A Time-predictable Open-Source TTEthernet End-System

Eleftherios Kyriakakis, Maja Lund, Luca Pezzarossa, Jens Sparsø, and Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: elky@dtu.dk, maja_lala@hotmail.com, {lpez, jspa, masca}@dtu.dk

***Abstract*—Cyber-physical systems deployed in areas like automotive, avionics, or industrial Internet of things are often distributed systems. The operation of such systems requires coordinated execution of the individual tasks with bounded communication network latency to guarantee quality-of-control. Both the time for computing and communication needs to be bounded and statically analyzable.**

To provide deterministic communication between end-systems, real-time networks can use a variety of industrial Ethernet standards typically based on time-division scheduling and enforced by real-time enabled network switches. For the computation, end-systems need time-predictable processors where the worst-case execution time of the application tasks can be analyzed statically.

This paper presents a time-predictable end-system with support for deterministic communication using the open-source processor Patmos. The proposed architecture is deployed in a TTEthernet network, and the protocol software stack is implemented, and the worst-case execution time is statically analyzed. The developed end-system is evaluated in an experimental network setup composed of six TTEthernet nodes that exchange periodic frames over a TTEthernet switch.

I. Introduction

Advancements in the field of safety-critical systems for industrial control and avionics/automotive automation have brought a recent focus on distributed real-time system communication [45]. This trend is emphasized by the upcoming paradigm of Industry 4.0 and the recent efforts of the time-sensitive networking (TSN) group [15] to develop a set of deterministic Ethernet standards that meets the requirement for system's interoperability and real-time communications.

The correct execution of real-time applications depends both on the functional correctness of the result as well as the time it takes to produce the result. A typical example that illustrates this criticality of functional and temporal correctness is a vehicle collision avoidance system. In this situation, the processor must correctly detect a possible object, but it is equally important that communication time, from the camera to the processor, and processing time, of the image on the processor, are deterministically bounded to consider the system as correctly functioning.

To achieve deterministic communication with bounded latency, real-time systems often employ a time-triggered (TT) communication schemes such as the well-known time-triggered protocol [16], which is based on a cooperative schedule and a network-wide notion of time [36]. This approach can be implemented on Ethernet communications, such as TTEthenet [18] and TSN [10], to provide the guaranteed networking services (latency, bandwidth, and jitter) required by distributed real-time applications such as avionics, automotive, and industrial control systems.

In this work, we focus on safety-critical systems and thus investigate the TTEthernet protocol. TTEthernet uses a fault-tolerant synchronized communication cycle with strict guarantees for transmission latency [18]. In addition, the protocol provides support for rate-constrained traffic and best-effort traffic classes. TTEthernet has been standardized under the aerospace standard SAE AS6802 [44] and has been implemented as a communication bus replacement in both automotive and aerospace real-time applications [7], [24].

This paper presents a time-predictable TTEthernet end-system implemented on the open-source processor Patmos [34], allowing for static WCET analysis of the networking code. This work enables the Patmos processor to communicate through a deterministic network protocol, allowing the possibility for end-to-end bounded latency communication that includes the software stack. We do so by extending the existing Ethernet controller driver and implement a TTEthernet software stack. We test the performance of the controller driver by evaluating the achieved clock synchronization and correct exchange of TT frames and perform a static WCET analysis of the software stack.

The main contributions of this work are:

- A TTEthernet node that combines time-predictable execution of tasks with time-triggered communication through TTEthernet
- A WCET analyzable Ethernet software stack, which allows to statically guarantee that all deadlines and end-to-end timing requirements are met
- Performing a comparative analysis of the effects of number of integration cycles against the achieved clock synchronization
- Implementing a PI controller that improves the achieved clock synchronization precision

To the best of our knowledge, this is the first WCET analyzable TTEthernet node that combines time-predictable communication over Ethernet with time-predictable execution of tasks. The presented design is available in open source.[1] An initial version of this work has been presented in [25].

[1] see https://github.com/t-crest/patmos

This paper is organized into six sections: Section II presents related work on TTEthernet. Section III provides a background on the TTEthernet internals. Section IV describes the design and implementation of a time-predictable TTEthernet node. Section V evaluates our design with measurements and static WCET analysis performed on a system consisting of a switch and six nodes. Section VI concludes the paper.

II. Related Work

Traditional real-time communication was based on bus protocols such as CAN and PROFIBUS that can send small prioritized frames with bounded latency. CAN was later extended with TT capabilities. This TTCAN bus restricts nodes to only transmit frames in specific time slots, thus increasing the determinism of the communication [22]. The main limitations of both CAN and TTCAN are a maximum bandwidth of 1 Mbit/s, and limited cable length, which depends on the bandwidth of the bus [3]. To overcome these obstacles, FlexRay was introduced to replace CAN discussed in [35].

As the demand for higher bandwidth increases, the industry has started looking towards Ethernet-based real-time protocols. Several different standards and protocols, such as EtherCAT, Ethernet Powerlink, and TTEthernet, were developed, some of which have been compared in [8].

Another emerging Ethernet protocol for real-time systems is TSN [10]. TSN emerges from the audio-video bridging (AVB) protocol with the addition of a time-scheduled mechanism for real-time communication through the use of a gate control list on the transmission paths. Worst-case analysis of TSN networks is presented in [48]. The schedulability of the gate control list has been investigated by various works such as [12], [29] that showed that rate-constrained traffic can co-exist with time-triggered but introduces small jitter. In [6], the authors achieve zero jitter determinism of TT frames by enforcing time-based isolation of the traffic flows but reducing the solution space for TSN networks. The timing synchronization mechanism of TSN is based on the well known IEEE 1588 Precise Time Protocol which has been characterized by [19] and experimentally verified to achieve sub-microsecond precision by various works such as [13], [20], [23].

Both protocols, TSN and TTEthernet, aim to provide support for TT communication. They have been directly compared in [47], and the two protocols focus on different real-time system requirements and provide different levels of criticality. TSN offers greater flexibility and bandwidth fairness over TTEthernet but is only suitable for soft-real time traffic due to its lack of fault-tolerant clock synchronization and low granularity scheduling mechanism [5].

Research on TTEthernet communication has been focused on the following perspectives: (a) timing analysis of the communication links [49], [50], (b) schedule synthesis for frame flows [9], [40], [42], and (c) investigating the clock synchronization [1], [37]. In contrast, this work investigates the implementation characteristics of a TTEthernet compatible end-system that supports WCET analysis of the software on the end-system and its integration within a TTEthernet network.

Latency and jitter through a single TTEthernet switch have been measured in [2] using off-the-shelf components combined with a proprietary TTEthernet Linux driver. A performance validation setup was presented for TTEthernet networks, and the relation between end-to-end latency, jitter, and frame size was investigated. A comparison between a commercial off-the-shelf switch and a TTEthernet switch was presented as a function of link utilization and end-to-end latency. This emphasized the benefits of TTEthernet over standard Ethernet switches. The measured jitter for the system was dependent on frame size, and the authors observed a jitter of 10 μs for frames smaller than 128 bytes and 30 μs for larger frames. Furthermore, a model of analyzing the worst-case latency of TTEthernet in the existence of rate-constrained traffic load is presented in [50]. The model shows that it is possible to provide safe bounds for rate-constrained traffic, and it is evaluated over a simulated network topology of an Airbus A380 aircraft.

