TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Integrierte Systeme

Enablement of Multi-Core-Based Automotive Embedded Systems through I/O- and Network Virtualization

Christian Herber

Vollständiger Abdruck der von der Fakultät für Elektrotechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Georg Sigl
Prüfer der Dissertation:	1. Prof. Dr. sc. techn. Andreas Herkersdorf
	2. Prof. Dr. sc. Samarjit Chakraborty

Die Dissertation wurde am 15.01.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 15.06.2016 angenommen.

Acknowledgments

I want to thank advisers, colleagues, partners, and friends who helped me making this PhD thesis a success story.

First, I would like to express my gratitude towards my doctoral supervisor Professor Andreas Herkersdorf. His guidance and scientific input enabled me to quickly gain traction in my research, produce continuous scientific output, and finally conclude my work with this thesis.

Further thanks go to:

My second examiner Professor Samarjit Chakraborty

Dr. Thomas Wild, for his valuable input in technical and organizational aspects

Andre Richter, whom I worked with in the ARAMiS project. Through our joint work on publications and project related activities like demonstrators, I was able to benefit from his technical expertise.

Holm Rauchfuss, who supervised me during my Master's thesis and is also responsible for writing the research proposal for the ARMAiS project, which accompanied me for more than three years.

All other colleagues, whom I cannot mention individually - you made life at the institute a very enjoyable, social, and diverting affair.

All students whom I advised in Bachelor's, Master's, or seminar thesis or who contributed as working students. Particularly, I want to mention Ammar Saaed, whose long term engagement led to a joint publication.

Partners form academia and industry within the ARAMiS project, who were very open in sharing their expertise. The closest collaboration was done conjunction with Intel Labs Europe, BMW Forschung und Technik GmbH, and BMW AG.

Finally, I want to thank my parents, Gerd and Roswitha Herber. Their loving support for 28 years forms the foundation this thesis is built upon.

Abstract

More than 70% of innovation in modern cars can be attributed to developments in electronics and associated software components. A steady growth in computing power and communication resources is necessary to sustain the introduction of new functions. However, scalability limits have been reached in automotive electronics due to installation space and weight constraints, so that the number of electronic control units (ECUs) cannot be increased any further. To enable continuing innovation, ECUs must be consolidated on shared hardware platforms. In this context, multi-core processors are a key technology, because they allow concurrent execution of previously distributed functions on one silicon chip. While this tight integration can solve today's scalability issues, it introduces new technological challenges.

The consolidation of electronic functions on multi-core ECUs and the introduction of new functions lead to increased communication demands. In future, consolidated ECUs could be interconnected by a backbone network. Such a network must support high bandwidth and real-time communication at the same time. This requirement can be satisfied by real-time capable Ethernet versions, e.g. Audio Video Bridging (AVB) Ethernet.

Electronic functions which are consolidated may differ in their requirements for safety, security, real-time capability etc. In such mixed-criticality scenarios, isolation among functions is of key importance for all shared components. Virtualization is a technology that is successfully used in server environments to provide isolated workload consolidation. To be applicable to multi-core ECUs, domain specific challenges like real-time capability have to be solved.

This thesis focuses on the enablement of communication in virtualized multi-core ECUs considering automotive specific requirements. To enable the use of legacy networking technologies in such scenarios, a hardware-assisted virtualization architecture and implementation for Controller Area Network (CAN) controllers is presented. Additionally, a network virtualization approach is introduced, which enables mixed-criticality communication on a CAN bus. To address the increased communication demand of multi-core ECUs, the integration of AVB Ethernet into legacy dominated automotive embedded systems is discussed. Specifically, gateway strategies between CAN and AVB Ethernet as well as AVB communication controllers are researched.

Zusammenfassung

Über 70% der Innovation in modernen Automobilen wird durch Fortschritt in Elektronik und zugehöriger Software erreicht. Eine stetige Verbesserung von Rechenleistung und Kommunikationsressourcen ist nötig, um zukünftig einen Zuwachs an elektronischen Funktionen zu ermöglichen. Aktuelle Bordnetzarchitekturen können aufgrund von Bauraum- und Gewichtslimitierungen nicht weiter skaliert werden. Insbesondere ist es nicht möglich, die Anzahl der verbauten Steuergeräte zu erhöhen. Um weiter innovationsfähig zu sein, müssen Steuergeräte auf gemeinsamen Hardwareplattformen konsolidiert werden. Multicore Prozessoren stellen eine Schlüsseltechnologie dar, da sie die nebenläufige Ausführung zuvor verteilter Funktionen auf einem einzigen Chip erlauben. Diese Integration erhört die Skalierbarkeit fahrzeuginterner Elektronik, führt jedoch auch zu technologischen Herausforderungen.

Durch Konsolidierung auf Multicore Prozessoren und einen Zuwachs an elektronischen Funktionen steigen die Kommunikationsanforderungen einzelner Steuergeräte. In Zukunft könnten konsolidierte Steuergeräte durch ein Backbonenetzwerk verbunden sein. Solche Netzwerke müssen einen hohen Durchsatz mit Realzeitfähigkeit verbinden. Diese Anforderung wird durch realzeitfähige Ethernet Varianten erfüllt wie z.B. Audio Video Bridging (AVB) Ethernet.

Konsolidierte elektronische Funktionen können sich in ihren Eigenschaften in Bezug auf Sicherheit, Realzeitfähigkeit etc. unterscheiden. Gemeinsam genutzte Ressourcen müssen eine Isolation von Funktionen unterschiedlicher Kritikalität sicherstellen. Virtualisierung ist eine etablierte Technologie im Serverbereich, welche eine sichere Integration mehrerer Betriebssysteme auf einer gemeinsamen Plattform in so genannten Virtuellen Maschinen ermöglicht. Um in Multicoresteuergeräten eingesetzt werden zu können, müssen domänenspezifische Herausforderungen adressiert werden.

Diese Arbeit erforscht Beiträge zur Unterstützung von Kommunikation in virtualisierten Multicoresteuergeräten. Um den Einsatz bestehender Netzwerktechnologien in solchen Szenarien zu ermöglichen wird eine dedizierte Hardwareunterstützung zur Virtualisierung von Controller Area Network (CAN) Controllern eingeführt. Zusätzlich wird ein Ansatz zur Netzwerkvirtualisierung von CAN Bussen präsentiert, wodurch isolierte Kommunikationskanäle auf einem einzigen physikalischen Bus geschaffen werden. Der erhöhte Kommunikationsbedarf von Multicoresteuergeräten wird durch die Integration von AVB Ethernet in bestehende Bordnetze adressiert. Im Speziellen werden Gateways zwischen CAN und AVB Ethernet sowie AVB Kommunikationscontroller erforscht.

Contents

List of Figures

List of Tables

I. Introduction

1. Motivation

Today's automotive embedded systems are highly complex networks of up to 100 computing nodes distributed throughout the car. In automotive terms, such computing nodes are called Electronic Control Units (ECUs). Historically, new electronic functions have been added to cars through the introduction of an additional ECU. This method allows an integration of functions in a way that little or no modification is required in legacy components. However, it is inefficient in terms of resource utilization and overall system complexity. This does not only include the number of ECUs, but importantly also the wiring harness. The length of a BMW 7 series wiring harness is 2.7 km [1]. Due to space limitations, scalability of the current architecture is limited.

Nevertheless, new applications have to be introduced to maintain a high level of innovation. Major trends are advanced driver assistance systems (ADAS) or even highly autonomous driving [2], the integration of cars into cyber-physical systems (CPS) [3], and powertrain electrification [4]. To enable such applications, significantly increased computing and communication resources are necessary [5]. For example, a camera-based lane keeping system is a safety critical system, which has to process video data in real-time and make appropriate control decisions. To be able to deliver the necessary performance, embedded system architectures in cars have to be modernized.

Recognizing the limited scalability of today's decentralized approach using one function per ECU and a central gateway, car manufacturers are looking to shift the architecture paradigm to a centralized one [6, 7, 8]. The main goal is to reduce the total number of ECUs and the complexity of the wiring harness. If multiple functions can be consolidated on a single processor, it not only reduces space and weight of the component but also improves communication among functions, because on-chip resources can be used instead of fieldbuses.

Shifting to the use of multi-core processors in automotive electronics is key to achieve centralized control units [9]. The integration of multiple cores on a single chip allows truly parallel execution of code. Thus, functions of previously parallel ECUs can be running side-by-side on a shared hardware platform. While multi-core processors offer isolated computing cores, much of the remaining hardware infrastructure is shared among cores. Resource sharing among functions can lead to performance degradation through temporal interference. This has been demon-

strated for many components like memory [10] or on-chip interconnects [11]. The resulting challenges with respect to real-time capability, safety, and security, which cannot be satisfied using today's technology.

In server environments, platform virtualization is an established technology to consolidate servers on a shared computing system. By inserting a privileged software layer, the so called hypervisor or virtual machine monitor (VMM), multiple operating systems (OSs) can be executed concurrently in an isolated fashion. OSs are running in virtual machines (VMs), which are abstract replicas of the actual physical system. Ideally, the behavior of a VM should be as close as possible to that of the physical system. However, in order to allow safe resource sharing, certain privileged instructions have to be monitored and moderated by the hypervisor. The isolation properties of virtualization could be used in an automotive setting to enable the concurrent execution of functions on a shared platform [12, 13, 14, 15, 16].

Due to the strong historic association of virtualization with server environments, current virtualization solutions are not designed to cater the specific needs of automotive applications. The main optimization goals for servers are high throughput and high utilization. Thus, virtualization has been optimized to be efficient for general purpose workloads, but doesn't address e.g. real-time requirements. Moreover, virtualization is nearly exclusively applied to x86 processor systems, while in the automotive domain other architectures like Infineon TriCore or PowerPC are present. To apply virtualization to automotive electronics, the architecture and associated requirements have to be understood.

ECUs are grouped into so called functional domains [1, 5]. Within each of these functional domains, ECUs are connected through one or more communication networks, mostly fieldbuses. Inter-domain communication is possible through a central gateway, which connects all domains. The total number and naming of domains differs among OEMs. An example configuration could include power train, chassis, body, comfort, infotainment, and driver assistance domains.

While non-functional requirements are homogeneous within a domain, they can be quite heterogeneous among different domains. For example, an anti-lock braking system is a safety critical chassis component with hard real-time requirements. On the other hand, a rear-seat entertainment system is neither safety critical nor constrained by real-time requirements. Loads in such a system are dynamic and may require significantly more computing resources and network throughput than control applications. Additionally, to provide dynamic content, entertainment systems rely on wireless interfaces and are consequently prone to attacks [17]. Therefore, security concerns are more prominent in such applications scenarios.

When functions from different domains are consolidated, mixed-criticality scenarios emerge. In such scenarios, functions of different criticality run concurrently using shared computing and communication resources. In an automotive setting, criticalities are usually associated with the so called automotive safety integrity level (ASIL). To allow integrated execution of functions with different ASIL, strong iso-

lation is paramount. If isolation cannot be guaranteed, all functions would have to developed according to the highest ASIL requirements on the platform, which is unfeasible due to the high qualification cost of highly safety-critical functions.

Due to the heterogeneous requirements and historic development, different networking technologies are used in automotive embedded systems. The list of technologies includes Controller Area Network (CAN), FlexRay, Local Interconnect Network (LIN), MOST, Ethernet, and LVDS. Due to its robustness, real-time capability, and cost-efficiency, CAN is the most widely used networking technology in automotive embedded systems. It provides up to 1 Mbit/s using an unshielded twisted pair as physical layer. Within a single car, up to eight CAN buses are used, yet the maximum bandwidth is limited to 500 kbit/s due to EMC concerns. In future, CAN FD (flexible data-rate) could be used to increase the throughput to around 10 Mbit/s [18]. Ethernet is currently used as technology for flashing ECUs and for diagnostics. In future scenarios, it could be used in infotainment, as backbone and potentially even for control applications.

Consolidation of functions in automotive embedded systems also requires architectural changes. Car manufacturers are looking to eliminate the bottleneck of the central gateway by connecting centralized nodes through a backbone network [19]. These consolidated ECUs serve as so called domain control units (DCUs), which are connected with sensors and actuators through a number of fieldbuses. The backbone network has high throughput requirements, but is safety-critical and real-time constrained at the same time.

To provide high throughput and real-time transmission at the same time, Ethernet-based networking technologies are being considered [20]. Legacy Ethernet has limited real-time capabilities [21]. Nevertheless, multiple real-time capable Ethernet variations exist including AVB Ethernet, TTEthernet, or PROFINET. Such technologies could be used to satisfy communication requirements of future automotive embedded systems.

The combination of multi-core processors and virtualization has the potential to enable centralized and consolidated control units to overcome the scalability issues of today's automotive embedded systems. However, technical key issues have to be addressed in order to produce a safe and secure solution. Within the scope of this thesis, communication among virtualized multi-core ECUs is the focus. Two key challenges for networking in future automotive embedded systems will be addressed within the scope of this thesis:

1. Legacy components are an important aspect in automotive systems. Because it is too costly to develop a car from scratch, legacy ECUs and thus legacy networking technologies will be a part of future car generations [22]. Therefore, it has to be ensured that legacy networks and network controllers can fulfill the requirements of future architectures like the support of virtualization and mixed-criticality.

2. To keep up with rapidly increasing application requirements, new networking technologies are required. To provide sufficient bandwidth and real-time operation at the same time, real-time capable Ethernet derivations can be used. However, such technologies have to be integrated efficiently into the existing ecosystem and interface legacy fieldbuses.

2. Problem Statement and Contributions

The sharing of network-I/O devices has traditionally been a major challenge in virtualized systems. Mostly, SW-based solutions are used to enable I/O sharing, where a privileged SW component moderates and forwards I/O transfers. The additional data copy operations and context switches introduce significant computing overheads and added latencies. To reduce I/O-virtualization overheads, HW-assistance for I/O-virtualization has been introduced. However, current solutions only focus on I/O devices for networks used in server environments (Ethernet, InfiniBand).

To enable sharing of automotive I/O devices, solutions have to be found, which address domain specific requirements. In contrast to server systems, where throughput is the most important property, automotive systems need to satisfy real-time and safety requirements. Within this thesis, **virtualization extensions** are researched for CAN, the most widely used automotive bus. A layered architecture is proposed that extends a standard CAN controller so that it can be used by multiple tenants. A virtualization layer within the hardware fulfills the tasks of moderating and forwarding CAN I/O request from and towards VMs.

In mixed-criticality scenarios, the sharing of resources should not lead to safety or security issues. Therefore, it must be ensured that the performance experienced for one virtual CAN controller is independent of the utilization of other virtual instances. This is achieved through a **temporal isolation** mechanism, which schedules any requests towards virtual CAN controllers in order to optimize utilization and minimize added latencies.

While hardware-assisted virtualization is a mature technology that provides near-native performance in x86 systems, there are still issues remaining with respect to interrupt forwarding. Interrupts still require forwarding from host into guest mode, and thus every interrupt creates a context switch. To reduce the resulting performance overhead, modern Ethernet Network Interface Controllers (NICs) implement interrupt coalescing mechanisms, which jointly forward multiple interrupts. Because such mechanisms lack real-time capabilities, the concept of **deadline-aware interrupt coalescing** is introduced and implemented using the example of a virtualized CAN controller.

Platform virtualization allows the consolidation of mixed-criticality functions in centralized control units. Nevertheless, automotive embedded systems are naturally distributed, because the centralized nodes have to communicate with sensors and actuators. To minimize wiring efforts, it is necessary that functions of different criticality can use shared networking infrastructures. Current networking technologies used in automotive systems like CAN lack the ability to efficiently support mixed-criticality traffic. Here, **network virtualization for CAN** is researched. The proposed concept divides the physical bus into multiple virtual communication networks. The approach guarantees performance and fault isolation among concurrent virtual networks in mixed-criticality scenarios.

In addition to the enablement of legacy networks, the integration of new networking technologies is key to satisfy the communication requirements of multi-core based applications. Connecting consolidated ECUs through a backbone network requires high bandwidth and real-time capabilities at the same time. Audio Video Bridging (AVB) Ethernet is a favored candidate, which extends legacy Ethernet with traffic shaping and policing as well as time synchronization to achieve real-time capable communication. However, AVB Ethernet is significantly different to current networking technologies with respect to many properties like packet size, network architecture, admission arbitration etc. Thus, **enabling AVB Ethernet** for use in automotive embedded systems is an important challenge.

One problem researched here is how to efficiently interconnect AVB Ethernet with a legacy fieldbus technology, specifically CAN. A **CAN to AVB Ethernet gateway** connecting these networking technologies has to overcome significant protocol and technology differences. Through packet aggregation and scheduling, efficient and real-time capable forwarding is achieved here.

Additionally, research was conducted in the field of **AVB Ethernet endpoints**. As AVB Ethernet is a relatively new technology, there is a lack in practical experience regarding the design and integration of HW/SW systems supporting AVB Ethernet. Thus, an AVB Ethernet controller was developed and integrated into a multi-core platform.

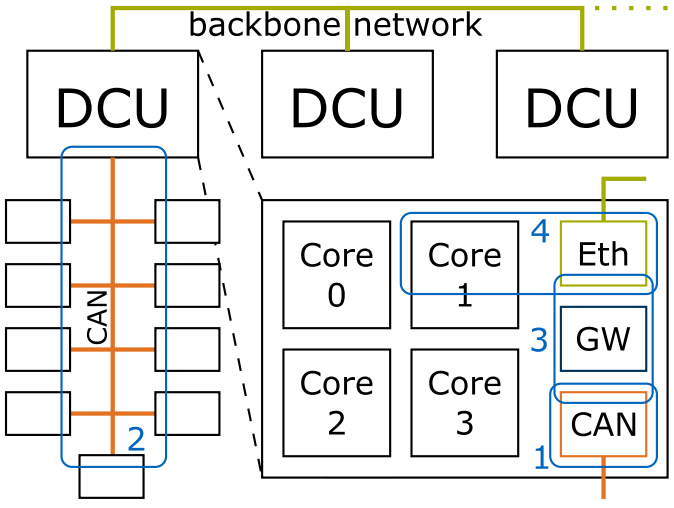


Figure I.1.: Contributions towards the enablement of multi-core ECUs in future automotive embedded systems.

Fig. I.1 summarizes the contributions of this thesis and illustrates, how their combination contributes to the enablement of multi-core ECUs in future automotive embedded systems. The enumerated areas within the figure correspond to the following contributions:

1. Virtualization of CAN controllers to provide efficient, scalable and real-time

capable communication in multi-core ECUs. This includes a layered architecture for HW-assisted virtualization, a temporal isolation scheme, and the deadline-aware interrupt coalescing mechanism

2. Network virtualization for CAN to enable mixed-criticality communication on a shared physical medium

3. A real-time capable CAN to AVB Ethernet gateway using frame aggregation and scheduling to maximize its efficiency

4. Design and integration of an AVB Ethernet controller into a multi-core SoC

3. Structure and Organization

The thesis is structured as follows: Chapter II introduces the state of the art in automotive embedded systems and virtualization. In chapter III, a concept for a virtualized CAN controller is proposed. A composable hardware architecture is developed, in which a virtualization layer is added to enable multi-tenancy for CAN controllers. A network virtualization approach for CAN is presented in Chapter IV. It subdivides the physical communication channel of the CAN bus into multiple, concurrent virtual networks. The enablement of AVB Ethernet is discussed in Chapter V. It includes the gateway for CAN to AVB Ethernet communication as well as the design of an AVB capable endpoint device. Finally, this thesis is concluded in Chapter VI.

II. State of the Art

This chapter presents and analyzes the state of the art in technologies relevant to this thesis. As this work aims to improve automotive embedded systems (Section II.1) through the introduction virtualization (Section II.2) techniques, these two areas are also the main focus. To get a sufficient view on the state of the art, this chapter provides information on industrial standards and technologies, commercial products, established research results, as well as ongoing research activities and trends.

1. Automotive Embedded Systems

The technical contributions of this thesis are primarily focused on automotive application scenarios. Typical requirements, embedded systems architectures and networking technologies relevant to automotive electronics will be introduced here.

1.1. Non-Functional Requirements

Besides functional requirements which specify the tasks a component or system has to fulfill, non-functional requirements are prominent in automotive systems. They specify the way in which the completion of a task must be fulfilled. A typical example is a temporal requirement that specifies a time span, in which a task must be performed. The most relevant requirements will be detailed below.

1.1.1. Real-Time Capability

Real-time capability is a central requirement in automotive embedded systems. In contrast to conventional systems, where an electronic function must simply deliver a correct result, the timely delivery of said result is just as important in a real-time system. For example, in an engine control system, the amount of fuel injected into a cylinder must be determined before the designated time of injection. This process involves reading out sensor values, calculating the value, and communicating it to the corresponding actor. The overall time needed consists of computation and communication delays.

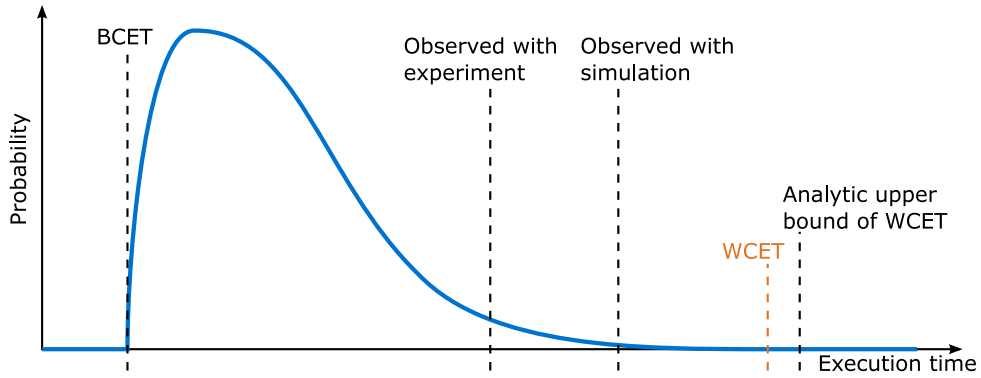


Figure II.1.: Example distribution of execution times illustrating worst-case prediction methods

To validate a system's ability to satisfy real-time constraints, several methods can be used including analytic, simulative and experimental approaches. Fig. II.1 shows a common distribution of execution times in a scheduled system. In this context, an execution time is defined as the time from the release of a task until its completion. The lowest possible execution time is called best case execution

time (BCET). Average execution times are often close to the best case, whereas the probability of execution times close to the worst-case are steadily decreasing. The longest possible execution time is called worst-case execution time (WCET). It happens only if multiple worst-case conditions occur at the same time and is thus unlikely to happen.

In an experimental assessment of execution times, a real system is stimulated through real or synthetic workloads. A measurement setup is used to determine execution times. Thus, a statistical distribution of execution times can be obtained. However, minimum and maximum execution times are not necessarily equal to BCET and WCET, because the experiment can only be carried out for a limited amount of time (usually hours or days).

The coverage of execution times can be improved through simulation. Simulation uses a computer model replicating the real system as close as possible. Through abstraction, a speed up can be achieved over experimental assessments. The precision of a simulation relies on the accuracy of the model used. Compared to an experiment, it introduces potential error due to modeling imprecision. At the same time, simulation does not require the actual hardware setup and can thus be done in earlier design stages, where no running prototype exists.

Finally, real-time analysis can be used to determine bounds for the WCET and BCET. Real-time analyses use mathematical models of the systems at stake to obtain the relevant information. In classic real-time analysis, no execution time distribution is obtained, but rather bounds for the measures of interest. An exception to this are stochastic real-time analyses. Real-time analysis is the only of the three methods introduced which is capable of providing an upper bound for WCET. This is important in safety-critical systems, as it allows to engineer a system, in which deadlines are guaranteed to be met. The mathematical model often includes abstractions or simplifications due to system complexity. To ensure the integrity of the analysis, the modeling must provide pessimistic guarantees.

Such methods are important in any scheduled system, not only processing systems. In networking, communication delays can be addressed in a similar fashion. In this context, the time from the release of a communication message until its complete reception at the receiver is called response time. Thus, the corresponding metrics are called worst/best-case response time (WCRT/BCRT).

1.1.2. Safety

Electronic functions implemented within automotive embedded systems can be highly safety-critical. Therefore, reliable hard- and software are necessary to guarantee safe operation of the car as a whole. How safety is guaranteed in an automotive context in standardized in ISO 26262 [23].

The standard defines multiple criticality levels. Such levels are called Automotive Safety Integrity Level (ASIL). The ASIL of a component is determined through a

hazard assessment. All possible hazardous events are classified in terms of their severity (S0: No injuries, ..., S3: Life-threatening), exposure (E0: Incredibly unlikely, ..., E4: High probability), and controllability (C0: Controllable in general, ..., C3: Difficult to control or uncontrollable). Based on this assessment, a safety goal is defined. The most critical level, ASIL D, is only reached for a combination of {S3, E4, C3}. Uncritical components are grouped into QM (Quality Management) and are not handled using ISO 26262.

Safety requirements vary between the functional domains. Only Powertrain, Chassis, and Driver Assistance reach the highest level ASIL D. Other domains like Comfort and Infotainment usually do not require qualification at levels higher than ASIL B [1].

Another important aspect within ISO 26262 is so called "freedom of interference" [23]. Freedom of interference is particularly important when using commercial of the shelf (COTS) components and in mixed-criticality scenarios. When hardware components are shared by function of mixed-criticality, it must be ensured that low-criticality do not interfere with highly safety critical functions.

1.1.3. Security

Historically, security was not a major concern in the automotive design process. Security concerns were only addressed in the form of anti-theft features. Cyber security was not considered.

In the last years, connectivity of cars has increased significantly with the introduction of wireless communication interfaces like Bluetooth, WiFi, and LTE. The increase in wireless interfaces leads to a growing attack surface [17, 24].

At the same time, the power of electronic functions in cars is increasing. Due to the introduction of advanced driver assistance, electronic systems increasingly take over the jobs like steering, acceleration, and breaking. At Black Hat 2015, a talk with the title "Remote Exploitation of an Unaltered Passenger Vehicle" by Charlie Miller and Chris Valasek showcased how an attacker can gain access and control a car via remote[1]. By gaining access to the head-unit through a mobile data interface, the hackers were able to perform a code injection towards a so called fast-boot microcontroller. This is a processing subsystem, which provides CAN access while the head-unit is still booting. With the gained CAN access, critical actors of the car like steering and brakes could be controlled. Such an attack could be prevented by applying state of the art counter-measures from the IT world like code signing [25].

In this thesis, security is a concern in the context of ECU consolidation. While most attack vectors are present in non safety-critical domains like infotainment [1], consolidation potentially leads to an integration of attack vectors with safety-critical

[1] `http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101`

domains. Thus, isolation among these partitions must be ensured within all components.

1.2. Embedded System Architecture and Technology

The architecture of automotive embedded systems is divided into functional domains, in which electronic control units (ECUs) communicate through a number of in-vehicle networks. This architecture is historically grown, highly complex, and heterogeneous. Currently, around 70 ECUs or up to 100 ECUs in high-end configurations are implemented in automotive systems. The overall wire length adds up to 2.7 km [1].

ECUs are components used to realize electronic functions within the car. Typically, there is a one-to-one mapping of functions to ECUs. Functions can range from safety-critical applications like engine control to comfort applications like seat control. ECUs usually implement at least one microcontroller. In safety-critical domains, automotive specific microcontrollers like Infineon AURIX or Freescale MPC are used that are designed according to ISO 26262 and provide features for fault-recognition like lock-step operation.

A simplified example architecture is illustrated in Fig. II.2. Key to the architecture is a component called the central gateway. It interconnects networks from all functional domains to allow inter-domain communication. Further details regarding gateways are presented in Section II.1.5.

The number and naming of functional domains varies for different car manufactures. Nevertheless, key domains can be found in nearly every car. They include:

- **Powertrain:** Components related to functions, which help the car to generate and deliver power in the direction of motion are grouped in the powertrain domain. This includes engine control, transmission, exhaust system etc. These components are interconnected through a high-speed CAN bus (500 kbit/s). This domain is safety-critical and its configuration is static.

- **Chassis:** Electronic functions which control the interaction of the car with the road are grouped in the chassis domain. Example functions include antilock braking, electronic stability programs, active damping, or steer-by-wire. Properties are similar to powertrain: real-time, safety-critical, and static configuration. Because chassis applications require low latencies, FlexRay is the networking technology of choice.

- **Infotainment:** Head-unit, instrument cluster, navigation, and entertainment applications including audio and video and associated functions constitute the infotainment domain. In contrast to powertrain and chassis, only lower criticality levels are reached (up to ASIL B). Due to the high connectivity of the infotainment domain through several interfaces like WiFi, Bluetooth, and

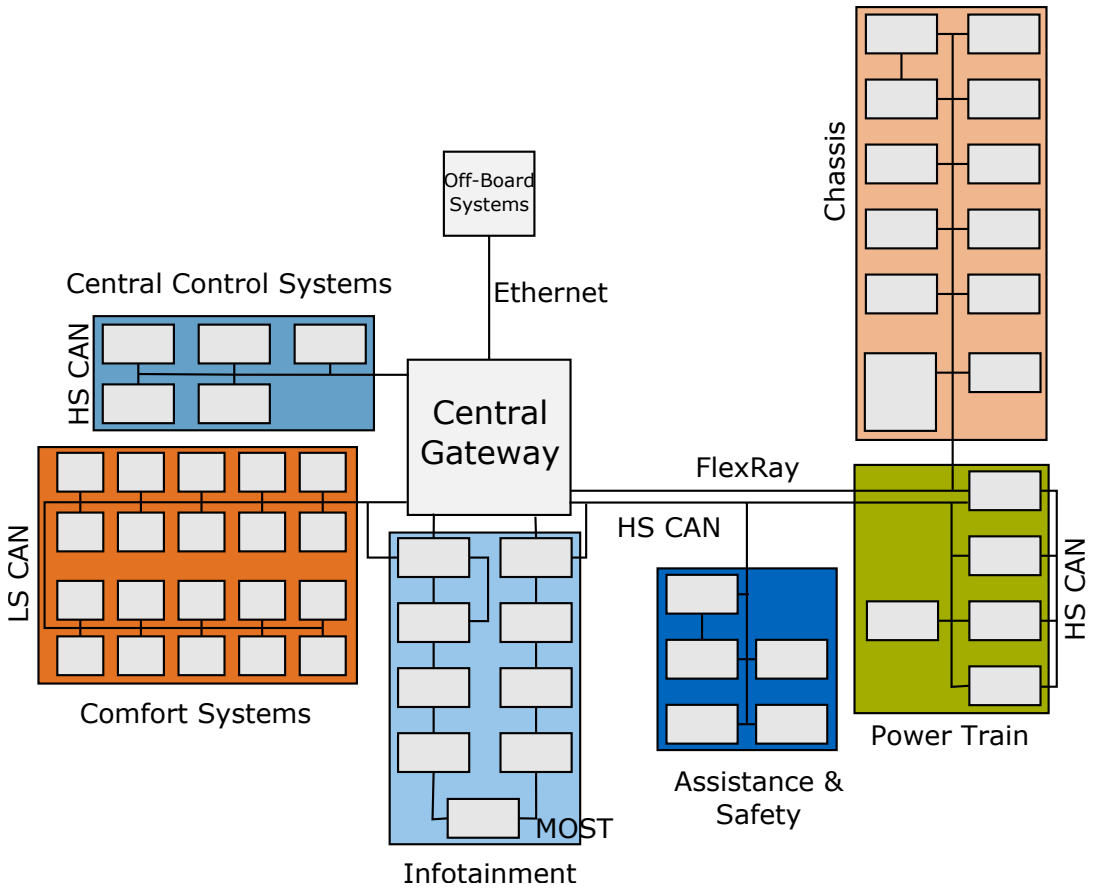


Figure II.2.: Example architecture of an automotive embedded system (simplified), adapted from [1]

wireless communication, it has the largest attack surface of all domains. Thus, security considerations are an important factor, while real-time and safety play a minor role. Current infotainment systems use MOST for audio/video transmission and optionally LVDS to connect camera systems. In future, these could be replaced by Audio Video Bridging (AVB) Ethernet.

- **Body:** This domain includes basic electronic functions like interior and exterior lighting, wipers, windows, car-access etc. It is implemented using one or multiple CAN buses. Due to the highly distributed nature of the functions, it is associated with high wiring effort.

Because sensors and actuators are distributed throughout the car, networking plays an important role within the car. The key networking technologies currently used in state of the art cars are:

- **Local Interconnect Network (LIN):** The cheapest networking technology LIN [26] uses master-slave topology on a shared bus to provide microcontrollers

with access to sensors and actuators. The highest possible bandwidth is 20 kbit/s.

- **Controller Area Network (CAN):** CAN [27] is a shared bus with a single logical bit line. It provides higher bandwidth than LIN of up to 1 Mbit/s, though in automotive applications only 500 kbit/s or less is used due to electro-magnetic interference considerations. Detailed information about CAN is given in Section II.1.3.

- **FlexRay:** FlexRay [28] uses time-triggered network arbitration to provide highly predictable and robust latencies. Data rates reach up to 10 Mbit/s.

- **Media Oriented Systems Transport (MOST):** MOST [29] is a serial bus developed for Audio/Video transmission. It is specified for 25, 50 and 150 Mbit/s. While it is currently the de-facto standard for infotainment networking, car makers are looking to replace MOST e.g. through the introduction of Ethernet.

- **Low-voltage differential signaling (LVDS):** LVDS is a point-to-point connection using twisted pair. It offers high bandwidth up to 655 Mbit/s and is typically used as flat panel display link. In the automotive context, it is used to connect camera systems. Point-to-point wiring is inefficient and the cables are expensive so that other solutions are required in the mid-term.

- **Ethernet:** In current automotive embedded systems, Ethernet is used for debugging and ECU flashing. In future cars, Ethernet could be used to replace LVDS and MOST or even in real-time scenarios [30]. Real-time capable variations of Ethernet exist, e.g. TTEthernet or Audio Video Bridging Ethernet (see Section II.1.4).

Current automotive embedded systems are facing significant scalability issues. The historical approach of introducing a new ECU for every function has led to an increase in the number of ECUs that can longer be continued due space and weight limitations. Thus, future automotive architectures need to consolidate functions on shared hardware platforms. Car manufactures are propagating the idea of so called domain controlled architectures, in which one powerful ECU (the domain control unit (DCU)) implements the majority of all functions. Simpler slave nodes remain to provide access to distributed sensors and actuators. A major challenge in the realization of DCUs is that multiple electronic functions have to run on a shared platform while guaranteeing freedom from interference (see Section II.1.1.2).

A second issue of the current architecture is the central gateway. As a single component, it is the designated traversal point for any inter-domain communication. Therefore, throughput requirements towards the central gateway scale directly with the amount of communication from the domains. A network backbone [19] could be

used to overcome these issues by decentralizing the task of inter-domain forwarding. Within an automotive embedded system, a backbone is a high bandwidth network that interconnects functional domains. Ideally, this backbone should be real-time capable to enable safety-critical inter-domain communication.

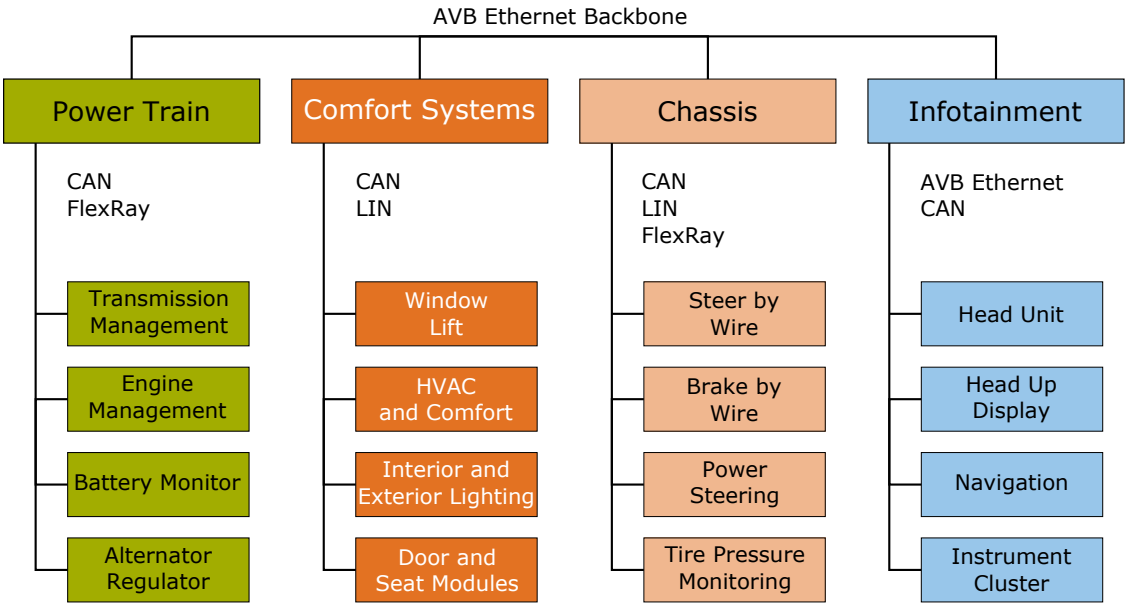


Figure II.3.: Domain control architecture with backbone network, adapted from [31]

Fig. II.3 shows an example of a domain controlled architecture. The main challenges arising are the safe and secure consolidation of functions within DCUs and the integration of a backbone network. To implement DCUs, multi-core processors are a key technology. By providing separate computing cores, they allow parallel execution of previously isolated workloads. Nevertheless, additional isolation is needed, which can be achieved e.g. through virtualization (see Section II.2).

1.3. Controller Area Network (CAN)

Controller Area Network (CAN) is an asynchronous serial bus, which is widely used in automotive electronics. It has a single logical bit line and can be implemented using an unshielded twisted-pair wire. CAN features error management and bounded latencies, which enables it to serve as communication medium in safety-critical domains. The following sections highlight basics regarding the CAN's communication protocol and timing analysis.

1.3.1. CAN Communication Protocol

CAN messages are transmitted through so called data frames. The structure of a CAN data frame is illustrated in Figure II.4. The beginning of a frame transmission is indicated by a Start-of-Frame (SoF) bit. It is followed by the message identifier (MSG ID), which specifies the contents of the message. CAN is a broadcast medium and does not use sender or receiver addressing. Each receiving node has to decide based on the ID if it is interested in the message contents. The CAN 2.0 standard allows the use of extended IDs, which contain 29 instead of 11 bits. However, due to the increased framing overhead, real implementations are usually constraint to standard IDs. The data length code (DLC) specifies the payload length to an integer value of 0 to 8 B. The actual payload is followed by a cyclic redundancy check. Receiving nodes acknowledge the reception of the frame using the ACK bit. The frame is concluded by an End-of-Frame (EoF) field containing 7 bits. CAN frames have a minimum interframe gap of 3 bits.

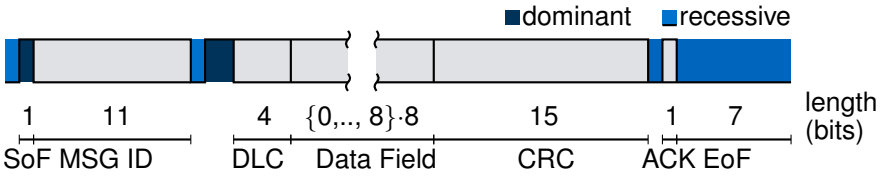


Figure II.4.: Standard CAN data frame [27]

CAN uses a concept of dominant and recessive logic levels: Multiple nodes can write a bit simultaneously, but if at least one dominant bit was written, only this bit will be visible for receivers. The dominant bit level corresponds to a logical '0', and the recessive to a '1'.

The message arbitration uses a strict priority scheme. Priorities are determined based on the MSG ID, where the numerically smallest ID has the highest priority. It uses dominant and recessive logic levels to guarantee a collision free arbitration phase. Nodes that want to send a message start participating in the arbitration process by sending the first bit of the ID. Each node samples this bit from the bus line. If a node sent a recessive bit, but samples a dominant bit, it recognizes a higher priority message and stops transmitting. This process is repeated until only one sending node remains.

CAN is an asynchronous bus, meaning that no shared clock exists. Nevertheless, CAN needs some form of coordination, as bits must be sampled at the same time to achieve consistent communication. CAN uses two forms of synchronization, hard and soft synchronization. If the bus was idle and a SoF bit gets transmitted, all nodes synchronize to the start of this bit (hard synchronization). Throughout the transmission of a frame, CAN nodes synchronize on positive and negative edges (soft synchronization).

To ensure the clock drift remains bounded, a change in polarity has to occur at

least every five bits. CAN uses bitstuffing, i.e. after five consecutive bits with equal polarity, a bit with opposite polarity is inserted. These bits are removed within the CAN controller of the receiver. While the bitstuffing is transparent for applications using CAN for communication, bitstuffing does affect the transmission time of a CAN frame.

CAN features an extensive error management protocol to detect and correct errors induced through electromagnetic interference (EMI) or hardware errors. Errors can be detected in different ways. A sender detects an error, if a different bit is sampled then the bit transmitted (excluding arbitration phase). Also, six consecutive bits of equal polarity trigger an error, as they violate the bitstuffing principle. Other errors can be detected through the CRC code or a general violation of the CAN data frame format. A single fault can trigger multiple of these errors. If an error is detected, the detecting node will transmit an error flag and the transmission will be repeated.

1.3.2. CAN Timing Analysis

CAN is often used in applications which require latency guarantees for the transmission of messages. Timing analysis of CAN has been an active research topic for over 20 years. This section covers the most essential parts of CAN timing analysis, which is used throughout major parts of this thesis. The presentation is based on the analysis of Davis et al. [32].

Goal of CAN timing analysis is to determine a pessimistic bound for the worst-case response time (WCRT) of all messages. The WCRT is defined as the time of the release of a message m until its successful transmission on the CAN bus. It is composed of the queuing delay w_m, the time it takes for the transmission to start, and the transmission time C_m, the time to transmit the message on the bus.

In worst-case analyses for CAN, it is assumed that maximum bitstuffing occurs. In this case, the transmission time of a standard CAN frame containing s_m payload bytes can be calculated as

$$C_m = (55 + 10s_m)\tau_{can}. \tag{II.1}$$

A necessary but not sufficient condition for schedulability is that the CAN utilization must be smaller than 100%. Each message is either assumed to be cyclic or to have a minimum inter-arrival time noted as T_m. Thus, the bus utilization can be calculated as

$$U = \sum_{\forall m} C_m/T_m. \tag{II.2}$$

CAN analysis is based on busy period analysis. A busy period is a time span in which a resource is blocked. For every message m, a different worst-case can be constructed and therefore, the busy period also differs. The level-m busy period t_m (maximum time during which the CAN bus is blocked by a message of priority m or

higher) can be constructed as follows. When message m is queued for transmission, the longest lower priority message has just started transmission. The time of this blocking B_m can be described as

$$B_m = \max_{\forall k \in lp(m)} (C_k), \tag{II.3}$$

where $lp(m)$ is the set of messages with a priority lower than message m. At the same time as message m is enqueued, all higher priority messages get queued as well. These higher priority messages are transmitted first. If, during this time period, additional messages of priority m or higher get queued, they will also contribute to the length of t_m. Also, the release of a message can be subject to a jitter J_m, e.g. due to variable software delays in the dispatching of a message. The level-m busy period t_m is given by the implicit formulation

$$t_m = B_m + \sum_{\forall k \in hp(m) \cup m} \left\lceil \frac{t_m + J_k}{T_k} \right\rceil C_k. \tag{II.4}$$

It can be solved through fixed point iteration. If $t_m > T_m - J_m$ it is possible that multiple instances q of message m get queued during t_m. In this case, the WCRT of all instances has to be analyzed. The actual number of instances, that have to be checked, can be computed as

$$Q_m = \left\lceil \frac{t_m + J_m}{T_m} \right\rceil. \tag{II.5}$$

The computation of the queuing delay is similar to (II.4). In contrast to the busy period, the queuing delay certainly ends after the transmission of the respective message instance. It can be determined as

$$w_m(q) = B_m + qC_m + \sum_{\forall k \in hp(m)} \left\lceil \frac{w_m(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k \tag{II.6}$$

for every instance q of message m. Using fixed point iteration, (II.6) can be represented in an explicit, iterative equation

$$w_m^{n+1}(q) = B_m + qC_m + \sum_{\forall k \in hp(m)} \left\lceil \frac{w_m^n(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k. \tag{II.7}$$

The WCRT for the q-th instance of message m can be computed by adding queuing delay, transmission time, and the release jitter

$$R_m(q) = J_m + w_m(q) - qT_m + C_m. \tag{II.8}$$

A general latency bound for a message m is consequently given as the maximum of WCRT of any message instance

$$R_m = \max_{\forall q \in Q} \left(R_m \left(q \right) \right). \tag{II.9}$$

It was shown in [33] that it is usually sufficient to only analyze the first instance. Message instances $q > 0$ only increase the WCRT R_m in high load scenarios (e.g. more then 99% for a CAN bus with 500 kbit/s), with a message set that is only just schedulable. In practical systems, bus loads are smaller and systems are not designed at the edge of schedulability to be able to respond to errors. While the analysis by Davis et al. [32] is the most general, it is sufficient to use the simpler version published earlier by Tindell et al. [34]. In this case, (II.7) is simplified to

$$w_m^{n+1} = B_m + \sum_{\forall k \in hp(m)} \left\lceil \frac{w_m^n + J_k + \tau_{bit}}{T_k} \right\rceil C_k. \tag{II.10}$$

1.4. Audio Video Bridging (AVB) Ethernet

Audio Video Bridging (AVB) is a real-time Ethernet technology, which is specified by a set of IEEE standards. It enables time-sensitive and synchronous communication by extending legacy Ethernet with stream reservation [35], traffic shaping and flow control [36], and time-synchronization [37]. AVB was originally developed to reduce the wiring effort in audio and video applications, where point-to-point connections are state of the art.

Features like high bandwidth, IEEE standardization, and low-latency communication make AVB Ethernet an interesting candidate for application in automotive embedded systems. Three possible applications exist:

1. Backbone: Upcoming embedded architectures in automotive electronics use backbone networks to interconnect different functional domains instead of a central gateway. Backbone networks need to transfer high data volumes, which include safety critical messages with strict real-time requirements. AVB Ethernet can fulfill these requirements.

2. Entertainment: Audio and video data of in-car entertainment systems is currently transferred through MOST. Ethernet is a cost-efficient replacement, which can cater the associated bandwidth requirements. While entertainment data is not safety-critical, it still requires reliable latencies to avoid buffer underruns.

3. Camera streaming: Currently, camera data is streamed using point-to-point LVDS connections. The increasing amount of cameras, especially used for

advanced driver assistance systems, makes a switched network solution increasingly desirable. In contrast to entertainment applications, such data is safety-critical and is constraint by hard real-time requirements.

1.4.1. Traffic Shaping and Flow Control

Every AVB Ethernet package is associated with a fixed priority, which is mapped to a specific traffic class. Most commonly, three classes/priorities are used, namely Class A (highest priority), Class B (medium priority) and a legacy class. Class A and B are dedicated to time-sensitive streaming and are subjected to additional traffic shaping. AVB uses VLAN frames. The VLAN tag within a frame specifies the traffic class and priority of the frame.

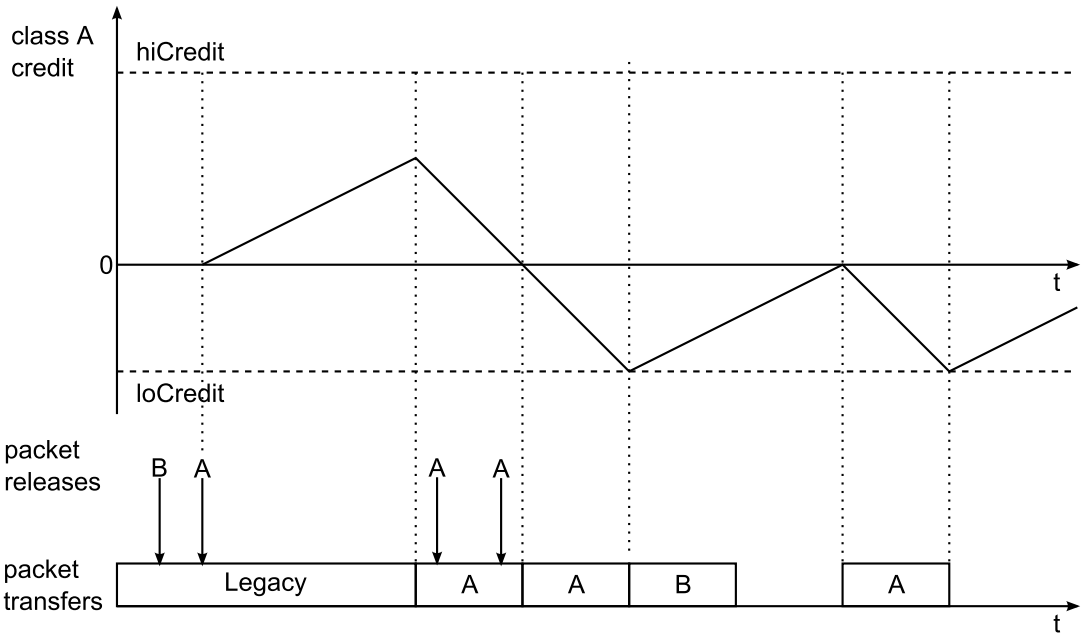


Figure II.5.: Operational principle of the Credit Based Shaper (CBS) algorithm

Core to the real-time capability of AVB Ethernet is the Credit Based Shaper (CBS) algorithm. The CBS uses a credit system to police a class to comply to its allocated bandwidth and to prevent long bursts. This way, interference with other classes can be limited and real-time guarantees can be given. Packages from a certain class are only eligible for transmission if the associated credit is greater or equal to zero. When no transmission is ongoing and no higher priority class is able to transmit, a transfer is initiated and the credit reduced proportional to the transmission time. When packets are buffered, but a transmission is not possible, the credit is increased proportional to its allocated bandwidth.

Figure II.5 depicts a scenario that applies the CBS principle to an AVB class A traffic stream. Initially, a legacy packet is transmitting. During the transmission time, packets from class B and class A are released but cannot be transmitted immediately, because the legacy transmission is non-preemptive. While the packet is waiting for transmission, the credit of class A is increased. When the transmission of the legacy frame has concluded, class A has the highest priority and starts transmitting and its credit is decreased consequently. During the transmission, two further class A frames get released. A second transmission can follow immediately, as the credit is zero when the transmission finishes. After the second transmission of a class A frame, the credit of class A is negative and class B is allowed to transmit. Before the transmission of the final class A frame can be started, the credit has to reach zero.

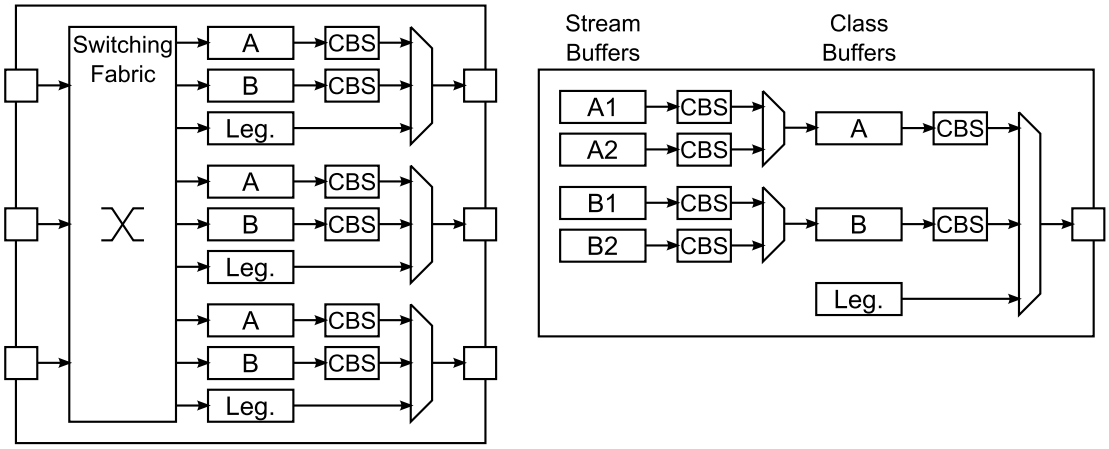


Figure II.6.: Logical architecture of a 3-port AVB Ethernet switch and a talker endpoint with 2 streams in classes A & B and a legacy traffic class.

AVB devices are generally egress buffered and every egress port has separate buffers for each traffic class. AVB differentiates endpoints into talkers and listeners, corresponding to devices which transmit or receive AVB streams. Endpoints can be both talker and listener at the same time. In talker endpoints with multiple streams, a second layer of CBS is used to join multiple streams into a single traffic class. An example architectures of a switch and a talker endpoint are depicted in Figure II.6.

1.4.2. Stream Reservation

At every egress port, information about the allocated bandwidth share for each traffic class is required. This information is communicated through a dynamic stream reservation protocol.

A reservation can be initiated through a *talker advertise*. A talker endpoint sends out a specific Ethernet packet that carries the stream ID and the desired bandwidth. When the stream reservation packet arrives at a switch, it checks whether

the required bandwidth is still available. If the reservation is possible, the stream is registered and the packet forwarded. If not, a *talker failed* message is returned to the initiator. If the talker advertise arrives at the listener, all ports within the communication path have sufficient resources and a *listener ready* message is returned. Streams can be deregistered by talkers and listeners.

In automotive systems, most communication requirements are static. For example, it is known at design-time, how many cameras are built into a specific car and what their bandwidth requirements are. Therefore, dynamic reservation might not be necessary in automotive scenarios.

1.4.3. Time Synchronization

Another feature included in AVB is timing synchronization. It is specified in IEEE 802.1AS and mainly composed of two protocols, the Precision Time Protocol (PTP, IEEE 1588) and the best master clock algorithm. The best master clock algorithm selects the clock within a network, which has the highest priority. This clock is called grandmaster clock and is used as master in the PTP. Similar to the stream reservation, the best master clock could be determined at design-time in an automotive system.

