HW/SW Trade-offs in I/O Virtualization for Controller Area Network

Christian Herber[1], Dominik Reinhardt[2], Andre Richter[1], and Andreas Herkersdorf[1]

[1]Insitute for Integrated Systems, Technische Universität München, Munich, {firstname.lastname@tum.de}

[2]BMW AG, Munich, dominik.reinhardt@bmw.de

Invited

ABSTRACT

Automotive embedded systems are highly complex and historically grown networks of single-core based control units. Due to space limitations and wiring complexity, the scalability of current architectures is limited. It can be overcome by consolidating multiple currently distributed functions onto shared multi-core platforms. Additionally, virtualization can be used to isolate these functions in separate virtual machines (VMs). However, access to peripherals like Controller Area Network (CAN) communication interfaces must be shared among these partitions, a task that is usually associated with high overheads.

In this paper, we present and quantitatively compare two approaches to enable sharing of CAN controllers among VMs. First, we use a software-based paravirtualization, in which I/O requests are moderated by a privileged software component. Second, we offload I/O virtualization tasks into the CAN controller itself, thus providing direct I/O access for VMs.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms

Design, Measurement, Performance

Keywords

Controller Area Network, Virtualization, Automotive Electronics

1. INTRODUCTION

Automotive embedded systems consist of up to 100 distributed computing nodes. They are referred to as electronic control units (ECUs) and traditionally realize one electronic function within the vehicle. This approach leads to a linear scaling of ECUs with the number of realized functions and thus, has limited scalability.

Multi-core processors are a promising technology to overcome the scalability issues of today's automotive architectures [13]. Compared to current single-core ECUs, they offer significantly increased computational density. Multi-core based ECUs are suited to reduce the total number of nodes by integrating multiple previously distributed functions on a shared platform. Therefore, the total number of nodes can be reduced and scalability for future architectures can be enabled [2].

Multi-core processors feature physically isolated computation cores, but share major parts of on-chip resources like caches, interconnects, co-processors, and network-I/O interfaces. To guarantee safe operation of functions running on a shared platform, isolation and cooperation among the partitions is required [1].

Virtualization is an established technology in server environments, which enables safe and secure sharing of a common physical platform among multiple partitions. Virtual machines (VMs) represent an isolated and abstracted version of the underlying hardware. Within VMs, operating systems can be executed in a similar fashion as on the non-virtualized system. Virtualization allows concurrent execution of multiple (different) operating systems, while the functional integrity the partitions is decoupled.

In the context of automotive embedded systems, virtualization can be used to safely consolidate previously distributed functions [12, 15]. Its intrinsic isolation and the ability to run multiple legacy functions concurrently makes it a suitable candidate to solve current multi-core issues.

A major challenge in virtualized systems is sharing of network-I/O among multiple VMs. To avoid collisions, VMs must not directly access shared hardware components. Therefore, a privileged software component, called hypervisor, (de-)multiplexes and forwards I/O requests between VMs and the respective hardware component.

Because Virtualization is usually used in server environments, most I/O virtualization solutions are focused at Ethernet, where the main optimization target is throughput maximization. In contrast, automotive systems are also time-sensitive, meaning low-latency operation is equally important.

We introduce two I/O virtualization approaches for CAN, the most widely used automotive networking technology.

DAC' 15 June 07 - 11, 2015, San Francisco, CA, USA

ACM 978-1-4503-3520-1/15/06 ...$15.00
http://dx.doi.org/10.1145/2744769.2747929.

One approach uses SW-based paravirtualization and one uses HW-assisted virtualization with direct device access. We prototyped the solutions using an Infineon AURIX TriCore TC27x and an Intel Core i-73770T, respectively.

The paper is structured as follows: Section 2 introduces background regarding I/O virtualization. Following, the concepts based on paravirtualization (Section 3) and HW-assisted virtualization (Section 4) are presented. In Section 5, we conduct two experiments to asses and compare latencies experienced in either solution.

2. RELATED WORK

I/O virtualization is one of the major challenges in virtualization and has been an active research area for over a decade. Usually, research focuses on Ethernet network interface controllers (NICs). Main design goal is to maximize throughput while providing secure and isolated access towards the NIC for a scalable number of VMs.

