Extensible FlexRay Communication Controller for FPGA-Based Automotive Systems

Shanker Shreejith, *Student Member, IEEE*, and Suhaib A. Fahmy, *Senior Member, IEEE*

***Abstract*—Modern vehicles incorporate an increasing number of distributed compute nodes, resulting in the need for faster and more reliable in-vehicle networks. Time-triggered protocols such as FlexRay have been gaining ground as the standard for high-speed reliable communications in the automotive industry, marking a shift away from the event-triggered medium access used in controller area networks (CANs). These new standards enable the higher levels of determinism and reliability demanded from next-generation safety-critical applications. Advanced applications can benefit from tight coupling of the embedded computing units with the communication interface, thereby providing functionality beyond the FlexRay standard. Such an approach is highly suited to implementation on reconfigurable architectures. This paper describes a field-programmable gate array (FPGA)-based communication controller (CC) that features configurable extensions to provide functionality that is unavailable with standard implementations or off-the-shelf devices. It is implemented and verified on a Xilinx Spartan 6 FPGA, integrated with both a logic-based hardware ECU and a fully fledged processor-based electronic control unit (ECU). Results show that the platform-centric implementation generates a highly efficient core in terms of power, performance, and resource utilization. We demonstrate that the flexible extensions help enable advanced applications that integrate features such as fault tolerance, timeliness, and security, with practical case studies. This tight integration between the controller, computational functions, and flexible extensions on the controller enables enhancements that open the door for exciting applications in future vehicles.**

***Index Terms*—Automotive systems, field-programmable gate arrays, networks.**

I. Introduction

MODERN high-end vehicles incorporate 100 or more embedded computing units that implement advanced capabilities such as automated parking, pedestrian detection with automatic braking, and other safety or comfort features. These algorithms perform complex processing on data gathered from a network of sensors, to produce control sequences for distributed actuators. The communication bandwidth and quality of service required for such advanced electronic control units (ECUs) exceed the capabilities of the event-triggered controller area network (CAN) protocol, which has been pervasive in automotive systems until now. Moreover, next-generation in-vehicle systems, specifically in electric vehicles that have a high level of automation, demand higher determinism, leading to the widespread adoption of time-triggered communication schemes and protocols such as FlexRay and time-triggered Ethernet [1]. FlexRay is gaining ground as a *de facto* communication standard for safety-critical functions such as drive-by-wire, cruise control, and adaptive braking systems, while also facilitating communication for noncritical ECUs.

Although time-triggered networks such as FlexRay provide higher determinism and communication bandwidth, increasing proliferation of embedded computing units increases the associated communication overheads and power consumption, which can degrade overall system performance. Typically, each ECU has a discrete communication controller (CC) to manage its access to the network. We show that, by closely coupling the controller with the ECU and extending the predefined communication framework, advanced and intelligent embedded compute units with enhanced capabilities such as fallback and fault tolerance can be designed. This scheme enhances the overall quality and performance of the system. However, such evolutions and extensions of the protocol cannot be implemented using off-the-shelf controllers or platform-agnostic solutions, and they require a modular flexible implementation, which is ideally implemented in reconfigurable logic. Moreover, reconfigurable technology enables us to merge the controller and multiple applications on the same device, while preserving the necessary isolation between them, and partial reconfigurability can be exploited to reduce power consumption further [2].

In this paper, we present an architecture-optimized FlexRay CC, which integrates configurable extensions that augment the CC's capabilities beyond those defined by the FlexRay standard. The controller provides enhancements to the data path, such as programmable width timestamping, data filtering, header insertion, and processing functions, which are abstracted away from the host function. Our flexible architecture can be used to design advanced ECUs on reconfigurable hardware, which consume less power and offer increased consolidation, while providing enhanced capabilities that are impossible to implement using standard controllers or IP cores. We also quantify the potential of the proposed controller using case studies based on existing and evolving automotive applications that are safety critical and data intensive. Our experiments show that advanced features, such as high-speed mode switching for fault-tolerant ECUs, low-latency data handling for high-performance gateways, timeliness, and security for messages can be efficiently achieved by integrating such extensions within the controller data path, rather than offloading them to the processing logic.

Manuscript received September 5, 2013; revised January 30, 2014; accepted April 22, 2014. Date of publication May 16, 2014; date of current version February 9, 2015. The review of this paper was coordinated by Dr. S. Anwar.

S. Shreejith is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798, and also with TUM CREATE, Singapore 138602 (e-mail: shreejit1@ntu.edu.sg).

S. A. Fahmy is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798 (e-mail: sfahmy@ntu.edu.sg).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVT.2014.2324532

The remainder of this paper is organized as follows. In Section II, we give a brief introduction to the FlexRay specification and protocol enhancements described in the literature and the related work in this area. Section III details the controller architecture. Implementation results and comparison with other implementations are provided in Section IV. In Section V, we present case studies using the customizable extensions and show the benefits of implementing advanced features this way, with a discussion of the approach in Section VI. Finally, we conclude this paper and outline future work in Section VII.

II. Related Work

The move toward time-triggered network standards in automotive systems has been driven by the more advanced requirements imposed by advanced mission-critical and comfort features in future vehicles. Widespread event-triggered networks such as CANs fail to address the requirements of such applications.

Time-triggered CAN (TT-CAN) is an extension of the CAN protocol that enables time-triggered operation by enforcing a slot-based structure, while retaining backward compatibility with the standard CAN. However, TT-CAN suffers from dependability issues and limited bandwidth; thus, it did not gain widespread adoption. Some research sought to overcome these limitations through hardware extensions on the network controller [3].

In recent years, FlexRay has emerged as the standard for time-triggered communication in the automotive domain. However, most recently, new hardware developments have seen time-triggered Ethernet emerge as a possible replacement for FlexRay, although standard communication protocols are still under development. The enhancements we present in this paper can be similarly applied to other time-triggered standards, although we use FlexRay to demonstrate the concepts within a realistic certifiable environment.

A. The FlexRay Protocol

The FlexRay protocol is developed and standardized by the FlexRay consortium and has since been adopted by various automotive companies in production vehicles [4]. These vehicles are complaint with the FlexRay AUTOSAR Interface Specification Standard [5], which is the industry standard for the software specification of FlexRay nodes, by which any controller implementation must comply.

The fundamental element of the media access scheme in the FlexRay protocol is the communication cycle, which is repeated over time, as shown in Fig. 1. Each cycle is comprised of four segments [6].

- The *static segment* uses a static slot-based access mechanism and is used to send critical data in a deterministic manner. Any ECU can send a frame of data in the one (or more) slot(s) assigned to it. The slot width is fixed across all nodes on the network.
- The *dynamic segment* uses a dynamic slot-based access scheme enabling communication of event-triggered data of arbitrary length. The slot width is dynamic, depending on the amount of data that needs to be transmitted, and access to the medium is controlled by priorities assigned to the ECUs.
- The *symbol window* is used to transmit special symbols such as the "wake-up" pattern used to wake up *sleeping* nodes to initiate communication.
- *Network idle time* is the idle period used by nodes to make clock adjustments and align and correct the global view of time to maintain synchronization.

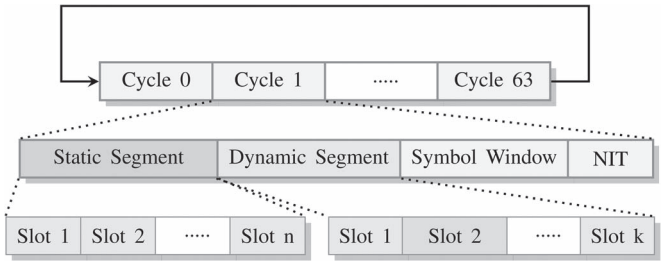

Fig. 1. FlexRay communication cycle.

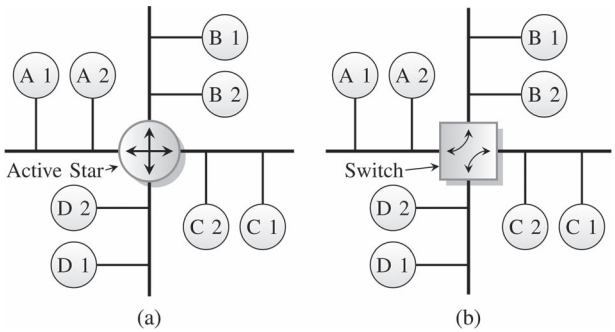

Fig. 2. (a) Standard and (b) switched FlexRay network topologies.

A typical FlexRay-based ECU integrates a discrete (or embedded) CC and the computational function, which is usually implemented as a software algorithm on a processor to provide flexibility and upgradability. The ECU can communicate over the bus, through the CC, by transmitting framed data in the slot(s) assigned to it in the static or dynamic segments.

