embeddedworld2021 Exhibition&Conference ... it's a smarter world DIGITAL

Control-Flow Evaluation on Multicore Controllers using Real-Time Trace Information

Stephan Radke
Department of Electrical Engineering
Hochschule Darmstadt – University of Applied Sciences
Darmstadt, Germany

Thomas Barth
Department of Electrical Engineering
Hochschule Darmstadt – University of Applied Sciences
Darmstadt, Germany

Prof. Dr.-Ing. Peter Fromm
Department of Electrical Engineering
Darmstadt University of Applied Sciences
Darmstadt, Germany

***Abstract*— Most safety standards require an evaluation of the control flow behavior of safety functions to ensure consistency of design and implementation. Control flows, especially in the automotive sector, are often realized based on runnables that are triggered by a runtime environment (RTE). These trigger events can be cyclic as well as asynchronous. To avoid potential races and to ensure that the data flow meets timing requirements, the final system needs to be evaluated. This necessity becomes even more challenging for projects that run on multicore systems. For investigation of the runnable sequences, a possibility to capture time and event information of a running target with minimal impact to the timing behavior is required.**

Although existing tracing tools provide good visualization of e.g., OS events and calls, solutions covering additional aspects such as RTE events do hardly exist or need extensive customization.

In this paper we will discuss a method for generation, capturing and evaluation of real-time trace data on a multicore Infineon TriCore AURIX. An own tool reads trace data from various sources on the target via the debugger and passes the events to an external commercial trace tool with configurable association of events, filtering and event grouping. Using this approach, we will explain how the captured trace can be used to detect complex timing issues.

Keywords— Control-Flow; RTE; multicore; functional safety; tracing; timing analysis;

I. INTRODUCTION

A. General

Multicore microcontrollers offer increased computational power, allowing the implementation of complex applications, which makes them attractive for the use in advanced process- and control-systems e.g., in the areas of automotive, but also industrial and medical engineering. Not only the demand for computational power is an aspect that favors the use of multicore microcontrollers, but they are also becoming more and more attractive for safety critical applications. Multicore controllers can be used to implement a variety of functional safety architectures, whether it be multichannel or monitoring based safety architectures. But multicore architectures lead to several challenges regarding consistency of shared data and resources due to the truly concurrent software execution (see also [1]).

Safety related systems require a high degree of reliability against dangerous failures. It is therefore required that such systems undergo a well-defined and structured design and development process. One aspect of this process is the definition and validation of timing requirements which impact the control flow behavior of the system. They are defined and determined for different design levels. Timing requirements on system level describe aspects such as reaction or cycle time, latencies, or tolerable jitter for application-level safety functions. E.g., “After detecting a malfunction of a target speed sensor, the system must come to a halt within 500ms.” These constraints are then broken down into smaller units, leading to tolerable timing gates for individual runnables at the end of this process. On a multicore system, these runnables may be independently scheduled on different CPU cores and OS tasks.

The timing behavior of runnables affects the correct system function. Wrong timing may result in a variety stochastic and hard to debug error patterns:

- Race conditions of runnables may lead to actuator calls using old or incorrect data, resulting in wrong movements of actuators or other dangerous failures.

- Monitoring function evaluate wrong combinations of signals, which may be wrongly interpreted as error conditions (e.g., inconsistent state variables).
- Control algorithms receive sensor data, which has been sampled at different times, resulting in a wrong perception of the environment.
- If the execution time of runnables violate the provided budget, lower priority runnables may not be executed at all.
- Finally, the violation of timing budgets very often is an indicator, that other hidden bugs are still present in the system.

Most safety standards require the use of formal or semi-formal modelling methods when creating the system architecture. As part of the publicly funded ZIM project "safe and secure" a modelling notation for safety critical architectures running on multicore (and single core) controllers was developed [2]. The notation supports modeling of signal flows, timing constraints, safety requirements, abstract functional descriptions, software components (SW-C), hardware allocation and more. The resulting model is used for code generation of a runtime environment (RTE) that is responsible for executing runnables according to the modelled behavior. The control flow behavior is realized by different OS tasks. Fig. 1 shows the design process of an electronic throttle control system, where an abstract functional description is mapped to software components. These are associated with OS tasks, which are allocated to hardware (cores).