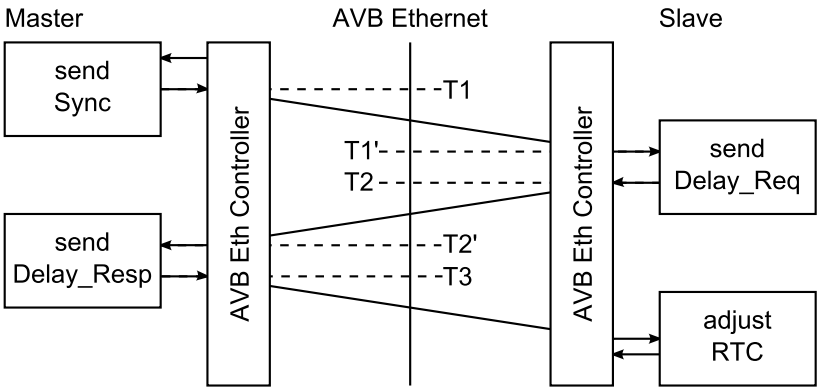


Figure II.7.: Working principle of Precision Time Protocol in AVB

PTP achieves clock synchronization using an exchange of timestamps as depicted in Figure II.7. The master initiates a synchronization sequence by sending a *sync* message, which contains the timestamp T1. Upon reception, the slave records a timestamp of the arrival time of the sync message T1'. The slave incorporates another timestamp T2 into a *delay_req* message. The master measures the arrival in timestamp T2' and responds by sending the timestamps {T1, T1', T2, T2'} in a *delay_resp* message to the slave. The slave can calculate the offset according to

$$offset = \frac{(T1' - T1) - (T2' - T2)}{2}. \tag{II.11}$$

The synchronization is most precise, if the transmission times of sync message and delay_req message across the network are equal. Any difference in these times will directly affect the offset between the two clocks. Also, recording a timestamp is usually an error-prone operation. Several research activities have been directed at quantifying these error sources.

The clock synchronization performance of hardware and software timestamping implementations was evaluated by Mahmood et al. [38]. The authors use a combination of system measurements and simulation. In an experiment, interrupt handling and timestamping delays were measured. These delays follow a distribution similar to a Gaussian distribution and serve as input for a simulation. It was conducted for two systems, which use a Core i7-37000K and a Core 2 T7400 CPU, respectively. The root mean square (RMS) error obtained in the simulations ranged between 409 ns and 1.78 μs.

Kern et al. [39] observed the clock synchronization error in an AVB network for varying temperature conditions. In slowly varying conditions, the error was smaller than 30 ns. Even in experiments with extreme temperature shocks (a change in 80 K over 6 minutes), the synchronization errors were found to be smaller than 200 μs.

Lim et al. [40] simulated an AVB Ethernet-based in-car network to assess synchronization errors. In a daisy-chain topology with seven hops (six switches), errors up to 985 ns were observed. The simulation assumes an ideal implementation of the synchronization protocol in the endpoints. The traffic scenario is not explicitly described within the paper, but seems symmetric (equal interference for both directions during synchronization). Also the simulation time is not given. In worst-case scenarios, the error could be larger and has to be evaluated for every configuration.

1.4.4. Timing Analysis

The majority of research regarding AVB Ethernet focuses on timing analysis. The area is still developing and no universally accepted approach exists. This section gives a brief overview regarding proposed methods, including network calculus, compositional performance analysis, and modular performance analysis.

Network calculus [41] is a method, in which network elements are described by a so called service function. The service curve is a function, which allows to derive the egress flow of network element from an input flow. From this relation, the delay of an element can be calculated. Queck [42] proposed a framework, which models AVB switches using network calculus. It allows to compute worst-case delays for AVB traffic classes in a given configuration. Azua et al. [43] provide a refinement of [42]. They improve formal proofs and incorporate shaping properties into the service curves. This improves timing bounds if there are multiple streams per class.

Another approach is compositional performance analysis (CPA) [44]. CPA subdivides a system into resources, which have ingress and egress events. For AVB Ethernet, this is a port with incoming and outgoing packets. Using busy period

analysis, the egress events can be derived from the ingress events. The approach assumes an initial system behavior (e.g. egress behavior at the talker endpoints) and computes further egress patterns at other resources. In an iterative approach, these egress patterns are used as ingress patterns for neighboring resources in the next iteration until convergence [45]. Axer et al. [46] improved the egress event model by exploiting load bounds of the CBS algorithm.

Reimann et al. [47] propose a method called modular performance analysis. Using busy period analysis, they derive an analysis to compute end-to-end delays for AVB messages, which can be transmitted through multiple frames. Bordoloi et al. [48] provide a formal refinement of previous analyses, focusing only on the busy period analysis of a single switch.

1.4.5. AVB Ethernet Devices

As AVB has only been around for a few years, there is a limited number of hardware devices commercially available. The majority of products exist within AVB's original application domain of audio and video devices. Recently, few automotive specific devices have also been announced.

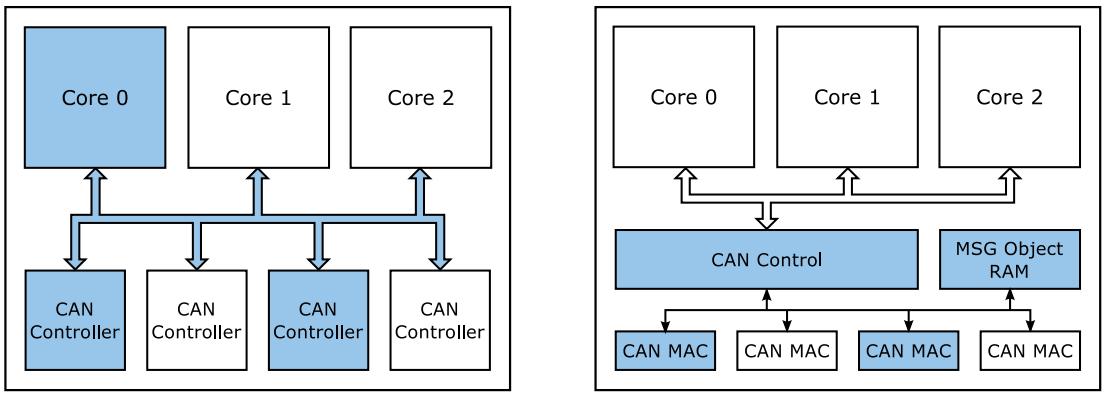
AVB capable switches are produced by NETGEAR, Pathway, and Extreme Networks. In August 2015, an automotive grade switch was also annouced by NXP (SJA1105). Production had not started at the time of writing.

In the field of endpoint devices, a number of AVB Ethernet enabled loudspeakers, signal processors, and amplifiers exist. Intel has integrated AVB capabilities in one of their Ethernet controllers (I210). While this controller is not available on any network cards, it was integrated in to mainboards by ASRock and Supermicro. Xilinx offers AVB extensions to their Tri-Mode Media Access Controller (TEMAC). These extensions include an implementation of the time-synchronization using an embedded MicroBlaze processor and traffic shaping for one traffic class. It is of use, if e.g. a single audio or video source needs to be transmitted over AVB Ethernet from an FPGA. For use in an SoC, more traffic classes and more than one stream should be supported. In Feburary 2015, Freescale introduced an automotive specific ARM-based SoC that features multi-stream AVB Ethernet support (i.MX6SX). However, implementation details are not available.

1.5. Automotive Gateways

In automotive context, gateways are defined as networking components that interface separate physical communication networks. In contrast to the conventional definition, it therefore also applies to interfaces between two networks of the same kind, e.g. two CAN buses. Because current architectures mainly rely on a central gateway to interface the domain-specific networks, this central gateway interfaces multiple protocols at the same time.

There different conceptual approaches for the implementation of gateways. Gateways can be implemented in software only or by use of dedicated hardware components [49]. Generally, the most flexible way is the implementation of gateway functionalities in software, and often it is the only possible one due to a lack in respective hardware. Given that two communication controllers exist in a system, a gateway can be constructed by transferring messages between the two drivers. Despite the flexibility, this approach imposes a high burden of computing load on the overall system, as well as increasing on-chip communication load. Nevertheless, SW-based forwarding is state of the art in current automotive embedded systems, as dedicated hardware support only exists for CAN to CAN communication.



(a) SW Gateway **(b) HW Gateway**

Figure II.8.: Gateway architectures based on SW and HW forwarding. Colored components are involved in gateway functionality between two example CAN networks.

Fig. II.8 shows the two design alternatives using the example of a CAN to CAN gateway. In the SW Gateway, a forwarding routine is executed on a processing core, which is invoked by in an interrupt indicating the reception of a CAN message. Based on a lookup table, this routine may decide to forward the message towards another CAN bus. When such a decision is made, the transmission routines of the respective CAN driver are called, passing the received CAN message. The HW gateway presented in Fig. II.8b is representative of the MultiCAN module used in Infineon AURIX processors. Instead of multiple stand-alone controllers, the CAN controllers are implemented in an integrated fashion. Only a CAN MAC is implemented individually for each physical CAN, while the message buffering and control is shared. This centralized control enables CAN messages to be forwarded efficiently among CAN nodes. CAN message objects can be configured as gateways, where the message is issued for transmission on another CAN bus after its reception. While the forwarding rules are configured by software, no involvement of the host system is necessary during operation.

Much of previous research has concentrated on interfacing operating principles and timing analysis of gateways. In [50], Guoqi et al. present an analytic framework for timing analysis of CAN to CAN gateways. The analysis is valid for a gateway model corresponding to the SW gateway illustrated in Fig. II.8a.

Kim et al. [51] present a gateway mechanism between FlexRay, CAN, and LIN communication protocols. Main objective is to overcome the differences in messaging format, bus arbitration, etc. No real-time analysis or experimental results are provided. Schmidt et al. [52] present a similar CAN to FlexRay gateway. The gateway performs protocol conversion by extracting signals from CAN frames and assembling new FlexRay frames. An analytic approach is proposed to map messages onto the buses under real-time constraints. The overall system is evaluated using experimental analysis.

Zinner et al. [22] present gateway concepts for Ethernet based embedded systems. They contribute gateway concepts for MOST to AVB Ethernet and FlexRay to AVB Ethernet communication. The work focuses on achieving QoS across network boundaries. A prerequisite that was addressed for both networks is time-synchronization across these networks.

CAN to Ethernet gateways have been proposed for legacy [53] and Avionics Full Duplex Switched (AFDX) [54] Ethernet, a real-time capable network technology used in avionics. Both use frame aggregation mechanisms. Gateways for AVB Ethernet have only been proposed for FlexRay and MOST [22].

Kern et al. [53] optimized a CAN to legacy Ethernet gateway to provide low forwarding latencies and small framing overhead. A buffer for CAN messages within the gateway is completely forwarded when the buffer is full, a timeout occurs, or upon arrival of a high priority message at the gateway. The outgoing Ethernet traffic pattern can be bursty, with multiple frames being sent back-to-back. Due to stream traffic shaping, this is not possible in AVB. Frames would queued within the stream buffer, which in consequence makes the scheme equivalent to FIFO scheduling.

Ayed et al. [54] proposed a similar strategy under consideration of AFDX specific characteristics. Communication happens across virtual links (VLs), which have a reserved bandwidth and maximum frame size. The gateway releases Ethernet frames periodically. The maximum frame size is chosen to fit the maximum amount of received CAN payloads in one period, resulting in significant overreservation. Schedulability analysis is used to evaluate the real-time capability.

Related is also the work of Reinhard et al. [55], which discusses the mapping of CAN messages into Ethernet communication channels in virtualized multi-core systems. To enable CAN communication of consolidated ECUs, CAN communication is migrated towards Ethernet channels and CAN messages are consolidated. The approach can be viewed as a virtual gateway within an integrated network.

Integration of multiple data units into one transmission unit has been applied in other areas. IEEE 802.11n wireless LAN uses frame aggregation techniques to reduce average framing overheads and increase overall throughput [56]. A frame

aggregation scheduler has been proposed in [57]. It prioritizes packets based on their expiration time and releases the aggregated packets to minimize the average drop rate. An optimal frame size is determined dynamically based on the bit error rate of the channel.

2. Virtualization

Virtualization is used to abstract isolated, logical resources from hardware resources. E.g. using platform virtualization, multiple virtual machines (VMs) can be hosted on a shared computing platform.

Traditionally, virtualization has mainly been used in server environments and is an important enablement technology for cloud computing [58]. Recently, its isolation properties have sparked interest in embedded systems.

Virtualization can be applied to various resources like computing, I/O, and communication resources. They will briefly be introduced in the subsequent sections.

2.1. Platform Virtualization

Platform virtualization creates multiple virtual instances on top of a computing platform. These instances are called virtual machines (VMs). Their properties and behavior should resemble that of the physical system as close as possible. Because multiple virtual machines can be hosted on a shared hardware platform, this form of virtualization can be used to increase utilization of servers and therefore efficiency.

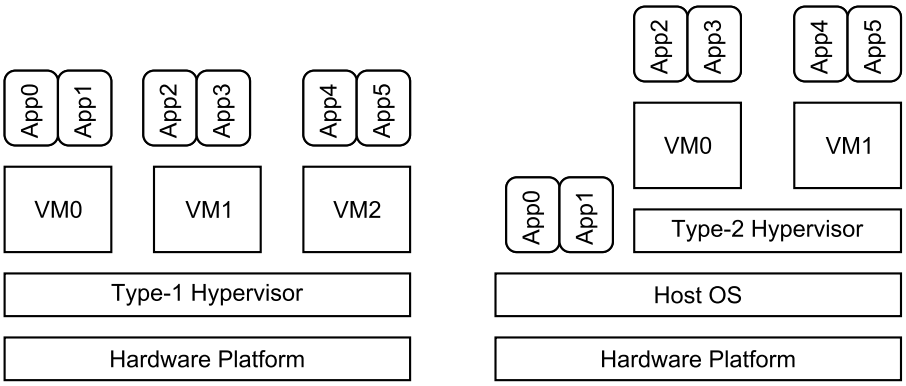
In contrast to physical systems, VMs, can be adjusted at run-time to adapt to varying performance requirements. For example, the number of cores a machine is executed on can be reconfigured, or the VM can be moved to completely different server (live-migration). Of course, VMs can be dynamically created or destroyed. The flexibility and high resource utilization possible with virtualization have made it the technological basis of cloud computing.

VMs are realized through an additional software layer called hypervisor or virtual machine manager (VMM). This privileged layer is responsible for configuring the overall systems as well as the VMs themselves. To avoid conflicts among concurrent VMs, they are not able to use privileged instructions. Such instructions have to be moderated by the hypervisor.

Two kinds of hypervisors can be differentiated [59]:

- **Type-1 (bare-metal hypervisor):** The hypervisor is directly running on the hardware platform and has exclusive access towards privileged resources (see Fig. II.9a). VMs are running as guests on top of the hypervisor. Applications can only be run within these VMs.

- **Type-2 (hosted hypervisor):** The hypervisor is running as a process within an operating system (see Fig. II.9b). Therefore, a host OS is the software layer with the highest privileges, which has to be passed by the hypervisor for many instructions. Applications can be executed either within the host OS or on top of VMs run by the hypervisor.



(a) Type-1 **(b) Type-2**

Figure II.9.: Type-1 and Type-2 hypervisor software architecture

Server environments almost exclusively employ type-1 hypervisors, because they provide the best performance. The additional OS layer under type-2 hypervisors causes overheads, as many instructions have to be emulated.

A complete computing platform consists of many subcomponents like CPUs, memory, I/O devices, and on-chip networks. Additionally, virtualized systems are usually employed in networked environments, where multiple platforms are connected through communication networks. All of these components face specific challenges in virtualization.

2.2. CPU Virtualization

Challenges for virtualization are highly dependent on the CPU type: While x86 CPUs struggle with the complexity of the architecture, simpler, embedded CPUs are limited by their lack of virtualization of awareness or computing power. Yet, the design space of possible solutions is very similar. In the following, CPU virtualization is explained using the example of Intel's x86 CPUs.

In a virtualized system, the hypervisor has to arbitrate the CPU among concurrent VMs. To allow portability of OSs from physical systems into VMs, the CPU behavior should be as close as possible to a non-virtualized system.

x86 CPUs provide four privilege levels, where 0 is the highest privilege that allows access to all critical instructions and registers. In a non-virtualized system, the OS is executed in level 0 whereas applications run in level 3. A hypervisor needs direct hardware access and must therefore run in level 0. Thus, guest OSs are moved to a lower privilege level. While this ensures isolation, as it prevents conflicting critical instructions from concurrent VMs, it also poses several challenges. Mainly, sensitive instructions from the guest OS have different semantics outside of ring 0. Therefore, an unmodified guest OS cannot run on an unmodified, virtualized platform.

Three major approaches to solve this issue exist, all of which have been implemented in commercial products: Binary translation, paravirtualization, and hardware-assisted virtualization.

1. **Binary translation:** Also called full virtualization, this technique runs an unmodified guest OS. However, critical instructions are translated on-the-fly by the hypervisor using trap-and-emulate. Binary translation for x86 virtualization was introduced by VMware in 1998 [60].

2. **Paravirtualization:** This approach modifies the guest OS on source-level, replacing critical instructions by so called hypercalls. In contrast to binary translation, the guest OS is fully aware that it is running on a virtualized CPU. However, an OS has to be ported before it can be run in such a VM. The prime example for paravirtualization is the XEN hypervisor introduced in 2003 [61].

3. **HW-assisted virtualization:** In 2007, Intel introduced Vt-x [62], a set of hardware extensions targeted at improving virtualization performance in x86 systems. CPUs with Vt-x have two forms of operation: VMX root operation and VMX non-root operation. All privilege levels are available in both operation forms. VM exit and VM entry operations are introduced to switch between these operation forms. Sensitive instructions within non-root operation trigger a VM exit and allow the hypervisor to resolve these instructions.

The presented approaches differ in performance and maintainability. The ability to run unmodified guest OSs is desirable, as it minimizes intital porting and continuous updating efforts. However, full virtualization with binary translation suffers from performance overheads due to the costly trap-and-emulate operations. Paravirtualization improves performance, but requires each OS to be ported to the specific hypervisor. Hardware-assisted virtualization combines the benefits of both approaches.

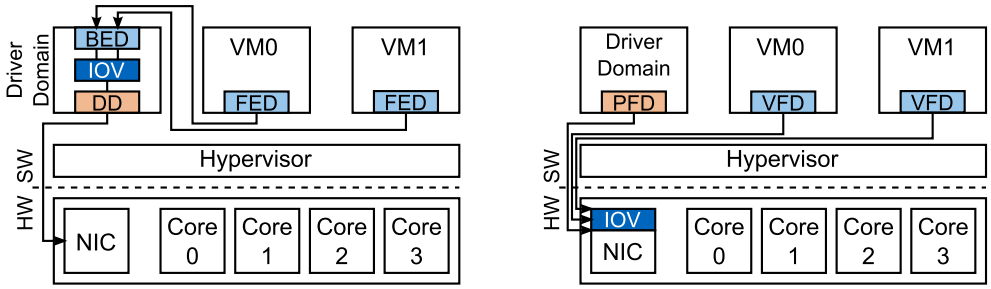
2.3. I/O Virtualization

Performance of network-I/O devices poses a major bottleneck in a virtualized system. Thus, many iterations of academic research and commercial products have been presented in the past to cope with this problem. The problem statement is similar to that of CPU virtualization: multiple tenants require to access a shared hardware resource. To avoid conflicts, a privileged component must moderate request from multiple VMs and forward it to the respective hardware. Due to the high data volume in network-I/O operations, I/O virtualization can contribute significant overheads.

Because the main application of virtualization is server virtualization, I/O virtualization is usually associated with Ethernet network interface controllers (NICs). In the following, key I/O virtualization concepts will be explained.

The initial concept used for I/O virtualization is called device driver **emulation** [63]. VMs can access I/O devices using unmodified device drivers. However, because VMs do not have direct hardware access, I/O requests out of VMs have to be trapped and emulated by the hypervisor. The hypervisor then arbitrates request from and towards concurrent VMs and communicates with the physical I/O controller.

The concept of device driver emulation is similar to full virtualization, as it enables the use of unmodified software components within a virtualized environment. However, it is inefficient, because it requires the hypervisor to emulate privileged instructions. Also, I/O requests have to pass the device driver twice, within the VM and the hypervisor, thus creating unnecessary overheads.



(a) Paravirtualization **(b) SR-IOV**

Figure II.10.: I/O virtualization concepts

The concept of **paravirtualization** can also be applied to I/O virtualization to reduce the overheads experienced using device driver emulation. Paravirtualized network-I/O devices were introduced in XEN [64]. An example architecture for paravirtualized I/O is illustrated in Fig. II.10a. VMs do not use the actual device driver (DD), but rather communicate with a privileged driver domain through a so called front-end driver (FED). The receiving side of this communication is a back-end driver (BED) within the driver domain. The device driver itself is only placed within this domain. Because device drivers are a common source of faults, it is usually preferred to implement this driver domain using a dedicated VM instead within the hypervisor. I/O virtualization (IOV) routines which (de-)multiplex requests are executed within the driver domain.

The Front-end/back-end driver model is more efficient than device driver emulation, as it replaces traps with inter-domain communication. Also, front-end drivers are more lightweight than complete device drivers.

Nevertheless, paravirtualization also suffers from virtualization overheads, which can limit the achievable throughput and cause significant performance degradation.

In [65], Menon et al. showcase a profiling toolkit for XEN. Using this toolkit, they were able to demonstrate throughput degradation of 20%, if driver domain and VM are executed on the different cores, and up to 66% if they share one core. This overhead is caused by context switches between VMs and the driver domain as well as data copy operations.

Recognizing the possibility reducing I/O virtualization overheads through dedicated hardware support, Raj et al. developed the concept of **self-virtualized I/O devices** [66]. Using a network processor board, they offload major I/O virtualization tasks into hardware close to the Ethernet NIC. This way, the driver domain is relieved from data copy operations. Compared to a paravirtualized baseline implementation, their implementation of a self-virtualized NIC is able to double the throughput.

State-of-the-art with respect to I/O virtualization in x86 servers is **Single Root I/O Virtualization** (SR-IOV) [67]. SR-IOV extends the PCIe standard to enable direct sharing of a single I/O device by multiple VMs. The address space of a single I/O device is divided into multiple partitions. One of these partitions is reserved for privileged management operations. The corresponding PCIe function is called physical function (PF). The remaining partitions are used to create virtual functions (VFs). These functions offer abstract data path functions and are directly mapped into the address space of VMs. Therefore, the VMs can directly perform network-I/O operations without involvement of the hypervisor or the driver domain. In an experimental study on SR-IOV, Dong et al. [67] achieved a throughput of 94.8% using a 10 Gbit/s Ethernet NIC.

I/O devices usually use interrupts to signal events to the host system. Within the context of network I/O controllers, the host system is interrupted regularly to notify the reception of messages. This process is associated with a significant overhead in any multi-threaded system, because it provokes one or more context switches within the host CPU. Additional time is consumed by interactions with the interrupt controller of the system. In scenarios with high throughput and small packet sizes, the interrupt rates may be so high that livelocks occur, where the CPU is not able to process interrupts at a sufficient rate [68]. In virtualized environments, this overhead is significantly increased, because the currently running VM has to be preempted when the host is interrupted.

The problem of interrupt-caused livelocks first occurred for UART [69], where an interrupt throttling mechanism had to be implemented in order to reduce the burden of interrupt processing. Today, such mechanisms are mainly researched and implemented for Ethernet based system, where a number of similar approaches aimed at reducing the interrupt rate have been introduced. Mogul and Ramakrishnan [68] proposed a scheme, in which interrupts are disabled temporarily if the interrupt processing cannot keep up with arrival rates. This scheme can be implemented in software only and is thus applicable independent of the network interface hardware. Using dedicated support for interrupt throttling within the NIC can improve the re-

sponsiveness. Druschel et al. [70] presented a modified network interface controller (NIC), which enables coalescing of interrupts within the hardware. An interrupt is only forwarded to the host system, after a configurable amount of interrupts have been requested or after a timeout has occurred. This scheme increases latencies of interrupts but also reduces the associated overhead in burst situations significantly. Nearly all of today's state of the art Ethernet NICs feature such interrupt throttling mechanisms.

In typical Ethernet application scenarios like server environments, throughput is the main optimization goal and latencies are less important. Nevertheless, added latencies caused by interrupt coalescing can also affect throughput. Zec et al. [71] showed that TCP throughput can be reduced by up to more than 90%. An important design goal is therefore a good trade-off between I/O communication latency and processing overhead associated with interrupts.

The increased overhead for interrupt processing in virtualized systems has sparked recent research activities regarding interrupt handling. In a virtualized system, a privileged software component is responsible for interrupt controller interaction. When the CPU is interrupted, VMs are preempted and an interrupt service routine within the hypervisor or driver domain is issued. This routing has to forward interrupts the corresponding VM, thus adding another (virtual) interrupting scheme.

To reduce the overhead of virtual interrupts, coalescing schemes can be implemented as part of the hypervisor. Similarly to coalescing within NICs, interrupts are buffered and forwarded only after a certain amount of buffered interrupts or a timeout. Such schemes have been implemented for multiple state of the art hypervisors including XEN [72] and VMWare ESX [73]. Virtual interrupt coalescing was shown to reduce the CPU load in a virtualized system with 9 VMs by 71% for TCP and 24% in the case of UDP.

The need for virtual interrupt forwarding is not only present in systems with paravirtualized I/O controllers. While solutions like SR-IOV [67] completely bypass the hypervisor with respect to data path operations, they still require hypervisor involvement for interrupt processing. This is due to limitations in x86 systems, which must be configured to receive interrupts either in host or in guest mode exclusively [62]. Solutions for exitless interrupts have been proposed, which receive interrupts exclusively in guest mode and significantly improve interrupt efficiency [74]. This comes at the price that interrupts targeted towards privileged components like hypervisor are received in VMs and have to be forwarded to the hypervisor. Guan et al. [75] tackle the issue by replacing interrupts with a polling scheme to avoid VM exits.

2.4. Network Virtualization

Network virtualization enables multiple logical networks to exist on shared physical network resources. In a virtual network, a subset of the nodes and edges of the physical network form logical connections. It is mainly used in Ethernet and IP-

based networks to improve flexibility, utilization, manageability, and security.

Different approaches towards network virtualization exist. Chowdhury et al. [76] define, e.g.:

1. Virtual Local Area Network (VLAN): IEEE 802.1Q introduces extended Ethernet frames which incorporate a so called VLAN tag. Any frame within the physical LAN is uniquely mapped to a VLAN which is indicated by this tag. All switches within the network forward VLAN frames only on links, which are assigned to the specific VLAN, thus guaranteeing the VLAN frames will not be transmitted outside the virtual network.

2. Virtual Private Networks (VPN): VPNs combine network virtualization with communication encryption. VPNs are mainly used to ensure secure communication e.g. between distributed locations of a company.

3. Overlay network: Overlay networks leverage existing physical networking resources by building another logical network on top. An example would be an IP network built on top of a communication or cable television network.

Most network virtualization technologies focus on the use of TCP/IP and Ethernet. Such applications are usually throughput focused. Latencies are rarely a direct optimization goal. This is strictly in contrast to automotive systems, where hard real-time requirements have to be met. Nevertheless, few QoS focused network virtualization approaches have been proposed, which will be presented here.

SecondNet [77] is a data center network virtualization architecture proposed by Microsoft Research to enable bandwidth guarantees for virtual networks in server environments. They define a virtual data center as a set of VMs with a service level agreement, which specifies computation, storage and also bandwidth requirements for each VM. This agreement allows customers to view the set VMs as an isolated data center. The bandwidth reservation are controlled by a centralized virtual data center manager. Bandwidth reservations are enforced decentral through traffic policing within the hypervisor of each VM.

A similar concept was proposed by Ranghavan et al. [78], who identified the challenge of rate limiting distributed cloud services. They introduce a logically centralized but decentralized executed token bucket policer, which constrains the communication resources for a cloud service running simultaneously at multiple locations. They face a trade-off between responsiveness and accuracy. In an embedded system, such a trade-off should be avoided, as both properties are of high importance.

In addition to classic Ethernet networks, network virtualization has been applied to many other networking technologies, including mobile communication networks like LTE [79] and on-chip interconnects like NoCs [80], [81]. This shows that network virtualization is a concept which is applicable and provides benefits in many application domains and network sizes/topologies.

Yet, no research or commercial activities have focused on virtualization of embedded networks. The main domain specific challenges are real-time requirements, fault-tolerance, and legacy compatibility.

2.5. Virtualization in Real-Time and Embedded Systems

While virtualization is a concept mainly applied in server environments and especially x86 CPUs, there has been a increasing interest to apply virtualization to embedded systems. Embedded systems differ from server environments, as they are constraint by real-time and safety requirements. Virtualization is a technology that can improve safety within a system, because the isolation provided guarantees that even in the case of system crash within one VM, other concurrent VMs are not affected. On the other hand, sharing of hardware leads to unwanted temporal interference effects. Also, the addition of a hypervisor introduces another level of scheduling, thus potentially reducing the responsiveness of operating systems within VMs [82].

Several efforts have been undertaken to improve the real-time behavior of general purpose hypervisors. RT XEN [83] is real-time hypervisor scheduling framework for XEN developed by Xi et al. In [82], Zhang et al. propose a KVM-based system architecture, which they tune to achieve sub-millisecond interrupt latencies.

Due to the expected demand in embedded systems, several domain specific hypervisors have been developed. The RTA-HV [84] by ETAS is a bare-metal hypervisor, which was ported to the Infineon AURIX, a typical automotive micro-controller. It uses a one-to-one mapping between VMs and cores and utilizes the privilege levels provided by the AURIX. XtratuM is an embedded hypervisor developed by the Universitat Politècnica de València [85]. It is available for ARM, x86, and LEON cores. The Windriver hypervisor[2] is an embedded hypervisor designed for small footprint and low latencies. It operates on Intel x86 systems and PowerPC. Sysgo's PikeOS [86] focuses on temporal determinism and safety. It is running on multiple platform architectures including PowerPC, x86, ARM, MIPS, SPARC und SuperH.

Rauchfuss et al. [87] propose an architecture for a virtualized Ethernet NIC in embedded systems. To enable scalability and low latency at the same time, they suggest a caching mechanism for high priority contexts.

Kim et al. [88] presented a SW-based virtualization solution that supports sharing of a CAN controller among multiple VMs in the context of avionics. The design suffers from average added latencies for CAN transmission up to around 200 µs. In [89], a similar approach is taken using the RTA-HV on an Infineon AURIX processor.

In [90], Münch et al. present a virtualized system intended for use in avionics. Using on a Freescale QorIQ P4080 multi-core processor, they were able to implement and benchmark a system using an SR-IOV capable NIC. As the system lacks an IOMMU they use a workaround using non-transparent bridges to provide spatial

[2] `http://www.windriver.com/announces/hypervisor/`

isolation [91]. Additionally, a temporal isolation scheme for DMA transactions of SR-IOV devices is presented [92].

Other research focuses on the use of co-processors in virtualized embedded multi-core processors [93, 94]. They introduce a flexible interface architecture for reconfigurable co-processors in embedded multi-core systems. They use SR-IOV features to enable an architecture where multiple co-processors are integrated on one FPGA and connected to a virtualized host system.

III. I/O Controller Virtualization for CAN

Multi-core processors are a key technology to satisfy the performance requirements of future automotive embedded systems. Using virtualization, isolated computing resources can be provided, which allow a consolidation of today's highly distributed architectures. To enable scalable communication for such virtualized multi-core control units, current network-I/O devices have to be extended.

As introduced in Section II.2.3, network-I/O virtualization has been an active research topic in server environments with both software-based and hardware-assisted solutions proposed. However, previous research has mainly focused on Ethernet NICs and was not conducted in a real-time constrained environment.

Here, an extensive concept for a virtualized CAN controller will be presented (Section III.1). The design focuses on an efficient and scalable implementation while maintaining spatial and temporal isolation among virtual CAN controllers. Because interrupts are associated with high overheads, especially in virtualized systems, the concept of deadline-aware interrupt coalescing is introduced. It reduces the number of necessary interrupts while maintaining real-time capability.

To show the applicability of the virtualized CAN controller in real-time applications, added latencies are evaluated using analytic (Section III.2) and simulative methods (Section III.3). An implementation for a virtualization layer which adds functions to enable multi-tenancy for legacy CAN controllers is presented in Section III.4.1. The processes of design, evaluation, and implementation are presented sequentially, but are in reality interleaved. For example, evaluation methods are used to verify the design, but require implementation details like operating frequencies.

Using the virtualization extensions for CAN controllers, a prototypical virtualized CAN controller is implemented. The prototype is based on a Virtex-7 FPGA board, which features an SR-IOV capable PCIe endpoint. The board is integrated into an Intel Core i7 system, with which experimental latencies are determined.

Throughout this chapter, a number of terms regarding I/O virtualization for CAN controllers are used repeatedly, which are introduced here to improve clarity:

- **Virtualized CAN Controller:** A CAN controller with dedicated extensions for I/O virtualization
- **Virtual CAN Controller:** Abstract version of a CAN controller that allows CAN data path operations. Multiple virtual CAN controllers exist on the

shared hardware of a virtualized CAN controller.

- **Virtual partition:** A virtual partition of a system includes all virtual components of a system, which are associated with each other. For example, a virtual partition can be made up of a VM, virtual memory, and a virtual CAN controller.
- **Virtual interface:** The virtualized CAN controller has a memory interface face, which is subdivided into multiple virtual interfaces. Each virtual interface is associated with a virtualized CAN controller, through which CAN operations can be accessed.

The presentation here is based on a number of publications: Preliminary concepts and results were published in [95, 96]. Extensions for isolation were presented in [97]. A real-time capable interrupt coalescing scheme was published in [98]. Finally, a discussion of HW/SW trade-offs is featured in [99].

1. Design

Within this section, functional and non-functional requirements and optimization objectives are formulated. Based on this, a concept for a layered hardware architecture is introduced, which extends legacy CAN controllers through the addition of a virtualization layer. Additional features are introduced to ensure temporal isolation among virtual CAN controllers as well as a deadline-aware interrupt coalescing mechanism to improve the efficiency of the virtualized CAN controller.

1.1. Requirements and Objectives

In this section, requirements and objectives for the design of the virtualized CAN controller are formulated. They facilitate the general design goal: Create a CAN controller with virtualization extensions, which allows multiple VMs to use this shared hardware as if it was multiple physical CAN controllers.

First, requirements are presented, which constrain the design and describe necessary features. Afterwards, design objective are introduced, which translate into optimization goals.

1.1.1. Requirements

1. **Multi-tenancy:** The virtualized CAN controller must be able to receive and perform transmission and reception requests from multiple VMs.

2. **Resource sharing:** Hardware resources must be shared among all virtual partitions. This is particularly important for the CAN protocol processor. This engine is responsible for handling all transactions on the CAN bus and can be compared to an Ethernet MAC.

3. **Spatial isolation:** Functions and memory contents of a virtual controller must only be accessible by the VM associated with it. This is necessary to avoid manipulations, where e.g. one VM cancels the message transmission of another VM.

4. **Temporal isolation:** The temporal behavior of a virtual controller must be isolated from the behavior of other virtual controllers. Perfect isolation is hard to achieve in an efficient way on a shared platform. Therefore, controllers will be considered temporally isolated, if temporal interference among CAN controllers is limited in a way, that other virtual controllers cannot impact another controllers performance in a negative way.

5. **Freedom from priority inversions:** To allow real-time capable transmission, priority inversions must be avoided. A priority inversion happens, when a message wins CAN bus arbitration while a higher priority message is buffered.

6. **Message ID to VM mapping/filtering:** Each VM receives only a subset of all messages. The virtualized CAN controller should perform a mapping of received CAN messages to VMs. One message can be received by multiple VMs.

7. **Scalable number of messages supported:** Virtualized CAN controllers are expected to be used in domain controller units, which consolidate all electronic functions of a domain. This means that nearly all intra-domain communication is either transmitted or received through the virtualized CAN controller. The message buffers and handling of the virtualized CAN controller must be sufficiently scalable to support such communication scenarios.

8. **Privileged management interface:** Privileged operations (e.g. CAN bus rate configuration, virtual controller configuration) are accessible only through a privileged management interface.

1.1.2. Design Objectives

While the requirements introduced must be fulfilled to ensure the applicability of the virtualized CAN controller in the desired application scenarios, the design objectives introduced here are intended to optimize the design.

1. **Efficiency:** The implementation of the virtualized CAN controller should utilize as little resources as necessary to satisfy the formulated requirements. The main resources targeted here are HW resources used by the implementation. Because FPGA prototyping is used to evaluate the resource utilization, it translates to flip-flop and look up table usage. Another goal is to provide an efficient interfacing scheme between software and hardware. It should introduce as little overhead to the processing system as possible.

2. **Scalability:** The design should be parameterized to support a scalable number of virtual controllers. For any number of virtual controllers, the design is only satisfactory if the same number of comparable stand-alone controllers have higher resource requirements.

3. **Layered architecture:** The architecture of the virtualized CAN controller should be layered in order to allow reusability of the virtualization extensions proposed here. Virtualization extensions should be implemented in a virtualization layer, which can be used to extend various legacy CAN controllers and can be interface with multiple host systems.

To satisfy the specified requirements and design objectives all at the same time is challenging, as some are contradicting to some degree. For example, sharing of resources is necessary to provide an efficient and scalable architecture, but leads to issues with respect to temporal interference.

1.2. Functional Design of the Virtualization Layer

The design of the virtualized CAN controller is structured in three layers as depicted in Fig. III.1. Core to the architecture is a virtualization layer, which implements all functions necessary to allow multi-tenancy towards a regular CAN controller in a virtualized setup. The tasks performed within the virtualization layer equivalent to the ones, which would be located within the hypervisor or privileged driver domain in a paravirtualized setup.

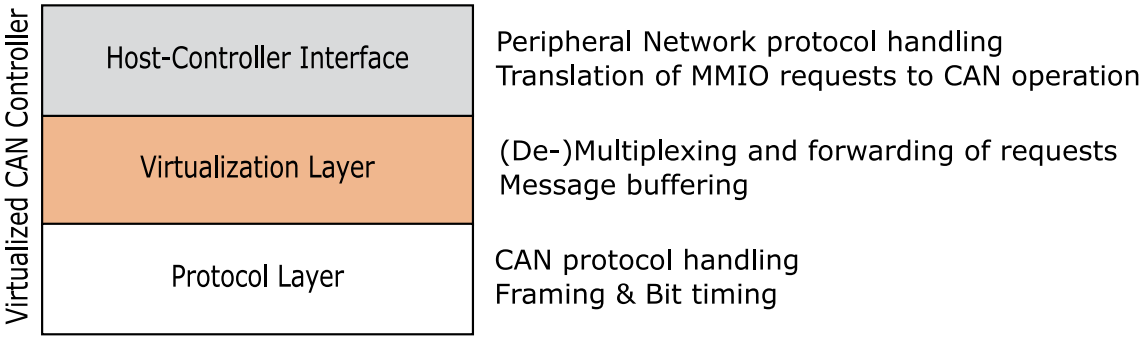


Figure III.1.: Layered architecture of the virtualized CAN controller

The host-controller interface is system specific. It implements the interaction with the peripheral or on-chip interconnect depending on the degree of integration. Applications running within the host system post requests towards this interface as memory-mapped I/O requests.

Finally, the protocol layer implements basic CAN controller functionalities. It directly interfaces the CAN transceiver, and is responsible for the bit level CAN bus interaction. On the side facing the virtualization layer, it must store message transmission requests and translate them into CAN frames.

Fig. III.2 presents the overall architecture and a component level illustration of the virtualization layer. The address space of the virtualized CAN controller is divided into $V + 1$ partitions. One of these partitions is used exclusively for privileged operations. It is called physical function (PF) and provides access to management functionalities. This PF is assigned to a privileged SW component within the system (here the hypervisor). The remaining V partitions are called virtual functions (VFs). They provide abstract data path operations through the virtual CAN controllers. There is a one-to-one mapping between VFs and VMs.

The virtualization layer is divided into a separate transmission (Tx) and reception (Rx) path. In the following, concepts for buffering, arbitration and message filtering will be derived for the respective parts of the virtualization layer.

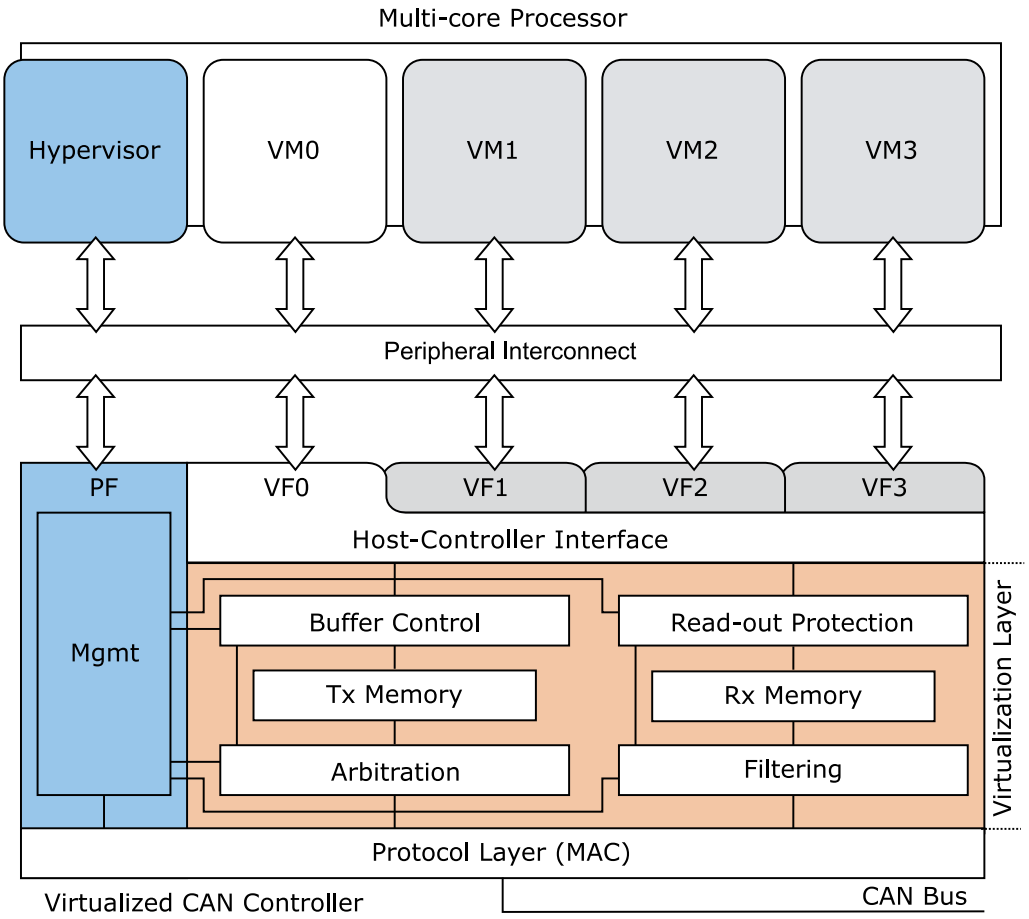


Figure III.2.: HW architecture of the virtualized CAN controller [99]

1.2.1. Transmission Path

According to the requirements, the Tx path must provide scalable and priority inversion-free buffering. Therefore, an appropriate combination of message buffering and selection for transmission has to be found. The CAN bus arbitration follows a strict priority scheme (see Section II.1.3). A collision free arbitration is started on the bus whenever the bus is idle and a node has a message buffered and ready for transmission. The virtualization layer has to buffer messages from multiple VMs and select the highest priority message whenever an arbitration is about to happen. To ensure the highest priority message is actually selected, the internal selection process should occur just before an arbitration on the CAN bus starts.

The tightest timing requirements are present when messages are transmitted back to back on the CAN bus, leaving only a three bit interframe gap between two transmission. Accordingly, the interframe gap corresponds to the time available after reception of a frame until the next arbitration starts and a message should have been selected in the virtualization layer. For a 1 Mbit/s CAN bus, this time equals

3 µs (6 µs for 500 kbit/s). The virtualization layer could operate at frequencies between 20 MHz (typical frequency of a CAN controller) and upto 100 MHz (typical frequency of a PCIe point). Usually, the lower frequency implementation is preferable due to the reduced energy consumption and relaxation of timing constraints during the implementation. In addition to the arbitration, the CAN message also has to be forwarded to the protocol layer. Stand-alone CAN controller often have low throughput interfaces (e.g. 8-bit shared address and data interface for Philips/NXP SJA 1000). To guarantee, the virtualization layer will work as an extension with many CAN controllers, a transfer time of 1 µs should be assumed. For a 20 MHz implementation, this leaves only 20 clock cycles to do a complete arbitration of all messages. This is insufficient in highly consolidated scenarios, where potentially more than 100 CAN messages are sent by a virtualized domain control unit. Therefore, more efficient schemes must be developed.

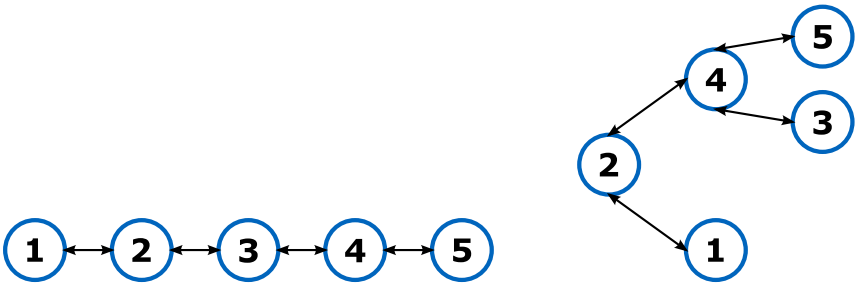
The requirement of spatial isolation implicates that separate message buffers must be available for every virtual controller. An effective arbitration can be achieved, if presorted lists are maintained within each of these buffers. This would significantly reduce the arbitration efforts necessary during the interframe spacing. Instead of comparing all messages buffered, only the highest priority message within each virtual controller must be compared.

To allow a flexible and scalable architecture, the separate message buffers for virtual controllers must not be implemented as separate physical buffers. Instead, they should be separated on a logical level and implemented within shared physical memory. This can either be achieved by statically dividing the memory space of the message into V partitions, or through the use of dynamically managed lists. While dynamic list management is more complex to implement, it increases flexibility, an important characteristic in a virtualized system.

Each virtual controller must have a fixed priority queue size and thus allocation of memory resources. This important, as a minimum queue size is necessary for real-time capability. Only if all messages of a partition can be buffered at the same time, it is ensured that no priority inversions can occur. Therefore, the dimensioning of resources should be performed by the PF.

Multiple options exist for the implementation of a priority queue. The two main possibilities are a sorted list and a binary tree and are depicted in Fig. III.3. A double-chained sorted list (Fig. III.3a) is the simplest possible implementation. Messages are sorted with respect to their priority. Each message object contains a pointer towards its predecessor and successor. Upon a message insertion request, the list has to be iterated to find the appropriate location.

A binary tree (Fig. III.3b) is another possible implementation of a priority queue. Each node can have a left and a right child, where the left child is of higher priority and the right child is of lower or equal priority. Each node has a pointer towards its parent and to both children. Upon an insertion request, the tree is iterated starting with the root node. Messages are always inserted as leaf nodes, meaning they have



(a) Sorted List (b) Binary Tree

Figure III.3.: Possible implementations of a priority queue

no children after they are appended to the tree.

The benefit of a binary tree compared to a linear list is a reduced complexity of the message insertion process. For a sorted list, the duration to serve an insertion request scales linearly O(n) with the number of buffered messages n. For binary trees, the scaling is logarithmic O($log\ n$). However, in worst-case scenarios, this scaling can degrade to a linear scaling, if messages are inserted in ascending or descending priority order. While methods exist to balance binary trees [100], they require costly reordering within the tree. The implementation complexity this brings does not seem worthwhile in a hardware implementation. The worst-case performance of a binary tree is equal to the sorted list. In concrete applications, the release order of messages might be known or constraint in a way, that allows to predict or exclude certain constellations and thus limit the maximum depth of the binary tree.

To provide an efficient architecture, not only memory but also control logic must be shared. Such control logic is usually implemented in finite state machines (FSMs). The state of the FSM describes the current condition of a virtual controller. This state could contain transmission counters, buffer fill level, pointers towards queue elements and so on. The set of values describing the state of the virtual controller will be called *context*. Within the FSM, the context is stored in registers. As only one virtual CAN controller can be active at a time, it is not necessary to provide a complete set of registers for each one. Instead, the registers can be shared among all virtual controllers with a context switch happening if a new controller gets activated.

The process of a context switch in a hardware FSM is illustrated in Fig. III.4. The upper half depicts a regular FSM, which consist of a combinatorial logic block and sequential logic. Only the sequential logic contains registers. A virtualization extension is able to perform a context switch, i.e. read-out and store all register values in a memory and load register values of a different virtual CAN controller into the FSM's registers. This process allows to share an FSM among multiple virtual partitions without the need to replicate all registers. Thus it provides a scalable and flexible solution for resource sharing.

The logical behavior of the Tx path can be summarized as follows. Messages

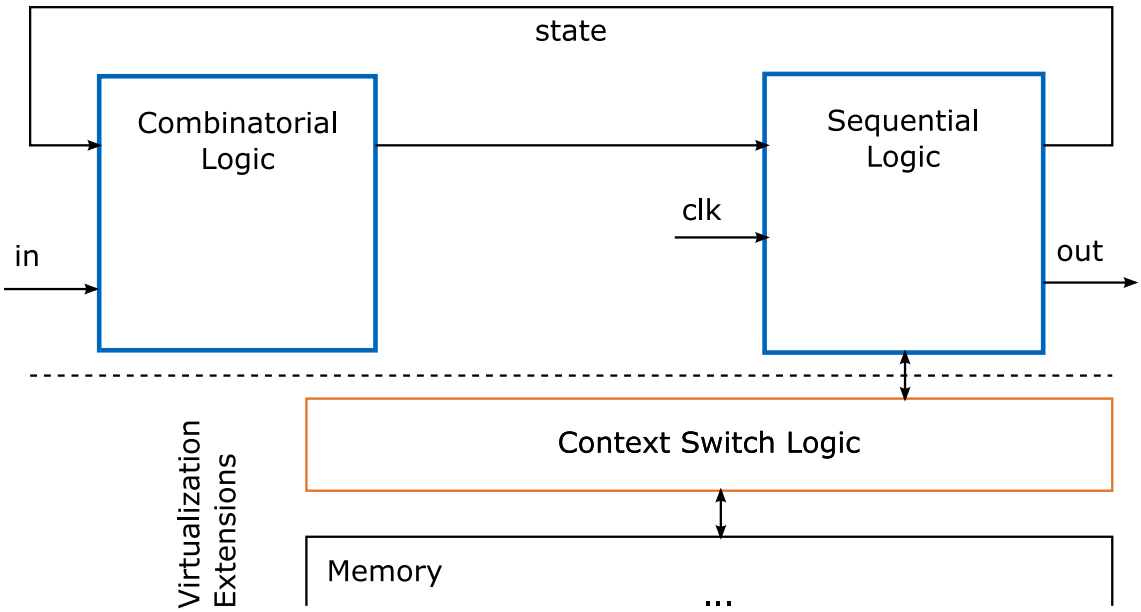


Figure III.4.: Regular finite state machine extended by context switch mechanism

are buffered in a separate priority queue for each virtual controller. These priority queues are represented by binary trees and stored in a shared memory. Just before a new transmission is possible on the CAN bus, the highest priority messages among every virtual controller is determined and forwarded to the protocol layer. A block level architecture is depicted in Fig. III.2. The *Buffer Control* block manages the priority queues of each virtual CAN controller. It must provide means for inserting and removing messages from the buffers. The message buffers are implemented in a shared memory module called *Tx Memory*. Finally, an *Arbitration* module must select the highest priority message buffered whenever a transmission is soon possible and forward the message towards the protocol layer.

1.2.2. Receive Path

The Rx path receives CAN messages after every successful transmission on the CAN bus. Because one VM might require to receive a CAN message sent by another VM, this also includes messages send from the virtualized CAN controller. Out of all these messages, every VM needs to receive a subset. These subsets may overlap. According to the requirements, the mapping of messages towards VMs and the filtering of messages must happen within the hardware. Here, an efficient and scalable way of satisfying this job will be explored.

Efficient buffering means that minimal memory resources should be used. Therefore, all virtual controllers should share one physical memory. Also, no duplicates should exist within the local memory, but stored messages should be accessible by

multiple VMs. This approach will reduce the overall memory consumption when single messages are received within multiple virtual controllers.

CAN controllers implement Rx message storage either as FIFOs or organized as a series of message objects. FIFOs require less implementation effort and are suitable if simple ID filtering is applied. More complex CAN controller implementations tend towards the use of multiple message objects, which contain a filter and a message buffer. This constellation allows more refined message filtering and enables the read of concrete messages. In FIFO buffering, only the message that has been buffered the longest can be directly read.

In the context of a virtualized CAN controller, a message object based buffering scheme within the receive side seems advantageous. It allows message filtering with high precision and message based interrupt configurations. This contributes towards interfacing efficiency, as it reduces the amount of falsely accepted messages and allows to select polling or interrupt based operation for each message object.

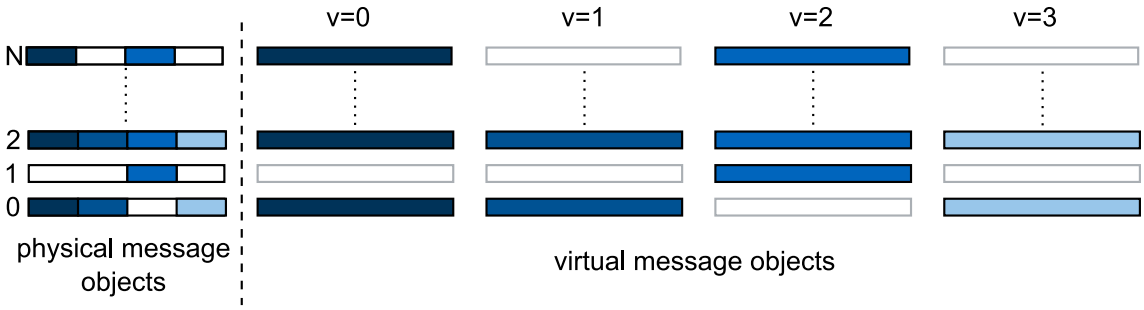


Figure III.5.: Physical and virtual message objects in the virtualized CAN controller. The color coding within the physical objects indicates its association with a virtual CAN controller [97].

Fig. III.5 depicts the concept for message storage within the Rx path. The physical memory contains a list of N message objects. Each of these message objects is configured to store one specific message. Therefore, these message objects will also serve as a list of filters for accepting and discarding received CAN messages. Each message object is associated with one or more virtual controllers. This could be indicated by an additional flag within the message object (color coding in Fig. III.5). From the list of physical message objects, a subset of virtual messages objects for each partition can be derived.

A block level architecture of the Rx path is depicted in Fig. III.2. All messages from the CAN bus are received by the protocol layer and forwarded to the *Filtering* module. Based on the message object configuration, the module decides whether to discard a message or to store it in the *Rx Memory*. At the same time, it may issue an interrupt through the *Host-Controller Interface* if the configuration requires it. Finally, messages can be fetched through the *Read-out Protection* module. For any read operation towards a message object, it checks whether the object belongs to

the requesting VM and then provides the received data.

1.3. Added Blocking in Virtualization Layer

CAN is a real-time capable communication protocol often used in safety-critical automotive applications. Proper real-time analysis is thus key to guarantee reliable operation. The real-time analysis for CAN presented in Section II.1.3.2 assumes an ideal behavior of CAN nodes. Whenever a CAN message is released for transmission on the CAN bus, it is immediately able to participate in the bus arbitration. In an actual implementation, finite latencies within the CAN controller data handling make this behavior impossible. Within a virtualized CAN controller, multiple partitions may access the controller at the same time. The arbitration among these requests also has to be considered.