Multiple nodes may share the same slot in different cycles, as in the case of odd/even cycle multiplexing where one set of nodes is assigned slots in all odd cycles, whereas another set of nodes (which may include some from the first set) is assigned slots in all even cycles. This scheme of cycle-level *slot multiplexing* can lead to higher overall bandwidth utilization. Fig. 2(a) shows a typical network setup, where node $B2$ may send data to node $A1$ in slot 1 of cycle 1, whereas node $A1$ may reply in slot 1 of cycle 2. The *active star* is an active repeater that passes information from one branch to all other branches.

The switched FlexRay network is a novel concept that can extend bandwidth without compromising reliability and determinism [7]. The switch architecture allows exploitation of branch parallelism, whereby the switch will repeat frames only on branches that contain the intended recipient [8]. This allows the same slots to be used simultaneously by multiple nodes in the same cycle, and the intelligent FlexRay switch schedules the branch to which information has to be relayed [9]. Thus, as shown in Fig. 2(b), while node $B2$ may be sending data

to node $A1$ in slot 1 of cycle 1, node $D1$ might be sending data to node $C2$ in the same slot, and the switch, knowing the schedule, connects the corresponding nodes through the switch fabric. By utilizing slot multiplexing and branch parallelism, each slot within a cycle may have different destinations and, thus, different switch configurations.

Research on FlexRay networks has been approached from diverse directions in the literature. In [10], Forest highlights challenges such as physical-layer design, cycle and schedule design, and selection of termination, sync, and startup nodes, which were all simpler design considerations for FlexRay's predecessor, i.e., CAN.

B. Scheduling

Much work has been done on scheduling communication on the shared bus. Optimization of the static and dynamic segments of the FlexRay protocol has been widely addressed in [11] to [15], among others. In [16], a detailed survey of scheduling algorithms is presented, and a comparison between optimization strategies such simulated annealing, genetic, hybrid genetic, and probabilistic approaches applied by various algorithms is provided. Given a set of communication requirements, all algorithms try to optimize the number of communication slots and cycles that are required to schedule the different messages, satisfying all requirements. The optimization in most cases is to find the minimum number of communication slots that can solve the problem, hence consuming minimum bandwidth. Alternatively, the problem can be formulated to maximize the number of unused slots, which provides flexibility for future expansion.

C. Network-Level Optimizations

In [17], an approach to improve the energy efficiency of a FlexRay controller by allowing it to be controlled by an intelligent CC (ICC) is discussed. The ICC, which takes over bus operations from the ECU when the latter goes to sleep, prevents the ECU from being woken by erroneous transmissions, allowing the node to achieve higher power efficiency. The proposed architecture and its validation are also discussed in this paper, using a proprietary implementation of the FlexRay CC that is not available to the research community. Similarly, in [18], the architecture of an FPGA implementation of the FlexRay controller with add-on features to aid functional verification is described. The features are primarily aimed at a verification framework and hence do not point in the direction of optimizations or enhancements for improving node/network functionality beyond standard implementations.

D. Controller Implementation

The work in [19] is the only one to discuss the implementation of a FlexRay CC on reconfigurable logic. It discusses the protocol operations control module, which controls the actions of the core modules of the CC. However, no specific details about hardware architecture are presented, and the implementation is designed purely to implement the existing specification, with no new features. In [20] and [21], implementation of the FlexRay CC is described using the specification and description language as the platform and later translation to hardware using Verilog. These studies approach the protocol from a high level of abstraction and, hence, do not discuss hardware design details or architectural optimization. A comprehensive outline of the FlexRay Bus Guardian specification and approaches to implement it on FPGAs have also been discussed in [22] and [23].

Bosch and Freescale both offer implementations of the FlexRay controller that can be mapped to a wide range of platforms [24], [25]. These are largely platform independent, suitable for implementing on application-specific integrated circuits or FPGAs. However, they are not optimal for implementation on reconfigurable hardware since they do not fully utilize the heterogeneous resources available in the fabric. For instance, the E-Ray IP core from Bosch, which is a dual channel controller, does not directly instantiate FPGA primitives such as digital signal processing (DSP) blocks or Block RAM but uses general-purpose logic to implement these functions.

By efficiently utilizing these hardware primitives, we can build custom controllers that are more efficient for architecture-specific implementations, while leaving aside logic for implementing functional components of the ECU. This approach results in limited portability between platforms but superior utilization and power efficiency for FPGA-based ECU implementations. Portability is also becoming less of an issue as FPGA manufacturers standardize hardware blocks across all their device families in a given generation. For example, the DSP48E1 primitive is available on all 7-series FPGAs from Xilinx, as well as the Zynq ARM-FPGA platform.

We aim, through this paper, to enable a number of investigations in the space of FlexRay on reconfigurable hardware. We focus on providing a flexible CC, which features rich extensions for enhanced applications and architecture optimizations for low device utilization, providing considerable savings in terms of area and power. The objective is to show how an FPGA-centric implementation can result in interfaces that provide advanced capabilities and power efficiency for FPGA-based in-vehicle systems.

III. Architecture Design

A node on the FlexRay network consists of a CC, an application running on a host ECU, and multiple bus drivers to independently support two communication channels. The host ECU is the computational implementation of an algorithm like adaptive cruise control or engine management, and it may communicate with other ECUs or sensor nodes over the network. The CC ensures conformance with the FlexRay specification when transmitting or receiving data on the communication channel. The Bus Driver (BD) handles the bit stream at the physical level and provides the physical level interface to the communication channel. The host ECU monitors the status of the CC and the BD independently and configures them appropriately at startup or during runtime.

A. Communication Controller

The FlexRay CC switches between different operating states, based on network conditions and/or host commands, ensuring

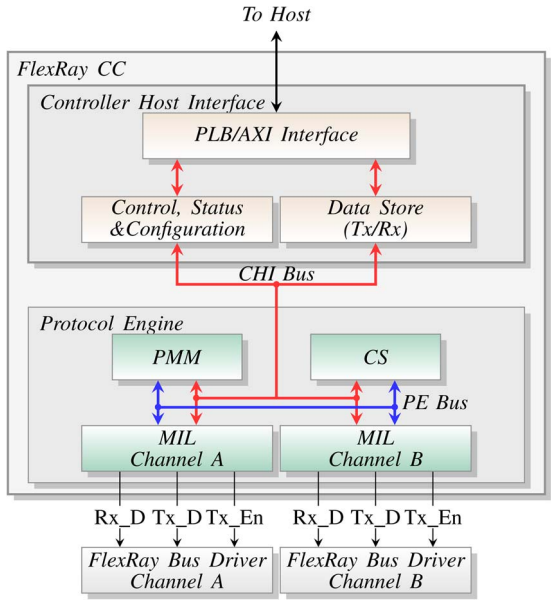

Fig. 3. Architecture of custom Flexray CC.

conditions defined by the FlexRay protocol are met at all times. The CC architecture, as shown in Fig. 3, comprises the Protocol Engine (PE), which implements the protocol behavior, and the Controller Host Interface (CHI), which interfaces to the host ECU.

The CHI module communicates with the host and handles commands and configuration parameters for the FlexRay node. These parameters are defined for the particular cluster that the node is operating on, and they are initialized during the node's configuration phase. The CHI feeds the current state and operational status to the host for corrective action if necessary. There are transmit and receive buffers and status registers for the data path to isolate control and data flow. The CHI may also incorporate clock-domain crossing circuitry to enable the different interfaces to work in distinct clock domains.

The clock synchronization (CS) and medium interface layer (MIL) submodules of the PE implement specific functions of the protocol, which are controlled and coordinated by the protocol management module (PMM). These submodules support multiple modes of operation and can alter their current operating mode in response to changes in any of the parameters, error conditions, or host commands. The PMM ensures mode changes are done in a way that complies with the FlexRay specifications. The MIL handles the transmission and reception of data over the shared bus. It encodes and decodes data, controls medium access, and processes decoded data to ensure adherence to protocol specifications. The CS module generates the local node clock, synchronized to the global view of time. It measures deviation in the node clock on a per-cycle basis so that it stays synchronized with other nodes in the cluster.

Timing in a FlexRay node is defined in *macroticks* and *microticks*. Microticks measure the granularity of the node's local internal time and are derived from the internal clock of a node. A macrotick is composed of an integer number of microticks. The duration of each local macrotick should be equal within all nodes in the cluster. The FlexRay protocol uses a distributed clock correction mechanism, whereby each node individually adjusts its view of time by observing the timing information transmitted by other nodes. The adjustment value is computed using a fault-tolerant midpoint algorithm. A combination of rate (frequency) and offset (phase) correction mechanisms is used to synchronize the global time view of different nodes. These corrections must be applied in the same way at all nodes and must fulfil the following conditions.