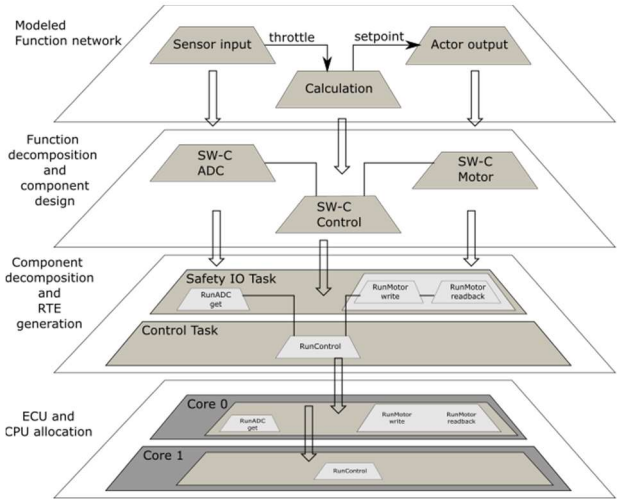

Fig. 1: Model-driven design process

Once the timing requirements have been defined, it is required to evaluate them on the running target. This is to validate that the defined timing constraints are met in the real existing system. The aspects to be evaluated include the start and end time of runnables, the sequences of the runnable execution and the timestamp of resources like signals which are used by the runnables. In addition, the timestamps of the operating system like task activations, semaphores and tick-objects are required for a deep analysis of the system. This evaluation can be performed with different methods including:

- analytical methods like code based static scheduling analysis,
- simulation, like scheduling simulation or Hardware-in-the-Loop simulation or finally by
- continuously measuring the timing behavior of each runnable and its associated events on the real target which is referred to as tracing.

In this document we will introduce a simple but flexible method for tracing runnables and data- as well as scheduling events, which is suitable for use in small and medium-sized companies. To provide a holistic view of the system timings and control flows, OS scheduling events shall also be included into the trace. This and the possibility to execute tasks and runnables on different cores requires an approach which allows capturing, preprocessing, merge, and visualization of several raw trace data sources.

B. System under analysis

The system under analysis is a software project based on a modelled and generated runtime environment (RTE) which is comparable to an AUTOSAR Classical Platform architecture. The process of modeling and code generation as well as the software architecture have been developed inhouse. It had been introduced in [3].

The code generator creates an entire C++ Runtime Environment, which consists of two main parts (see Fig. 2):

- A signal layer that is used to implement the data flow between the runnables.
- An activation engine controls the runtime behavior of the system. The execution order of runnables is defined by cyclic triggered timing events, (signal) data-update events and system state change triggers.

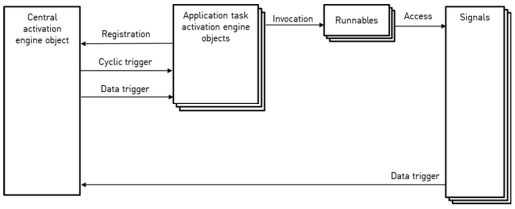

Fig. 2 Overview of the RTE and the Activation engines and the signal layer

The RTE and its activation engine tasks run on the real time operating system PXROS-HR which is designed for safety applications on multicore microcontroller systems (see also [4]). An Infineon TriCore AURIX TC297 multicore microcontroller serves as hardware platform.

The exemplary project used for evaluation is an electronic throttle control system (ETC). This system consists of three software components, containing 7 runnables (see also Fig. 3).

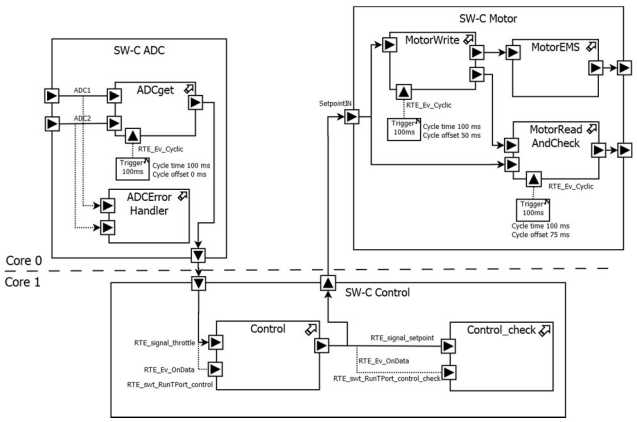

Fig. 3. Software components and Runnables of the ETC project

Five of these runnables are called during normal operation while the remaining two are used in case of error. The runnables of the software component "ADC" read the ADC values and feed them as a signal into the signal layer of the RTE. This data event triggers the runnables "Control" and "Control_check" of the software component "Control". In these runnables, control signals for the actuator are calculated from the ADC values. The "MotorWrite" runnable of the "Motor" software component uses this data to operate the motor with the control signal. The runnables with direct access to critical IO ports (ADC and Motor) run in the context of the "SafetyIO" task on CPU core 0. The part of the application that calculates the output variables form the input data (Control) runs in the context of the "Control" task on core 1, without access to hardware ports.