Therefore, this section presents a modified analysis for CAN, which models finite added latencies within the CAN controller to determine accurate worst-case communication latencies. Head-of-line blocking can occur before a message has been successfully inserted into the priority queues. Afterwards, it is guaranteed to win arbitration if it is the highest priority message present. Thus, the analysis focuses on the insertion of messages into the priority queues of the virtual CAN controllers.

Messages are not sorted with respect to their priority when entering the virtualization layer. Thus, all messages can contribute to blocking at this stage. The worst-case scenario is assumed to start with a critical instant, where all messages are released in the host system at the same time. Insertion times into the priority queue are maximized, if messages are inserted in either increasing or decreasing priority. To maximize head-of-line blocking at the same time, it is assumed here that messages are issued in increasing priority order.

In a virtualized CAN controller, messages from all virtual partitions can contribute to the overall blocking. The total number of messages within a virtual controller v that have a lower priority than message m is assumed to be

$$M_{lp(m),v} = \left|\{k | k \in lp(m) \wedge k \in \mathcal{M}_v\}\right|, \tag{III.1}$$

where $|\cdot|$ is the cardinality of a set and $\mathcal{M}_v$ is the set containing all messages corresponding to virtual controller v.

The time to enqueue a CAN message into the priority buffer is made up of two parts: The time to switch to the respective context of virtual CAN controller v t_{switch} and the time to append the message to the buffer t_{insrt}. The latter is dependent on the current buffer depth. When inserting a message at the n-th level, it is given as

$$t_{insrt}(n) = (4 + n)/f_{clk}. \tag{III.2}$$

The constant value of this equation is implementation specific and a look ahead into the actual delay as seen in the implementation shown in Section III.4.1.

In a worst-case scenario, all lower priority messages have to be enqueued first before message m can be added to priority buffer. While higher priority messages could further delay the insertion of this message, it would not be considered head-of-line blocking. Higher priority messages are anyway supposed to be transmitted first. Thus, the overall blocking experienced can be calculated as the sum of insertion times of all lower priority messages with one context switch happening between each insertion

$$B_{virt,m} = \sum_{v=0}^{V-1} \sum_{k=0}^{M_{lp(m),v}-1} t_{insrt}(k) + t_{switch}. \tag{III.3}$$

Using this added blocking, a real-time analysis can be conducted similar to the one presented for ideal CAN nodes in Section II.1.3.2. For this, a modified blocking has to be considered. In a worst-case scenario, a transmission of a maximum sized low priority message has just started transmission when message m is completely enqueued in the priority buffer. Therefore, modified blocking can be formulated as

$$B_m = \max_{k \in lp(m)} (C_k) + B_{virt,m}. \tag{III.4}$$

This blocking is not a flat added latency with respect to overall communication latency. As it extends the queuing delay w_m given by (II.7), further instances of higher priority may be enqueued during this time, which may overtake message m while being queued. Actual additional delays can only be determined by analyzing concrete traffic scenarios. Another important aspect of the added blocking is its priority dependence. The lowest priority message does not experience any additional blocking by higher priority messages, while the highest priority message experiences maximum blocking. The delay for high priority messages is unfortunate, but can only be reduced through reduction of the insertion time, e.g. by modifying the clock frequency. For infinite clock frequencies, the added blocking $B_{virt,m}$ would be zero and the analysis equal to that of an ideal CAN node.

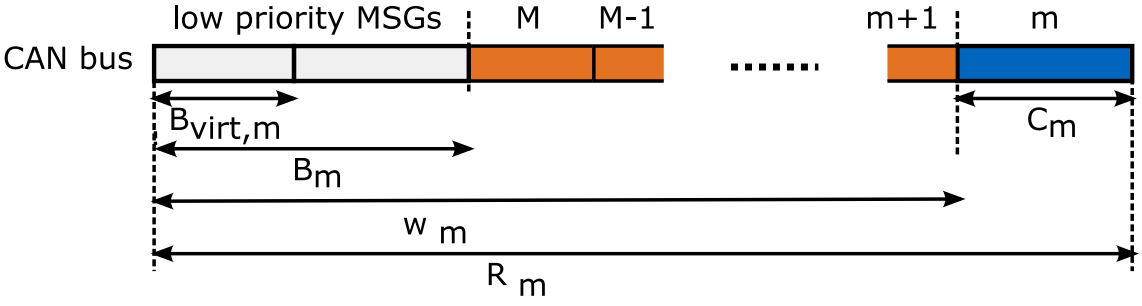


Figure III.6.: Bus occupation in a worst-case scenario for message m [96]

The most important measures of the analysis are summarized in Fig. III.6. At the beginning of the worst-case, the insertion of message m into the priority queue is delayed by $B_{virt,m}$. When the message is inserted, a low priority message has just started transmission, leading to the overall blocking of B_m. Afterwards, higher priority messages are transmitted. This potentially includes more than one instance of each message. The queuing delay w_m extends, until no further high priority message is buffered. Now, message m can be transmitted. The overall latency is given by the worst-case response time (WCRT) R_m.

1.4. Temporal Isolation

The architecture presented above enables multiple VMs to communicate on a CAN bus using a shared virtualized CAN controller. The analysis presented above shows that added latencies from the virtualization layer are bounded. However, this requires each VM to follow a predefined communication pattern. If their use of the virtualized controller does not conform to the predefined scheme, other virtual partitions might be affected in a negative way, which leads to increased latencies and possibly deadline misses.

Temporal isolation of virtual CAN controllers is an important feature in mixed-criticality scenarios. To enable such applications, temporal interference among virtual controllers must be limited so that performance with respect to timing can be guaranteed for each virtual partition independent of the use by others.

Goal of the temporal isolation scheme proposed here is to create isolated virtual CAN controllers that feel as close as possible to separate physical controllers. Temporal interference on the CAN bus itself can occur also with multiple physical CAN controllers, if one CAN node e.g. behaves like a 'babbling idiot', i.e. sending messages in an uncontrolled way and flooding the bus. This problem has been addressed extensively [101, 102] and will be considered solved in this context. This work will only focus on the interference occurring within the shared resources of the virtualized CAN controller.

1.4.1. Virtual I/O Interface Scheduling

To ensure isolation among virtual CAN controllers, a virtual I/O interface scheduling scheme is proposed. CAN controller functionalities can be accessed through memory mapped I/O transactions. While all VMs can issue requests in parallel, e.g. when running concurrently on multiple processor cores, they cannot be served in parallel. As all virtual controllers share common hardware modules like FSMs, the access towards them has to multiplexed. A scheduling among such request should minimize latencies while providing isolated temporal performance.

Peripheral interconnects are designed to serve traffic from multiple devices. Because peripheral devices like 10 Gpbs Ethernet controllers or video encoders require

bandwidth significantly larger than necessary for CAN controller operation, available peripheral communication bandwidth is often multiple orders of magnitude larger. This creates a need to buffer incoming requests. As temporal isolation can only be achieved effectively if it is ensured for all involved components, it is assumed that isolated communication channels are also provided within the on-chip and peripheral interconnect. For PCIe, this is possible e.g. through the use of virtual channels. It is therefore assumed that incoming requests arrive sorted with respect to virtual CAN controllers and are e.g. buffered within the virtual egress buffers of the peripheral interconnect.

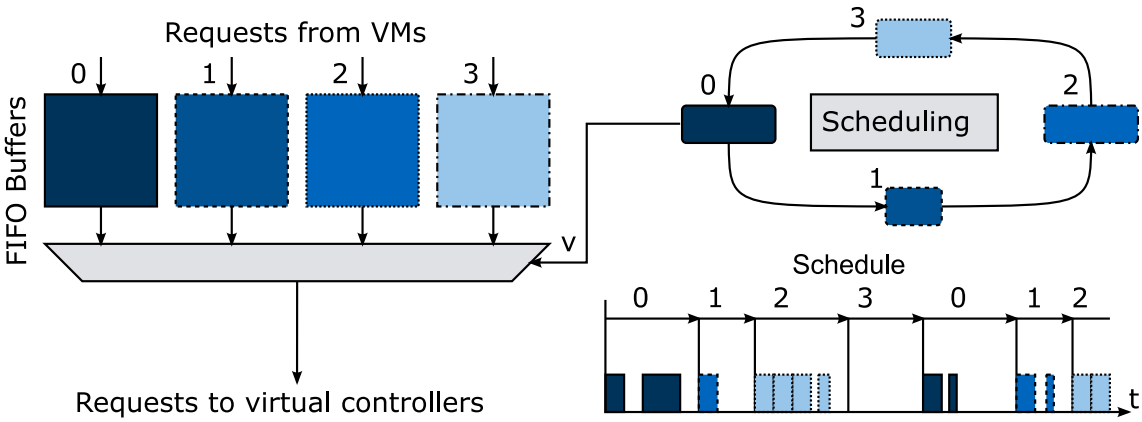


Figure III.7.: Scheduling of virtual interfaces: Requests from VMs are buffered within the host-controller interface and issued to the respective virtual controllers based on a scheduling algorithm [97].

Fig. III.7 shows an illustration of a virtual interface scheduling mechanism. Buffered requests for multiple virtual CAN controllers are multiplexed towards the virtualization layer. The sequence in which requests are forwarded is determined by a scheduling mechanism.

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

1. **Predictability:** The scheduling mechanism must be deterministic in a way, that allows predicting bounds for the virtual interface blocking. This is necessary for real-time analysis to be derived and therefore for the virtual controller to be used in safety-critical applications.

2. **Temporal isolation:** Latencies must be bounded independent of the interface utilization by other virtual partitions. Additionally, temporal interference should be minimized in order to reduce jitter within a virtual partition.

3. **Low-latency**: The scheduling algorithm should minimize latencies experienced at the virtual controller interface. This objective is non-trivial, as the

latencies of all messages and virtual controllers cannot be minimized at the same time, but may have to be traded off. Here, no prioritization should be done among virtual controllers. Latencies of high priority messages should be the primary optimization target. Latencies should be reduced by eliminating unnecessary context switches.

4. **Scheduler complexity:** The scheduler must be simple in order to perform scheduling decisions at high rate and in real-time. Thus, schedulers with complexity O(1) will be considered.

1.4.2. Exploration of Round-Robin Scheduling Mechanisms

Round-robin (RR) is a simple scheduler that schedules resources in a round-based fashion. It guarantees that all sources are served each round. Different versions of RR exist, where weights are used to allow one source to be served more often than once during a round. Also, time-triggered versions exist, where sources are scheduled at static predefined times. The following versions of RR will be evaluated with respect to their ability to satisfy above requirements. Similar RR variations are also used by the PCIe standards for virtual channel and port arbitration [103].

- Simple RR: Requests are scheduled in turns from different buffers. This scheme is repeated in a cyclic manner. If a buffer holds no requests, it is skipped and the next non-empty buffer is scheduled.
- Weighted RR (WRR): During one cycle, multiple requests from each buffer can be served. The number of served requests corresponds to a configurable weight, which can differ for every buffer. It allows accounting for specific communication demands of different applications.
- Time-based RR (TBRR): Time windows of configurable length are assigned to each virtual interface. During each window, requests from the respective virtual partition are scheduled. The scheme is repeated cyclic. If a buffer does not hold requests during its window, it will not be skipped.
- Weighted time-based RR (WTBRR): Time windows have a configurable length, which can differ for each partition. The scheduling mechanism shown in Fig. III.7 corresponds to a WTBRR scheme.

All of the above satisfy the requirement of scheduling complexity. Regular RR schemes make the scheduling decision depending on whether requests are currently buffered. Time-based versions make the scheduling decision based on the current time.

Also, all of the schedulers are predictable. Particularly, time-based versions can be predicted without the need to make worst-case assumptions regarding the use by

other partitions. Request can take a variable duration to be served (e.g. insertion requests take longer if insert deeper into the priority buffer). Therefore, real-time analyses for regular RR would have to assume that message insertions occur at maximum depth. On the other hand, time windows need to be large enough to serve a maximum sized request. Predictability is closely related to the requirement of temporal isolated. Bounded latencies can be achieved with all schedulers and therefore isolation can be achieved. However, time-based versions are superior with respect to temporal isolation as they do not suffer from temporal interference.

Weighted versions of RR scheduling allow scheduling partitions to be tailored to the communication needs of applications. By serving multiple requests of the same virtual partition in a row, the number of necessary context switches can be reduced. This may be used to improve the scheduling with respect to worst-case latencies.

The predictability and temporal isolation properties favor a (weighted) time-based RR implementation. As the non-weighted implementation is a special case of a the weighted version, WTBRR will be analyzed in the following.

1.4.3. Configuration and Added Latencies for WTBRR

A weighted time-based round robin (WTBRR) scheme can be used to provide isolated performance at the interface towards the virtualization layer of the virtualized CAN controller. This allows to provide bounded latencies for operations like a message insertion and thus, for message transmission across the CAN bus. Within this section, two objectives should be achieved: First, a configuration that minimizes latencies according to the requirements formulated, and secondly, a timing analysis that allows to calculate the blocking contributed at the interface towards the virtualization layer.

The scheduler should be configured to minimize latencies while focusing on requests for high priority message transmission. As seen in the analysis without any isolation measure in Section III.1.3, a message can only experience head-of-line blocking by lower priority messages. While the original analysis had to consider lower priority messages from all partitions, here, a message can only be blocked by messages within the same partition due the temporal isolation.

Generally, the worst-case scenario for a message m within virtual controller v can be assumed as follows. Just before the message transmission request for message m was issued, all other messages are released as well (critical instant). At the same time, the time window of the WTBRR scheduler for partition v has just concluded. The head-of-line blocking ends, when message m or a higher priority message has been inserted to the priority queue and is thus ready for CAN bus arbitration. At this time, the worst-case scenario on the CAN bus begins.

The overall added blocking for message m in virtual controller v can thus be summarized as the time needed to serve all lower priority requests for controller v with the resources provided by the scheduler. Depending on the message priority,

different WTBRR configurations are optimal with respect to latencies. For the lowest priority messages, minimal window sizes are optimal as they reduce the time until the first request for each virtual controller will be served. This scheme has the downside that it causes many context switches. Also, small window sizes are harder to utilize, as a high percentage of the window may be unusable if the remaining time is insufficient to finish an insertion request.

Here, it is proposed to dimension windows so that a transmission request for every message of a virtual controller v can be served. This scheme minimizes the amount of context switches during a worst-case. Also, it guarantees minimum latencies for high priority messages at the cost of increased latencies for lower priority messages. A sufficiently large window can be determined as

$$t_{window,v} = t_{switch} + \sum_{k=0}^{M_v-1} t_{insrt}(k). \tag{III.5}$$

Using this window size, the insertion of a message m is guaranteed to be completed during the first complete window corresponding to virtual controller v. Using a WTBRR scheme and window sizes according to (III.5), the added blocking can be calculated as

$$B_{virt,m} = \sum_{w \in \mathcal{V} \setminus v} t_{window,v} + t_{switch} + \sum_{k=0}^{M_{lp(m),v}-1} t_{insrt}(k), \tag{III.6}$$

where $\mathcal{V} = \{v : v \in \mathbb{N}; 0 \leq v < V\}$ describes the set of virtual controllers. The blocking consists of the time needed for time windows corresponding to other virtual controllers to be served, context switches, as well as the time needed to insert lower priority messages. Because the window size is chosen at design-time, no run-time metrics of other virtual controllers affect these latencies. Equation (III.6) directly corresponds to (III.3), which describes added latencies without temporal isolation measures. By replacing this equation in the original analysis, worst-case communication latencies can be computed.

Fig. III.8 shows an example scenario, showcasing blocking during a message insertion. It shows all components of the blocking including time windows of other partitions, context switches and the insertion of lower priority messages. While there are more context switches in the non-isolated case, the overall blocking is smaller nevertheless. This is due to the reservation of time windows in the WTBRR scheme, and the trade-off that has to be made to achieve isolation. It should be noted that the highest priority message would actually have a reduced latency in the isolated scenario due to the reduction of context switches.

The configuration proposed here gives equal priority to all virtual partitions of the virtualized CAN controller. In many application scenarios, low-latency operation might only be required in a subset of all virtual controllers. In this case, latencies

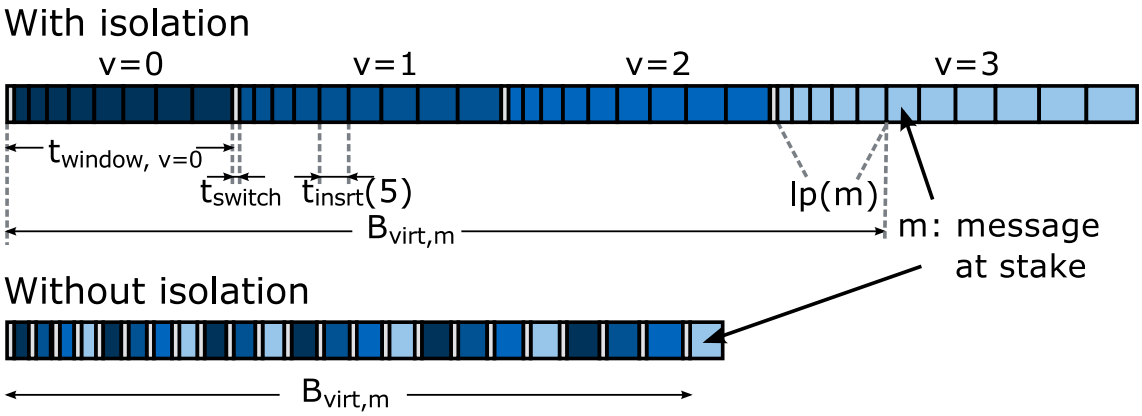


Figure III.8.: Comparison of added latencies experienced by a medium priority message m in virtual controller $v = 3$ using WTBRR isolation and no isolation

for certain virtual controllers could be reduced by minimizing the time windows assigned to other virtual partitions. The configuration of such differentiated service levels is highly application specific. The evaluations in Sections III.2 and III.3 focus on scenarios where all partitions prioritized equally.

1.5. Deadline-Aware Interrupt Coalescing

Interrupts are an essential component for low-latency notification of events like message reception in I/O devices. However, interrupts are also associated with high overheads in the host system, especially in virtualized systems.

Interrupt requests (IRQs) force the processor to suspend running tasks and switch to a privileged mode to run an interrupt service routine (ISR). The time needed to switch into this privileged mode is greater in virtualized systems, as it involves exiting the currently running VM and switching from guest to host mode. Additional issues that occur are cache pollution caused by the frequent execution of ISRs and wake-ups from low power states.

Approaches like interrupt coalescing reduce the overall amount of interrupts by consolidating multiple notifications in one interrupt. However, state of the art solutions for interrupt coalescing (see Section II.2.3) are designed for server conditions and thus lack real-time capability. Here, an interrupt coalescing approach is presented, which incorporates a knowledge of message deadlines and thus enables interrupt coalescing in real-time settings. It is applied to CAN and implemented for the virtualized CAN controller.

In CAN communication, interrupts are an important mechanism to achieve low latency message reception [33]. To reduce the overall load, CAN controllers within every node are programmed to only accept a subset of all received frames depending on the application requirements. The arrival of frames is notified to the host system

through interrupt service requests. In a virtualized system, the currently running VM is exited and the processor switches into a privileged mode to forward the interrupt. The actual ISR is executed within the VM, which reads out the message from the CAN controller and stores it in the main memory. After conclusion of the ISR, the VM is left again so that the hypervisor can complete the process by notifying the interrupt controller. An example of a sequence of such message receptions is illustrated in Fig. III.9. Depending on the hypervisor implementation and platform, additional VM exits may be necessary. The time consumed by context switches and hypervisor routines is considered overhead and can be reduced by consolidation multiple IRQs.

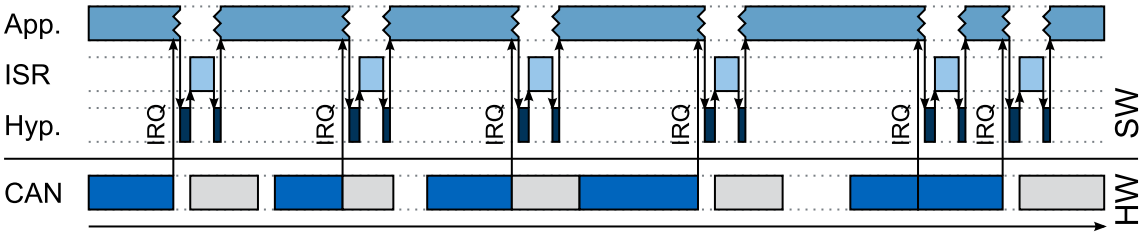


Figure III.9.: CAN interrupt handling in a virtualized environment [98]

CAN is a communication medium often used in real-time settings. That means, a CAN message must be successfully transmitted before a predefined deadline. An interrupt coalescing mechanism for CAN must not interfere with real-time design goals. The fundamental principles of the CAN protocol and timing analysis were presented in Section II.1.3.

CAN follows a non-preemptive strict priority arbitration scheme, which enables the computation of an upper bound for the worst-case response times (WCRT) [32]. For every message m, the WCRT R_m must be smaller or equal to the deadline D_m. Typically, messages on the CAN bus or transmitted cyclic or sporadic with a minimum inter-arrival time T_m.

CAN is used in real-time systems, i.e. message transmissions have to meet predefined deadlines. The worst-case response time (WCRT) of each message m can be calculated analytically and should be smaller than its deadline D_m to guarantee a system's schedulability. Messages are either transmitted in cyclic manner or sporadically with a minimum inter-arrival time T_m. Fig. III.10 shows an exemplary configuration of these values for a single message.

In a real system, average latencies experienced by messages are significantly smaller than the corresponding WCRT. [104]. Fig. III.10 shows a histogram of message latencies in an example configuration along with key metrics. In this scenario, the typical case response time (TCRT) and the WCRT differ by more than one order of magnitude. The difference between a message's deadline and the actual transmission time will be called slack from here on. During the slack, the system can postpone the processing of the message without risking a deadline violation.

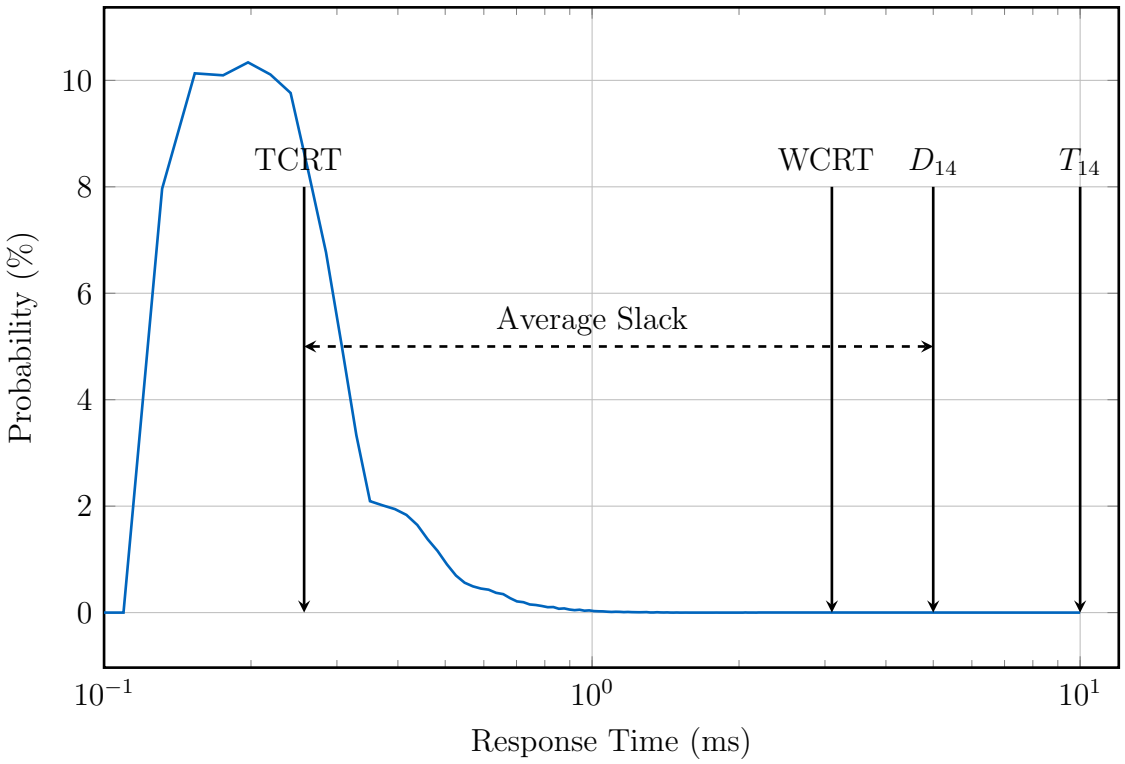


Figure III.10.: Response time distribution of CAN message $m = 14$ with cycle time $T_{14} = 10$ ms at 90 % bus load on 500 kbit/s bus [98]

This means that interrupts can be held back for a long time in cases where the transmission latency is smaller than the WCRT. If multiple interrupt requests are stored at the same time, a consolidated IRQ can be forwarded.

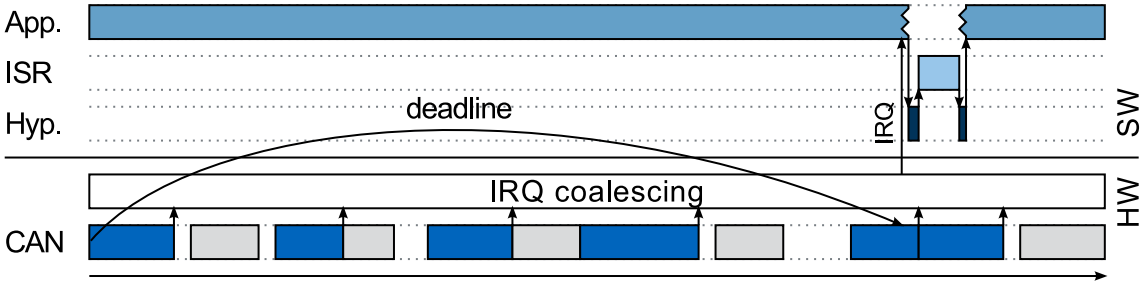


Figure III.11.: Deadline-aware interrupt coalescing for CAN [98]

Fig. III.11 demonstrates an message reception sequence using deadline-aware interrupt coalescing. Received messages trigger an IRQ within the hardware. However, these IRQ are not directly forwarded, but buffered within a hardware extensions layer. This layer maintains knowledge of the closest deadline of currently buffered messages. In the example, three additional messages are received before the dead-

line of the first received message requires the IRQ to be forwarded, thus reducing the number of interrupts by 75%. Such a mechanism is only possible, if information regarding message deadlines is available within the I/O hardware.

To explore possible implementations, an analytic framework will be established first. It is assumed that $t^i_{D,m}$ corresponds to the instant of time, at which an interrupt has to forwarded in order to guarantee that the deadline requirements of the i-th instant of message m are met. The deadline D_m of a message m is the absolute time available between the release of any instance of message m and the point in time, at which the message has to be received.

Therefore, $t^i_{D,m}$ is given by the relation

$$t^i_{D,m} = t^i_{release,m} + D_m, \tag{III.7}$$

where $t^i_{release,m}$ is the release time of the i-th instance of message m. Thus, the release time of each message is essential to have accurate knowledge of when an interrupt has to be forwarded. When the bus is idle at the time of message release, the release time is equivalent to the start of the message transmission. Otherwise, receiving nodes cannot accurately tell the release time, because it is masked by ongoing transmissions.

To enable deadline-aware interrupt coalescing, predictions regarding the release time of messages have to made. In order to not violate real-time constraints, the predictions must be pessimistic, meaning that the estimated release time must be greater or equal the actual release time. For the actual interrupt forwarding decision, the remaining slack of a message instance is deciding. The slack at time t can be derived using the release time as

$$t^i_{slack,m} = t^i_{D,m} - t^i_{Rx,m} = D_m - (t^i_{Rx,m} - t^i_{release,m}). \tag{III.8}$$

Thus, the issue can be formulated either as estimation of the deadline, release time, or the slack. Here, three approaches are introduced, which use estimates to enable deadline-aware interrupt coalescing. They are presented in the order of increasing complexity.

- **Fixed delay:** This approach is similar to state of the art solutions. Interrupts are forwarded after a fixed timeout. To guarantee real-time capability, the timeout has to be smaller or equal the minimal slack which can occur for a given configuration. It can be calculated as the minimal difference between deadline and worst-case response time $\min_{\forall m} D_m - R_m$.

- **Message-based delay:** Using an approach with fixed delay fails to address the diversity with respect to timing properties for different messages. If only one message is at the edge of schedulability, no coalescing is possible even if other messages have high slack. The approximation precision can improved by increasing the granularity. After every message reception, the minimal slack

of said message is computed as $D_m - R_m$. This value is solely derived from static configuration data and represents a lower bound for the actual slack of a message instance given by (III.8). If this value is smaller than the current timeout of the interrupt coalescing, it will be updated.

- **Dynamic deadline estimation:** Both previous approaches rely only on static information. Thus, they fail to exploit dynamic measures of every message instance. Specifically, they cannot leverage the fact that typical latencies are significantly smaller than worst-case latencies. Therefore, a dynamic approach for deadline estimation will be presented in the following.

The actual release time of a CAN message is masked, when the bus is busy during the release. To make a pessimistic of the release time, information regarding the CAN activity can be used. When a message m starts transmission on the CAN bus, it is known that no other CAN messages of the same priority are currently queued if all CAN nodes enqueue message within their CAN controller sorted with respect to priority. This allows to estimate that message m had not been released the last time a message of lower priority won the arbitration process. However, legacy CAN controllers, which use FIFO based queuing still exist, which makes this approach useless. Either way, it is for certain that a message was not yet queued the last time the bus was idle. Therefore, a safe prediction can be made, when the release time of a message is assumed to coincide with the last time the bus was idle.

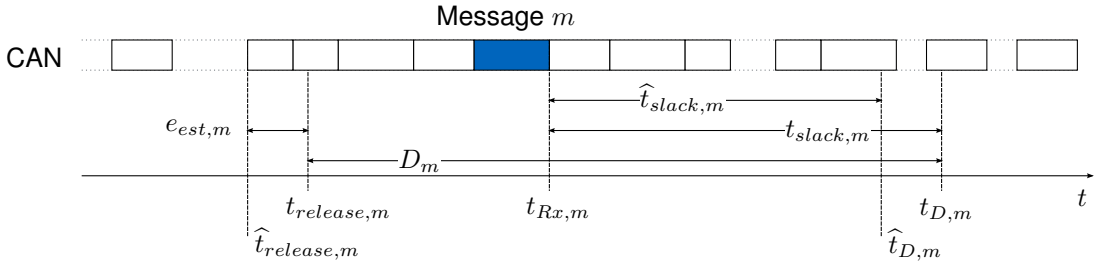


Figure III.12.: Deadline estimation of a message based on the last idle time [98]

An example of the deadline estimation is illustrated in Fig. III.12. Values that are estimated are characterized by the operator $\langle\,\widehat{\cdot}\,\rangle$. The estimation error of the slack using dynamic deadline estimation can be calculated as

$$e^i_{est,m} = \widehat{t}^{\,i}_{slack,m} - t^i_{slack,m} \leq 0. \tag{III.9}$$

A negative error means that the estimated slack is smaller than the actual slack. A non-positive estimation error is a necessary property to guarantee real-time capability, which is given here. The performance of deadline-aware interrupt coalescing in a realistic setting will be evaluated in Section III.3.2 using simulative methods.

2. Analytic Evaluation

Based on the analytic framework presented above, the virtualized CAN controller should be evaluated using realistic transmission patterns. The traffic scenario used here is based on in-car measurements presented in [105]. Message transmissions are assumed to occur cyclic. Thus, their behavior can be described by a cycle time T_m.

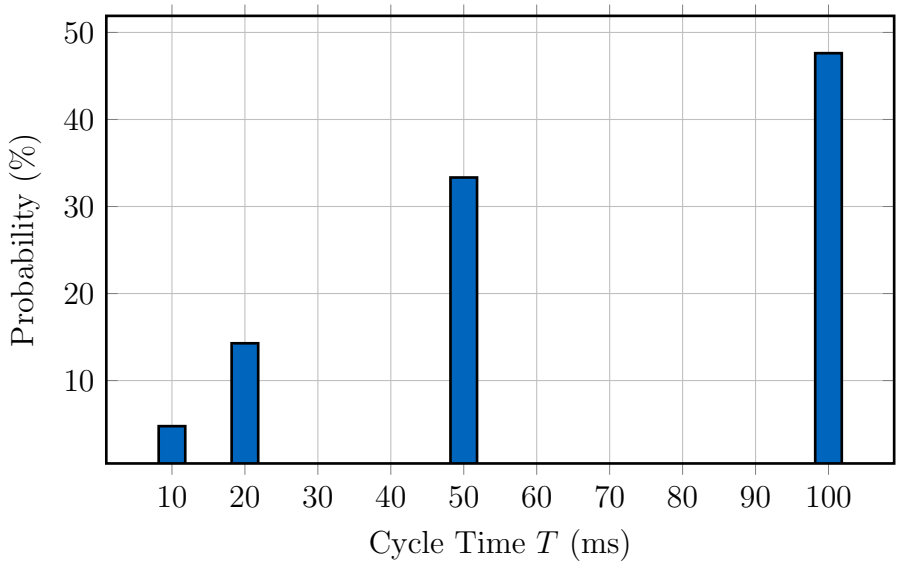


Figure III.13.: Cycle time distribution used for traffic pattern generation throughout CAN-related evaluations

The cycle time distribution used throughout this and following evaluations is depicted in Fig. III.13. It is comparable to distributions used in related work [32, 106]. Message payload sizes are uniformly distributed between 0 and 8 bytes.

The design of the virtualized CAN controller is optimized to provide real-time capable CAN operation for multiple concurrent virtual partitions of a system. The implementation adds blocking at the interface towards the virtualized CAN controller, through which the virtual controllers can be accessed. This blocking was modeled analytically and can be calculated through (III.3). When using the temporal isolation scheme, it is calculated using (III.6). In this evaluation, the added blocking should be quantified using a realistic scenario.

For the scenario used here, a message set with 85% bus load on a 500 kbit/s CAN bus was generated using the cycle time distribution from Fig. III.13. Half of the traffic is randomly assigned to $V = 4$ virtual CAN controllers. The virtualization layer is assumed to operate at 24 Mhz.

Fig. III.14 depicts the results for every virtual controller with and without temporal isolation. Added latencies of upto 30 µs can be seen. The overall largest latencies are actually seen in the non-isolated case, where more context switches can

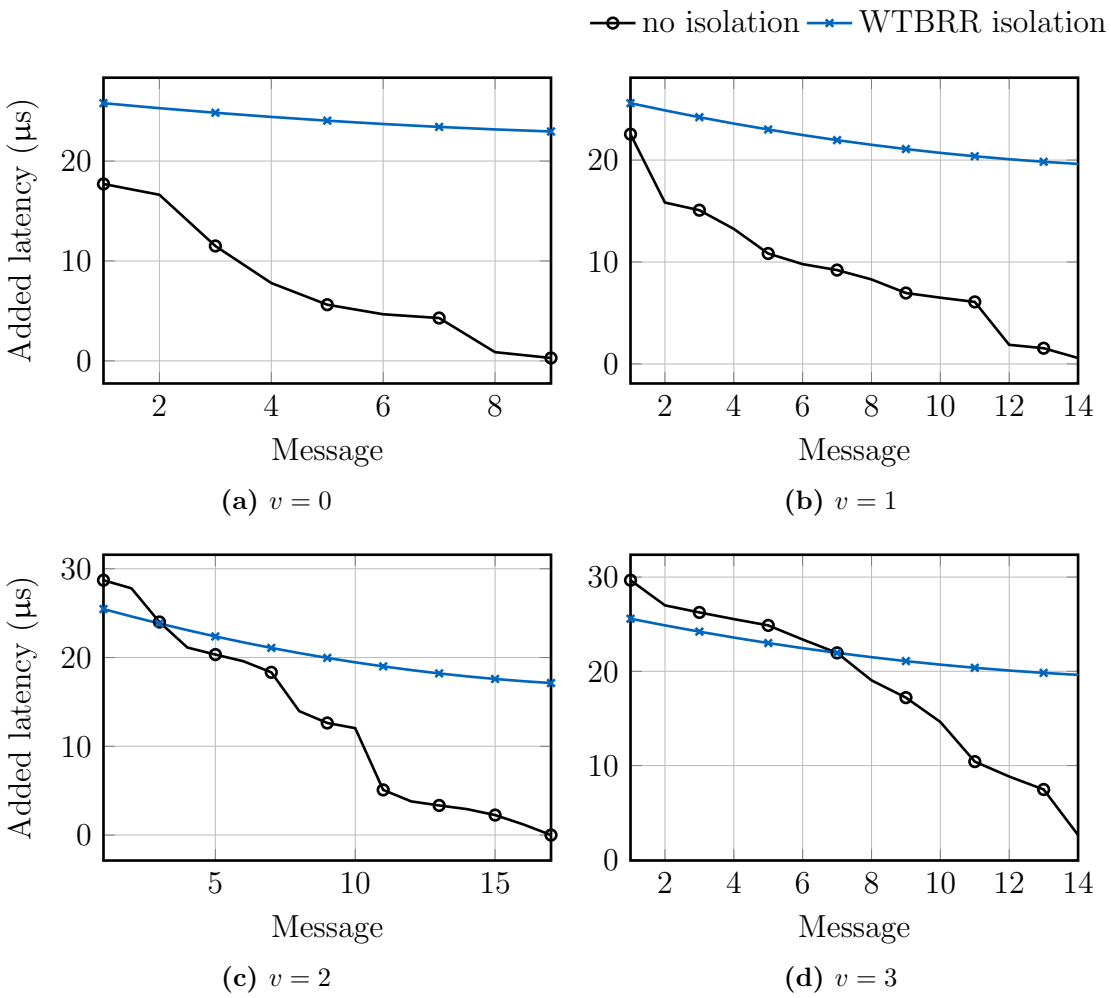


Figure III.14.: Worst-case bound for added latencies experienced at the virtualization layer

occur. The highest priority message in each virtual controller does not necessary have lower latency using the isolation scheme. It is only the case, if these messages are globally high priority as seen for $v = 2$ and $v = 3$.

Generally, all latencies are decreasing along with message priority. Without isolation measures, the lowest priority can hardly experience head-of-line blocking. With activated temporal isolation, any message in a specific virtual controller can be blocked by all other virtual controllers independent of the message priority, because there are statically assigned scheduling windows. The largest partition (here $v = 2$) has the lowest blocking, as the combined window sizes of all other virtual controllers is minimal. Also, with temporal isolation, low priority messages experience smaller blocking, as they cannot suffer head-of-line blocking by the higher priority messages in the same virtual controller. However, the latency decrease for

low priority messages is less steep. Specifically, the slope using isolation is around one forth of the slope without isolation. This corresponds to the total number of virtual CAN controllers $V = 4$.

The evaluation above is limited to a single setting involving the fixed number of $V = 4$ virtual CAN controllers. To evaluate the scalability of the approach, a second scenario is considered. Here, the same traffic configuration is divided in to 16 equal parts. Added latencies are evaluated for increasing degrees of consolidation, where an increasing amount of partitions are integrated in the virtualized CAN controller.

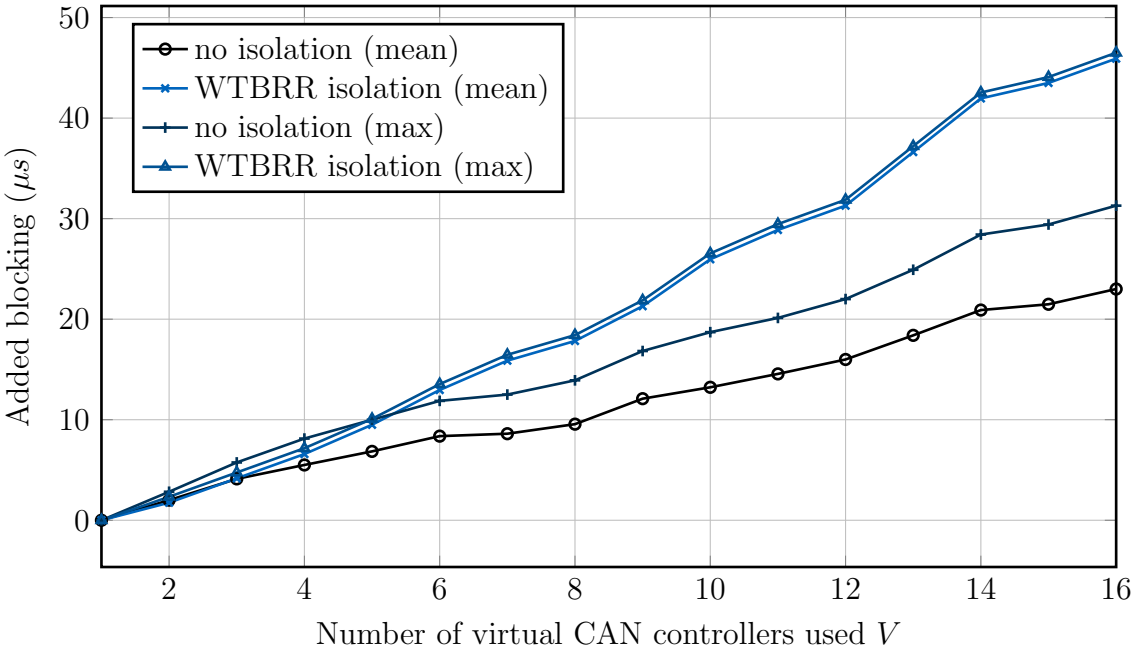


Figure III.15.: Scaling of added latencies due to virtualization extensions

Fig. III.15 shows added latencies as a function of the number of consolidated partitions. There is a linear trend for increasing latencies with the number of virtual controllers V. It is due to the linearly increasing amount of messages, which have to share the interface towards the virtualized CAN controller.

The added latencies for high priority messages are similar with or without temporal isolation. This was an optimization goal specified in the design requirements. The price necessary for the added isolation property is an increase in latencies of low priority messages. Nevertheless, latencies did not exceed 50 µs in the analysis conducted here even in highly consolidated scenarios. This latency is smaller than a minimum CAN frame transmission time on a 500 kbit/s bus (96 µs). Also, it is more than two orders of magnitude smaller than the smallest cycle time used in the scenario. Often, these cycles times equal the message's deadline (implicit deadlines).

The latency bounds without isolation can only hold true, if every partition behaves as specified in the usage model. In Section III.3.1, the temporal isolation property

will be evaluated using simulation.

3. Simulation

Analytic methods are well suited to provide worst-case latency bounds and are thus an essential tool in real-time systems. However, when measures for typical behavior or statistical distributions are required, simulation can be used to provide such results. For consistency, the simulations use the same traffic distributions as introduced in Fig. III.13 and used in the analytic evaluation.

3.1. Temporal Isolation

Temporal isolation is an important property in shared systems, where the behavior of single tenants cannot be completely specified. This is especially the case in mixed-criticality scenarios, where single partitions are developed to lower safety goals and are thus more prone to failures, or where security concerns vary among partitions. Goal of the temporal isolation mechanism presented in Section III.1.4 is to provide virtual CAN controllers, which behave as close as possible to separate physical controllers. Attacks directed towards the CAN bus itself are out of scope here and assumed to be protected against. However, interference within the virtualized CAN controller can occur, if e.g. one virtual machine issues a high amount of register writes. Such behavior could occur due to a failure if e.g. a code segment is stuck in a loop or due to an attack.

3.1.1. Simulation Scenario

Goal of this simulation is to assess how latencies change when the virtualized CAN controller is exposed to flooding of requests from one VM. The simulation tries to cover a critical instant, i.e. the first instances of all messages are released at the same time. Afterwards, messages are sent cyclic. Cycle times are derived from the distribution given by Fig. III.13. If two message instances are issued at the same time, the lower priority message is released first. Additionally, the VM attached to virtual CAN controller $v = 0$ issues malicious requests towards its partition. The rate at which such request are issued is varied throughout the simulation. The intensity of the attempted denial-of-service attack is quantified by the factor I_{Dos}. It specifies a ratio between the number of valid request, which correspond to the traffic scenario specification, and the amount of malicious requests. The factor I_{Dos} is chosen as 100, 1000, and 10,000. In the scenario, 1010 regular requests are issued towards virtual CAN controller $v = 0$ within 100 ms. The maximum intensity chosen here would lead to a 1.3 Gbit/s rate of malicious requests on a PCIe interconnect(assuming 128 bit Transaction Layer Packets (TLPs) for write requests).

The simulation scenario is triggered from a SystemC model. The virtualization layer was originally also modeled in SystemC and was incrementally replaced by

a Verilog RTL description with increasing confidence in the design. The final implementation is presented in Section III.4.1. It takes four clock cycles to serve each malicious request and the clock frequency of the virtualization layer is kept at 24 MHz. The simulated time is 100 ms, as the scenario would repeat in a cyclic fashion afterwards.

3.1.2. Results

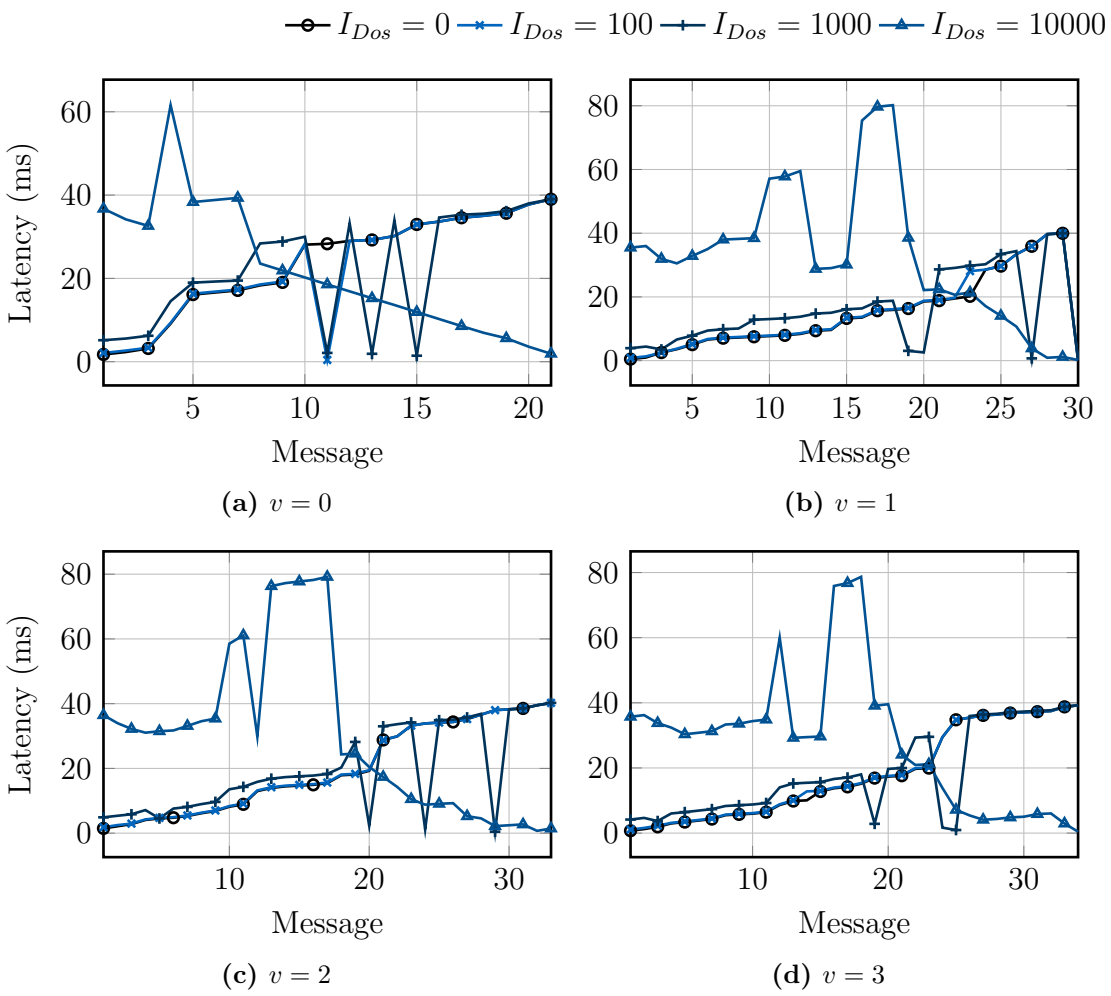


Figure III.16.: Latencies experienced during a flooding of the virtualized CAN controller within partition 0

Fig. III.16 shows results obtained using no temporal isolation scheme. The baseline latencies are already increased for the lowest value of $I_{DoS} = 100$ by around 270 µs. However, only one priority inversion occurred, where the lowest priority message in $v = 1$ and the 11-th highest priority message in $v = 0$ are transmitted

first. For the next higher intensity, high priority messages have a latency incrcease around 3.5 ms caused by multiple priority inversions. These occur, when medium priority messages are already buffered in the priority queue and ready for CAN transmission, while higher priority messages are stuck at the interface towards the virtualization layer.

For the highest intensity, priorities are nearly completely inverted, i.e. lower priority messages have lower latencies than high priority messages. Essentially, the high amount of malicious requests delays message insertions for so long, that messages are mainly transmitted in the order they were issued rather than the appropriate priority ordering. At this point, the latencies of high priority messages are close to 40 µs and thus significantly larger than typical deadlines of such messages.

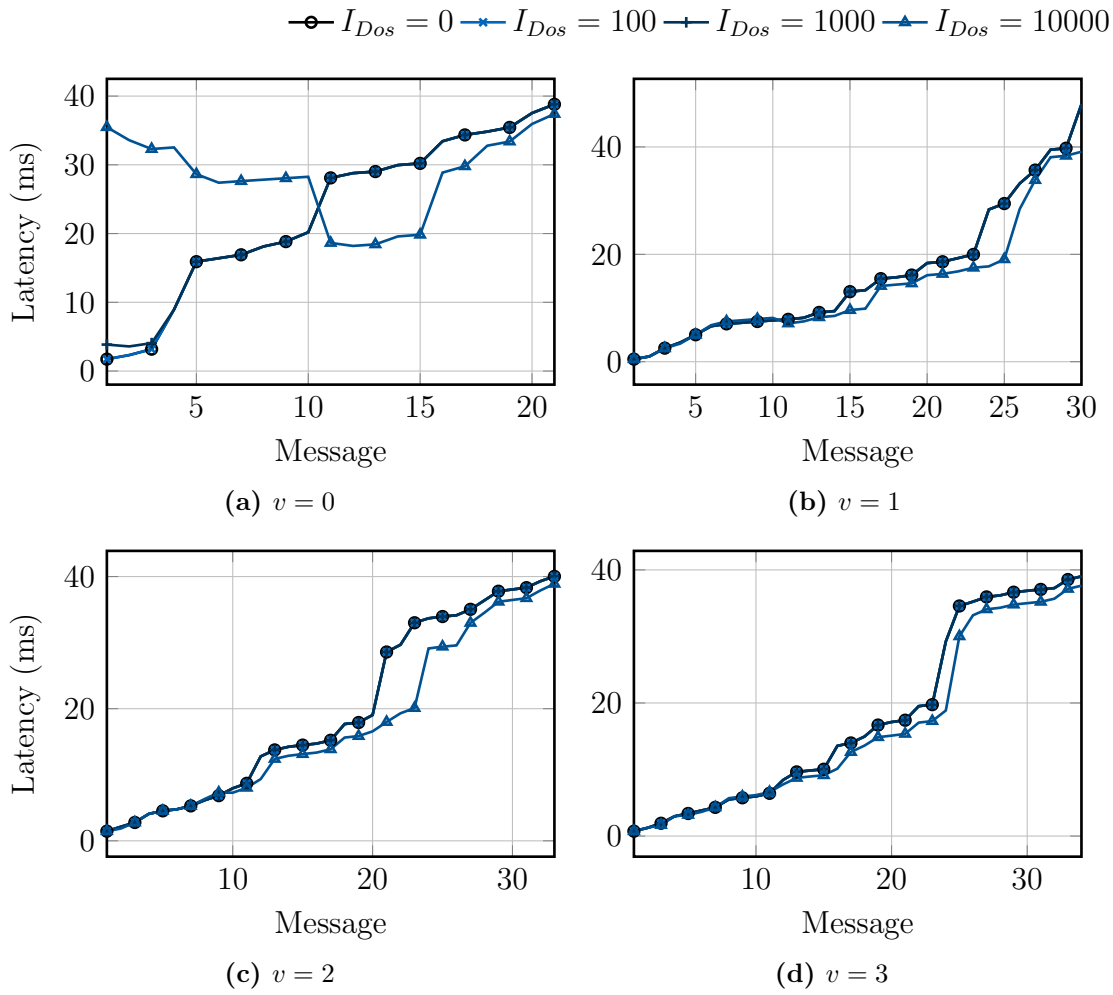


Figure III.17.: Latencies experienced during a flooding of the virtualized CAN controller within partition 0 using temporal isolation through WTBRR

The scenario presented shows how flooding a single virtual controller with write requests can lead to a denial of service in all concurrent partitions. Because all messages are still successfully transmitted, this attack might not be considered a denial of service attack in general purpose server systems. However, service in real-time systems is not only characterized by throughput, but also temporal requirements. The attack at hand leads to a violation of these temporal requirements and thus denies the necessary service.

The scenarios were repeated using a WTBRR temporal isolation scheme. The respective results are presented in Fig. III.17. It is evident that any virtual CAN controllers not $v = 0$ are free from priority inversions. In many cases, latencies are smaller when the attack is run to virtual CAN controller $v = 0$, because its service is delayed, leaving more room for other partitions to transmit.

The temporal isolation also leads to a reduced impact of the attack to the transmissions of the attacking partition itself. This is because all other partitions are free from priority inversions. Only for the highest intensity attack, latencies are heavily increased to a similar order of magnitude as seen in the first scenario.

The simulations results shown here demonstrate the ability of the virtualized CAN controller to provide isolated performance for each virtual controller using a temporal isolation scheme. This enables its use in mixed-criticality scenarios, where partitions with different safety and security requirements are consolidated on a shared platform.

3.2. Deadline-Aware Interrupt Coalescing

Deadline-aware interrupt coalescing (see Section III.1.5) is a concept intended to reduce the number of interrupts imposed on a system while being able to guarantee real-time capability. The concept is generally applicable in real-time communication an was detailed for CAN. It is most efficient in virtualized systems, where interrupt-related overheads are greatest. Here, its ability to reduce interrupts in an automotive CAN setting was evaluated.

