# VEGa: A High Performance Vehicular Ethernet Gateway on Hybrid FPGA

Shanker Shreejith, *Member, IEEE*, Philipp Mundhenk, *Student Member, IEEE*, Andreas Ettner, Suhaib A. Fahmy, *Senior Member, IEEE*, Sebastian Steinhorst, *Member, IEEE*, Martin Lukasiewycz, *Member, IEEE*, and Samarjit Chakraborty, *Senior Member, IEEE*

**Abstract**—Modern vehicles employ a large amount of distributed computation and require the underlying communication scheme to provide high bandwidth and low latency. Existing communication protocols like Controller Area Network (CAN) and FlexRay do not provide the required bandwidth, paving the way for adoption of Ethernet as the next generation network backbone for in-vehicle systems. Ethernet would co-exist with safety-critical communication on legacy networks, providing a scalable platform for evolving vehicular systems. This requires a high-performance network gateway that can simultaneously handle high bandwidth, low latency, and isolation; features that are not achievable with traditional processor based gateway implementations. We present VEGa, a configurable vehicular Ethernet gateway architecture utilising a hybrid FPGA to closely couple software control on a processor with dedicated switching circuit on the reconfigurable fabric. The fabric implements isolated interface ports and an accelerated routing mechanism, which can be controlled and monitored from software. Further, reconfigurability enables the switching behaviour to be altered at run-time under software control, while the configurable architecture allows easy adaptation to different vehicular architectures using high-level parameter settings. We demonstrate the architecture on the Xilinx Zynq platform and evaluate the bandwidth, latency, and isolation using extensive tests in hardware.

**Index Terms**—Automotive networks, automotive Ethernet gateway, network gateways, field programmable gate arrays, FlexRay

## 1 Introduction

Modern vehicles are complex distributed computer systems where interactions between the vehicle and its driver or the environment are constantly monitored and controlled by a large number of embedded computing units (ECUs). These include time- and safety-critical functions like anti-lock braking systems, mixed-criticality features like driver assistance systems, and non-critical comfort features like infotainment systems. The different ECUs that implement these features in a distributed manner are interconnected using efficient time-triggered and event-triggered networks that provide high levels of determinism and reliability. ECUs are typically segmented into domains, such as the Drive-Train, Body, and Infotainment domains, depending on the criticality and bandwidth requirements of these functions, and are often served by different network protocols like FlexRay, Controller Area Network (CAN), or Media Oriented Systems Transport (MOST).

Multiple in-vehicle networks are normally interconnected by a central gateway (CG), allowing ECUs to communicate and share information with other ECUs in a different domain, as shown in Fig. 1. The gateway receives messages from one branch, performs translation from one protocol to another and broadcasts them on the corresponding target branches. CG implementations are usually processor-based, running software to perform the tasks of message reception, lookup, translation, and transmission [1].

For evolving applications like advanced driver assistance systems (ADAS) that integrate a large number of sensors and actuators, new network protocols with higher bandwidth interconnect are appearing. Recent developments point to Ethernet as a likely candidate for backbone connectivity, with Broadcom's BroadR-Reach physical layer chips shown to support 100 Mbps bandwidth reliably over unshielded cables [2]. However, vehicular systems will still depend on CAN and FlexRay for safety-critical functionality. The Ethernet backbone infrastructure illustrated in Fig. 2 is a viable solution for allowing existing systems to operate without modification, while providing high performance interconnect for functions that utilise information from existing systems as well as volume data sensors, which are connected through an Ethernet Gateway (EG). The EG would be an additional ECU on the network, with software

- *S. Shreejith and S. A. Fahmy are with the School of Engineering, University of Warwick, Coventry CV4 7AL, United Kingdom. E-mail: {s.shanker, s.fahmy}@warwick.ac.uk.*
- *P. Mundhenk, A. Ettner, and M. Lukasiewycz are with the Technical University of Munich Campus for Research Excellence and Technological Enterprise, 138602, Singapore. E-mail: {philipp.mundhenk, martin.lukasiewycz}@tum-create.edu.sg, a.ettner@mytum.de.*
- *S. Steinhorst is with the Department of Electrical and Computer Engineering, Technical University of Munich, Munchen 80333, Germany. E-mail: sebastian.steinhorst@tum.de.*
- *S. Chakraborty is with the Institute for Real-time Computer Systems, Technical University of Munich, Munchen 80333, Germany. E-mail: samarjit@tum.de.*

*Manuscript received 7 Sept. 2016; revised 8 Mar. 2017; accepted 20 Apr. 2017. Date of publication 1 May 2017; date of current version 14 Sept. 2017. (Corresponding author: Suhaib Fahmy.) Recommended for acceptance by K. Schneider. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2017.2700277*

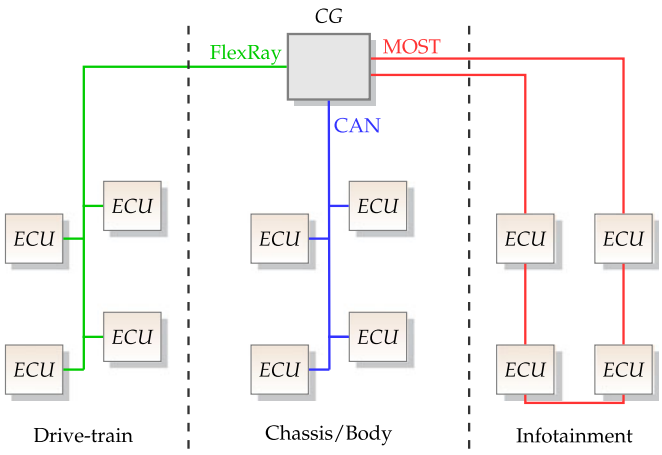


Fig. 1. Typical vehicular network structure with multiple network protocols interconnected by Central Gateway.

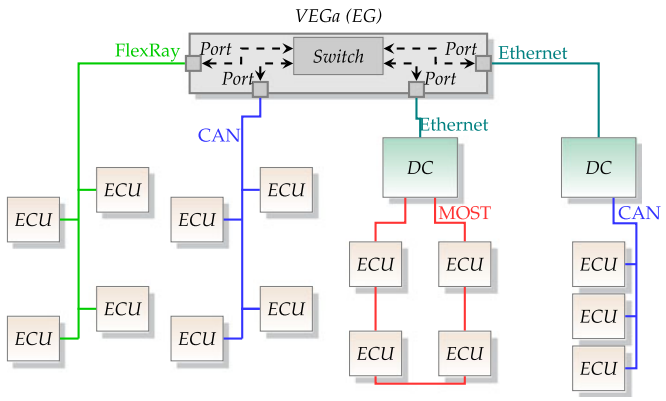


Fig. 2. Proposed vehicular network structure using the VEGa Ethernet gateway. Non-critical functions on legacy networks use the Ethernet backbone via domain controllers (DC), while critical functions are directly interfaced to the EG over corresponding networks.

control mechanisms allowing each branch of the network to be independently controlled and disconnected (if needed), to meet the reliability requirements of critical systems. Also, the interconnect must offer low-latency switching with priority-based routing to support exchange of mixed criticality messages across domains.

Though an ASIC implementation of the EG would provide performance, energy and cost benefits, ECU and network architectures often differ across a range of vehicle models and thus the gateway needs flexibility to accommodate the different architectures (existing and evolving). This has prompted the use of processor-based gateways in current vehicles, with software-based routing and hardened network interfaces to cater to multiple architectures [1], [3], [4]. Adaptability is also a requirement for integrating security standards into automotive embedded systems, which have a lifetime of 10 or more years. However, achieving real-time routing on mixed-criticality networks at high bandwidth is difficult on general purpose processors, and scales poorly with increasing network complexity and requirements (like security).

An alternative solution is the use of FPGA-based gateways that use custom architectures to provide accelerated computation and routing, while providing adaptability through reconfiguration. Researchers have explored FPGA-based gateways for legacy automotive networks, utilising software-based routing on soft processors [1] and with dedicated routing hardware [5]. Extending these architectures for Ethernet backbone networks is impractical since they are optimised for low-bandwidth legacy networks like CAN and FlexRay which have very different properties compared to standard Ethernet (priority-driven and time-triggered versus best-effort). FPGAs are widely used in general networking due to their ability to offer low-latency switching [6], [7]. Although a standalone Ethernet switch on an FPGA can cater to the performance requirements in vehicles, the switch architecture must be significantly adapted to support the deterministic nature and real-time requirements of automotive systems. Furthermore, the architecture must be modular to enable easy (parametric) adaptation to different network architectures that are employed in low/high-end vehicles.

Unlike generic Ethernet gateways, vehicular ECUs require some level of software intervention for monitoring, control, debugging and certification. On an FPGA, software control can be added using soft processors instantiated in the logic; however, they do not offer sufficient computational capability to implement complex algorithms with time-bound performance [8]. The FPGA fabric can also be connected as an extension of a standard automotive microcontroller unit (MCU) as an accelerator. This approach only provides a loose coupling between the function on the MCU and the accelerator, impacting overall latency, especially when there is frequent data movement between the two processing elements (MCU and accelerator). Tightly coupling the computational logic with the network interface can allow for extended functionality as well as enhanced capabilities [9].

In this paper, we present VEGa, a vehicular Ethernet gateway architecture. It exploits the tight coupling between processors and programmable logic in hybrid FPGAs like the Xilinx Zynq to provide a scalable switching architecture with software control. The reconfigurable fabric implements the communication interfaces and the switching infrastructure, which can be controlled and configured at run-time by the software on the processing system. Switching branches are designed in a modular fashion to allow adaptability to different in-vehicle network designs, by tuning configurable parameters at design time. The tightly coupled architecture and modular design offers pathways for integration of security approaches such as light weight authentication [10] in software or hardware [11]. It also ensures isolation between the communication channels and predicable latencies through hardware implementation, unlike the case with software-based switching. We evaluate VEGa on the Zynq ZC702 and ZC706 platforms to quantify switching performance (predictability and latency) and compare it against existing FPGA-based and processor-based automotive gateways as well as low-latency Ethernet switch architectures.

