Spatial and Temporal Isolation of Virtual CAN Controllers

Christian Herber, Andre Richter, Holm Rauchfuss, Andreas Herkersdorf
Technische Universität München - Institute for Integrated Systems
Munich, Germany
{christian.herber, andre.richter, holm.rauchfuss, herkersdorf}@tum.de

ABSTRACT

Virtualization is a key technology to enable the use of multi-core processors in automotive embedded systems. For side-by-side execution of mixed-criticality applications that access shared communication infrastructures, a secure and safe virtualization of I/O devices is required, which features a complete spatial and temporal isolation of individual virtual interfaces. We extended existing approaches of hardware-based CAN virtualization to achieve a full isolation while maintaining the bounded latencies achieved in previous implementations. It is shown, that even a denial-of-service attack towards one virtual controller does not influence the behavior of other virtual controllers. In addition, the scheduling mechanism implemented to guarantee temporal isolation can be configured to provide differentiated service levels for real-time and best effort application domains.

1. INTRODUCTION

The automotive IT landscape is a heterogeneous and historically grown network of electronic control units (ECUs) that are connected via a variety of fieldbuses. Common practice has been the introduction of an additional ECU for every new electronic function, leading to more than 100 ECUs in premium cars. Additionally, new functions like pedestrian detection, traffic sign recognition or parking assist systems are increasingly complex and require more computing power.

ECUs are grouped in functional domains like body, powertrain, chassis and infotainment. The nodes within these domains are connected through fieldbuses like Controller Area Network (CAN), FlexRay and Media Oriented Systems Transport (MOST). The most used bus is CAN, which connects nodes within the body, powertrain and infotainment domain. It is a message based broadcast bus that uses bitwise arbitration as access scheme and reaches a bandwidth of up to 1 Mbit/s.

Automotive OEMs are planning to redesign this grown architecture to a domain controlled architecture [15]. A domain controller is a centralized processing unit, which consolidates a variety of functions that have previously been partitioned onto a number of distributed ECUs. Decentralized ECUs remain in control of sensors and actuators. A centralization of functions increases the demand for computing power in single devices.

VtRES'13 Taipei, Taiwan

The introduction of multi-core processors is a promising trend in automotive electronics that could satisfy the computing needs of future automotive embedded systems [10], while also providing improved energy efficiency. In addition, the intrinsic parallelism of multi-core processors can enable the consolidation of multiple electronic functions onto one chip.

However, a parallel execution of functions on a multi-core processor introduces safety and security risks. Specifically, mixed-criticality scenarios that integrate functions with different trust levels and real-time requirements are of interest. Virtualization is a validated approach to ensure the isolation of such mixed-criticality applications. It can enable the side-by-side execution of e.g. best effort and safety critical real-time applications [6], [14] on a shared processor. Virtualization provides abstract, isolated computing resources in the form of virtual machines (VMs). In an automotive head-unit, this could e.g. allow a safety critical real-time AUTOSAR partition to be integrated side-by-side with a multimedia partition and an android partition, which runs untrusted third party applications.

Concurrent access by VMs towards shared I/O devices is one of the major challenges in virtualization. Hardware and software-based methods for I/O Virtualization (IOV) exist for Ethernet. To be used in automotive embedded multi-core systems, application specific needs have to be addressed. In contrast to traditional virtualized systems, which mostly require best effort communication over Ethernet, hard real-time requirements have to be met in an automotive environment. While for few, trusted VMs fieldbus access could be provided through multiple stand-alone controllers, a system with an increasing number of cores requires an I/O virtualization approach to be resource efficient and scalable.

In this paper, we introduce such an I/O virtualization approach for CAN controllers, which enables robust real-time communication for multiple VMs through a shared hardware device. The performance of each virtual instance is guaranteed by a strict spatial and temporal isolation, making sure that no additional safety or security risks are introduced.

The remainder of this paper is structured as follows: Section 2 presents related work regarding I/O virtualization. Section 3 gives an overview of the architecture of the self-virtualized CAN controller. Additional concepts to ensure the spatial and temporal isolation of virtual CAN controllers are discussed in Sections 4 and 5 respectively. Section 6 describes the simulation used and the results obtained, while Section 7 concludes the paper.