3.2.1. Simulation Scenario

Again, the simulation scenario is based on the traffic distribution introduced in Section III.2. A centralized CAN node is evaluated, which is configured to receive 50% of all messages. The mechanism should not be evaluated using a single traffic constellation, but rather using a wide range of scenarios. CAN message sets are randomly generated at predefined bus loads, with the available bandwidth being 500 kbit/s. As deadlines play a central role in this mechanism, multiple more or less tight deadline configurations will be considered. Message deadlines D_m are randomly chosen from the range between 50% and 100% of a messages respective cycle time T_m. The random values used to generate the scenario are uniformly

distributed.

Using an event based simulation of CAN, the number of interrupts forwarded in state of the art systems and with variations of deadline-aware interrupt coalescing was measured. To be able to draw general conclusions independent of single message set configurations, the simulation was repeated 10,000 times for every bus load. This gives a realistic estimate on what can be achieved in the average case.

3.2.2. Results

The central measure used throughout this evaluation is the ability of the mechanism at stake to reduce the total number of interrupts. The IRQ reduction $R_{coal.}$ is defined as a ratio between the number of interrupts forwarded using coalescing IRQ_{coal} and the number without coalsecing IRQ_{SotA}, representing the performance of state of the art systems. It can be calculated as

$$R_{coal.} = \frac{IRQ_{SotA}}{IRQ_{coal.}}. \tag{III.10}$$

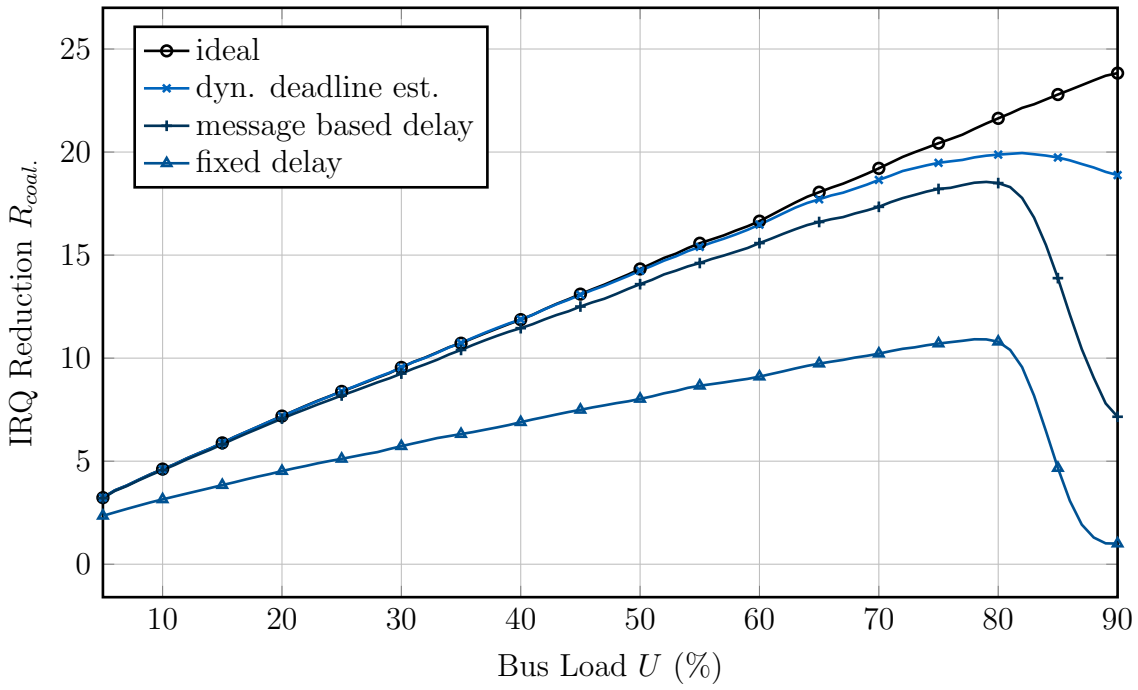


Figure III.18.: Reduction of interrupts using different approximations of deadline-aware interrupt coalescing [98]

Results of the simulation are shown in Fig. III.18. It compares deadline-aware interrupt coalescing using the three deadline approximations proposed to an ideal interrupt coalescing scheme. In ideal interrupt coalescing, all static and dynamic

information is available to make the interrupt forwarding decision. In actual implementations, this behavior is not achievable, because the release time of messages can be masked by ongoing transmissions. If no coalescing were to be used, it would be represented by a horizontal line $R_{coal.} = 1$.

Ideal interrupt coalescing leads to a linear increase in IRQ reduction for increasing bus loads. This is expected, as an increase in the bus load means that more messages arrive in the same time interval and therefore can be coalesced. For increasing bus loads, worst-case latencies increase heavily, which could lead to reduced slack and limit the efficiency of deadline-aware interrupt coalescing. However, even for bus loads around 90% the typical latencies are still low and the IRQ reduction performance is not affected.

Using approximated deadlines, significant degradation from the linear trend is witnessed for high bus loads around 80%. The reasoning for this degradation differs between the static approaches, fixed delay and message-based fixed delay, and the dynamic deadline estimation. For static approaches, high loads mean increasing WCRTs and thus decreasing worst-case slack. This results a smaller time window, in which interrupts can be buffered. The fixed delay approach stops working completely, as soon as one message has slack smaller than a minimum transmission time of a frame. In this case, the timeout to forward buffered interrupts occurs before a second message can arrive.

Increasing the granularity of the fixed delay approach to message-based differentiation leads to an IRQ reduction that is nearly double for most loads. Nevertheless, a performance drop is experienced at a similar load, when single messages reach a static slack close to zero. However, few messages will still have large enough slack so that some interrupts can be forwarded together.

The best performance is achieved using dynamic deadline estimation. Generally, it provides the highest IRQ reduction and maintains close to ideal performance for bus loads smaller 70%. Past this point, a degradation is also experienced, yet it is more graceful than with static versions. Even for high bus loads around 90%, significant interrupt reduction is achieved, where 18 interrupts can be merged into a single one. Here, the degradation is caused by release time masking on the bus. An increased bus load means that the probability of an accurate prediction is decreasing, and the uncertainty increases.

The distribution of deadlines plays a central role in deadline-aware interrupt coalescing and directly influences the possible IRQ reduction. To compare the effects of deadline variations with respect to the proposed approaches, another evaluation is conducted. The tightness of deadline requirements will be measured as the ratio between deadline and cycle time

$$D_{rel} = D_m / T_m. \tag{III.11}$$

The relative deadlines D_{rel} will be kept constant within each evaluated messages set. The ability of a mechanism to perform IRQ reduction at a given level of D_{rel}

will be measured as a weighted IRQ reduction $W(D_{rel})$. It can be computed as

$$W(D_{rel}) = \frac{\sum_{\forall U} U \cdot R_{coal.}(U, D_{rel})}{\sum_{\forall U} U}. \tag{III.12}$$

The weighted IRQ reduction represents a weighted average of $R_{coal.}$, where high bus loads are given more emphasis. This is weighting reflects the importance of high loads in application scenarios as well as the technical difficulty associated with it. This approach allows to analyze the influence of D_{rel} over a wide range of bus loads. It is comparable to a weighted schedulability as introduced by Bastoni et al. [107].

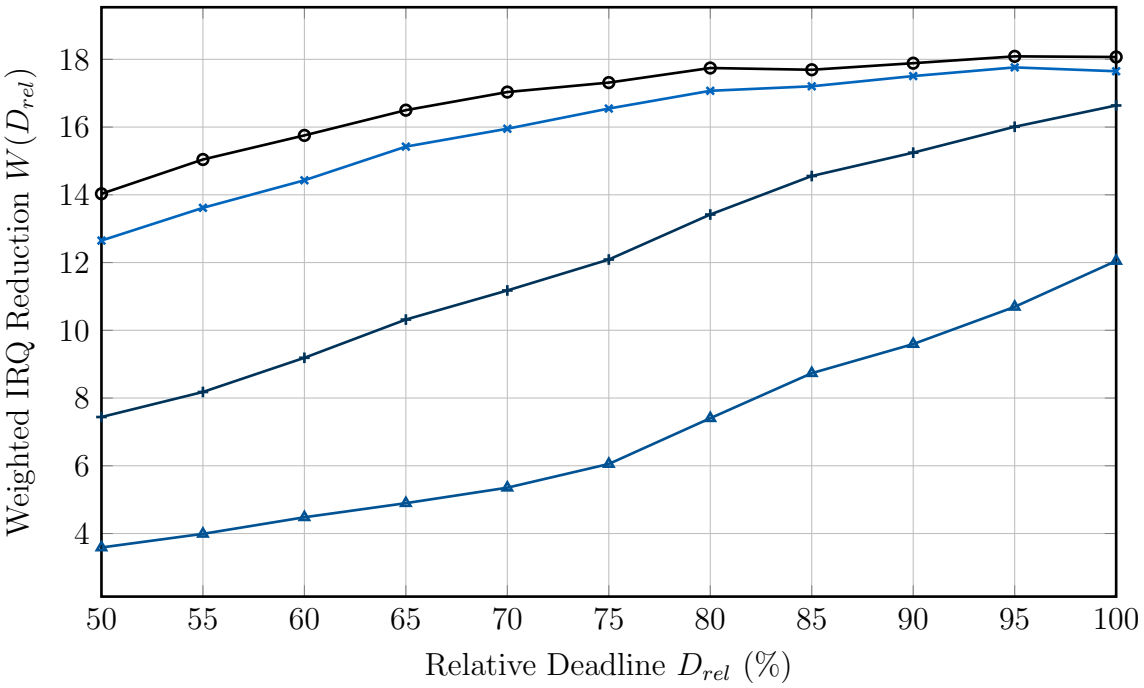


Figure III.19.: Weighted reduction of interrupts depending on latency requirements. A small relative deadline implicates low latency requirements. [98]

Fig. III.19 shows that dynamic deadline estimation outperforms the static implementations for any relative deadline D_{rel}. The difference to static approaches is greater, if deadline requirements are tighter. In the case of implicit deadlines, where the deadlines equals the cycle time, the IRQ reduction achieved using messages-based delays approaches the performance of dynamic deadline estimation.

The discrepancy between ideal deadline-aware interrupt coalescing its approximation using deadline estimation is due to the pessimism in the estimation algorithm. Throughout the previous experiments, the absolute estimation error and the percentage of accurate predictions were recorded.

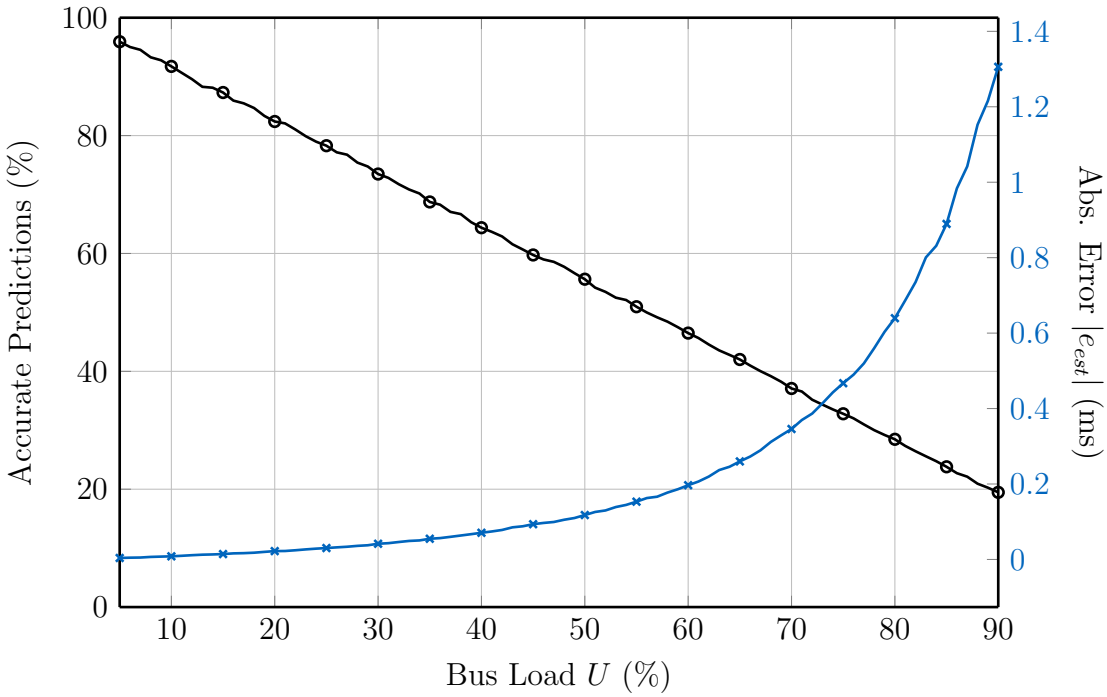


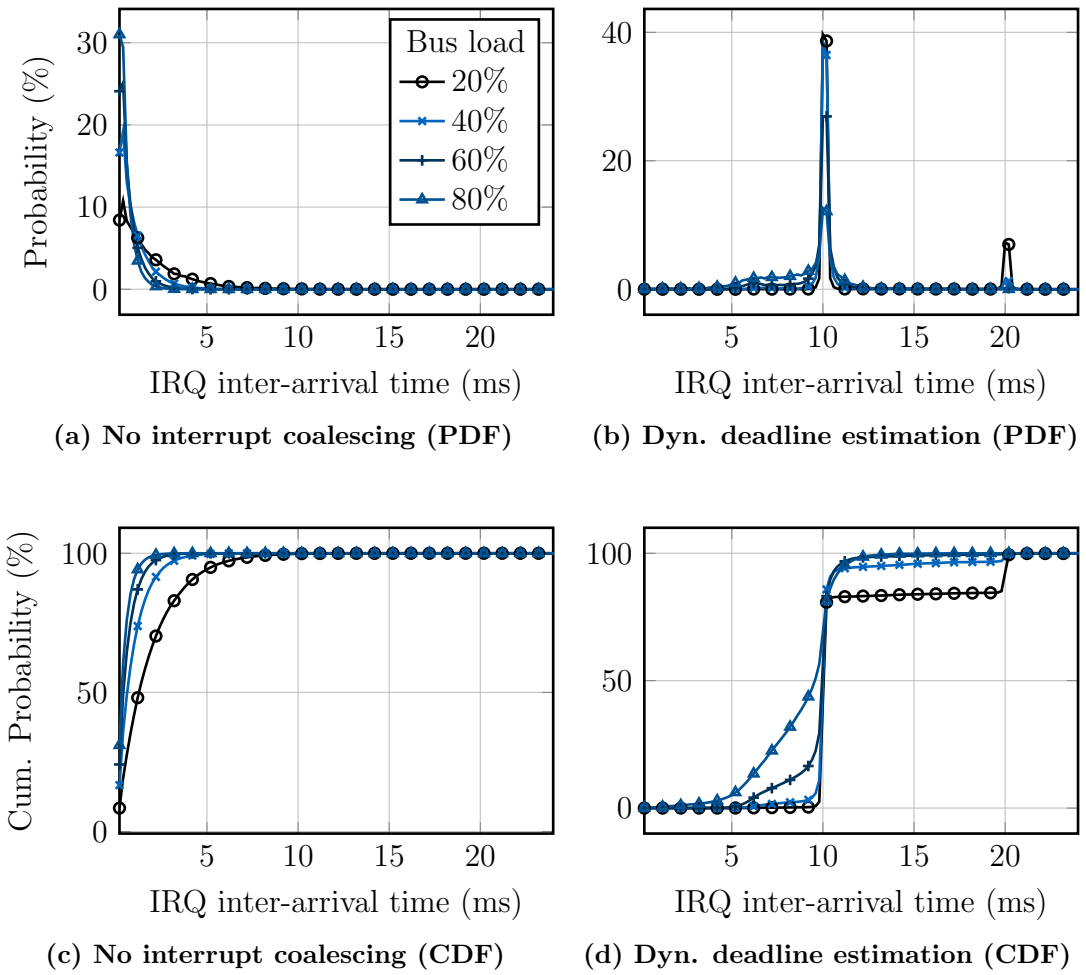
Figure III.20.: Percentage of accurate predictions and average error due to overestimation of deadlines. [98]

Fig. III.20 presents estimation error and prediction ratio as a function of the bus load. The percentage of accurate predictions decreases linearly with the overall bus load. An accurate prediction is only possible, if the bus is idle at the time of a message release. As the probability of the bus being idle is directly related to the bus load, the results is reasonable. Accurate predictions are goal, but not a necessity for deadline-aware interrupt coalescing to perform well.

If the prediction error is pessimistic and sufficiently small, the coalescing can still be performed efficiently. As can be seen in Fig. III.20, the prediction error scales superlinear with the bus load. At 100% bus load, the prediction error would be infinite, as the bus could never be idle. Around 85% load, the error exceeds 1 ms. Because deadlines in CAN are usually greater than 5 ms and range up to 100 ms, this error still leaves room for interrupt coalescing.

Another important figure to estimate the impact of interrupts with respect to the overall system performance is the IRQ inter-arrival time. Between two interrupts, the CPU can execute tasks without being interrupted. Also, if no tasks are to be executed and long inter-arrival times are expected, the CPU can switch to low-power modes.

Here, inter-arrival times of interrupts with and without interrupt coalescing were researched. This evaluation focuses on dynamic deadline estimation as implementation of deadline-aware interrupt coalescing. The inter-arrival were also evaluated



(a) No interrupt coalescing (PDF) **(b) Dyn. deadline estimation (PDF)**

(c) No interrupt coalescing (CDF) **(d) Dyn. deadline estimation (CDF)**

Figure III.21.: IRQ inter-arrival time evaluated for different bus loads: The plots present probability density functions (PDF) and cumulative distribution functions (CDF) with and without interrupt coalescing [98].

at multiple bus loads. Fig. III.21 presents probability density functions (PDF) and cumulative distribution functions (CDF) of the IRQ inter-arrival time $t_{irq2irq}$ at multiple levels of bus utilization.

The PDF of IRQ inter-arrival times without coalescing presented in Fig. III.21a peaks between 400 and 600 µs. Afterwards, probabilities are monotonically decreasing. For increasing bus load, the arrival times are decreasing. Inter-arrival times are up to one order of magnitude smaller than those achieved using deadline-aware interrupt coalescing.

Fig. III.21b demonstrates how the interrupt coalescing affects the IRQ inter-arrival time distribution. The main peak of inter-arrival time is shifted to around 10 ms.

As 10 ms is the smallest cycle time in this example, it can be assumed that cyclic interrupt scenarios emerge, where the release of interrupts is regularly triggered by the same message. In some low load scenarios, message sets with a smallest cycle time of 20 ms were randomly generated, leading to the peak around 20 ms. In few cases, inter-arrival times between 5 ms and 10 ms occur, as these are the smallest possible deadlines.

Fig. III.21c shows the cumulative distribution of IRQ inter-arrival times. It can be seen that there is a 50% probability that an IRQ will be followed by another IRQ with more than in less than 1 ms. For high bus loads of 80%, this point is around 300 µs. By using deadline-aware interrupt coalescing (see Fig. III.21d), this point can be shifted to 9.5 ms even for high bus loads. A minimum IRQ inter-arrival time of 5 ms for 80% bus load can be guaranteed with 95% certainty. This predictable behavior and the long spacing between consecutive IRQs can be used for efficient and uninterrupted task processing (no VM exits, cache pollution etc.) or to utilized deep low power states with low probability of wake ups triggered by interrupts from CAN events.

4. Implementation and Cost Evaluation

This section presents an HDL implementation of the virtualization extensions proposed for CAN. Additionally, a module to perform deadline-aware interrupt coalescing is introduced, which can be integrated with the virtualized controller. Afterwards, the virtualization layer is used to implement a prototype card using a Virtex-7 based FPGA board with PCIe SR-IOV support. The implementation was done in Verilog.

4.1. Virtualized CAN Controller

The implementation of the CAN virtualization extensions will be presented in a top-down fashion on a component basis. The top-level view of the virtualization layer is shown in Fig. III.22. It directly corresponds to the layered concept presented in Fig. III.1.

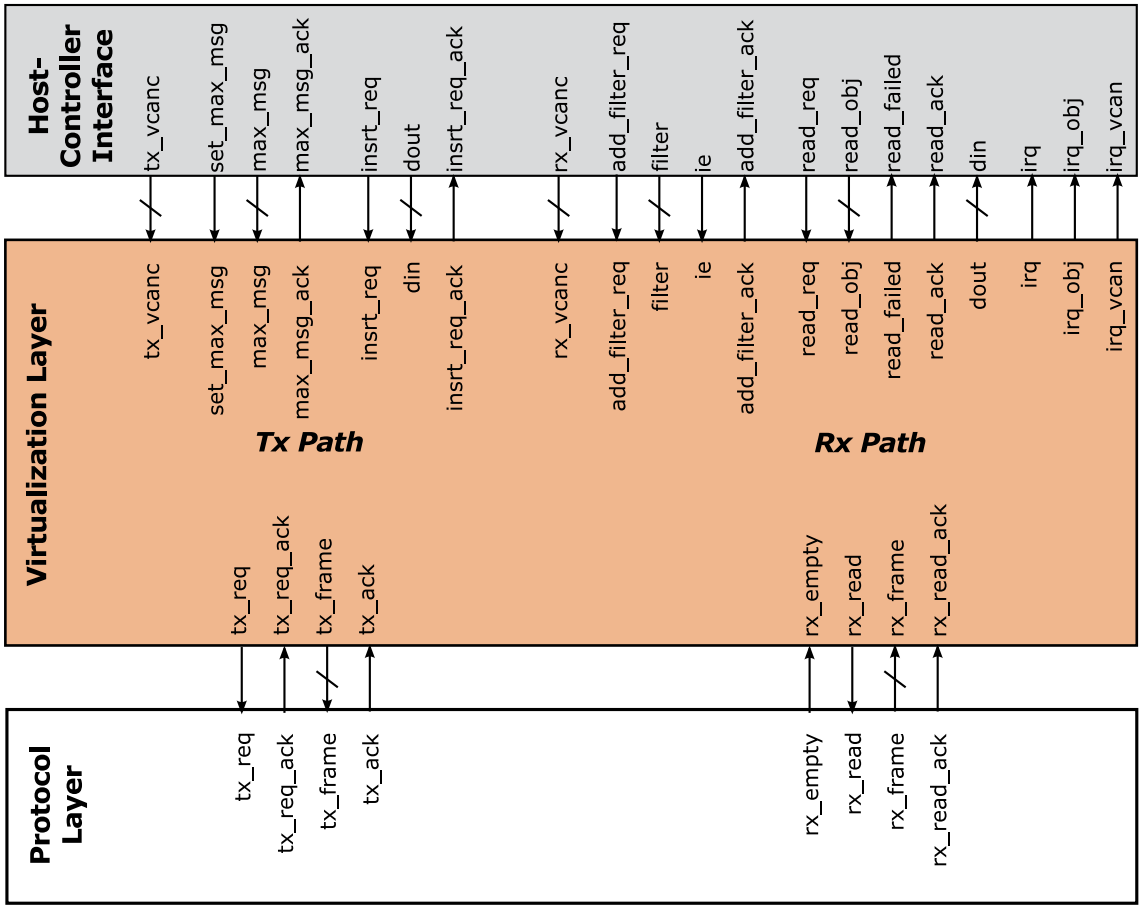


Figure III.22.: Block diagram of the implementation of the architecture layers

The main contribution here is the virtualization layer and it will be the focus of this

presentation. Signals shown are either associated with Tx or Rx path operations. This isolation allows full duplex operation. For each part, there is a signal that selects the current virtual CAN controller (*tx_vcanc* and *rx_vcanc*, respectively). All operations selected through other signals are performed for the indicated virtual controller.

For both Tx and Rx, there are configuration signals (setting maximum number of messages, setting message filters) and data path signals. On the Tx side, messages coming from the host-controller interface are inserted into the virtualization layer queues. Later on, messages are issued to the protocol layer in the form of a transmission request. *insrt_req* and *tx_req* follow the same signaling protocol and could thus be directly connected without the virtualization layer. However, if multiple partitions were to use this component in a shared fashion, they could override each others transmission requests. Resolving such conflicts is job of the virtualization extensions. The Rx side has corresponding signals to read out CAN messages and an additional interrupt interface. Signal interactions will be discussed in detail in context of the lower level modules. Generally, reset and clock signals or omitted to improve clarity of the illustrations.

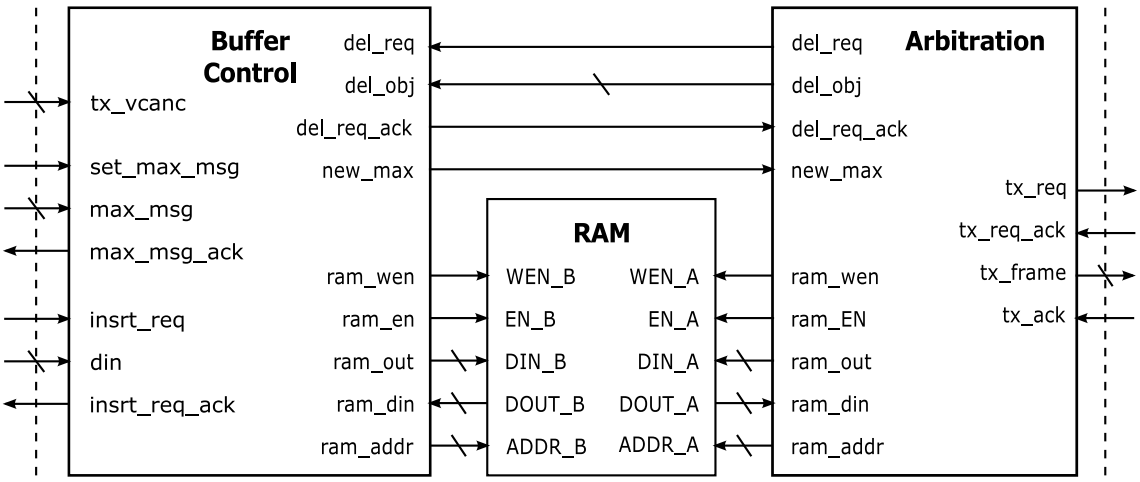


Figure III.23.: Block diagram of the implementation of the Tx path

The Tx path architecture is depicted in Fig. III.23. It is divided into three stages, manifested in the three submodules *Buffer Control*, *RAM*, and *Arbitration*. The *Buffer Control* is responsible for managing priority queues for each virtual CAN controller within the *RAM*. This memory module is implemented as a dual-port Block RAM. The use of dual-port memory allows concurrent and non-blocking message insertions and arbitration. The memory port facing the *Arbitration* module is read-only to avoid write conflicts. This is sufficient for the arbitration process. However, after a successful transmission, the buffer control module has to be notified to remove the corresponding transmitted message. This is done through the

del_req signals. Another signal is driven by the *Buffer Control* to notify the *Arbitration*, when the message currently buffered represents a new maximum priority in the buffer. It can be used to trigger an arbitration in case all buffers were empty.

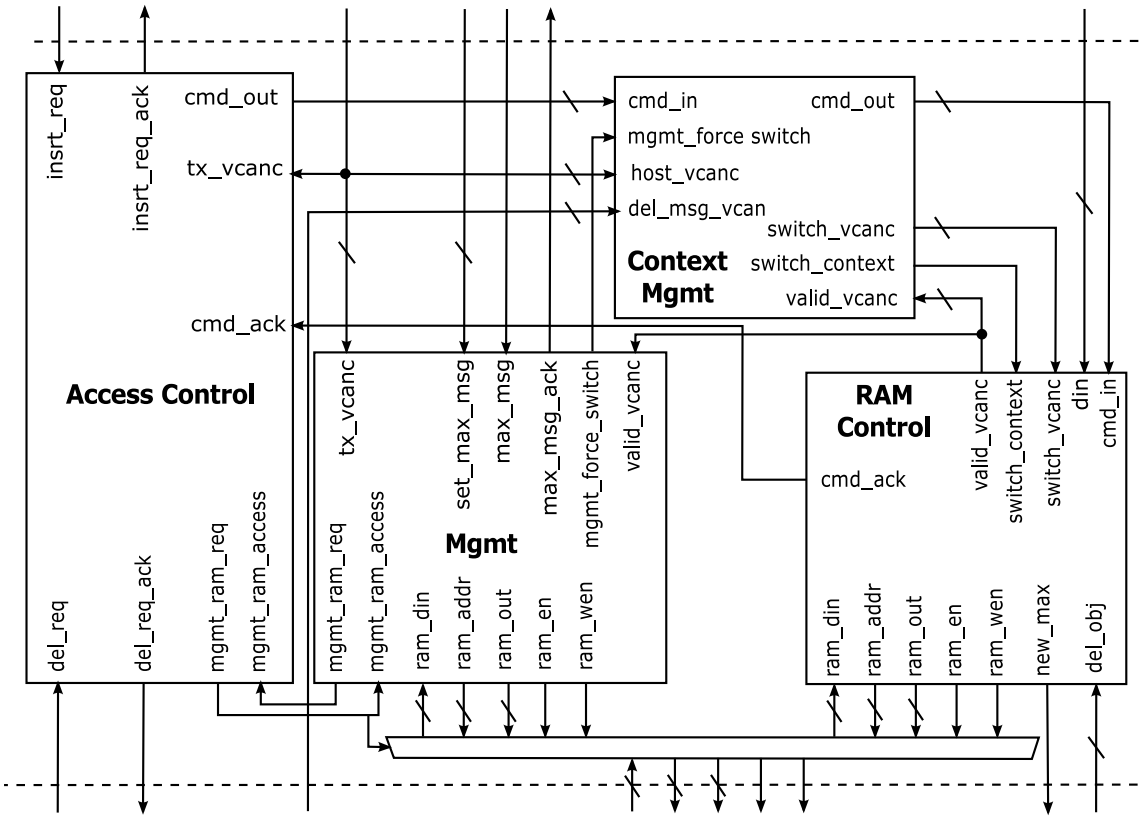


Figure III.24.: Block diagram of the implementation of the Buffer Control module

A detailed block diagram of the *Buffer Control* is depicted in Fig. III.24. It is the most complex module within the virtualization layer and will thus be discussed in detail. Generally, the module has to manage the data buffer structures within the Tx memory and it is the only component with write access to it. Requests to modify the memory contents are issued from the host-controller interface and from the *Arbitration*.

Because requests from multiple sources can lead to memory modifications, the access towards the RAM has to arbitrated. This is the main task of the *Access Control* module. It receives requests for message insertion through the *insrt_req* signal. It is accompanied by the *din* signal, which specifies the data input in the form of a CAN ID, CAN payload length and CAN payload data. The completion of the insertion request is notified through the *insrt_req_ack* signal. Similarly, a message deletion is requested through *del_req*. Deletion requests are directed towards concrete message objects indicated by *del_obj*. This signal specifies a pointer which

can be translated to the memory address of the message. A third source requiring RAM access is the *Mgmt* module. Access is provided in non-preemptive fashion. When multiple sources require access at the same time, management operations are prioritized before deletions and insertions. The *cmd* signal is used to indicate that either a deletion ('1') or insertion ('2') has to be performed by the *RAM Control*. The *Mgmt* module is granted direct RAM access.

When an insertion or deletion request is issued, the *Context Mgmt* module checks whether the correct context is presented by comparing the currently active virtual CAN controller (*valid_vcanc*) to the one indicated by either *host_vcanc* or *del_msg_vcan*. If a context switch is necessary, it is requested through *switch_context*. Additionally, the *Mgmt* module can request a switch through *mgmt_force_switch*.

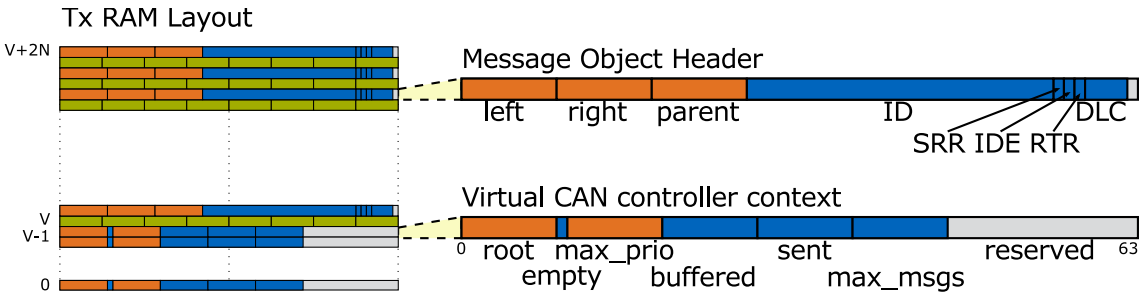


Figure III.25.: Memory layout of the Tx RAM

The functionality of the *RAM Control* module is directly linked to the memory layout, which is visualized in Fig. III.25. The layout is divided into two parts. First, the virtual CAN controller contexts are stored. Each consumes a single RAM line, which is 64 bit wide. The rest of the memory is used to store message objects. Within the figure, orange fields specify message object pointers. Blue fields contain control information and CAN data is stored in green fields. Each message object requires two RAM lines, with one storing header information and one containing the payload data.

The context of each virtual CAN controller contains the partition-specific information used in the state machines of *RAM Control* and *Arbitration* module. It contains two message object pointers, one to the root of the respective queue and one to maximum priority message buffered. These pointers are only valid if the buffer is not empty. Additionally, the number of buffered and sent messages is stored as well as the maximum amount of buffered messages allowed. Each message object header contains three pointers. It has a pointer to the parent and both children. In the same line, the message ID, CAN control bits and the data length code (DLC) are stored.

When a request for message insertion or deletion is forwarded to the *RAM Control*, the *Context Mgmt* has already made sure that the correct values are loaded into the registers. The memory resources are managed dynamically. A linked list of

unallocated objects is maintained. For an insertion, an object is taken from that list and deleted objects are appended to the list.

A message insertion is performed by parsing the existing priority queue to locate the proper position. If the queue is empty, the inserted message will be used as root of the binary tree implementing the priority queue. Otherwise, the tree will be iterated starting with the root. If the message has higher priority, the iteration continues with the left child. A child pointer towards the root indicates that no child object exists. In that case the message can be appended there. If the inserted message has maximum priority within the buffer, the max_prio pointer is redirected and a flag is raised on the *new_max* signal. Deletion requests are issued directly towards a message object indicated by the *del_obj* pointer. Here, no iteration is required to find the object, yet the tree structure has to be adjusted to compensate for the deletion.

The *Mgmt* module performs privileged operations. After a reset, it instantiates the memory contents of the RAM. Each context is written and configured to contain an empty priority queue with zero messages sent. The values of the pointers are irrelevant at this point, as they will be set when the first message is stored. Additionally, the list of unallocated message objects is instantiated. This involves writing pointers within every object to create the linked list. The payload field are not reset and will first be written, when a message is inserted in the respective object. Also, the maximum buffer size can be configured through the *Mgmt* module by using the *set_max_msg* signal. As this operation directly modifies the context of a virtual CAN controller, a context switch is forced so that the registers within the *RAM Control* are consistent with the memory contents.

The *Arbitration* module (see Fig. III.23) selects the highest priority message within the virtualized CAN controller whenever an arbitration on the CAN bus is possible. This is the case, when either a transmission on the CAN bus just finished (flag in *tx_ack*), or the CAN bus was idle. Consequently, the module will iterate all virtual CAN controllers by reading the ID of the message indicated by the *max_prio* pointer. After comparing all maximum priority messages, the overall highest priority message will be send to the protocol layer by raising the *tx_req* signal. The successful transfer to the protocol layer is acknowledged through *tx_req_ack*. A pointer to the message forwarded to the protocol layer has to be saved in order to request its deletion after successful transmission on the bus indicated by *tx_ack*.

Fig. III.26 presents the architecture of the Rx path of the virtualization layer, which is structured into three submodules. CAN messages from the protocol layer are read out by the *Filtering* module and stored in the *RAM* if they are accepted. The reception of messages is signaled through interrupts, which cause the host system to trigger a read operation towards the *Read-out Protection* module, which guarantees that only reads towards messages associated with the respective virtual CAN controller can occur.

The memory within the Rx path consists of a series of message objects. According

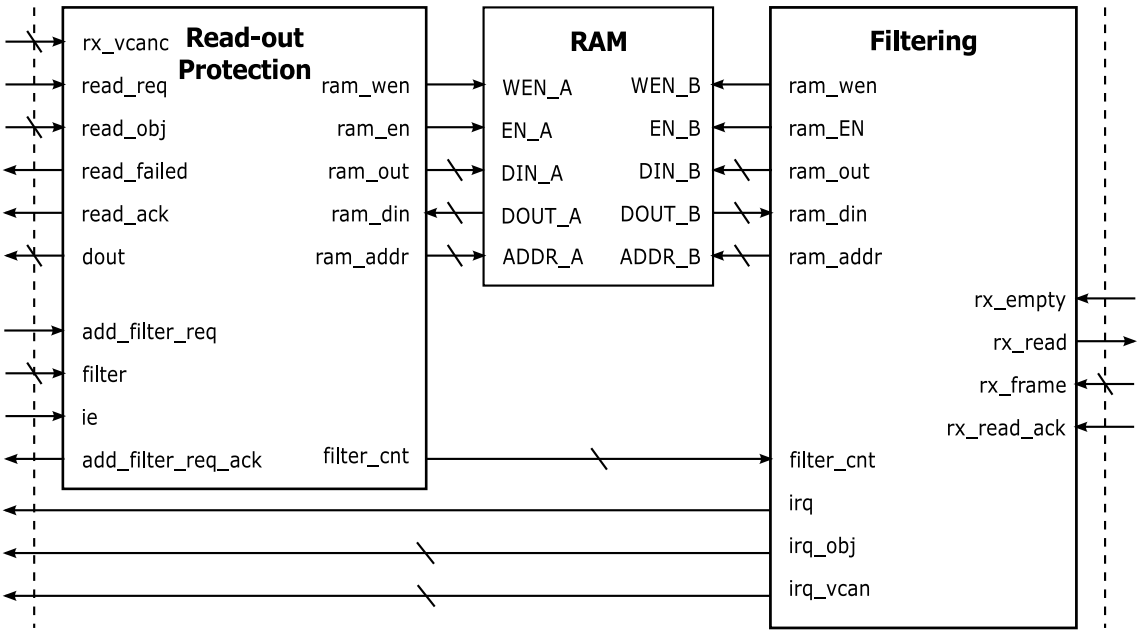


Figure III.26.: Block diagram of the implementation of the Rx path

to the concept designed in Section III.1.2.2, each object serves as a message filter and buffer at the same time. An illustration of an Rx message object is shown in Fig. III.27. It consists of two 64 bit wide RAM lines, where the first line contains the filter and control information, and the second line contains the received payload.

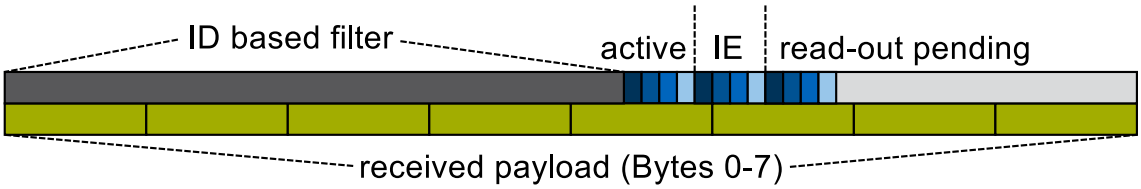


Figure III.27.: Memory layout of Rx message objects for $V = 4$ virtual CAN controllers [97]

When a received message is buffered in the protocol layer and thus ready to be processed, it is indicated by a low level in the *rx_empty* signal. By raising a flag on *rx_read*, the frame is delivered through *rx_frame*. Consequently, it has to be checked for acceptance. For that, the *Filtering* module iterates through the list of instantiated filters. The ID of the received frame is compared to the ID based filter of each object. If a match is found, the frame is stored. If no matching filter was found after a number of iterations corresponding to the *filter_count* signal, the message is discarded.

In addition to the ID based filter and the payload, each message object contains control fields specifying the configuration of the virtual CAN controllers. The *active*,

IE, and *read-out pending* fields are each subdivided into V bits, corresponding to the number of virtual CAN controllers. The *active* fields specify, which of the virtual CAN controllers is supposed to receive CAN messages stored in this object. If the respective interrupt enabled (*IE*) bit is set additionally, an interrupt will be forwarded whenever a message is stored, unless the *read-out pending* bit is high. This bit indicates that a message has not yet been read.

On the other side of the Rx RAM, the *Read-out Protection* module handles read accesses towards stored message objects as well as the filter configuration. Filters can be added through the *add_filter_req* signal. When the addition of a filter is requested for a virtual CAN controller *rx_vcan*, the list has to be checked as to whether a particular filter already exists. In this case, only the *active* flag has to be set and potentially the *IE* bit. If no filter with the ID exists, a new one will be installed and the *filter_cnt* is incremented.

Reads from the host system are issued through the *read_req* signal. They are targeted at a concrete message object specified by *read_obj*. The data will only be returned, if the corresponding *active* flag is set and the message has not yet been read by the respective partition. If the read is not justified according to the configuration, a flag is issued in the *read_failed* signal. Otherwise, the data is output using *dout* and the *read-out pending* flag is set low.

Table III.1.: Virtualization layer hardware resource requirements

module	FFs	LUTs as Logic
Tx Path	387	1095
Arbiter	154	227
Buffer Control	233	854
Access Control	2	46
Mgmt	14	32
Context Mgmt	0	6
RAM Control	217	760
Rx Path	52	480
Filtering	30	243
Read-out Protection	22	227
Virtualization Layer	439	1575
Protocol Layer	844	1239

The implementation was synthesized for the VC709 FPGA board using Vivado 2013.3. The implementation provides $V = 4$ virtual CAN controllers. Hardware resource requirements are listed in Table III.1. The table lists resource utilization

in terms of flip-flops (FFs) and lookup tables (LUTs) as logic. Additionally, the protocol layer uses 12 LUTRAMs and both Tx and Rx path use 2 RAMB36 Block RAMs.

The protocol layer serves as a baseline for comparison. It was implemented on the basis of an OpenCores CAN protocol controller[1], which is equivalent to a Philips/NXP SJA 1000. It was adjusted to fit the interfaces as shown in Fig. III.22.

The virtualization layer requires roughly have the FFs of the protocol layer, but needs 27% more logic resources. The virtualization not only enables multi-tenancy towards a shared CAN controller, but also provides significantly more advanced buffering capabilities (e.g. the SJA 1000 only provides a single message Tx buffer). Despite the added functionality, a break even point is roughly reached when two virtual CAN controllers are used, with increasing efficiency for higher numbers.

The per module resource consumption presented in Table III.1 allows an in depth inspection of the resource distribution. Listed are Tx and Rx path overall resource consumption alongside the submodules. If a top-level module requires more resources than the sum of submodules, it is caused by additional hardware in the top level module itself. This is e.g. the case for the *Buffer Control*, where a multiplex is implemented in the module itself (see Fig. III.24). Within the Rx path, resources are distributed evenly between the *Filtering* and the *Read-out Protection* module. More resources are used by the Tx path. Here, especially the priority queue manipulations implemented in the *RAM Control* require significant resources, which amount to ca. 50% of the overall virtualization layer. While the virtualization layer is efficient compared to the use of multiple stand-alone CAN controllers, its HW overhead could be reduced if simpler buffering mechanisms are sufficient.

4.2. Deadline-Aware Interrupt Coalescing

Deadline-aware interrupt coalescing was proposed as an extension to reduce the interrupt-associated overhead experienced in the host system. It can be implemented as an extension to the virtualized CAN controller.

Fig. III.28 presents a virtualized system, in which a virtualized CAN controller uses an interrupt coalescing module to reduce the number of IRQs. It is integrated within the Rx path of the virtualization layer.

The concepts presented in Section III.1.5 cover three versions of deadline-aware interrupt coalescing with varying degrees of granularity. Each version was evaluated with respects to its ability to reduce IRQs in Section III.3.2. Here, an implementation will be proposed and evaluated with respect to its resource consumption.

Because the module performing deadline-aware interrupt coalescing was implemented for the virtualized CAN controller, it must be able to perform interrupt coalescing for multiple virtual CAN controllers. In the case of of V virtual controllers, the same amount of interrupts must be managed by the module.

[1] `http://opencores.org/project,can`

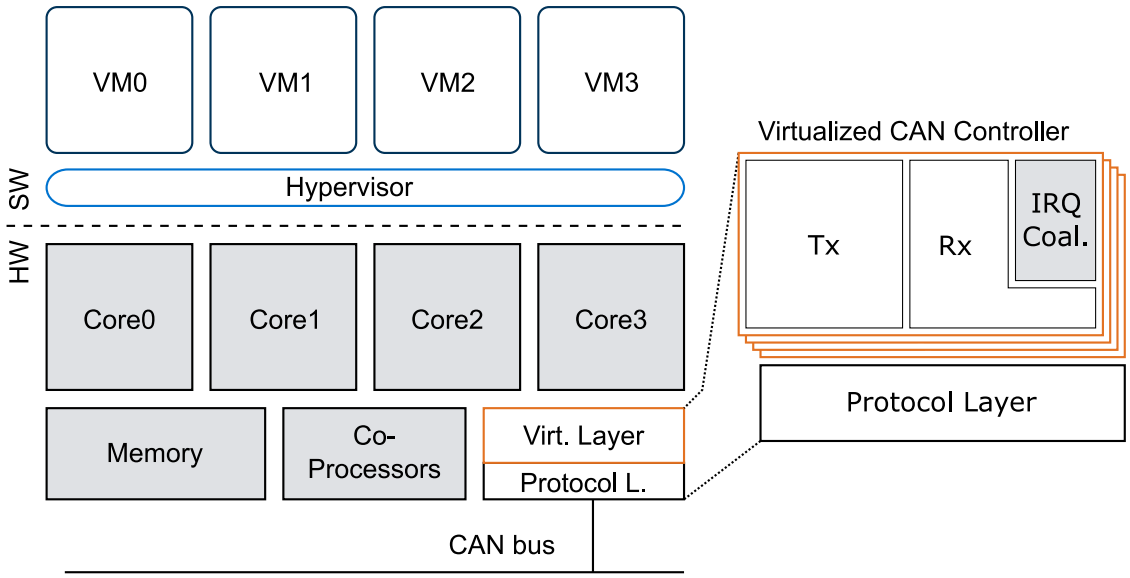


Figure III.28.: Architecture of a virtualized system (left) with virtualized CAN controller (right) that features interrupt coalescing extensions [98]

All three versions of deadline-aware interrupt coalescing were implemented in a conditional Verilog module. The actual version can be selected before synthesis by setting the corresponding parameter. Based on this, hardware implementation costs of the module can be compared for all versions.

Fig. III.29 shows the architecture of the interrupt coalescing module. Conditional components, ports, and signals are drawn dashed or dotted for dynamic deadline estimation and message-based delays, respectively. The task of the module is to buffer incoming interrupts and to forward them just before a deadline violation could occur. An interrupt request is raised after the reception of a frame. It is notified using the *irq_in* signal. Additionally, *irq_vcan_in* indicates which virtual CAN controllers correspond to the interrupt. While virtual controllers can receive the same message, they can be expected to receive a different overall subset of available messages. Thus, separate interrupt coalescing has to be performed for each virtual controller.

The function of the *timeout_update* module is to maintain a timeout counter indicating the time at which an interrupt has to be forwarded. All timers are regularly decremented using a clock derived from the CAN bus activity. Every time a bit is sampled on the CAN bus, *can_sample_point* has a positive flag and every timeout value is decremented by one bit time (2 µs at 500 kbit/s). When a new interrupt is requested, the timer may have to be updated. If the timeout equals zero, no interrupt is currently buffered and the timer is set to the estimated slack of the message. Otherwise, the timeout is only updated if the slack of the message just received is

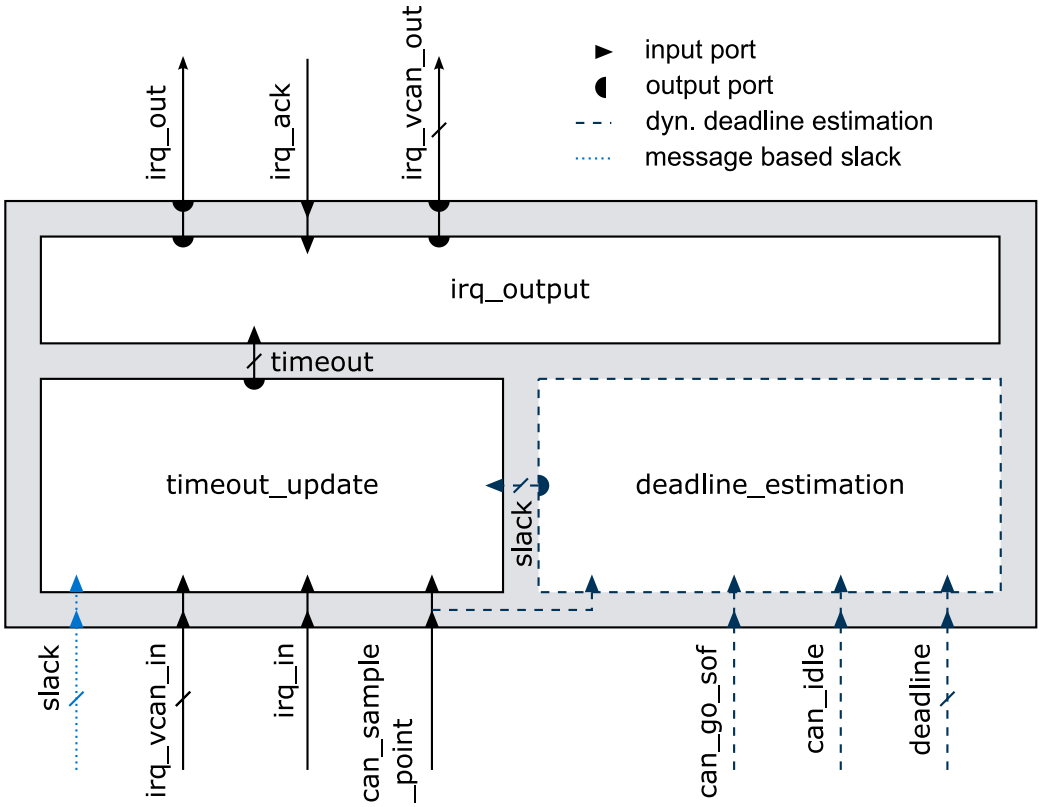


Figure III.29.: Implementation of the deadline-aware interrupt coalescing extension. It is illustrated as conditional implementation, covering all three design alternatives [98].

smaller than the current timeout value.

The slack of a message just received is determined in different ways depending on the mechanism implemented. Using fixed delay, the slack value is stored in a register within the module. Otherwise, a slack value is given from other modules. For message-based delay, the slack determined at design time is input through the *slack* signal. In the case of dynamic deadline estimation, the slack is determined at run-time by the additional *deadline_estimation* module.

The dedicated deadline estimation module relies on two signals taken from the protocol layer to determine the last time the CAN bus was idle. A rising edge in *can_go_sof* indicates a start of a frame. If the bus was idle previously (*can_idle* is high), a timestamp is taken. When the next message was received, the last idle time is assumed to be the release time of the message. Using static deadline information input to the module, the remaining slack can be calculated.

The *irq_output* module checks the interrupt timeouts and forwards the IRQs accordingly. The timeout values are checked sequentially for all VCANs. If a timer reaches zero, a positive flag on *irq_output* as well as the corresponding virtual con-

troller indicated by *irq_vcan_out* are issued to the host controller interface. The operation is acknowledged through *irq_ack*.

Table III.2.: Interrupt coalescing hardware resource requirements

module	FFs	LUTs as Logic
Fixed delay	74	160
Message based delay	74	244
Dyn. deadline estimation	108	267
Virtualization Layer	439	1575
Protocol Layer	844	1239

Table III.2 presents synthesis results for the VC709 (Virtex-7) FPGA board using Vivado 2013.3. The resource utilization of the respective implementations of the module is compared to protocol and virtualization layer overall resource consumption. The resource consumption of the virtualization layer excludes the interrupt coalescing extensions.

Implementation costs of deadline-aware interrupt coalescing increase with deadline estimation complexity. This is expected and leads to a trade-off between area overhead and IRQ reduction capabilities. The difference between fixed and message-based delay is caused by additional logic to check if the currently received message has smaller slack than the timeout value. This is not possible with fixed delays and does not have to checked. Dynamic deadline estimation requires additional flip-flops and LUTs, as it introduces an additional module to estimate the release time and calculate the remaining slack of a received message. Compared to the message-based delay implementation, around 45% additional flip-flops 10% additional LUTs are necessary.

Extensions for deadline-aware interrupt coalescing require less than 10% of the overall design of the virtualized CAN controller consisting of protocol layer and virtualization layer.

4.3. Prototypical x86 System Implementation

The virtualization extensions presented so far are platform independent. To validate the concept on a system level, it has to be integrated with a concrete host system. As virtualization technology is most matured in x86 systems, it was chosen as a prototyping platform. At the same time limitations exist, because x86 systems are designed for general purpose applications and are not tailored to safety critical real-time applications. System level latency measurements using the prototype presented here are conducted in Section III.5.

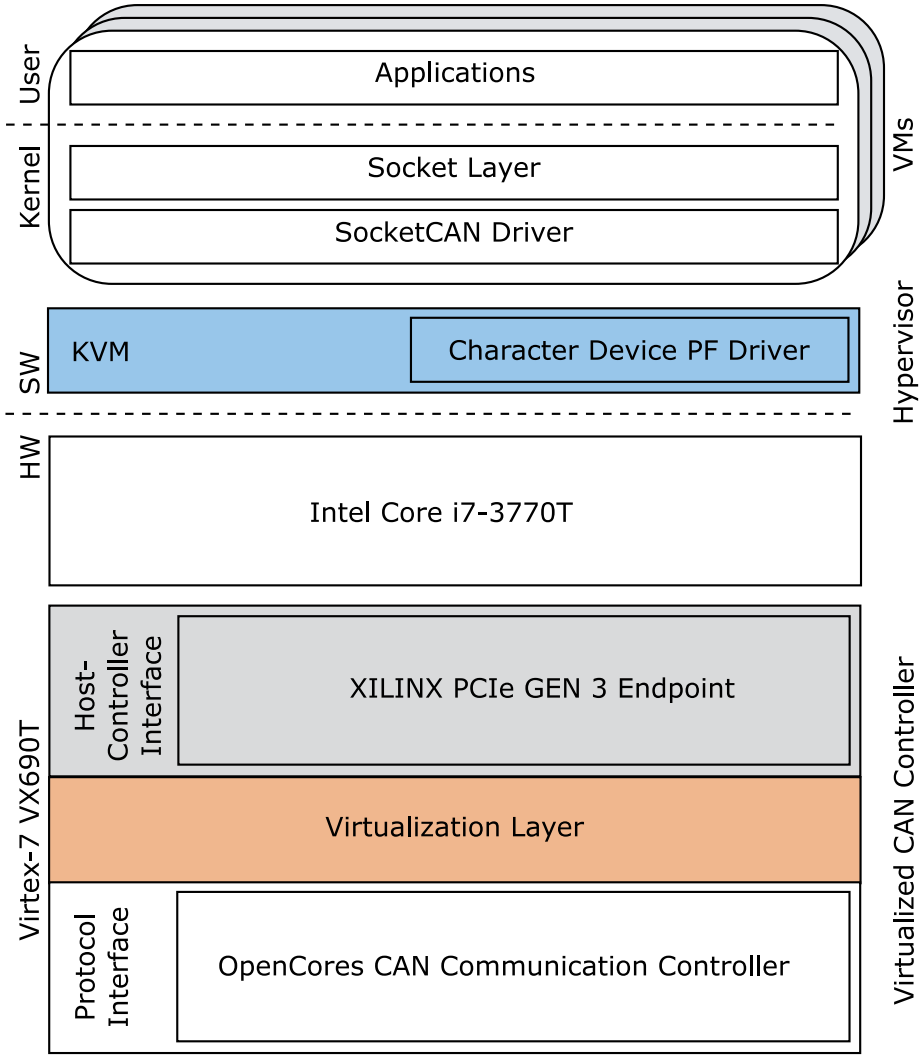


Figure III.30.: Prototypical Implementation of a System using a virtualized CAN controller

Fig. III.30 depicts the HW/SW architecture of the prototype. A listing of the components used is given in Table III.3.