Scheduling of TTEthernet communication has been investigated in [40]. It proposes a scheduling approach for TT traffic that allows the calculation of the transmission and reception time instants by each connected real-time application. The synthesis for static scheduling for mixed-criticality systems has been investigated in [42]. The concept of schedule porosity was introduced, allowing un-synchronized (best-effort or rate-constrained) traffic to be mixed with time-triggered traffic without suffering from starvation. Moreover, in [41], the authors further optimize TTEthernet schedules for mixed-criticality applications by presenting a schedule that allocates more bandwidth to best-effort traffic while still preserving determinism of TT traffic.

Clock synchronization is an essential part of TTEthernet as it guarantees the synchronized communication of the network end-systems according to the global schedule. Depending on the clock synchronization accuracy requirements of an application, the minimum number of integration cycles per cycle period can be calculated [31]. In [46], the authors investigate in a simulated environment a least-squares algorithm that manages the compensation of the error. In both cases, accurate measurements of the achieved synchronization accuracy, i.e., standard deviation and avg/max/min values, are not discussed, and the methodology is implemented on a simulated environment of a TTEthernet clock. In our setup, we use the TTEthernet clock synchronization mechanism but improve the clock error by adding a PI controller.

In our review of related work, we identified that most papers focus on analyzing the communication components of TTEthernet. We found just a single paper that described the implementation and analysis of a TTEthernet end-system. A software-based TTEthernet end-system has previously been developed for AUTOSAR [11], which is a standardized software architecture for control units in cars. The implemented AUTOSAR system acts as a synchronization client and uses existing hardware capabilities of Ethernet controllers to timestamp incoming clock synchronization frames, and the authors observed a jitter of approximately 32 μs. Regarding the processing time of the protocol the authors provide CPU

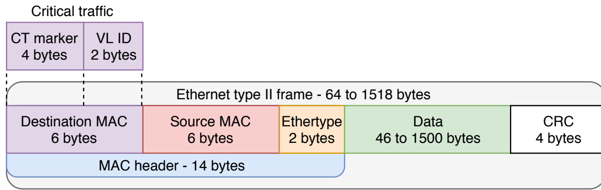

Fig. 1. Ethernet type II frame. Critical traffic identifies a destination using a CT marker and VL ID instead of a MAC address.

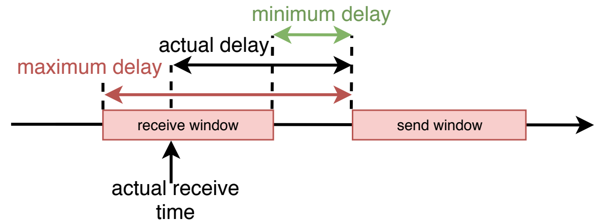

Fig. 2. Switch delay in relation to the receive and send windows.

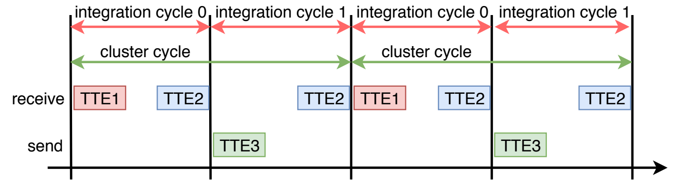

Fig. 3. Example of integration and cluster cycles.

utilization and memory overhead metrics. Precise end-to-end latency of the system is unclear due to a non-deterministic dispatch and receive function. In contrast to our work, the authors do not provide WCET analysis of these functions, and although they discuss the importance of these delays in the calculation of the end-to-end latency, they do not provide measurements or static timing analysis. We provide static WCET analysis of all software components of our TTEthernet stack.

To the best of our knowledge, our paper is the first to present a WCET analyzable TTEthernet end-system that combines time-predictable communication over Ethernet with time-predictable execution of tasks. The presented design is available in open source.

III. TTEthernet Background

A. Overview

The deterministic communication capabilities offered by TTEthernet are based on special switches that handle TT traffic according to a global schedule, as well as end-system equipped with TTEthernet capable controllers for transmission and clock synchronization. TTEthernet technology is proprietary. However, an initial version of the switch architecture is presented in [39]. The design of the first hardware TTEthernet controller is presented in [38].

TTEthernet supports best-effort (BE) traffic and two types of critical traffic (CT): rate-constrained [4], [49] and time-triggered (TT). TT traffic takes priority over rate-constrained traffic, which takes priority over best-effort traffic. This paper focuses on TT traffic.

CT is sent and received using the concept of a virtual link (VL). A VL is a relation between a sender and one or more receivers and is identified by a unique ID. Switches know the VL definitions, and nodes know on which VL they are allowed to send. CT is formatted as standard Ethernet frames, but it differs from best-effort traffic by having the destination MAC address field used for the CT marker and for the VL on which the frame belongs, as shown in Figure 1. Depending on the VL, switches can forward the CT frame to the right port.

B. Time-Triggered Traffic

TT traffic is transmitted at pre-defined time slots. Thus, a VL definition includes a receive window where the switch accepts frames from the sender, and send windows where the switch passes frames to the receivers. The switch ignores all frames received outside of the defined window in order to guarantee bounded end-to-end latency and minimal jitter for other frames. The latency depends on the delay between the receive and the send windows in the switch. This latency is called switch delay. Figure 2 shows the possible minimum and maximum delays of an outgoing frame in relation to the receive and send windows.

The latency also depends on the transmission time (frame size over bandwidth) and the propagation delay (cable distance over propagation speed). For a system with short wires, the propagation delay is in the range of nanoseconds. The expected minimum and maximum latency for a VL in a given TTEthernet system can be calculated using Equations 1 and 2. 64 and 1518 are the minimum and maximum possible Ethernet frame sizes, SD_{min} and SD_{max} are the minimum and maximum switch delays, d the network cable length, and s the propagation speed.

$$L_{min} = SD_{min}(s) + \frac{64\ bytes \cdot 8\frac{bit}{byte}}{bandwidth(\frac{bit}{s})} + \frac{d_{cable}(m)}{s_{cable}(\frac{m}{s})} \tag{1}$$

$$L_{max} = SD_{max}(s) + \frac{1518\ bytes \cdot 8\frac{bit}{byte}}{bandwidth(\frac{bit}{s})} + \frac{d_{cable}(m)}{s_{cable}(\frac{m}{s})} \tag{2}$$

C. Clock Synchronization

All nodes and switches need a global notion of time to send frames at the right moment in time. Clock synchronization is carried out periodically every integration cycle (typically in the range of 1 to 10 milliseconds). The schedule for TT traffic is also periodic and repeats every cluster cycle, which is an integer multiple of the integration cycle. Figure 3 shows an example of TT traffic in a system with an integration period of 10 ms and two integration cycles per cluster cycle. The schedule defines TT traffic by its period and the offset from the start of the cluster cycle. For example, TTE3 in Figure 3 has a period of 20 ms and an offset of 10 ms.