2. RELATED WORK

In a virtualized environment, the virtual machine monitor (VMM) provides computing resources to virtual machines (VMs), which are abstracted and isolated instances of the actual machine. Conflicts can arise when multiple VMs require access to a shared I/O device. Solutions for this problem of I/O virtualization (IOV) have been researched for Ethernet, and are presented and discussed in the following.

State-of-the-Art with respect to software-based IOV methods is the paravirtualization approach [11]. VMs communicate through a front-end/back-end driver structure with the VMM or a dedicated, trusted driver domain. The VMM or driver domain owns the actual device driver and forwards requests and data between the physical device and the VMs.

While software-based IOV solutions require no specialized hardware, they suffer from several shortcomings. The forwarding of requests and data done by the VMM can lead to CPU overheads that increase the CPU utilization to 100% and therefore limit the achievable bandwidth [8]. This overhead is not only a function of the served bandwidth, but also increasing with smaller request sizes. For small requests of 1 kB, the achievable net I/O rate decreases to 1 Mbit/s [2][12]. Additionally, paravirtualization introduces packet delays that can reach 100 ms [16].

These performance decreases can be overcome by offloading the virtualization mechanisms into hardware close the the actual I/O device [13]. Today, commercial of-the-shelf solutions, which support Single Root I/O Virtualization (SR-IOV), are available for Ethernet network interface controllers (NICs). SR-IOV allows PCIe devices to communicate directly with a VM via DMA and therefore bypasses the VMM. Using SR-IOV, 98.24 % of line rate performance could be achieved for a 10 Gbit/s NIC shared by 60 VMs [4].

Little research has been conducted regarding the virtualization of automotive I/O devices like CAN controllers. A software-based approach for CAN controller virtualization for integrated modular electronics has been presented in [7]. However, the paper does not present CPU overheads and measures message latencies only on an idle system. In [5], a hardware-based solution is presented that offloads virtualization tasks like a message based arbitration between virtual CAN controllers and VM based acceptance filtering for received frames into an existing CAN controller. It was shown that the additional latency suffered from sharing modules in the virtualized controller is within the order of 10 µs.

While CAN has a comparably small maximum bandwidth (1 Mbit/s), the small frame sizes (min. 47 bit) can lead to frame rates of more than 20,000 frames per second. Results obtained for Ethernet suggest that a paravirtualization at such request rates will introduce CPU overheads and additional latencies, which would not allow real-time capable communication.

We introduce mechanisms for spatial and temporal isolation of self-virtualized CAN controllers. In Ethernet NICs, it is sufficient to guarantee bandwidth shares for virtual NICs. For a real-time capable I/O controller, the performance of virtual controllers must be isolated from each other to ensure that no additional security issues are introduced compared to setups with distinct physical CAN controllers.

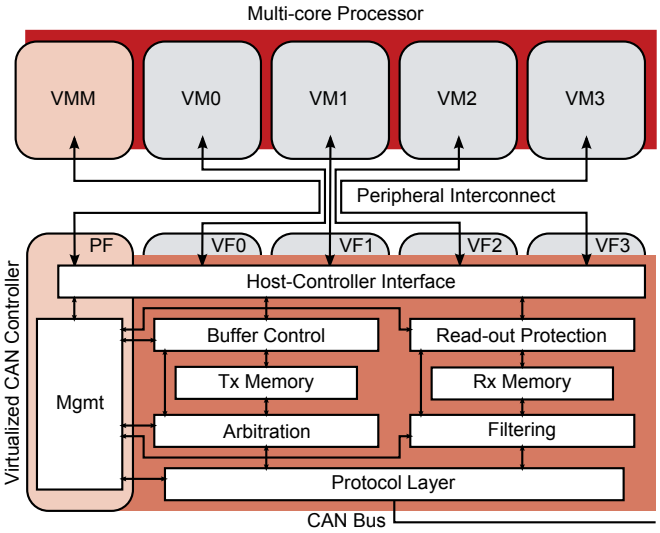


Figure 1: Architectural overview of a multi-core processor connected to the virtualized CAN controller. VMs can acces the CAN bus through virtual functions (VFs) that are managed by the VMM through the physical function (PF).

3. CONCEPT FOR A SELF-VIRTUALIZED CAN CONTROLLER

In this Section, we present an overview of the self-virtualized CAN controller, which is based on [5]. In the subsequent Sections 4 and 5, architectural aspects are introduced regarding the contributions for spatial and temporal isolation among virtual instances of the controller.