On the software side, KVM version 3.8.13.6 was used as hypervisor. KVM stands for Kernel-based Virtual Machine and is contained in the Linux kernel. By loading the KVM kernel module, Linux works as a type-1 hypervisor. As the hypervisor is in control of the physical function (PF) of the virtualized hypervisor, it also integrates a character device driver.

The virtual machines are running Ubuntu 13.04 The driver structure uses the SocketCAN concept [108]. SocketCAN was developed at Volkswagen and integrates CAN drivers into the Linux networking stack. Applications can send and receive CAN messages through standardized socket interfaces.

The software is running on an Intel Core i7 CPU, which features all relevant state-of-the-art virtualization support including Vt-x and SR-IOV support.

The virtualization layer is integrated with the protocol layer and a host-controller interface on a Virtex-7 FPGA. The FPGA used here features an integrated and SR-IOV capable PCIe GEN 3 endpoint. It can thus be used for I/O virtualization scenarios as presented here. Requests are forwarded to the user logic through AXI stream interfaces. The protocol layer implements the CAN protocol through the use of an OpenCores IP core as introduced in Section III.4.1.

Table III.3.: Components of the prototype system

Component	Description
CPU	Intel Core i7-3770T (Quad-core, 2.5 GHz)
Mainboard	Intel DQ77MK
Memory	4x Corsair Vengeance 8 GB
Peripheral Interconnect	PCIe 3.0 with SR-IOV capability
FPGA Board	Xilinx Virtex 7 VC709 Connectivity Kit

This system prototype comes with some limitations that mainly arise from the fact that most of the components used are designed for general purpose applications. For instance, none of the involved components feature virtual channel support for PCIe. This feature is required for strict temporal isolation in the peripheral interconnect, but is only available in high-end server systems. Additionally, Xilinx' PCIe endpoint does not allow out-of-order processing of non-posted requests is possible. This means that one request has to be served, before the next can even be accessed. This limitation makes the implementation of a scheduling scheme for temporal isolation as introduced in Section III.1.4 impossible. Finally, a general purpose hypervisor and operating system were used. Nevertheless, the system can be used to verify general design objectives, like the applicability of direct access to a shared embedded networking device.

Because no CAN transceiver in integrated on the FPGA board, it has to be added to the setup using an extension card. The board has a dedicated connector for FPGA mezzanine cards (FMCs). Certain I/O pins of the FPGA are routed to the FMC connector and can thus be accessed on a custom hardware board. Here, an FMC with CAN transceiver was used to enable CAN access from the FPGA.

Fig. III.31 shows a schematic of how the FPGA is connected to the CAN bus. From the FPGA, to digital signals *can_tx* and *can_rx* are routed across the FMC connector to the CAN transceiver (PHY). Here, the transceiver receives the digital Tx signal as input and outputs the digital Rx signal. Communication on the CAN bus occurs using differential signaling. This means that the difference among the signals *can_H* and *can_L* determines if the signal is logical high or low. The sampled

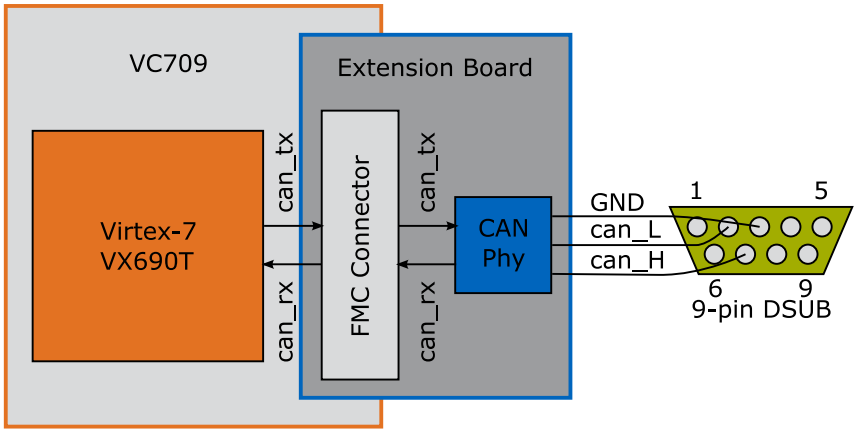


Figure III.31.: Schematic of the physical setup connecting the FPGA to the CAN bus

signal of the can bus is output as *can_rx*. If the *can_tx* has a dominant logic level (logical '0'), it is broadcasted on the bus. An additional ground output *GND* is offered by the transceiver. Its connection to the CAN bus is optional. The CAN bus signals are connected to a 9-pin DSUB connector.

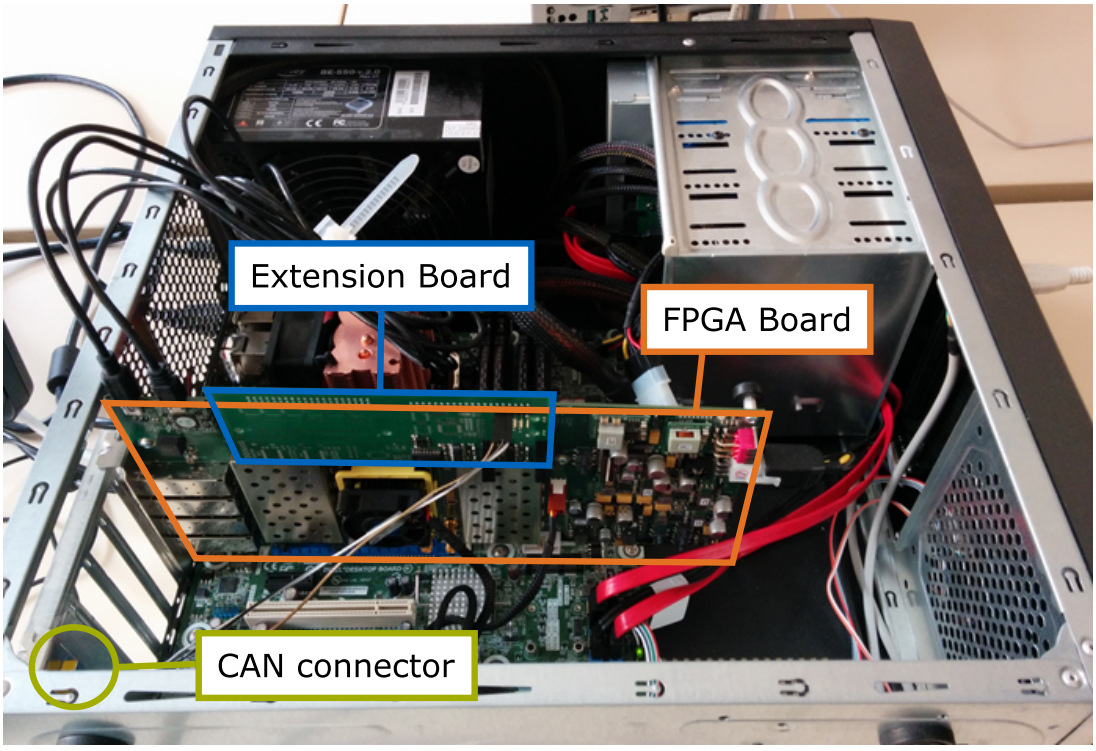


Figure III.32.: Photograph of the system depicting the key hardware components

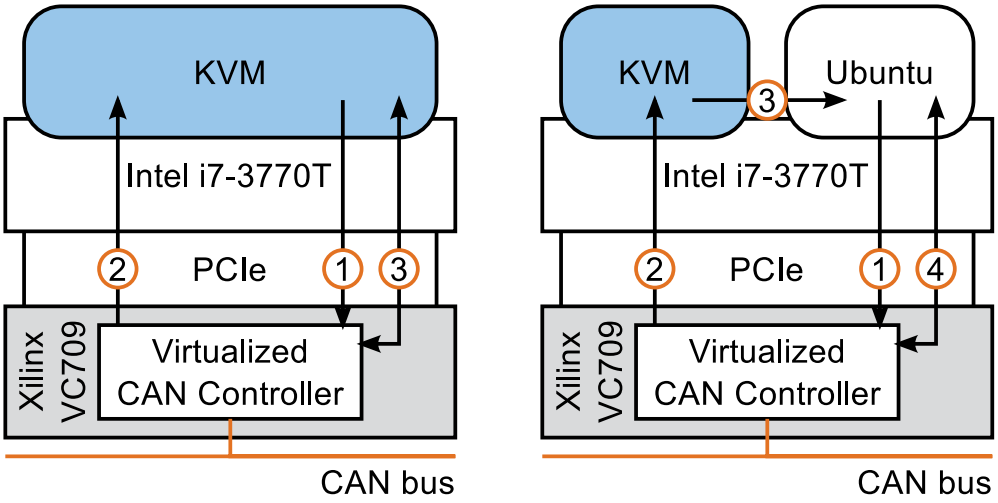
Fig. III.32 shows the assembled system. The FPGA board is plugged into a PCIe slot and the FMC with transceiver is plugged into the FPGA board. The DSUB connector is integrated into a separate slot bracket.

5. Experimental System Latency Evaluation

Based on the prototype system presented in Section III.4.3, an experimental latency evaluation is conducted here. This experiment is designated to evaluate the overall performance of virtualized system with SR-IOV enabled, virtualized CAN controller with respect to message transmit and receive latencies. The system performance will be compared to an equivalent setup without virtualization and a with results from related work using paravirtualization.

5.1. Experimental Setup

The experimental setup is designed to obtain results regarding latencies of message releases and transmission in a virtualized setup. The system under test corresponds to the x86 prototype system detailed in Section III.4.3. To obtain baseline results, an equivalent experiment was conducted in native conditions.



(a) Native (b) Virtualized

Figure III.33.: 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 [99].

Fig. III.33 depicts the setup for each scenario. In both cases CPU functions that lead to performance variability are deactivated. This includes hyper-threading, SpeedStep and Turbo Boost. The hypervisor used is KVM (3.8.13.6) and the VM is running Ubuntu 13.04. The CAN bus is operating at a bandwidth of 500 kbit/s.

The application level procedure carried out is the same in both experiments. A CAN message transmission is requested. The virtualized CAN controller is configured to receive this message after its transmission on the CAN bus. By receiving

the message on the same system, precise timing measurements can be made, as the very same real-time clock can be used for timing transmission and reception. After the application receives the message, it is sent again. This loop of sending and receiving is repeated 10,000 times, with the time between transmission request and successful reception being recorded. The experiment is repeated with all possible payload sizes, with the payload data being set to an alternating pattern of zeros and ones to avoid bitstuffing.

Fig. III.33a details the setup and experimental procedure in the native case. Only KVM is running and the virtualized CAN controller is assigned to it. Therefore, all functions are executed in host mode, providing native performance. During each measurement, the following actions occur (numbers corresponding to the one in Fig. III.33a).

(1) A message is issued by an application within KVM for transmission via the virtualized CAN controller

(2) The message is received after successful transfer on the CAN bus and an interrupt is immediately issued to the host system

(3) The message is read-out by the driver and received within the application.

The corresponding experiment for evaluating virtualized performance is illustrated in Fig. III.33b. On top of KVM, a VM is running Ubuntu. Within this VM, the measurement process is executed. Therefore, it is executed in guest mode of the processor. The sequence of events is similar to the native case and goes as follows:

(1) A message is issued by an application within Ubuntu for transmission via the virtualized CAN controller

(2) The message is received after successful transfer on the CAN bus and an interrupt is immediately issued to the host system

(3) The interrupt is received within KVM and forwarded to the VM

(4) The message is read-out within Ubuntu by the driver and received within the application.

The main difference within the sequence is step (3), where the interrupt has to be forwarded to the VM. x86 CPUs have to be configured to either receive all interrupts in host or guest mode. Therefore, virtualized systems require a privileged component running in host mode (here the hypervisor) to forward these interrupts into the respective VMs. This additional interrupt forwarding is expected to cause an overhead, even in setups using SR-IOV capable I/O devices.

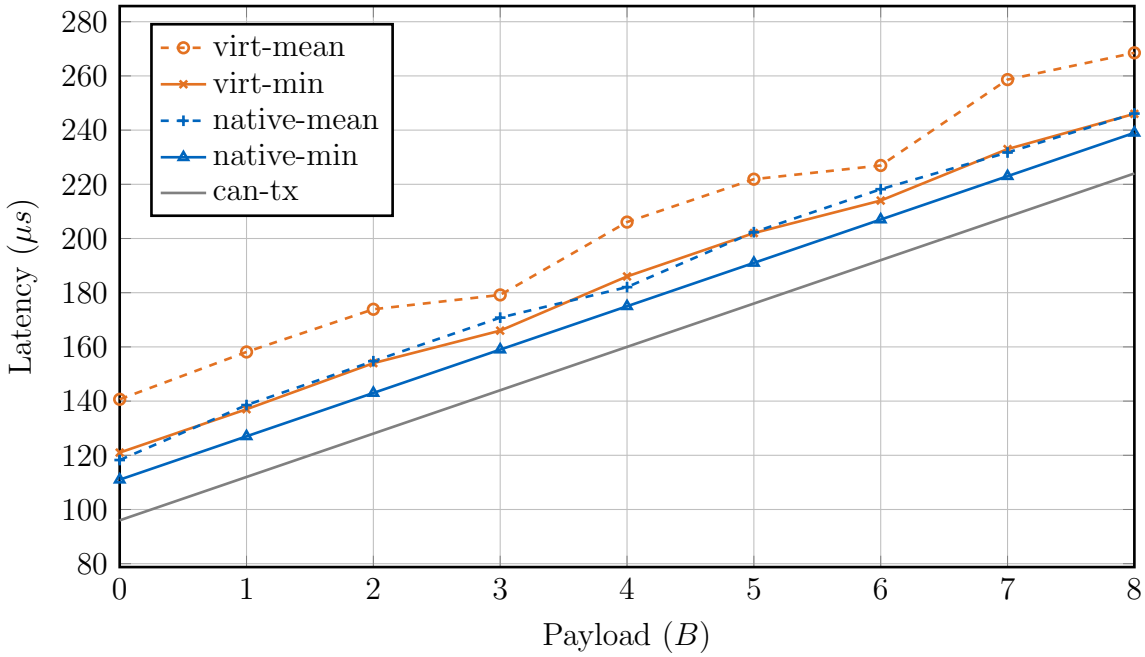


Figure III.34.: Latencies measured for a complete transmit-receive loop CAN communication in a native scenario on from within a VM in virtualized x86 system [99]

5.2. Results

Fig. III.34 shows minimum and average latencies measured the two experiments outlined, showcasing native and virtualized performance. These latencies also include the transmission time of the CAN frame on the bus. This transmission time is a theoretical lower bound for the latencies measured here. It is plotted alongside for reference (*can-tx*).

Measured latencies are increasing with the payload size of the CAN frame. This increase is only due to the transmission time on the bus. CPU as well as the virtualized CAN controller are able to process the whole payload data in parallel.

The measurements are intended to provide a comparison between latencies in a native and virtualized scenario. For measurements in a complex system like a virtualized x86 computer, many sources of interference exist, which can take unwanted influence on the results. This includes cache behavior, interrupts, as well as operating system and hypervisor scheduling. The hypervisor used lacks real-time support itself, so that a routine executed within the hypervisor may e.g. delay the release of a CAN frame within the VM.

The experiment was repeated 10,000 times for each payload. Therefore, little interference was present during the measurement with the smallest latency. It can thus be used as an isolated reference for message release latencies. To evaluate

average latencies, a three sigma mean [109] was used. To calculate a three sigma mean, only values within three standard deviations are used to compute a mean. In normal distributions, it still contains 99.7% of all values. Outliers, which would otherwise corrupt the average, can be eliminated.

Minimum latencies excluding the CAN transmission time range around 13 µs in the native case and between 20 µs and 24 µs in the virtualized scenario. This difference can be attributed to the additional interrupt forwarding necessary(corresponding to step (3) in Fig. III.33b). Other data path and control path operations that are done through memory-mapped I/O requests take the same time in both scenarios.

Mean latencies are around 11 µs larger than the minimum latencies in the native scenario and 25 µs in the virtualized case. The deviation from minimum latencies can be explained through competing tasks on the CPU, cache and TLB pollution, and varying on-chip and peripheral interconnect latencies. Latencies within a VM are increased more compared to the interference-free case, because tasks within hypervisor and within the VM are able to block CAN transmission and reception tasks.

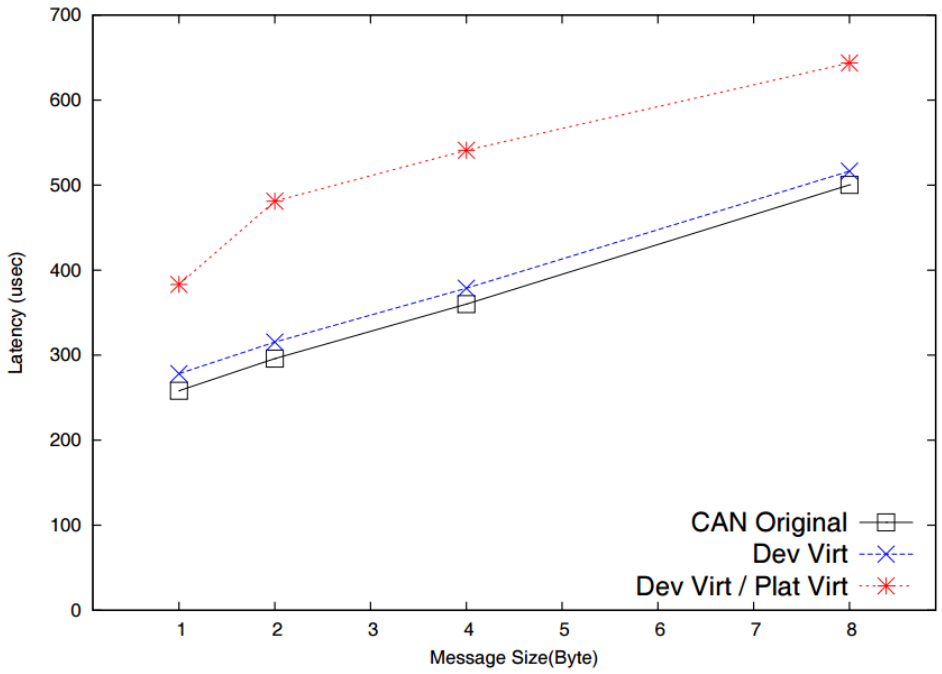


Figure III.35.: Results obtained in related work [88]

Kim et al. developed a paravirtualized CAN communication concept, which they implemented and evaluated using a comparable platform (Intel i7 2.8 GHz) and Virtual Box as hypervisor. For reference, their results are shown in Fig. III.35. The experience differs slightly, as the loop back was done through a second (non-virtualized) PC. Therefore, the overall latency includes two times the CAN trans-

mission and the loop back time on the second PC. Nevertheless, the virtualization overhead (difference between original and virtualized latency) can be directly related to the experiments at hand. It ranges between 120 µs and 200 µs and therefore exceed the results achieved here using hardware-assisted virtualization by an order of magnitude. It should be noted that a hosted (type-2) hypervisor was used in their experiments so that possible improvements could be achieved using a bare-metal hypervisor like KVM.

6. Summary

Scalability limitations of current automotive embedded system architectures require car manufacturers to consolidate ECUs on shared platforms. Multi-core processors provide separate processing cores, but still rely on shared hardware resources like caches, memory, and peripherals. Network-I/O controllers are highly critical with respect to the overall real-time capability of the system. In a scenario, where multiple partitions share one network-I/O controller like a CAN controller, safe, secure, and real-time capable operation must be ensured.

In this chapter, the enablement of multi-tenant communication through a shared I/O device was researched. An I/O virtualization concept was developed, which ensures the spatial and temporal isolation of concurrent partitions. A layered architecture was proposed, where a virtualization layer extends the capabilities of a commodity CAN controller to allow multi-tenancy. The architecture facilitates efficiency and scalability by focusing on resource sharing.

The sharing of resources in the virtualization layer leads to good efficiency, but makes temporal isolation hard to achieve at the same time. Here, a scheme that enables temporal isolation by scheduling requests towards the virtual CAN controllers was proposed. A weighted time-based round robin mechanism is used to forward requests to the virtualization layer. It is configured to minimize the latency of high priority messages and the number necessary context switches in worst-case scenarios.

In an analytic evaluation, the added latencies using temporal isolation were researched. For the highest priority messages, latencies smaller than without isolation were achieved due to the reduced amount of context switches necessary. In the scenario at hand, added latencies for lower priority messages reach upto around 20 µs. It was shown that latencies increase linearly with the number of consolidated partitions using the shared virtualized CAN controller. For scenarios with 16 virtual controller, maximum added latencies around 45 µs were witnessed. Such latencies are multiple orders of magnitude smaller than end-to-end latency requirements and range around 50% of the transmission time of a minimum sized CAN frame.

A simulated denial-of-service scenario was used to show, how the temporal isolation mechanism can be used to protect the integrity of each virtual CAN controller against each other. Without temporal isolation mechanism, significant head-of-line blocking can occur, which leads to complete priority inversions for high injection rates of malicious requests. The temporal isolation mechanism was shown to limit performance degradation to the corrupted partition.

Interrupts are a significant source of overhead in virtualized systems. Therefore, the concept of deadline-aware interrupt coalescing was proposed. This concept reduces the amount of interrupt requests (IRQs) forwarded to the host system by buffering and forwarding multiple interrupts at once using a single IRQ. To be applicable in real-time scenarios, knowledge of upcoming deadlines has been incorporated. Three different versions were researched, which differ in complexity and granular-

ity. The favored approach uses dynamic deadline estimation, to make pessimistic assumptions on the upcoming deadline of a message instance.

A simulative assessment was chosen to evaluate the ability of the proposed concept to reduce interrupts. Using deadline-aware interrupt coalescing, upto 20 interrupts could be coalesced to a single IRQ. While generally it is possible to accumulate more interrupts for increasing bus loads, the precision of the deadline estimation is degrading, so that a maximum efficiency is achieved around 80% bus load. The approach was shown to work in settings with tight deadlines around half the cycle time as well as implicit deadlines. Additionally, it was shown that the inter-arrival time of IRQs at the host is not only increased, but also more deterministic than without coalescing.

The virtualized CAN controller was implemented at RTL level using Verilog. The design was synthesized for a Virtex-7 FPGA to evaluate the efficiency with respect to hardware resource utilization. The baseline implementation is made up of a standard CAN controller, which is extended by the virtualization layer to provide real-time capable communication in multi-tenant scenarios. The overhead reaches a break even point for two virtual CAN controllers, i.e. resource utilization is similar as the use of two separate CAN controllers. Thus, the virtualization approach leads to a benefit for three or more virtual CAN controllers. The extensions for deadline-aware interrupt coalescing can be added at around 10% increased hardware cost.

Based on a prototypical implementation using an Intel Core i7 platform and an SR-IOV capable FPGA board, system latencies were evaluated. As the goal is to provide near-native performance with respect to latencies, measurements from virtual machines were compared to ones conducted in a native setting. The experiment showed that added latencies around 11 µs were measured within the VM. The added latencies are attributed to the need to forward interrupts from the hypervisor. Nevertheless, these latencies are small compared to ones measured in state of the art approaches using paravirtualization, which range between 120 µs and 200 µs.

IV. Network Virtualization for CAN

The trend of centralization and consolidation in automotive embedded systems requires isolation for concurrent workloads that are executed on a shared hardware platform. However automotive systems are highly networked systems with diverse communication requirements. The integration of mixed-criticality functions on shared hardware platforms also poses challenges and opportunities for the networks connecting them.

In current architectures, separate communication mediums are used in different functional domains, thus guaranteeing a physical isolation, but also being inefficient and providing limited scalability. Consolidating multiple communication flows of different criticality requires isolation in a sense, that freedom from interference has to be guaranteed. The only way of guaranteeing performance isolation of mixed-criticality communication in CAN is to use a robust ID assignment (i.e. messages with higher criticality have higher priority). This assignment policy is suboptimal with respect to resource utilization.

Network virtualization (c.f. Section II.2.4) can be used to achieve a logical isolation of communication flows on a shared physical medium. The following sections present a network virtualization approach applied to CAN.

Fundamentals of this chapter were published in [110]. The concepts were extended to incorporate traffic shaping (Section IV.3) in addition to policing. Also, schedulability analysis was introduced for the evaluation.

1. Design

Goal of the network virtualization concept presented here is to have multiple virtual CANs coexisting on a shared physical CAN bus. Figure IV.1 represents an example scenario, in which network virtualization is used to create virtual subnetworks. The physical bandwidth is divided among three virtual CANs (VCANs). The nodes within one VCAN are a subset of all physical nodes and can be overlapping with other VCANs.

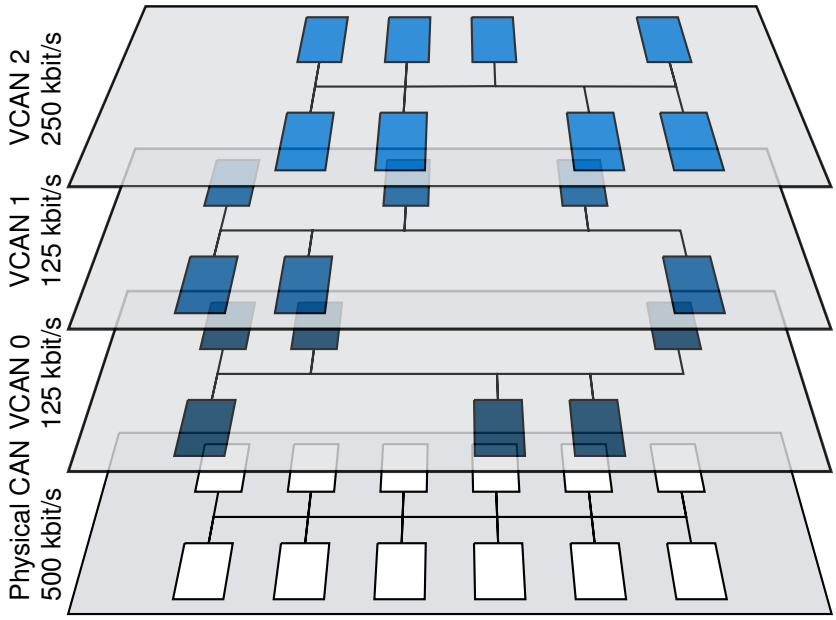


Figure IV.1.: Virtual Controller Area Networks (VCANs) coexisting on a shared physical CAN bus [110].

The main requirements for the concept stem from typical automotive requirements for safety-critical systems:

1. Spatial Isolation: A node within a specific VCAN must not be able to transmit CAN messages within other VCANs.

2. Performance Isolation: Temporal interference among concurrent VCANs must be bounded in a way that isolated performance guarantees can be given for each VCAN.

3. Fault isolation: Faults caused by a node within a node of a specific VCAN must not be able to cascade into other concurrent VCANs.

Two major design components in the design of a virtualized network are naming and admission control. They will be outlined in the following sections.

1.1. Naming within VCANs

When subdividing a physical network into virtual networks, it is important to avoid collisions with respect to the address space. CAN does not use the concept of sender/receiver addressing, but uses message IDs to specify the content of a message. Additionally, this ID also serves as a priority in CAN's arbitration scheme.

To avoid collisions within the arbitration process, CAN IDs have to be unique in classic CAN buses. Therefore, CAN IDs must also be unique within VCANs. If the same CAN IDs should be allowed within multiple concurrent VCANs, it must be ensured that these VCANs cannot enter arbitration at the same time. The asynchronous nature of CAN and the lack of a central arbiter make this hard with reasonable effort.

For the same reasons, for which CAN uses unique IDs, CAN IDs should also be unique across all concurrent VCANs. Assuming a set of messages $\mathcal{M}_v$ is used within VCAN v, the following condition must hold:

$$\mathcal{M}_v \cap \mathcal{M}_u = \varnothing, \forall v \neq u. \tag{IV.1}$$

While a subdivision of IDs guarantees collision free arbitration, it reduces the number of available IDs within each VCAN. This is acceptable, because VCANs are naturally restricted to a fraction of the available physical bandwidth, which is correlated with the number of unique messages.

To ensure the constraint of spatial isolation, a node within must be restricted to its own ID space. This can be done either by translating an arbitrary ID space into the constraint space within the specific VCAN, or by protection mechanisms, which prevent the transmission of messages with IDs that do not belong the respective VCAN.

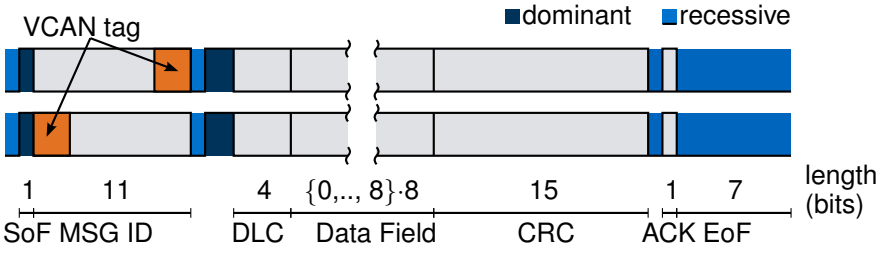


Figure IV.2.: Possible placement of VCAN tags within the MSG ID field of a VCAN frame

Figure IV.2 shows two possible solutions for dividing the ID space by placing a VCAN tag within the MSG ID field of each frame. The solutions use different positions of the VCAN tag: Placing it within the most significant bits assigns the highest priority messages IDs to the VCAN with the numerically lowest VCAN tag. Placing it within the least significant bits does not cause such a strict priority ordering among VCANs. Rather, the actual IDs used within each VCAN have to

be considered. This is disadvantageous, because it is contrary to the thought of performance isolation, where the behavior of a VCAN should not significantly affect others.

A strict priority arbitration scheme among VCANs is not a feasible approach to achieve efficient performance isolation, as it unproportionally benefits a single VCAN. The admission within VCANs has to be additionally constraint, which is subject of the following section.

1.2. Admission Control towards VCANs

While the naming scheme presented ensures spatial isolation, it is not sufficient to provide performance isolation. For a CAN communication, the performance measures are bandwidth and latencies. These measures are closely linked, as increasing the bandwidth of a bus means decreasing the latencies of all messages. The physically available bandwidth r_{phy} should be fully distributed among VCANs so that

$$\sum_{\forall v} r_v = r_{phy}. \tag{IV.2}$$

CAN uses a single logical bit line, so that the only resource which can be distributed along the VCANs is access time towards the bus. The time each VCAN has access to the physical bus must be proportional to their reserved bandwidth r_v. If this is guaranteed, the bandwidth reservation is automatically guaranteed as well. A variety of time division multiplex (TDM) methods exist and will be explored in the following.

Time division multiple access (TDMA) uses a cyclic schedule of time slots with fixed sized. These time slots are statically assigned to the distributed nodes. However, such schemes (used e.g. in Time-Triggered CAN (TTCAN) [111]) require global clock synchronization, which would have to be implemented additionally for the asynchronous CAN bus. The fixed size of the slots introduces additional overhead, because the bus will be idle after the transmission of smaller messages. A time-triggered version of CAN, TTCAN [111], was developed, yet it is not widely adopted.

Because synchronous schemes like TDMA seem too complex for the problem at hand, asynchronous admission control schemes might be fit. Statistical TDM (STDM) is such an asynchrnous TDM method, which enforces bandwidth guarantees by policing and/or traffic shaping based on statistical measures. Token or leaky bucket concepts are an established concept of STDM, which have been used in networking for more than two decades, e.g. for ATM virtual channel arbitration [112].

The operating principle of token bucket policing is analogue to a bucket, into which tokens are continuously added at a fixed rate proportional to the desired bandwidth. Upon transmission of data, tokens are removed proportional to the transmission time of the data unit. The bucket size (here b_v) is usually given as a number of bits. It

cannot be exceeded and therefore limits the size of a possible burst. Consequently, the rate at which tokens are added corresponds to the designated bandwidth of the policed traffic flow (here r_v).

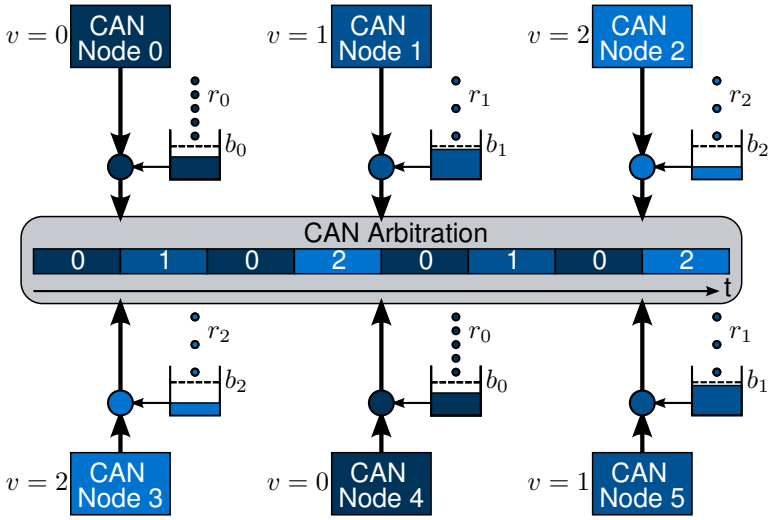


Figure IV.3.: Admission control towards the CAN bus is handled by token buckets for each VCAN, which enforce the reserved bandwidth. The fill level within each node of a VCAN is consistent [110].

Applied to the concept of CAN network virtualization, token buckets can be used to constrain the bandwidth used within each VCAN. While on a logical level, only one token bucket is necessary, an actual implementation has to rely a decentralized admission control. Therefore, a dedicated token bucket policer is necessary within each node of the network. The policing operation has to be consistent within in all distributed nodes of one VCAN, meaning the fill level must be equal at any time to avoid inconsistent scheduling decisions. Figure IV.3 depicts such an architecture.

A message m within VCAN v will be transmitted on the CAN bus, if the following conditions are met:

1. The message is currently the highest priority message queued within any node of VCAN v.
2. VCAN v has sufficient tokens to allow a transmission.
3. No higher priority VCAN wants to transmit a message and is able to do so.

The concept can only work, if information, especially the token bucket fill levels are consistent in all nodes. However, CAN is asynchronous and CAN nodes can have significant clock drift. An explicit synchronization of fill levels through CAN messages seems unfitting, as such synchronization cannot happen continuously and decreases the available bandwidth. Without explicit synchronization, CAN nodes

can only rely on information broadcasted on the CAN bus and thus available to all nodes. This includes any information available from CAN frames including their transmission time. Also, CAN nodes can derive a clock signal from the CAN bus during frame transmission. Token buckets must be updated using this clock. During idle periods, this clock signal is not available. Therefore, messages with the lowest possible priority will be constantly sent by a dedicated node within the network, thus guaranteeing a consistent clock signal at all times. These messages do not count towards the bandwidth quota of any VCAN and do not increase any latencies.

Information, which cannot be derived from the CAN bus and is only available to single node like the priority or length of a message buffered locally within must not be used, as it would lead to inconsistent scheduling decisions. Specifically, the following scenario must be avoided: VCAN v has sufficient tokens for the transmission of short message but insufficient tokens for long messages. Node A has a short, low priority message buffered and queues it for CAN arbitration. Node B has a long, high priority message buffered. In this case, the low priority would get transmitted, meaning that low priority messages can block high priority messages (priority inversion). To avoid such situations, transmissions should only be allowed, if sufficient tokens are available for the transmission of a maximum length message, independent of the length of the actual message. The corresponding token bucket fill level $fl_{tx,v}$, at which tokens suffice for transmission of maximum length messages can be derived as

$$fl_{tx,v} = \lceil C_{max,v} \cdot (r_{phy} - r_v) \rceil , \tag{IV.3}$$

where $C_{max,v}$ is the worst-case transmission time of the longest message within VCAN v and $r_{phy} - r_v$ is the rate at which tokens are reduced during transmission.

1.3. VCAN Delay and Token Bucket Size

The bucket dimensioning is an important design parameter within the VCAN concept. It controls the size of the inital burst a VCAN is allowed to do. This burst length directly influences the blocking experienced by lower priority VCANs. Therefore, the bucket size should be minimized to optimize the latencies within VCANs. However, the bucket must be big enough to guarantee no overflow of tokens occurs during a blocking by higher priority VCANs. In case of an overflow, the specific VCAN loses a share of its reserved bandwidth. The duration in which a VCAN v can be blocked from CAN access will be denoted as Θ_v. Therefore, an optimal bucket size is the minimal size, with which no overflow can occur. It can be computed as

$$b_v = fl_{tx,v} + \lceil \Theta_v r_v \rceil . \tag{IV.4}$$

It is equal to the sum of the minimum fill level necessary for transmission derived in (IV.3) as well as tokens added during a maximum sized blocking by other VCANs Θ_v.

The VCAN delay Θ_v is the central performance parameter of a VCAN v. It will be derived in the following by constructing a worst-case scenario for the blocking by other VCANs. This upper bound for Θ_v can be used to determine the bucket sizes as shown in (IV.4). The scenario is constituted by three different phases, which are illustrated in Figure IV.4:

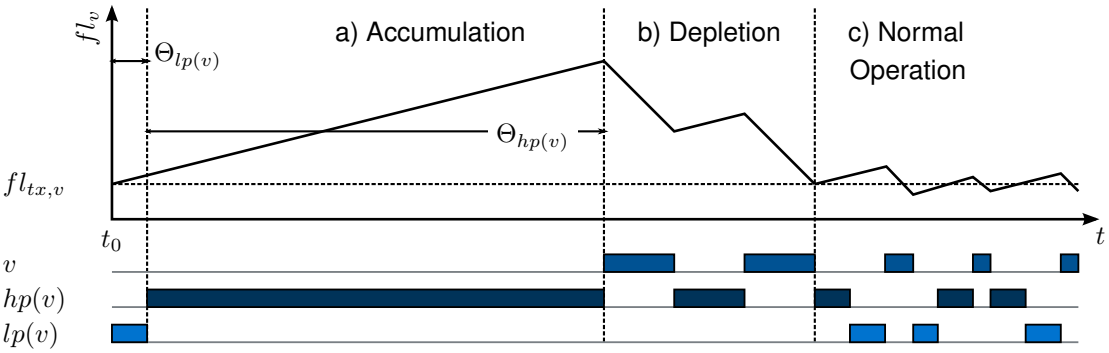


Figure IV.4.: Fill level of VCAN v during a burst scenario by higher priority VCANs [110]

Accumulation At $t = t_0$, the longest message within a lower priority VCAN has just started transmission. Its transmission time is $C_{max,lp(v)}$. Here, $lp(v)$ is the set of lower priority VCANs defined as $lp(v) = \{u \in \mathcal{V} : prio(u) < prio(v)\}$, with $\mathcal{V}$ being the set of VCANs. VCAN v reaches a suffient token level for transmission ($fl_v = fl_{tx,v}$) an infinitesimal time after t_0 and is delayed during the interval $(t_0, t_0 + \Theta_v]$. Afterwards, higher priority VCANs are assumed to have maximum number of tokens available to maximize the blocking. They occupy the bus until their buckets are completely depleted.

Depletion At the beginning of this phase, higher priority VCANs are not allowed to transmit, because they consumed their tokens. VCAN v has accumulated tokens equivalent to its bucket size. The accumulated tokens allow VCAN v to transmit at an average bandwidth equivalent to $\sum_{v \cup lp(v)} r_v$, the sum of all bandwidths reserved by VCAN v and lower priority VCANs. This leads to a depletion of the tokens until no more transmission is possible. The average bandwidth used by VCAN v in phases a) and b) combined is equivalent to r_v

Normal Operation Afterwards, VCAN v and higher priority VCANs are transmitting at their designated bandwidth r_v, thus giving room for lower priority VCANs transmit. For another worst-case scenario to occur, higher priority VCANs need to accumulate sufficient tokens for the burst to be possible. If the initial conditions are reestablished, the scenario can be repeated.

A bound for the VCAN delay Θ_v can be derived by analyzing the accumulation phase. It is composed by blocking from lower and higher priority VCANs

$$\Theta_v = \Theta_{lp(v)} + \Theta_{hp(v)}, \tag{IV.5}$$

where $\Theta_{lp(v)} = C_{max,lp(v)}$ and $\Theta_{hp(v)}$ is the blocking caused by the longest possible burst from higher priority VCANs. An upper bound for this blocking can be derived as

$$\Theta_{hp(v)} \leq \sum_{\forall u \in hp(v)} \frac{b_u + r_u \Theta_{hp(v)}}{r_{phy}}. \tag{IV.6}$$

The right side of this implicit inequality is equivalent to the transmission time of all data either allowed by the initial bucket fill level b_v or of tokens added during the delay $\Theta_{hp(v)}$ itself. It is impossible for any high priority VCAN to transmit any more data, as this is not possible by the means of the admission control.

Equation (IV.6) can solved for $\Theta_{hp(v)}$. Inserting it into (IV.5) gives an upper bound for the VCAN delay Θ_v as

$$\Theta_v \leq C_{max,lp(v)} + \frac{\sum_{\forall u \in hp(v)} b_u}{r_{phy} - \sum_{\forall u \in hp(v)} r_u}. \tag{IV.7}$$

As shown previouly, the bucket fill level will decay after the initial VCAN delay Θ_v. The fill level at the end of the blocking therefore corresponds to the optimal bucket size. It can be calculated by using (IV.4) with the upper bound for the VCAN delay Θ_v as given in (IV.7).

Calculating the VCAN delay Θ_v requires knowledge about bucket size b_v of all higher priority VCANs to calculate the size of a worst-case burst. Therefore, b_v has to be determined starting with the highest priority VCAN, and iteratively computing it for lower priority ones as well.

2. Timing Analysis

CAN timing analysis was presented in Section II.1.3.2. To be applicable in VCAN, it has to be adapted. Compared to a physical CAN bus, two additional effects have to be considered. First, communication within a VCAN can be delayed by up to Θ_v. This means that the bandwidth assigned to a certain VCAN is only guaranteed to be usable after this delay. Second, the transmission of messages within VCAN v is restricted by the admission control mechanism. On average, transmissions are allowed only with a bandwidth of r_v. Following, the worst-case scenario in classic CAN buses will be extended to incorporate these constraints, thus providing a real-time analysis for a message m within any VCAN v.

The worst-case scenario is similar to the one used to derive the necessary bucket size b_v. Traffic from other VCANs is assumed in a way that maximized the blocking inflicted upon VCAN v. Therefore, the scenario assumes that at the beginning, the maximum amount of tokens is available to higher priority VCANs.

The worst-case scenario for message m within VCAN v starts with a critical instant, meaning that all higher priority messages are released at the exact same time. An infinitely small time before, the longest lower priority message within VCAN v has just started transmitting. Also, the token bucket had the smallest fill level, at which a transmission is possible. Consequently, the minimum fill level is only recovered after $B_{m,v} \cdot r_{phy}/r_v$, where $B_{m,v}$ is the longest transmission time of lower priority message within VCAN v.

The transmission of message m can be delayed by other VCANs by up to Θ_v. The same blocking scenario, that was presented when deriving the bucket sizes b_v, can be applied again. As shown before, VCAN v can transmit at an increased rate after such a blocking and will eventually equalize the experienced delay. Therefore, the blocking is maximized, if other VCANs start delaying VCAN v just before message m is eligible for transmission, thus giving no chance to recover by using the accumulated tokens. The VCAN delay Θ_v poses an upper bound for the interference, which can be caused by other VCANs. While the burst scenario happens multiple times, it requires the higher priority VCANs to recover their tokens, giving time for VCAN v to make up for the delay. Therefore, the increased blocking for message m in VCAN v is derived as

$$\hat{B}_{m,v} = B_{m,v}\frac{r_{phy}}{r_v} + \Theta_v. \tag{IV.8}$$

This blocking is analog to the blocking in classic CAN analysis described by (II.3) and can be used to iteratively compute the queuing delay

$$w_{v,m}^{\mu+1} = \hat{B}_{m,v} + \sum_{\forall k \in hp_v(m)} \left\lceil \frac{w_{v,m}^{\mu} + J_k + \tau_{phy}}{T_k} \right\rceil C_k \frac{r_{phy}}{r_v}, \tag{IV.9}$$

where $hp_v(m)$ describes the set of messages within VCAN v which has higher priority than message m.

In addition to the increased blocking, the equation also considers the reduced bandwidth available to VCAN v. It introduces the factor r_{phy}/r_v, by which the transmission time is increased. The physical transmission of individual messages does not actually take longer, but has to be extended within the timing analysis, because during this period, admission control prevents immediate transmission due to insufficient tokens. Finally, the WCRT of message m within VCAN v can be calculated as

$$R_{m,v} = w_{v,m} + C_m + J_m. \tag{IV.10}$$

In contrast to the queuing delay, where the transmission times of messages are virtually extended to account for the reduced bandwidth, only the physical transmission time C_m of message m has to be considered here. While tokens might require additional time to recover, the transmission is completed afterwards.

The presented analysis allows to evaluate individual message delays within any VCAN. While the delay in bandwidth of VCAN v is given by Θ_v, this does not automatically mean messages have the same increased latency. Especially medium and low priority messages can have significant additional delay, if multiple further instances of high priority messages get queued during the queuing delay. This effect will be discussed in detail in the Evaluation (Section IV.4).

3. Traffic Shaping Through Dual Token Bucket Admission Control

Bandwidth and latencies are tightly interconnected in communications. An increased bandwidth usually means that latencies decrease. In CAN, sufficient bandwidth is a necessary requirement to guarantee bounded latencies. Whether a message set is schedulable (all messages are guaranteed to meet their deadlines) has to be decided by means of timing analysis. If a message set is not schedulable at a bandwidth comparable to its requirements, it can only be schedulable by increasing the bandwidth, as there is no way of decreasing latencies without increasing the bandwidth.

Using the VCAN concepts presented so far, the same principle applies. However, an overreservation of bandwidth to meet latency requirements is ineffecient, because it reduces the maximum possible utilization. This section presents a concept aimed at decoupling latency from bandwidth, thus increasing the efficiency of the approach.

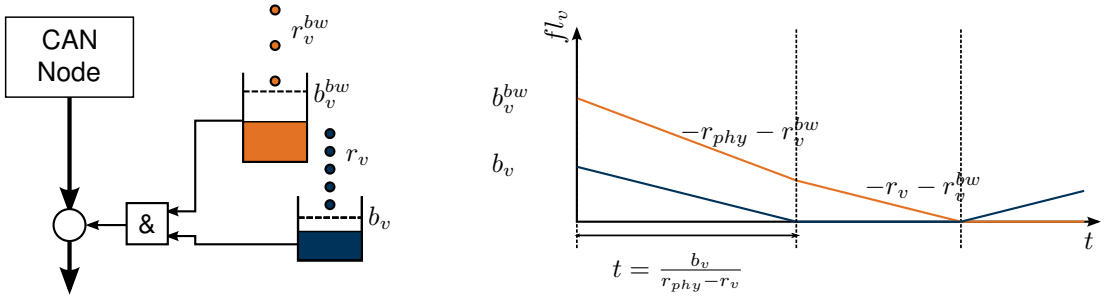


Figure IV.5.: Operating principle of dual token bucket policing admission control

Two buckets are used for admission control. One bucket bucket characterized by $(r_v,\ b_v)$ is equivalent to the one used previously. It is dimensioned to guarantee schedulability for all messages within VCAN v. A second token bucket is used for long term bandwidth policing. It is characterized by $(r_v^{bw},\ b_v^{bw})$, where $r_v^{bw} \leq r_v$. The minimum necessary bandwidth r_v^{bw} is given as

$$r_v^{bw} = \sum_{k \in \mathcal{M}_v} \frac{C_k}{T_k}. \tag{IV.11}$$

For a transmission to be possible, both token buckets need to have sufficient tokens $(\geq fl_{tx,v})$. During a worst-case scenario, transmission should be possible at r_v. Such a scenario can be described as a busy-period. Analog to the busy period in classic CAN buses (II.4) and using the worst-case scenario derived for VCANs (IV.9), the

maximum busy period of a VCAN can be calculated in the iterative form

$$t_{busy,v}^{\mu+1} = \sum_{\forall k \in \mathcal{M}_v} \left\lceil \frac{t_{busy,v}^{\mu} + J_k}{T_k} \right\rceil C_k \frac{r_{phy}}{r_v}. \tag{IV.12}$$

The secondary bucket, designated for long term bandwidth enforcement at r_v^{bw}, should not interfere in such scenarios to guarantee latencies as expected from a transmission rate r_v. During a busy-period, tokens are taken from the buckets at a rate equivalent to r_v. Accounting for the tokens added, the total of number tokens reduced from the secondary bucket is equivalent to $\left\lceil t_{busy,v}(r_v - r_v^{bw}) \right\rceil$. Therefore, the bucket size b_v^{bw} is constrained as

$$b_v^{bw} \geq fl_{tx,v}^{bw} + \left\lceil t_{busy,v}(r_v - r_v^{bw}) \right\rceil, \tag{IV.13}$$

where $fl_{tx,v}^{bw}$ is the minimum fill level of the secondary bucket. It can be calculated using (IV.3). Another constraint is equivalent to the one formulated for the primary bucket. The bucket must be big enough, so that no overflow can happen. The condition is

$$b_v^{bw} \geq fl_{tx,v}^{bw} + \lceil \Theta_v r_v^{bw} \rceil. \tag{IV.14}$$

The calculation of the VCAN delay Θ_v has to be adjusted, as the burst from higher priority VCANs is now more complex: it starts at line rate, continues at the higher rate r_v and finally continues as r_v^{bw}. Depending on the duration of the blocking by higher priority VCANs, the amount of tokens available during a time t is equal to

$$tokens_v(t) = \begin{cases} b_v + r_v t, & \text{if } t \leq t_{busy,v} \\ b_v + (r_v - r_v^{bw}) t_{busy,v} + r_v^{bw} t, & \text{otherwise.} \end{cases} \tag{IV.15}$$

The maximum blocking by higher priority VCANs is constrained by the time it takes to use up these tokens at line rate r_{phy}

$$\Theta_{hp(v)} \leq \sum_{\forall u \in hp(v)} \frac{tokens_u\left(\Theta_{hp(v)}\right)}{r_{phy}}. \tag{IV.16}$$

This implicit equation is not well suited to be solved analytically for lower priority VCANs, as it combines multiple case statements. However, a bound can be numerically computed through fixed-point iteration:

$$\Theta_{hp(v)}^{\mu+1} = \sum_{\forall u \in hp(v)} \frac{r_{u,avg}\left(\Theta_{hp(u)}^{\mu}\right) \cdot \Theta_{hp(u)}^{\mu}}{r_{phy}}. \tag{IV.17}$$

The overall VCAN delay Θ_v can be computed as previously using (IV.9), thus enabling a WCRT analysis as described in IV.2. The dual bucket approach leads to improved results with respect to schedulability, which are at least equal to the ones obtained with a single bucket, as this is just a special case of the dual bucket principle. The benefits of dual bucket admission control over single buckets will be quantified in the following section.

4. Analytic Evaluation

In real-time applications, analytic methods are used to determine, whether a system will meet all required deadlines (schedulability). As part of this evaluation, real-time analysis will be used to compare the proposed methods to state of the art approaches. For consistency, the same traffic distribution was used as in the evaluations of Chapter III. It is used to generate multiple scenarios within this and the following section.

First, the different behavior among the two proposed solutions will be demonstrated using the example of a single traffic scenario and configuration. The scenario uses four different criticality levels. Messages belonging to each criticality level are assigned to separate VCANs in order to ensure performance isolation among them. Highest critical messages are assigned to VCAN $v = 0$. The assignment of bandwidth to VCANs will be optimized to first ensure real-time capability in high-criticality VCANs. The lowest priority VCAN is assumed to contain only best effort traffic, which is only constrained by bandwidth requirements.

The token bucket configuration policy differs for single and dual token bucket approaches. They are as follows:

- **Single Token Bucket (VCAN-S):** Every VCAN v is configured through the parameters r_v (token rate) and b_v (bucket size). To ensure sufficient bandwidth is allocated for every VCAN, it has to be ensured that $r_v \geq U_v$, where U_v is the bandwidth utilization of all messages within VCAN v. Remaining bandwidth can be allocated towards VCANs in order to reduce latencies and ensure real-time capability. For that, starting with the highest priority VCAN, a real-time analysis is conducted using the minimal bandwidth. If latency requirements are not met, the bandwidth is increased in steps of 1% of the physical bandwidth. When the VCAN is schedulable, the procedure is repeated with the next highest priority VCAN. The bucket sizes are determined according to (IV.4).

- **Dual Token Bucket (VCAN-D):** The dual token bucket approach introduces an additional set of parameters $\{r_v^{bw}, b_v^{bw}\}$. As the purpose of the shaper defined by these parameters is the enforcement of long term bandwidth availability, it will be configured to bandwidth requirements of each VCAN. The latencies for each VCAN can be controlled through the second token bucket. The assignment of token rates follows a similar iterative approach as for single token bucket admission control. Starting with the minimum bandwidth requirement, r_v is increased iteratively if deadlines are missed. For the highest priority VCAN, r_v could be chosen as high as the physical transmission rate r_{phy}. This would mean that transmissions are possible at line rate for a limited sized burst. Generally, the highest value for this token rate in any VCAN is

$r_v \leq r_{phy} - \sum_{\forall u \in hp(v)} r_u$. If, due to this constraint, a VCAN cannot receive a token rate $r_v \geq r_v^{bw}$ its operation is equivalent to that of a single token bucket.

