hypervisor level. Furthermore the scheduling approach stated below is fully transparent for the (software) layers above and on top of the coprocessor hardware interface. The scheduling concept is a hardware extension of the coprocessor and part of the coprocessor resource sharing extension layer introduced in the following paragraph.

B. *Coprocessor Architecture*

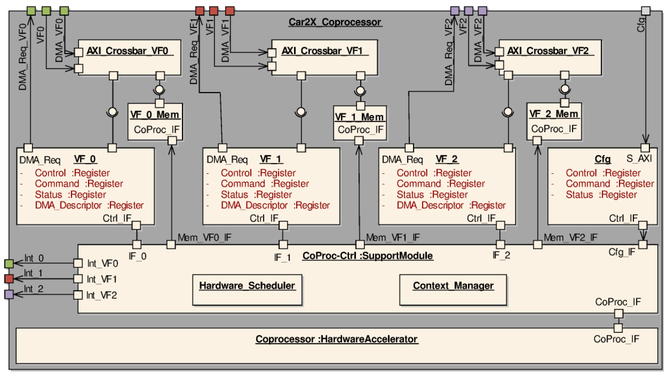

Fig. 4. Resource Sharing Layer for Coprocessors in Multicore Systems

The proposed architecture for shared coprocessors shown in Fig. 4 consists of one configuration interface (Cfg) and three access interfaces for virtual functions (VF0 to VF2) supporting the system partitioning concept presented in section II. Access requests from cores as well as data exchanges corresponding to different partitions are to be routed through different interfaces. This way spatial segregation of partitions can be guaranteed, if the system uses a Memory Management Unit (MMU). The interfaces are available through AXI busses from two different sources each: the completer interface described in section III and the DMA engine. Therefore each AXI crossbar switch contains two master inputs (VF0 as completer access interface and DMA_VF0 as DMA interface). Both bus masters are able to access the virtual function control module (VF0 to VF2) as well as a virtual function local memory (e.g. for data, DMA descriptors etc.). Incoming requests will be scheduled for execution on the coprocessor according to a defined configuration by a hardware scheduler that is part of the coprocessor control module shown in Fig. 4 as CoProc-Ctrl. Potential context switches, that are necessary to achieve the scheduling policy, are handled by a dedicated context manager within the coprocessor control module.

C. *QoS Ensurance for Mixed-Criticality Multicore Systems*

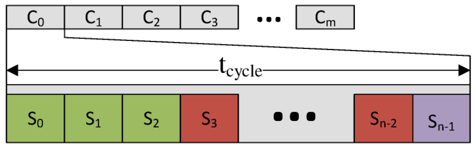

Fig. 5. QoS Scheduling Algorithm for Safety-Critical Systems

The developed scheduling approach partitions the execution time of the coprocessor into scheduling periods c_α that consist of a constant number of slots s_i, each having a fixed assignment to a specific partition p_j (see Fig. 5). In total we assume a number of n slots. The set of all slots is given by $S = \{s_0, s_1, \ldots, s_n\}$. The execution time of one cycle is defined as $t_{cycle} = n * t_{slot}$. P is the set of all partitions and each slot belongs to exactly one partition:

$$p_j = \{s_a, \ldots, s_b\} : a, b < n; j < |P|$$

$$p_i \cap p_j = \emptyset \forall p_i, p_j \in P; i \neq j; \bigcup_{r=0}^{|P|} p_r = S$$

The execution can then be defined as a set

$$EX = \{c_0, c_1, \ldots, c_\Omega\} : t_{c_i\_start} < t_{c_{i+1}\_start}$$

where c_Ω is the last cycle before the system is shut down. In general it is not expected, that a partition makes use of all assigned slots during EX. Therefore an unused slot will be automatically granted to the next partition with a request hence improving the efficiency of the execution. A minimum service guarantee of at least $\frac{|p_j|}{n}\%$ to partition p_j can be given in a worst case scenario. Whenever not all slots are used, a low priority task gets more slots and therefore achieves lower execution times. Partition request priorities can be managed by adjusting the number of slots $|p_j|$ that are assigned to a partition p_j. The static assignment of processing time per scheduling period allows fulfilling the requirement of temporal segregation, determinism and real-time behavior.