The architecture and concepts presented here are not limited to a certain processor type or interconnect. However, it is assumed that the processor supports virtualization and that (virtual) I/O devices can be directly assigned to a VM.

The goal of our work is to provide real-time capable CAN bus access for a number of VMs, enabled by a resource efficient and well scaling architecture. Therefore the controller should provide a number of virtual functions (VFs) or virtual controllers, which 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.

Therefore a privileged interface is necessary, which we call physical function (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 VMM or a privileged VM as depicted in Fig. 1.

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 al-

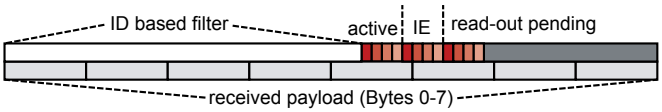


Figure 2: Rx message object memory layout: Message objects used for reception are composed of an ID based acceptance filter, several flags belonging to respective virtual controllers and the payload of the message received last.

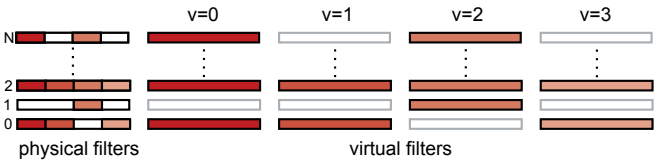


Figure 3: Virtual filters as seen by the respective VMs. Colored virtual filters can be accessed by the associated VM.

located 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 results in priority inversions, causing increases in worst-case latencies that make real-time operation impossible.

Received messages (Rx) are sorted towards virtual controllers based on a predefined set of filters. Each message will be stored only once, even if accepted by multiple virtual controllers. The read-out of messages can be done based on interrupts or by polling.

4. SPATIAL ISOLATION

Spatial isolation is important in the context of virtualization to ensure that VMs only have access to hardware resources (e.g. registers and RAM), which were assigned to them. If this isolation is missing or incomplete, VMs might be able to manipulate or read data belonging to other VMs.

We assume that the virtual controllers have a unique address space at system-level, which is ensured by a memory protection unit (MPU) or memory management unit (MMU). This allows unique association of requests with VMs or virtual controllers within the virtualized controller. Remaining challenges will be discussed for the Tx and Rx memory respectively.

Within the Tx memory, two kinds of data are stored: 1. The values of the registers (called context in the sequel) within the buffer control state machines for each virtual controller and 2. the buffered frames. When serving a request for a certain virtual controller, its context will be loaded first if necessary. Because virtual controllers do not have direct RAM access, the context of other virtual instances cannot be seen.

The distribution of Tx memory resources is crucial with respect to the real-time capability of a virtual controller. While from a memory utilization perspective, it would be desirable to share the memory among all virtual controllers dynamically, a minimum amount of memory is necessary to ensure real-time capability. If not all messages that are ready for transmission in a worst-case scenario can be buffered, priority inversions might arise. Therefore, the PF will setup virtual controllers with a fixed amount of memory allocated. For best effort controllers, it is sufficient to provide memory for a single frame.

The challenge in the Rx memory is to provide a resource efficient, but secure way of sharing message objects among multiple virtual controllers. The Rx memory consists of a list of message objects, which contain a unique ID based filter (see Fig. 2). Stored alongside are a number of flags for each virtual controller, which indicate whether the associated message can be accessed by a certain VM ('active'), whether interrupts are enabled (IE) and whether a new message has arrived since the last read-out ('read-out pending'). Based on this list of filters, the filtering module accepts and stores received messages.

This scheme improves the systems scalability, because only one consolidated list has to be checked when receiving a frame, and neither filters or data are redundant within the local memory. The memory layout as shown in Fig. 2 limits the number of virtual controllers to 9, but can be extended by e.g. using an additional RAM line to support 32 virtual controllers.

By making use of the 'active' flag the read-out protection module can provide a separation between virtual controllers. No look-up tables are required to decide if a read operation is allowed, which further improves resource efficiency and scalability. The memory map seen within a VM contains all possible Rx message objects, but reads towards messages not configured for their virtual controller will fail. With the configuration shown in Fig. 3, messages accepted e.g. by filter 1 can only be read-out through virtual controller $v = 2$.