Originally, I/O access was provided to VMs using device driver **emulation** [16]. Here, VMs use an unmodified device driver. Privileged instructions are moderated by the hypervisor using trap-and-emulate. Because traps are costly in terms of execution time, this method introduces significant overhead.

To reduce the overhead experienced in emulation, **paravirtualization** was introduced for network-I/O [8]. The device driver is only placed within the hypervisor, or preferably, a dedicated driver VM. VMs can access the physical device by communicating with the driver domain using a front-end back-end driver model. VMs use a front-end driver, which provides basic data path functionalities. The driver domain receives these requests through a back-end driver and forwards them to the actual device driver. By using modified drivers within the VM, no traps are needed, thus reducing the overhead. Also, device drivers only need to be available to the operating system running in the driver domain. Placing device drivers within a dedicated driver VM is preferred, as it reduces safety risks within the hypervisor.

However, also paravirtualized I/O introduces overheads that lead to throughput degradation. Menon et al. [7] developed a profiling toolkit for XEN, with which they were able to demonstrate a throughput degradation of 20%, if driver domain and VM are executed on the different cores, and up to 66% if they share the same core. The overhead is mainly constituted by data copy operations and context switches between guest and host mode.

To reduce overheads associated with paravirtualization, Raj et al. introduced **self-virtualized I/O devices** [9]. They used a network processor board to offload major I/O virtualization tasks into hardware close to the Ethernet NIC. The self-virtualized NIC achieves approximately double the throughput of a baseline implementation using paravirtualization.

Currently state-of-the-art with respect to I/O virtualization is **Single Root I/O Virtualization** (SR-IOV) [3]. Using SR-IOV, the address space of an I/O device is divided into multiple virtual functions (VF) and one physical function (PF). VFs are directly assigned to VMs and offer data path operations similar to the function of a front-end driver. Privileged management options are only accessible through the PF, which is mapped into the hypervisors' address space. Therefore, data path operations are possible for VMs without hypervisor involvement, thus minimizing virtualization overheads. Dong et al. [3] demonstrated a throughput of 94.8% using a 10 Gbit/s Ethernet NIC with SR-IOV support.

Few work focusing on I/O virtualization has been conducted in the domain of embedded systems. In [10], a concept for a virtualized Ethernet NIC for embedded application is presented, which uses caching of context information to reduce latencies.

Kim et al. [6] proposed a paravirtualization based solution for sharing CAN controllers in Integrated Modular Avionics. Their experiments show added latencies of around 200 µs in an interference-free scenario compared to a native solution.

Reinhard et al. [11] and Herber et al. [4, 5] have presented paravirtualized and HW-assisted I/O sharing solutions for CAN in an automotive setting. These solutions form the basis of this paper. The work is extended by conducting a common experiment that allows a quantitative comparison of both approaches.

3. SW-BASED VIRTUALIZATION OF CAN CONTROLLERS

Automotive ECUs are small and cheap hardware devices, optimized for series production. There are additional requirements for energy awareness and the resistance for operation in rough environments like high differences in temperatures or strong vibrations. Furthermore, most current automotive ECUs lack a Memory Management Unit. This fact causes severe problems for the integration of state-of-the-art hypervisors. Software systems within vehicles are statically configured before build time. Memory protection can be achieved by using a statically configured Memory Protection Unit.

To counteract the problem of missing hardware features we use a type-1 hypervisor (supported by ETAS/Bosch), which can handle E/E platforms without a Memory Management Unit [14]. All VMs use the paravirtualized RTA-OS, which is supplied by ETAS, as well. The hypervisor is built for TriCore processor designs and is ported to the Infineon AURIX TriCore TC27X. All settings for the hypervisor and integrated VMs can be configured offline.