Clock synchronization is achieved through the exchange of protocol control frames (PCF). There are three types of PCFs

in TTEthenet: integration frame, cold-start frame, and cold-start acknowledge frame. Integration frames are used in the periodic synchronization, while the last two types of PCF are used exclusively during start-up. PCFs are used for synchronization only when they become permanent. This happens at the point in time when the receiver knows that all related frames that have been sent to it prior to the send time of this frame have arrived or will never arrive [17]. The permanence point in time ($Permanence_{PIT}$) is calculated by the TTEthernet protocol as the worst-case delay (D_{max}) minus the dynamic delay (D_{actual}) that a synchronization frame experiences plus the reception timestamp as shown in Equation 3. The dynamic delay (D_{actual}) is provided by the frames transparent clock value. This mechanism allows for a receiver to re-establish the send order of frames, and it used for remote clock reading during a synchronization operation. Assuming the transparent clock depicts the transmission time (D_{actual}) and based on the statically scheduled receive point in time ($ScheduledRX_{PIT}$) the clock difference ($ClockDiff$) is calculated as in Equation 4.

$$Permanence_{PIT} = RX_{PIT} + (D_{max} - D_{actual}) \quad (3)$$

$$ClockDiff = ScheduledRX_{PIT} - Permanence_{PIT} \quad (4)$$

Switches and nodes are involved in the exchange of PCFs for synchronization in three different roles: synchronization masters, synchronization clients, and compression masters. Typically, the switches act as compression masters, and the nodes are either synchronization masters or clients. Each node keeps track of when they believe the integration cycle has started, which is when synchronization masters send out integration frames. Compression masters use the permanence times of these frames to decide on the correct clock and send integration frames to all synchronization masters and clients. A returning integration frame is expected to be permanent $2 \cdot max_delay + comp_delay$ after the beginning of the integration cycle, where $comp_delay$ is the time it takes a compression master to evaluate the frames. An acceptance window around this expected permanence point defines whether or not the node should accept the PCF as correct. The acceptance window has a width of twice the expected precision of the system, defined as the maximum difference between two correct local clocks. If the PCF is accepted, the difference between the expected and actual permanence time is used to correct the clock. Correction is typically delayed until it is sure that the corrected clock will not fall back within the acceptance window, as shown in Figure 4. If more than one compression master is present, the synchronization masters and clients receive multiple PCF in the acceptance window. In this case, the clock correction uses a fault-tolerant average of the differences between expected and actual permanence time.

IV. Design and Implementation of the TTEthernet Node

In this section, we present the design and implementation of our TTEthernet node. First, we describe the hardware platform. Then, we explain the functionality of the developed software stack. Finally, we present the theoretical limits of the implementation.

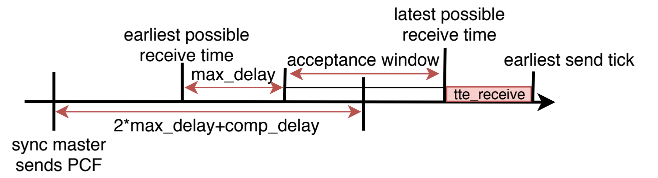

Fig. 4. Overview of clock synchronization, adapted from [28].

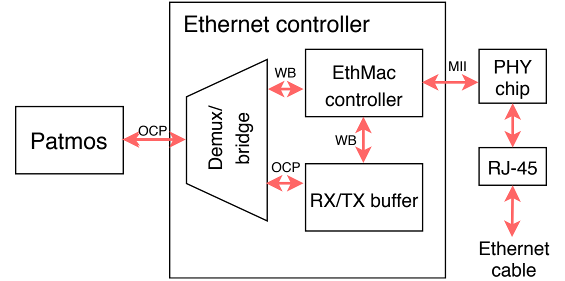

Fig. 5. Overview of the Patmos Ethernet controller, adapted from [30]. It is connected to Patmos through OCP signals, and to a physical PHY chip through MII.

We provide a time-predictable end node for a TTEthernet system, including hardware design and WCET analysis of the network software. We focus on time-predictable program execution and traffic transmission. Generating the static schedule for the time-triggered traffic and allocation of TT frames is out of the scope of this paper. We rely on available solutions, e.g., the scheduling tool that is part of the TTEthernet toolset.

A. Hardware

The proposed TTEthernet node is based on the Patmos [34], a time-predictable processor used in the T-CREST platform [32], [33], and on an open-source Ethernet controller. The controller is based on the EthMac block from OpenCores [27], which was previously ported for Patmos [30].

Figure 5 shows the hardware architecture of the node. The Patmos processor, as well as the RX/TX buffer, uses a variant of the OCP interface, while the EthMac block uses the Wishbone interface. A dedicated multiplexing bridge component manages the conversion between the two protocols. It allows Patmos to access the configuration registers in the EthMac controller and the RX/TX buffer as memory-mapped IO devices. The EthMac controller connects to the PHY chip through the media-independent interface (MII).

Receiving and transmitting frames from the EthMac block is based on buffer descriptors. These are data structures stored in the EthMac controller and containing the address to an associated buffer in the RX/TX buffer component, as well as the length and status of the buffer. The EthMac controller can receive a new frame only when there is at least one available receive buffer. Otherwise, the EthMac controller discards the frame. After receiving a frame, the controller writes the receive

status into the associated buffer descriptor, and the controller may generate an interrupt (if enabled). The buffer will stay unavailable until the driver software marks the buffer descriptor empty again.

To send and receive TT traffic, no changes are required to the hardware architecture of the existing Ethernet controller [30]. However, the EthMac core was configured in promiscuous mode to avoid any filtering of frames on MAC addresses. Additionally, it was configured as full-duplex to avoid sending and receiving blocking each other. The functionality of the proposed node entirely lies in software, in the C library `tte.c`.

We implemented two different versions of the proposed solution: (1) where the program discovers received frames through polling, and (2) where the Ethernet controller triggers an interrupt whenever a frame is received. Using interrupts for time stamping of an arriving Ethernet frame is not the best solution since the start of the execution of the interrupt routine introduces jitter due to cache hits and misses. This receive jitter is critical in our implementation as it degrades the timestamp precision and results in lower clock synchronization quality. The jitter was measured at -26 μs, with the resulting clock precision varying between -10 μs and 16 μs. Further results regarding the evaluation of the clock synchronization are discussed in Section V-B.

The polling solution solves this problem by using a periodic task that is scheduled to be released just before the next synchronization frame arrives. The release time needs to include enough time to contain the worst-case preemption delay and the possible jitter of the PCF itself. In this case, the processor is ready to listen to the Ethernet port in a tight loop in order to get a better timestamp in software. Therefore, in the polling solution, the actual polling runs only for a short time, without wasting processor time. As future work, we plan to change the Ethernet controller to include hardware support for time-stamping [21].

B. Software

Our node only acts as a synchronization client and only connects to a single switch. The developed software stack offers three main functionalities: initialization, receiving, and sending.

Figure 6 shows the intended flow of programs using the developed system. At first, the program initializes the controller with static information regarding the system, information on VLs, and the schedule. After initialization, the periodic task starts. It contains a call to the application code, which the programmer needs to organize as a cyclic executive, and then polling the Ethernet controller when a new frame is expected to arrive.

It is necessary to ensure that the receive time of integration frames is recorded as precisely as possible to enable correct clock synchronization. The received frame is then passed through the `tte_receive` function, which will synchronize the local clock in case of an integration frame, or otherwise return a value indicating the frame type. The rest of the body depends on the purpose of the program itself.