The contributions of this work are two-fold: First, it describes a modular infrastructure for Ethernet backbone automotive architectures, that can seamlessly exchange information between legacy protocols and high-speed Ethernet devices. Our evaluation shows that VEGa is capable of achieving low-latency and deterministic switching in a priority-aware manner, which can scale to complex networks and high network utilisation. Second, the extension and integration of a scheduling mechanism to translate

information between legacy networks and standard Ethernet, in a manner that is transparent to the networks involved. The remainder of the paper is organised as follows: Section 2 introduces the related work on automotive networks and gateways. In Section 3, we describe the architecture of VEGa and highlight the design considerations and optimisations. In Section 4, we present a performance evaluation of VEGa in actual hardware with different traffic conditions. Finally, we conclude the paper in Section 5 and outline future work.

## 2 Background

Early automotive systems used simple switches and actuators, and their functionality was achieved using point-to-point wiring. As more complex systems were introduced, point-to-point connections became infeasible due to the complexity of the wiring harness and the resulting additional weight and volume [12]. The early 1980s marked the introduction of vehicle networking as a step to reduce wiring costs and complexity. Bosch introduced the Controller Area Network (CAN) in the mid-1980s and it gained widespread acceptance in the automotive industry, later becoming the most widely used networking backbone for in-vehicle systems. CAN provides flexibility to the user, since it can be operated at multiple speeds and thus varied costs. For instance, low-speed CAN runs at 125 kbps and can cater to all the user oriented electronics in a car, like power windows, electric seats and air conditioning, while high-speed CAN can run at up to 1 Mbps, serving real-time and safety-critical applications like engine management, anti-lock braking system (ABS), and others.

Following CAN, a variety of in-vehicle networks evolved, driven primarily by cost and performance requirements. CAN proved too expensive and complicated for simple functions like power windows or boot release. Simpler schemes like the Local Interconnect Network (LIN) offered similar functionality at lower cost per module and power consumption, and thus found widespread adoption for non-critical functions. CAN also proved too slow for high bandwidth applications like multimedia in higher end vehicles resulting in the development of high bandwidth protocols like Media Oriented Systems Transport for such applications. As the number of communicating nodes in networks has increased, CAN has proven incapable of consistently providing deterministic data transfer rates, primarily due to its event-triggered architecture. Emerging safety-critical applications demand higher levels of determinism, which cannot be consistently ensured by event-triggered networks. The FlexRay protocol, developed by the FlexRay consortium, offers a combination of time-triggered and event-triggered communication for in-vehicle applications to enhance reliability with higher bandwidth [13]. Similarly Time-triggered Ethernet (TTE), has recently emerged for such applications. However, more widespread adoption of FlexRay and TTE is limited by the higher cost per node.

Computation in vehicle is segmented into different domains, and each domain uses network protocol(s) that closely match its requirements and properties [12], [14], [15]. Message exchange between different domains is enabled by a gateway ECU. With traditional automotive network protocols (like CAN, LIN and FlexRay), gateway ECUs are implemented on automotive microcontrollers with software-based routing algorithms to control message exchange [1], [16], [17], [18], [19]. Many papers describe gateway architectures without reporting key performance parameters like end-to-end latency measured through experiments. Processor-based gateway architectures that incorporate Ethernet have also been discussed in the literature [20], [21], [22]. Evaluations show that this approach cannot provide reliable and efficient switching at full throughput with large payload sizes. A simulation model for switched Ethernet-based in-vehicle networks is presented in [23] for evaluating topologies and to predict network latencies under realistic automotive conditions.

FPGAs are ideally suited for such mixed mode data exchange where custom datapaths can analyse traffic criticality and prioritise switching in real-time. FPGA-based ECUs have been proposed in the automotive domain for compute-intensive non-safety-critical functions like real time vision-based driver assistance [24], [25]. FPGA-based ECU architectures have also been explored for critical applications. In [26], the authors describe an architecture for implementing fail-safe safety-critical ECU systems on FPGAs leveraging dynamic reconfiguration (complete reconfiguration). The described architecture uses FPGA logic as a fail-safe back-up, which is reconfigured to one of the back-up modes when errors are detected. Partial reconfiguration (PR) has been used in non-safety-critical automotive applications such as driver assistance systems [27], [28]. In such cases, using PR can allow a reduction in the required target FPGA size by time-multiplexing different functionality.

Fault-tolerant ECU architectures have also been demonstrated for critical ECUs where PR was used to dynamically reconfigure a faulty network controller [29] and for integrating self-healing properties at the ECU-level [30]. A generic mechanism to ensure AUTOSAR compliance for FPGA-based ECUs is discussed in [31], using soft processors as microcontroller unit (MCU) replacements. Here, the AUTOSAR run-time environment is mapped to a register interface on the FPGA, providing the same functionality as standard AUTOSAR compliant MCUs.

Automotive gateways on FPGAs have been proposed in the literature [1], [5], providing deterministic message routing between traditional automotive networks like LIN, CAN, and FlexRay. In [1], a multi-FPGA architecture is used with a Xilinx Virtex-4 host platform that performs the switching and a Altera Stratix-3E daughter card implementing the interface protocols (CAN, LIN and FlexRay). Switching is performed using software running on the PowerPC core available on the Virtex-4 device. However, no experimental results like end-to-end latency or implementation results like resource utilisation are presented in the work for comparison with our approach. In [5], the authors present a modular gateway architecture with dedicated routing modules that allows accelerated switching between different interfaces. Their results show that custom architectures can provide deterministic routing between the different CAN networks even at high network utilisation.

FPGAs have been coupled with dedicated network host processors for reliable exchange of control information over time-triggered networks in critical systems like

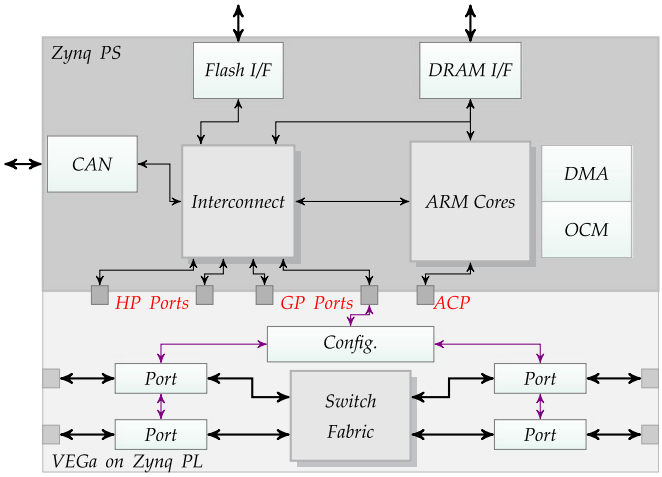


Fig. 3. Zynq architecture showing the processor subsystem (PS), programmable logic (PL), and the high-level architecture of VEGa in the PL region.

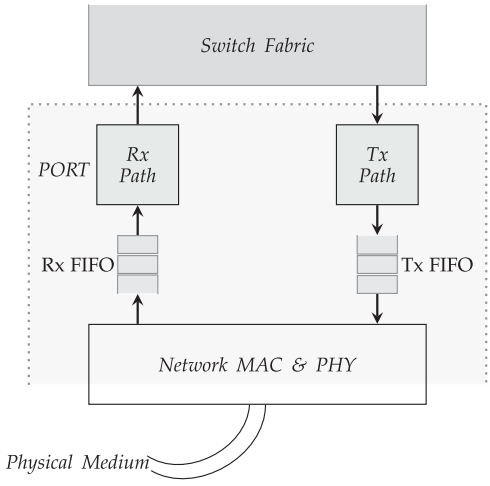


Fig. 4. High-level block diagram of a port.

X-by-wire [32]. Such work mainly explores the case for accelerating the routing of messages between legacy automotive networks, which have a maximum data-rate of 10 Mbps and packet size of up to 256 bytes (in the case of FlexRay), which can be statically defined. The introduction of high-speed Ethernet presents new challenges like variable message sizes, mixed criticality messages, and higher bandwidth, which must be interfaced with traditional automotive networks. Our approach explores the use of hybrid FPGAs to achieve deterministic message exchange in a heterogeneous network environment involving legacy networks and Gigabit Ethernet, modelling the scenario in future Ethernet-backbone vehicular architectures.

In the case of generic networks, FPGAs have been widely employed in line-rate switching systems for high performance Ethernet [7], [33], where custom architecture design offers low-latency switching performance. FPGA-based custom network interface modules were shown to offer improved switching latency over off-the-shelf components in Ethernet networks supporting real-time high-bandwidth communication [34]. Customisable datapaths allow FPGA-based switches to analyse traffic during the switching operation [35], which can also be extended to incorporate some level of security like intrusion detection [36], [37]. Our approach aims to bring such high-performance, low-latency switching to the multi-standard mixed-criticality network structure used in the automotive domain.

## 3 The VEGa Architecture

### 3.1 Overview of Zynq FPGAs and VEGa on Zynq

The Zynq family from Xilinx are hybrid reconfigurable devices that offer tight integration of a capable processing system with configurable programmable logic on the same die, as shown in Fig. 3 [38]. The PS is a hardened region of the die that combines a dual-core ARM Cortex-A9 processor along with several memory and connectivity interfaces. The Cortex-A9 along with its memory subsystem is capable of hosting a fully-fledged operating system like Linux and can operate as a standalone device without any support from the PL, providing a familiar environment for embedded software developers. The connectivity to peripheral blocks is established through ARM's AMBA eXtensible Interface (AXI) interconnect. This wide array of interfaces in the PS makes it ideally suited as a hardware platform for a highly connected embedded system.