5. TEMPORAL ISOLATION

The virtual CAN controllers share common hardware modules to enable a resource efficient architecture. This concept introduces safety and security issues, because it could enable a corrupted VM to influence the temporal behavior of virtual CAN controllers attached to other VMs.

In this section, we present mechanisms intended to resolve temporal conflicts between different VMs trying to access their respective virtual CAN controller at the same time. It is assumed that a fair temporal distribution of resources is present in other involved components like the peripheral interconnect.

5.1 Motivation

Denial-of-service (DoS) attacks are a prime example for attacks capable of exploiting such vulnerabilities. In a DoS scenario, a VM would be sending requests to its own virtual CAN controller at a rate much higher than intended, thus potentially decreasing the performance of other virtual controllers, and eventually making real-time communication impossible.

The security and safety measures introduced here aim at resolving such issues, which are introduced due to I/O virtualization. Threats present in a setup with multiple physical, directly assigned I/O devices are not within the scope of this paper. This means that attacks aimed at other hardware entities like the CAN bus will not be prevented.

In this context, it is important to note that the source of an unexpected behavior cannot be determined within the virtualized CAN controller. A functional failure and a secu-

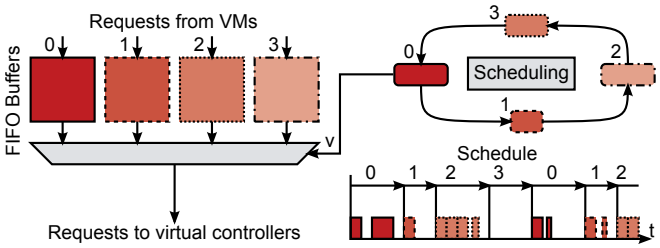


Figure 4: Scheduling of virtual interfaces: Requests from VMs are buffered within the host-controller interface and issued to the respective virtual controllers based on a scheduling algorithm.

rity attack can show the same symptoms when arriving at the CAN controller. Therefore, both cases can be treated using the same mechanisms.

5.2 Virtual I/O Interface Scheduling

All services realized within each virtual CAN controller are provided to their respective VM via a virtual interface implemented in the host-controller interface. While these interfaces appear to be independent, they use common hardware resources like FSMs so that only one interface can be active at a time. Here, we introduce mechanisms to achieve a temporal separation of the interface access. They enable highly predictable access to virtual interfaces while minimizing additional latencies in a CAN specific worst-case scenario.

Requests from the VMs may arrive at the host-controller interface at peak rates that cannot be served immediately. Therefore requests have to buffered in FIFOs. To enable a separation of virtual interfaces, a distinct buffer is provided for each virtual controller v. A scheduling algorithm has to ensure that all buffers are served as shown in Fig. 4.

To select a scheduling mechanism, we determine a number of criteria that should be used to evaluate its feasibility. Ideally, a scheduling mechanism for virtual interfaces satisfies the following properties:

1. Predictable: The scheduling mechanism is assumed to be deterministic. Its behavior should be easy to predict to allow a real-time analysis to be developed.

2. Interface utilization: The resources of the virtualized controller should be idle for as little time as possible and context switches should be minimized.

3. HW Overhead: A feasible solution should be achieved in a resource efficient way.

4. Added latencies: The scheduling algorithm should add minimal latencies to the response time of CAN messages during a worst-case scenario.

5.3 Exploration of Scheduling Mechanisms

Round-robin (RR) scheduling intrinsically satisfies many of the requirements stated above. Its simplicity allows it to be easily predictable and implemented at low hardware cost. A number of RR variations are used by the PCIe standards for virtual channel and port arbitration [1]. To optimize the scheduling mechanism for the virtualized CAN controller the following variations of RR scheduling will be considered:

- Simple RR: Requests are issued in turns from different buffers. This scheme is repeated in a cyclic manner. If a buffer holds no requests, it is skipped.

- Weighted RR (WRR): During one cycle each buffer is capable of issuing a configurable amount of requests. It allows accounting for the different communication needs of different VMs.

- Time-based RR (TBRR): Time slots of configurable length are assigned to each virtual interface. The interfaces are iterated cyclic. If no request is available during a time slot, the interface still does not lose its slot.

- Weighted time-based RR (WTBRR): The length of the time slot is configurable for each interface. The scheduling depicted in Fig. 4 represents a WTBRR scheme.