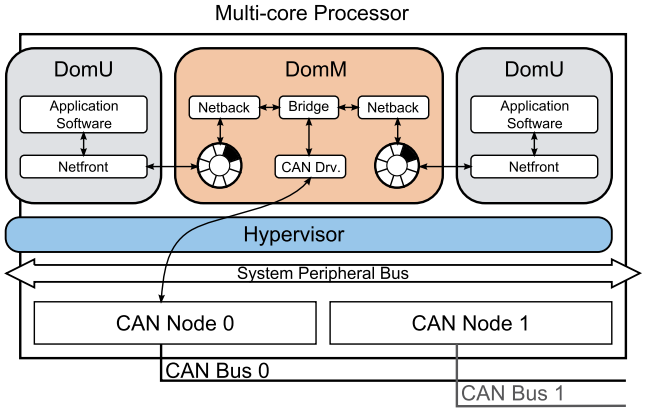


Figure 1: SW architecture of a paravirtualized CAN controller. Accesses from DomUs to CAN node 0 are moderated by the privileged DomM.

We use a one-to-one mapping of cores and VMs to avoid further timing overheads (scheduling of multiple VMs on a

single core). An independent hypervisor instance is assigned to every VM. To achieve a proper real-time behavior we use paravirtualization instead of full virtualization. The interaction between VMs is controlled and processed hypervisor-internally by using programmable interfaces, called Virtual Device Emulators. These Virtual Device Emulators can also be used to emulate hardware features like communication controllers.

The adaption of hardware drivers by using the Infineon AURIX TriCore can be realized in different ways. In case there are sufficient registers within the Memory Protection Unit available, it can be used to grant access to the registers of specific communication controllers. No changes within the already existing drivers' source code are necessary. If accesses to the controllers' space cannot be configured within the hardware, read or write attempts to the controller's space must be paravirtualized and wrapped with hypercalls. In our setup, hypercalls are realized through system calls, which are typically a type of a trap. To save registers of the Memory Protection Unit, the grant to memory space should be configured consecutively in bigger address ranges.

If hardware devices like CAN controllers should be paravirtualized, we identified three technical possibilities to integrate hardware drivers into the system: Either the AUTOSAR MCAL and its drivers are integrated within each VM (a) and run in user space, where concurrent accesses can be avoided by using a concerted configuration. Or all hardware drivers are integrated within the hypervisor itself (b) and run within the trusted computing base. A third possibility is to integrate all drivers in an independent virtual machine (c) and route all attempts for hardware access over that central virtual instance.

To be prepared for many core systems, where probably several virtual ECUs are integrated, we analyze the concept to consolidate hardware drivers in an independent virtual machine (c). This method is illustrated in Fig. 1. Here, the AUTOSAR Memory Abstraction layer is integrated within an privileged VM, which is assigned with special access rights [11]. This approach is similar to Xen's paravirtualization technique using a Dom0, which is responsible to encapsulate hardware drivers, controls the inter VM communication and strongly interacts with the hypervisor. Here, the dedicated driver domain is called DomM, while remaining VMs are called DomU. DomUs are not allowed to interact directly. Rather, all information must be routed over DomM, either to other VMs or to a connecting fieldbus. The overall communication flow between DomUs is routed over DomM by using a Layer-2 CAN bridge [11]. The information exchange between VMs is realized over shared ring buffers (Circular Buffer Addressing Mode of the AURIX TriCore).

4. HW-ASSISTED VIRTUALIZATION OF CAN CONTROLLERS

CAN is an interconnect technology used in real-time applications. Therefore, it benefits from predictability and low-latency associated with HW-assisted virtualization in combination with direct assignment of virtual functions (VFs) into VMs. In this section, we present an architectural concept and a prototypical implementation of a virtualized CAN controller using HW offloading.

The architecture of the virtualized CAN controller (see Fig. 2) is structured as follows: The controller has a memory mapped interface connected to the host system, which is divided into one physical function (PF) and multiple virtual functions (VFs). The physical function is assigned to the hypervisor and offers privileged management functions like controlling the bitrate setting of the bus or configuring VFs. Each VF is connected to a VM and offers access to a virtual CAN controller, which provides abstract CAN communication features.