The functionality of the PS can be further extended with custom logic in the PL region. The Zynq offers high bandwidth interconnect between the PS and PL. Furthermore, dedicated direct memory access (DMA) blocks enable high-speed data movement between the PL and interfaces managed by the PS, like DRAM or the Ethernet interface. The PL is based on the Xilinx 7-series architecture, combining flexibility features like partial reconfiguration, advanced computational capabilities (like advanced DSP48E1 blocks) and lower power consumption. The hybrid architecture enables scalable and parallel implementations of complex processing blocks in the PL, while retaining software-based control through the tightly coupled ARM cores [8]. As shown in Fig. 3, VEGa is completely contained within the PL region of the Zynq, whose behaviour can be controlled dynamically through software running in the PS.

### 3.2 System Architecture of VEGa

At a high-level, VEGa instantiates multiple physical ports and a priority aware switching fabric that allows information to be exchanged between the different physical ports, as shown in Fig. 3. The top-level architecture of VEGa offers a configurable set of parameters (in the Verilog description) that control the number and type of network interfaces that need to be implemented. Multiple physical switching port combinations can be implemented by configuring a parameter (4 in the default case, numbered 1 to 4) during the physical implementation phase (FPGA design phase), each capable of providing up to 1 Gbps throughput. Each port implements independent transmit (Tx) and receive (Rx) paths to handle connections from the network interfaces to the switch fabric, as shown in Fig. 4. FIFOs embedded within the paths help to decouple the network interface from the switch fabric. The network interface logic is responsible for implementing the communication protocol, and interfaces the gateway through its corresponding *port*. The network interfaces can be Gigabit Ethernet, FlexRay, or CAN, which are also configured using the top-level parameters, and for our experiments we use Gigabit Ethernet and

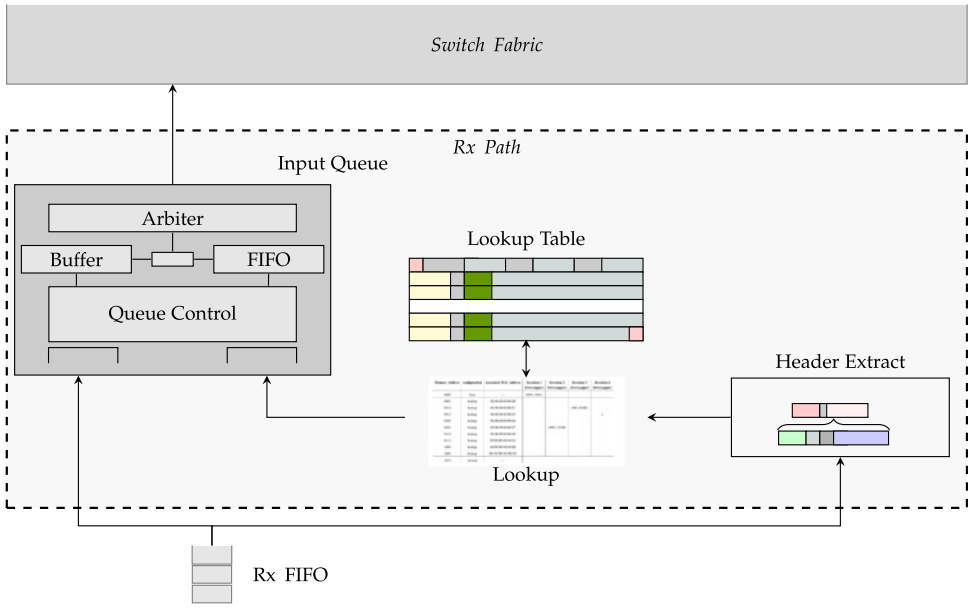


Fig. 5. Receive path of a port with functional sub-modules.

FlexRay interfaces. The number of branches and interface types are defined using top-level (Verilog) parameters, which can be altered for different configurations (during the FPGA design phase). All forwarding decisions are based on the Ethernet layer-2 headers, with each non-Ethernet ECU having a virtual mapping in the medium access control (MAC) address space.

### 3.2.1 Receive Path

The receive path buffers incoming frames and makes the forwarding decision based on the Ethernet MAC header. It employs three modules that operate on-the-fly on received frames as they are passed up to the switching infrastructure. The *header extraction* logic determines the header segment, classifies the frame and passes the information to the *lookup* module which determines the output port for the frame in the form of a binary vector (called the *port vector*). The *input queue* acts as a temporary buffer for incoming packets, before they are forwarded to the switch logic. These modules operate in parallel, allowing the destination port for a packet to be determined before the complete frame has been received, even for the smallest allowed payload size. The high-level block representation of the receive path is shown in Fig. 5.

The *header extraction* module determines the start of each frame and extracts the destination MAC address, virtual LAN (VLAN) tag, and frame priority from the frame header as they are received. With the first data byte, it also records the 64-bit arrival timestamp for the packet (called the *ingress timestamp*), with a resolution of 8 ns (125 MHz). This timestamp is used by later stages of the logic to ensure latency-based routing and also by the management interface at higher layers to determine performance and worst case delays. The extracted header information is passed to the *lookup* module to determine the destination port mapped to the destination MAC address.

The *lookup* module implements a binary search on the sorted list of MAC address values to determine the destination port for a given address. Since all possible destinations are predefined in an automotive system, a sorted table structure presents a more efficient scheme for look up than specialised associative mechanisms like a content-addressable memory. Use of binary search allows the 1,000 MAC entries in the table to be searched within a maximum of $\log_2 1000 \approx 10$ clock cycles. The lookup memory contains three types of memory entries; the *configuration entry*, the *lookup entries*, and the *default entry*, as shown in Fig. 6.

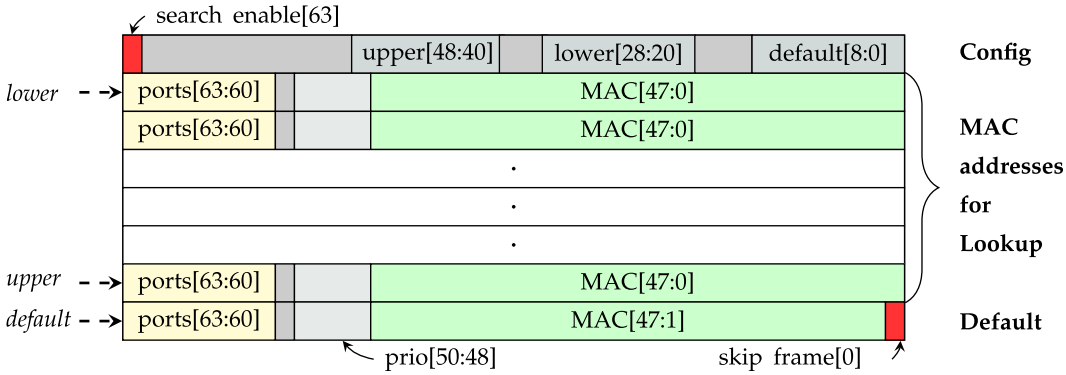


Fig. 6. Lookup table structure in the receive path.

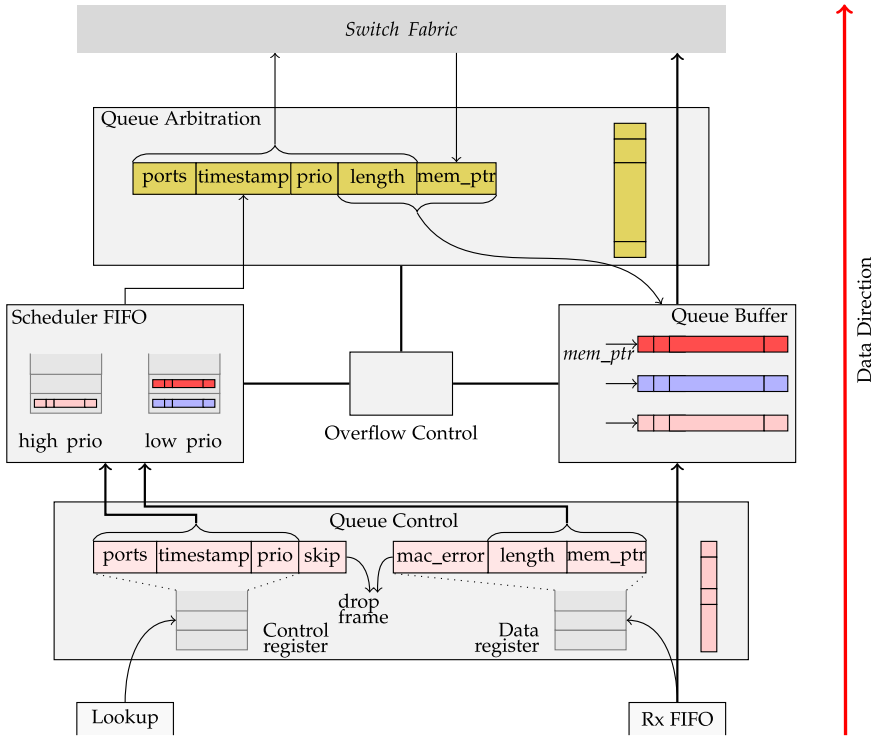


Fig. 7. Input queue block diagram: Each frame in the memory has a corresponding entry in one of the FIFOs.

The *configuration entry* is the first in the lookup table and is read by default before each search is initiated. It provides information about the lookup memory organisation; the *lower* field indicates the lowest MAC address, the *upper* field indicates the highest address in memory and the *default* field specifies the default destination port in case no entries are matched. The *search_enable* bit indicates the status of the lookup table and if set to zero, indicates that the lookup should not be performed, forcing the use of the *default* setting (which could be to drop the frame). This bit can also be used to isolate a branch, in the case of persistent faults, allowing communication from other ports to be handled without introducing errors.