II. CATEGORIES OF TIMING PARAMETERS

In the following sections we will give an overview on how a trace solution is implemented. Before going into technical detail, the most important timing parameters are introduced.

For the analysis and evaluation timing parameters that are introduced in [5] can be divided into four categories.

A. Utilization

Utilization is the term used to describe the sum of work handled by a system. This can be the utilization of a CPU core or other shared resources like communication interfaces. It measures the amount of time a resource is occupied by one or a set of individual operations within a given time frame. If the accumulation of all occupations on a resource is higher than the provided capacity of the resource, it comes to an overload. This means the system will not be able to handle all occupations sufficiently, which will lead to unwanted behavior. For safety architectures it is essential to know how much utilization is generated by the safety functions as well as QM code.

B. Timing behavior

The term timing behavior summarizes all timely information of processes or operations.

First, the execution time represents the time an operation needs to complete on a system. It does not involve time required for handling interrupts or preemptions that might occur during execution. So, it only describes the time an operation needs to be executed, which depends only on the behavior of the operation and the capabilities of the hardware that runs it. Commonly known terms are the worst-case execution time (WCET) which represents the upper bound of uninterrupted execution time of a function or the best-case execution time (BCET) that stands for the lower bound.

The response time is the time that passes between start and end of an operation or process. It involves influences that block, interrupt or delay the execution of the operation. So, it includes the execution time plus the time portions of each interference during the execution. Known terms are the worst-case response time (WCRT) and best-case response time (BCRT).

The relation between execution times and response time is displayed in Fig. 4.

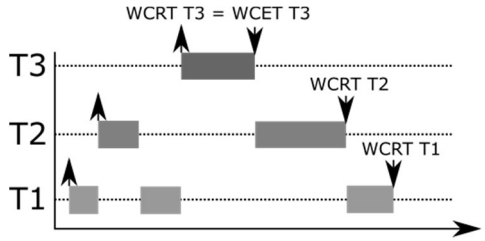

Fig. 4. Worst-case execution time (WCET) vs. Worst-case response time (WCRT), task T1 has the lowest priority, task T3 the highest

Another aspect is the end-to-end response time which represents the complete demand of time to execute functions on system level between a sensor input and an actuator output. It summarizes the timing budget of all executed runnables that are involved between sensor input and actuator output.

The Logical Execution Time (LET) defines the time between points of data interchange. So, the time of input read, and output write defines the LET which includes response time of the executed functions but also the time consumed for passing parameter and results through the underlying RTE (see also [6]).

C. Sequences

Real-time control systems are often implemented by a sequence of actions that take place in a defined order. This execution order may repeat periodically or may occur sporadically. Also, it is common that actions are chained together in a causal manner. These sequences of actions are also referred to as event chains, constituting the control flow of the application.

Misconfigurations or errors during implementation may lead to races of the event chain, causing unwanted or even dangerous behavior of the application. Therefore, the evaluation of the execution orders of runnables is an essential aspect of the system verification.

D. Execution context

Software components and runnables on multicore and multitasking systems are assigned to CPU cores and tasks to run them. As the allocation of software components is part of the generation process of the RTE it needs to be validated if the software components and runnables run on the desired CPU cores and in the contexts of the desired tasks.

III. TRACE TOOL CONCEPT

Existing solutions for tracing events on multicore systems are often quite expensive and require a lot of customization effort. This is especially true for solutions involving the collection and fusion of trace data from multiple sources (OS and Application). The solution we introduce is capable of configurable mapping of trace event data to data objects of a third-party trace tool, which allows to customize trace views and analyses.

Dealing with trace solutions on multicore targets requires further consideration of the interfaces used to transfer the trace data from the target to a PC performing the analysis. Due to the large amounts of data involved, interfaces with a high data rate are required. The required data rate depends on the number of traceable events that are to be captured within a time frame and the data size of each trace entry. Given a size of 32 Byte for each entry and a system with 100 runnables and a cycle time of 10 ms a resultant event rate of 10000 events/s will require an estimated needed data rate of about 1 MByte/s for three cores. As the trace buffers are read periodically a timely overlapping need to be budgeted as well to ensure each event can be captured. So, doubling the estimated data rate will provide enough headroom. Modern on-chip debug solutions like Infineons Cerberus with the Device Access Port interface (DAP) provide data block access rates of up to 15 Mbyte/s which will be sufficient (see [7]).