1) Rate correction is continuously applied over the entire cycle.
2) Offset correction is applied only during the NIT in an odd cycle and must finish before the start of the next communication cycle.
3) Rate correction is computed once per double cycle, following the static segment in an odd cycle. The calculation is based on values measured in an even–odd double cycle.
4) Calculation of offset correction takes place every cycle but is applied only at the end of an odd cycle.

Rate correction indicates the number of microticks that need to be added to the configured number of microticks per cycle and may be negative, indicating that the cycles should be shorter. Offset correction indicates the number of microticks that need to be added to the offset segment of the network idle time and may also be negative.

The FlexRay bus supports two independent channels for data transmission and reception. The transmission rate can be set at 2.5, 5, or 10 Mb/s. The protocol also defines multiple bus access mechanisms, in the form of *static slots* for synchronous time-triggered communication and *dynamic slots* for burst-mode event-triggered (priority-based) data transfer. Special symbols can be transmitted within the *symbol window*, such as wake-up during operation (WUDOP) and collision avoidance symbol (CAS). During the *network interval time*, all nodes synchronize their clock view with the global clock view so that they stay synchronous. Each transmitted bit is represented using eight bit times to ensure protection from interference. At the receiving end, these are sampled, and the majority voted to generate a voted bit. Transmission and reception must be confined to slot boundaries, and transmission (or reception) across slot boundaries is marked as a violation. The node should transmit only on slots that are assigned to it (either in the static or dynamic segments). Each node is assigned a *key slot*, which it uses to transmit startup or synchronization frames (along-with data).

B. Implementation and Optimizations of Custom CC

The state of the PMM, at any instant, reflects the current operating mode of the CC. The PMM triggers synchronized changes in the CC and MIL submodules, and describes the different operating modes of the node, as shown in Fig. 4. These mode changes can be triggered by host commands or by internal and/or network conditions encountered by the node. Table I describes the different commands issued by the host and how the operation of the CC is modified in response. As shown, certain commands demand an immediate response from the controller, whereas others are to be applied at specific points

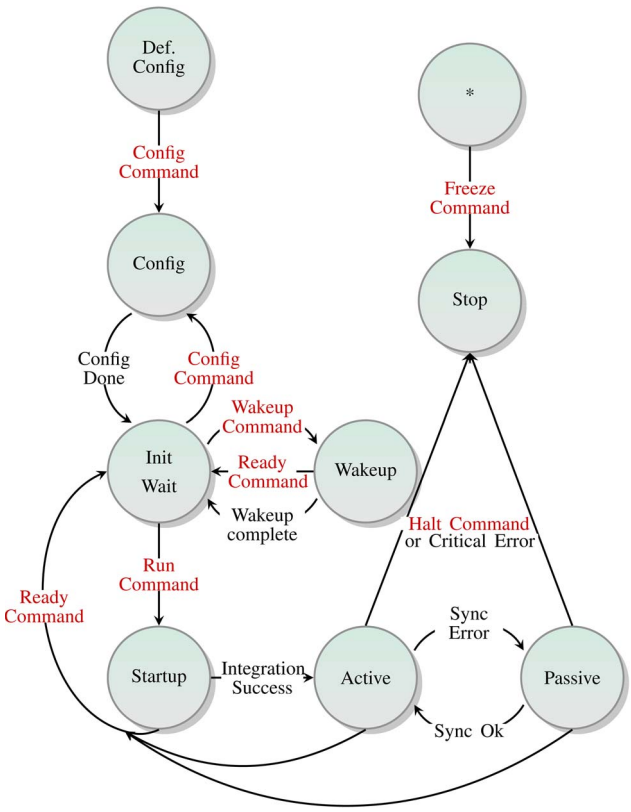

Fig. 4. FlexRay CC modes of operation.

TABLE I
COMMANDS FROM HOST THAT AFFECT CC OPERATING MODES

Host Command	Affected States	Final State	Processed at
ALL SLOTS	Active, Passive	no state change	End of cycle
ALLOW COLD-START	All states except Def. Config, Config, Stop	no state change	Immediate
CONFIG	Def. Config, Init Wait	Config	Immediate
CONFIG COMPLETE	Config	Init Wait	Immediate
DEFAULT CONFIG	Stop	Def. Config	Immediate
FREEZE	All States	Stop	Immediate
HALT	Active, Passive	Stop	End of cycle
READY	All states except Def. Config, Config, Init Wait, Stop	Init Wait	Immediate
RUN	Init Wait	Start-Up	Immediate
WAKEUP	Init Wait	Wake-Up	Immediate

within the communication cycle. This distinction makes the control flow more complex than the case of a straightforward finite state machine.

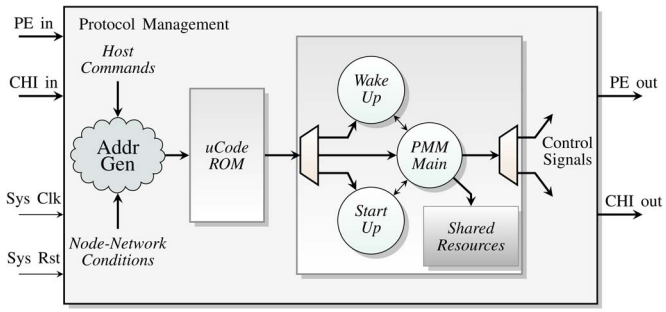

Fig. 5. Protocol management module architecture.

The FlexRay protocol allows a cluster and its associated nodes to switch to sleep mode to conserve power. When any node needs to start communication on the network, a wakeup sequence is triggered by the host by putting the CC into the wakeup state. In the wakeup state, the node tries to awaken a sleeping network by transmitting a wake-up-pattern (WUP) on one channel. Sleeping nodes decode this pattern and trigger a node wakeup. Nodes that have dual channel capability then trigger a wakeup on the other channel to complete a cluster-wide wakeup. The node cannot, however, verify the wakeup trigger at all connected nodes since the WUP has no mechanism to communicate the ID of the nodes that have responded. The nodes then follow the startup procedure to initialize communication on the cluster. The startup operation also caters for reintegration of a node onto an active network. To do so, the node must start its local clock so that it is synchronized with the network time.

Within the startup state, the clock synchronisation startup (CSS) logic in the CS module is initialized, which extracts timing information from a pair of synchronization frames received from the bus and starts the macrotick generator (MTG) in alignment with them. Over the next few cycles, it monitors the deviation of its clock from the actual arrival time of sync frames on the bus, and if these are within predefined limits, the process is signaled as successful. If, at any point, the observed deviation is beyond the configured range, the integration attempt is aborted, and the node restarts the process. Once it integrates, the node moves to the *active* state, with a clock that is synchronized with the network. After successfully joining the network, the PMM normally follows a cyclic behavior switching between active and passive states, in response to network-node conditions, causing synchronized changes in all modules.

In our design, the PMM also encapsulates Wakeup and Startup. Confining the Wakup and Startup operations within the respective PMM states result in a hierarchical structure, as in Fig. 5, with the combined state encoding stored in a microcoded ROM. Combining the two functions into the same module also allow us to share resources between the two operations, which are not required concurrently, using a simplified control flow. Since CS and MIL are also controlled by Wakeup and Startup logic for their respective operations, integrating them with the PMM results in centralized control for all operating conditions, simplifying interfaces to the submodules. The responses to different conditions or stimuli is now reduced to the process of generating appropriate addresses for the ROM, similar to

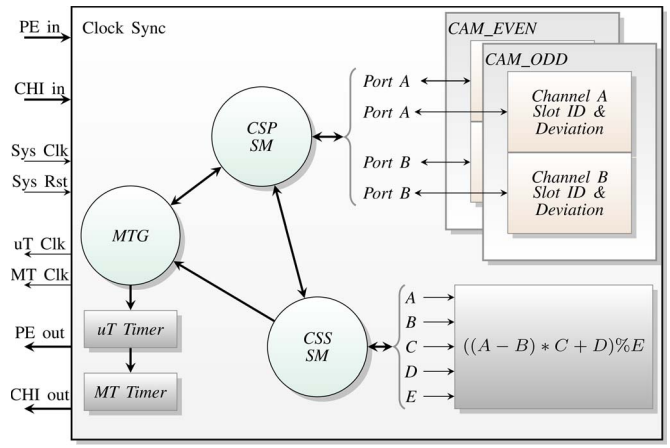

Fig. 6. Clock sync module architecture.

the program counter implementation on a standard processor. The ROM is efficiently implemented using distributed memory (lookup tables, LUTs) because of its small size.