If enabled, the search algorithm operates on the *lookup entries*, the sorted array of MAC addresses and their destinations in MAC address order. Each entry has a 48 bit destination MAC field, the associated output port vector and a 4 bit priority value. During the lookup, if a match is found between the incoming MAC address and the location, then the corresponding port vector is used. The priority field indicates the priority that will be allocated to the frame if it is not VLAN-tagged, else this field is ignored. The *default* configuration also uses the same structure as the *lookup entries*, but uses a *skip_frame* bit, which, when set, forces the input path to drop the frame instead of forwarding it.

The *input queue* module buffers incoming frames until they can be transmitted to the output port, and is composed of five sub-modules: Queue control, queue buffer (labelled as *Buffer* in Fig. 5), scheduler FIFO (labelled as *FIFO* in Fig. 5), overflow control logic and the queue arbitration logic (labelled as *Arbiter* in Fig. 5). The high level organisation of the different sub-modules is shown in Fig. 7.

The *queue control* module interfaces the lookup module with the Rx FIFO. It initiates a read from the Rx FIFO when a frame becomes available, and a write into the *queue buffer*, saving the location of the first write as the *mem_ptr* for that frame. Locations are dynamically allocated (next free space) in the *queue buffer* which is configured as a ring buffer. Further, the *length* of the frame and the error indications from the MAC (*mac_error*) are also saved when an entire frame is received. Once the lookup information for the corresponding frame is available, and if there are no errors (*mac_error* and *skip_frame* are both 0), the *mem_ptr* and *length*, along with the lookup information (*port vector*, *prio*) and timestamp are written into one of the *scheduler FIFO* blocks, depending on the frame priority (i.e., highest priority frame written to the highest priority FIFO and so on). In the present system, we have only two possible priority settings (real-time traffic and non-real-time traffic) and thus only two *scheduler FIFOs* are used, but this is modifiable at design time.

The *queue arbitration* logic forms the interface to the switch fabric and constantly monitors the *scheduler FIFOs* for new entries. If an entry is available in any of them, a read is issued to the highest priority *scheduler FIFO* among the ones which have an entry. The arbitration logic requests access to the switch fabric for the destination ports, along with the priority and timestamp information. The switch fabric acknowledges the request allowing the data to be forwarded. In the case of multi-port forwarding, the switch fabric can issue a partial acknowledge for only a subset of the requested ports. For example, if the request was 0111 (i.e, access to ports 3, 2, and 1), the switch fabric may approve only ports 3 and 1.

If at least one port is acknowledged, and no higher-priority frames have arrived at the *scheduler FIFO*, the arbiter initiates a read from the *queue buffer*. At the end of transmission, the arbitration logic updates its port request vector by setting the acknowledged port to '0' (i.e., in the

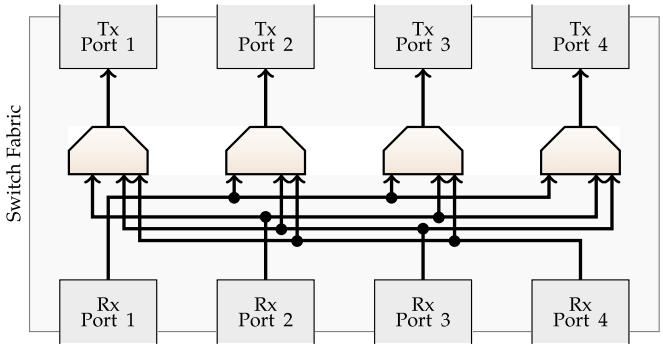


Fig. 8. Crossbar matrix implementation of the switch fabric.

above case, the vector now is reset to 0010), indicating that the frame still needs attention. If no higher priority frames are available, the updated port vector is presented to the switch to request transmission to port 2. Once all ports have been acknowledged and transmission is complete, the vector is updated to 0000 and the arbiter waits for the next frame. However, if a higher priority frame becomes available in memory, the updated port vector (0010) and its corresponding *timestamp*, *prio*, *length*, and *mem_ptr* signals are buffered, and the higher priority frame is serviced first. Once all higher priority frames are served, the arbiter serves the buffered frame.

The *overflow control* logic monitors the *scheduler FIFOs* and the circular *queue buffer* for overflows. Since frame transmissions could be deferred in case of higher priority traffic, it is possible for incoming frames to overwrite a lower priority frame. The overflow control logic tracks such issues and drops stale frames by removing buffered entries in the arbitration logic (or its corresponding entries in the FIFO) in favour of incoming data. To ensure that such cases are minimised in normal operation, the *queue buffer* is designed with sufficient depth to handle multiple outstanding frames and incorporates a wider 4-byte interface to the switch interface (compared to the byte-wide interface to the Rx FIFO) for providing 4× higher read-bandwidth.

### 3.2.2 Configurable Switch Interconnect

The central element of the gateway architecture is the configurable crossbar switching interconnect, allowing multiple ports to be active simultaneously. Fig. 8 shows the architecture of our configurable switching interconnect. Each receive interface can be connected to any transmit path except its own. To guarantee latencies we use a *strict latency* arbiter that selects the next port based on the priority value (*prio*) of the frame (range 0 to 7, 7 being highest) to be transmitted. The priority of frames is determined during the scheduling process (offline) and loaded into the *lookup* module in the receive path of each port.

At each clock cycle, the arbiter checks the states of all receive interfaces to see if they are requesting a connection to a transmit path. If any request is active, the arbitration module checks if the connection can be established to the corresponding transmit path. If the transmit path is free and can hold another frame, the arbitration module enables the connection and initiates transfer of the frame, along with the status information of the frame like *priority*, *timestamp* and *length*. If multiple ports request access to the same transmit path, the arbiter chooses the receive port with the higher priority frame and enables this connection. Simultaneously, the arbiter can also establish another non-intersecting path, allowing multiple ports to be connected in the same cycle. Once the frame has been transferred, the arbitration module releases the connection and enables any pending requests to this path, if the transmit path has enough space to hold the frame.

Integrating the fabric arbitration module into the switch logic, rather than the interfaces, allows the design to be scaled more easily as the necessary paths can be scaled for any number of connections. For example, upgrading to a 6-port switch requires minor changes to the switch fabric and replication of transmit and receive interfaces.

### 3.2.3 Transmit Path

The transmit path receives frames from the switch fabric, buffers them and schedules the output messages according to their priority. The functionality is implemented in multiple sub-modules of the output *queue control* logic, which have similar functions to those in the receive path. These sub-modules *queue memory monitor*, *scheduler FIFOs*, *queue buffer*, and *queue arbitration* logic along with their operations are shown in Fig. 9.

When a new frame is to be received into the transmit path from the switch logic, the *queue memory monitor* logic examines the output *queue buffer*, and acknowledges the transfer, if the entire frame can be buffered. Depending on the priority of the arriving frame (*prio*), the frame data and its control information are directed to the appropriate priority *queue buffer* and *scheduler FIFO*. The memory pointer corresponding to the first data word is appended to the received control information and stored in the corresponding priority FIFO, once a complete frame is received.

The output *queue arbitration* logic forms the interface to the physical transmit path and constantly monitors the *scheduler FIFOs* for available frames. When the Tx path MAC is ready to accept a frame and a frame is ready for transmission, the arbitration logic fetches the highest priority control information and the data corresponding to that entry from the corresponding output *queue buffer*. With the last data word, the *queue arbitration* signals an end-of-frame to the MAC using the *last_word* signal, and starts arbitration for the next frame. The MAC then performs the required protocol operations and pushes the data to the PHY to drive the physical network.

The *queue arbitration* takes a timestamp corresponding to the first data write to the Tx FIFO (called the *egress timestamp*), marking the completion of the switching process. The module computes the switching latency as the difference between the *egress timestamp* and *ingress timestamp* (from the control information) of the frame, which is then passed to the VEGa monitoring module for software-level monitoring.

### 3.2.4 Translation for FlexRay/CAN Systems

Unlike Ethernet, which uses explicit addresses to identify destination and source nodes, automotive networks are based on a broadcast scheme with no explicit identification for sources/sinks. The priority of messages in CAN and

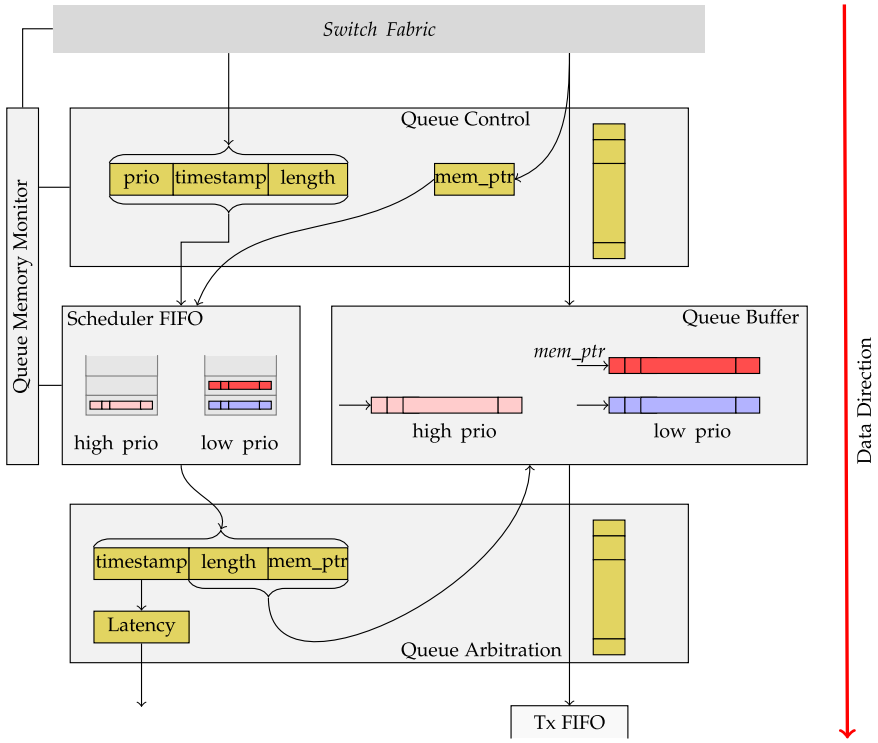


Fig. 9. Block diagram of the output queue module and its sub-modules.

assigned transmission slots in FlexRay are statically defined parameters that can implicitly identify a source node. Further, a translation scheme is required to determine the destination port on the switch interface and to manage message mapping due to different payload sizes and priorities.