The operation of interfaces which must be served by the CPU or consume other timing critical resources represents a principally unwanted intervention in the timing behavior of the overall system. Furthermore, the complex MPU configuration of safety architectures very often is a limiting factor for existing trace libraries. For this reason, an interface is required which

- has full access to all data (without the need to reconfigure the MPU),
- provides sufficient bandwidth, and
- is minimal intrusive, i.e., while measuring, the timing characteristics are not influenced in an inacceptable extend.

For these reasons, it was decided to use the debugging interface (DAP) of the PLS debugger to extract the required data from the target to the PC and then pass it to an own application via the Microsoft COM interface. Advantages of this approach are that no additional communication library or explicit communication interface is required and that the timing characteristics are only influenced by the internal bus load.

IV. PRINCIPLE TO GENERATE AND EVALUATE TRACE DATA

A. Relevant measurements

In order to gather information about CPU utilization it is required to trace OS scheduling events. For this, the OS needs a trace interface or instrumentation must be added to the OS scheduler. Using this data, the active time of tasks and thus the CPU utilization and the scheduling overhead can be measured.

Additional information is needed to evaluate the timing behavior of the application, which is determined by runnable calls. Since runnables are called centralized by the activation engine, the timestamps of runnable activation and termination can be captured by adding timestamp measurement to the generated code of the activation engine/RTE. Based on this data runnable characteristics like the response times can be calculated. Additionally, the sequences of runnable calls can be comprehended due to the utilization of a system global time source for the timestamp.

Combining OS and application data, it can be evaluated that each runnable is allocated on the desired tasks according to the specification.

B. Principle of operation

As mentioned before, side effects that change the temporal and control flow behavior of the software project under analysis caused by the tracing tool must be avoided or at least minimized. Therefore, a tracing solution is required that has only little influence on the control flow of the running software.

Typical trace data transfer solutions using either the Ethernet, CAN or UART port often require complex additional communication stacks and architectural modifications, which may have an undesired impact on the overall behavior. To limit this interference, it was decided to use the debug interface of the target CPU. This solution provides access to variables and memories without implementing additional functionalities for communication interfaces and protocols.

After receiving the trace data from the target, a tool for visualization is required, allowing further investigation and evaluation. Often, these visualization tools provide predefined mappings of trace events to OS calls only and are limited to single core applications. As we want to visualize OS and application data running on several cores, flexible mapping is required.

Together with the company Percepio, an experimental interface of their trace tool Tracealyzer was evaluated and extended to allow the visualization of user defined trace records.

C. Dealing with the instrumentation overhead

Since there is virtually no trace solution without instrumentation overhead, it must be considered in advance how the instrumented functions of the RTE and the OS are to be handled after the end of the investigations and evaluations. Removing the trace functions on the target could result in a different timing behavior than what was evaluated using the trace solution. To ensure that the timing behavior does not change, it may be advisable to leave the trace instrumentation in the target's release version.

Depending on the capabilities of the debugger, the observed temporal behavior may also be changed by its influence. Reading data from the target with the help of the debugger usually causes load on the system bus which might influence the timing behavior of the application. Typically, this effect can be neglected but if the influence on the bus shall be avoided, trace data should be read only from halted targets or with a dedicated memory interface, depending on the controller.

D. Setup

The setup (see Fig. 5) to capture trace data form the target consists of a PLS UAD2 debugger together with the Universal Debug Engine (UDE) from PLS (see [7]). Since the UDE provides an Automation and Scripting support based on a Microsoft Component Object Model (COM) it is possible to implement a customized application to control the debugger. The UDE Automation library provides interfaces to execute memory accesses on the target as well as dedicated variable read and write expressions. It is also possible to retrieve symbol names of given types to display the named values of enumerator types.

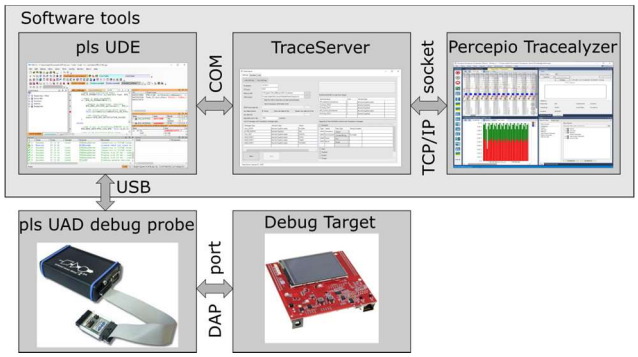

Fig. 5. Overview of the components of the setup

After capturing the trace data from the debugger, a tool is needed to display and investigate the data. Percipio Tracealyzer (see also [8]) provides a platform for visual trace diagnostics that can be accessed through a message-based socket interface, the so called TraceAPI. The TraceAPI messages can be formatted in JSON as well as XML and contain event information that represent task switches and their timestamps as well as user defined events that are linked to the currently displayed tasks.