Weighted variations are useful when applications have diverse communication needs. Fitting the weights to the actual bandwidth requirements in a WRR scheme allows to increase the utilization of the interfaces, because it enables the virtualized CAN controller to conclude a couple of requests from the same VM in succession without the need of a context switch.

For a time-based version, using different weights is beneficial, because interface bandwidth reserved for one VM cannot be used by another one, even if it is not in use. Reducing the time windows of interfaces with low utilization can increase the actual utilization in a burst scenario and decrease the time that other interfaces have to wait to issue their requests.

The downside of adding a weight to RR scheduling is an increased complexity regarding the HW implementation, as an additional configurability of the weights by the PF is needed. Otherwise, weighted RR versions are assumed to be superior or equal in all other cases, because they can be configured to show the same behavior as their non-weighted counterparts.

Time-based versions have the advantage of being highly predictable. At any time, it can be determined when the next turn for a specific virtual controller starts or ends. This is true independent of its own behavior and the behavior of other VMs. It therefore leads to a strong isolation, through which timing properties hold true even e.g. under DoS attacks.

When not using time-based schemes, the scheduling is harder to predict, because the actual utilization of the virtual interfaces influences the time windows in which requests are served by a specific virtual CAN controller. Because of the inferior predictability of these schemes, it is more difficult to determine the added latencies suffered here.

We determine a WTBRR scheduling scheme to deliver the best trade-off with regards to the criteria introduced above. Its high predictability allows for good isolation. The option to configure time window lengths increases the complexity, but also enables the scheme to be fitted to the actual application requirements in order to optimize the added latency and interface utilization. An appropriate window size for individual virtual controllers has to be determined by minimizing the added latencies in a worst-case scenario.

5.4 Added Latencies for Weighted Time-based Round-Robin

Because the physical interface to the virtualized CAN controller is shared by multiple virtual controllers and the requests take a finite time to conclude, an added latency is experienced compared to the case, where a physical interface is exclusively reserved for one application.

In [3], an analytic real-time analysis for ideal CAN nodes is presented. In the worst-case scenario for a certain message, it is assumed that all higher priority messages are ready for transmission at the same time and a lower priority message has just started transmitting.

This analysis was extended in [5] to be applied to non-ideal, virtualized CAN controllers. The worst-case scenario was modified, because here, low priority messages can also contribute to an additional blocking of high priority messages. The additional blocking introduced due to virtualization is assumed to be

$$B_{virt,m} = \sum_{v=0}^{V-1} \sum_{k=0}^{M_{lp(m),v}-1} t_{insrt}(k), \tag{1}$$

where V is the number of virtual controllers, $M_{lp(m),v}$ is the number of lower priority messages in virtual controller v and $t_{insrt}(k)$ is the worst-case time it takes to insert a message into a queue that already contains k messages. Here, it is implicitly assumed that a context switch happens after each insertion and the time for it is included in the insertion time.

Because context switches can be reduced by choosing a feasible configuration, these times will be considered explicitly in the following analysis. The insertion time remains $t_{insrt}(k)$ and the time for a context switch t_{switch} is introduced. Based on the actual architecture, these times can be determined as

$$t_{insrt}(k) = (4 + k) \cdot T_{clk} \tag{2}$$

$$t_{switch}(k) = 2 \cdot T_{clk} \tag{3}$$

To minimize unnecessary context switches, each VM should have a window long enough to conclude all requests during a worst-case scenario. It is assumed that all messages from all VMs are to be inserted nearly at the same time. In this case, the minimal number of context switches can be achieved, when all requests from VM can be served during a single time window in close succession. This leads to V context switches during a worst-case scenario. The window size of virtual controller v containing M_v messages would be configured as

$$t_{window,v} = t_{switch} + \sum_{k=0}^{M_v-1} t_{insrt}(k). \tag{4}$$

On the other hand, such relatively long windows increase the time interval during which requests from other VMs cannot be served. However, in a worst-case scenario for a message m, this message is assumed to be inserted last into its respective virtual controller and that the corresponding time window has just passed. In this scenario, shorter time windows do not reduce the latency experienced in a worst-case scenario.