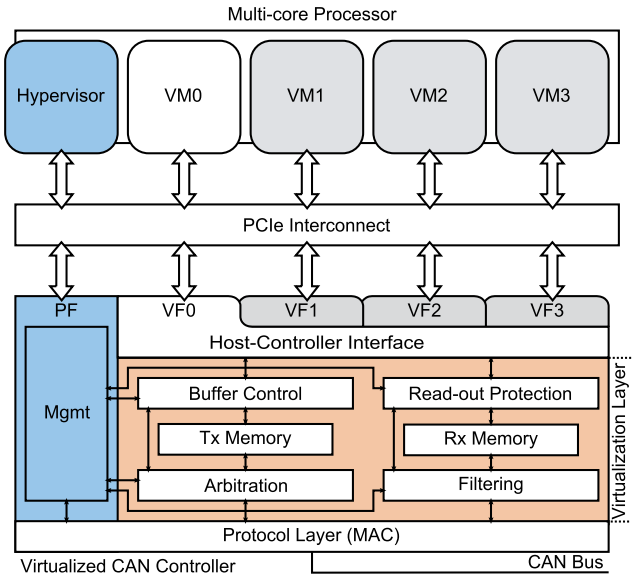


Figure 2: Architecture of the HW-virtualized CAN controller.

Vertically, the virtualized CAN controller is composed of the *Host-Controller Interface*, the *Virtualization Layer* and the *Protocol Layer*. The *Protocol Layer* offers basic CAN bitstream processing and protocol handling. We implemented it using an OpenCores CAN controller that is equivalent to an NXP SJA1000. It offers only little message buffering capabilities, which are extended in the *Virtualization Layer*.

The *Virtualization Layer* is a hardware extension that realizes multiplexing, demultiplexing, buffering, and forwarding of communication requests from and towards multiple VMs. These tasks are equivalent to the ones implemented in the hypervisor or a dedicated driver domain when using paravirtualization. Hardware components and registers within this layer are shared by all virtual CAN controllers to guarantee and efficient and scalable architecture. Every time a different virtual CAN controller is accessed, its associated register values have to be loaded from the memory. Such a context switch is a costly operation on CPUs, but can be done within few cycles in the hardware.

The *Virtualization Layer* maintains a logically separate transmit buffer for each virtual CAN controller in a shared memory. These buffers are realized as priority queues, in which messages are sorted with respect to their CAN priority. The *Arbitration* module selects the globally highest priority message just before a CAN bus arbitration starts. This mechanism ensures that arbitration among virtual CAN controllers is equivalent to the arbitration that would be possible using separate physical CAN controllers.

While CAN is a broadcast medium, applications do not require to receive all messages. Therefore, a mapping be-

tween virtual CAN controllers and desired messages is stored within the *Rx Memory*. New mappings can only be added by through the PF. Using this information, the *Filtering* module decides whether to store or discard received messages.

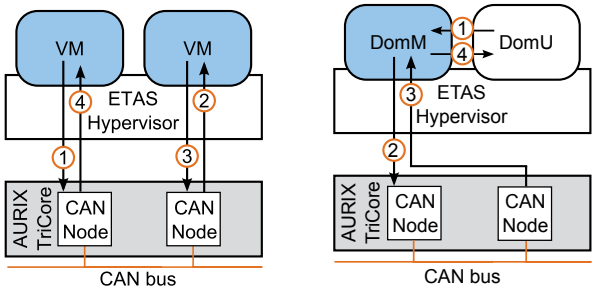
We prototypically implemented the presented architecture using a Xilinx Virtex-7 VC709 development board. While smaller boards would be sufficient to fit the design, this board is one of the few to feature an SR-IOV capable PCIe endpoint.

5. EXPERIMENTS & RESULTS

To compare the approaches introduced above, we conducted four experiments on two different platforms, which compare native and virtualized performance. Paravirtualization was evaluated using an automotive multi-core platform. Because such platforms lack dedicated hardware support for virtualization, HW-assisted I/O virtualization was evaluated using a general purpose Intel x86 platform.

5.1 Paravirtualization

SW-based virtualization allows to integrate different software systems on a common hardware platform, independent of the needed technologies or hardware features like several CAN communication controllers. However, the overheads to process additional software parts must be taken into account. The measurement setup is depicted in Fig. 3. Because CAN is an asynchronous bus, end-to-end timing measurements using multiple platforms are hard to carry out, as they would require an additional synchronization channel. Therefore, all experiments send and receive CAN messages on the same platform. The AURIX TriCore has multiple CAN nodes, which we connected using a shared CAN bus.