D. *Integration of Secure Car2X Communication into the VCT Demonstrator*

In the following, the concept for the integration of the secure Car2X communication into the VCT demonstrator will be described. A Car2X control application running inside the Linux server VM as described in section II extracts the Car2X messages from incoming UDP packets and passes them to the cryptographic coprocessor to verify authentication and trustworthiness. The Car2X software stack of the trusted Linux VM processes the messages and shows the contained information on a display, for example, keeping the real-time requirements. Furthermore the trusted Linux VM as well as the Android VM shall be able to send Car2X messages to other communication partners, which have to be processed by the cryptographic coprocessor first in order to calculate their valid signatures. Therefore one interface to the coprocessor is assigned exclusively to each partition. The Car2X app within the Android partition represents other future Car2X user applications, which may utilize and potentially misuse the Car2X communication interface and thus the coprocessor. To demonstrate the feasibility of the developed temporal isolation and QoS scheduling concept for coprocessors that is described below, the Android Car2X app tries to send out the maximum amount of messages at the maximum speed available and it is handled in a best-effort manner.

E. Selected Results

To investigate the scalability and robustness of our scheduling approch, we run a simulation setup with an increasing number of best-effort partitions. The first setup consists of two active partitions, a request length of 2000 computation cycles representing a safety-critical partition p_1 with 20 assigned slots and 500 cycles for a best-effort partition p_2 with 10 slots, each slot with a length of 110 cycles. The first partition sends regular requests, which are assumed to be relatively seldom but highly safety-critical. Requests from the other partitions are generated whenever its previous request was finished. Subsequently we added more partitions to the system, each of them gets 1 slot, which is deducted from the slot amount of partition p_2. Therefore we could add up to 9 additional partitions as listed in the first column of table I.

TABLE I. EVALUATION OF THE SCALABILITY

	Partition p_1			Partition p_2			Partition p_n	
$\|P\|$	max	min	avg	max	min	avg	max	min
2	3204	2015	2894	2511	500	1015	-	-
3	3205	2015	2892	2623	500	1085	16500	16491
4	3205	2010	2898	2731	500	1173	16500	16381
5	3204	2015	2896	2848	500	1265	16495	16271
6	3205	2015	2896	2958	500	1385	16500	16161
7	3205	2015	2897	3068	500	1531	16500	16051
8	3204	2015	2895	3187	500	1693	16500	15941
9	3205	2015	2892	3297	500	1915	16500	15831
10	3205	2015	2898	6283	500	2173	16495	15721
11	3204	2015	2889	6502	500	2556	16500	16491

For each partition the aggregated maximum, minimum and average values of a 100.000 cycles lasting simulation are shown in table I. As a first outcome, the variance of the response times for safety-critical Partition p_1 is minimal. This shows the guaranteed quality of service, which ensures an interference-free operation of the safety-critical partition regardless of the number of active best-effort partitions. As expected, the average response time of partition p_2 increases nearly linearly with the number of added partitions, because the number of slots available for partition p_2 is reduced equally. The response times of partition p_n, which represents all remaining partitions, are uneffected, because all these n partitions get 1 slot for their computations.

V. HARDWARE VIRTUALIZATION OF I/O PERIPHERALS

Automotive communication is constrained by real-time requirements. Because SR-IOV offers the higher predicatability and lower latencies than other device virtualization approaches, we developed a self-virtualized I/O controller [10] for CAN, the most common fieldbus used in automotive settings. This section gives an overview of the proposed architecture, temporal isolation of the virtual CAN controller instances, and security extensions.