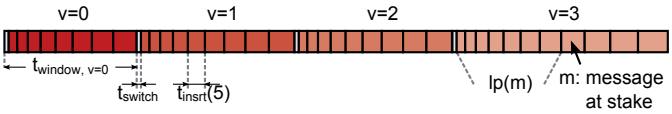


Figure 5: Added latencies experienced by message m in virtual controller v=3: It is comprised of blocking by time windows assigned to other virtual controllers and lower priority messages issued to the same virtual controller.

We therefore propose a configuration of time windows that uses a window length as indicated in (4). Under these circumstances, the additional blocking for message m in virtual controller v is

$$B_{virt,m} = \sum_{w \in \mathcal{V} \setminus v} t_{window,v} + t_{switch} + \sum_{k=0}^{M_{lp(m),v}-1} t_{insrt}(k) \tag{5}$$

where $\mathcal{V} = \{v : v \in \mathbb{N}; 0 \leq v < V\}$ describes the set of virtual controllers. This equation consists of the window lengths of the other virtual controllers, a context switch time and the blocking, that is contributed from lower priority messages within the own virtual controller. Higher priority messages do not contribute to an additional blocking, because they will overtake the message independent of where they are inserted.

Assuming that the window sizes are determined at design-time based on (4), then (5) does not depend on the traffic pattern actually issued to other virtual controllers. This implies that a DoS attack would not influence the blocking experienced at the interface of a virtual controller. In this case, a temporal isolation of virtual controllers is guaranteed.

The components adding up to the overall blocking introduced by virtualization are visualized in Fig. 5. The results can serve as input to a complete real-time analysis as shown in [5]. When (1) is replaced with (5), the existing analysis is still applicable.

In contrast to [5], no information is needed about lower priority messages outside of the virtual controller v at stake (isolated temporal behavior). Higher priority messages from other (virtual) CAN nodes influence the worst-case response time of message m, because they win arbitration on the CAN bus, but not because of effects introduced due to virtualization.

The configuration of scheduling windows presented above is intended to minimize the worst-case latencies experienced by messages with hard real-time requirements. However, increasing a window for one virtual controller causes other virtual controllers to experience a decreased performance at their respective interface.

In mixed-criticality scenarios, different applications accessing a common communication infrastructure might have different requirements regarding quality of service (QoS). Therefore, a configuration of temporal resources that allows real-time capability might not be needed for every virtual controller. For best effort interfaces, it is sufficient to provide small time windows capable of serving the longest possible request. This guarantees minimal temporal effect on the additional latencies experienced by real-time interfaces.

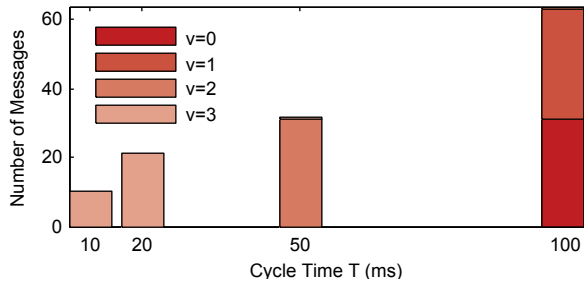


Figure 6: Distribution of messages with respect to their cycle time and their partitioning into a virtual CAN controller v

6. EXPERIMENTS AND RESULTS

To verify the results presented in the previous sections, we simulated a virtualized CAN controller under a DoS attack. First, the approach, scenario and measurements used are introduced. The results of this simulation are presented afterwards.

6.1 Simulation

We implemented a virtualized CAN controller as presented in the previous Sections in Verilog. It is embedded in a ModelSim SystemC/Verilog co-simulation, which is used to generate requests like message insertions towards the virtualized controller. The controller features full spatial, and temporal isolation through WTBRR scheduling as an optional feature. An operating frequency of 100 Mhz is assumed.

We further assume a bandwidth of 500 kbit/s on the CAN bus, which is the fastest version that is currently employed in cars. We generated a traffic pattern based on a typical cycle time distribution in a modern premium car [9]. Under the assumption of a bus utilization of 90% and a constant payload size of 8 bytes, 127 messages are generated as shown in Fig. 6. It is assumed that messages with lower cycle times have higher priority and that their deadlines are equal to their cycle time.