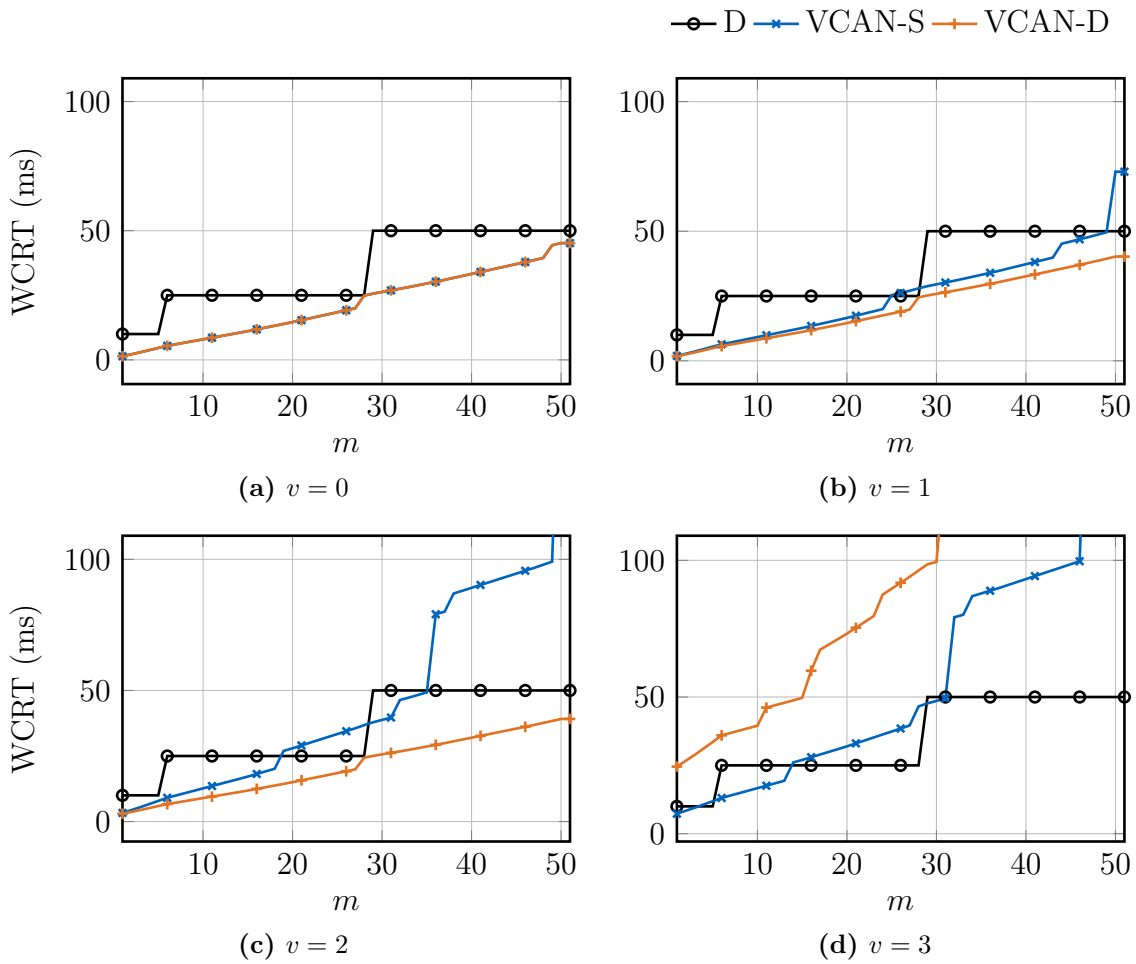


Figure IV.6.: Worst-case latencies for multiple VCANs with a total of $V = 4$ VCANs with 85% overall utilization

Fig. IV.6 shows the worst-case latencies in an example scenario using the procedure described above to configure the admission control. Plotted along the latency are the deadlines of each message. A VCAN is considered schedulable, if all latencies are smaller than the deadlines, i.e. the plotted line of the latencies is always below the deadlines.

Using the single token bucket admission control, schedulability can only be guaranteed for the highest priority VCAN. The second highest experiences small deadline violations for medium and low priority messages. For the two lowest priority VCANs, significant deadline violations can be seen. Because these VCANs have bandwidth

reservations close to their minimum bandwidth requirement, worst-case latencies of low priority messages get very large and exceed 100 ms.

The introduction of a second token bucket in the admission control leads to a significant improvement in schedulability. Three out of four VCANs are deemed schedulable. The final VCAN is not able to meet any deadline in the worst-case scenario. Increased latencies in the lowest priority VCANs are the price that has to be payed in order to lower the latencies in high priority ones.

This example shows the decoupling of bandwidth and latencies using the two level admission control mechanism. While maintaining bandwidth guarantees for all VCANs, latencies can be reduced for highly critical VCANs in the favor of best effort VCANs. This feature combined with the performance isolation provided makes it desirable technology in mixed-criticality communications.

The results presented in Fig. IV.6 are limited to a single traffic scenario. To evaluate the applicability of the approaches to a large set of scenarios, a schedulability analysis was conducted. Using randomly generated message sets at multiple levels of overall utilization, real-time analysis is used to determine the level of schedulability at each utilization level. Utilizations considered here range between 30% and 100% with increments of 2%. At each level, 10,000 message sets were generated, leading to a total of 1,400,000 evaluated scenarios. In order to reflect the mixed-criticality nature of the scenarios, real-time capability is only required for the $V - 1$ highest priority VCANs. The schedulability S corresponds to the percentage of scenarios, in which all real-time constraint messages are deemed schedulable.

The schedulability will be used to compare the VCAN variations to state-of-the-art approaches. Two approaches will be compared:

- **Optimal priority assignment (OPA):** In [113], Audsley presented an optimal priority assignment policy for fixed priority. Applied to CAN, it ensures that a message set is schedulable if it is schedulable using any priority assignment. This method does not ensure performance isolation and is therefore not applicable in mixed-criticality scnenarios.

- **Robust priority assignment (RPA):** The ID space is partitioned into V areas. Messages with the highest criticality have the highest priority. Within each partition, IDs are assigned to Audsley's assignment policy. While providing downward performance isolation (low-criticality messages cannot interfere with high-priority messages), this approach is inefficient, as it provides schedulable message sets only for low network utilization [114].

Fig. IV.7 shows the results of the exploration. Both VCAN approaches provide a significant advantage with respect to schedulability compared to RPA, while guaranteeing performance isolation. Neither approach is able to reach the schedulability levels of OPA. However, OPA does not provide performance isolation and thus, is not applicable in mixed-criticality scenarios.

RPA is only able to provide schedulability for low bus utilizations smaller than 50%. This is inefficient and thus uneconomical. The low possible utilization stems from the fact that very low latencies can only be achieved in the highest criticality partition. The more messages are transmitted in the highest criticality partition, the longer the latencies become in other partitions. The shortest worst-case latency achievable in a partition is always greater than maximum latency in the next higher critical partition.

The VCAN approach suffers high degredation around 76% using a single token bucket (VCAN-S) and around 88% using dual token bucket (VCAN-D) admission control. VCAN-D is dominant to VCAN-S in terms of schedulability, as a set that is not schedulable using VCAN-D cannot be schedulable with VCAN-S. This is guaranteed, as VCAN-S is a special case of VCAN-D (with the two buckets sharing the same configuration). Of course, VCAN-D has increased implementation and configuration effort. The significant difference in schedulability indicates that it might be worthwhile.

OPA achieves the best results regarding schedulability. Even for very high utilizations, around 50% of the sets are still schedulable. This is achieved because the lowest-critical messages do not need to meet their deadline for schedulability. However, these results only hold true as long as all senders comply to their specification,

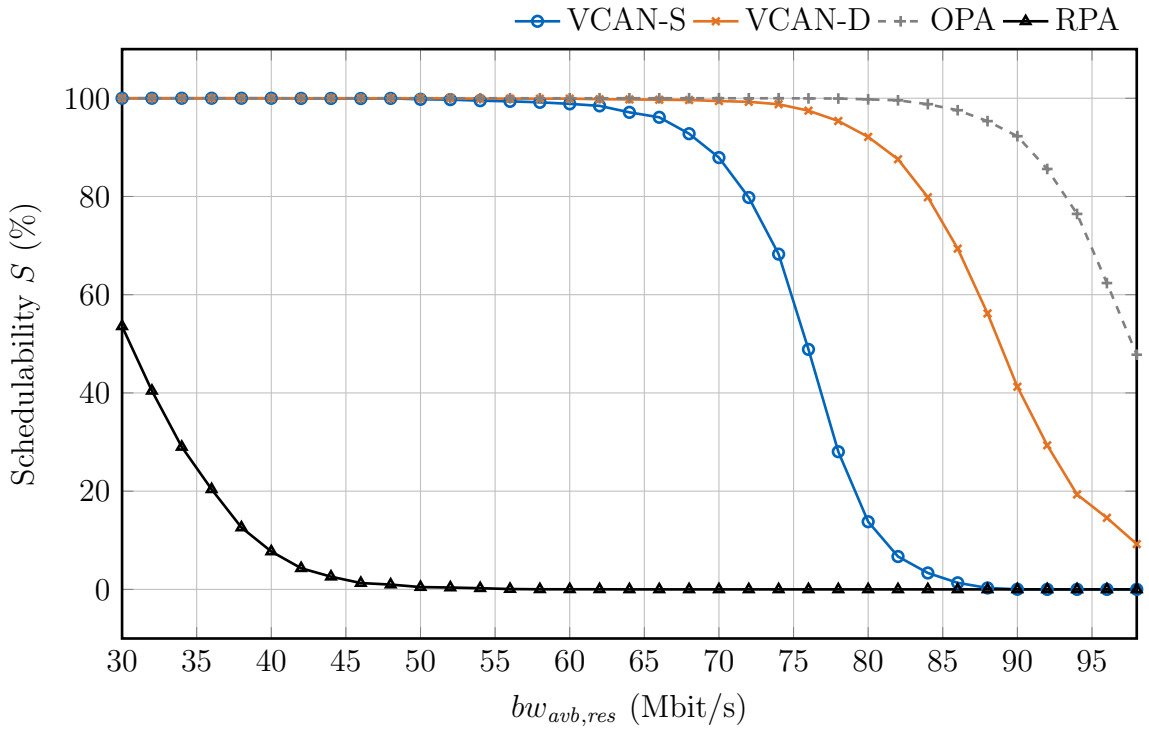


Figure IV.7.: Schedulability results for $V = 5$ VCANs using single and dual token bucket admission control

which is not guaranteed in mixed-criticality scenarios.

The schedulability shows the applicability of the VCAN approach using a wide range of scenarios. So far, the evaluation only covered worst-case analysis. While worst-case latencies are crucial in real-time applications, average or typical performance is also important in mixed-criticality scenarios, especially for lower priority VCANs.

5. Simulation

To gain information on typical latencies and their distribution, a simulation was conducted. The evaluation was done using an event-based Matlab simulation and is cycle and bit accurate. For comparability, the same scenario as depicted in Fig. IV.6 is used. Thus, the simulation also allows a verification of the worst-case analysis.

Two different experiments were conducted. The first experiment represents a typical traffic scenario in normal operation. The second demonstrates the performance isolation properties by showcasing a flooding scenario.

Experiment 1: Goal of the simulation is to provide realistic measurements using an example traffic configuration. The traffic scenario specifies a cycle time T_m for every message m. In the simulation, messages are always released cyclic. The initial release of each message is determined randomly and happens in the interval $\{0...T_m\}$. The cyclic behavior of the messages also means that similar traffic scenarios occur in a cyclic fashion. To improve the coverage of the simulation, it is started 10,000 times with varying random initial releases. Each configuration is simulated for 200 ms, leading to an overall simulated time of 2,000 s.

Fig. IV.8 shows the measured average and maximum latencies using single and dual token bucket admission control. Average latencies are generally around an order of magnitude smaller than the maximum recorded latencies. The highest priority VCAN suffers from the least blocking from other VCANs and averages latencies from around 400 µs up to 2.5 ms. Similar latencies are measured for VCAN $v = 1$ (500 µs up to 3.5 ms). In the highest priority VCANs, average latencies for single and dual token bucket implementation are comparable. For VCAN $v = 2$, significant differences can be measured. The analytic evaluation indicated that not all deadlines are guaranteed to be met using a single token bucket and thus, schedulability is not guaranteed. The increased latencies are also reflected in the measurement, while the dual token bucket provides latencies similar the highest priority VCANs.

Maximum latencies are essentially equal for the case of the highest priority VCAN. This stems from the fact that the configuration procedure described above leads to equal token rates and thus latencies. For VCANs $v = 1$ and $v = 2$, the dual token bucket approach achieves significantly lower latencies, because it is able to increase the token rate during a burst while maintaining bandwidth guarantees for the lowest priority VCAN. Two observations can be made regarding the lowest priority VCAN. First, this is the only VCAN in which the single token bucket solution offers smaller latencies. This was expected from the analytic evaluation. It is the price that has to be paid in order to reduce latencies for higher priority VCANs. Secondly, it can be observed that latencies for VCAN-S are smaller than in the next higher priority VCAN. This possible, because in the experiment presented, overreservation was used in higher priority VCANs to reduce latencies. In normal operation, this bandwidth is not actually consumed and can be used by the lowest priority VCAN. However, these latencies cannot be achieved, if e.g. erroneous behavior in higher

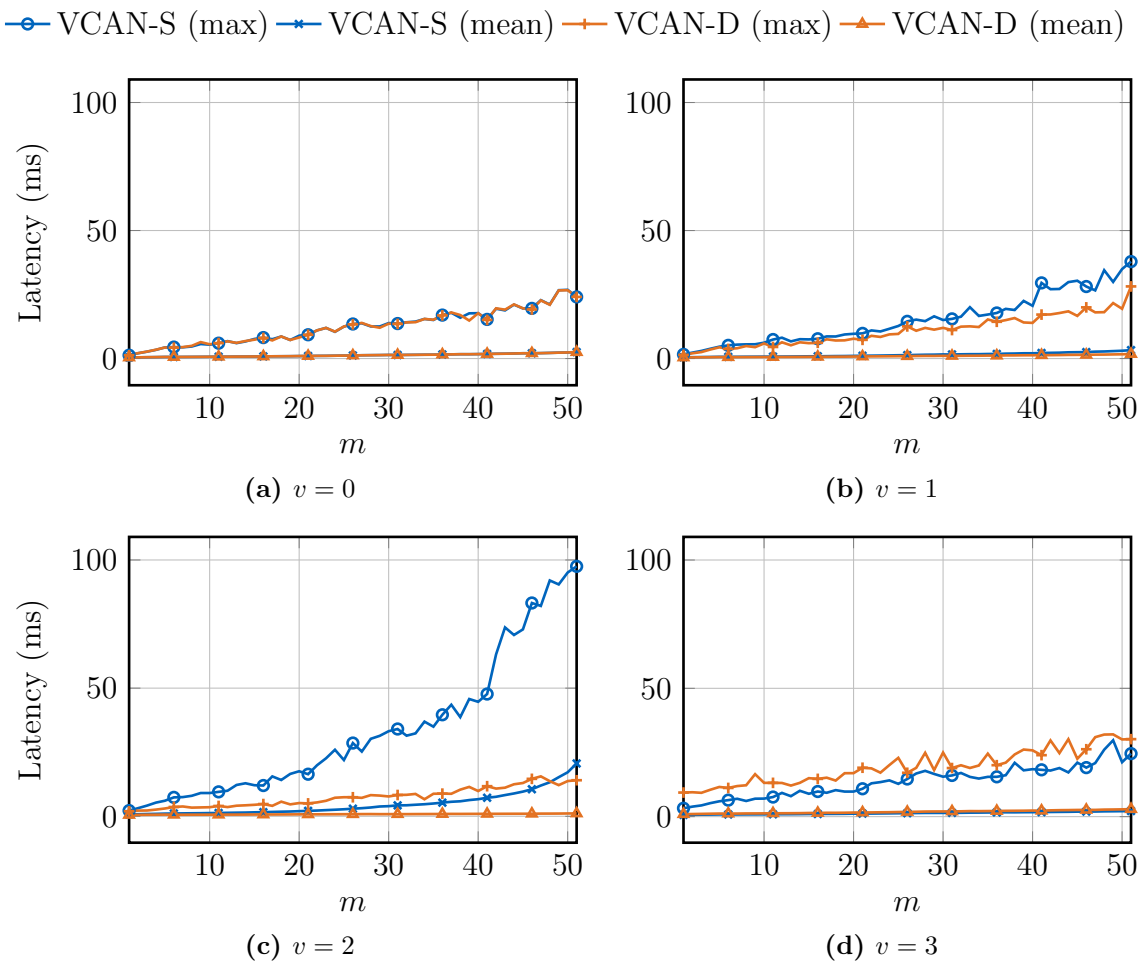


Figure IV.8.: Simulated latencies for multiple VCANs with a total of $V = 4$ VCANs

priority VCANs led to a complete utilization of reserved resources.

The main objective of the network virtualization concept is to provide performance isolation among partitions. This means that performance guarantees must hold independent of the operation in other partitions. The timing analysis from Section IV.2 already indicated on a mathematical level that timing bounds for a VCAN v are only dependent on the static VCAN configuration and the traffic scenario within the respective VCAN v. As part of the evaluation, these analytic assumptions should be verified using simulative measurements.

Experiment 2: Goal of the second experiment is to evaluate the behavior of VCANs if other partitions behave erroneous or maliciously. The experiment is carried out consecutively for every VCAN. When evaluating a VCAN v, the input traffic scenario is equivalent to the first experiment. All other VCANs have a modified behavior, where as many maximum sized frames as possible are sent. This behavior

represents a flooding and would lead to total performance degradation in a legacy CAN bus.

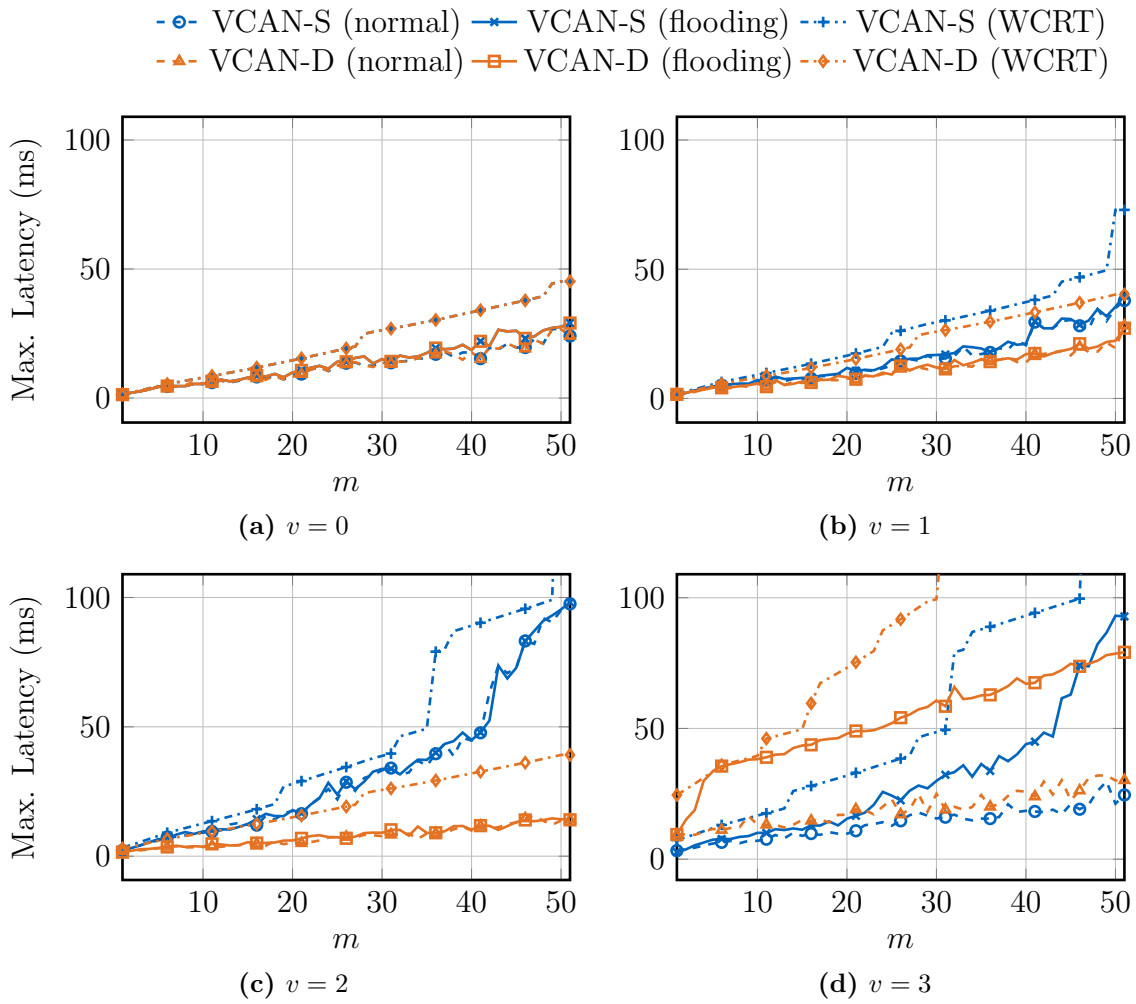


Figure IV.9.: Simulated latencies for $V = 4$ VCANs. Depicted are latencies in flooding conditions and normal operation.

The results of the experiment are illustrated in Fig. IV.9. For reference, the results in normal operation without flooding as well as the analytic bounds computed in Section IV.4 are presented alongside.

Measured latencies in both experiments are strictly smaller than the analytic worst-case latency bounds. This result gives confidence regarding the correctness of the analysis. Due to the limited coverage of the simulation, which cannot cover an infinite set of traffic configurations, measured latencies are smaller in the majority of the cases. Especially low priority messages differ strongly in measured and predicted worst-case latencies, as the respective worst-cases are more complex in the sense that

more higher priority messages need to behave in the specified way. Thus, the worst-case is likely to occur during a simulation for high priority messages.

The difference between normal operation and flooding scenarios varies depending on the priority of the VCAN. The highest priority VCAN suffers little temporal interference. Thus, performance in normal operation and flooding are nearly equivalent. Because only little blocking by other VCANs can be inflicted, only small buffer sizes are needed (c.f. (IV.4)). There, even if all other VCANs are idle, similar latencies are expected.

Within VCAN $v = 1$ and $v = 2$, increased latencies upto 2 ms have been measured in the flooding scenarios. Nevertheless, latencies remain well within the analytic latency bounds. The lowest priority VCAN on the other hand suffers from significantly increased latencies, but again, all within the analytic bounds. The increase is due to the fact that higher priority VCANs completely utilize their bandwidth in the flooding scenario. Thus, the lowest priority VCAN is constrained to use only the latencies assigned to it. Bandwidth guarantees held true in all scenarios.

Experiment 2 demonstrated the performance isolation properties of the VCAN approach. Within each partition, latencies are bounded by an analytic worst-case latency, which holds true independent of the behavior within other partitions. Therefore, guarantees regarding bandwidth and latencies can be given for every VCAN in an isolated fashion, which satisfies the main design objective.

6. Implementation and Cost Evaluation

The evaluation above showed the applicability of the VCAN approach in mixed-criticality communication scenarios. Also, the dual token bucket approach significantly outperformed the one with a single token bucket with respect to schedulability. However, another important aspect is implementation cost. Therefore, this section discusses how and at what cost the network virtualization extensions can be implemented in the context of the virtualized CAN controller presented in Chapter III.

Core to the implementation is the admission control module. It extends the existing design by adding another layer of network access arbitration. The transmission of messages is constraint to conform with the virtualization concepts presented above. This section presents an architectural description as well as implementation results in terms of FPGA resource utilization.

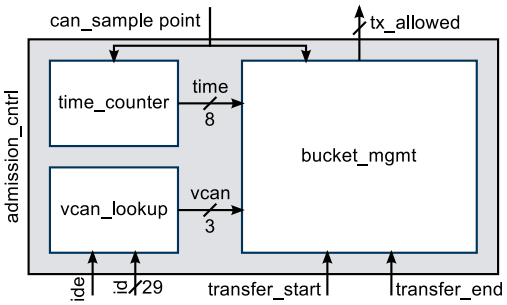


Figure IV.10.: Block diagram of the admission control module [110]

The architecture of the admission control module is depicted in Fig. IV.10. It implements the function of the token buckets used for policing and traffic shaping. As discussed in Section IV.1.2, the admission control is part of a distributed network of token buckets, which need to be synchronized using information retrieved from the CAN bus. Accordingly, all input signals are directly related to layer-1 CAN bus activities. This includes the sampling of a bit on the bus (*can_sample_point*), the start of a frame transmission (*transfer_start*), and the end of a transmission (*transfer_end*). Additionally, *ID* and *IDE* give information regarding the ID of the frame currently in transmission. These signals show the same behavior for any controller connected to the same CAN bus. The only output of the module *tx_allowed* indicates, whether a potential transmission conforms with the admission control rules.

The overall module is composed of three main functions: The *time_counter* and *vcan_lookup* provide a time value and the VCAN ID of the frame currently in transmission to the *bucket_mgmt* module. Using these inputs the *bucket_mgmt* module makes the admission control decision. The VCAN lookup is currently implemented in a way, that the highest priority bits are interpreted as the VCAN ID. Due to the

modular nature of the implementation, it could be extended to support other ID placements. In this exemplary implementation, the *vcan* signal consists of two bits and therefore supports 4 VCANs.

The time provided by the *time_counter* module is used to calculate transmission times of frames on the CAN bus. Therefore, the absolute time is not important, as it is only used for relative measurements. The timer should be dimensioned in the most efficient way, which will still provide accurate measurements. The timer is incremented with each rising edge of *can_sample_point*, leading to a resolution equal to a CAN bus bit time τ_{phy}. A worst-case maximum sized CAN messages is 160 τ_{phy} for 8 bytes payload and maximum bitstuffing. Using an 8 bit wide counter (value range $\{0...255\}\tau_{phy}$), an accurate measurement is guaranteed. If the difference between start and end of frame is positive, it is the actual transmission time. If the difference is negative, an overrun has occurred and the transmission time is obtained by adding the maximum counter value 255.

The majority of the admission control is done within the *bucket_mgmt* module, which implements the token buckets. Depending on the implementation either, one or two token buckets (VCAN-S/VCAN-D) are provided per VCAN. The token buckets are updated continuously. Depending on their reserved rate, a token is added to a bucket after every n_v-th sample point, where n_v is derived from the reserved bandwidth as $n_v = r_v/r_{phy}$. Tokens are deducted from a bucket after every transmission within its corresponding VCAN. The procedure involves accurately measuring the transmission time of a frame. A rising edge in *transfer_start* indicates the start of a frame transmission on the CAN bus. This point in time is recorded as a time stamp. It is used to compute the actual transmission time as difference to the time at the end of the message transmission indicated by *transfer_end*. Finally, tokens are deducted from the token bucket(s) corresponding to the VCAN indicated by the *vcan_lookup* module.

Using the information of the bucket fill level, the module determines whether a VCAN is allowed to transmit a frame on the CAN bus. In the single token bucket implementation, a transmission is allowed if the fill level exceeds the minimum fill level according to (IV.3). Similarly, in the dual token bucket implementation, both token buckets need to have sufficient tokens. The signal *tx_allowed* indicates in parallel the status of every VCAN.

The admission control module was implemented in Verilog, integrated with the design of the virtualized CAN controller, and prototyped using a Virtex-7 FPGA. Resource utilization results after place and route are presented in Table IV.1. The results clearly show that the addition of the network virtualization extensions add little resource overhead to the overall design. Specifically, even the dual token bucket implementation (VCAN-D) amounts to less than 5% additional LUTs and 8.5% additional registers compared to the overall design.

The dual token bucket implementation of the admission control requires 67% additional LUTs and 66% additional Flip-flops. The two versions only differ in the

Table IV.1.: Hardware resource requirements of network virtualization extensions

module	LUTs	FFs
Virtualization Layer	1651	514
Protocol Layer	1151	819
VCAN-S admission control	83	66
VCAN-D admission control	139	110

implementation of the *bucket_mgmt*. Both *time_counter* and *vcan_lookup* are equal in each case. The overhead is caused by the addition of the second token bucket. Parts of the update logic are shared among the buckets, namely the calculation of the amount of tokens to be deducted after a transmission.

7. Summary

The introduction of multi-core technology within automotive embedded systems enables the consolidation of ECUs on shared hardware platforms. The consolidated functions may have different requirements with respect to safety, real-time, security, etc. Such scenarios are often referred to as mixed-criticality. To enable reliable operation in mixed-criticality scenarios, proper isolation is required. While platform virtualization can be used to isolate workloads on ECU level, isolated communication channels are necessary in addition when dealing with networked systems. Currently employed networks are not designed to satisfy such requirements.

For that reason, this chapter introduced a network virtualization approach for controller area network (CAN), which divides the physical network into concurrent virtual partitions. These virtual partitions are called VCANs and provide isolated performance, thus enabling mixed-criticality communication scenarios.

The design of the VCAN approach focuses on two aspects. First, a naming scheme was derived that divides the ID space up into multiple partitions. This is achieved by addition of a VCAN tag within the ID itself. Secondly, an admission control scheme was designed that enforces bandwidth restrictions within each VCAN based on a token bucket policing. The admission control is implemented in a distributed fashion, but achieves consistent admission decisions through layer-1 clock synchronization.

Two versions of the admission control were implemented using either one or two token buckets. The dual token bucket uses the second bucket for traffic shaping, which allows to control the transmission rate during bursts in addition to long term bandwidth policing. Thus, the two main performance metrics - bandwidth and latency - can be controlled individually.

Real-time analyses were presented for both approaches. The analytic framework allows to compute bounded worst-case latencies for a given VCAN configuration and traffic scenario. Importantly, the computation of latencies within one specific VCAN does not require any knowledge of the traffic constellation in other partitions. This demonstrates the temporal isolation property required for mixed-criticality operation.

The VCAN approaches were compared to state of the art solutions with respect to their ability to meet deadline requirements. Currently, safe operation can only be achieved using criticality-monotonic priority assignments. However, this approach is inefficient and only produces schedulable configurations for low network utilizations smaller than 50%. Using the VCAN approach with a single token bucket policing, more than have of all tested message sets were schedulable at 75% utilization. The addition of a second token bucket further increases the efficiency of the approach, resulting in more than 50% schedulable message sets at 88% utilization. The efficiency with respect to bandwidth utilization and performance isolation among partitions make VCAN a suitable approach for mixed-criticality communication scenarios.

Additional measurements were conducted using a cycle accurate simulation to de-

termine typical behavior as well as to demonstrate the performance isolation property. Two experiments were conducted: One using normal operation with cyclic message transmissions in all VCAN. In another experiment, all but one VCANs flooded their VCAN partition with messages. During both experiments, analytically computed latency bounds were not exceeded.

Finally, a Verilog implementation of the VCAN admission control was presented. This module was implemented for integration with the virtualized CAN controller, thus enabling virtual CAN controllers to transmit on different VCANs. Using FPGA prototyping, an additional hardware overhead of ca. 4.5% in logic and 8% in registers was estimated.

V. AVB Ethernet Integration

AVB Ethernet is a new networking technology that combines high bandwidth with real-time capability. It is an important technology to satisfy the communication demand of future, multi-core based architectures. Background information regarding AVB Ethernet was introduced in Section II.1.4.

While many car manufacturers see AVB Ethernet as a promising technology to serve as a backbone network in future automotive embedded systems, it has not yet been commercially integrated. Thus, many open questions remain regarding how AVB Ethernet can be integrated into existing and highly legacy dependent automotive systems.

In the following, two important questions will be addressed: How to interface legacy bus systems (specifically CAN) with AVB Ethernet and how to design and integrate AVB Ethernet controllers into CPU host systems.

1. CAN to AVB Ethernet Gateway

Ethernet is not expected to unify communication in automotive embedded systems and to replace all legacy communication buses. This is partly due to the higher cost of Ethernet compared to simpler technologies like CAN and LIN, but also due to the high amount of legacy components used in the automotive domain [22]. Thus, AVB Ethernet will coexist and communicate with existing networking technologies. In particular, CAN is expected to remain an essential part in automotive electronics long after the introduction of Ethernet due to its low cost, high reliability, and wide adoption.

While CAN and AVB Ethernet have to coexist, they differ strongly in nearly every aspect. Packet size and bandwidth differ in two orders of magnitude. CAN is a shared bus using a strict priority arbitration scheme - AVB Ethernet is a full-duplex switched network using a credit-based shaper algorithm at egress ports.

Nevertheless, their coexistence requires an efficient way of interfacing both technologies while maintaining real-time capability. While no standardized way of forwarding exists, IEEE P1722 citeieee2015p1722 describes a method of encapsulating CAN messages within AVB Ethernet frames. Fig. V.1 shows an AVB Ethernet frame containing multiple CAN frames within its payload.

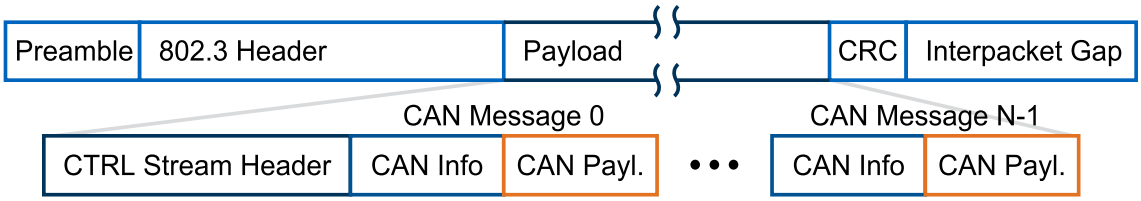


Figure V.1.: Encapsulation of CAN frames in AVB Ethernet frames [115]

Encapsulation of multiple CAN frames within a single AVB Ethernet frame is an important aspect in achieving efficient forwarding. The framing overhead depending on the number of CAN frames within one AVB Ethernet frame N_{can} is depicted in Fig. V.2. As reference, the framing overhead of maximum length CAN frames are given with maximum and minimum bitstuffing, respectively. The overhead using a one-to-one mapping of CAN to Ethernet frames exceeds 600%, but decreases proportional to $1/N_{can}$ and eventually converges to an overhead similar to CAN.

Fig. V.2 assumes that the length of an AVB Ethernet frame encapsulating N_{can} CAN frames is given as

$$L_{avb} = (336 + 128N_{can})\ bit. \tag{V.1}$$

While the encapsulation method described is useful to reduce framing overheads, it leaves the challenge of finding an efficient and real-time capable forwarding scheme. Existing CAN to Ethernet forwarding schemes are either not applicable to AVB

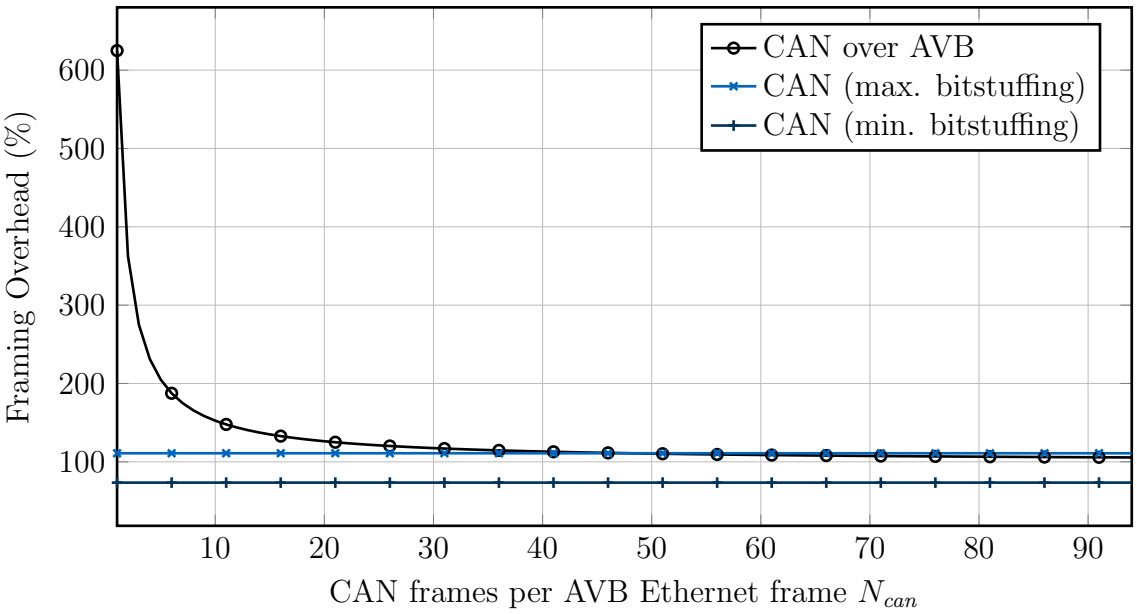


Figure V.2.: Framing overhead when encapsulating multiple CAN frames within an AVB Ethernet frame according to IEEE P1722a

Ethernet or inefficient. Therefore, the following sections introduce and evaluate a concept for a CAN to AVB Ethernet gateway that introduces scheduling as an important mechanism to improve the efficiency.

Parts of this section have previously been published in [115]. Here, an extended evaluation as well as a method for optimal gateway configuration using EDF are presented in addition.

1.1. Concept of the Gateway Forwarding Strategy

The gateway presented here is designed to interface a CAN bus with an AVB Ethernet network. It has to overcome the protocol mismatch between these technologies and provide efficient and real-time capable forwarding. Main focus here is the forwarding direction from CAN towards AVB, as this direction poses the most challenges. In the opposite direction, CAN messages have to be unpacked from the AVB Ethernet packet and queued for transmission on the CAN bus. Here, the CAN arbitration scheme dictates the way in which these messages are transmitted on the bus.

The forwarding mechanism is constraint by the following design goals:

1. **Real-time capability:** The forwarding of messages must successfully conclude a finite amount of time after arriving at the gateway. CAN messages are send cyclic at rates T_m or equivalently with a minimum inter-arrival time. The gateway is designed to guarantee implicit deadlines, meaning deadlines

are equivalent to the respective cycle time of a message $D_m = T_m$. This guarantees that at any time, at most one instance of each message is residing within the gateway buffers. If the timing constraints for all messages are guaranteed to be fulfilled, the system is considered schedulable.

2. **Efficiency:** Schedulability should be achieved using minimal resources. Specifically, CAN over AVB streams should require minimal bandwidth. Key is not the actual bandwidth usage, but rather the bandwidth reservation. Reserved bandwidth is not available for other real-time capable streams even if left unused. In addition, the bandwidth reservation of a stream directly affects fan-in delays experienced by same priority streams and blocking of lower priority streams [46].

The following gateway design is based on the assumption that all CAN messages arriving from one CAN bus are forwarded in a single stream, which is multi-cast towards receiving nodes with the AVB Ethernet network. Many of the sensor information like the cars speed or geolocation are relevant to nearly all functional domains, thus justifying the multi-cast approach. Additionally, even if each CAN message was needed at specific nodes only, multi-casting might still be most efficient, because a single stream allows better accumulation and scheduling of CAN frames into AVB Ethernet frames.

It should be noted that the design and analyses focus on CAN communication and gateway forwarding latencies. Additional communication latencies arise within the AVB network itself. Such latencies can be calculated using the methods described in Section II.1.4.4.

AVB data transmission happens in the form of streams. These streams are subject to a rigid traffic policing and shaping. A CAN to AVB Ethernet gateway represents an AVB talker endpoint. At such talker endpoints, each outgoing stream is shaped in a way that leads to an essentially cyclic sending behavior. The interval between two CAN over AVB frames (i.e. the cycle time) will be T_{avb}.

Fully utilized systems are hardly ever schedulable. Thus, an overprovisioning of resources is necessary. Therefore, an overreservation factor OR is introduced, which describes the relative amount of additional bandwidth reservation. This overreservation is realized through a reduction of the sending interval T_{avb}

$$T_{avb}(OR) = T_{avb}\,(0)\,/\,(1 + OR)\,. \tag{V.2}$$

For example, $OR = 100\%$ means that the cycle time is equal to have of the cycle time without overreservation $T_{avb}\,(0)$, thus doubling available resources. The sending interval without overreservation $T_{avb}\,(0)$ depending on the number N_{can} of CAN frames encapsulated in a single AVB Ethernet frame can be calculated as

$$T_{avb}\,(0) = N_{can} / \sum_{\forall k \in \mathcal{M}_{fwd}} \frac{1}{T_k}. \tag{V.3}$$

Using the definition of the frame length L_{avb} from (V.1), the resulting bandwidth reservation of the CAN over AVB stream is

$$bw_{avb,res}(N_{can}, OR) = L_{avb}/T_{avb}. \tag{V.4}$$

The two principles of aggregation of CAN frames within a shared AVB Ethernet frame and overreservation represent an important trade-off within the design space of the gateway. Increasing N_{can} at a constant level of OR reduces framing overheads and thus the necessary bandwidth reservation, but leads to increased latencies. On the other hand, overreservation behaves the opposite way. An increase in overreservation decreases latencies due to the decreased release interval T_{avb}, but increases the bandwidth reservation. For a given traffic scenario, the optimal configuration of these two parameters must be found with respect to the previously stated design goals of real-time capability and efficiency.

In efficient configurations, overreservations might not be high enough to guarantee that all frames, which arrive within one cycle, can be forwarded all the time. This means that not all CAN messages buffered within the gateway can be forwarded when an AVB Ethernet frame is released. Therefore, a method is needed to determine a subset from the set of buffered messages, which should be forwarded. This task will be done by scheduler, for which FIFO, strict priority and earliest deadline first will be evaluated in the following sections.

1.1.1. FIFO Forwarding

FIFO is the simplest forwarding principle researched here. Messages arriving from the CAN bus are stored within the buffers of the gateway. When a CAN over AVB Ethernet frame is about to be released, it may not be possible to forward all buffered CAN messages. FIFO selects the N_{can} messages which have been stored the longest waiting for transmission. These will be merged into an Ethernet packet and released towards the AVB Ethernet network. Because messages can be stored for multiple forwarding cycles, determining the forwarding latency is non-trivial. It will be discussed below.

Forwarding latencies are analyzed based the queuing model depicted in Fig. V.3. It shows the release of CAN messages for CAN bus transmission, the arrival of CAN messages at the gateway, and the departure of Ethernet frames. The analysis is based on network calculus [41], with which worst-case arrival and departure patterns of the traffic are computed. Once these patterns are derived, the worst-case latency is given as the maximum vertical deviation between the curves.

To cover the worst-case for the arrival of messages at the gateway queue, a critical instant is assumed to occur at $t = 0$, i.e. all CAN messages are released synchronously at this time. This way, the cumulative amount of messages that are queued for transmission on the CAN bus within the interval $[0, t)$ is maximized.

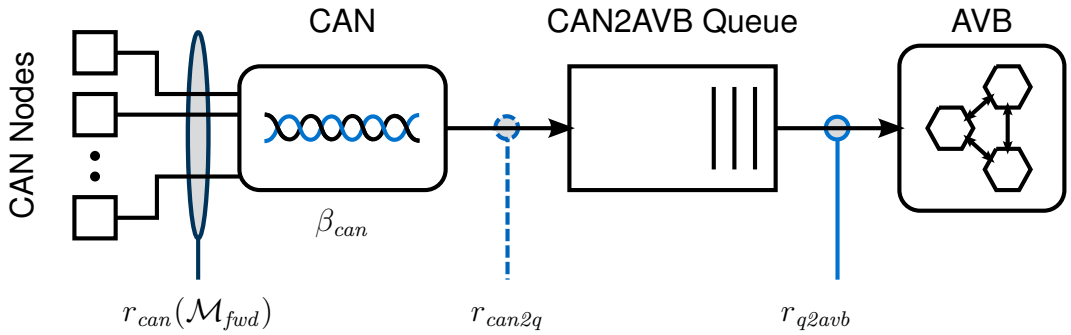


Figure V.3.: Queuing model of the CAN-AVB gateway [115]

This includes all message instances, which are either released or transmitted during this interval. It is computed as

$$r_{can}(\mathcal{M}_{fwd}, t) = \sum_{\forall k \in \mathcal{M}_{fwd}} \lceil (t + R_k) / T_k \rceil . \tag{V.5}$$

The equation assumes cyclic release of CAN messages. The amount of CAN messages released or transmitted in the interval $[0, t)$ is maximized, if the CAN messages transmitted at the beginning of the interval have experienced a worst-case delay. This means, that a second instance of such messages is queued for CAN transmission shortly afterwards. The reduction of the inter-arrival time of the first two instances of each CAN message m is thus equal to its worst-case response time R_m.

While messages are assumed to be released synchronously, they arrive one after another at the gateway. On the CAN bus, only one message can be transmitted at a time. The limited transmission speed thus has to be considered in the overall model. The maximum amount of message arrivals in the interval $[0, t)$ is given as

$$\beta_{can}(t) = \lceil t / C_{min} \rceil . \tag{V.6}$$

Using network calculus [41], (V.5) and (V.6) can be used to determine a worst-case egress pattern from the CAN bus and therefore a worst-case arrival pattern at the gateway. Equation (V.6) represents a service curve, which can be convoluted with the arrival curve represented by (V.5) to determine the corresponding departure curve. Network calculus is based on min-plus algebra, and thus the convolution is also a min-plus convolution [41]. According to the definition of this min-plus convolution, an upper bound for the cumulative arrival of messages at the gateway queue can be calculated as

$$r_{can2q}(t) = \left(r_{can}\left(\mathcal{M}_{fwd}\right) \otimes \beta_{can}\right)(t) \tag{V.7}$$
$$= \inf_{0 \le \tau \le t} \left\{ r_{can}\left(\mathcal{M}_{fwd}, \tau\right) + \beta_{can}\left(t - \tau\right) \right\}. \tag{V.8}$$

On the egress side of the gateway, N_{can} messages are removed from the queue and transmitted across the AVB Ethernet network in cyclic fashion in an interval of T_{avb}. In the worst-case scenario, the last frame has been sent an infinitesimal time before $t = 0$. Therefore, the cumulative number of messages removed until time t is

$$r_{q2avb}(t) = \lfloor t / T_{avb} \rfloor \, N_{can}. \tag{V.9}$$

Equations (V.5), (V.7), and (V.9) are illustrated in Fig. V.4 for an exemplary configuration. In this configuration, 50% of messages from a CAN bus with a bandwidth of 500 kbit/s are forwarded to the AVB network. The gateway packs $N_{can} = 10$ CAN frames within each AVB Ethernet frame and uses 50% overreservation. To allow fair comparison, the same scenario and configuration will also be used for the other scheduling mechanisms. The distribution of cycle times corresponds to the scenario introduced in Fig. III.13.

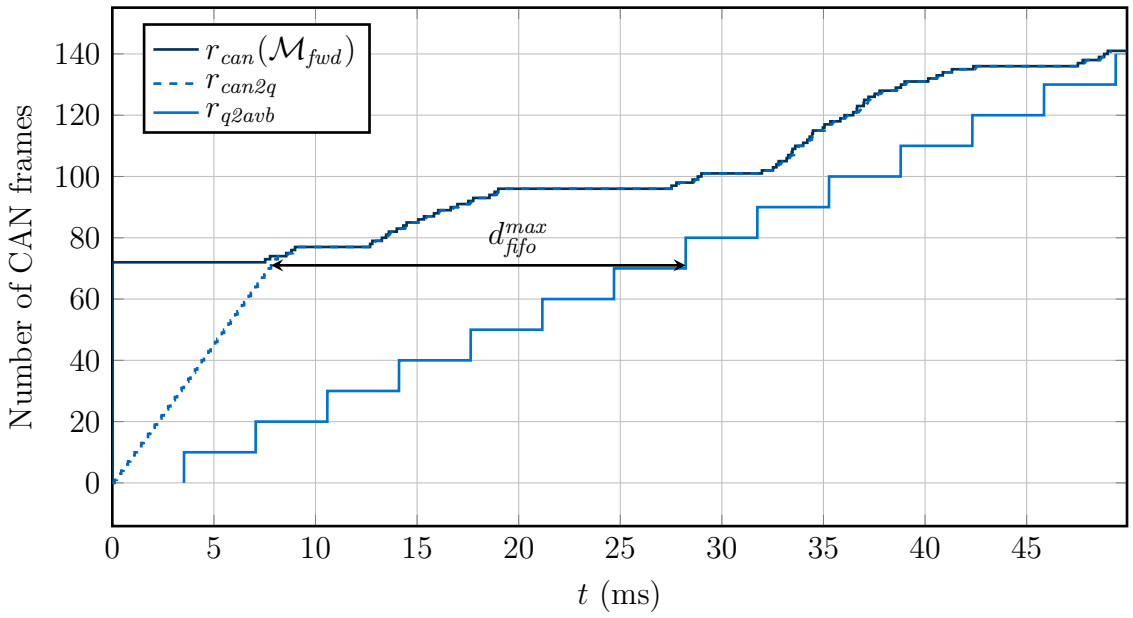


Figure V.4.: FIFO forwarding: $N_{can} = 10$, $OR = 50\%$ [115]

The vertical discrepancy between r_{can2q} and r_{q2avb} in Fig. V.4 translates to the backlog observed in such a worst-case scenario. The horizontal discrepency is often called a virtual delay. The maximum horizontal derivation between cumulative

inflow and outflow of messages therefore corresponds to the maximum delay. Within this worst-case scenario, a message engueued at time t is forwarded after

$$d_{fifo}(t) = \inf_{\forall t \geq 0} \{t_d : r_{can2q}(t) - r_{q2avb}(t + t_d) \leq 0\}. \tag{V.10}$$

Therefore, the forwarding delay of any message is guaranteed to be bounded as

$$d_{fifo}^{max} = \sup_{\forall t \geq 0} \{d_{fifo}(t)\}. \tag{V.11}$$

Equation (V.11) is not message specific, but rather gives the same latency guarantee to all messages independent of their priority or deadlines. This makes sense, because the FIFO scheme used does not prioritize certain messages. Therefore, it has to be dimensioned to cater the timing requirements of the highest priority messages. The lack of differentiation means that the scheduler is inefficient. In the example shown in Fig. V.4, the delay is around 20 ms, despite having 50% over-reservation. However, high priority messages often have latency requirements that involve deadlines smaller than 10 ms. To overcome the efficiency issues in FIFO scheduling that stem from a lack of differentiation, scheduling approaches that use static and dynamic prioritization are the focus of the following sections.

1.1.2. Strict Priority (SP) Forwarding

Strict priority is a scheduling mechanism, which uses static priorities determined at design time to make scheduling decisions. It is used e.g. in CAN arbitration. Applied to the gateway at hand, it can be used to select the subset of buffererd CAN messages, that should be forwarded in an AVB Ethernet frame. When a CAN over Ethernet frame is about to be sent, the scheduler must determine the N_{can} highest priority messages. Ideally, these messages are already stored in a priority sorted list.

In contrast to FIFO forwarding, no dynamic information is included in the forwarding decision. However, SP uses static information about each message's priority. This means that the forwarding delay is message dependent. It will be denoted as $d_{m,sp}$. To find an upper bound for this delay, the blocking by higher priority messages within the gateway has to be considered. Any higher priority message arriving while a message m is buffered will overtake this message and thus increase the queueing delay. The worst-case amount of higher priority message instances which block the forwarding of message m can be derived as

$$I_m = \sum_{\forall k \in hp(m)} \left\lceil \frac{d_{m,sp} + R_k}{T_k} \right\rceil, \tag{V.12}$$

with $hp(m) \subset \mathcal{M}_{fwd}$ being the set of forwarded higher priority messages. This equation makes the same assumption on the arrival of CAN messages as (V.5).

Using the definition of (V.12), the queuing delay can be calculated. First, it is assumed that a CAN over AVB Ethernet frame has just been released when message m arrives at the gateway. While message m is buffered, higher priority messages can arrive and delay its forwarding. Once less than N_{can} higher priority messages are buffered when an Ethernet frame is released, message m will be forwarded. Thus, the queueing delay is given by the implicit formulation

$$d_{m,sp} = T_{avb} \left(1 + \left\lfloor \frac{I_m(d_{m,sp})}{N_{can}} \right\rfloor \right). \tag{V.13}$$

Because this equation cannot be solved analytically, a numeric solution is required. This can be achieved e.g. using fixed-point iteration. Using a starting value of $d_{m,sp}^{\mu=0} = T_{avb}$ and substituting (V.12), a solution can be obtained through

$$d_{m,sp}^{\mu+1} = T_{avb} \left(1 + \left\lfloor \frac{\sum_{\forall k \in hp(m)} \left\lceil \frac{d_{m,sp}^{\mu} + R_k}{T_k} \right\rceil}{N_{can}} \right\rfloor \right). \tag{V.14}$$

The equation above can be used to calculate queuing latencies within the gateway for a given configuration. This includes the assignment of priorities towards messages. Yet, the question of how to assign such priorities is not yet addressed. The simplest way of priority assignment is to use CAN priorities. These are assigned to maximize the schedulability of the message set at hand during CAN communication. Messages with small deadlines have high priorities and vice versa. However, using CAN priorities also within the gateway may lead to an over-prioritization of high priority messages. Reassigning priorities within the gateway may improve the overall schedulability of CAN communication and gateway forwarding combined.

In preemptive SP systems, optimal priority assignment corresponds to monotonic assignment of priorities with respect to "deadline minus jitter" (D–J) [116]. However, this is not generally true in non-preemptive SP systems (e.g. CAN) if message or tasks can vary in lengths [32].

With the forwarding mechanism of the gateway, each CAN message occupies a fixed part of an Ethernet frame independent of its length. Thus, the individual message length does not have to considered when calculating blocking times. Accordingly, a (D–J) monotonic priority assignment will be considered the optimal priority assignment (OPA).

To assign optimal priorites, the deadline and jitter of messages arriving at the gateway has to be determined. Messages arrive at the gateway queue and therefore at the SP scheduler earliest after their minimum transmission time C_m^{min} and latest after their WCRT R_m. Thus, the deadline of messages is reduced to $D_m - C_m^{min}$ with a jitter of $R_m - C_m^{min}$. (D–J) monotonic priority assignment therefore translates into a priority assignment inversely proportional $D_m - R_m$ (small value, high priority). The calculation is as follows, assuming D_m^{gw} to be the deadline remainging at the

gateway and J_m^{gw} the jitter of the message arrival at the gateway

$$D_m^{gw} - J_m^{gw} = (D_m - C_m^{min}) - (R_m - C_m^{min}) = D_m - R_m. \tag{V.15}$$

A timing analysis was conducted for the same configuration as presented for FIFO forwarding. The results are illustrated in Fig. V.5. Presented are the cycle time T_m, the WCRT on the CAN bus R_m and the combined delay of CAN bus and gateway forwarding queuing delays for an example message set using CAN priorities and optimal priorities (OPA), respectively.