Fig. 6 shows a simplified architecture of the CS module in our design. The CS module generates the clock, computes the deviations of the generated clock from the distributed timing information, and applies corrections. The CS module is comprised of two concurrent operations (or submodules): first, the MTG process that controls the cycle counter and the macrotick counters and applies the rate/frequency and offset/phase correction values; and second, the clock synchronization process (CSP) that performs the initialization at cycle start, the measurement and storage of deviation values during the cycle, and the computation of the offset and rate correction values. In addition, the CSS module is responsible for starting a synchronous clock when the CC tries to integrate into either an active network or initiate communication on an idle network. The CSP state machine controls and coordinates the operations of the CS module by interacting with the CSS and MTG submodules.

During startup, the CSS process monitors the arrival time of the even synchronization frames and generates the global reference time by computing the initial *macrotick* value as

$$Macrotick = (pMacroInitialOffset + gdStaticSlot \times (\mathrm{ID} - 1)) \bmod gMacroPerCycle$$

where *pMacroInitialOffset*, *gMacroPerCycle*, and *gdStaticSlot* are FlexRay parameters. The computation is implemented using cascaded DSP48A1 slices, whose inputs are multiplexed between channels A and B to handle startup requests from either channel. If a subsequent odd frame arrives within the predefined window, the integration attempt is flagged as successful by the CSS module, and the CSP commands the MTG state machine to start the *macrotick* clock (MTClk) using the computed *macrotick* value for this channel. The MTG then generates the *macrotick* clock from the *macrotick* clock (uTClk) using the configured parameter values.

Fig. 7 shows the clock deviation computation for each cycle once the CC successfully integrates onto the network. The measuring cycle refers to the duration of the static segment,

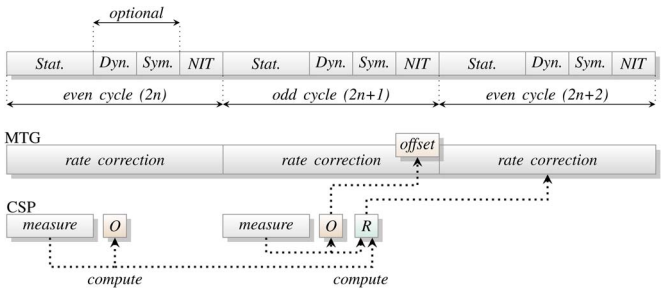

Fig. 7. Rate and offset computation by MTG and CSP.

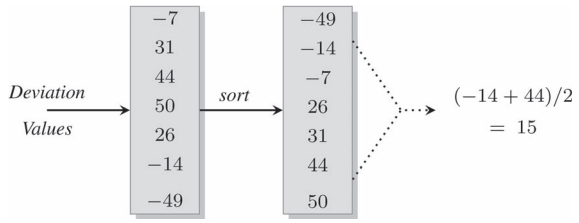

Fig. 8. Fault-tolerant midpoint illustration for seven deviation values.

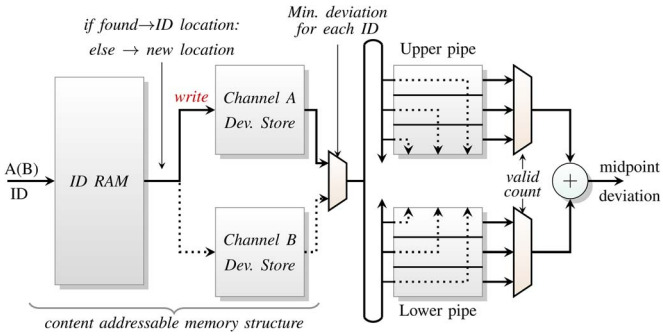

Fig. 9. CAM organization and fault-tolerant midpoint computation for offset correction.

where sync-nodes transmit synchronization frames that are used to compute rate and offset corrections. During each measurement cycle, the node measures the deviation of time of arrival registered at the node from the calculated time of arrival of the synchronization frame, which is stored in memory. At the end of measurement phase, the node computes the offset and rate correction factors from the stored values using a fault-tolerant midpoint algorithm. The operation is shown in Fig. 8, for a cycle that recorded seven deviation values. The real challenge here is that a network may be configured without dynamic and symbol window segments. Hence, the offset and rate computations have to be completed, consuming a minimum number of cycles to ensure that correction values are available to be applied at the network interval time segment.

Fig. 9 shows our solution to the midpoint computation mechanism, expanded from the slotID and deviation store in Fig. 6, for an even cycle. The fault-tolerant midpoint algorithm computes the rate and offset corrections that are to be applied to the macrotick clock. During normal operation, the CSP module handles the computation and storage of individual deviation values and the computation of midpoint correction values. As a frame is received, its ID is used to address the slotID RAM,

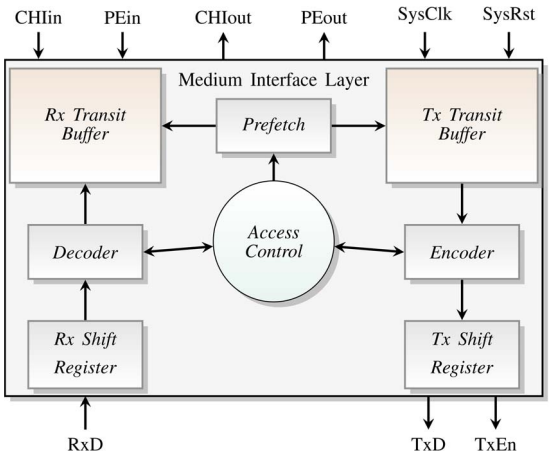

Fig. 10. MIL architecture.

the output of which is used as the address for the deviation store, mimicking a content-addressable memory. The deviation from the expected arrival time of sync frames to their actual arrival time is stored in the deviation store. The upper and lower pipes perform dynamic sorting (descending and ascending) as and when the deviation values are replayed from the store, at the end of the cycle. Dynamic sorting is implemented using a first-in–first-out (FIFO) structure and multiple comparators. Hierarchical comparison is performed from top to bottom (bottom to top) in the upper (lower) pipe. At any level, if the input value is greater (less) than the existing values at that level, the input value is pushed into the FIFO at that level.

For each ID, the multiplexer chooses the minimum deviation among the two channels, in the case of offset computation, and the difference between the corresponding channels in a pair of cycles, in the case of rate computation. The midpoint deviation is the average deviation over the corresponding stages in the upper and lower pipes, the stage chosen depending on the number of valid deviation values stored. The MTG uses the computed midpoint deviation values to make corrections to the node's view of time. Utilizing the tagging established by the content-addressable memory and the pipelined architecture, the midpoint computation can be efficiently implemented at the system clock rate to meet protocol requirements. A more conventional architecture would require a higher clock rate for this computation. Architectural optimization also enables us to utilize fewer resources while maximizing performance.

The MIL instantiates independent transmit and receive transit buffers to manage temporary storage of a frame, as shown in Fig. 10. The MIL ensures that medium access occurs only at slots assigned to the node. The access control state machine handles the bus access, depending on the current slot counter value and slot segment. The access control logic generates and maintains the slot counter and the slot segment, which are used by other modules in the CC. Within each slot, the logic generates control signals called action points, which mark points at which transmission can start (in static and dynamic slots) or end (in dynamic slots). These signals trigger the encoding logic to start transmission of frame in the transmit buffer, provided the current slot is allocated to this node.

The data to be transmitted is moved to the transmit buffer over a 32-bit data bus. If no data are available for transmission, the node transmits a null frame. The module also handles encoding and serial transmission of data (at the oversampled rate) to be transmitted in the current slot. Decoder functionality is also integrated into this module, which performs bit-strobing, majority-voting, byte-packing, and validation of received data at the end of the slot.

The transmit interface is implemented using shift registers with gated clocks. This allows us to provide multiple functions with the same set of registers: encode and transmit data bytes, control signals, and symbols. The shift register reads each byte from the transmit buffer, encodes it within the shift register, and pushes it to the transmit line at the transmit clock, along with the transmit control signals. At the receiving interface, sampling, bit strobing, and edge synchronization are implemented using a sequence of shift-register modules: one set samples the data and produces a majority voted bit every cycle, and the second set performs byte-packing of the data. This system offers the advantage of simpler control and higher throughput. The byte-packed data are written into the receive transit buffers. As and when protocol errors or violations are detected (such as reception crossing boundary points), appropriate flags are set locally, which are used to validate the data at the slot boundary. At the end of the current slot, the flags are checked to signal valid data, which can then be written into the receive data memory in the host interface.