A. Architecture of a Self-Virtualized CAN Controller

The virtualized CAN controller is connected to the host system through one physical and multiple virtual interfaces (PF/VF) as described in Section III. The VFs allow data path operations (Tx/Rx) to be executed through abstract interfaces. While VFs can provide status information like counters to the VMs, it is not possible for VMs to manipulate memory contents or settings directly.

Privileged requests are issued through the PF. The PF configures the VFs (e.g. by assigning an amount of message memory to a VF) and the protocol specific settings like the CAN bus frequency. The PF driver will be operated by the hypervisor or a privileged VM as depicted in Fig. 6.

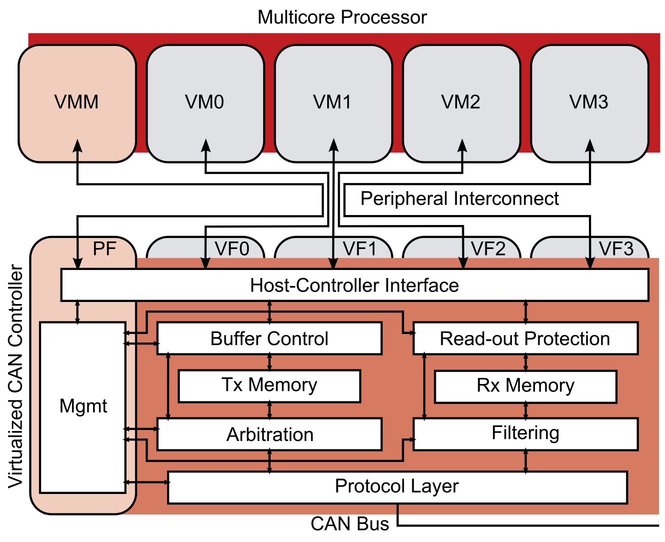

Fig. 6. Architectural overview of a multicore processor connected to the virtualized CAN controller. VMs can access the CAN bus through virtual functions (VFs) that are managed by the hypervisor through the physical function (PF).

A key aspect in the virtualization of CAN controllers is how the access towards the CAN bus is divided among the virtual controllers. Normally, physical controllers compete on the CAN bus in a bitwise arbitration scheme, which is based on the message ID of the frame (with the lowest ID having the highest priority). Emulating this behavior in the arbitration module creates a setup, in which virtual controllers compete with other CAN nodes on the CAN bus.

This is realized by providing priority queues within the Tx memory for each virtual controller, which are freely allocated in a RAM module. During an interframe spacing on the CAN bus, the arbiter module finds the highest priority message and forwards it to the CAN bus. Using other data structures like FIFOs would result in priority inversions, causing increases in worst case latencies that make real-time operation impossible. Incoming CAN frames are discarded or accepted based on a list of filters. Configurable interrupts can be issued to all receiving VMs. This relieves the VMs from the task of message sorting and avoids duplicating the data locally within the I/O device.

Each VM can read a preconfigured subset of all frames. This restriction is ensured by the read-out protection module.

B. Temporal Isolation of Virtual CAN Controllers

Hardware modules are shared in order to guarantee resource-efficiency. This makes the architecture prone to temporal interference among the VFs. To resolve this issue, we introduce a scheduling mechanism within the host-controller interface in order to isolate the behavior of each virtual CAN controller.
Requests from the VMs may arrive at the host-controller interface at peak rates that cannot be served immediately. Therefore requests have to be buffered in FIFOs. To enable a separation of virtual interfaces, a distinct buffer is provided for each virtual controller v. A scheduling algorithm ensures that all buffers are served as shown in Fig. 7.
The algorithm is based on a time-based, weighted round-robin