In our test case, these messages are sent by four different VMs, each assigned to a virtual controller within the same virtualized CAN controller, which corresponds to the architecture shown in Fig. 1. The messages are distributed amongst the VMs in equal shares. The priority of messages is increasing from VM0 to VM3. The messages send through virtual controller 3 therefore are the most critical with respect to real-time capability, because they have the shortest cycle times and deadlines. The window sizes associated with the WTBRR scheme are chosen as described in (4) and therefore guarantee real-time capability. The window size for each virtual controller is around 6 µs, which makes the additional blocking due to virtualization around 20 µs according to (5). This time is equivalent to the transmission time of 10 bit on the CAN bus and three orders of magnitude lower than the deadlines in our scenario.

In order to investigate the effect of the architecture and different scenarios on the timely transmission of messages, the maximum response time R_m of each message m will be measured. It is defined as the time from the initiation of the request until the successful transmission of the message on the CAN bus [3].

In a real-time environment, the worst-case delay is the most important figure. The simulation tries to match this case as closely as possible. All messages are assumed to be scheduled at the beginning of the simulation, with VMs first requesting the transmission of their low priority messages and the requests from VMs with low priority messages arriving first.

Additionally, the simulation allows to spam arbitrary requests (e.g. write towards a register) from VM0 to its respective virtual controller. In our model, these requests can be completed within four clock cycles. This scenario corresponds to a DoS attack and can be modified by a factor a_{DoS}. For every message issued by VM0, a_{DoS} requests will be issued additionally. These requests are issued just before the message insertions.

6.2 Results

The results are shown in Fig. 7, which illustrates, how the maximum response time of messages changes when the virtualized CAN controller is subject to a DoS attack.

When no temporal isolation is implemented, requests are forwarded within the host-controller interface with a first-come-first-served (FCFS) policy. This means that the requests issued by the DoS attack block the insertion requests. Fig. 7a shows that in this case, a general increase in response time for all messages can be witnessed, which is proportional to the amount of requests issued by VM0.

Despite being forwarded using a FCFS scheme, high priority messages still overtake lower priority messages inside the message buffers of the virtual CAN controllers, resulting in an increase in response time with decreasing message priority.

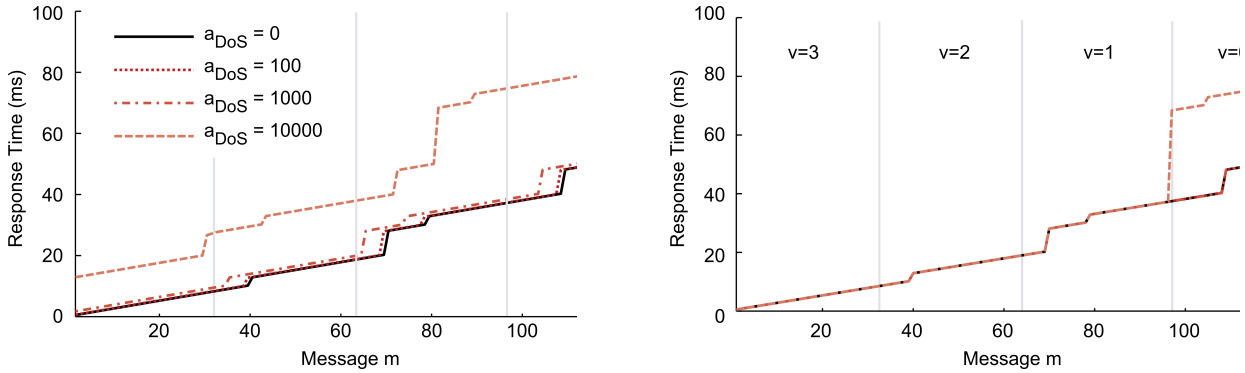
We observe that the response times are increasing with respect to a_{DoS}. While this increase might be acceptable for $a_{DoS} = 100$, the deadlines of all high priority messages with cycle times of 10 ms are violated for $a_{DoS} = 10000$. Real-time applications in VM3 would fail to fulfill their task due to the corruption of VM0.

The increase in transmission time is grounded in the additional blocking suffered at the host-controller interface. At the beginning of the simulation, the requests issued by the DoS attack have to be served first. Afterwards, messages from all VMs are being inserted and will be transmitted on the bus. Because the whole transmission process is delayed, lower priority messages can suffer additional blocking due to second or further instances of high priority messages that get reissued. Therefore, medium and low priority messages can experience a higher increase in response time than high priority messages.