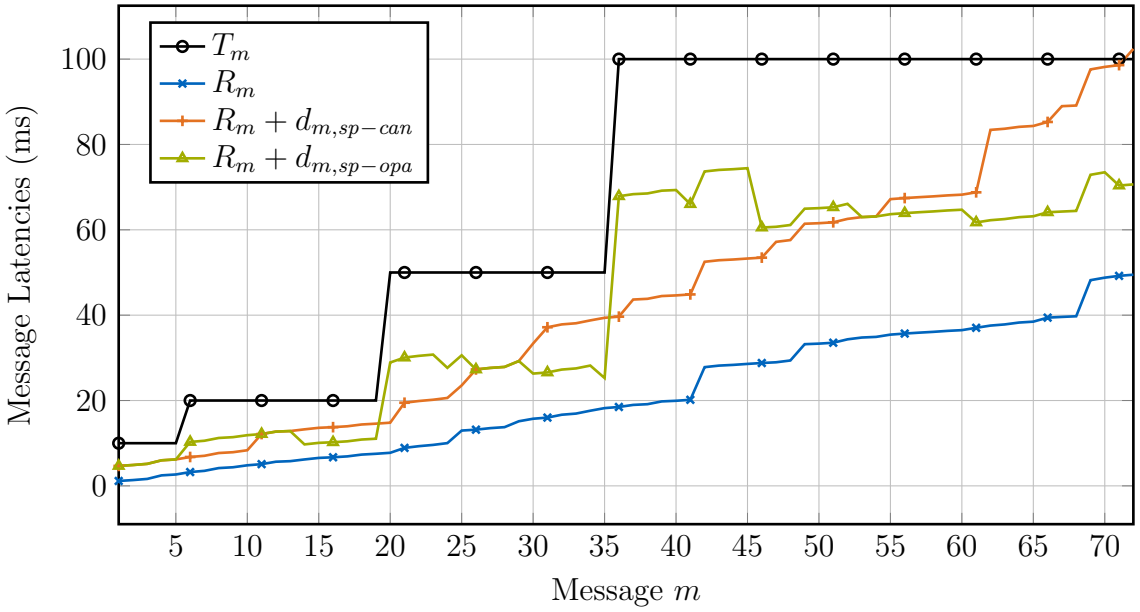


Figure V.5.: Strict priority forwarding delays with CAN and optimal (OPA) priorities, respectively: $N_{can} = 10$, $OR = 50\%$ [115]

The difference in the two priority assignment schemes is evident. Using CAN priorities, latencies increase monotonically. While most messages have significant slack towards their cycle time (and thus deadline assuming implicit priorities), the latency of the lowest priority message exceeds the cycle time. This indicates an inefficient configuration. Using optimal priorities means that messages which had low priority during CAN arbitration can have increased priority in the forwarding to compensate. The result is a latency distribution, which is significantly smoother with respect to the slack messages have to their deadline. Finally, it guarantees all messages to meet their deadlines, therefore improving the overall efficiency of the forwarding scheme. Of course it should be mentioned, that the priority modification adds complexity to the forwarding mechanism, as it requires a look-up table to determine the respective priorities.

1.1.3. Earliest Deadline First (EDF) Forwarding

Using dynamic scheduling methods, it is possible to overcome the efficiency of static schedulers. By using run-time information, scheduling decisions can be optimized. Within this gateway, the time for which a message has been buffered can be used to optimize the scheduling decision. While previously introduced SP scheduling uses no dynamic information, the FIFO forwarding scheme is actually a dynamic scheduling mechanism, even if a simple one. In FIFO forwarding, the decision is only made on basis of the time the messages are queued.

Earliest Deadline First (EDF) is a dynamic scheduling mechanism, which selects the message with the closest upcoming deadline, thus minimizing deadline misses. In the context of the gateway, the forwarding mechanism has to find the N_{can} messages which fit this criterion.

The biggest challenge is to accurately determine the time t_d, at which a message has to be forwarded to meet its deadline. At design time, the overall deadline D_m is known for each message. It describes the time span available after messages initiation until its successful forwarding towards the AVB network. Also available within the gateway is the information, how long a message instance has been buffered. Unfortunately, it is not possible for the receiver of a CAN message to determine when it was queued for transmission. This problem already occurred during the design of the deadline-aware interrupt coalescing scheme presented in Section III.1.5. This lack of information has to be dealt with through a pessimistic approximation. Assuming that t_d is the accurate deadline and a message arrived at the gateway at time t_{gw}, it can be approximated by assuming the CAN message arrived after a worst-case delay R_m on the bus

$$\widehat{t}_d = D_m - (t_{gw} + R_m) \leq t_d. \tag{V.16}$$

To determine whether a gateway configuration is schedulable, existing analysis for EDF has to be modified. Baruah et. al [117] showed that a task set is schedulable on a uniprocessor, if the so called processor demand $h(t)$ is smaller or equal to t. Additionally, a processor demand function is formulated, which equals the added execution time of all jobs that have a deadline until t. To analyze the demand at the gateway, let $h(t)$ be the communication demand function, which describes the number of CAN frames that must be forwarded until t in order to meet their deadline. It can be formulated as

$$h(t) = \sum_{\forall m \in \mathcal{M}_{fwd}} \max \left\{ 0, 1 + \left\lfloor \frac{t - (D_m - R_m)}{T_m} \right\rfloor \right\}. \tag{V.17}$$

The equation assumes that deadlines D_m are reduced by the WCRT on the CAN bus R_m as described by (V.16).

In addition, let $g(t)$ be the communication service function, which describes the number of CAN frames guaranteed to be forwarded via AVB Ethernet at t. As this

function describes a guaranteed service, a worst-case scenario has to be considered. It is assumed that an Ethernet frame has been released an infinitesimal amount of time before $t = 0$. Afterwards, a constant flow of Ethernet frames containing N_{can} CAN frames is forwarded cyclic at a rate corresponding to the cycle time T_{avb}. The cumulative service function is this given as

$$g(t) = N_{can} \left\lfloor t/T_{avb} \right\rfloor . \tag{V.18}$$

The condition for schedulability is simply that the communication service has to be greater than the demand at all times. Thus, the system is considered schedulable, if the following condition holds

$$h(t) \leq g(t). \tag{V.19}$$

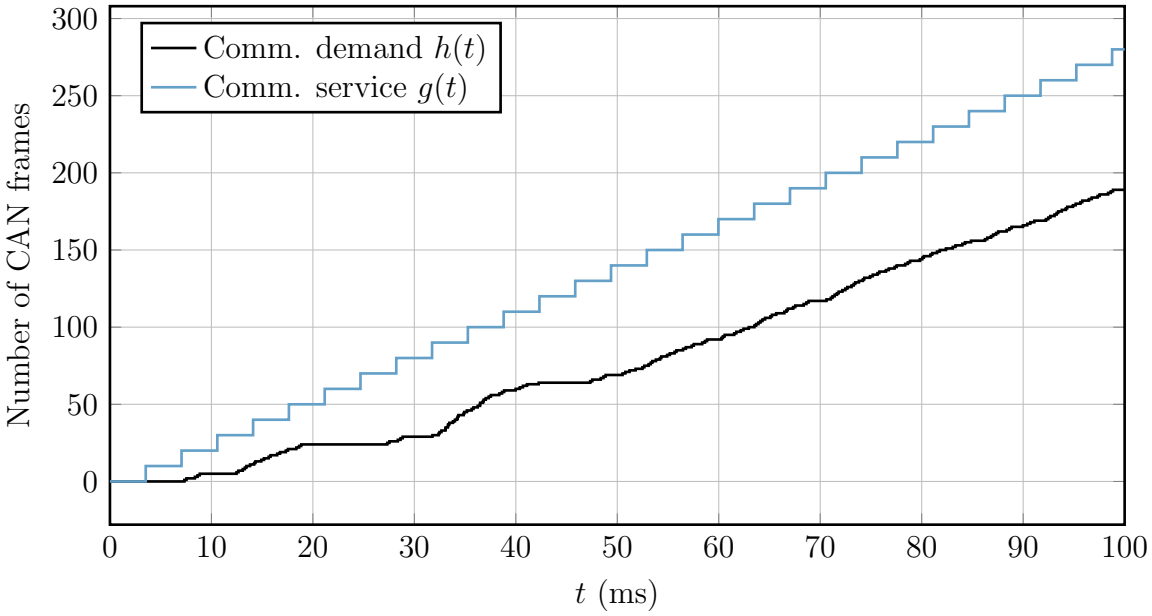


Figure V.6.: EDF forwarding schedulability test: $N_{can} = 10$, $OR = 50\%$ [115]

Using the analysis presented above, a schedulability test was conducted using the same scenario and gateway configuration as in the examples for FIFO and SP forwarding. The results are presented in Fig. V.6.

The offered communication service $g(t)$ is a step function, where the height of the step is equivalent to the number of CAN frames within an AVB frame and the width corresponds to the cycle time. Because the demand is lower than the service at any time, schedulabiltiy can be guaranteed. In fact, it can be seen that communication service could be reduced in order to improve the efficiency of the system. This can be done by decreasing the step height or increasing the step length of the communication service.

1.2. Evaluation

The design of the gateway is targeted at the timely forwarding of messages before a designated deadline using minimal resources. Specifically, the necessary bandwidth reservation for the outgoing AVB stream containing CAN frames should be minimal. Thus, the main objective of this evaluation is to determine the efficiency of various configurations with respect to their ability to provide sufficient schedulability. The design space covers the number of CAN frames encapsulated in an Ethernet frame, the amount of overreservation used, the ratio of forwarded CAN messages, and all of the forwarding mechanisms introduced above (FIFO, SP with CAN priorities and with optimal priorities, and EDF). First, the exploration scenario is introduced, followed by the respective results.

1.2.1. Scenario

The scenario used here is based on the same statistical distribution of CAN messages, which was used in the previous chapters. In summary, CAN messages are released cyclic towards the CAN bus with cycle time T_m. Cycle times are chosen from harmonic sets corresponding to $T_m \in \{10, 20, 50, 100\}ms$. These cycle times are selected randomly for each message with probability 4.8%, 14.3%, 33.3% and 47.6%, respectively. The overall utilization of the CAN bus is fixed at $U = 80\%$ throughout all of the evaluation. The CAN bus is operating at a bandwidth of 500 kbit/s. The share of forwarded messages (not all messages may require forwarding to the Ethernet backbone) will be called Q_{fwd}. In the initial evaluation, it is assumed that half of the CAN traffic is forwarded to the CAN bus. Later on, the effect of Q_{fwd} is analyzed. The subset of forwarded messages is $\mathcal{M}_{fwd} \subset \mathcal{M}$. The quotient of forwarded traffic can be calculated as

$$Q_{fwd} = \frac{U(\mathcal{M}_{fwd})}{U(\mathcal{M})}, \tag{V.20}$$

which corresponds to the relative part of the bandwidth forwarded to the AVB network using the definition of the CAN utilization U in (II.2). How much of the CAN data is forwarded depends on its relevance to other functional domains. Values for this quotient are chosen as $Q_{fwd} \in \{30, 40, ..., 70\}\%$.

Four different forwarding techniques are considered including first in, first out (**FIFO**), strict priority based on CAN IDs (**SP-CAN**) and with optimal priority assignment (**SP-OPA**), and earliest deadline first (**EDF**). For comparison to state-of-the-art forwarding mechanisms, a scheme adapted from [54] will also be included in the analysis. In contrast to the gateway presented here, this work uses a forwarding scheme that guarantees forwarding of all frames every time a CAN over Ethernet frame is released. It is thus referred to as complete release (**CR**).

The number of CAN frames encapsulated in one AVB Ethernet frame is explored in the range of $N_{can} \in \{1, 2, ..., 35\}$. In addition, overreservation of bandwidth is

allowed in the range of $OR \in \{0, 10, ..., 400\}\%$.

1.2.2. Schedulability Results

A message set is considered schedulable, if and only if all forwarded messages $m \in \mathcal{M}_{fwd}$ are guaranteed to be forwarded before their deadline D_m. To provide results that cover a wide range of traffic scenarios, 10,000 message sets were generated and tested for their schedulability within different configurations. The schedulability S for each configuration is defined as the ratio of message sets deemed schedulable using the analyses developed in Section V.1.1. The schedulability is a function of the gateway configuration and thus dependent on the scheduling algorithm used for forwarding, the number of CAN frames encapsulated within an AVB Ethernet frame N_{can}, and the overreservation OR of the outgoing Ethernet stream.

Fig. V.7 presents all Pareto optimal configurations found in the exploration differentiated by the schedulers used for forwarding. Here, a Pareto optimal configuration achieves the highest level of schedulability S at a specific bandwidth reservation. No other configuration of N_{can} and OR achieved higher schedulability using less or an equal amount of bandwidth. Therefore, a scheduling mechanism is most efficient,

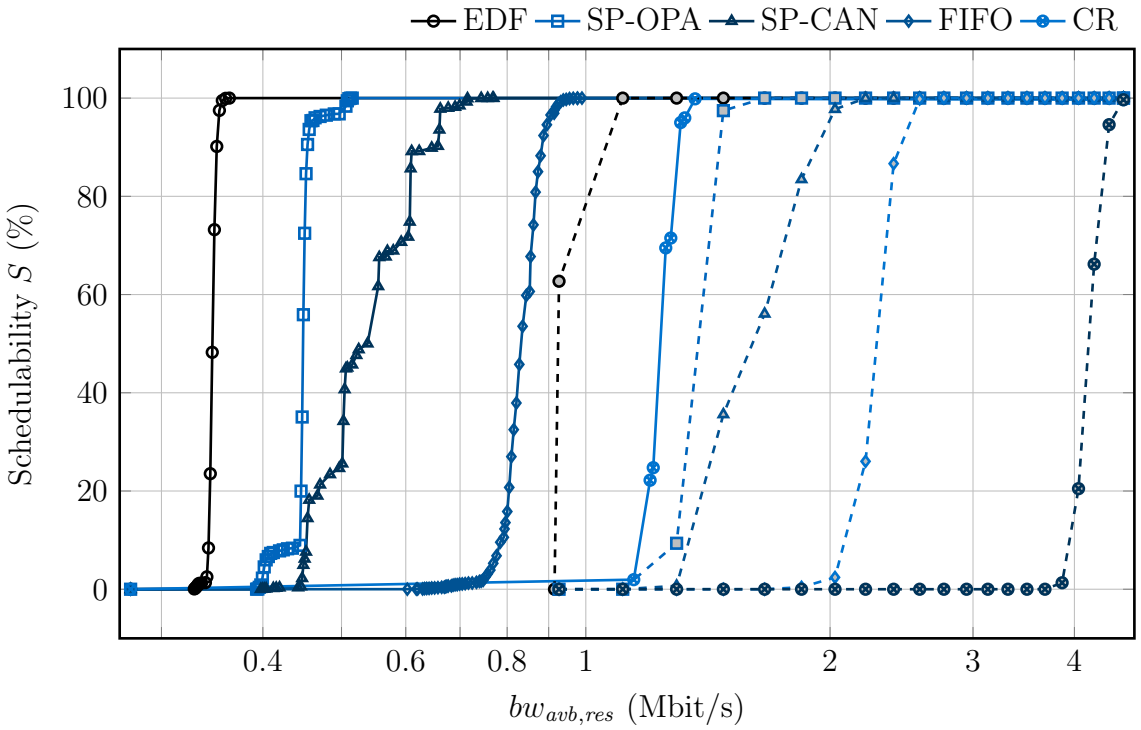


Figure V.7.: Schedulability for optimal gateway configurations with frame aggregation (solid lines, $N_{avb} > 1$) and without (dashed lines, $N_{avb} = 1$) [115]

if the area under schedulability curve is maximal. It can be used to compare the schedulers ability in real-time capable forwarding. For reference, the schedulability of configurations restricted to $N_{can} = 1$ is included. In these cases, no frame aggregation is used.

The results clearly indicate a significant improvement in efficiency over state-of-the-art gateway forwarding mechanisms (CR), where even a simple FIFO forwarding is able to reduce the necessary bandwidth reservation for real-time forwarding. This is expected, as CR uses a level of overreservation that guarantees that the gateway buffer is completely released every time an Ethernet frame is sent. Using such overreservation, the scheduling algorithms are effectively irrelevant, as all messages are selected for forwarding anyway. The advantage of the approach here is that because not all buffered messages have to be forwarded, more efficient configurations at lower levels of OR are available.

Additionally, it can be seen that bandwidth efficiency is directly linked to the complexity of the scheduler. The scheduling mechanisms provide increasing results for increasing scheduler complexity. Specifically, EDF is the dominant scheduler, which offers best performance at any bandwidth level. This is expected, as EDF is optimal schedule in non-preemptive single resource systems [118], i.e. if a configuration is not schedulable using EDF, it is not schedulable at all. Similarly, the optimality of reordered priorities in SP forwarding guarantees that performance is equal or better than the use of CAN priorities.

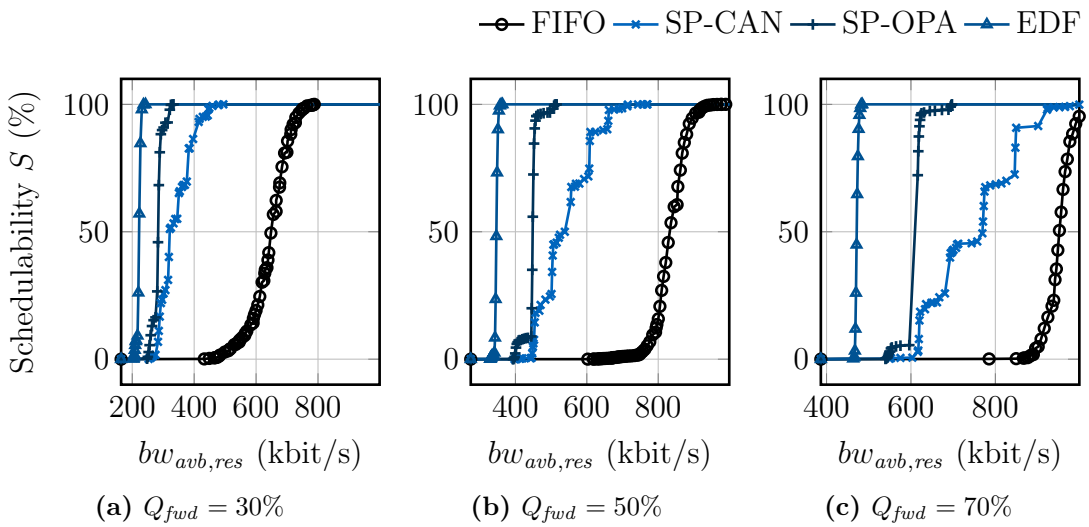
Throughout the evaluation, SP forwarding has outperformed FIFO. In theory, FIFO could provide better performance SP in a few unrealistic scenarios, where messages arrive at the gateway with similar deadlines. Then,it makes sense to forward messages in the order of their arrival. Because automotive latency requirements are usually diverse with typical deadlines ranging between 10 ms and 100 ms, the lack of message specific prioritization leads to bad real-time performance in FIFO forwarding.

The results can be interpreted in a quantitative manner by considering the bandwidth reservation necessary to achieve schedulability $S \geq 50\%$. Compared to CR, FIFO forwarding is already able to achieve 33 % bandwidth reduction compared to CR. Using the most complex scheduler, EDF, a total reduction around 72.21% is achieved compared to CR. In fact, it can be seen that scheduling of messages within the gateway has a similar effect as the accumulation of CAN messages within a single Ethernet packet.

While EDF has shown the best performance in the evaluation, it is also the most complex mechanism. In certain scenarios, e.g. when the gateway is implemented in hardware, a FIFO scheduler might be preferred due to its low resource requirements. Also, other non-functional requirements may be important in certain application scenarios. In error scenarios, where the CAN load is increased due to retransmissions, SP scheduling can be more robust than EDF scheduling.

Complex real-time systems can not always be modeled perfectly in a sense that

the derived timing bounds are tight. To avoid false positive results in real-time analyses, they have to be pessimistic. In this gateway scenario, worst-case scenarios for the CAN bus and gateway forwarding are considered. The analyses are derived in a similar fashion and therefore also introduce a similar degree of pessimism. For example, it is assumed for all mechanisms that one message instance can experience both worst-cases (CAN and gateway) during one transmission. Because such assumptions are made for all schedulers, comparability among them is guaranteed. Also, despite pessimism, significant improvements in schedulability are achieved.



(a) $Q_{fwd} = 30\%$ **(b)** $Q_{fwd} = 50\%$ **(c)** $Q_{fwd} = 70\%$

Figure V.8.: Schedulability for optimal gateway configurations with different ratios of forwarded CAN traffic

Fig. V.8 shows schedulability results for multiple levels of Q_{fwd}. The results generally indicate that real-time capability is easier to achieve when large fractions of the CAN traffic Q_{fwd} are forwarded. Because lower levels of Q_{fwd} require less bandwidth, the sending interval of AVB frames is larger without overreservation. This can be coped with by sending smaller frames but more often, or by increased overreservation. Both measures increase the necessary bandwidth.

The rate at which CAN frames can arrive at the gateway in a short time is independent of Q_{fwd}, yet the outgoing bandwidth is not. Reducing a backlog that occurred due to back to back arrival of CAN frames is therefore harder to deal with at small values of Q_{fwd}. Because FIFO forwarding lacks the ability to prioritize messages in such scenarios with high backlog, it suffers more from small values of Q_{fwd} than the other forwarding mechanisms.

Another interesting parameter is the number of accumulated CAN frames within an AVB Ethernet frame N_{can} and its influence towards the schedulability of the system. To evaluate the schedulability at a constant value of N_{can} for multiple levels of overreservation OR, a weighted schedulability [107] can be used. Here, a greater

weight is given to configurations with low bandwidth, as this is more demanding for the scheduler and more desirable in application scenarios. The weighted schedulability is computed as

$$wS(Q_{fwd}, N_{can}) = \frac{\sum\limits_{\forall OR \in \mathcal{OR}_{wS}} \frac{S(OR)}{bw_{avb,res}(OR)}}{\sum\limits_{\forall OR \in \mathcal{OR}_{wS}} 1/bw_{avb,res}(OR)}. \tag{V.21}$$

The weighted schedulability wS represents a weighted average along multiple values of reserved bandwidth. The maximum bandwidth in this evaluation is limited to

$$bw_{max} = bw_{avb,res}(Q_{fwd}, N_{can}^{max}, OR^{max}) \tag{V.22}$$

This allows fair comparison among different values of N_{can}. Accordingly, levels of overreservation, which provide bandwidth smaller than bw_{max} are included in the set

$$\mathcal{OR}_{wS}(Q_{fwd}) = \{OR : bw(OR) \leq bw_{max}(Q_{fwd})\}, \tag{V.23}$$

which is used in (V.21).

Fig. V.9 shows weighted schedulabilities for each scheduling mechanism. Also, curves for multiple ratios of forwarded traffic are included. The results indicate that there is not a single optimum with respect to N_{can}. Rather, scheduling algorithms and the traffic configuration have to be considered. FIFO forwarding is different to all other schedulers, as the optimal value of N_{can} does not significantly depend on the ratio of forwarded traffic. Encapsulating 10 CAN frames within a single Ethernet frame seems to provide consistent results.

1.3. Optimal EDF Configuration

An optimal forwarding configuration guarantees real-time capability using minimal resources. Within the evaluation above, optimal configurations were found with respect to a discrete set of configurations analyzed in the exploration. While the number of CAN frames encapsulated within an AVB Ethernet frame is discrete, the overreservation of a stream is a continuous parameter, which was quantized to enable the exploration.

In a real-world setting, it would be useful to analytically determine an optimal configuration for a given traffic scenario. A configuration is considered to be optimal here, if it guarantees schedulability and there is no other configuration, which uses less resources and still provides schedulability. The resource considered in this context is the bandwidth reserved for the AVB stream carrying CAN frames. The

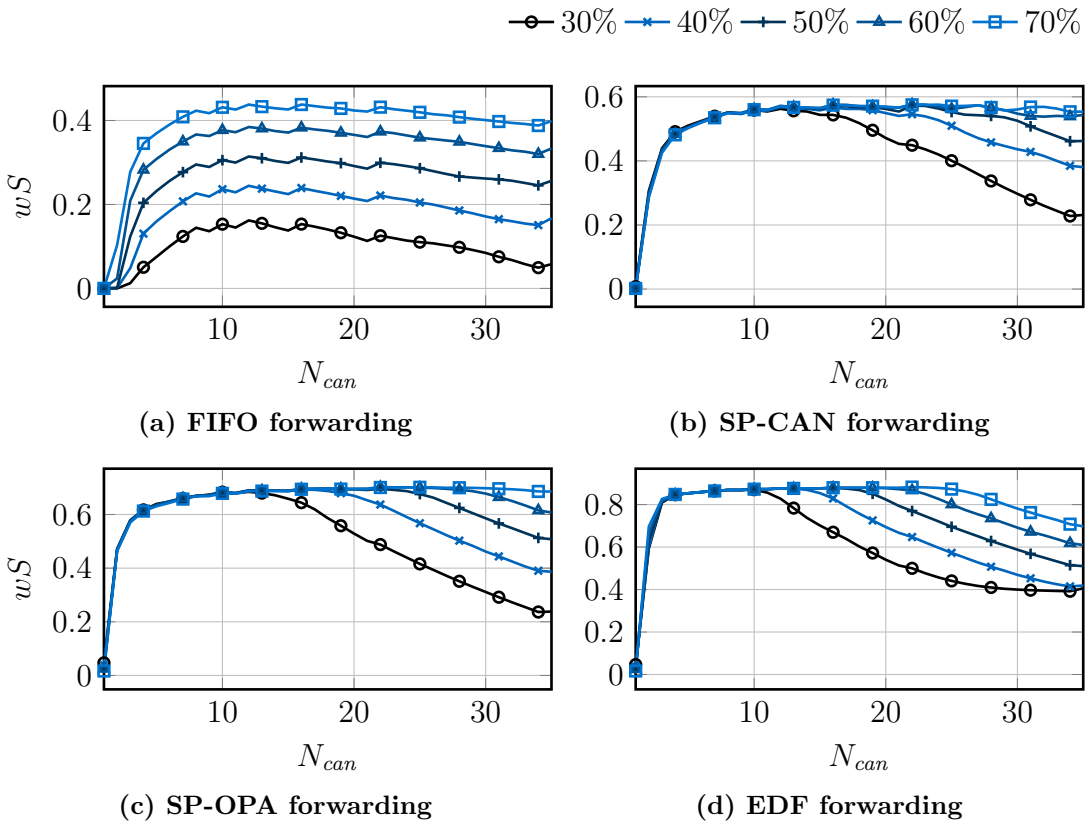


Figure V.9.: Weighted Schedulablity of the gateway at multiple ratios of forwarded traffic Q_{fwd} between 30% and 70%

two parameters which can be tuned are the number of CAN frames fitted into an AVB Ethernet frame N_{can} and the overreservation of the stream OR.

Section V.1.1.3 introduced a schedulability analysis for EDF forwarding. For a given traffic scenario and gateway configuration, it allows to determine whether all messages are guaranteed to meet their deadline. According to (V.19), real-time forwarding is guaranteed if the communication demand $h(t)$ is smaller or equal the communication service $g(t)$ at all times. While $h(t)$ is given, $g(t)$ can be tuned using N_{can} and OR.

Figure V.10 visualizes the part of a schedulability test, which contains the communication demand at the edge of schedulability. The communication demand function $h(t)$ is equal to the one shown in Figure V.6. Additionally, the communication service without overreservation $g_0(t)$ and an optimal configuration $g_{OR}(t)$ are presented.

If a configuration is not schedulable for a given N_{can}, the overreservation can be increased to reduce the sending interval of AVB Ethernet frames. First, the horizontal deviation between demand and available service for points at which the schedulability analysis fails will be observed (see Figure V.10). The next point in

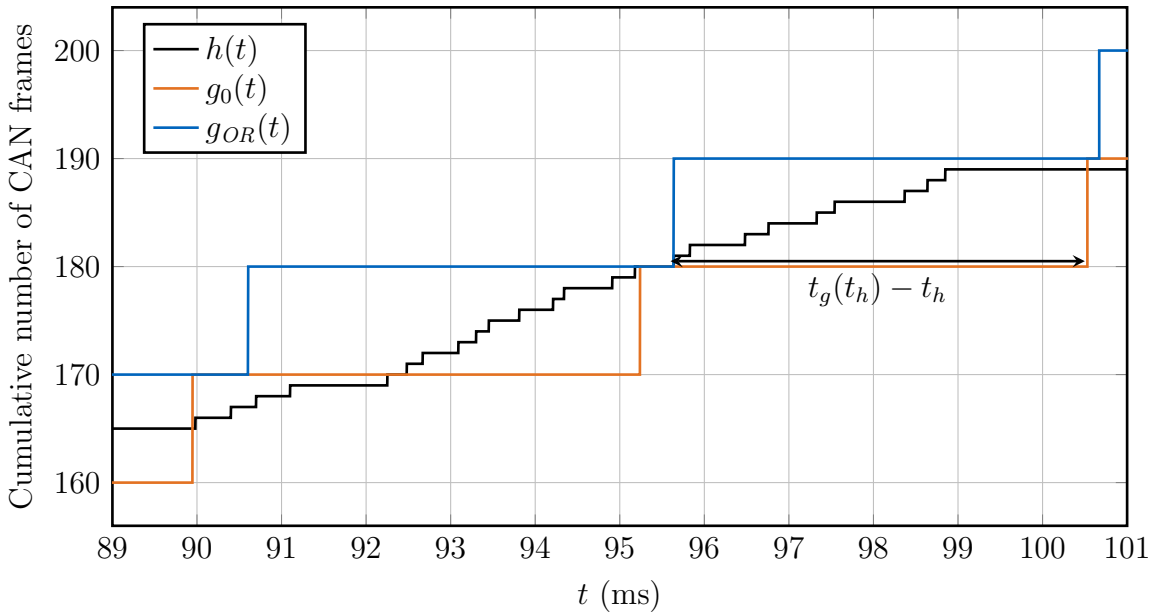


Figure V.10.: Detail of EDF forwarding schedulability test without and with optimal overreservation (here: $OR = 5.11\%$) using $N_{can} = 10$

time, which offers sufficient service to satisfy the demand at t_h is

$$t_g(t_h) = \inf\{t : g_0(t) \geq h(t_h)\} = \left\lceil \frac{h(t_h)}{N_{can}} \right\rceil T_{avb}, \tag{V.24}$$

for any time t_h with $h(t_h) > g_0(t_h)$.

The minimal overreservation, which will guarantee the schedulability test to succeed at t_h, has to provide the service $g_0(t_g)$ already at t_h. Therefore, it should hold that

$$g_{OR}(t_h) \stackrel{!}{=} g_0(t_g). \tag{V.25}$$

Substituting the definition of $g(t)$ from (V.19), it can be concluded that

$$\left\lfloor \frac{t_h(1+OR)}{T_{avb}} \right\rfloor = \left\lfloor \frac{t_g}{T_{avb}} \right\rfloor. \tag{V.26}$$

The floor function on the right side of (V.26) can be neglected, because t_g is the first point in time after a jump in service. In order to just satisfy the communication demand at t_h, the floor function of g_{OR} must also have a jump at t_h. Therefore, the ceil functions can be neglected to derive an optimal value for OR.

The optimal overreservation to satisfy communication requirements at all times

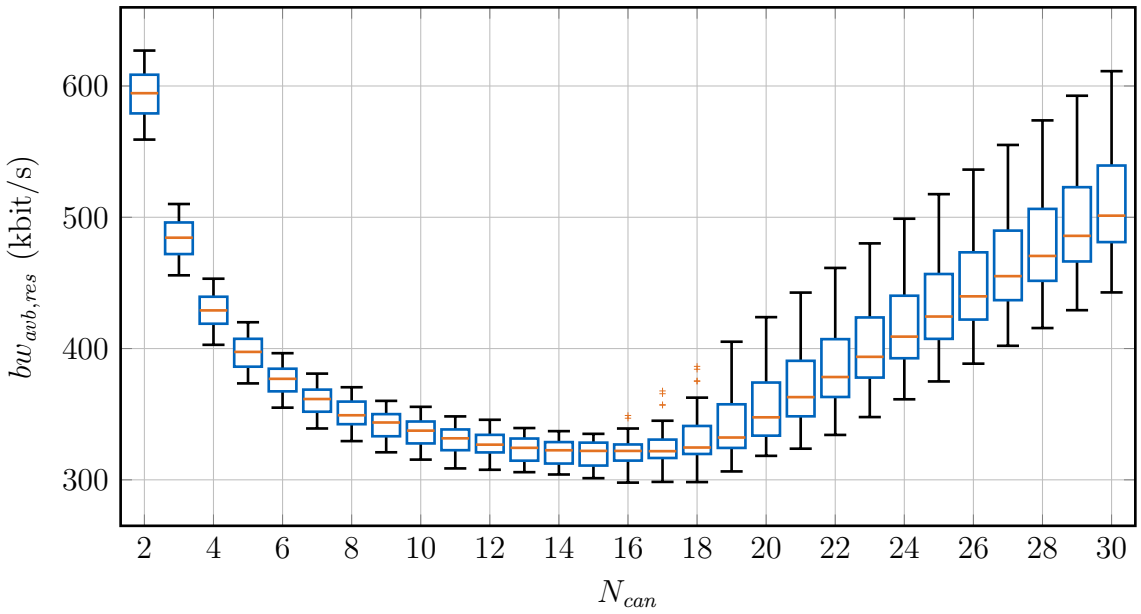


Figure V.11.: Statistical distribution of configurations in EDF forwarding at $Q_{fwd} = 50\%$. At every level of N_{can}, 50 message sets were evaluated.

can be determined as

$$OR = \sup\left\{\frac{t_g - t_h}{t_h}\right\}. \tag{V.27}$$

This level of overreservation is optimal, as any smaller value would cause the configuration to be unschedulable.

Using the scenarios from Section V.1.2, the necessary bandwidth to guarantee schedulability using optimal overreservation is evaluated. Figure V.11 shows statistical results obtained from 50 different message sets with 50 % of the CAN traffic getting forwarded. The evaluation was restricted to 50 message sets to allow a clear presentation of the results.

Optimal configurations are found for values of N_{can} between 13 and 19. Aggregating less CAN frames in one AVB Ethernet frame causes an increase in framing overhead that proves inefficient. Further increasing N_{can} introduces delays between two forwarding events, which can only be coped with through higher overreservation.

The statistical distribution shows a high value spread for large values of N_{can}. When forwarding many CAN frames aggregated in a single AVB frames, the sending interval T_{avb} should be large. Because of the bursty nature of the forwarding, small changes in a traffic configuration can require big adjustments of the configuration.

While some configurations give good results over a wide range of setups, there is no clear optimum. Specifically, the only outliers observed exist for the three

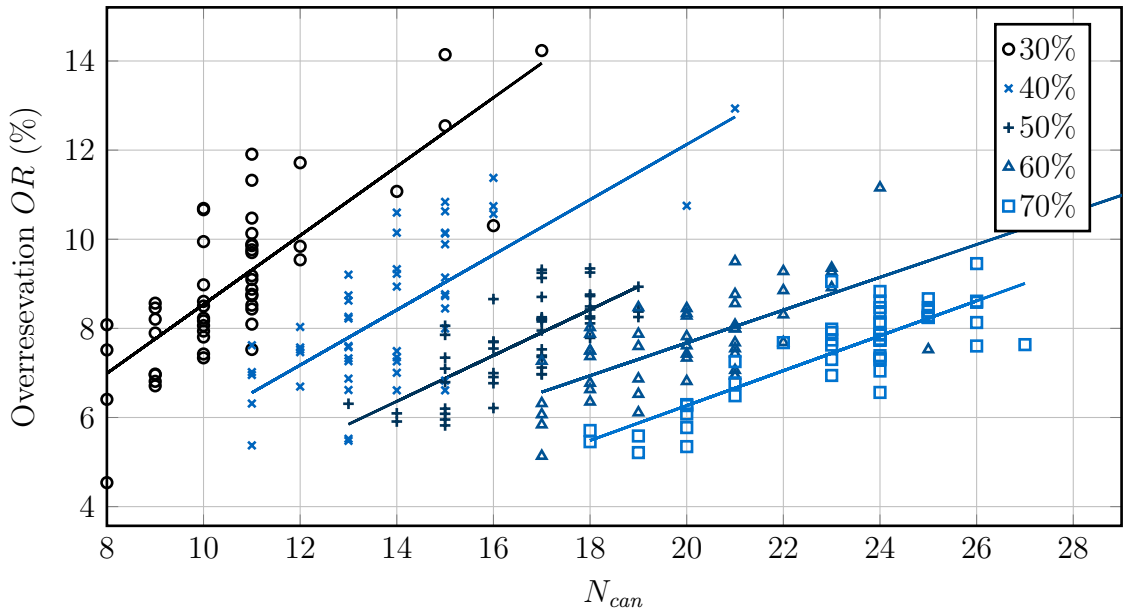


Figure V.12.: Optimal configurations for EDF forwarding and different levels of Q_{fwd}

values of N_{can}, which provide schedulability with minimum resources. An optimal configuration must therefore be derived for every setup.

For each of the 50 scenarios, the optimal pair of N_{can} and OR at every level of Q_{fwd} was determined. Figure V.12 shows these configurations along with linear regression lines. Efficient configurations have small OR and large N_{can}. With increasing Q_{fwd}, less overreservation is necessary and more CAN frames can be aggregated in a single AVB frame. Values of N_{can} in optimal configurations range between 8 and 30.

2. AVB Ethernet Controller Design and Integration

Because AVB Ethernet is not yet a commercially wide-spread technology, there is a lack of AVB-capable hardware as detailed in Section II.1.4.5. Thus, the vast majority of current research is theoretical. To fill the lack of architectural and experimental experience with AVB Ethernet controllers, the objective was formulated to design and implement an AVB-capable Ethernet controller. This controller should be integrated into an SoC to enable an evaluation of the system performance, thus providing hands on experience and reliable data.

The design is mainly constrained by the respective set of IEEE specifications, which specify the protocols for traffic shaping/policing and time-synchronization. Nevertheless, degrees of freedom exist in the HW/SW partitioning of the protocol components and the implementations themselves. For example, the time-synchronization could be implemented either in software or hardware, leading to a trade-off between precision and chip area cost. The following design is targeted to minimize the area cost (here estimated through FPGA resource consumption) while providing sufficient performance for automotive applications.

The design was implemented in VHDL using Xilinx Zynq 7000, a SoC with integrated FPGA fabric. This method combines the benefits of FPGA prototyping and SoC design. It allows time and bit accurate system evaluation of hardware extensions considering all aspects like memory and interconnect interference, cache behavior, etc. that are hard to fully cover using traditional system level simulation approaches. The design decisions taken within the next section will be evaluated in Section V.2.2 on system level.

2.1. Design & Implementation

This section presents a HW/SW partitioning targeted to comply with automotive application requirements while minimizing resource usage. The design is closely integrated with the processing and memory subsystem of the SoC. Because design decisions are directly affected by the system architecture of the Zynq 7000, the architecture will be introduced first. Following, the hardware and software extensions to realize the AVB Ethernet endpoint are presented.

2.1.1. System

The system used for the implementation is a Xilinx Zynq 7000 SoC. It consists of two parts: a processing system (PS) and programmable logic (PL). Because processing system and programmable logic are integrated on a single chip, it is possible to tightly interconnect hardware extensions with on chip resources. A prototyping board called ZedBoard was used for implementation. Besides the Zynq 7000 it features various connectors including and FPGA Mezzanine Card (FMC) connector.

It can be used to connect extensions boards to the FPGA. Linux based operating systems are ported to the Zynq 7000 SoC. The OS used here uses kernel version 3.14.

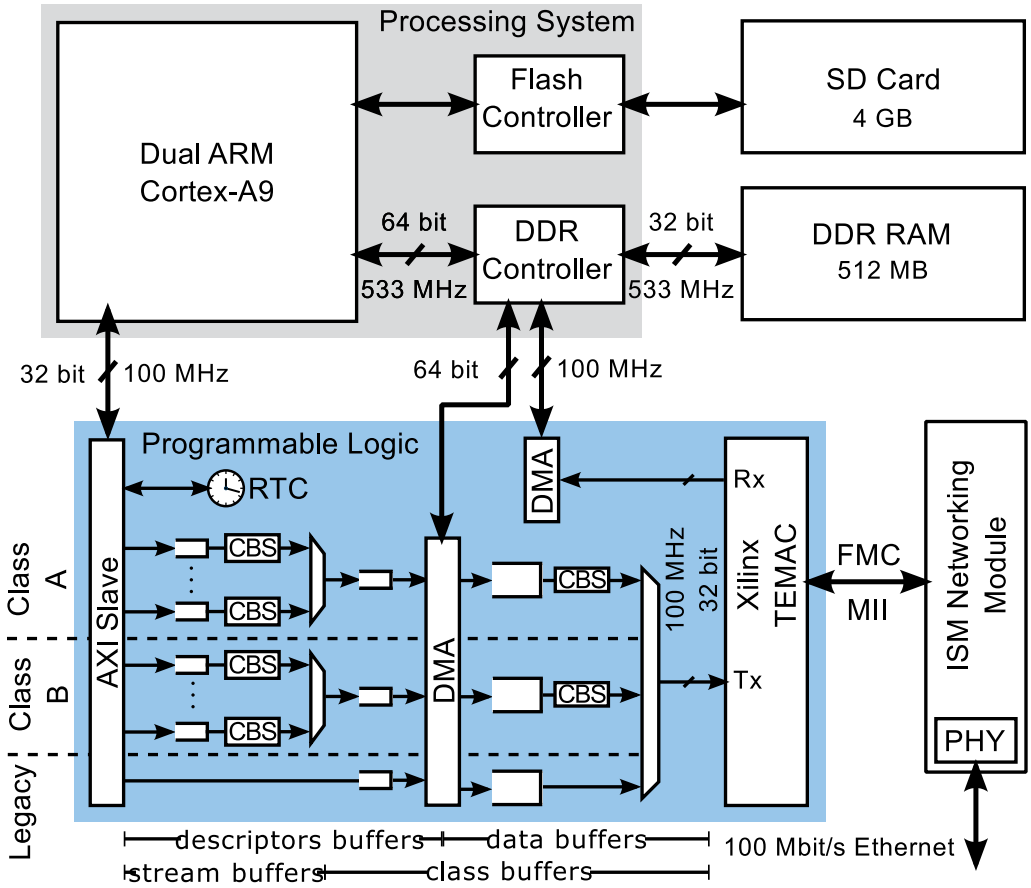


Figure V.13.: System overview of the prototype system including a detailed presentation of the AVB Ethernet controller implementation [119]

A simplified representation of the system is depicted in Fig. V.13. The majority of peripheral modules included in Zynq are omitted, as they are not used by the AVB endpoint implementation here. Processing within Zynq is done using a dual ARM Cortex-A9, which is running at 533 MHz. Part of the SoC are also controllers for DDR RAM and Flash memory. They provide access to off-chip resources placed on the ZedBoard: an SD card (4 GB) storing the boot image and the main memory (512 MB DDR3).

The integrated PL is equivalent to a Xilinx Artix-7 FPGA fabric with 85K overall logic cells. The PL is integrated alongside the PS within the SoC and can access its resources using AXI interfaces. These AXI buses are all clocked with 100 MHz. The ARM processors are connected to the PL with a 32 bit bus to perform memory

mapped I/O accesses. Additionally, up to four AXI buses can be used for DMA operations by the PL. Two physical ports of the DDR controller are reserved for transactions from the PL. The PL can be connected to the outside world trough the FMC connector located on the ZedBoard. Here, a networking specific breakout board called *ISM Networking FMC module* featuring a 10/100 Ethernet PHY is used. This PHY can be connected through an Media Independent Interface (MII) interface.

2.1.2. AVB Ethernet Controller

An AVB Ethernet controller is a layer-2 hardware device. It is intended to provide medium access according to the AVB specification. The AVB controller does not have to provide a complete implementation of the AVB protocols, as some may be possible to be implemented in software. Highly time-critical subsets of the specification like flow control and traffic shaping must be implemented in hardware, because software scheduling granularities are insufficient for the precision needed. For other parts like time-synchronization, a software implementation is possible but provides lower precision compared to a hardware implementation.

A schematic of the AVB Ethernet controller is presented in the lower half of Fig. V.13. It uses three interfaces towards the processing system: The control path is realized through the AXI directly connected to the computing cores. The driver can thus configure the controller through memory mapped I/O writes towards the AXI slave interface. Two additional AXIs are used to connect transmit (Tx) and receive (Rx) data path to the DDR controller. All data path operations are performed through direct memory access (DMA) to minimize software overheads.

One design objective is to achieve a composable architecture. Basis of this architecture is he Xilinx Tri-Mode Media Access Controller (TEMAC). It is a full-duplex Ethernet MAC supporting 10/100/1000/2500 Mbit/s operation. Here, 100 Mbit/s will be used exclusively as this is the only speed at which automotive grade transceivers are available. It is interconnected with the PHY through MII. Several configuration and interrupt lines are not shown within the figure for presentation reasons.

The HW/SW partitioning of has to be optimized to fit the setting and requirements of an automotive application scenarios. Multiple applications in an automotive setting require time-synchronization at different levels of precision. In video systems, frame accuracy at 30 fps can be achieved with around 30 ms synchronization precision. For audio signals at sampled 44.1 kHz, sample accuracy is achieved at 20 µs. In control applications, synchronous sensor sampling is an important objective. Sensors are usually connected with a LIN bus, which operate at up to 20 kHz (equivalent to a bit time of 50 µs). Thus, the precision is anyway bounded as larger than 50 µs.

As mentioned in Section II.1.4.3, Mahmood et al. [38] showed that synchronization

precisions within microsecond range can be achieved with a software-based time-synchronization. While a pure HW implementation can offer nanosecond precision, it is not achievable in real-world scenarios. This is because the synchronization protocol used in AVB is only error-free if networking delays are symmetrical. Based on related work, which showed around 1 µs synchronization errors due to asymmetric networking delays in a symmetric setting [40], the increased precision of a hardware-based implementation does not seem to be worthwhile.

To enable a software-based implementation of the time-synchronization protocol, a real-time clock (RTC) with read and write capabilities is needed. Because no internal clock of the PS satisfied the requirements, an RTC was implemented within the AVB Ethernet controller. It can be read and written through the AXI slave interface. When used as a master clock, it also needs to generate interrupts to trigger the synchronization process. The interrupts are generated in a configurable interval between 125 ms and 1 s (as required by the specification).

Outgoing traffic has to be buffered within the AVB Ethernet controller and policed according to the Credit Based Shaper (CBS) defined in the AVB protocol. Within a talker endpoint, the CBS maintains a credit value for each traffic class and stream. This credit value has to be updated frequently. During transmission, it is reduced for every bit transmitted. Additionally, credits proportional to the reserved bandwidth of the stream or traffic class are constantly added. Because Ethernet frames are quantized as multiples of octets, it is sufficient to update the credit score after a time that corresponds to the transmission time of an octet on the physical medium. For 100 Mbit/s, this time corresponds to 80 ns. Maintaining accurate credit scores is essential, because it decides which stream or traffic class is able to transmit (c.f. Section II.1.4.1). A software implementation of the traffic shaping would be inadequate, because software scheduling granularities are usually multiple orders of magnitude larger (e.g. Linux' minimum scheduling granularity is 75 ms). Thus, a hardware implementation of the CBS protocol is required.

The architecture of the stream and class buffers as well as the traffic shaping and flow control is depicted in Fig. V.13. As the trend in automotive ECUs is going towards consolidation and centralization of functions, it is foreseeable that an endpoint requires to send more than one stream, e.g. a combination of video, audio, and control streams. To enable such applications, the design allows offers multi-stream support. The number of streams supported is configurable at design-time. During run-time, the streams can be assigned to one of the traffic classes A & B.

The buffering strategy was implemented in order to minimize resource consumption while avoiding buffer underruns. To achieve this, Ethernet packets are buffered as DMA descriptor as long as possible. These descriptors consist of a memory address and the packet length. However, DMA operations take a finite and also variable amount of time to fetch the data from the main memory. Because a transmission from a class egress buffer has to happen instantly e.g. when a sufficient credit level is reached, it is essential that at least one frame is already buffered for

each class. Therefore, a data buffer able to contain one complete maximum sized frame was implemented for each traffic class. During every packet transmission, a DMA operation for the transmitting class is initiated just in time to avoid underruns.

An additional traffic class for legacy traffic is provided, which is not subjected to credit based shaping. Outgoing AVB traffic is prioritized over legacy traffic. However, only 75% of the overall bandwidth may be reserved of AVB traffic, thus guaranteeing that legacy traffic is not subject to starvation.

Reception of frames (the listener portion of the endpoint) is simpler, because no traffic shaping or flow control has to be applied here. The bandwidth for DMA operations is significantly larger than the one of incoming frames (6.4 Gbit/s compared to 100 Mbit/s). Therefore, a frame is always completely transferred before the next one arrives, even with back-to-back frames. This means that no additional buffers are needed, and data can directly be transferred from the internal Rx buffer of the TEMAC. After the complete transfer of data to the main memory, the PS will be notified through an Rx interrupt. To minimize interference among Tx and Rx DMA transactions, Tx and Rx DMA engines are connected through separate AXI interconnects to different ports on the DDR controller. During the evaluation, this approach will be compared to one using a shared bus and DDR controller port.

2.1.3. Linux Driver

A driver for the AVB Ethernet controller was developed that serves two main purposes: First, it provides access to the hardware component. Secondly, it implements portions of the AVB specification which are not covered by the hardware implementation introduced above. It was implemented as a character device driver in Linux' kernel space. The main tasks performed include descriptor and data buffer management, interrupt handling and time-synchronization.

AVB data transmissions are achieved through DMA transfers. When a user application initiates a data transmission, it uses a system call to notify the kernel module. It passes along the stream ID, the packet length and a data pointer towards the driver. The kernel module has to copy this data into kernel space. The driver maintains separate ring buffers for each stream. DMA descriptors are issued towards the AVB Ethernet controller and enqueued in the respective stream buffer. Once a packet has entered the traffic class data buffer, i.e. it has been copied from the main memory, the driver is notified through an interrupt and deallocates the memory.

Upon the reception of a message, the AVB Ethernet controller initiates a DMA transfer of the data into the main memory. It uses a DMA descriptor that was issued previously by the device driver and that is pointing towards the next empty slot within a ring buffer. The completion of the transfer is notified through an interrupt. Received packets are sorted towards stream-specific buffers. These buffers are descriptor-based ring buffers. Thus, no data copy operations are necessary when

sorting.

Another important part of the driver is the time-synchronization. AVB specifies a time-synchronization protocol called 802.1AS. It was presented in detail in Section II.1.4.3. As discussed above, an RTC was implemented within the AVB Ethernet controller, which can be read/written and triggers interrupts in a configurable interval. The synchronization itself has to be implemented in software. It differs for master and slave nodes.

The task of a master endpoint is to periodically initiate a synchronization procedure. For that, an interrupt is issued at a configurable interval by the RTC within the AVB Ethernet controller. This triggers the beginning of a synchronization sequence through a *Sync* message containing a current timestamp $T1$. A second time stamp is taken by the master upon the reception of the reply from the slave node.

Slave nodes participate in the synchronization sequence initiated by the master node. Upon receiving the initial *Sync* message, a timestamp $T1'$ is recorded. Another timestamp $T2$ corresponds to the sending of the so called *Delay_Req* message replied to the master node. Upon reception of the final message at the slave node, the value of the RTC has to be adjusted. Based on the recorded timestamps, an updated time value is calculated using (II.11). The slave can adjust the RTC by writing to the corresponding register within the AXI slave interface of the AVB Ethernet controller.

If and only if communication delays are equal in master-slave and slave-master direction, this scheme provides error-free synchronization. Asymmetries with respect to communication delays. Such delays can occur on a network level, but also within the SoC itself. Delays within the SoC are implementation specific. A major contributor to such delays within the SoC are variable interrupt latencies [38], contributing to errors upto 1 μs. Thus, the synchronization scheme is approximate in real-world settings, and non-perfect endpoint implementations are acceptable, as long as the error contributed is small compared the errors from other sources. The accuracy of the actual implementation will be researched within the scope of the following section.

2.2. Experiments & Results

To evaluate central performance and cost metrics of the overall system, a series of experiments was conducted. As the contribution under research is the AVB endpoint and the AVB controller, networking effects are kept minimal throughout the experiments. The general experimental setup consists of two endpoints directly interconnected. No switches and therefore no cross traffic are present in this setting. This controlled environment allows to access isolated measurements of the endpoint performance. The following sections present results regarding latencies, synchronization precision, and the hardware resource utilization by the AVB controller.

2.2.1. Endpoint-Related Latencies

Because AVB Ethernet will be used in time-sensitive applications, low-latency message transmission is a central target of the design. The real-time analyses presented in Section II.1.4.4 neglect endpoint-related latencies and only focus on networking delays. For this approach to be valid, latencies within the endpoint have to significantly smaller than networking delays. The real system implementation introduced here is constrained by finite processing times and on-chip communication delays. The following set of experiments is targeted at assessing the corresponding latencies associated with AVB Ethernet packet transmission. Separate experiments are conducted to evaluate latencies within the soft- and hardware portions of the system.

Experiment 1: Hardware latencies are measured by extending the AVB Ethernet controller using an additional timer with 10 ns resolution. At the beginning of the experiment, the endpoint is idle. No packets are buffered and streams have default credit score of 0. When the driver indicates the start of a transmission by writing to the AXI slave interface, the timer is started simultaneously. The DMA descriptor of the packet is forwarded through the buffers and traffic shaping. A DMA transfer is triggered and the complete packet is fetched from the main memory. The packet is afterwards forwarded to the MAC. The completion of this transaction (last word written to the MAC) stops the timer. The timer is read out at the end of the measurement routine depicted in Fig. V.14. To make sure any transmission is finished by the time the timer is read, it waits 500 µs after the initiation of the transmission until reading out the final timer value.