(a) Native (b) Virtualized

Figure 3: Assessment of latencies to measure round trip times of CAN messages using a paravirtualized CAN controller.

Fig. 3a represents the native case, where we use direct device assignment to assign an exclusive CAN node to each VM within the platform. The hypervisor acts as a monitoring function, and only intervenes in case of an exception. In case a VM transmits messages over the CAN bus, communication controllers are addressed directly (1). Upon receiving these messages (2), a second VM replies by looping this message back (3). This is similar to the ICMP echo request (*ping*). Messages are sent back to the second CAN node (3) and are routed by the hypervisor to the VM (4), where the message transfer started.

The measurement process includes two complete send and receive cycles. Thus, we determine the end-to-end delay by

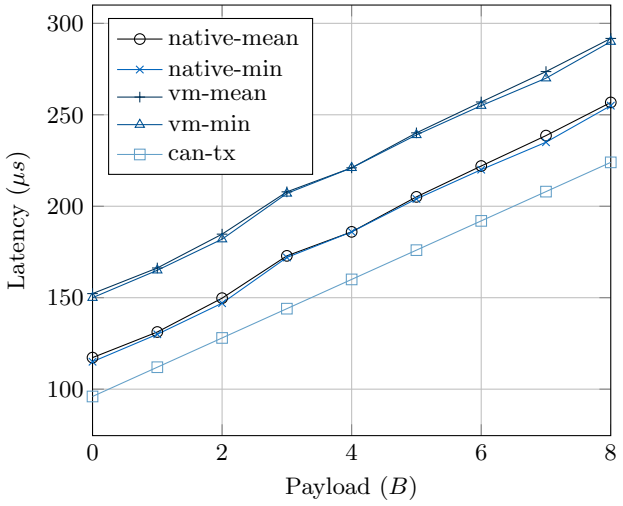


Figure 4: Latencies measured for a complete transmit-receive loop for communication between VMs and a connected CAN fieldbus on the AURIX TriCore.

dividing the overall latency by two. To minimize inaccuracies, we repeat our measurements 100 times. Each measurement is started only when the previous one has finished. Thus, measurements cannot interfere with one another.

Fig. 3b illustrates the measurement conducted using the DomM/DomU concept (paravirtualization). We create CAN message within DomU and request its transmission through DomM (1). Here, the CAN message is processed and routed to the connecting CAN node (2) by usage of an appropriate hardware driver [11]. Again, a second CAN node is connected over a common CAN bus. The transmitted CAN frame is received and forwarded by the hypervisor to DomM (3). Now, DomM routes the message to the appropriate DomU (4), where it is processed again.

The measured data path starts with the creation of the CAN message within DomU. It stops, when the driver within DomM has fully received the message. The timing behavior measured here is equivalent to two distinct physical platforms connected over a CAN bus, without the need of complex time-synchronization.

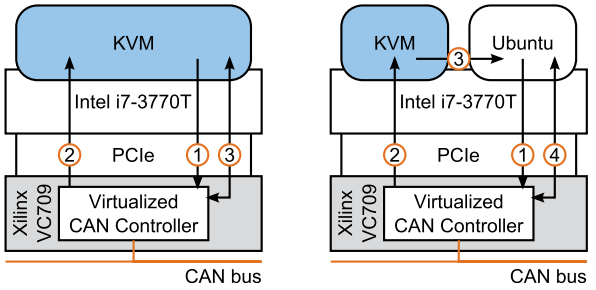
Fig. 4 shows the results obtained from the latency measurements of SW-based CAN virtualization on the Infineon AURIX TriCore 275 (see Table 1). The transmission time of the CAN messages on the bus for the respective payload sizes (*can-tx*) represents a lower limit for the overall end-to-end delay. The actual latencies for the native and virtualized case are larger due finite processing and memory access times. In the *native* case, the added latencies are produced by sending (around 8 µs) and receiving (around 11.5 µs) CAN messages within the software. To receive CAN packages there is little routing overhead produced by the hypervisor [14]. Again, the hypervisor acts as a monitoring function, and only intervenes in case of an exception. To transmit CAN messages, the communication controllers were addressed directly by the VMs, resulting in a native transmission behavior.