In order to connect and access the debugger to receive the trace data and to connect to the Tracealyzer an own tool, named TraceServer, was implemented which captures the trace data and converts it into the Tracealyzer message format. The TraceServer provides the configuration of the variables and buffers to read from the target and a possibility to link these data to the Tracealyzer message object elements. After starting the trace, it captures the data from the target, converts it and sends the Tracealyzer messages periodically. The implementational aspects of the trace server are discussed in the following sections.

E. Trace instrumentation and data structures

To capture a complete set of data two categories of trace data need to be collected:

- The timestamps of scheduling points of the RTOS are required to understand the base temporal behavior of the system
- The runnable calls as well as the triggers activating the runnables are required to verify the application behavior.

With this information it is possible to investigate runnable executions regarding the task context in which they have been executed and the CPU core on which the task runs. Further investigation can use this data to evaluate the chains of sequentially called runnables.

Meta information is stored in a runnable trace buffer whenever a runnable is activated. This functionality is configurable and is integrated in the task activation engines as part of the modelling and automatic code generation process of the RTE. The following information is gathered for each runnable call:

- Runnable identifier
- System state in which the runnable was called
- Trigger (which Event-/Datasignal)
- Event (OnData, OnError, Cyclic)
- Start timestamp
- End timestamp
- Return value
- User added attributes

To have a common time base for the OS and Runnable trace data, a global timestamp source with a sufficient frequency/granularity depending on the application is required. In case of the introduced setup a system timer of the TriCore AURIX TC297 was chosen which runs with a frequency of 100 MHz providing a granularity of 10 nanoseconds.

Utilizing the PxROS trace interface, OS trace data is gathered for each task on each core. During OS initialization the PxROS trace data buffers are assigned and a trace hook function is activated. Every time the OS scheduling service is called the trace function writes an entry into the trace data buffer. For each entry, the following information is stored:

- Task identifier
- Event (e.g., task scheduling)
- Event specific arguments and error codes
- Timestamp

Runnable trace buffers are declared for each RTE task activation engine while OS trace buffers are declared for each CPU core. Therefore, the read process initiated from the trace server tool needs to read several buffers depending on how many task activation engines and OS tasks are implemented and are point of investigation.

F. Trace message object types

As different types of trace data buffers need to be captured to differentiate between OS task events and runnable or application layer events, it is also necessary to have different types of trace message objects that are to be sent to the Tracealyzer visual trace diagnostics tool. Tracealyzer provides three types of trace messages that are used for converting the captured trace buffer entries into the Tracealyzer live trace information:

1) Trace actor switch

Trace actor switch messages are used to trace events when actors start to execute. Actors can be tasks as well as interrupt service routines. This message combines the timestamp of the actor switch, the name of the actor that is started, and the CPU core on which the actor is running. It is also possible to define idle task names in order to trace the intervals when the CPU cores are in idle state.

2) Service call entry/exit

Service call messages are used to trace OS services like semaphores, messages or mutexes that put the calling tasks into blocking states (e.g., while pending on a semaphore). So, these messages are used to display intervals in which observable actions are performed. Also, each call of a runnable is such an observable action that can be traced by using this kind of messages. Additional to this, service call messages can contain additional fields, that are displayed as call parameters and return values. Since entry and exit of service calls are separate messages the interval of a service requires a start and end timestamp. These intervals also reflect the response times of the actions that have been traced.

3) User event

User event messages can be used to provide additional information to the trace. They contain a timestamp, a text field and a channel that is used to group user event messages together. In case of the exemplary project this trace message type could be used for return values of executed runnables as well as additional signal and trigger information.

G. Trace server operation

TraceServer supports configurable mapping of trace objects captured form the target to the trace message objects of Tracealyzer. Using this approach acquired trace data can be visualized with different views, depending on the current focus of debugging/evaluation. It also is possible to generate multiple trace message objects out of one captured trace entry, which is useful e.g., for runnable trace entries that consist of multiple atomic values.

Fig. 6 displays the steps the TraceServer executes to prepare and send the trace data to the visualization tool.