The *tblock* handles this message translation from the switch to CAN/FlexRay networks – it includes:

- An address mapping scheme that relates the implicit identifiers on CAN or FlexRay to destination ports/addresses on the gateway.
- A data-packing scheme that respects the deadlines and payload sizes on the CAN/FlexRay networks for mapping messages.
- A stream-based data interface to the Tx/Rx path of the gateway from the CAN/FlexRay Communication Controllers.

In this paper, we present the approach for FlexRay (discussed below), which can be modified for application to CAN networks.

*Mapping Logic*: For the FlexRay static segment, a purely time-triggered scheme is used where messages are assigned fixed slot(s) in every FlexRay cycle. This means that a receiving ECU can subscribe to a set of slots on which messages are scheduled, creating an implicit addressing scheme.

Policy-based scheduling [39] provides a mechanism to translate this implicit addressing scheme, and is employed in our gateway. It allows packing of event-triggered messages into time-triggered FlexRay slots, with consideration for their real-time deadlines.

As with standard FlexRay, messages are analysed at design time when generating the message schedule for the FlexRay network. This schedule is used by all participating nodes on the FlexRay network, including the gateway's FlexRay interface. The static schedule assigns transmission slots to all nodes and allows normal FlexRay interfaces to subscribe to pre-determined slots to receive messages from the network. These slots, however, do not directly contain messages, but are combined to form a virtual communication channel per node (see Fig. 10).

To send messages, a node participating in policy-based FlexRay needs to add a header to every message before passing them to the virtual communication channel. As the virtual communication channel abstracts away slots, the implicit addressing scheme following these slots is lost. Nodes can thus not identify where messages begin and which type of message is received. To combat this, the message header contains the length of the message, allowing receivers to identify beginnings and ends of messages as they arrive. Furthermore, a message type identifier is added to allow identification of the kind of message received. This needs to be performed for

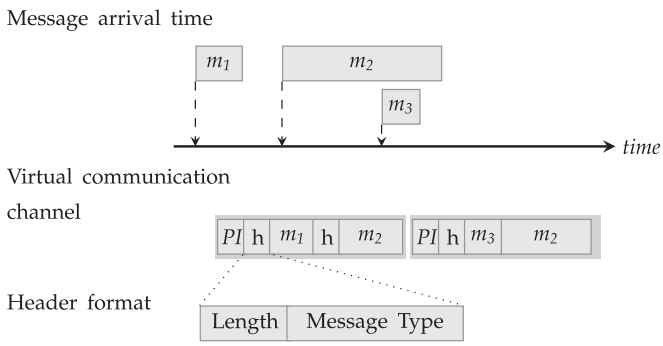


Fig. 10. Message transmission in policy-based FlexRay via packing in the virtual communication channel. Preemption indicator (PI) and headers, including message length and message type identifier, need to be added by the transmitting node. Messages can be split across slot borders and preempted. Reproduced from [39].

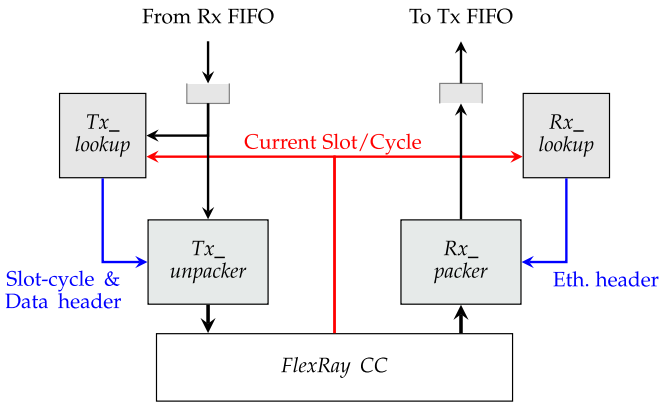


Fig. 11. Architecture of the tblock and its integration with the FlexRay communication controller (CC).

every message sent via policy-based scheduling. Every slot further contains a Preemption Indicator. This PI allows preemption of long messages by high priority messages and can be used to implement Ethernet priority schemes, such as Time-Sensitive Networking (TSN).

When the gateway is forwarding messages from Ethernet to FlexRay, it acts as a sender on the FlexRay network. In this case, message lengths are straightforward to fill for the gateway, based on the received message length on the Ethernet branch. The message identifier is based on the message received. Here, a translation based on a virtual MAC address is used. The virtual MAC address corresponds to a data type and a set of receivers and which is pre-loaded into the *tblock* (details in *tblock* architecture discussed below) together with the FlexRay message type identifier. This assignment is highly dependent on the content of the message and is done at design time, similar to message IDs in other communication systems (e.g., CAN). After translating the virtual MAC address into a message type identifier, the message is sent via the virtual communication layer in the next available time slot(s) (see Fig. 10).

Similarly, when the gateway is forwarding messages from FlexRay to Ethernet, the messages need to be translated. Here, the message length from the policy-based FlexRay header is used as the content length for the Ethernet message. Furthermore, the message type needs to be translated to a virtual MAC address, used as the receiver address when transmitting the message on Ethernet. The sending MAC is the MAC address of the gateway.

Through the use of conventional FlexRay time slots for policy-based messages, policy-based FlexRay is compatible with the FlexRay standard. Thus, devices which do not implement policy-based scheduling and communicate only via plain FlexRay, do not need to be altered to be used in this setup.

*Architecture of the tblock*: The *tblock* comprises *Tx_lookup* and *Rx_lookup* memories, an *Rx_packer* module to handle FlexRay → Ethernet messages, and a *Tx_unpacker* module to handle Ethernet → FlexRay messages, as shown in Fig. 11. The FlexRay parameter *KeySlotID*, which is a unique slot assigned to each node by the schedule, is used as the transmit/receive identifier for the data-headers, forming a one-to-one mapping to a virtual MAC address. These mappings are generated offline (during scheduling) and preloaded into the *Tx_lookup* and *Rx_lookup* memories of the *tblock*.

When a frame is received from the switch for forwarding on the FlexRay network, the *tblock* buffers the frame and strips the Ethernet headers. A lookup is performed in the *Tx_lookup* memory to determine the corresponding data-header and this is appended to the frame data. The list of transmit slots assigned to the gateway's FlexRay interface is then sorted based on the current slot-cycle in progress on the network. The frame along with the sorted list is forwarded to the *Tx_unpacker* module, which then segments the data (if needed), adds FlexRay frame headers and writes to the FlexRay interface's frame buffers. As the data segment of the frame is temporarily buffered in the *Tx_unpacker* module, a configurable pattern detector examines the frame to detect the presence of the specific pattern in the data segment, like an ECU configuration request. If detected, the *tblock* module can interrupt the software running on the ARM core to monitor the source port and to disable the sender address (or the port itself) in case of a security threat.

On the receive side, the messages received from the FlexRay network are handled by the *Rx_packer* module. At every slot/cycle boundary, the *tblock* performs a prefetch on the *Rx_lookup* memory to determine if there are any Ethernet addresses mapped to the slot-cycle combination that has just ended. Since the FlexRay protocol requires the frame to be completely received before validating it, the prefetch operation enables *Rx_packer* to prepare the Ethernet container before the FlexRay frame is completely received from the network. The FlexRay payload is directly filled into the Ethernet container and presented to the Rx Path of the port. The *Rx_packer* also appends the receive packet with '0's if the FlexRay frame does not satisfy the minimum payload size for Ethernet.

We have integrated our extensible FlexRay communication controller (CC) [9] as the network interface for one of the ports. The host interface of the FlexRay CC has been altered to a streaming interface to the *tblock*, based on the AMBA Advanced eXtensible Interface (AXI) streaming interface standard. FIFOs are instantiated at the interface to decouple the *tblock* from the CC's clock and data rates. The configuration of the protocol parameters is now integrated using a state machine that interfaces over a separate AXI bus. The isolation of the configuration and datapath interfaces of the CC enable a generic *tblock* architecture that can be directly reused for other network protocols like CAN. Further, security aware FlexRay interface(s) (such as the one discussed in [11]) could be integrated into the port(s) and extended using software-security schemes to ensure that the safety-critical network segments are protected from malicious attacks.

#### 3.2.5 Run-Time Management of Interface Configurations

The lookup memory within the receive path of each port is mapped as an addressable location from the ARM core on the Zynq device. The mapping allows the configuration of each individual port to be altered in isolation or to update all tables in a single write operation. The routing paths, message priorities as well as the behaviour of the switching system can thus be controlled from the monitoring software on the ARM core. The lookup memory within the *tblock* is also mapped to the address space, allowing changes to be made

TABLE 1
VEGa: Resource Consumption on Zynq XC7Z020

| Function | Submodule | FFs | LUTs | BRAMs | DSPs |
|---|---|---|---|---|---|
| FlexRay Port | CC | 5,572 | 9,768 | 20 | 2 |
| | Tblock | 2,227 | 1,849 | 13 | 0 |
| | Rx_Path | 662 | 492 | 4 | 0 |
| | Tx_Path | 160 | 121 | 5 | 0 |
| | Total | 8,576 | 12,230 | 42 | 2 |
| | Frequency | | 80 MHz | | |
| Ethernet Ports ×3 | MAC | 2,619 | 1,851 | 1 | 0 |
| | Rx_Path | 661 | 476 | 4 | 0 |
| | Tx_Path | 160 | 121 | 5 | 0 |
| | Total | 3,549 | 2,559 | 10 | 0 |
| | Frequency | | 125 MHz | | |
| Switch | - | 204 | 856 | 0 | 0 |
| | Frequency | | 125 MHz | | |
| Total (%) | | 19,585 (18.4) | 21,095 (39.7) | 72 (54.75) | 2 (0) |

in the FlexRay/CAN message routing. In addition, the software on the processor can also monitor the latency of packets on the individual interfaces to further fine-tune the routing performance and to isolate paths in real-time. We show the effectiveness of this run-time management using a case study that detects a suspicious activity and isolates the corresponding port by reconfiguring the lookup memory content in Section 4.1.