The control modules are efficiently implemented as multiple state machines at different levels to ensure parallel and independent operation. The transmit buffers prefetch data from the transmit data store in the CHI at the start of each slot to minimize latency. Similarly, the data location for each received frame is precomputed to enable complete data to be written to the receive data store in the CHI before the start of the next frame, minimizing latency between the time of frame reception and it being passed to the Host. Also, the data available flag and interrupts (if enabled) are set, as soon as the first D-word is written into the receive buffer in the CHI. The data store and the associated control and status store in the data path mimic a content-addressable architecture in Block RAM to enable prefetching and addressing using the slot–cycle–channel complex, as required by the protocol.

Two such MILs are instantiated within the controller to support independent dual channel operation. These modules may transmit and receive data in the same slots, as configured by the host. To facilitate this, we have implemented a configurable scheduler, which can be configured for priority access (Channels A over B or *vice versa*) or first come first served mode. High word-length interconnects are used between the data store in the host interface and transit buffers within the MIL module to ensure low-latency prefetch and write back for both channels. Using such an architecture, the prefetching can be handled at system clock rates, without high latency. The physical layer can be configured to support multiple bit rates of 2.5 Mb/s, 5, or 10 Mb/s. The usage of shift-register-based encoder/decoder modules simplifies the logic requirements for handling multiple bit rates.

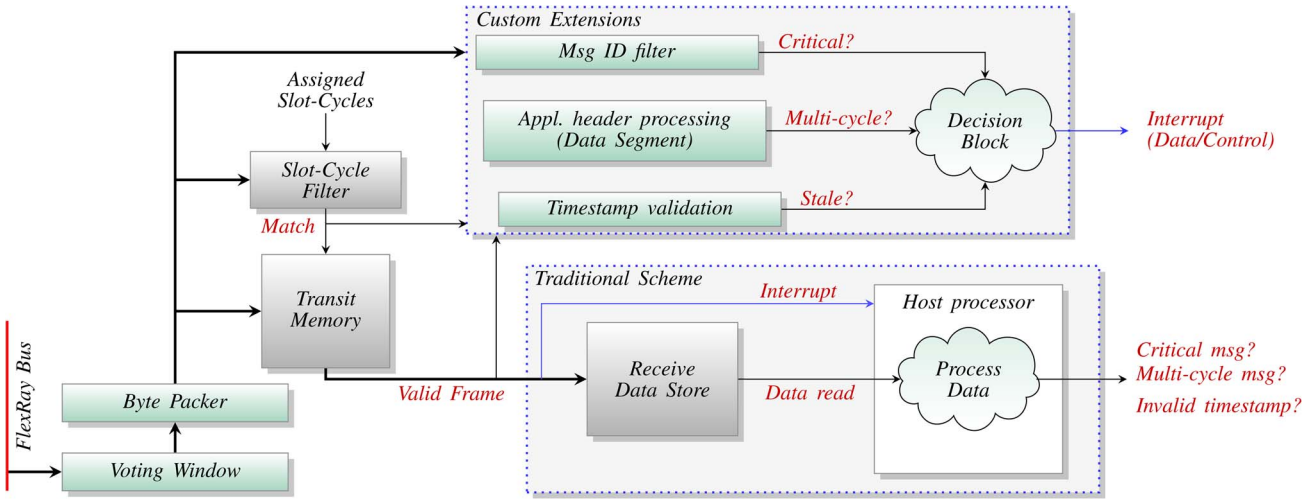

Fig. 11. Receive path extensions on custom CC versus traditional schemes.

The interface to the host processor is designed to be compatible with the Processor Local Bus (PLB) interface and AMBA Advanced eXtensible Interface 4 (AMBA AXI4) standards, two of the widely used high-performance low-latency peripheral interconnects for system-on-a-chip (SoC) designs. The host interface supports parameterized widths and a wide range of system and interrupt configurations to provide a rich interface to the host processor (or logic). The control path comprising the command, status, and configuration registers are isolated from the data path and implemented as a register stack. Data corresponding to each cycle, slot, and channel is addressed using an indirect addressing technique. The data pointer is stored at an address determined by the cycle–slot–channel complex. This allows us to use true dual-port Block RAM modules and simpler address generation as opposed to the complex FIFO-based schemes used by existing controllers. Another advantage is that the memory can be configured as a cyclic buffer resulting in an indefinite memory space, as opposed to the limited memory space available in a FIFO-based scheme. The memory space is dynamically allocated at the end of each slot that is configured as a receive slot, only if valid data has been received, thus optimizing memory usage.

Asynchronous FIFOs are instantiated between the host interface and the control/data stores, enabling the host interface to run at a clock speed independent of the PE. Using such a low-level design paradigm, we are able to leverage FPGA resources within the modules of the FlexRay Controller, thereby saving the remaining area for host implementation.

C. Controller Data Path Extensions

Traditional controllers depend on the host processor to read the received data and determine the usefulness of it. The controller issues a data interrupt, to which the processor responds with a status register read followed by a data read request, subsequently receiving the data. These overheads are wasted in the case of frames with irrelevant data (such as obsolete or untimely data) or multicycle data frames where the processor cannot process the received fragment until more data is available. In the case of critical data frames such as error state that require immediate attention, the latency introduced by the traditional scheme limits the performance of safety-critical systems, which rely on host-triggered recovery. With custom extensions, such exceptions can be handled at the controller, which processes the information and informs the host processor (using interrupts). The host retains absolute control but is not involved in the low-level processing, which is handled instead by the configurable extensions. Fig. 11 describes the functioning of such extensions on the receive path of the controller

On the receive path, the extensions can monitor the received data for matching FlexRay *message ID*, application-based *custom headers* or *timestamp* information, which are contained in the data segment of the FlexRay frame. The FlexRay message ID can be used for application/user-defined communication in *dynamic segment* data frames. An interesting use case is to embed the error status of the ECU into the message ID, which can trigger a fault-recovery procedure in safety-critical units. Application-specific headers may be embedded into the data segment in any frame. Such headers convey information about the data contained in the frame, such as sequence number and length, and are particularly useful in the case of large data transfers, which are accomplished as multicycle transactions on the FlexRay bus. Information in the headers can be used by the controller to repack the multicycle data. The header processing extensions on the receive path can look for such information and reorganize the segmented data and present it as a single transaction to the host.

Similarly, the timestamp validation extension can be configured to reject frames, which are obsolete or untimely. On the transmit path, these extensions can insert relevant headers and timestamp information, as configured. Timestamp resolution is configurable, with a finest resolution of one macrotick and maximum length of 4 bytes (4 B). The header is entirely user configurable and can be matched at the receiver by programming the corresponding registers.

Such extensions on the controller can help extend the functionality and overcome the inherent limitations of the FlexRay network and are impossible to achieve on discrete controllers.

TABLE II
FLEXRAY NODE PARAMETERS

Parameter	Value
Number of Cycles	64
Cycle Duration	5 ms
Number of Static Slots	62
Static Slot duration	65 (macroticks)
Payload Length (Static)	21 words
Number of Dynamic Slots	10 (max)
Symbol Window duration	139 (macroticks)
NIT duration	208 (macroticks)
Sample Clock	12.5 ns
Keyslot ID Assigned	Slot 7
Transmission slots	Slot 7 in cycles 32 and 62

TABLE III
CC IMPLEMENTATION ON HARDWARE

Usage	PM Module	CS Module	MIL Module
Registers	222	1864	732
LUTs	537	3579	1050
BlockRAMs	0	2	2
DSP48A1s	0	3	0
Est. Pow. (mW)	45	66	54
Actual Power	121mW (at 80MHz system frequency)		

Our pipelined architecture in the transmit and receive paths allows us to add this functionality with no additional latency. By standardizing such extensions, automotive networks such as FlexRay can be enhanced to implement a data-layer segment that provides security against replay attacks (using timestamps) and a standard methodology to communicate the health state of ECUs (using headers) [26]. Although such enhancements can be handled by the application in software, this would incur additional processing latency and unwanted complexity at the software level (e.g., timing synchronization).

IV. IMPLEMENTATION RESULTS

To validate our design and to measure the actual performance on hardware, we have implemented the design in a low-power Xilinx Spartan 6 XC6SLX45 FPGA with a host module described using a state machine, modeling a complete ECU. We choose the Spartan 6 as it is a low-cost low-power device, which would be a likely choice for an automotive implementation. To test the network aspects, we emulate a FlexRay bus within the FPGA, using captured raw bus transactions from a real FlexRay network (using Bosch E-Ray controllers) communicating using a predefined FlexRay schedule; these are stored in onboard memory. The information is replayed to create a cycle accurate replica of the transactions on the bus. Our CC is plugged into this FlexRay bus and configured with the same FlexRay parameters. Table II shows a specific set of parameters that was used for our experiments.