After starting a trace session, the trace server collects the content of the trace data buffers from the target periodically by using the pls UDE debug interface. All trace buffers contain timely overlapping trace data entries which must be filtered first. Then, each trace entry together with its properties is used to create one or more trace message objects.

These trace message objects may hold the named enum values of types that are used in the trace buffer entries such as runnable names. As the buffer read operation receives arrays of trace buffer entries which do not contain the named enum values it is required to retrieve the named enum values (strings) from the debugger with an additional operation. The PLS UDE Automation and Scripting support provides an interface to retrieve named values of known enum types. The debugger symbols list contains named enum values which can be determined by comparing the integer value of the trace buffer entry. For optimization, the named enum values are cached during a running live trace session.

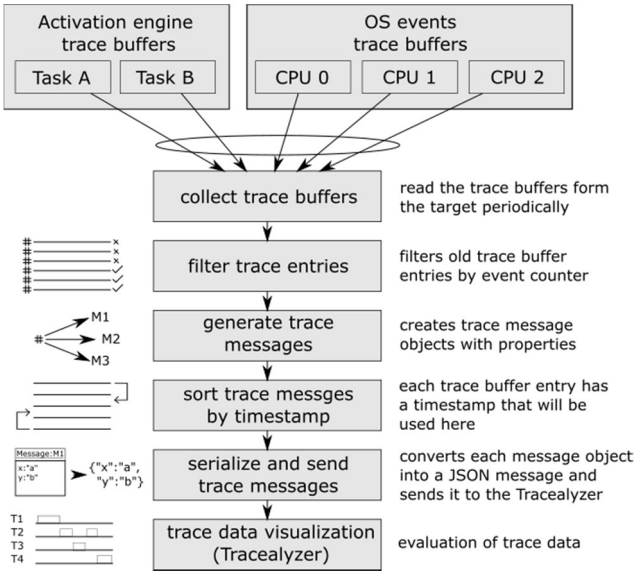

Fig. 6. TraceServer: Overview of the operational steps

After generating the trace message objects, they are sorted by timestamp and then serialized into JSON and send to the running Tracealyzer live trace session. The Tracealyzer then interprets, stores, and visualizes the trace messages.

V. INVESTIGATION OF TRACE DATA

In Tracealyzer the traced data can be visualized in different views. First, the trace view (see Fig. 7) can be used to display task activations on each CPU core together with event information that occurred during the task operations. In this view it can be analyzed which runnables are called in which task context and on which core. It also displays the start/end time of each runnable together with additional events information like return value, trigger event and the names of the associated signal.

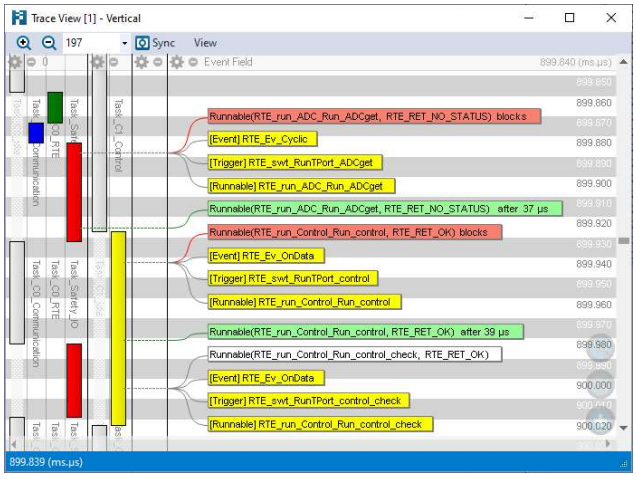

Fig. 7. Tracealyzer trace view

For the exemplary project, the runnable “ADCget” has been called in the context of the “SafetyIO” task on CPU core 0. The runnable was called by a cyclic event and had a response time of

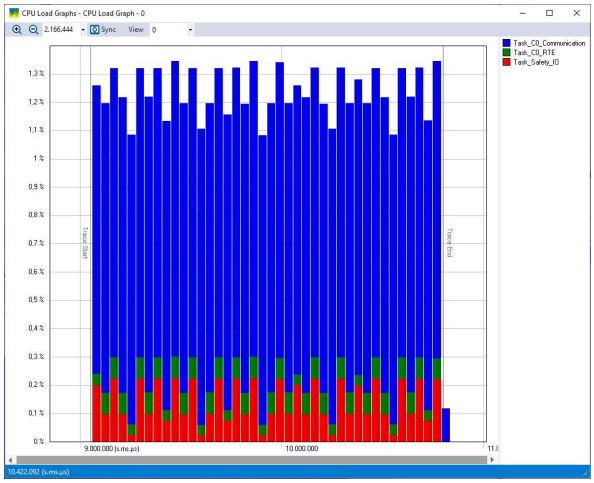

Fig. 8. Tracealyzer CPU load view

37 microseconds. Directly afterwards the runnable "Control" was called by an "OnData" event that activated this runnable on CPU core 1 in the context of the "Control" task. This runnable needed about 39 microseconds to finish.