If CAN messages must be routed over a central DomM (see Fig. 3b), there is little routing overhead which is negligible.

There is a constant overhead to exchange CAN messages between DomU and DomM (around 35 µs). The message size dependency of the end-to-end latency is dominated by the transmission time on the CAN bus, whereas processing delays on the computing platform show little variation.

5.2 Hardware-Assisted Virtualization

HW-assisted virtualization provides near native I/O performance for VMs. We designed an experiment to compare latencies experienced in a native case with those seen from within a VM. Because there is no platforms with HW-virtualized CAN controllers available, we extended an Intel Core i7 platform with a Virtex-7 FPGA board, featuring a prototypical implementation of an SR-IOV-capable CAN controller as described in Section 4. The setup is depicted in Fig. 5.



(a) Native (b) Virtualized

Figure 5: Assessment of latencies using an SR-IOV enabled virtualized CAN controller. The experimented was conducted within the hypervisor to assess native performance and within a VM.

Fig. 5a represents the native scenario. A message is send via the virtualized CAN controller (1). After its complete transmission, it is also received in the controller and an interrupt is issued to the host system (2). Finally, the received message is read from the controller (3). Steps (1)-(3) are repeated 10,000 times to achieve reliable data.

When conducting the same experiment from within a VM, the sequence is slightly different. Because x86 CPUs are limited to either receive interrupts in host or in guest mode, we cannot directly signal the reception of a message to the VM. Rather, the interrupt triggers the hypervisor and is then forwarded to the VM (step (3) in Fig. 5b).

Throughout the experiment CPU functions that lead to performance variability are deactivated. This includes hyper-threading, SpeedStep and Turbo Boost. As hypervisor, we used KVM (3.8.13.6) and the VM is running Ubuntu 13.04. The CAN bus has a bandwidth of 500 kbit/s.

Fig. 6 shows minimum and average latencies measured in a native scenario and within a VM using HW-assisted virtualization. Plotted alongside is the transmission time of the respective message on the CAN bus (can-tx). The increase in overall latencies with the payload size is only caused by the increase in the bus transmission time, while CPU and I/O controller related latencies are constant.

Excluding the CAN transmission time, minimum latencies range around 13 µs in the native case and between 20 µs and 24 µs from within a VM. The difference in latencies can mainly be attributed to the additional interrupt forwarding necessary (step (3) in Fig. 5b).

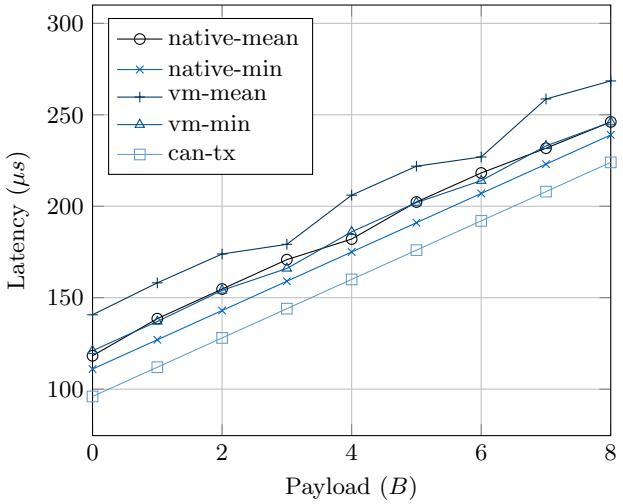


Figure 6: Latencies measured for a complete transmit-receive loop for native communication on from a VM on the Intel i7.

Compared to minimum latencies, mean latencies are 11 µs larger in the native case and 25 µs from within a VM. The latency variation can be attributed to interfering tasks on the CPU, cache and TLB pollution, and variable PCIe latencies. Interference from other tasks is rather an artifact within the measurement due to the lack of real-time support of the operating system. It is larger within the VM, as here, not only tasks within the VM but also the hypervisor can interfere. We assume the increased mean latencies for the VM compared to the native case are caused by the additional source of interference in the measurement.