The effect of temporal isolation in such a scenario is demonstrated in Fig. 7b. Here, requests towards the virtualized CAN controllers are stored in V queues. These queues are then scheduled in in a WTBRR scheme.

Generally, no increase in response time can be seen for messages outside of VM0. We therefore conclude that a complete isolation of virtual controllers has been achieved and applications in trusted VMs are guaranteed to experience a consistent level of QoS.

On the other hand, the response time of messages issued by VM0 only increases when the blocking caused by the DoS attack is within the order of their response time. Otherwise, the insertion is delayed, but no change can be seen on the bus, because these messages would have to wait for transmission anyway.



(a) No temporal isolation

(b) Temporal isolation through WTBRR

Figure 7: Simulation results obtained during a DoS attack towards virtual CAN controller v=0: The graphs show the response time for all messages in the network, starting with the highest priority message.

7. CONCLUSION

In this paper, we presented concepts for a self-virtualized CAN controller and extensions that guarantee a spatial and temporal isolation of virtual controllers. Our contributions allow virtual machines to access the CAN bus concurrently through shared hardware resources without additional security issues, while not suffering from the increased latencies and CPU loads that come along with a paravirtualization.

Spatial isolation is ensured through context switching and memory protection mechanisms. Registers and RAM can only be accessed if they have been assigned to a specific virtual controller. The temporal isolation is accomplished by a weighted time-based round robin scheduling of requests. The scheduling was optimized to introduce minimal blocking at the interface (∼20 µs, less than the transmission of 10 bit on the bus) during worst-case scenarios. Additionally, this scheme decouples the temporal behavior of one virtual controller from the actual requests issued towards other virtual CAN controllers.

We demonstrated the robustness of the method by applying a denial-of-service attack scenario to a virtualized CAN controller with and without temporal isolation. Here, one VM is assumed to be corrupted and issues high amounts of requests to its virtual CAN controller. It is shown that the isolation mechanisms ensure that the virtual controllers attached to non-corrupted VMs are not influenced, allowing a secure execution of mixed-criticality applications that access a common virtualized CAN controller.

Acknowledgments

This work was funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding IDs 01|S11035. The responsibility for the content remains with the authors.

8. REFERENCES

[1] R. Budruk, D. Anderson, and T. Shanley. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[2] L. Cherkasova and R. Gardner. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *Proceedings of the USENIX annual technical conference*, pages 387–390, 2005.

[3] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[4] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 2012.

[5] C. Herber, A. Richter, H. Rauchfuss, and A. Herkersdorf. Self-virtualized can controller for multi-core processors in real-time applications. In *International Conference on Architecture of Computing Systems (ARCS)*, pages 244–255, 2013.

[6] A. Herkersdorf, H.-U. Michel, H. Rauchfuss, and T. Wild. Multicore enablement for automotive cyber physical systems. *it-Information Technology*, 54(6):280–287, 2012.

[7] J. Kim, S. Lee, and H. Jin. Fieldbus virtualization for integrated modular avionics. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4. IEEE, 2011.

[8] 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*, pages 13–23. ACM, 2005.

[9] B. Müller-Rathgeber, M. Eichhorn, and H.-U. Michel. A unified car-it communication-architecture: Design guidelines and prototypical implementation. In *Intelligent Vehicles Symposium, 2008 IEEE*, pages 709–714. IEEE, 2008.

[10] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion. Multi-source and multicore automotive ecus-os protection mechanisms and scheduling. In *International Symposium on Industrial Electronics-ISIE 2010*, 2010.

[11] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the art of virtualization. In *Linux Symposium*, pages 65–77, 2005.

[12] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and

C. Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 51–58. IEEE, 2010.

[13] H. Raj and K. Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *High Performance Distributed Computing: Proceedings of the 16 th international symposium on High performance distributed computing*, volume 25, pages 179–188, 2007.

[14] D. Reinhardt, D. Kaule, and M. Kucera. Achieving a scalable e/e-architecture using autosar and virtualization. In *SAE World Congress*, 2013.

[15] D. Reinhardt and M. Kucera. Domain controlled architecture: A new approach for large scale software integrated automotive systems. In *Pervasive and Embedded Computing and Communication Systems*, 2013.

[16] J. Whiteaker, F. Schneider, and R. Teixeira. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1):38–44, 2011.