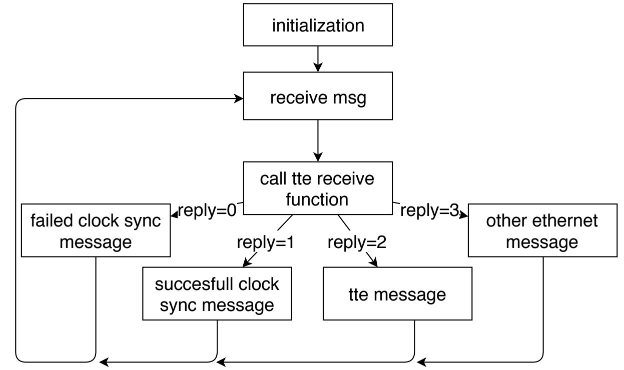

Fig. 6. The intended flow of user programs. The controller is first initialized with all constants of the system. The regular operation is performed by a loop where the program continually waits until a frame is received, calls the `tte_receive` function, and then reacts to the reply.

Outgoing TT frames can be scheduled anywhere in the program body and will be sent according to the system schedule through timer interrupts. To avoid fluctuations in the clock synchronization, the system schedule, and the WCET of the program body should follow the limits described in Section IV-C.

1) Initialization: The system needs to be initialized before a program loop can start. The initialization data includes: the integration cycle, the cluster cycle, how many VLs the node can send on, the maximum transmission delay, the compression delay of the switch, and the system precision. Furthermore, the permitted send time for each VL needs to be known, so each VL gets initialized with an ID, an offset, and a period. The TTEthernet switch ensures that only relevant and correctly timed TT frames are passed on to the network. As soon as the TTEthernet receive function receives the first PCF, it starts the timer for the sending function.

During initialization, RX buffers are also set up (by configuring buffer descriptors). Multiple buffers are needed to ensure that frames are not lost while the latest frame is still in use. The precise number of buffers depends on the system and schedule.

2) Receiving and Clock Synchronization: Frame reception is performed periodically based on the scheduled receive point in time. At each reception cycle, a function starts continuously polling the interrupt source register for a specified timeout duration, until the bit signifying that a frame has been received is set. This is done with a function called `tte_wait_for_frame`, which is also responsible for recording the receive time by reading the current cycle count. After a frame has been received and the receive time has been stored, we mark the buffer as empty and clear the interrupt source (implemented in the function `tte_clear_free_rx_buffer`). Afterwards, the `tte_receive` function (described below) is called.

The `tte_receive` function initially checks the type of the frame. If it is a PCF type, the integration frame is used to synchronize the local clock to the master clock. If the received

frame is not a PCF type, the function returns the received frame to the application.

For clock synchronization, the permanence point is calculated by adding the maximum delay and subtracting the transparent clock. For keeping track of when the controller expects synchronization masters to have sent the PCF, a variable called `start_time` is used. On receiving the very first PCF, this is set to $permanence_time - (2 \cdot max_delay + comp_delay)$. `start_time` is used to calculate the scheduled receive point, which is used to calculate the acceptance window. If the permanence point is outside the acceptance window, the `start_time` is reset to zero, and the function returns zero. In this way, the user program can immediately see that an error has occurred, and the controller returns to regular operation when it receives a correctly timed PCF once again.

If the permanence point is within the acceptance window, the difference between permanence point and scheduled receive point is added to the `start_time`, synchronizing the local clock to the master clock. The controller does not need to wait until after the acceptance window to synchronize, because the implementation only assumes one switch in the network, and thus only one PCF per integration cycle. Therefore, it is irrelevant whether or not the local clock goes back within the acceptance window.

3) Sending: Frames must be sent according to a predefined schedule, which requires some queuing mechanism, as the program should not be expected to calculate the exact send times itself. The `ethlib` send function expects outgoing frames to be stored in the RX/TX buffer and requires the specific address and size of the frame. One send queue is created per VL during initialization and is allowed to hold the maximum amount of frames that the VL can send in one cluster cycle, calculated as $\frac{clustercycle}{VLperiod}$. The send queues hold addresses and sizes of frames scheduled for sending. Each queue operates in a FIFO manner, keeping track of the head and tail through two variables.

The programmer has the responsibility to create frames in the RX/TX buffer according to its intended use by mean of the function `tte_prepare_header` to create the header of a TTEthernet frame. Frames are scheduled through the function `tte_schedule_send`, which takes the address and size of the frame and which VL it should be scheduled. The function then checks the queue of the VL and, if not full, schedules the frame for sending. The programmer shall not overwrite the buffer before the software stack sends the frame.

4) Generating the send schedule: An end-system should know the TTEthernet schedule running on the switch. By knowing the maximum frame size of a VL, the period, and the offset, it is the possible send times for each VL can be computed by repeatedly adding the period to the offset. A small algorithm is then used to combine these into a single schedule, generated on the fly at the startup time of the the end-system. This is explained through the example presented in Figure 7. To simplify the scheduling of the next timer interrupt, each entry in the schedule represents the time until the next interrupt. We represent time in tenths of ms.

	offset	period
VL0	2.6ms	4.0ms
VL1	1.0ms	2.0ms

VL.current

i	startTick	0	1	2	3	4	5
VL0	26	26	66	66	66	106	106
VL1	10	30	30	50	70	70	90

generated schedule

i	startTick	0	1	2	3	4	5
time	10	26-10 16	30-26 4	50-30 20	66-50 16	70-66 4	90-70 20
VL		1	0	1	1	0	1

Fig. 7. Example of schedule generation. Two VLs with the offset and period shown in the top left will generate the schedule in the bottom right when the cluster cycle is 8 ms. The top right shows the next send time of each VL at different steps in the algorithm.

The first event for each VL is at their offset, thus the very first event in the schedule can be found by finding the smallest offset. The starting offset is stored in a global variable `startTick` and the VL it belongs to in the first place in the schedule. For all VLs, a temporary variable named `current` is set to the offset of the VL. The VL that had the starting offset, adds its period to this value, which is the $VL.current$ value when i is 0 in Figure 7. We calculate the schedule time when $i = 0$ as the difference between the minimum current value (here 26) and the last current value (here 10). We store the VL with the smallest current value in the next place in the schedule (when $i = 1$), and we increment its current value by its period. We repeat these steps until the smallest current value is larger than the cluster-cycle (8 ms in this example).

C. Theoretical Limits of the Implementation

Because of the single-threaded nature of the implementation, the controller is characterized by certain limits, which we describe in the following three subsections.

1) Earliest outgoing TT frame: In the start of every cluster cycle, the transmission of frames is scheduled by the PCF handle function right after the clock has been corrected. Since the start of the cycle is defined as the point were the PCF frame is sent by the synchronization masters, scheduling a VL to send at 0 ms would cause the function to schedule a timer-tick in the past.

We do not know the exact receive times of PCFs at compile time, but we can assume that a PCF is permanent within the acceptance window, the latest possible receive time would be the same as the latest possible permanence time. Since the acceptance window is twice as wide as the precision, the latest receive time can be calculated with Equation 5.

$$rec_{latest} = 2 \cdot max_delay + comp_delay + precision \quad (5)$$

To ensure that the first outgoing TT frame is never scheduled too soon, it should be scheduled no earlier than the latest possible receive time plus the WCET of the `tte_receive` function. Figure 8 illustrates this timing relationship.