This view also provides the possibility to estimate the latency time between finishing "ADCget" and start of "Control". This is the time the OS together with the RTE needed to process the interaction between the tasks "SafetyIO" and "Control".

Additionally, Tracealyzer generates CPU load graphs (see Fig. 8) that displays the CPU utilization of each core. These differentiate the tasks that causes the CPU utilization. So, it is possible to see which task may cause an overload. In case of the CPU utilization diagram of CPU core 0 of the ETC project it can be seen that the CPU is not really busy.

In order to investigate sequences of events and runnable calls a state machine graph can be generated out of the life trace data (see Fig. 9). It displays traced events as state intervals and brings them into a sequence of states based on the periodical appearance within the trace. This view can be used to find

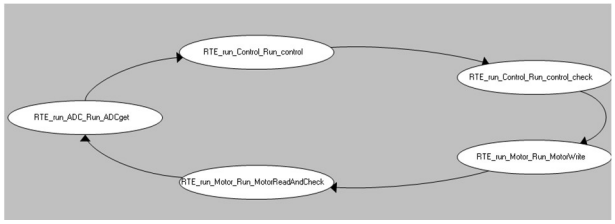

Fig. 9. Generated state machine graph, displaying a periodical sequence of called Runnables

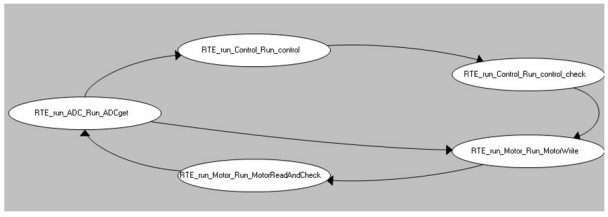

Fig. 10. Generated state machine graph, displaying a sequence of Runnables with failure in the execution order

unexpected transitions in the execution order which may happen when runnables are not called properly. Fig. 10 shows a failure of this kind. In the exemplary system of five runnables the runnables "Control" and "Control_check" which run on CPU core 1 have not been called in the appropriate order during at least one periodical cycle.

This can be interpreted as a race that needs further investigation. The race can be retraced in the trace view. This view provides an interval field that can be used to display application states. In case of the ETC project this field is used to show the intervals of the runnables that have been called (see Fig. 11). After finding the position where the failure in the runnable sequence occurred, it is possible to find nearby events or errors that might have led to this failure.

To extend user specific analyses scenarios, it is also possible to store the trace data of a live trace session into a JSON formatted file.

With the mentioned views in Tracealyzer and the possibility to implement customized tools for trace data evaluation, it becomes possible to find misconfigurations and failures in the execution of runnable sequences. It is also possible to provide statistics of execution and response times to validate if the timing requirements are met.