## 4 Evaluating VEGa

To evaluate the switching performance of VEGa, we have implemented the architecture on both the ZC702 development board and the ZC706 board featuring two different Zynq devices. The ZC702 board features a smaller Xilinx Zynq XC7Z020 device that incorporates an Artix-7 grade fabric, while the ZC706 board features a XC7Z045 device that incorporates a superior ARM core and a Kintex-7 grade fabric. For the evaluation, we have chosen a design that incorporates 3 Ethernet ports and 1 FlexRay port. The Ethernet port interfaces are connected through the FPGA Mezzanine Card (FMC) interface to two Gigabit Ethernet FMC cards (FMCL-GLAN-B) from Inrevium Inc. The resource utilisation on the XC7Z020 device is shown in Table 1. As can be observed, VEGa consumes 55 percent of the resources on a small device, and can easily be extended to support more ports. On the larger XC7Z045 device, the maximum resource consumption is under 15 percent (BRAMs) for the same combination (20,191 LUTs, 20,741 FFs 76 BRAMs, 2 DSPs). The superior programmable fabric on the XC7Z045 offers higher performance and is thus a better implementation platform for supporting more interfaces (ports). On both platforms, we are able to achieve the required operating frequencies for the different modules: 80 MHz for the FlexRay port and 125 MHz for the Ethernet ports and the switch module. For our experiments, we have configured the FlexRay interface for 10 Mbps data rate (80 MHz clock, to support the 8× serial redundancy required by the FlexRay standard [13]) and the Ethernet links were set for 1 Gbps throughput (125 MHz clock).

To observe the end-to-end latencies of VEGa under cross-traffic and isolated traffic conditions, we use a test setup

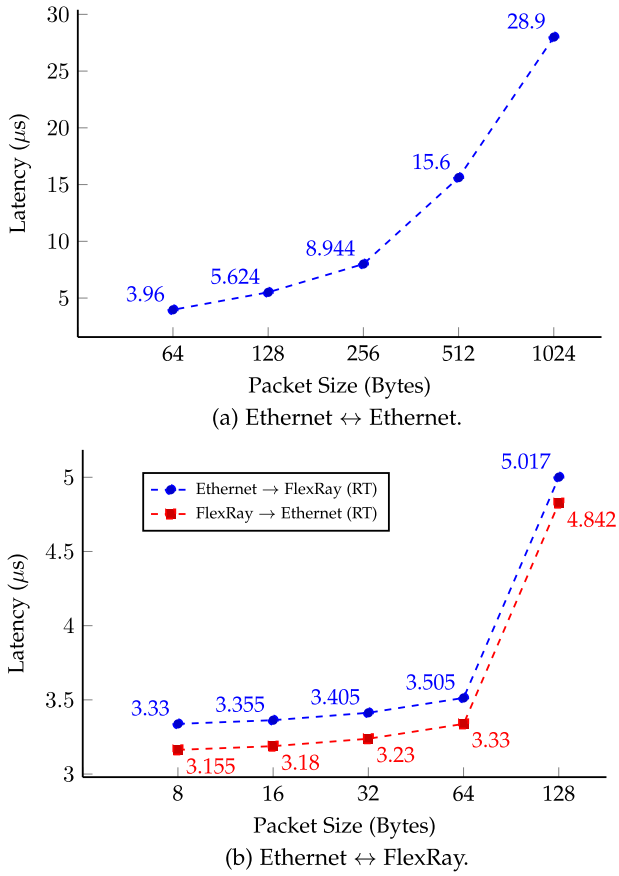


Fig. 12. Switching bandwidth of VEGa for different data-sizes in absence of cross traffic: Plot (a) corresponds to non-real-time ethernet↔ethernet traffic, while plot (b) corresponds to real-time ethernet↔FlexRay traffic.

with 3 Ethernet links and one FlexRay link on the larger XC7Z045 device (ZC706). The Ethernet ports are connected to independent and isolated traffic sources and sinks which are implemented on separate AC701 and VC707 FPGA development boards. Each link is capable of handling traffic at 1 Gbps. For FlexRay, we have integrated a small cluster of nodes on the ZC706 device that also includes the FlexRay interface of VEGa. This allows us to measure end-to-end latencies in all possible combinations: Real-time network ↔ Non-real-time network, Non-real-time ↔ Non-real-time and Real-time (FlexRay/Ethernet) → Real-time (Ethernet) ← Non-real-time cross traffic. The frame sizes and rates for our evaluation were generated based on the case studies in [40] and in [7] for FlexRay and Ethernet networks respectively.

Fig. 12 shows the average latency incurred by VEGa for different data sizes in the absence of cross traffic for non-real time Ethernet ↔ Ethernet traffic (Fig. 12a) and real-time FlexRay ↔ Ethernet traffic (Fig. 12b). The latency is computed end-to-end: From the start of frame transmission at the transmitter to the frame header reception at the receiver. For larger data sizes, we see an increase in latency due to the increased data movement within the switch, the transmit and receive interfaces of the gateway and at the source and sink interfaces. In the absence of cross traffic, we observe that the maximum variation in latency is about 40 ns and is insignificant compared to the end-to-end latency values.

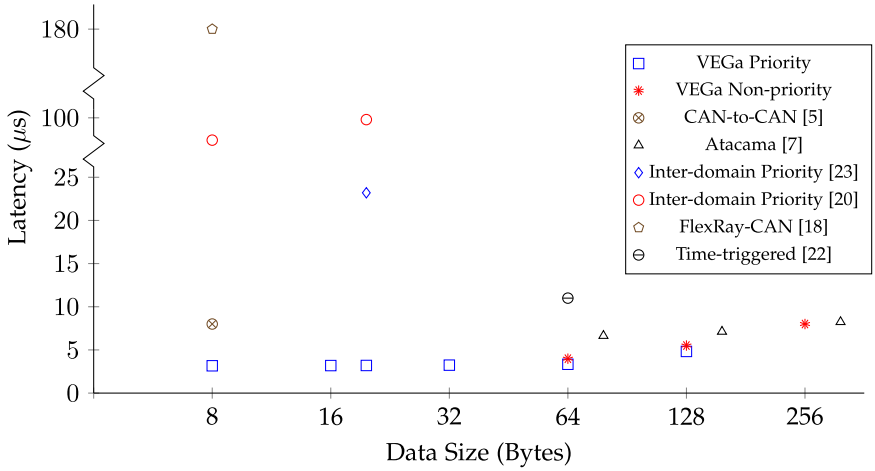


Fig. 13. Comparison of end-to-end latencies of VEGa with other implementations from literature.

For Ethernet → FlexRay transfers (Fig. 12b), the measurement is terminated at the FlexRay interface, since the transmission of the message on the FlexRay network is guaranteed by the policy-based schedule. It can be observed that there is no appreciable variation in latency below the 64-byte frame size, since the *tblock* pads smaller frames to meet the minimal Ethernet frame size requirement. For 128-byte data, we observe a slightly increased latency due to the larger data size. For data packets forwarded from FlexRay to an Ethernet link, the prefetch mechanism helps to reduce lookup latencies compared to the Ethernet → FlexRay transfers.

Fig. 13 shows worst case VEGa latency compared with existing work in the literature, in the absence of cross traffic. As observed, our architecture outperforms gateway structures based on software-based approaches (Lim et al. [23] {simulation results}, Kim et al. [20], Yang et al. [18], and Müller et al. [22]) and FPGA-based gateways for traditional networks (Sander et al. [5]). The proposed architecture also provides over 300× lower latency (priority mode, 8 byte message) than automotive-grade microcontroller-based gateways between traditional networks (LIN, CAN, and FlexRay) described in Schmidt et al. [17], Kim et al. [16] and Seo et al. [19]. Our architecture also outperforms the FPGA-based Ethernet switching infrastructure Atacama (Carvajal et al. [7]), though the margin is small (1.3× lower latency at 128-byte priority data). Further improvements in latency can be achieved by enhancing the Ethernet MAC/PHY modules (over the standard MAC/PHY modules that we have used in our platform) at the expense of increased hardware resources and limited portability [34].