2) Maximum execution time after a TT frame: If the program has a long execution time, the reception of PCF might be delayed, negatively impacting the clock synchronization. Part

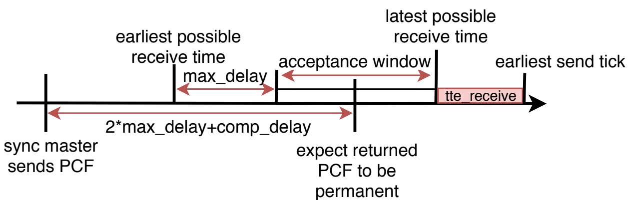

Fig. 8. The earliest possible receive time is max_delay before the start of the acceptance window. The latest possible receive time is at the end of the acceptance window. The first TT frame should be scheduled no earlier than the WCET of `tte_receive` after the latest possible receive time.

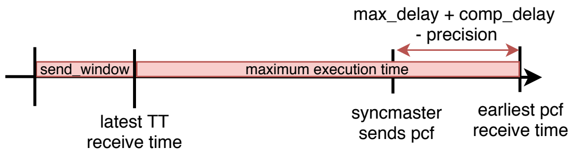

Fig. 9. The code executed after receiving a TT frame should take no longer than from the latest TT receive time until the earliest receive time of the next PCF.

of this could arise from the code executed after receiving a TT frame. The maximum allowed execution time after a TT frame depends on the TT frame scheduled with the smallest gap to the next integration frame. In our switch implementation, the *send_window* defined in the switch dictates the latest possible receive time in the node.

The earliest possible receive time of a PCF (assuming it is on schedule) would be if the actual transmission time was 0 and the frame was permanent as early as possible in the acceptance window. This is equivalent to *max_delay* before the acceptance window, as seen in Figure 8. All in all, the maximum execution time of the code executed on receiving a TT frame can be calculated with Equation 6, as illustrated in Figure 9.

$$\begin{aligned} max_{tt} =& start_time - ttrec_{latest} + max_delay \\ &+ comp_delay - precision \end{aligned} \quad (6)$$

3) Maximum execution time during integration cycle: Even if code executed upon receiving TT frames follow the limits described above, the reception of a PCF could still be delayed if the combined execution times of everything executed during an integration cycle exceeds the integration period. This limit can be expressed with Equation 7, where inc_{tt} is the number of received TT frames.

$$period > WCET_{int} + inc_{tt} \cdot WCET_{tt} + send_ticks \cdot WCET_{send} \quad (7)$$

D. Source Access

The TTEthernet controller and the relevant software are in open source and are available at https://github.com/t-crest/patmos. The software can be found at https://github.com/t-crest/patmos/tree/master/c/apps/tte-node.

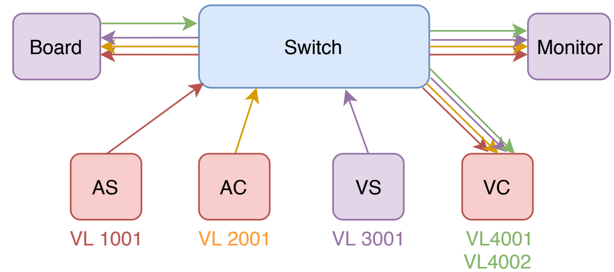

Fig. 10. Illustration of components and VLs in the system. Red nodes are synchronization masters; purple nodes are synchronization clients. This is also the physical setup used for tests presented in Section V.

V. Evaluation

A. System Setup

For implementation and testing, we used the star network configuration shown in Figure 10. It consists of a TTEthernet Chronos switch from TTTech Inc., four Linux end nodes (AS, AC, VS, and VC), a MS Windows node used for configuration and monitoring, and our TTEthernet node. Three of the Linux end nodes (AS, AC, and VC) act as synchronization masters. The fourth Linux node (VS) as well as our TTEthernet node act as synchronization clients.

The TTTech Chronos switch has 24 ports: six supporting Gigabit Ethernet and 18 supporting fast Ethernet (10/100 Mbit/s). We use 100 Mbits/s. The four Linix nodes are Dell precision T1700 PCs running Ubuntu. They are all equipped with a TTEthernet PCIe network card. The network card has two small form-factor pluggable ports that support connections with 100/1000 Mbit/s. The PCs execute a TTEthernet driver and API, as well as various test programs. The Windows PC does not contain any TTEthernet specific hardware. It runs the TTEthernet tools from TTTech Inc. that is used for configuring the TTEthernet system. In addition, the Windows PC is used to monitor the traffic on all VLs using Wireshark. The final node is our TTEthernet node, which is implemented on an Altera DE2-115 FPGA board using two Ethernet controllers: one for sending and one for receiving. All implemented hardware and software is open-source.

The TTEthernet tools are a TTTech development suite for configuring TTEthernet systems [43]. The tools include a GUI editor based on Eclipse [43]. The typical process for generating configuration files for all devices in a TTEthernet system can be seen in Figure 11. The first step is to create a network description file. It contains information about: senders, receivers, physical links, virtual links, and synchronization domain. The TTE-Plan tool uses this file to generate a network configuration file including the schedule for the entire network. For a specific schedule, we can also manually edit the network configuration file. From the network description and the network configuration files the tool TTE-Build generates the device-specific binary images that is loaded into the switch and the end nodes at initialization time.

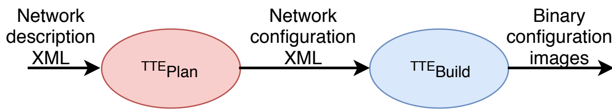

Fig. 11. Typical process using TTEthernet tools.

We have experimented with different schedules all implementing the same set of VLs, as shown in Figure 10. We tested the network at different integration periods and cluster cycles, but most experiments have used an integration period of 10 ms and a cluster cycle of 20 ms. The maximum possible transmission delay of any frame in the system was calculated with the TTEthernet tools and configured as 135600 ns. The compression delay of 10500 ns was used for collecting the results in Section V. We calculated the precision with the provided TTEthernet tools, and configured it as 10375 ns.

B. Clock Synchronization

The clock error is calculated as the difference between the scheduled receive point in time, and the actual permanence point in time at each received integration frame as specified by the TTEthernet standard SAE AS6802 [44]. To remove any interference of the `printf()` function, the error was first stored in an array and after a pre-specified amount of collected samples the clock error values were printed over a serial port. The clock error was measured in two different setups using different integration periods (synchronization cycles), a 10 ms period and a 100 ms period. Figure 12 presents a comparison of the measured clock error for the two integration periods. For an integration period of 100 ms the clock error ranges between 2875 ns and 3225 ns with a mean of 3055 ns, while for an integration period of 10 ms the clock error ranges between 2812 ns and 3112 ns with a mean of 2948 ns.

To reduce the measured error, we implemented a proportional/integral (PI) controller. The controller was manually tuned by first increasing the proportional part until there was a steady oscillation and then increasing the integral part until the systematic error was removed, this procedure led to the coefficient values $K_i = 0.3$ and $K_p = 0.7$. The results of the PI controller implementation are presented in Figure 13 and compared between the two different integration periods. When the control loop stabilizes, the clock error is just a few clock cycles with a mean of 126 ns for the integration period of 100 ms and a mean of 16.23 ns for the integration period of 10 ms. This is the best that can be achieved by taking timestamps in software in a tight loop. Similar methods for compensating the clock error have been investigated in [26]. The authors presented in simulation the use of a Least Squares Algorithm that managed to achieve 2000 ns offset. By applying a simple PI controller not only we reduce the complexity but we also measured significant increase in accuracy.