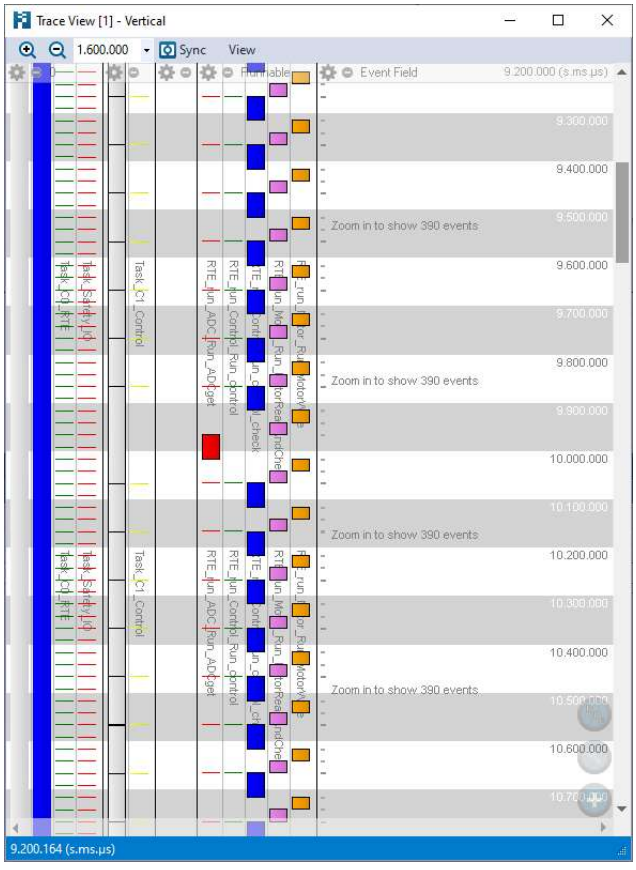

Fig. 11. Trace View interval field showing failure in Runnable call sequence

VI. Results and Conclusion

Although the implemented solution still is on a prototypic level, the concept of tracing data from the OS and the RTE on a multicore system and the flexible mapping of different trace events to a commercial trace tool has proven to be extremely helpful when it comes to analyzing the highly complex timing behavior of safety applications on multicore systems.

In the following table an overview is given which timing constraints can be evaluated with the current solution.

TABLE 1: COMPARISON OF EVALUABLE TIMING PARAMETERS

Timing parameter category	Value	suitable	Comment
Utilization	CPU load	Yes	CPU load graphs can be generated for multiple CPU cores
Timing behavior	Response time (e.g. WCRT)	Yes	Displayed in Trace view, as well as statistics
	Execution time (e.g. WCET)	No	More trace information needed from the target to subtract time for ISR
	LET	No	More trace information needed from the target (data read, data write events)
	End-to-End time	Partial	Can be seen in trace view, but no statistics available
Sequnces	Event chains	Partial	In trace view by adding intervals and generating state machine graphs
Execution context of Runnables	Runnable/ Task/ CPU core correlation	Yes	The correlation of CPU core, task and Runnables can be seen in the trace view

The introduced solution can be used to evaluate the basic timing parameters like CPU utilization and response times of processes that are called by the application. It is also possible to investigate the execution context of the applicational processes that will be useful to detect failures on systems that schedule processes dynamically over several cores and tasks.

The investigation of the control flow behavior that is represented by sequences of runnables or so-called event chains can be evaluated on the base of the traced data provided that the event chains are not complex. For investigation of more complex event chains the traced data need to be related to the modeled behavior. Here, future solutions analyzing the timing semantics of the architectural model and mapping those to the measured traces can be implemented.

Additionally, quantitative analyses can be implemented on the captured and stored trace data in order to allow statistical evaluations on this data base. These can be used to validate event chains as well as End-to-End times. In the current solution this option is limited.

Although the overall architecture is pretty complex: The combination of OS hooks with RTE data traces, using the debugger as a trace interface, and a dynamic translation and mapping of the data to the visualization records of a commercial tool has proven to be a good approach. It combines the flexibility when sampling the data with the features of the visualization tool.

References

[1] T. Barth und P. Fromm, „Functional Safety on Multicore Microcontrollers for Industrial Applications," in *embedded world Conference*, Nürnberg, 2016.

[2] P. Fromm und T. Barth, „A showcase for model based code generation for multicore safety systems," in *embedded world Confernece*, Nürnberg, 2021.

[3] T. Barth und P. Fromm, „Modeling and code generation for safety critical systems," in *embedded world Conference*, Nürnberg, 2020.

[4] HighTec EDV-Systeme GmbH, "Multi-core RTOS for TriCore," 2021. [Online]. Available: https://hightec-rt.com/en/products/real-time-os.html. [Accessed 20 Januar 2021].

[5] AUTOSAR, *Recommended Methods and Practices for Timeing Analysis and Design within the AUTOSAR Development Process,* vol. Document ID 645: AUTOSAR_TR_TimingAnalysis, AUTOSAR, 2019.

[6] C. M. Kirsch und A. Sokolova, „The Logical Execution Time Paradigm," in *Advances in Real-Time Systems*, Berlin, Heidelberg, Springer, 2012.

[7] Infineon Technologies AG, *AURIX™ TC29x B-Step 32-Bit Single-Chip Microcontroller,* Munich, Germany: Infineon Technologies AG, 2014.

[8] PLS Programmierbare Logik & Systeme GmbH, "Universal Debug Engine (UDE)," 2021. [Online]. Available: https://www.pls-mc.com/products/universal-debug-engine/. [Accessed 20 Januar 2021].

[9] Percepio AB, "Percepio tracealyze - Stop Guessing with Visual Trace Diagnostics," 2021. [Online]. Available: https://percepio.com/tracealyzer/. [Accessed 20 Januar 2021].