Table III details the resource utilization of the individual modules of the controller and the power estimates generated by the Xilinx XPower Analyzer tool, using activity information from simulation. We have configured the core to support all extensions on the transmit and receive path: a 2 byte data header and a 4 bytes timestamp. The maximum achievable frequency for this configuration was 88 MHz. The core is initialized with parameters using a logic-based host model over a PLB/AXI interface. The actual power measured using a power supply probe during operation in hardware is also shown.

TABLE IV
COMPARISON OF IMPLEMENTATIONS

Utilisation	E-Ray[24]	Proposed Implementation		
		Extentions Disabled		Enabled
	Altera Stratix II		Xilinx Spartan 6	
Registers	7754	4966	4910	5612
LUTs	12780	7856	7978	8767
BRAMs	23×M4K	33×M4K	5×9k + 12×18k	5×9k + 13×18k
DSPs	—	12×9-bit	3×DSP48A1	

Table IV compares the resource utilization of our implementation against the platform agnostic E-Ray IP core on the same Altera Stratix-II device. For the purpose of comparison, the consolidated utilization on a Xilinx Spartan 6 is also shown in the same table. It can be observed that the hardware-centric approach results in much better utilization of the heterogeneous resources, leading to a compact implementation. The design can be also easily ported to other Xilinx and Altera devices, as well as to other platforms with a little more effort. The resource utilization and optimizations that we have achieved in comparison with the platform agnostic E-Ray core is significant enough to justify the somewhat reduced portability. With DSP inference disabled, our implementation consumed 8282 LUTs and 5248 registers (on the Stratix-II), which is still less than the E-Ray core. Another advantage is that the power consumption at full operation on a Spartan 6 device is below the power consumed by typical standalone controller chips such as the Infineon CIC-310, which uses the E-Ray IP module [27] and consumes about 150 mW in normal operating mode.

A key advantage of implementing the CC in the FPGA fabric is the ability to compose more intelligent ECU nodes with enhanced communication capabilities on a single device. As an example, we have integrated a fully functional ECU node that combines this controller with a MicroBlaze softcore processor on a Xilinx Spartan 6 XC6SLX45 device, as in Fig. 12. The ECU functions as a front-end processing node for radar-based cruise control and is built using Xilinx Fast Fourier Transform (FFT) IP cores and pipelined logic that performs target detection using a constant false alarm rate (CFAR) scheme [28]. The test data generate 1024 data points every 30 ms, which are transformed to the frequency domain by the FFT module. The CFAR module performs detection on the frequency-domain data using multistage pipelined logic and writes results into the dual-port RAM. The processor is then interrupted, and it consolidates the data over a configurable number of cycles. The controller is configured with parameters defined in Table II. Thus, at cycles 32 and 62, consolidated results are sent on the FlexRay bus.

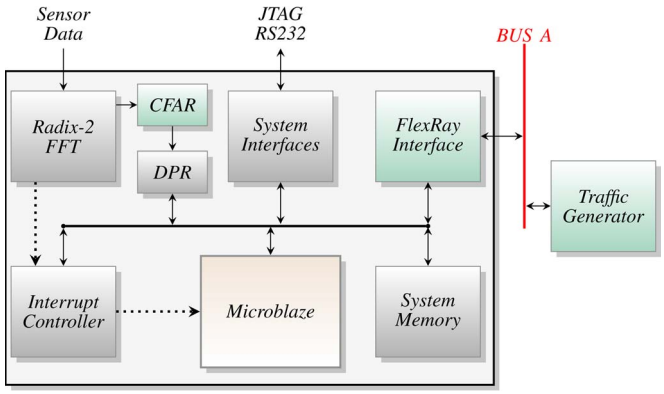

Fig. 12. Integrated ECU function on Spartan-6 FPGA.

TABLE V
SPARTAN-6 IMPLEMENTATION OF ECU ON CHIP

Solution	Metrics	Proposed CC	Hardware Accelerator	Full ECU
Proposed Scheme	Registers	4922	4216	11778
	LUTs	7969	3221	13566
	BlockRAMs	13	11	60
	DSPs	3	44	48
	Power	291 mW		
Discrete Solution	ADSP TS202	596 mW @ 100 MHz clock		
	Discrete CC	150 mW [27]		
	Total Power	746 mW		

Table V details the resource utilization and power consumption measured during operation in hardware. Such an application would otherwise require specialized DSP processors since the latency cannot be met by software implementation on a general-purpose processor [28]. Similar performance can be obtained by interfacing high-performance DSP devices such as the Analog Devices ADSP-TS202S [29] with standalone FlexRay controllers like the Infineon CIC-310 or Freescale S12XF [30], but the node would consume much higher power overall than the integrated FPGA implementation. The key advantage here is that integrating ECU functionality and the network interface on the same device only increases power usage marginally, and this interface can be shared between multiple functions on the same FPGA.

MicroBlaze offers a low-power low-throughput processing option for sensor applications. Alternatively, hybrid platforms such as the Xilinx Zynq can be used for more compute-intensive and real-time applications since they offer a more powerful hard ARM processor. By using AXI-4 for communication between the CC and the host, our design can be used with the ARM in the Zynq (consuming 5612 registers, 8685 LUTs, and two DSP48E1s) or with a MicroBlaze soft processor or a custom hardware ECU.

V. CASE STUDIES

We now present three distinct case studies that showcase the effectiveness of the custom extensions in the context of existing or proposed automotive applications. In each use case, we observe that the application can leverage the intelligence built into the controller, leading to smarter and more efficient systems when compared with standard implementations.

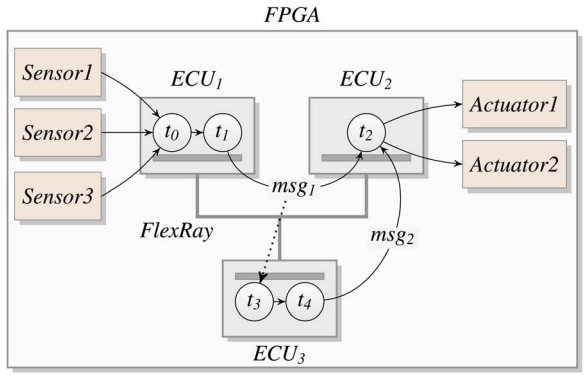

Fig. 13. Test setup for brake-by-wire system.

A. Error Detection and Fallback for Safety-Critical Systems

Safety-critical systems employ redundant or fallback modes, which enable minimum guaranteed functionality, even in the presence of hardware/software faults. One of the critical parameters in such a system is the time taken to switch to fallback mode once a fault has been identified. For this experiment, we model a brake-by-wire system comprising two MicroBlaze ECUs on the FlexRay network: the brake sensor ECU, which interfaces to the sensor modules, and the actuator ECU, which issues commands to the braking system. Each ECU incorporates fallback logic, which is triggered when a fault-status message is received. These status messages are generated by centralized fault detection logic that monitors bus transactions for unsafe commands/data. The sensors and actuators are modeled using memory. Sensor data are generated from a *Sensor BRAM*, and commands are pushed to the *Actuator BRAM*. The sensor ECU combines inputs from the different sensor interfaces periodically and passes it over the FlexRay bus to the processing ECU. The processing ECU uses these data to compute commands and issues them to the actuators. Both ECUs run software routines on the popular *FreeRTOS* platform (denoted RT). A simplified model of the test setup is shown in Fig. 13.

To mimic the behavior of off-the-shelf controllers, we disable the custom extensions on the CC. A fault-status message is triggered on the sensor ECU system by configuring invalid data in the *Sensor BRAM*, causing incorrect sensor data to be issued to actuator ECU over the FlexRay bus. The fault-detector logic detects the error and transmits the error code in the next slot assigned to it. A normal controller decodes this message and passes it to the MicroBlaze processor, where the data is processed to trigger fallback mode. The latency from the transmission of the error message to the triggering of fallback mode is largely determined by the interrupt-based data passing mechanism used in off-the-shelf controllers. Even for an RTOS-based (real-time) system, this latency can be significant and was measured at an average of 9.05 ms for our implementation, as shown in Fig. 14.

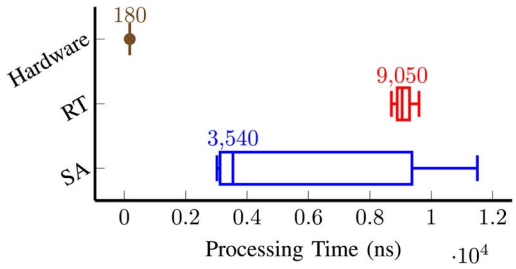

Fig. 14. Latency distribution for interrupt-based critical data processing.