C. Latency and Jitter

To precisely measure latency and jitter of TT frames from the implemented controller, we used a physical setup similar to the one described in [2] and shown in Figure 10.

To perform the measurement, the test uses the two available Ethernet ports on the Altera board and sets up a schedule with a VL from one port to the other. Both ports are considered to be their own device by the schedule and are both synchronization clients. The second controller is accessed like the first, but uses a different address in local memory. This was not supported by the original ethlib, and was accommodated by duplicating several ethlib IO functions. A single VL with a period of 10 ms and an offset of 8.2 ms was used. This simplifies the test program, since one frame can be sent and received each integration cycle.

The test program follows the form described in Figure 6 with the first controller receiving frames in the overall loop. After a successful synchronization frame, a TT frame is scheduled, and the program waits until the frame is received by the second controller before proceeding. This enables the program to collect both the schedule and receive points of frames as the current clock cycle count. A slightly modified version of the sending interrupt function was used to measure the send point as current clock cycle count, making it possible to calculate latency and jitter. Both receive and send window in the switch where 263 μs wide.

Figure 14 shows latency measured as the average difference in send and receive time over 2000 measurements for 3 different frame sizes with various minimum switch delays. Lmin and Lmax have been calculated using Equations 1 and 2, disregarding propagation delay, and plotted alongside the values. All measured values are inside the expected minimum and maximum values.

The expected transmission times for frames of the measured sizes are 5.12 μs, 32 μs and 121.12 μs respectively, which means that the actual switch delay for these experiments must be approximately 200 μs higher than the minimum, judging by the trend-lines. This indicates that the switch receives the frames approximately 63 μs into the receive window in these tests. The jitter, measured as the smallest latency subtracted from the highest, did not vary significantly as a function of switch delay, but stayed between 4.5 us and 4.6 us throughout all experiments.

D. Worst-Case Execution Time

To enable the WCET analysis of the software, all loops need to be bounded. Therefore, we needed to perform some small modifications of our code. The function which converts the transparent clock from the format used in PCF to clock cycles initially performed division on an unsigned long. According to the analysis tool, the division function contains an unbounded loop. We replaced the division by an almost equivalent series of multiplication and bit-shifting to make the function analyzable. Additionally, we added pragmas containing loop bounds to all loops in the code, in order to aid the analysis tool.

We performed WCET analysis on significant parts of the controller using the platin WCET tool [14]. For the analysis, the board configuration was assumed to be the default for DE2-115.

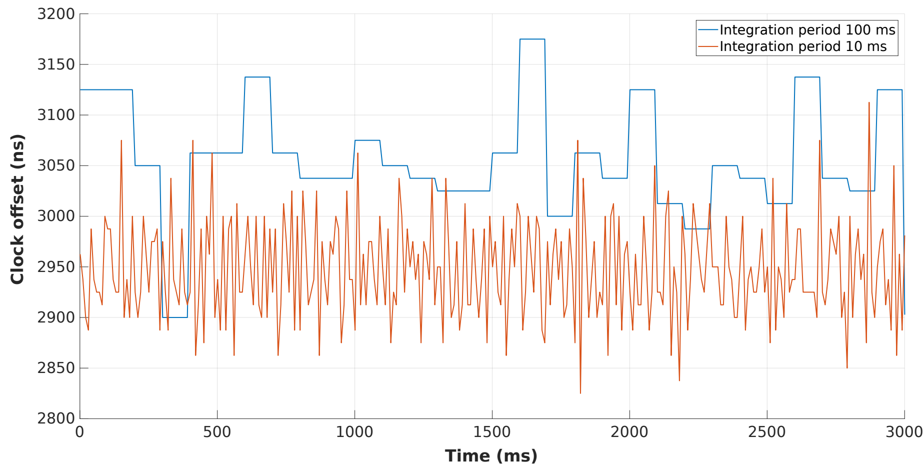

Fig. 12. Clock error comparison between two different integration periods.

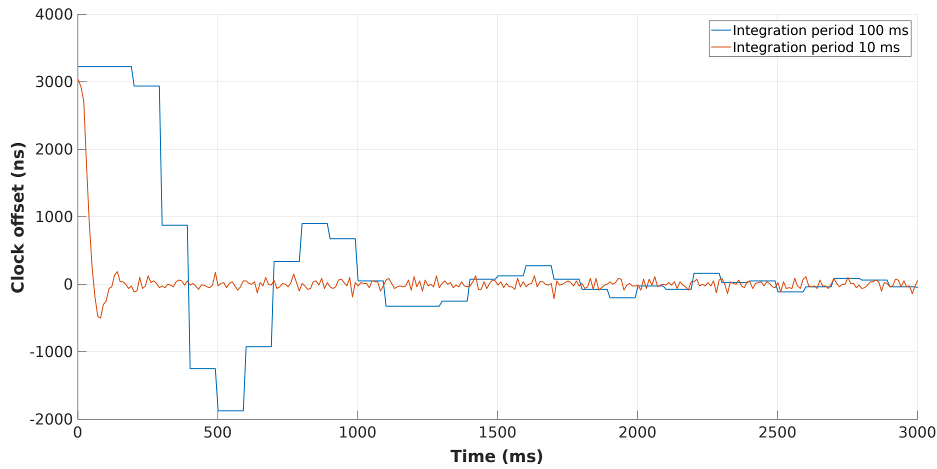

Fig. 13. Clock error with two different integration periods using a PI controller.

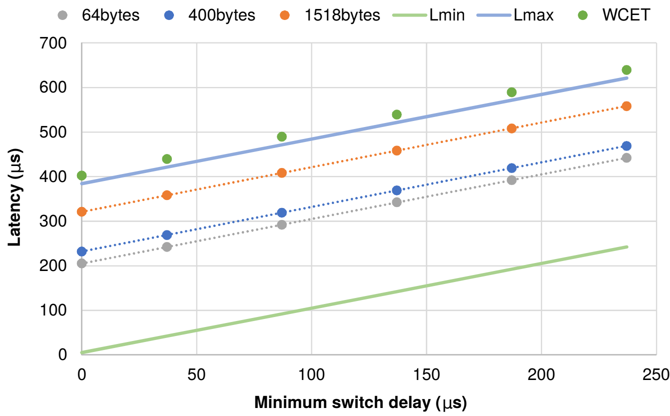

Fig. 14. Latency for various frame sizes as a function of minimum switch delay. The correlation is very strong, and well within expected minimum and maximum values.

We run the WCET analysis on a program resembling the demo program used with the regular implementation in Section V-B. In order to verify that the program and schedule satisfies the limits presented in Section IV-C, parts of the program have been moved into separate functions. Additionally, in order to analyze the timer interrupt function, it had to be explicitly called.

The tool requires that analyzed functions have the attribute `noinline` to prevent inlining. This makes the analysis slightly more pessimistic than necessary. The results can be seen in Table I, where all functions are part of the implemented `tte.c` library, except the final two, which are part of the tested demo program. The parentheses indicate WCET with PI implementation.