5.3 Discussion

Above, we presented experimental assessments of virtualized CAN controllers using paravirtualized and HW-assisted virtualization. Due to the lack of virtualization support within current commercially available automotive CPUs, we chose a general purpose x86 platform to evaluate HW-assisted virtualization. When comparing results, it is important to consider differences in these platforms. Table 1 shows key properties of the platforms used in the experiments.

Table 1: Comparison of hardware platforms.

Property	Platform 1	Platform 2
CPU	Infineon AURIX TC275	Intel i7-3770T
# of Cores	3	4
CPU frequ.	200 MHz	2.5 GHz
Interconn.	System Peripheral Bus	PCIe 3.0
Price	~$10	~$294

The most significant difference is the CPU frequency (factor 30) and the resulting performance discrepancy. However, the AURIX is less complex an therefore requires significantly less cycles for many operations. For example, the CAN nodes of the AURIX are connected through a specialized on-chip interconnect called System Peripheral Bus. It

is designed for predictable low-latency communication. On the other hand, the virtualized CAN node used with the Intel platform is not directly connected to the CPU. Rather, it connects through PCIe to the Platform Controller Hub, which itself is connected through another PCIe to the Intel Core i7. Here, further on-chip communication resources are involved.

Latencies in the native cases are similar. On the AURIX, we measured added latencies around 20 µs and 13 µs on the Core i7. However, added latencies introduced in the virtualized cases differ more significantly with 35 µs for the AURIX, but only 7 µs for the Core i7. Nevertheless, send/receive latencies are sufficiently small (smaller than CAN transmission time) in both platforms to achieve line-rate performance.

An aspect which was not covered by the experiments is interference. If multiple VMs send messages simultaneously, their requests have to arbitrated. The added latencies from interference are also expected to be larger using software.

6. CONCLUSION

Current scalability issues of automotive embedded systems can be solved through the introduction of multi-core CPUs and virtualization. However, this poses challenges for shared network-I/O devices. Here, we addressed the problem of virtualization a CAN controller so that multiple concurrent VMs can access it collision free.

The evaluation of virtualized CAN controllers using paravirtualization and HW-assisted virtualization demonstrated the applicability of both approaches. HW-assisted virtualization is currently not possible in automotive multi-core controllers, but seems desirable due lower latencies and better predictability. Paravirtualization is a valid approach to enable I/O sharing in current commercial platforms. In a long-term perspective, it would be preferable if chip vendors introduce dedicated HW support for virtualization and thus improve the efficiency of such platforms.

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.

7. REFERENCES

[1] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. *Proceedings of Embedded Real Time Software and Systems*, pages 36–42, 2010.

[2] M. Di Natale and A. L. Sangiovanni-Vincentelli. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proceedings of the IEEE*, 98(4):603–620, 2010.

[3] 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.

[4] 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.

[5] 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)*, pages 7–13, 2013.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] H. Rauchfuss, T. Wild, and A. Herkersdorf. A network interface card architecture for i/o virtualization in embedded systems. In *Proceedings of the 2nd conference on I/O virtualization*. USENIX Association, 2010.

[11] D. Reinhardt, M. Güntner, and S. Obermeir. Virtualized Communication Controllers in Safety-Related Automotive Embedded Systems. In *Architecture of Computing Systems (ARCS), 2015 28th International Conference on*, Mar. 2015.

[12] D. Reinhardt, D. Kaule, and M. Kucera. Achieving a scalable e/e-architecture using autosar and virtualization. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 6(2):489–497, 2013.

[13] 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*, pages 221–226, 2013.

[14] D. Reinhardt and G. Morgan. An embedded hypervisor for safety-relevant automotive E/E-systems. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 189–198, June 2014.

[15] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer, and C. Hilbert. Towards automotive virtualization. In *Applied Electronics (AE), 2013 International Conference on*, pages 1–6. IEEE, 2013.

[16] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.