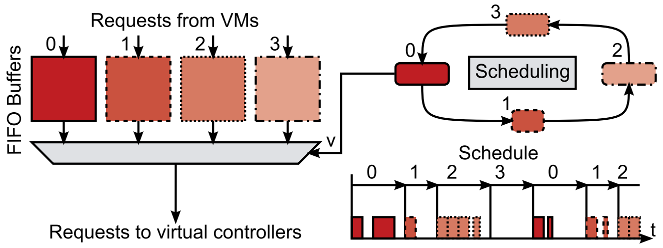

Fig. 7. Scheduling of virtual interfaces: Requests from VMs are buffered within the host-controller interface and issued to the respective virtual controllers based on a scheduling algorithm.

scheme, in which requests from each buffer are served during a designated time window. The window size is configured by the hypervisor. A window configuration, that ensures isolation, minimizes context-switches, and guarantees low latencies, is described in [10].

C. Security Extensions

As mentioned earlier, the virtualized CAN controller is shared between several VMs which might have different trust levels and criticalities. So in order to guarantee the functionality of critical functions security features are added to the virtualized CAN controller.
In order to detect and prevent denial of service (DoS) attacks a Bus Guardian (BG) is integrated in the virtualized CAN controller's data path. It uses Token Bucket based timing measurements to detect possible DoS attacks. Thus attacks which originate from one of the VMs can be prevented by blocking frames from VMs that violate their minimum cycle times. On the other hand the BG can also detect attacks originating at some other CAN node and report the detection to the hypervisor.
To prevent VMs from malicious injection of CAN messages the messages are secured with a keyed-hash message authentication code (HMAC). With this HMAC a receiving node can detect whether the message originates from a trusted node and shall be processed. Furthermore the HMAC includes a value that guarantees the freshness of the message and thus prevents replay attacks. Finally an optional encryption of CAN frames can be used to prevent eavesdropping on the bus.
Hardware accelerators for the HMAC calculation and encryption are integrated into the data paths of the virtualized CAN controller.

VI. Conclusion

Performance requirements of centralized ECUs in modern automobiles demand for multicore based solutions. This centralization of functions also leads to a combination of very different safety and security requirements that need to be fulfilled. Using virtualization in addition is one promising way of achieving segregation in time and space of shared resources - both important fundamentals of a safe and secure system. In this paper we presented selected approaches for hardware virtualization support that provide mechanisms to ensure segregation in time and space as well as quality of service (QoS) levels in terms of exectution time, throughput and latency. Our future work includes the final integration of these approaches into the demonstrator vehicle and evaluation of the overall architecture.

Acknowledgment

This work was funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding IDs 01IS11035. The responsibility for the content remains with the authors.

References

[1] H.-U. Michel, D. Kaule, and M. Salfer, "Vision einer intelligenten vernetzung," in *Elektronik Automotive*, 2012.

[2] A. Herkersdorf, H.-U. Michel, H. Rauchfuss, and T. Wild, "Multicore enablement for automotive cyber physical systems," *it-Information Technology*, vol. 54, no. 6, pp. 280–287, 2012.

[3] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 129–142.

[4] XILINX, *VC709 Evaluation Board for the Virtex-7 FPGA*, 2012, user Guide.

[5] SR-IOV, *Single Root I/O Virtualization and Sharing Specification*, 1st ed., 2007.

[6] A. Menon, J. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 2005, pp. 13–23.

[7] XILINX, *Virtex-7 FPGA Gen3 Integrated Block for PCI Express*, 2012, product Guide.

[8] ARM, *AMBA AXI and ACE Protocol Specification*, 2012, specification.

[9] A. Richter, C. Herber, H. Rauchfuss, T. Wild, and A. Herkersdorf, "Performance isolation exposure in virtualized platforms with pci passthrough i/o sharing," in *International Conference on Architecture of Computing Systems (ARCS)*, 2014.

[10] C. Herber, A. Richter, H. Rauchfuss, and A. Herkersdorf, "Spatial and temporal isolation of virtual can controllers," in *Workshop on Virtualization for Real-Time Embedded Systems (VtRES 2013)*, 2013, pp. 7–13.