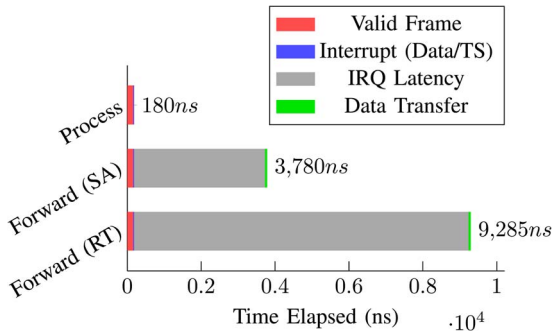

Fig. 15. Timestamp processing at interface.

By moving such critical data processing to the controller, it becomes possible to significantly reduce this delay and enhance the determinism of the system. To quantify this, a processing extension that detects packets on a user-configured slot with a user-specified data header is enabled on the CC. On detecting this combination, the controller can either process the remaining data for specific patterns or trigger an interrupt. In this particular experiment, it is configured to process the critical error flags and the consecutive error numbers to decide whether to trigger fallback mode. This generates a direct interrupt to the MicroBlaze processor and enables fallback mode, resulting in a faster and more consistent turnaround time (average 50× faster than RT), as shown in Fig. 14.

We have also repeated the experiment using the Xilinx Standalone operating system (denoted SA), the lightweight minimalist OS for MicroBlaze. It is shown in Fig. 14 that, although the simplified SA OS results in lower average interrupt latencies than the RT, it results in a larger spread of latencies.

B. Time Awareness for Messages

A major security risk in time-triggered systems such as FlexRay is the lack of time awareness for messages. By monitoring bus transactions, an external agent can easily employ simple replay attacks, flooding the bus with stale data, as described in [31]. The FlexRay protocol leaves this vulnerability to the higher layer applications to manage. In our controller, the transmit path allows messages to be optionally timestamped to make the message time aware, at the cost of increased payload size. By inserting the header and timestamp within the data segment of the FlexRay frame, it is transparent to other FlexRay controllers present on the network, ensuring interoperability with off-the-shelf controllers. With timestamps enabled, the receive path can be configured to automatically drop frames that are outside an allowed time window. This creates a basic security layer at each ECU, which can be augmented further by incorporating encryption/decryption logic in the data path.

An interesting use case is in high-performance gateways that move data between network clusters. With traditional interfaces, messages arriving from each interface will be forwarded to the switch logic, which decides whether to forward the data to their destination or drop them because they have expired. By building intelligence into the controller, the validity of data can be determined before they are forwarded to the switching logic. We modify the experimental setup described in Fig. 13 earlier to model a gateway configured to discard untimely data, either at the processing logic (MicroBlaze), mimicking off-the-shelf interfaces, or at the interface using our enhanced controller extensions. Our tests show that the interface can process the timestamp and discard the message within 180 ns of frame reception. A standard approach consumes a further 3.6 and 9.1 μs on average, for SA and RT, respectively, as shown in Fig. 15, since the data must be processed by the host.

C. Handling Volume Data at Interfaces

Applications such as radar-based cruise control utilize volume data gathered by the radar sensors to compute distance and relative velocity of other vehicles in the vicinity. A complete data set from a sweep is required by the processing logic to determine these parameters, and these data are received over many data slots. The processing ECU must reassemble these fragments before the data can be processed. By moving this packing/repacking to the controller interface, the processing logic can overlap the computation with data reception, enabling it to run at lower frequencies and, hence, consume less power.

To demonstrate this, we use the experimental setup for the radar-based cruise control ECU, described earlier in Section IV. The data from the radar sensor are received over the FlexRay bus in bursts of 256 bytes, which is the maximum payload size defined by FlexRay standard. The MicroBlaze runs software routines on top of the Xilinx Standalone OS. In a normal design, the processor is interrupted each time a block of data is received. The processor responds with the first data read request 12 ms (worst-case) after receiving the interrupt, with the burst read consuming a further 3.84 ms. This is repeated over four cycles to complete the data transfer, cumulatively consuming 63.36 ms.

We then test the same application with an extension that allows the controller to intelligently buffer the entire frame in a buffer, only interrupting the processor at the end of the transaction. This enables the processor to issue back-to-back reads from the controller, completing the entire data movement in 27.36 ms from the reception of the interrupt. To provide a balance between multicycle and single-cycle data, the design has been constrained to handle up to four data cycles at full payload size. To support larger data sizes, larger buffer memory must be added to the controller, resulting in higher device utilization; however, this may be a tolerable cost for some ECUs, and the CC architecture supports it.

The experiment was also repeated using the FreeRTOS-based software, which provided better determinism than the Standalone-based scheme, resulting in a lower worst-case interrupt latency, as shown in Fig. 16.

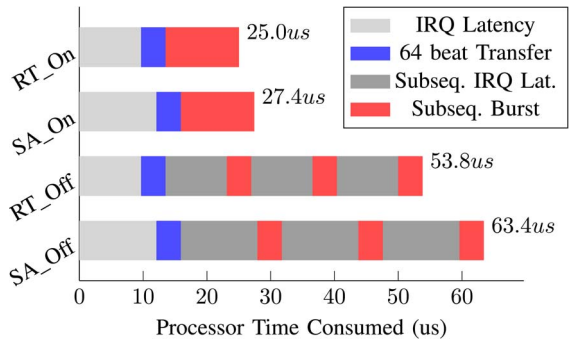

Fig. 16. Data repacking for multicycle data transfers.

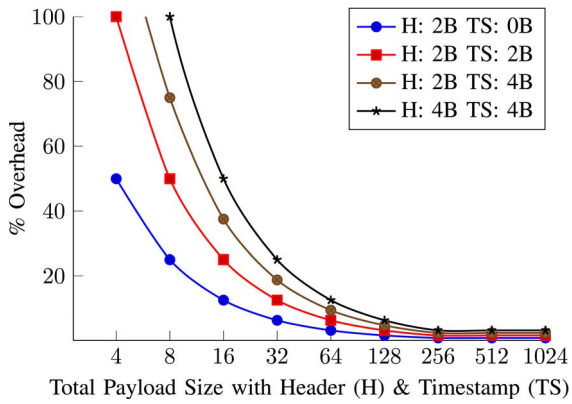

Fig. 17. Overheads for including headers and timestamps.

VI. Discussion

The FlexRay protocol does not define the usage of headers within the data segment, which is entirely dependent upon user implementation. While the usage of headers and timestamps within data provides the aforementioned advantages, it may result in significant payload overheads for small data sizes, while also limiting the payload capability of a FlexRay frame. Fig. 17 compares the overheads associated with different configurable values for the application header and timestamp, as a function of the payload size. As can be observed, at lower payload sizes, the inclusion of a timestamp and application header results in large overheads, but for large payload sizes, the penalty paid is very small. Beyond the maximum payload size of 256 B, additional data have to be handled as multicycle transactions, causing the curve to flatten out for higher payload sizes. Since the application header and timestamp data are inserted within the data segment of the FlexRay frame, it is transparent to other FlexRay controllers on the network, ensuring interoperability with standard controllers.

We have purposefully designed the controller's architecture to coexist with ECU functions on the same FPGA. Doing so allows us to leverage the computational capabilities of FPGAs for implementing ECU functions while no longer requiring a discrete network controller. We can also incorporate partial reconfiguration to allow multiple applications to interface with the bus through a single controller and to define fault-tolerant ECUs for safety-critical functions [32].

Furthermore, timestamps and data processing capabilities within the controller can also be used extensively for functional validation of novel applications, architectures, and network features. On a large enough FPGA such as the Virtex-7, we can integrate up to ten ECUs, network controllers, and the actual network to create a validation platform (replicating an actual car network) for functional verification [33].

VII. Conclusion

In this paper, we have given an overview of the FlexRay protocol and the generic architecture of the CC, as defined by the specification. By identifying and extracting operations that are mutually exclusive or natively parallel, we have designed a custom controller that takes advantage of the heterogeneous resources on modern FPGAs, resulting in reduced logic footprint and low power consumption, while providing a host of features beyond those described by the standard. Advanced computational capabilities such as fault tolerance and function consolidation can be built into nodes that integrate complex ECU functions with advanced CCs. This approach also improves power consumption compared with the use of discrete controllers. We hope that our flexible and configurable architecture can be leveraged for continued research on intelligent FlexRay nodes and switches on FPGAs, leading to wider adoption of reconfigurable hardware for in-vehicle applications.