`tte_clear_free_rx_buffer` and `tte_receive` are mentioned in Section IV-B2. `tte_receive_log` is the TTEthernet receive function with logging enabled. `handle_integration_frame` and `handle_integration_frame_log` are called by the `tte_receive` function if the frame is an integration frame. `tte_prepare_test_data` creates a TT frame where the data part repeats a specified byte until the frame has a

TABLE I
WORST CASE EXECUTION TIME OF TTETHERNET SOFTWARE STACK FUNCTIONS.

Function	WCET (in clock cycles)
tte_clear_free_rx_buffer	10
tte_receive	3018 (3424)
tte_receive_log	3154 (3561)
handle_integration_frame	1454 (1860)
hande_integration_frame_log	1590 (1997)
tte_prepare_test_data	63149
tte_schedule_send	244
tte_send_data	306
tte_clock_tick	1721
tte_code_int	392419
tte_code_tt	40156

certain length. `tte_schedule_send` is described in section IV-B3. `tte_clock_tick` and `tte_clock_tick_log` are the timer interrupt functions with and without logging, and call `tte_send_data` when actually sending a frame. `tte_code_int` is executed after each successfully received integration frame, and `tte_code_tt` is executed after each received TT frame. Addition of logging adds about 150 clock cycles to the WCET. It is worth noting that the PI implementation adds an additional 400 clock cycles while using fixed point calculations.

E. Verifying Theoretical Limits of the Demo Program

With this example program and schedule, it is possible to verify that it satisfies the theoretical limits. The earliest outgoing TT frame in this example has an offset of 0.8 ms. Equation 8 presents the calculation of the earliest allowed transmission of a TT frame. It accounts for the equations presented in Section IV-C1, the system constants and the the WCET of the `tte_receive` function (the log version) as $WCET_{tte_rx}$. Since 332.3 μs is approximately 0.33 ms, the example follows this limit.

$$\begin{aligned} tt_{out} &= 2 \cdot max_delay + comp_delay + precision + WCET_{tte_rx} \\ &= 2 \cdot 135.6\mu s + 10.5\mu s + 10.4\mu s + 3216 cycles \cdot \frac{12.5\frac{ns}{cycle}}{1000\frac{ns}{\mu s}} \\ &= 332.3\mu s \end{aligned} \tag{8}$$

The TT frame which arrives closest to a PCF in this example arrives between 18.6 ms and 18.863 ms. Using this information, Equation 6 and the system constants, the maximum allowed execution time after TT frames is calculated with Equation 9. The WCET of `tte_code_tt` in this example can be seen in Table I. Since it is less than the 101,816 calculated cycles, the example program follows this limit.

$$\begin{aligned} max_{tt} &= sched_send - ttrec_{latest} + max_delay \\ &+ comp_delay - precision \\ &= 20,000\mu s - 18,863\mu s + 135.6\mu s + 10.5\mu s - 10.4\mu s \\ &= 1272.7\mu s \end{aligned} \tag{9}$$

$$\begin{aligned} max_{cc} &= 1272.7\mu s \cdot \frac{1000\frac{ns}{\mu s}}{12.5\frac{ns}{cycle}} \\ &= 101,816 cycles \end{aligned} \tag{10}$$

The example schedule has a maximum of 3 incoming TT frames in a single integration cycle. One of the VL can send a maximum of 3 outgoing TT frames, and the other a maximum of 5. An integration period of 10 ms is assumed, which is equivalent to 800,000 clock cycles. This information, Equation 7 and the WCET in Table I are used to verify the final limit in Equation 11 (in clock cycles (cc)).

$$\begin{aligned} &int_period > WCET_{int} + inc_{tt} \cdot WCET_{tt} + send_ticks \cdot WCET_{send} \\ &800,000cc > 392,419cc + 3 \cdot 40,156cc + 8 \cdot 1824cc \\ &800,000cc > 527,479cc \end{aligned} \tag{11}$$

F. Future Work

The presented time-predictable TTEthernet controller is a good basis for future work. We plan to re-implement the whole TCP/IP stack in a time-predictable version. We will avoid the blocking calls to read and write, as the usual implementation of sockets. We will use non-blocking functions that can be called from periodic tasks.

Furthermore, we are working on coordinating scheduling of tasks with scheduling of TTEthernet frames. With a tight coupling of time-triggered execution and time-triggered communication the end-to-end latency can be reduced.

Furthermore, we plan to add support of TSN to our node. Then we can directly compare TTEthernet with TSN.

VI. CONCLUSION

This paper presented a time-predictable TTEthernet end-system, built on top of the time-predictable Patmos processor. To the best of our knowledge this solution is the first TTEthernet end-system that can be analyzed for the worst-case execution time.

We evaluated the TTEthernet node in a test environment with one TTEthernet switch and six TTEthernet nodes that exchanged frames in various periods. The presented end-system is able to synchronize to the network clock with nanosecond precision by using a PI controller that significantly improved the synchronization error measured in previous work.

We performed WCET analysis of the main functions of the network code. This allowed to statically estimate the end-to-end latency of transmitted time-triggered frames and verify the expected maximum latency. Overall, this paper provides a solution for deterministic communication with TTEthernet and WCET analyzable tasks and network code on the Patmos platform. To the best of our knowledge, this is the first open-source TTEthernet node with a WCET analyzable network stack.

ACKNOWLEDGEMENT

This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation

REFERENCES

[1] Astrit Ademaj and Hermann Kopetz. Time-triggered ethernet and ieee 1588 clock synchronization. In *2007 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 41–43. IEEE, 2007.

[2] Florian Bartols, Till Steinbach, Franz Korf, and Thomas C. Schmidt. Performance analysis of time-triggered ether-networks using off-the-shelf-components. *Proceedings - 2011 14th Ieee International Symposium on Object/component/service-oriented Real-time Distributed Computing Workshops, Isorcw 2011*, pages 49–56, 2011.

[3] William Buchanan. CAN bus. *Computer Busses*, pages 333–343, 2000.

[4] R Courtney. Aircarft data network, part 7-avionics full duplex switched ethernet (afdx) network, 2004.

[5] Silviu S Craciunas, Ramon Serna Oliver, and TC AG. An overview of scheduling mechanisms for time-sensitive networks. *Proceedings of the Real-time summer school LÉcole dÉté Temps Réel (ETR)*, pages 1551–3203, 2017.

[6] Silviu S Craciunas, Ramon Serna Oliver, Martin Chmelík, and Wilfried Steiner. Scheduling real-time communication in ieee 802.1 qbv time sensitive networks. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 183–192. ACM, 2016.

[7] Rodney Cummings, Kai Richter, Rolf Ernst, Jonas Diemer, and Arkadeb Ghosal. Exploring use of ethernet for in-vehicle control applications: Afdx, ttethernet, ethercat, and avb. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 5(2012-01-0196):72–88, 2012.

[8] Peter Danielis, Jan Skodzik, Vlado Altmann, Eike Bjoern Schweissguth, Frank Golatowski, Dirk Timmermann, and Joerg Schacht. Survey on real-time communication via ethernet in industrial automation environments. *19th Ieee International Conference on Emerging Technologies and Factory Automation, Etfa 2014*, 2014.

[9] Sascha Einspieler, Benjamin Steinwender, and Wilfried Elmenreich. Integrating time-triggered and event-triggered traffic in a hard real-time system. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 122–128. IEEE, 2018.