We also evaluate the performance of VEGa in the presence of cross traffic. For this evaluation, priority and non-priority traffic were directed to the same destination at an aggregate bandwidth that nearly saturates the gateway. The setup generates non-priority traffic at 600 Mbits/s with a 1 KB payload size and variable rate priority traffic (10-200 Mbits/s) with a 64 byte payload. Fig. 14 shows the variation in latency in the presence of cross traffic, measured with long duration tests. It can be observed that additional (and varying) latency is incurred in the case of priority frames compared to the fixed deterministic latency in the absence of cross traffic. This is due to the non-preemptive nature of the switch fabric that blocks the priority frame once a non-priority frame has entered the switch, resulting in a maximum end-to-end latency of 21.3 $\mu$s for priority data. In the case of non-priority traffic, the smaller size of the priority frame causes only a minor increase in end-to-end latency as the blocking period (due to the real-time frame) is much shorter than the transmission latency of the non-priority frames. When the rate of priority frames reaches 200 Mbits/s, we observe that the non-priority frames start accumulating within the Rx port buffers, which eventually leads to dropped frames. However, in the same conditions, the priority frames were routed without any data loss, ensuring that critical data is always delivered to the destination. In comparison, the Atacama switch achieves better performance in cross traffic conditions due to its dedicated routing structure for priority traffic. However, this higher performance is achieved at the expense of increased resource consumption and lower scalability: A 4-port Atacama switch consumes 19,223 LUTs and 138 18K BRAMs on a Virtex-2 device (LUT-4 architecture) compared to 10,422 LUTs and 40 BRAMs (32×36 K BRAMs and 8×18 K BRAMs) used by VEGa on a low-end Zynq ZC7020 (LUT-6 architecture). Furthermore, for a higher number of ports, the dedicated routing resources in Atacama would increase complexity and resource consumption significantly compared to VEGa.

### 4.1 Case Study: Managing Ports in Real-Time

Researchers have identified vulnerabilities in current automotive gateways that allow a *reconfigure* command from a low priority network to force a configuration change on a

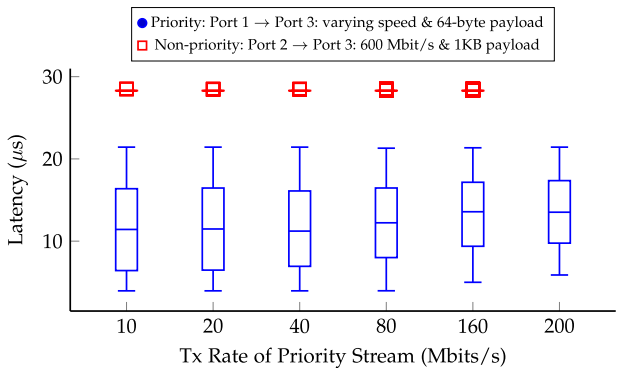


Fig. 14. End-to-end latency measurements using VEGa in the presence of cross-traffic.

TABLE 2
VEGa: Latency Incurred in Run-Time Management of Ports

| Block Mode | Latency Components | | | Total |
|---|---|---|---|---|
| | Routing | Interrupt | Software | |
| | $\mu$s | $\mu$s | $\mu$s | $\mu$s |
| Src/Dest MAC | 5.101 | 2.96 | 0.3 | 8.361 |
| Port forwarding | 5.101 | 2.96 | 21.25 | 29.311 |

higher priority network [41], [42]. This bridging enables a simple exploit on a low criticality ECU to gain access to safety-critical network segments and to alter the software code on safety-critical ECUs. The software monitoring on VEGa can respond to such threats dynamically by isolating the compromised segment of the network, preventing the attack from spreading.

For this experiment, we create a remote reconfiguration message from a low priority Ethernet port to a safety-critical ECU on the FlexRay network. The message is configured as a sequence of 64-byte messages with a specific identifier that can trigger a software update sequence on the ECU. The pattern detector extension in the *tblock* is configured to detect this identifier in messages received from the switch fabric and pass the interrupt to the monitoring software task running on the ARM core. The trigger can also be used to drop the current frame within the *tblock* by setting a bit in the *tblock* configuration (which also configures the pattern detector). We measure the latency incurred from receiving the malicious frame at the receiving port of VEGa to the software updating the configuration entry in the *lookup memory*, using a dedicated hardware timer that was added for this experiment. The latencies of these steps are shown in Table 2.

The measurements are performed with the ARM cores running the standalone OS, a minimalist operating system from Xilinx and in the absence of any cross traffic. We have assumed that the *reconfigure* message is mapped as a high-priority message, to ensure that the switching latency is minimised. It was observed that the average latency from the time the packet was recieved at the Ethernet port to the interrupt being raised by the *tblock* was 5.101 $\mu$s. The interrupt was serviced within an average latency of 2.96 $\mu$s, while a single configuration entry could be updated in 300 ns, resulting in a total delay of 8.361 $\mu$s to update the *lookup* memory content. However, this would only block a single Ethernet address from transmitting a message to a single location on the safety-critical network. It is also possible to block the entire port from forwarding any information to the safety-critical FlexRay port; for this, the entire *lookup* memory must be updated. The software code was able to generate the new routing table and update it in the *lookup memory* in 21.25 $\mu$s using the DMA write mechanism, thus completely disabling the port in 29.311 $\mu$s from the reception of the malicious frame. Finally, it is also possible to completely disable the port by setting the *search_enable* bit in the *lookup* memory to '0'.

This experiment demonstrates how VEGa can react to dynamic network conditions by actively managing the central switching module in real-time (with a bounded latency of under 30 $\mu$s). Other scenarios that can benefit from this capability include dynamic routing of messages to support active fault-tolerance, run-time priority management for task migration in case of permanent faults in critical systems, and on-demand services such as emergency assistance.

## 5 Conclusion

Ethernet is widely expected to be the network backbone for next generation vehicular architectures, while existing networks like CAN, LIN and FlexRay continue to support their respective classes of applications. While gateways based on MCUs allow reliable message exchange for traditional networks like CAN or FlexRay that operate at 1–10 Mbps, they cannot cope with the 100 Mbits or higher data rates of Ethernet networks, especially when operating close to full utilisation. In this paper, we presented VEGa, a configurable low-latency gateway architecture on hybrid FPGAs like the Xilinx Zynq. VEGa allows accelerated message exchange with minimal latency with selectable priority levels for handling critical control messages. Also, the tight coupling with a capable processing system allows traffic monitoring and control of individual ports in real-time. We also defined a message translation mechanism that allows high-priority CAN and FlexRay networks to be directly interfaced to gateway ports for better determinism and lower latency message exchange. Experiments show that VEGa is able to achieve sustained switching performance at high network utilisation and provides a reliable path for critical messages even in the presence of cross traffic, without using dedicated switching paths. The modular architecture maps well to the requirements of the automotive domain, allowing model variations with a manageable, scalable architecture. The run-time adaptability of the switch also addresses emerging concerns like network security in connected vehicles. We are also exploring the integration of security primitives (like light-weight encryption and authentication frameworks) within VEGa to provide a secure communication infrastructure for next generation vehicular systems.

## Acknowledgments

This work was supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme. With the support of the Technische Universität München - Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement no. 291763.

## References

[1] A. Puhm, P. Roessler, M. Wimmer, R. Swierczek, and P. Balog, "Development of a flexible gateway platform for automotive networks," in *Proc. Int. Conf. Emerging Technol. Factory Autom.*, 2008, pp. 456–459.

[2] BroadCom Inc, *Product Brief : BCM89810 Single-port BroadR-Reach automotive Ethernet transceiver*, October 2011, https://docs.broadcom.com/docs/12358277

[3] *MPC5646C: Microcontroller Data Sheet*, Freescale Semiconductor, Inc., 2014, http://www.nxp.com/assets/documents/data/en/data-sheets/MPC5646C.pdf

[4] *SG187: Automotive Selector Guide*, Freescale Semiconductor, Inc., 2015, http://www.nxp.com/assets/documents/data/en/product-selector-guide/SG187.pdf

[5] O. Sander, M. Hubner, J. Becker, and M. Traub, "Reducing latency times by accelerated routing mechanisms for an FPGA gateway in the automotive domain," in *Proc. Int. Conf. Field Program. Technol.*, 2008, pp. 97–104.