We aim to investigate extending this controller for use with partial reconfiguration to provide flexible use of the FPGA fabric, enabling further sharing of communication resources between ECUs. We intend to develop intelligent FlexRay nodes and switches on reconfigurable hardware that are energy efficient and that will allow us to explore more advanced network setups. Finally, the principles demonstrated in this paper are also applicable to other time-triggered interfaces, and we hope to explore this for time-triggered Ethernet.

References

[1] S. Chakraborty *et al.*, "Embedded systems and software challenges in electric vehicles," in *Proc. Des., Autom. Test Eur. (DATE) Conf.*, Mar. 2012, pp. 424–429.

[2] S. Shreejith, S. A. Fahmy, and M. Lukasiewycz, "Reconfigurable computing in next-generation automotive networks," *IEEE Embedded Syst. Lett.*, vol. 5, no. 1, pp. 12–15, Mar. 2013.

[3] I. Sheikh, M. Hanif, and M. Short, "Improving information throughput and transmission predictability in Controller Area Networks," in *Proc. IEEE Int. Symp. Ind. Electron. (ISIE)*, Jul. 2010, pp. 1736–1741.

[4] J. Kötz and S. Poledna, "Making FlexRay a Reality in a Premium Car," in *Proc. Convergence Transportation Electronics Association, SAE Int.*, 2008, pp. 391–395.

[5] *Specification of FlexRay Interface Version 3.2.0*, AUTOSAR Std. [Online]. Available: http://www.autosar.org

[6] *FlexRay Communications System, Protocol Specification Version 2.1 Revision A*, FlexRay Consortium Std., Dec. 2005. [Online]. Available: http://www.flexray.com

[7] P. Milbredt, B. Vermeulen, G. Tabanoglu, and M. Lukasiewycz, "Switched FlexRay: Increasing the effective bandwidth and safety of FlexRay networks," in *Proc. Conf. Emerging Technol. Factory Autom. (ETFA)*, Sep. 2010, pp. 1–8.

[8] T. Schenkelaars, B. Vermeulen, and K. Goossens, "Optimal Scheduling of Switched FlexRay Networks," in *Proc. Des., Autom. Test Eur. (DATE) Conf.*, Mar. 2011, pp. 1–6.

[9] M. Lukasiewycz, S. Chakraborty, and P. Milbredt, "FlexRay switch scheduling—A networking concept for electric vehicles," in *Proc. Des., Autom. Test Eur. (DATE) Conf.*, Mar. 2011, pp. 1–6.

[10] T. Forest *et al.*, "Physical architectures of automotive systems," in *Proc. Des., Autom. Test Eur. (DATE) Conf.*, Mar. 2008.
[11] M. Lukasiewycz, M. Glaß, J. Teich, and P. Milbredt, "FlexRay schedule optimization of the static segment," in *Proc. Int. Conf. Hardware/Software Codes. Syst. Synthesis (CODES+ISSS)*, 2009, pp. 363–372.
[12] K. Schmidt and E. G. Schmidt, "Message scheduling for the FlexRay protocol: The static segment," *IEEE Trans. Veh. Technol.*, vol. 58, no. 5, pp. 2170–2179, Jun. 2009.
[13] E. G. Schmidt and K. Schmidt, "Message scheduling for the FlexRay protocol: The dynamic segment," *IEEE Trans. Veh. Technol.*, vol. 58, no. 5, pp. 2160–2169, Jun. 2009.
[14] J. J. Nielsen and H. P. Schwefel, "Markov chain-based Performance evaluation of FlexRay dynamic segment," in *Proc. Int. Workshop Real Time Netw.*, 2007, pp. 1–6.
[15] B. Kim and K. Park, "Probabilistic delay model of dynamic message frame in FlexRay protocol," *IEEE Trans. Consum. Electron.*, vol. 55, no. 1, pp. 77–82, Feb. 2009.
[16] X. He, Q. Wang, and Z. Zhang, "A survey of study of FlexRay systems for automotive net," in *Proc. Int. Conf. Electron. Mech. Eng. Inf. Technol.*, Aug. 2011, pp. 1197–1204.
[17] C. Schmutzler, A. Lakhtel, M. Simons, and J. Becker, "Increasing energy efficiency of automotive E/E-architectures with intelligent communication controllers for FlexRay," in *Proc. Int. Symp. Syst. Chip (SoC)*, Oct./Nov. 2011.
[18] J. Sobotka and J. Novak, "FlexRay controller with special testing capabilities," in *Proc. Int. Conf. Appl. Electron. (AE)*, Sep. 2012, pp. 269–272.
[19] J. Y. Hande, M. Khanapurkar, and P. Bajaj, "Approach for VHDL and FPGA implementation of communication controller of FlexRay controller," in *Proc. Int. Conf. Emerging Trends Eng. Technol., ICETET*, Dec. 2009, pp. 397–401.
[20] Y.-N. Xu, Y. E. Kim, K. J. Cho, J. G. Chung, and M. S. Lim, "Implementation of FlexRay communication controller protocol with application to a robot system," in *Proc. IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Aug./Sep. 2008, pp. 994–997.
[21] Y.-N. Xu, I. Jang, Y. Kim, J. Chung, and S.-C. Lee, "Implementation of FlexRay protocol with an automotive application," in *Proc. Int. SoC Des. Conf. (ISOCC)*, Nov. 2008, pp. II-25–II-28.
[22] P. Szecowka and M. Swiderski, "On hardware implementation of FlexRay bus guardian module," in *Proc. Int. Conf. Mixed Des. Integr. Circuits Syst. (MIXDES)*, Jun. 2007, pp. 309–312.
[23] G. N. Sung, C. Y. Juan, and C. C. Wang, "Bus guardian design for automobile networking ECU nodes compliant with FlexRay standards," in *Proc. Int. Symp. Consum. Electron.*, Apr. 2008, pp. 1–4.
[24] *Product Information: E-Ray IP Module*, Robert Bosch GmbH, Jul. 2009.
[25] *FRCC2100: Product Brochure, Freescale FlexRay Communications Controller Core*, IPextreme, Inc., Campbell, CA, USA.
[26] S. Shreejith and S. A. Fahmy, "Enhancing communication on automotive networks using data layer extensions," in *Proc. Int. Conf. Field Programmable Technol. (FPT)*, Dec. 2013, pp. 470–473.
[27] *SAK-CIC310-OSMX2HT, FlexRay Communication Controller Data Sheet*, Infineon Technologies AG, Munich, Germany, Jun. 2007.
[28] J. Saad, A. Baghdadi, and F. Bodereau, "FPGA-based radar signal processing for automotive driver assistance system," in *Proc. Int. Symp. Rapid Syst. Prototyping*, Jun. 2009, pp. 196–199.
[29] F. Greg, "Estimating Power for the ADSP-TS202S TigerSHARC Processors," Analog Devices, Norwood, MA, USA, EE170, 2006, Tech. Rep.
[30] *MC9S12XF512 Reference Manual, Rev.1.20 ed.*, Freescale Semiconductors, Denver, CO, USA, Nov. 2010.
[31] I. Rouf *et al.*, "Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study," in *Proc. USENIX Conf. Security*, 2010, pp. 1–16.
[32] 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. (DATE) Conf.*, Mar. 2013, pp. 721–724.
[33] S. Shreejith, S. A. Fahmy, and M. Lukasiewycz, "Accelerating validation of time-triggered automotive systems on FPGAs," in *Proc. Int. Conf. Field Programmable Technol. (FPT)*, Dec. 2013, pp. 4–11.

Shanker Shreejith (S'13) received the B.Tech. degree in electronics and communication engineering from University of Kerala, India, in 2006. Since 2011, he has been pursuing Ph.D. degree with the School of Computer Engineering, Nanyang Technological University, Singapore, working on reconfigurable computing in automotive systems with TUM CREATE, Singapore.

From 2006 to 2008, he was a Design and Development engineer with ProcSys, India. From 2008 to 2011, he was a Scientist with the Vikram Sarbhai Space Centre, Trivandrum, India, under the Indian Space Research Organisation (ISRO).

Suhaib A. Fahmy (M'01–SM'13) received the M.Eng. degree in information systems engineering and the Ph.D. degree in electrical and electronic engineering from Imperial College London, London, U.K., in 2003 and 2007, respectively.

From 2007 to 2009, he was a Research Fellow with Trinity College Dublin, Dublin, Ireland, and a Visiting Research Engineer with Xilinx Research Labs, Dublin. Since 2009, he has been an Assistant Professor with the School of Computer Engineering, Nanyang Technological University, Singapore. His research interests include reconfigurable computing, high-level system design, and computational acceleration of complex algorithms.

Dr. Fahmy is a Senior Member of the Association for Computing Machinery. He received the Best Paper Award at the IEEE Conference on Field Programmable Technology in 2012 and the IBM Faculty Award in 2013.