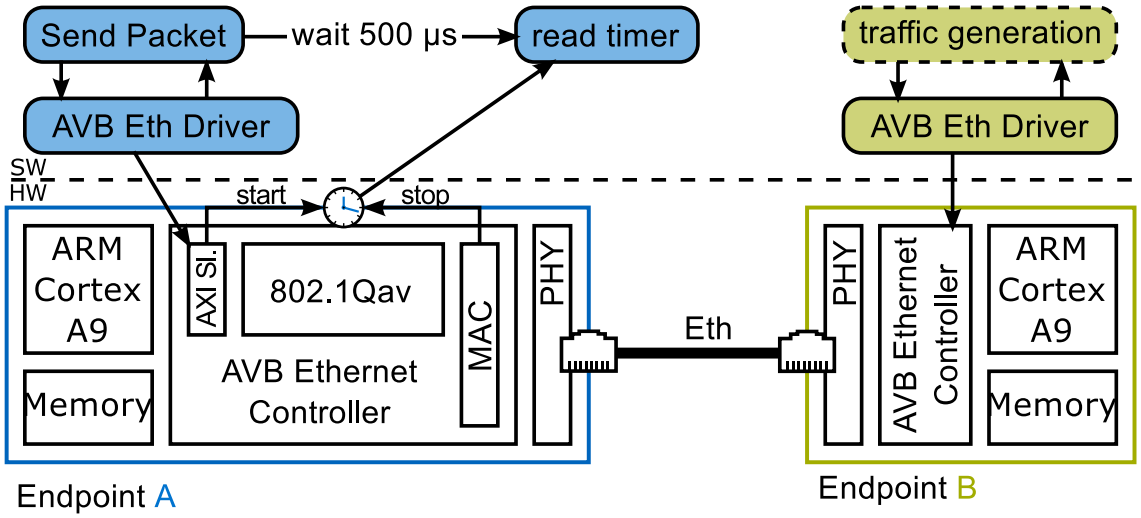


Figure V.14.: Measurement setup to determine latencies within the AVB Ethernet controller (Experiment 1) [119]

Several parameters were modified throughout the experiment. It was conducted using various packet sizes to evaluate the scalability of the architecture with data

size. This mainly affects the DMA packet fetch and the copy operation towards the MAC, as only they involve the actual payload data. Additionally, two different ways of interconnecting the DMA engines to the DDR controller were used. One uses a shared AXI and DDR controller port, the other uses completely separate interfaces. This configuration is especially important in the presence of simultaneous transmission and reception of frames, as the DMA operation is subject to interference. Therefore, optional receive side traffic was also included within the experiments. The traffic is generated by injecting 8000 packets/s with 1500 B (96 Mbit/s) using a second endpoint B. For every configuration, the experiment is repeated 1000 times.

Experiment 2: In the second experiment, delays within the software parts of the AVB endpoint are evaluated. For measurement, a counter within the processing system can be used. It is incremented at 200 MHz providing a resolution of 5 ns. The physical setup is equivalent to that of experiment 1 depicted in Fig. V.14.

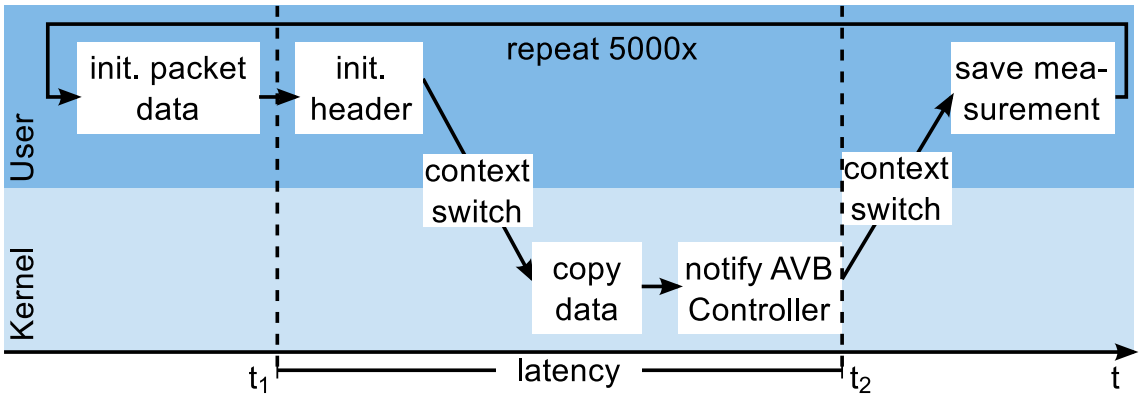


Figure V.15.: Measurement flow to determine software latencies (Experiment 2) [119]

To determine software delays, the measurement flow depicted in Fig. V.15 was used. The measurement starts after initiating the packet data. After recording a timestamp t_1, the AVB packet header is created. Following the kernel module is called, which causes the core to switch from user to kernel mode. This context switch is associated with an additional latency. Afterwards, the kernel module copies the data into kernel space and subsequently notifies the AVB controller, signaling the location and size of the packet within the main memory. The measurement is stopped at t_2 after the final write towards the AXI slave interface of the AVB Ethernet controller. This is the same write operation that triggers the beginning of the measurement in experiment 1. A user space application will save the measurement, with the latency being equivalent to the difference of the timestamps $t_2 - t_1$.

Similar parameters are varied as in experiment 1. The experiment is done with multiple packet sizes and uses optional receive side traffic to evaluate potential interference effects. However, this time, minimum sized packets are used in the receive traffic to maximize the number of interrupts and thus context switches. The

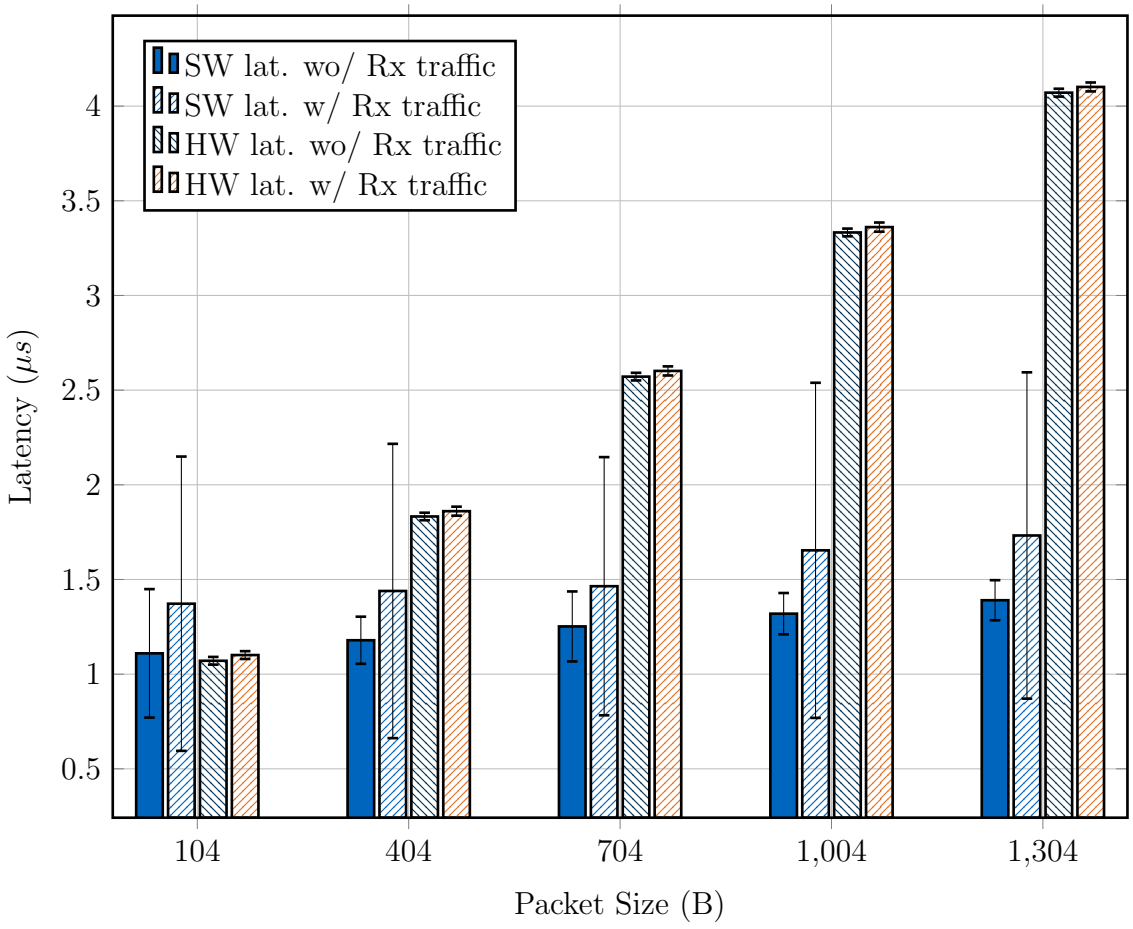


Figure V.16.: SW and HW transmit latencies observed in experiment 1 and 2 [119]

experiment is repeated 5000 times for every configuration. The increased number compared to experiment is necessary, as software latencies are subject to greater variations.

The combined results of experiment 1 (hardware latencies) and 2 (software latencies) are illustrated in Fig. V.16. Generally, it can be seen that latencies in hard- and software are similar for small packet sizes. Latencies increase with packet size in both cases, yet the increase is significantly larger for hardware latencies. Precisely, latencies increase by 0.23 ns/B for the software part and 2.42 ns/B for the hardware part. The ratio of these rates thus equals 10.52 and can be explained by observing the on-chip interconnects involved. From Fig. V.13 it can be seen that the processing cores access the main memory with 34.11 Gbit/s (64 bit × 533 MHz). On the other hand, the hardware part is facing a bottleneck in the interface of the MAC. It provides only 3.2 Gbit/s (32 bit × 100 MHz). The ratio among the interfacing bandwidths is 10.6 and thus closely resembles the ratio of the latency scaling with

data size. Thus, the scaling can solely be attributed to the respective interconnect bottlenecks.

The latencies for a packet transmission are increased when the AVB endpoint is subjected to concurrent receive traffic. This is expected, as transmit and receive path share on-chip resources, e.g. interconnects, memory, or computing resources. The results show that the software part of the AVB endpoint is more sensitive to receive side interferences. This is expected, because the hardware part has isolated buffering and forwarding mechanisms for receive and transmit operations. On the software side, a transmission can be significantly delayed e.g. by a receive interrupt. This leads to an average increase in transmit latencies by 30% in the interference case. Hardware latencies only increase by 2.09% on average using separate connection for transmit and receive side towards the DMA controller. If an AXI interface and DMA controller port is shared, this increase is around 6.57%. The use of separate interfaces is advisable if a strong isolation and predictability is required. This is the case for safety-critical embedded applications.

Generally, the latencies observed were in a microsecond range. The complete release of an AVB Ethernet frame can be expected to be completed in less than 10 µs. Latency requirements in the automotive domain can go as low as 2.5 ms in FlexRay-based chassis systems. This still leaves multiple orders of magnitude in comparison. However, the experiments also showed that software-based latencies are subject to large variations. In part, this problem could be overcome by using a real-time operating system or a real-time patched Linux kernel.

2.2.2. Synchronization Errors

The synchronization protocol used in AVB Ethernet leads to unprecise results, if the communication between two nodes has asymmetrical delays. This cannot only happen on a network level, but also within the chip due to variable interrupt latencies, cache hit rates etc. Within the following experiment, the error caused by system variances will be quantified.

Experiment 3: To get an isolated measurement on endpoint-related synchronization errors, all other sources of errors must be minimized. Therefore, no background traffic is present during the experiment, guaranteeing symmetric networking delays. Another important factor is temperature, as it directly influences the oscillating frequency of the clocks. The temperature cannot be influenced in the measuring setup, it remains as a source of error.

The measurement setup is illustrated in Fig. V.17. Two endpoints are directly interconnected, where endpoint A serves as master an endpoint B as slave. To evaluate the synchronization precision, the RTCs within each AVB controller must be read out and compared simultaneously. To enable such a measurement, the AVB controller had to be slightly modified. The RTC of endpoint A is routed towards parallel I/O pins and connected to endpoint B. Within endpoint B, the

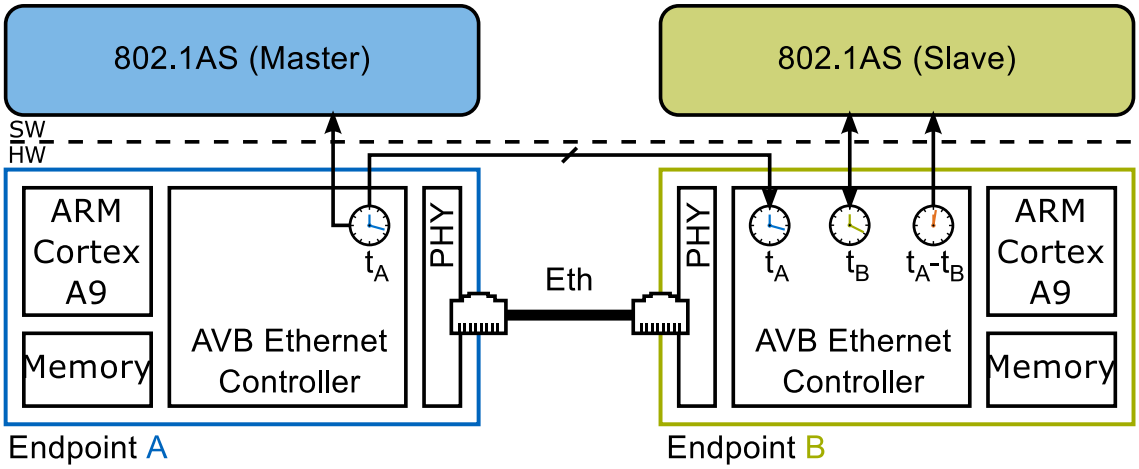


Figure V.17.: Measurement setup to determine synchronization errors (Experiment 3) [119]

time values of t_A and t_B are subtracted. The result is readable from the processing system end corresponds to the clock synchronization error. Right after completing a synchronization sequence and adjusting the RTC within the slave, this error value is sampled. In case of an error-free synchronization, this error should be zero. Reading the value at this time thus gives the error, that was caused by system non-idealities. In addition, the error due to clock drift can be assessed by considering the difference between the error before the current synchronization and after the last synchronization.

The experiments were conducted for varying synchronization intervals. In total, eight different intervals were chosen between 125 ms and 1000 ms. The measurement was repeated 1000 times for every respective synchronization interval.

Fig. V.18 shows the root mean square (RMS) synchronization errors contributed system non-idealities as well as the general clock drift. Root mean square error was chosen as it is used in related work and thus allows fair comparison.

The results show a trend of linear increasing clock drift for increasing synchronization intervals, ranging between 0.61 µs for 125 ms and 2.44 µs for 1 s. The scaling is not perfectly linear due to variations in the oscillator frequency (not measurement errors). A reason for this can be temperature variations between the measurements.

Focus of this evaluation is the error, that is directly related to the system and protocol implementation. This system error exists because of variable interrupt latencies, memory access times, cache behavior etc. RMS errors between 1.98 µs and 3.37 µs were measured. While there is no clear linear relation between the system error and synchronization intervals, a slight tendency of larger errors for larger synchronization intervals can be observed. It can be attributed to cache pollution, which increases with the time in which the synchronization protocol was not called.

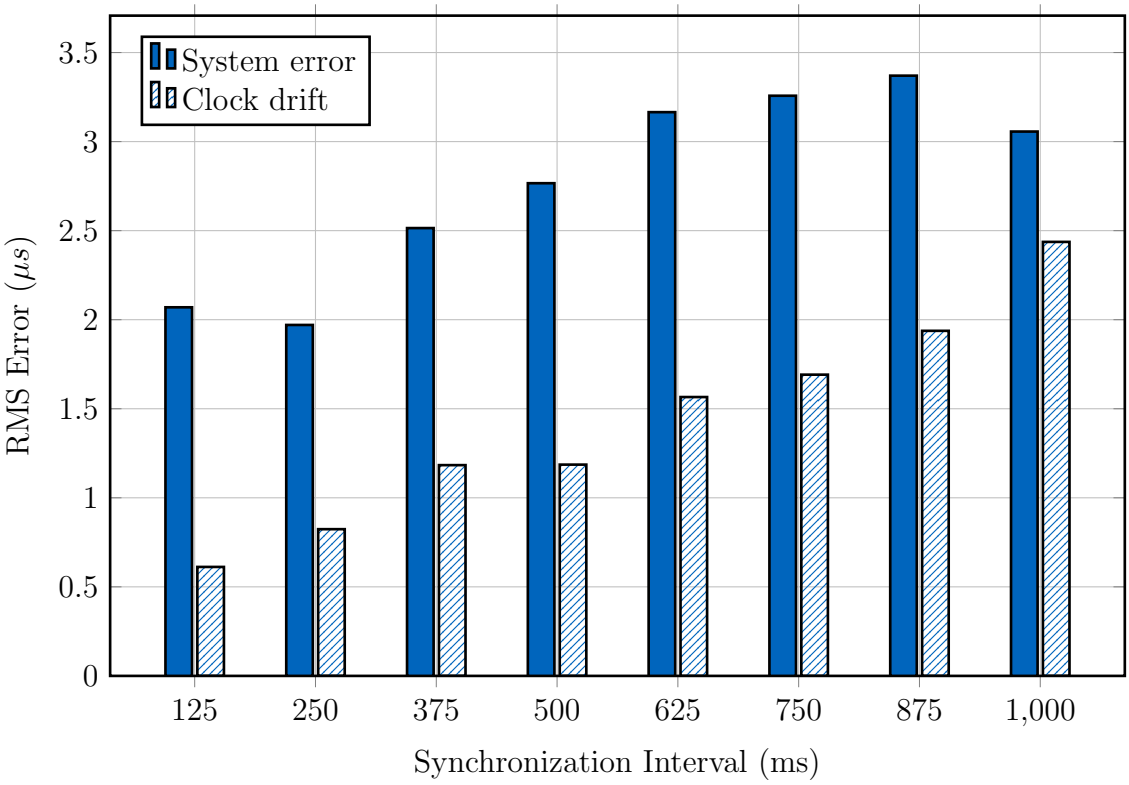


Figure V.18.: Synchronization error from system non-idealities as well as clock drift in experiment 3 [119]

In comparison with related work [38], the errors witnessed here are larger. Depending on the synchronization interval and CPU type, Mahmood et al. determined errors between 409 ns and 1.78 µs. They used a simulative approach relying on experimental data for interrupt latency distributions. Thus, interrupt latencies were the only non-ideality considered, which explains added errors in this case. Error sources like cache pollution were neglected.

The maximum system error measured throughout the experiment was 8 µs. To obtain a realistic view of the synchronization performance within a networked system, errors due background traffic on the network and clock drift have to be considered in addition.

2.2.3. Hardware Overhead

An important aspect for vendors of automotive SoCs is the hardware cost associated with an AVB Ethernet controller. Low cost means that an adoption of this technology comes at a lower economic risk, which accelerates the availability of AVB-capable SoCs. To quantify the resource utilization of the design presented in Section V.2.1, a comparison between the cost of a simple legacy Ethernet design and

the AVB-enabled on will be made. The legacy design includes a MAC, DMA engine and AXI interface. The designs were synthesized and implemented using Xilinx Vivado 2013.4 and evaluated for resource consumption with respected to logic as look up tables (LUTs) and flip flops (FFs). The results presented in Fig. V.19 are valid for a design with four streams.

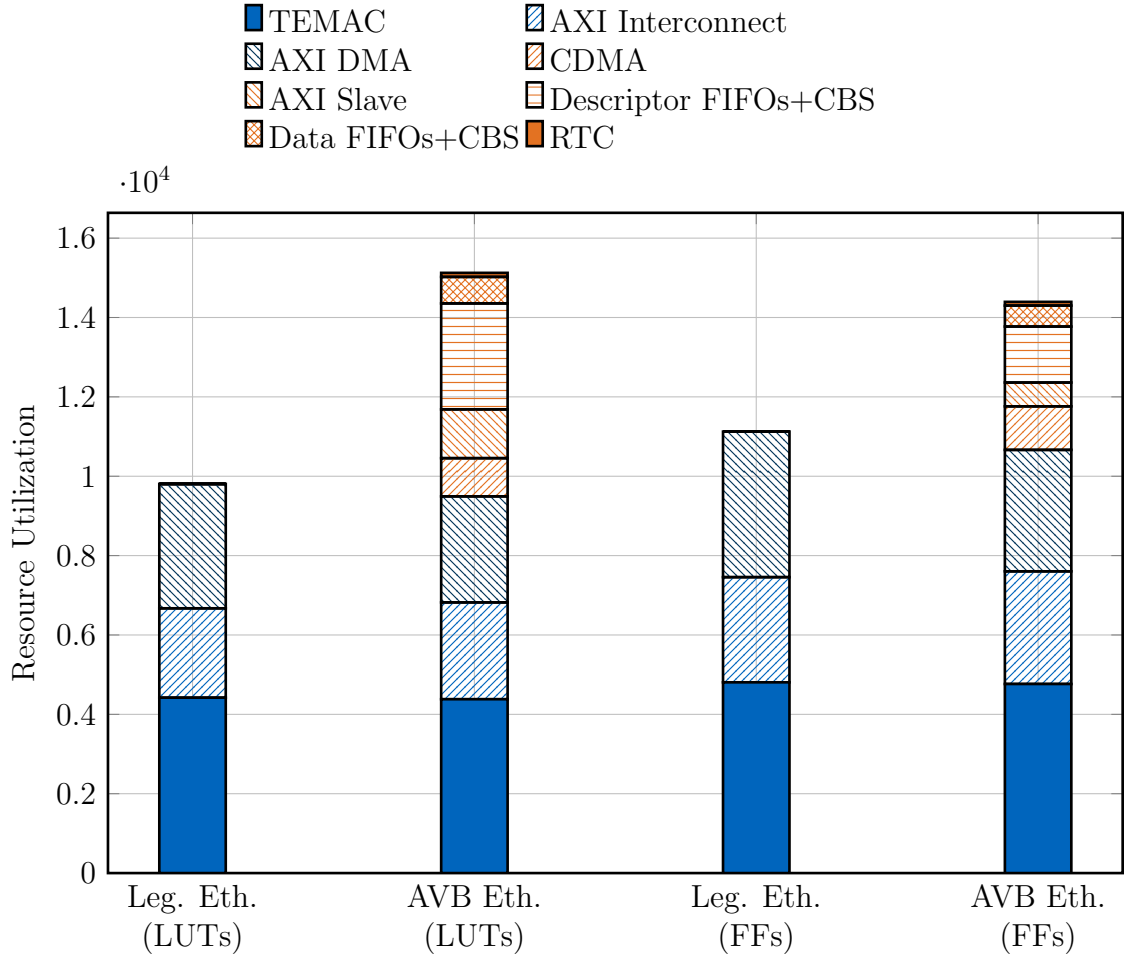


Figure V.19.: Hardware resources used by the AVB Ethernet controller compared to a legacy controller [119]

Compared to the legacy Ethernet design, the AVB Ethernet controller has increased hardware utilization by 54% in LUTs and 28% in FFs. The majority of added resources is consumed by the stream descriptor buffering and traffic shaping. The added RTC for the time synchronization requires less than 1% of resources within the AVB Ethernet controller.

In the implemented design, two separate DMA engines were used for transmit and receive path. The TEMAC uses AXI stream interfaces, which can be directly connected to the AXI DMA module for receive side transfers. AXI stream does

not use and addressing scheme. This is troublesome in the transmit operations, because transfers are executed towards different class FIFOs. To ease the design, a Central DMA (CDMA) was used, where a DMA transfer can be directed towards a specific FIFO. The increase in resource utilization due to the use of two DMA engines constitutes around 4.8% of the total overhead.

A 40% increase in resource consumption is significant, yet it was compared to the most simple Ethernet solution possible. Commercial Ethernet controllers have many advanced features and accelerators like TCP offload engines etc. Extending such a controller would lead to much smaller overheads.

3. Summary

The bandwidth demand of automotive embedded systems is steadily increasing. Multi-core processors can provide the necessary computing power for applications like advanced driver assistance and autonomous driving. However, to deliver sufficient communication resources in such architectures, new interconnect technologies like AVB Ethernet are required. Within this chapter, key challenges for the integration of AVB Ethernet into existing automotive embedded systems were researched. It was addressed, how AVB Ethernet can be interconnected with existing networking technologies in automotive embedded systems like CAN. Additionally, a HW/SW partitioning and a composable hardware architecture of an AVB Ethernet controller was designed and integrated into a dual-core ARM SoC to determine performance an cost of such an extension.

To enable the coexistence of AVB Ethernet and the most prevalent legacy interconnect Controller Area Network (CAN), a **CAN to AVB Ethernet gateway** that interfaces both protocols was presented. Two main objectives constrained the design: First, the forwarding must happen in real-time capable manner. Secondly, minimal resources should be used. In the case of an AVB network, this means that minimal bandwidth should be reserved for the CAN over AVB Ethernet stream.

The framing overhead is minimized by using a frame aggregation technique, which encapsulate multiple CAN frames within a single AVB Ethernet frame. To provide real-time conforming latencies, overreservation is employed. Finding an optimal combination of these two parameters - the number of CAN messages within one AVB Ethernet frame and the amount of overreservation - is the central design trade-off. Different scheduling mechanisms were explored to select a subset of buffered CAN messages for forwarding. These include FIFO, strict priority (SP), and earliest deadline first (EDF). An analytic framework was derived for each mechanism.

In a design space exploration, it was shown that EDF is the most efficient mechanism with respect to the formulated design goals. Optimal configurations usually encapsulate around 15 CAN frames within one AVB Ethernet frame. Compared to state of the art gateway designs, a reduction of the necessary bandwidth reservation by 72.21% was achieved. Additional, an algorithm to determine the optimal configuration for a given scenario using EDF forwarding was presented.

The lack of commercially available AVB Ethernet hardware means that there is little to no experimental data available. By designing and implementing an **AVB Ethernet endpoint** using a Zynq-based prototyping platform, data on HW/SW partitioning, communication latencies, synchronization precision and hardware consumption was gained. The proposed design extends a conventional MAC with AVB capabilities at the cost of 53% additional logic and 28% additional flip flops.

Due to precision constraints within the traffic shaping algorithm, queuing and forwarding was implemented in hardware. The overall latency when transmitting an AVB Ethernet frame consists of driver delays (creating and moving a packet from

user to kernel space) and delays within the hardware (forwarding throughout the buffers, fetching data from main memory). Hardware latencies within the design range between 1 µs to 4 µs, depending on packet size. These latencies are robust to interfering receive side traffic with deviations of less than 25 ns. Combined with driver delays, latencies smaller than 6 µs can be expected. To put this number into context, it can be compared to the transmission time of a minimum sized Ethernet frame at 100 Mbit/s (5.12 µs). On the other hand, end-to-end networking delays are usually expected in the order of multiple milliseconds.

Another important performance metric is the synchronization precision within the endpoint. AVB Ethernet uses an approximate synchronization protocol that is error-prone if communication latencies are asymmetrical. Within the endpoint, variances are caused by variable interrupt latencies, cache pollution etc. In an experiment, it was shown that these non-idealities within the endpoint cause errors smaller than 8 µs. This accuracy is sufficient for the majority of automotive application scenarios like synchronous sensor sampling. If future applications require increased precision, the synchronization protocol could be offloaded into hardware.

The contributions presented in this chapter significantly extend related work in the field of AVB Ethernet. The combination of AVB capable SoCs and gateways interfacing legacy fieldbuses allows an integration of this new networking technology into existing automotive embedded networks. Nevertheless, many challenges are yet to come, when AVB Ethernet is implemented in automotive embedded systems on a large scale. With the emergence of real-world application scenarios and data, further optimzation and innovation is possible.

Additionally, requirements towards the networking infrastructure are expected to continue increasing. This may reach a point, where AVB Ethernet alone might not be sufficient anymore. While AVB Ethernet could be scaled further to 1 Gbit/s Ethernet, it has certain limitations when it comes to very low latency communication for advanced control applications. For that reason, AVB's successor called Time-Sensitive Networking (TSN) is in development. The pre-standard extends AVB Ethernet by time-gated admission control. This allows highly predictable communication, but also changes requirements towards endpoints and gateways.

VI. Conclusion & Outlook

1. Conclusion

The contributions of this thesis are target the enablement of multi-core processors in automotive embedded systems, focusing on I/O and networking aspects. The ever-growing demand for computational power and the need to consolidate functions on shared platforms can be coped with through the introduction of multi-core processors within embedded systems.

Challenges arising from this scenario are efficient sharing of peripherals and networks, real-time capability, and spatial and temporal isolation among tenants in mixed-criticality scenarios. To fulfill these requirements, virtualization methods are researched in the context of embedded systems. An architecture for a virtualized CAN controller is derived, which enables concurrent access to the CAN bus for multiple tenants. To enable isolated communication among nodes, network virtualization is applied to the CAN bus itself.

In-vehicle networks need to be scaled along with the computing power of single nodes. Further contributions address the integration of real-time capable AVB Ethernet into automotive embedded systems through a proposed gateway mechanism for CAN and an AVB endpoint design.

To enable low-latency access to a CAN controller for multiple tenants in a virtualized setup, an I/O virtualization concept was developed that is based on a layered architecture. Through hardware extensions in the CAN controller itself, overheads that go along with paravirtualized state-of-the-art solutions were minimized. Through a prototypical FPGA implementation in an Intel Core i7 system, it was shown that added latencies can be reduced by 91% to 96%. The hardware resources required by the virtualization extensions are similar to that of a non-virtualized controller so that a break-even in terms of chip area efficiency is reached for only two virtual controllers.

The efficiency of the virtualization extensions can only be achieved, if underlying hardware modules are shared by all virtual CAN controllers. This introduces challenges with respect to temporal isolation of concurrent partitions, especially in mixed-criticality scenarios. Using real-time analysis and an automotive traffic scenario, it was shown that head-of-line blocking within the virtualization layer is bounded to 45 µs even in a configuration with 16 virtual CAN controllers. However, these bounds are only valid when all partitions behave according to their specification. To ensure that every partition receives guaranteed performance, a temporal

isolation scheme based on a weighted and time-based round-robin is introduced. In a simulated scenario, where one partition behaves as a 'babbling idiot', the feasibility was verified.

Virtualized systems are subject to high overheads associated with interrupts, as these are required to be handled within a privileged component and therefore force exits in the VMs. The overhead can be limited through interrupt throttling mechanisms. However, state of the art approaches are not designed to protect real-time capability at the same time. This problem is overcome by the deadline-aware interrupt coalescing approach presented here. Applied to CAN, it was shown that upto 20 interrupts can be reduced to a single one while maintaining real-time capability. The virtualization layer of the virtualized CAN controller was extended with deadline-aware interrupt coalescing at the cost of around 10% increased hardware resources.

While the virtualized CAN controller provides isolated access towards the CAN bus, it does not prevent interference on the shared bus itself. Traditionally, communication flows with different criticality levels are mapped to separate physical buses. However, when mixed-criticality functions are consolidated on multi-core platforms, it is increasingly inefficient to pursue this physical segregation approach. Network virtualization is researched and applied to the CAN bus to allow concurrent mixed-criticality communication through isolated virtual sub-networks.

Key to the concept is the admission control towards virtual CANs, which needs to provide bandwidth and latency guarantees. This is done with a combination of a strict prioritization enforced through ID space partitioning of the CAN bus and a token bucket policer. As this approach led to bad scalability with respect to latencies in low priority VCANs, a second token bucket was added to perform traffic shaping. Thus dual token buckets allow to tune the latencies within a VCAN without adjusting its bandwidth reservation.

Schedulability analysis showed that the network virtualization approach enables real-time capability for mixed-criticality communication in high load scenarios. While a normal CAN with criticality-monotonic partitioning of mixed-criticality traffic is only real-time capable for bus loads smaller 50%, the single token bucket and dual token bucket approaches are still working efficiently at loads around 75% and 88%, respectively. The virtualized CAN controller was extended with support for network virtualization at the cost of around 6% increased hardware resources compared to the virtualization layer.

The currently favored candidate for high bandwidth real-time communication is AVB Ethernet. Therefore, its integration into automotive embedded systems is another key step towards highly integrated and multi-core based architectures. Within the scope of this thesis, it is facilitated through research regarding gateways between CAN and AVB Ethernet as well as HW/SW architectures of AVB Ethernet endpoints.

A tight integration of AVB Ethernet into legacy dominated embedded systems

is only possible if an efficient way of interfacing existing networking technologies is available. A gateway between two networking technologies like AVB Ethernet and CAN has to overcome significant differences in terms of protocol, framing, bandwidth, arbitration etc. A concept for such a gateway was proposed, which is optimized to guarantee real-time capable forwarding while using minimal bandwidth resources.

To minimize the framing overhead, it is common practice to encapsulate multiple CAN messages within one Ethernet frame. In contrast to state of the art approaches, the concept here does not require the gateway's buffer to be emptied completely when an Ethernet frame is transmitted. This has the potential to reduce bandwidth, but also requires to select a subset of buffered CAN messages for forwarding. Multiple schedulers were considered including FIFO, strict priority, and earliest deadline first (EDF). Throughout the evaluation, EDF proved to be the best choice, leading to a bandwidth reduction of 72% compared to related work.

Currently, there is hardly any practical research ongoing regarding AVB Ethernet, which can be attributed to the lack of commercially available AVB Ethernet hardware. Here, architecture and HW/SW partitioning of AVB Ethernet endpoints were researched through a prototypical implementation. The design focuses on the relevant protocol components in automotive use cases, namely time-synchronization and traffic shaping.

The traffic shaping is highly time-critical and has to be carried out at a time granularity of 80 ns. It was thus implemented in hardware only. The design space for the time-synchronization allows hardware and software implementations. Here, a software-based implementation was chosen, which leads to a lower hardware overhead but reduced synchronization precision. The proposed design extends a conventional MAC with AVB capabilities at the cost of 40% increased hardware resources.

The performance of the prototype was assessed using system level experiments. The expected latency for a packet release from the invocation at the driver until the start of the transmission was determined to be smaller than 6 µs for maximum sized packets. Variable interrupt latencies and cache behavior introduce an error in the synchronization process. The experiments showed that this error is smaller than 8 µs.

The contributions and corresponding results are key steps towards an enablement of multi-core processors in automotive embedded systems. Nevertheless, further research can be done to improve the approaches presented here and to tackle further challenges, which were not addressed in the scope of this thesis. They will be outlined in the following section.

2. Outlook

The virtualization extensions presented here have been shown to fulfill necessary requirements for safe operation like temporal isolation as single components. However, their integration into a real-world system also showcased their reliance on the rest of the system. For example, isolated performance on the peripheral interconnect is necessary to safely use virtualized I/O controllers in mixed-criticality systems. Enablement of virtualization and resource sharing is required on all relevant parts of a multi-core platform including caches, interconnects, memory, storage, and co-processors. If temporal or spatial isolation is not guaranteed within a single part of the system, efforts within the rest of the system may be rendered ineffective.

The virtualization concepts presented here focus on single components, e.g. a CAN communication controller. In future efforts, holistic separation concepts could be researched. Isolation is usually traded off with increased latencies. Using separate isolation stages within each component leads to accumulated added latencies and performance variations. Cross-component coordination with respect to resource sharing and isolation could optimize the system-level efficiency in mixed-criticality multi-core systems. For example, temporal isolation was achieved in the context of this thesis on CAN controller and on CAN bus level. Coordination could be improved, if the temporal isolation mechanism prioritizes virtual CAN controllers attached to VCANs, which are able to transmit during the next arbitration phase, thus reducing added latencies experienced in the virtualized CAN controller. On the other hand, a potential isolation mechanism within the peripheral interconnect should also be considered.

Schedulers provide a deep design space that could be explored further and tailored to concrete application scenarios. For virtual CAN controller schedulers, configurations for differentiated service levels could be used to improve latencies of highly critical partitions in mixed-criticality scenarios. The admission control mechanism towards VCANs provides low latency for high priority VCANs. However, scalability is limited as worst-case latencies of low priority VCANs in scenarios with four or more partitions are too high in real-time applications. While the addition of a second token bucket layer is able to improve the scalability, non-prioritized round-robin based admission control might be required to balance the performance of VCANs.

Network virtualization is powerful technique to improve the flexibility of networked system. While physical communication resources have to be planned in the early stages of a design process, logical communication resources can still be partitioned later on. Combined with the ability to provide isolated communication channels on a shared physical, it enables new network architectures. Currently, networks are directly associated with a functional domain. A more efficient architecture could be achieved by employing physical networks throughout the car and dividing it into isolated logical partitions using network virtualization. To enable an easier introduction of the network virtualization approach without the need to replace all

CAN controllers, a software-only version could be researched, which guarantees the same isolation at the cost of potentially increased worst-case latencies.

The virtualization solutions here focused on CAN, the most widely used networking technology in an automotive context. Nevertheless, consolidated control units will require shared access towards I/O controllers of other technologies like FlexRay or real-time capable Ethernet. The architectural extensions shown here need to be adapted to address the specific requirements of each technology like time-triggered network arbitration or high throughput.

Research with respect to AVB Ethernet is currently gated by a lack of industrial implementation data. If future car generations lead to a large scale employment of AVB Ethernet, analyses and design can be refined using real-world typologies and network configurations. The gateway concept proposed here focused on delays on the CAN bus and within the gateway itself. Further optimization would possible if latencies in the AVB network would be included in the analysis.

The design of the AVB Ethernet endpoint provides low latency communication. Further research could be conducted to explore HW/SW partitioning possibilities. The time-synchronization is currently mainly located in the host system, only with an RTC implemented in the AVB controller. Other options include the use of an embedded processor to implement the protocol as well as a full hardware implementation.

At the time of writing, AVB's successor, Time-Sensitive Networking (TSN), is under development. It extends the current standard by methods to improve the real-time capability of the network, e.g. through frame preemption and time-gated admission control. These additions extend the design space of endpoints and switches. The modularity of the endpoint architecture presented here facilitates an integration of these extensions associated with TSN.

Bibliography

[1] Hans-Ulrich Michel. Taming multicores for safe transportation: Aramis in the automotive domain. In *Workshop on the Integration of mixed-criticality subsystems on multi-core processors, presented at HiPEAC 2013*, 2013.

[2] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 163–168. IEEE, 2011.

[3] Daniel Work, Alexandre Bayen, and Quinn Jacobson. Automotive cyber physical systems in the context of human mobility. In *National Workshop on high-confidence automotive cyber-physical systems*, pages 3–4, 2008.

[4] Samarjit Chakraborty, Martin Lukasiewycz, Christian Buckl, Suhaib Fahmy, Naehyuck Chang, Sangyoung Park, Younghyun Kim, Patrick Leteinturier, and Hans Adlkofer. Embedded systems and software challenges in electric vehicles. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 424–429. EDA Consortium, 2012.

[5] Shane Tuohy, Martin Glavin, Ciarán Hughes, Edward Jones, Mohan Trivedi, and Liam Kilmartin. Intra-vehicle networks: A review. *Intelligent Transportation Systems, IEEE Transactions on*, PP(99):1–12, 2014.

[6] Georg Gut, Christian Allmann, Markus Schurius, and Karsten Schmidt. Reduction of electronic control units in electric vehicles using multicore technology. In *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools*, pages 90–93. Springer-Verlag, 2012.

[7] Dominik Reinhardt and Markus Kucera. Domain controlled architecture-a new approach for large scale software integrated automotive systems. In *PECCS*, pages 221–226, 2013.

[8] Marco Di Natale and Alberto Luigi 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.

[9] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems.

In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 220–229. IEEE Press, 2015.

[10] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, page 18. USENIX Association, 2007.

[11] Andre Richter, Christian Herber, Holm Rauchfuss, Thomas Wild, and Andreas Herkersdorf. Performance isolation exposure in virtualized platforms with pci passthrough i/o sharing. In *Architecture of Computing Systems–ARCS 2014*, pages 171–182. Springer International Publishing, 2014.

[12] Dominik Reinhardt, Dirk Kaule, and Markus 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] Kristian Sandstrom, Aneta Vulgarakis, Markus Lindgren, and Thomas Nolte. Virtualization technologies in embedded real-time systems. In *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–8. IEEE, 2013.

[14] Marius Strobl, Markus Kucera, Andrei Foeldi, Thomas Waas, Norbert Balbierer, and Carolin Hilbert. Towards automotive virtualization. In *Applied Electronics (AE), 2013 International Conference on*, pages 1–6. IEEE, 2013.

[15] Rolf Schneider, Andre Kohn, Karsten Schmidt, Sven Schoenberg, Udo Dannebaum, Jens Harnisch, and Qian Zhou. Efficient virtualization for functional integration on modern microcontrollers in safety-relevant domains. In *SAE World Congress 2014*. SAE International, 2014.

[16] Salvador Trujillo, Alfons Crespo, and Alejandro Alonso. Multipartes: Multicore virtualization for mixed-criticality systems. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 260–265. IEEE, 2013.

[17] Florian Sagstetter, Martin Lukasiewycz, Sebastian Steinhorst, Marko Wolf, Alexandre Bouard, William R Harris, Somesh Jha, Thomas Peyrin, Axel Poschmann, and Samarjit Chakraborty. Security challenges in automotive hardware/software architecture design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 458–463. EDA Consortium, 2013.

[18] Florian Hartwich. Can with flexible data-rate. In *13th International CAN Conference (iCC2012), Hambach, Germany*, 2012.

[19] Peter Hank, Steffen Müller, Ovidiu Vermesan, and Jeroen Van Den Keybus. Automotive ethernet: in-vehicle networking and smart mobility. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1735–1739. EDA Consortium, 2013.

[20] Lucia Lo Bello. Novel trends in automotive networks: A perspective on ethernet and the ieee audio video bridging. In *Emerging Technologies Factory Automation (ETFA), 2014 IEEE 19th Conference on*, pages 1–9, Sept 2014.

[21] Hyung-Taek Lim, Lars Völker, and Daniel Herrscher. Challenges in a future ip/ethernet-based in-car network for real-time applications. In *Proceedings of the 48th Design Automation Conference*, pages 7–12. ACM, 2011.

[22] Helge Zinner, Josef Noebauer, Thomas Gallner, Jochen Seitz, and Thomas Waas. Application and realization of gateways between conventional automotive and ip/ethernet-based networks. In *Proceedings of the 48th Design Automation Conference*, pages 1–6. ACM, 2011.

[23] International Organization for Standardization. ISO 26262:2012 road vehicles - functional safety - part 1-10.

[24] RR Brooks, S Sander, J Deng, and J Taiber. Automotive system security: challenges and state-of-the-art. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*, page 26. ACM, 2008.

[25] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[26] LIN Consortium. Lin specification package, 2010.

[27] Robert Bosch GmbH. Can specification 2.0, 1991.

[28] FlexRay Consortium. Flexray communications system-protocol specification, 2005.

[29] MOST Cooperation. Media oriented systems transport, 2011.

[30] Lucia Lo Bello. The case for ethernet in automotive communications. *ACM SIGBED Review*, 8(4):7–15, 2011.

[31] Freescale AMPG Body Electronics Systems Engineering Team. Future advances in body electronics. *White Paper*, 2013.

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

[33] Marco Di Natale and Haibo Zeng. Practical issues with the timing analysis of the controller area network. In *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–8. IEEE, 2013.

[34] KW Tindell, Hans Hansson, and Andy J Wellings. Analysing real-time communications: controller area network (can). In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 259–263. IEEE, 1994.

[35] IEEE. 802.1qat: Virtual bridged local area networks - amendment: 9: Stream reservation protocol (srp), 2010.

[36] IEEE. 802.1qav: Virtual bridged local area networks - amendment: Forwarding and queuing enhancements for time-sensitive streams, 2009.

[37] IEEE. 802.1as: Timing and synchronization for time-sensitive applications in bridged local area networks, 2011.

[38] Aneeq Mahmood, Reinhard Exel, and Thilo Sauter. Impact of hard-and software timestamping on clock synchronization performance over ieee 802.11. In *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*, pages 1–8. IEEE, 2014.

[39] Andreas Kern, Helge Zinner, Thilo Streichert, Josef Nöbauer, and Jürgen Teich. Accuracy of ethernet avb time synchronization under varying temperature conditions for automotive networks. In *Proceedings of the 48th Design Automation Conference*, pages 597–602. ACM, 2011.

[40] Hyung-Taek Lim, Daniel Herrscher, L Volker, and Martin Johannes Waltl. Ieee 802.1 as time synchronization in a switched ethernet based in-car network. In *Vehicular Networking Conference (VNC), 2011 IEEE*, pages 147–154. IEEE, 2011.

[41] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer, 2001.

[42] Rene Queck. Analysis of ethernet avb for automotive networks using network calculus. In *Vehicular Electronics and Safety (ICVES), 2012 IEEE International Conference on*, pages 61–67. IEEE, 2012.

[43] De Azua, Joan Adrià Ruiz, et al. Complete modelling of avb in network calculus framework. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 55. ACM, 2014.

[44] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis–the symta/s approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

[45] Jonas Diemer, Jonas Rox, and Rolf Ernst. Modeling of ethernet avb networks for worst-case timing analysis. *MATHMOD, Austria*, 2012.

[46] Philip Axer, Daniel Thiele, Rolf Ernst, and Jonas Diemer. Exploiting shaper context to improve performance bounds of ethernet avb networks. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[47] Felix Reimann, Sebastian Graf, Fabian Streit, Michael Glas, and Jurgen Teich. Timing analysis of ethernet avb-based automotive e/e architectures. In *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–8. IEEE, 2013.

[48] Unmesh D Bordoloi, Amir Aminifar, Petru Eles, and Zebo Peng. Schedulability analysis of ethernet avb switches. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

[49] Jan Taube, Florian Hartwich, and Helmut Beikirch. Comparison of can gateway modules for automotive and industrial control applications. In *Proceedings of the 10th international CAN Conference (iCC2005)*, 2005.

[50] Guoqi Xie, Gang Zeng, Ryo Kurachi, Hiroaki Takada, and Renfa Li. Gateway modeling and response time analysis on can clusters of automobiles. In *High Performance Computing and Communications, 2015 IEEE 7th Intl Symp on Cyberspace Safety and Security, 2015 IEEE 12th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2015 IEEE Intl Conf on*, pages 693–700. IEEE, 2015.

[51] Seung-Han Kim, Suk-Hyun Seo, Jin-Ho Kim, Tae-Yoon Moon, Chang-Wan Son, Sung-Ho Hwang, and Jae Wook Jeon. A gateway system for an automotive system: Lin, can, and flexray. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 967–972. IEEE, 2008.

[52] Ece Guran Schmidt, Melih Alkan, Klaus Schmidt, Emrah Yuruklu, and Utku Karakaya. Performance evaluation of flexray/can networks interconnected by a gateway. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 209–212. IEEE, 2010.

[53] Andreas Kern, Dominik Reinhard, Thilo Streichert, and Jürgen Teich. Gateway strategies for embedding of automotive can-frames into ethernet-packets

and vice versa. In *Architecture of Computing Systems-ARCS 2011*, pages 259–270. Springer, 2011.

[54] Hamid Ayed, Ahlem Mifdaoui, and Christian Fraboul. Frame packing strategy within gateways for multi-cluster avionics embedded networks. In *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–8. IEEE, 2012.

[55] Dominik Reinhardt, Maximilian Guntner, Markus Kucera, Thomas Waas, and Winfried Kuhnhauser. Mapping can-to-ethernet communication channels within virtualized embedded environments. In *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[56] Dionysios Skordoulis, Qiang Ni, Hsiao-Hwa Chen, Adrian P Stephens, Changwen Liu, and Abbas Jamalipour. Ieee 802.11 n mac frame aggregation mechanisms for next-generation high-throughput wlans. *Wireless Communications, IEEE*, 15(1):40–47, 2008.

[57] Tamizh Selvam and Subramanian Srikanth. A frame aggregation scheduler for ieee 802.11 n. In *Communications (NCC), 2010 National Conference on*, pages 1–5. IEEE, 2010.

[58] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[59] Robert P Goldberg. Architectural principles for virtual computer systems. Technical report, DTIC Document, 1973.

[60] Mendel Rosenblum. Vmwares virtual platform. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

[61] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177. ACM, 2003.

[62] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[63] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[64] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Linux Symposium*, pages 65–77, 2005.

[65] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy 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.

[66] Himanshu Raj and Karsten 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.

[67] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[68] Jeffrey C Mogul and KK Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[69] Lewis C Eggebrecht. *Interfacing to the IBM personal computer*. Sams, 2nd edition edition, July 1990.

[70] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 2–13, New York, NY, USA, 1994. ACM.

[71] Marko Zec, Milienko Mikuc, and Mario Zagar. Estimating the impact of interrupt coalescing delays on steady state tcp throughput. In *SoftCOM 2002: international conference on software, telecommunications and computer networks*, pages 219–224, 2002.

[72] Yaozu Dong, Dongxiao Xu, Yang Zhang, and Guangdeng Liao. Optimizing network i/o virtualization with efficient interrupt coalescing and virtual receive side scaling. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 26–34. IEEE, 2011.

[73] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. vic: Interrupt coalescing for virtual machine storage device io. In *2011 USENIX Annual Technical Conference (USENIX ATC11)*, 2011.

[74] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. *ACM SIGARCH Computer Architecture News*, 40(1):411–422, 2012.

[75] HaiBing Guan, YaoZu Dong, Kun Tian, and Jian Li. Sr-iov based network interrupt-free virtualization with event based polling. *Selected Areas in Communications, IEEE Journal on*, 31(12):2596–2609, 2013.

[76] NM Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[77] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM.

[78] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. Cloud control with distributed rate limiting. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 337–348. ACM, 2007.

[79] Liang Zhao, Ming Li, Yasir Zaki, Andreas Timm-Giel, and Carmelita Görg. Lte virtualization: from theoretical gain to practical solution. In *Proceedings of the 23rd International Teletraffic Congress*, pages 71–78. ITCP, 2011.

[80] Jose Flich, Samuel Rodrigo, Jose Duato, T Sodring, AG Solheim, Tor Skeie, and Olav Lysne. On the potential of noc virtualization for multicore chips. In *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 801–807. IEEE, 2008.

[81] Jan Heisswolf, Aurang Zaib, Andreas Weichslgartner, Ralf König, Thomas Wild, Jürgen Teich, Andreas Herkersdorf, and Jürgen Becker. Virtual networks – distributed communication resource management. *ACM Trans. Reconfigurable Technol. Syst.*, 6(2):8:1–8:14, August 2013.

[82] Jun Zhang, Kai Chen, Baojing Zuo, Ruhui Ma, Yaozu Dong, and Haibing Guan. Performance analysis towards a kvm-based embedded real-time virtualization architecture. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, pages 421–426. IEEE, 2010.

[83] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48. IEEE, 2011.

[84] Dominik Reinhardt and Gary 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. IEEE, 2014.

[85] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

[86] Robert Kaiser. Combining partitioning and virtualization for safety-critical systems. *White Paper WP CPV*, 10:A4, 2007.

[87] Holm Rauchfuss, Thomas Wild, and Andreas Herkersdorf. A network interface card architecture for i/o virtualization in embedded systems. In *Proceedings of the 2nd conference on I/O virtualization*, pages 2–2. USENIX Association, 2010.

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

[89] Dominik Reinhardt, Maximilian Güntner, and Simon Obermeir. Virtualized communication controllers in safety-related automotive embedded systems. In *Architecture of Computing Systems–ARCS 2015*, pages 173–185. Springer, 2015.

[90] Daniel Münch, Ole Isfort, Klaus Müller, Michael Paulitsch, and Andreas Herkersdorf. Hardware-based i/o virtualization for mixed criticality real-time systems using pcie sr-iov. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 706–713. IEEE, 2013.

[91] Daniel Münch, Michael Paulitsch, Oliver Hanka, and Andreas Herkersdorf. Mpiov: scaling hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (multi-core) multi-processor systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 579–584. EDA Consortium, 2015.

[92] Daniel Muench, Michael Paulitsch, and Andreas Herkersdorf. Temporal separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using pcie sr-iov. In *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pages 1–7. VDE, 2014.

[93] Oliver Sander, Steffen Baehr, Enno Luebbers, Timo Sandmann, Viet Vu Duy, and Juergen Becker. A flexible interface architecture for reconfigurable coprocessors in embedded multicore systems using pcie single-root i/o virtualization. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 223–226. IEEE, 2014.

[94] Duy Viet Vu, Timo Sandmann, Steffen Baehr, Oliver Sander, and Jurgen Becker. Virtualization support for fpga-based coprocessors connected via pci express to an intel multicore platform. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 305–310. IEEE, 2014.

[95] Christian Herber. Can controller virtualization for multicore processors. Master's thesis, Technische Universität München, 2012.

[96] Christian Herber, Andre Richter, Holm Rauchfuss, and Andreas 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.

[97] Christian Herber, Andre Richter, Holm Rauchfuss, and Andreas Herkersdorf. Spatial and temporal isolation of virtual can controllers. *ACM SIGBED Review*, 11(2):19–26, 2014.

[98] Christian Herber, Andre Richter, Thomas Wild, and Andreas Herkersdorf. Deadline-aware interrupt coalescing in controller area network (can). In *Embedded Software and Systems (ICESS), 11th IEEE International Conference on*, pages 693–700. IEEE, 2014.

[99] Christian Herber, Dominik Reinhardt, Andre Richter, and Andreas Herkersdorf. Hw/sw trade-offs in i/o virtualization for controller area network. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun 2015.

[100] Jürg Nievergelt and Edward M Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973.

[101] Giuseppe Buja, Juan R Pimentel, and Alberto Zuccollo. Overcoming babbling-idiot failures in can networks: A simple and effective bus guardian solution for the flexcan architecture. *Industrial Informatics, IEEE Transactions on*, 3(3):225–233, 2007.

[102] Ian Broster and Alan Burns. An analysable bus-guardian for event-triggered communication. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 410–419. IEEE, 2003.

[103] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[104] Haibo Zeng, Marco Di Natale, Paolo Giusto, and Alberto Sangiovanni-Vincentelli. Statistical analysis of controller area network message response times. In *Industrial Embedded Systems, 2009. SIES'09. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[105] 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.

[106] Mathieu Grenier, Lionel Havet, and Nicolas Navet. Pushing the limits of can-scheduling frames with offsets provides a major performance boost. In *4th European Congress on Embedded Real Time Software (ERTS 2008)*, 2008.

[107] Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of OSPERT*, pages 33–44, 2010.

[108] Oliver Hartkopp. *Programmierschnittstellen für eingebettete Netzwerke in Mehrbenutzerbetriebssystemen am Beispiel des Controller Area Network*. PhD thesis, Dissertation an der Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2011.

[109] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[110] Christian Herber, Andre Richter, Thomas Wild, and Andreas Herkersdorf. A network virtualization approach for performance isolation in controller area network (can). In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 215–224. IEEE, 2014.

[111] Gabriel Leen and Donal Heffernan. Ttcan: a new time-triggered controller area network. *Microprocessors and Microsystems*, 26(2):77–94, 2002.

[112] Gerd Niestegge. The leaky bucket policing method in the atm (asynchronous transfer mode) network. *International Journal of Digital & Analog Communication Systems*, 3(2):187–197, 1990.

[113] Neil C Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[114] Alan Burns and Robert I Davis. Mixed criticality on controller area network. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 125–134. IEEE, 2013.

[115] Christian Herber, Andre Richter, Thomas Wild, and Andreas Herkersdorf. Real-time capable can to avb ethernet gateway using frame aggregation and scheduling. In *Proceedings of the conference on Design, Automation & Test in Europe*, pages 61–66. European Design and Automation Association, 2015.

[116] Laurent George, Nicolas Rivierre, Marco Spuri, et al. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical report, Institut National de Recherche et Informatique et en Automatique (INRIA), 1996.

[117] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE, 1990.

[118] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.

[119] Christian Herber, Ammar Saeed, and Andreas Herkersdorf. Design and evaluation of a low-latency avb ethernet endpoint based on arm soc. In *Embedded Software and Systems (ICESS), 12th IEEE International Conference on*. IEEE, 2015.