[6] J. W. Lockwood, et al., "NetFPGA–an open platform for gigabit-rate network switching and routing," in *Proc. IEEE Int. Conf. Microelectron. Syst. Edu.*, 2007, pp. 160–161.
[7] G. Carvajal, M. Figueroa, R. Trausmuth, and S. Fischmeister, "Atacama: An open FPGA-based platform for mixed-criticality communication in multi-segmented Ethernet networks," in *Proc. Int. Symp. Field-Program. Custom Comput. Mach.*, 2013, pp. 121–128.
[8] K. Vipin, S. Shreejith, S. A. Fahmy, and A. Easwaran, "Mapping time-critical safety-critical cyber physical systems to hybrid FPGAs," in *Proc. Int. Conf. Cyber Phys. Syst. Netw. Appl.*, 2014, pp. 31–36.
[9] S. Shreejith and S. A. Fahmy, "Extensible FlexRay communication controller for FPGA-based automotive systems," *IEEE Trans. Veh. Technol.*, vol. 64, no. 2, pp. 453–465, Feb. 2015.
[10] P. Mundhenk, S. Steinhorst, M. Lukasiewycz, S. A. Fahmy, and S. Chakraborty, "Lightweight authentication for secure automotive networks," in *Proc. Des. Autom. Test Eur. Conf.*, 2015, pp. 285–288.
[11] S. Shreejith and S. A. Fahmy, "Security aware network controllers for next generation automotive embedded systems," in *Proc. Des. Autom. Conf.*, 2015, pp. 39:1–39:6.
[12] G. Leen, D. Heffernan, and A. Dunne, "Digital networks in the automotive vehicle," *Comput. Control Eng. J.*, vol. 10, no. 6, pp. 257–266, 1999.
[13] *FlexRay Communications System, Protocol Specification Version 2.1 Revision A*, FlexRay Consortium Std., December 2005.
[14] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proc. IEEE*, vol. 93, no. 6, pp. 1204–1223, Jun. 2005.
[15] C. A. Lupini, "In-Vehicle Networking Technology for 2010 and Beyond," *SAE2010 - World Conference & Exhibition*, 2010.
[16] S.-H. Kim, et al., "A gateway system for an automotive system: LIN, CAN, and FlexRay," in *Proc. Int. Conf. Ind. Informat.*, 2008, pp. 967–972.
[17] E. Schmidt, M. Alkan, K. Schmidt, E. Yü Andrü Andklü, and U. Karakaya, "Performance evaluation of FlexRay/CAN networks interconnected by a gateway," in *Proc. Int. Symp. Ind. Embedded Syst.*, 2010, pp. 209–212.
[18] J.-S. Yang, S. Lee, K. C. Lee, and M. H. Kim, "Design of FlexRay-CAN gateway using node mapping method for in-vehicle networking systems," in *Proc. Int. Conf. Control Autom. Syst.*, 2011, pp. 146–148.
[19] S.-H. Seo, J.-H. Kim, S.-H. Hwang, K. H. Kwon, and J. W. Jeon, "A reliable gateway for in-vehicle networks based on LIN, CAN, and FlexRay," *ACM Trans. Embedded Comput. Syst.*, vol. 11, no. 1, p. 7, 2012.
[20] J.-H. Kim, S. Seo, T. Nguyen, B. Cheon, Y. Lee, and J. Jeon, "Gateway Framework for In-Vehicle Networks based on CAN, FlexRay and Ethernet," *IEEE Trans. Veh. Technol.*, vol. 64, no. 10, Oct. 2015.
[21] H. Zinner, J. Noebauer, T. Gallner, J. Seitz, and T. Waas, "Application and realization of gateways between conventional automotive and IP/Ethernet-based networks," in *Proc. Des. Autom. Conf.*, 2011, pp. 1–6.
[22] K.-R. Müller, T. Steinbach, F. Korf, and T. C. Schmidt, "A real-time Ethernet prototype platform for automotive applications," in *Proc. Int. Conf. Consum. Electron.*, 2011, pp. 221–225.
[23] H.-T. Lim, B. Krebs, L. Völker, and P. Zahrer, "Performance evaluation of the inter-domain communication in a switched Ethernet based in-car network," in *Proc. IEEE Conf. Local Comput. Netw.*, 2011, pp. 101–108.
[24] N. Alt, C. Claus, and W. Stechele, "Hardware/software architecture of an algorithm for vision-based real-time vehicle detection in dark environments," in *Proc. Des. Autom. Test Eur. Conf.*, 2008, pp. 176–181.
[25] C. Claus, R. Ahmed, F. Altenried, and W. Stechele, "Towards rapid dynamic partial reconfiguration in video-based driver assistance systems," in *Proc. Int. Symp. Appl. Reconfig. Comput.*, 2010, pp. 55–67.
[26] N. Chujo, "Fail-safe ECU system using dynamic reconfiguration of FPGA," *R&D Rev. Toyota CRDL*, vol. 37, pp. 54–60, 2002.
[27] C. Claus, J. Zeppenfeld, F. Muller, and W. Stechele, "Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system," in *Proc. Des. Autom. Test Eur. Conf.*, 2007, pp. 498–503.
[28] S. Shreejith, S. Fahmy, and M. Lukasiewycz, "Reconfigurable computing in next-generation automotive networks," *IEEE Embedded Syst. Lett.*, vol. 5, no. 1, pp. 12–15, Mar. 2013.
[29] H.-M. Pham, S. Pillement, and D. Demigny, "Reconfigurable ECU communications in AUTOSAR Environment," in *Proc. Int. Conf. Intell. Transp. Syst. Telecommun.*, 2009, pp. 581–585.
[30] S. Shreejith, K. Vipin, S. A. Fahmy, and M. Lukasiewycz, "An approach for redundancy in FlexRay networks using FPGA partial reconfiguration," in *Proc. Des. Autom. Test Eur. Conf.*, 2013, pp. 721–724.
[31] F. Fons and M. Fons, "FPGA-based Automotive ECU Design Addresses AUTOSAR and ISO 26262 Standards," *Xcell J.*, vol. 78, pp. 20–31, 2012.
[32] S. Shaheen, D. Heffernan, and G. Leen, "A gateway for time-triggered control networks," *Microprocessors Microsyst.*, vol. 31, no. 1, pp. 38–50, 2007.
[33] D. Kohler, "A practical implementation of an IEEE1588 supporting Ethernet switch," in *Proc. Int. Symp. Precision Clock Synchronization Meas. Control Commun.*, 2007, pp. 134–137.
[34] G. Carvajal, C. W. Wu, and S. Fischmeister, "Evaluation of communication architectures for switched real-time ethernet," *IEEE Trans. Comput.*, vol. 63, no. 1, pp. 218–229, Jan. 2014.
[35] D. V. Schuehler and J. W. Lockwood, "A modular system for FPGA-based TCP flow processing in high-speed networks," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2004, pp. 301–310.
[36] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. IEEE Symp. Field-Program. Custom Comput. Machi.*, 2002, pp. 111–120.
[37] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2003, pp. 880–889.
[38] *UG585: Zynq-7000 All Programmable SoC Technical Reference Manual*, Xilinx Inc., Mar. 2013.
[39] P. Mundhenk, F. Sagstetter, S. Steinhorst, M. Lukasiewycz, and S. Chakraborty, "Policy-based message scheduling using FlexRay," in *Proc. Int. Conf. Hardw./Softw. Codes. Syst. Synthesis*, 2014, pp. 19:1–19:10.
[40] M. Lukasiewycz, M. Glaß, J. Teich, and P. Milbredt, "FlexRay schedule optimization of the static segment," in *Proc. IEEE/ACM Int. Conf. Hardw./Softw. Codes. Syst. Synthesis*, 2009, pp. 363–372.
[41] K. Koscher, et al., "Experimental security analysis of a modern automobile," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 447–462.
[42] S. Checkoway, et al., "Comprehensive experimental analyses of automotive attack surfaces," in *Proc. USENIX Security Symp.*, 2011, pp. 6–6.

**Shanker Shreejith** (S'13,M'17) received the BTech degree in electronics and communication engineering from University of Kerala, India and the PhD degree in computer science and engineering from Nanyang Technological University, Singapore, in 2006 and 2016, respectively. From 2006 to 2008, he worked as an FPGA design and development engineer. From 2008 to 2011, he worked as a scientist at Digital Systems Group, Vikram Sarbhai Space Centre, Trivandrum, under the Indian Space Research Organisation (ISRO). From 2015 to 2016, he was a research fellow at the School of Computer Science and Engineering, Nanyang Technological University, Singapore. Since 2017, he is a teaching fellow at the School of Engineering, University of Warwick, United Kingdom. He is a member of the IEEE.

**Philipp Mundhenk** (S'12) received the Bachelor of engineering degree in computer engineering from University of Cooperative Education Baden-Württemberg Lörrach and the Master of Science degree in electrical and computer engineering from the Technische Universität München, in 2010 and 2012, respectively. From 2007 to 2010 he worked as development engineer for industrial and building automation systems software and networks at the embeX GmbH in Freiburg, Germany. From 2012 to 2016, he was a Research Associate in the research project for embedded systems at TUM CREATE in Singapore. Since 2016, he is a developer for data transfer technologies and security at the Audi Electronics Venture GmbH, a subsidiary of the Audi AG in Ingolstadt, Germany. He is a student member of the IEEE.

**Andreas Ettner** received the Dipl-Ing degree in electrical engineering and information technology from Technische Universität München, Germany, in 2014. In 2013 and 2014, he was a research assistant with TUM CREATE in Singapore. Since 2015, he is working toward the PhD degree in electrical engineering and information technology at Robert Bosch GmbH in Renningen, Germany. His research focuses on model-based design and optimization of automotive embedded systems.

**Suhaib A Fahmy** (M'01, SM'13) received the MEng degree in information systems engineering and the PhD degree in electrical and electronic engineering from Imperial College London, UK, in 2003 and 2007, respectively. From 2007 to 2009, he was a research fellow with the University of Dublin, Trinity College, and a Visiting Research Engineer with Xilinx Research Labs, Dublin. From 2009 to 2015, he was an assistant professor with the School of Computer Engineering, Nanyang Technological University, Singapore. Since 2015, he has been an associate professor at the School of Engineering, University of Warwick, UK. His research interests include reconfigurable computing, high-level system design, and computational acceleration of complex algorithms. He was a recipient of the best paper award at the IEEE Conference on Field Programmable Technology in 2012, the IBM faculty award in 2013, and is a senior member of the ACM. He is a senior member of the IEEE.

**Sebastian Steinhorst** (M'11) received the PhD degree in computer science from Goethe-University of Frankfurt am Main, Germany, in 2011. He is an assistant professor in the Department of Electrical and Computer Engineering, Technische Universität München (TUM) in Germany and a Fellow of the TUM Institute for Advanced Study. Before joining TUM, in November 2016, he was an assistant professor at Aarhus University in Denmark between May and September 2016. From 2011 to 2016, he was with the Embedded Systems Group at TUM CREATE Ltd. in Singapore which he was leading as principal investigator since 2015. He is a member of the IEEE.

**Martin Lukasiewycz** (M'11) received the PhD degree in computer science from the University of Erlangen-Nuremberg, Germany, in 2010. From 2011 until 2015, he was a principal investigator of the Embedded Systems Group at TUM CREATE Ltd. in Singapore. From 2014 until 2015, he was also an adjunct assistant professor with Nanyang Technological University, Singapore. He is a member of the IEEE.

**Samarjit Chakraborty** (SM'15) received the PhD degree in electrical and computer engineering from ETH Zurich, Switzerland, in 2003. He is a professor of electrical engineering with Technical University of Munich (TUM), Germany, where he holds the chair of real-time computer systems. Prior to joining TUM, he was an assistant professor of Computer Science with the National University of Singapore, from 2003 to 2008. He also leads a research program on embedded systems design for electric vehicles at the TUM Campus for Research Excellence and Technological Enterprise Centre for Electromobility in Singapore, where he also serves as a Scientific Advisor. His research interests include system-level design of embedded and cyber-physical systems, with applications in automotive, healthcare, and sensor network-based information processing. He is a senior member of the IEEE.