[10] Janos Farkas, Lucia Lo Bello, and Craig Gunther. Time-sensitive networking standards. *IEEE Communications Standards Magazine*, 2(2):20–21, 2018.

[11] Thomas Fruhwirth, Wilfried Steiner, and Bernhard Stangl. TTEthernet sw-based end system for AUTOSAR. *2015 10th Ieee International Symposium on Industrial Embedded Systems, Sies 2015 - Proceedings*, pages 21–28, 2015.

[12] Voica Gavriluţ, Luxi Zhao, Michael L Raagaard, and Paul Pop. Avb-aware routing and scheduling of time-triggered traffic for tsn. *Ieee Access*, 6:75229–75243, 2018.

[13] Shiying He, Liansheng Huang, Jun Shen, Ge Gao, Guanghong Wang, Xiaojiao Chen, and Lili Zhu. Time synchronization network for east poloidal field power supply control system based on ieee 1588. *IEEE Transactions on Plasma Science*, 46(7):2680–2684, 2018.

[14] Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[15] IEEE. Time-Sensitive Networking (TSN) Task Group.

[16] H. Kopetz and G. Grünsteidl. TTP - A time-triggered protocol for fault-tolerant real-time systems. In Jean-Claude Laprie, editor, *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS '93)*, pages 524–533, Toulouse, France, June 1993. IEEE Computer Society Press.

[17] Hermann Kopetz. Temporal relations. In *Real-Time Systems*, pages 111–133. Springer, 2011.

[18] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (TTE) design. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 22–33, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Tamás Kovácsházy. Towards a quantization based accuracy and precision characterization of packet-based time synchronization. In *2016 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, pages 1–6. IEEE, 2016.

[20] Eleftherios Kyriakakis, Jens Sparsø, and Martin Schoeberl. Hardware assisted clock synchronization with the ieee 1588-2008 precision time protocol. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, pages 51–60. ACM, 2018.

[21] Eleftherios Kyriakakis, Jens Sparsø, and Martin Schoeberl. Hardware assisted clock synchronization with the ieee 1588-2008 precision time protocol. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 51–60. ACM, 2018.

[22] G Leen and D Heffernan. TTCAN: A New Time-Triggered Controller Area Network. *Microprocessors and Microsystems*, 26(2):77–94, 2002.

[23] Maciej Lipiński, Tomasz Włostowski, Javier Serrano, and Pablo Alvarez. White rabbit: A ptp application for robust sub-nanosecond synchronization. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication (ISPCS)*, pages 25–30. IEEE, 2011.

[24] Andrew T Loveless. On ttethernet for integrated fault-tolerant spacecraft networks. In *AIAA SPACE 2015 Conference and Exposition*, page 4526, 2015.

[25] Maja Lund, Luca Pezzarossa, Jens Sparsø, and Martin Schoeberl. A time-predictable ttethernet node. In *2019 IEEE 22nd International Symposium on Real-Time Computing (ISORC)*, pages 229–233, May 2019.

[26] D Macii, D Fontanelli, and D Petri. A master-slave synchronization model for enhanced servo clock design. In *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 1–6. IEEE, 2009.

[27] Igor Mohor. Ethernet ip core design document, 2002.

[28] Roman Obermaisser. *time-triggered communication*. CRC press, 2012.

[29] Maryam Pahlevan, Nadra Tabassam, and Roman Obermaisser. Heuristic list scheduler for time triggered traffic in time sensitive networks. *ACM Sigbed Review*, 16(1):15–20, 2019.

[30] Luca Pezzarossa, Jakob Kenn Toft, Jesper Lønbæk, and Russell Barnes. Implementation of an ethernet-based communication channel for the patmos processor, 2015.

[31] Miladin Sandić, Ivan Velikić, and Aleksandar Jakovljević. Calculation of number of integration cycles for systems synchronized using the as6802 standard. In *2017 Zooming Innovation in Consumer Electronics International Conference (ZINC)*, pages 54–55. IEEE, 2017.

[32] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[33] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. A multicore processor for time-critical applications. *IEEE Design Test*, 35:38–47, 2018.

[34] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[35] F Sethna, FH Ali, and E Stipidis. What lessons can controller area networks learn from flexray. In *2006 IEEE Vehicle Power and Propulsion Conference*, pages 1–4. IEEE, 2006.

[36] Wilfried Steiner. Advancements in dependable time-triggered communication. In Roman Obermaisser, Yunmook Nah, Peter Puschner, and Franz J. Rammig, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 57–66, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[37] Wilfried Steiner and Bruno Dutertre. The ttethernet synchronisation protocols and their formal verification. *International Journal of Critical Computer-Based Systems 17*, 4(3):280–300, 2013.

[38] Klaus Steinhammer and Astrit Ademaj. Hardware implementation of the time-triggered ethernet controller. In Achim Rettberg, Mauro Cesar

Zanella, Rainer Dömer, Andreas Gerstlauer, and Franz-Josef Rammig, editors, *Embedded System Design: Topics, Techniques and Trends, IFIP TC10 Working Conference: International Embedded Systems Symposium (IESS), May 30 - June 1, 2007, Irvine, CA, USA*, volume 231 of *IFIP Advances in Information and Communication Technology*, pages 325–338. Springer, 2007.

[39] Klaus Steinhammer, Petr Grillinger, Astrit Ademaj, and Hermann Kopetz. A time-triggered ethernet (TTE) switch. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 794–799, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[40] Ekarin Suethanuwong. Scheduling time-triggered traffic in ttethernet systems. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–4. IEEE, 2012.

[41] Domiţian Tămaş-Selicean and Paul Pop. Optimization of ttethernet networks to support best-effort traffic. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–4. IEEE, 2014.

[42] Domitian Tamas-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of communication schedules for ttethernet-based mixed-criticality systems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 473–482. ACM, 2012.

[43] TTTech. TTETools – TTEthernet Development Tools v4.4.

[44] TTTech. *AS6802: Time-Triggered Ethernet*. SAE International, 2011.

[45] TTTech. Deterministic ethernet & tsn: Automotive and industrial iot. *Industrial Ethernet Book*, 89, July 2015.

[46] Yingjing Zhang, Feng He, Guangshan Lu, and Huagang Xiong. Clock synchronization compensation of time-triggered ethernet based on least squares algorithm. In *2016 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*, pages 1–5. IEEE, 2016.

[47] Lin Zhao, Feng He, Ershuai Li, and Jun Lu. Comparison of time sensitive networking (tsn) and ttethernet. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–7. IEEE, 2018.

[48] Luxi Zhao, Paul Pop, and Silviu S Craciunas. Worst-case latency analysis for ieee 802.1 qbv time sensitive networks using network calculus. *Ieee Access*, 6:41803–41815, 2018.

[49] Luxi Zhao, Paul Pop, Qiao Li, Junyan Chen, and Huagang Xiong. Timing analysis of rate-constrained traffic in ttethernet using network calculus. *Real-Time Systems*, 53(2):254–287, 2017.

[50] Xuan Zhou, Feng He, and Tong Wang. Using network calculus on worst-case latency analysis for ttethernet in preemption transmission mode. In *2016 10th International Conference on Signal Processing and Communication Systems (ICSPCS)*, pages 1–8. IEEE, 2016.