NITRO: Non-Intrusive Task Detection and Monitoring in Hard Real-Time Systems

Max Brand
max.brand@infineon.com
Infineon Technologies AG
Neubiberg, Germany

Albrecht Mayer
albrecht.mayer@infineon.com
Infineon Technologies AG
Neubiberg, Germany

Frank Slomka
frank.slomka@uni-ulm.de
Ulm University
Ulm, Germany

ABSTRACT

With the increasing complexity of real-time systems, developers need advanced tools for detailed analysis. Existing tools usually come with a high configuration effort which prevents their usage or leads to configuration errors. Further problems are contributed by analysis techniques that may introduce software overhead, which changes the system's behavior. Another major issue is, that modern real-time systems make use of purchasable software components, so-called *Intellectual Property* (IP) blocks. Since not all information is accessible on these IP blocks, they can lead to the development of opaque real-time systems, which are then even harder to analyze. With NITRO we want to present a new methodology to tackle these problems. It enables developers to analyze hard real-time systems without requiring detailed knowledge of the system. NITRO is capable of detecting and monitoring the complete task set of a hard real-time operating system, without error-prone user configuration. Designers neither need to understand how the operating system is implemented, nor how to configure a tool. Moreover, NITRO works completely non-intrusive and is therefore superior to instrumentation based analysis techniques. We also show that non-intrusive monitoring does not require expensive tracing hardware.

CCS CONCEPTS

• **Information systems → Data stream mining**; • **Software and its engineering → Dynamic analysis**; *Embedded software*; *Real-time systems software*; *Software maintenance tools*; • **Computer systems organization → Real-time operating systems**; *Real-time system architecture.*

KEYWORDS

real-time operating systems, real-time systems, tooling, real-time analysis, hardware tracing, monitoring, reverse engineering, AURIX

ACM Reference Format:
Max Brand, Albrecht Mayer, and Frank Slomka. 2020. NITRO: Non-Intrusive Task Detection and Monitoring in Hard Real-Time Systems. In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394810.3394812

RTNS 2020, June 9–10, 2020, Paris, France

ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394812

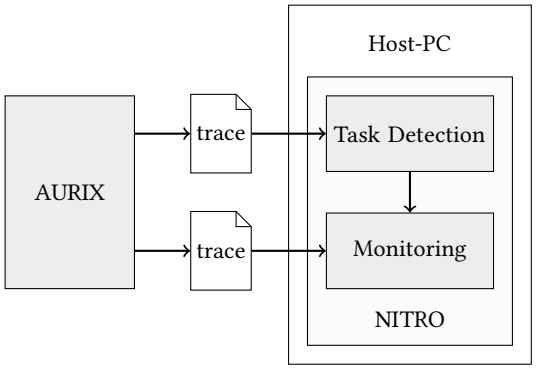


Figure 1: Overview of NITRO and the complete work-flow.

1 INTRODUCTION

Functional safety in the automotive domain is indispensable for some components like braking systems and therefore has to be fulfilled. Besides QM (*Quality Management*), a new classification was introduced in the ISO26262 [16] standard: the ***A**utomotive **S**afety **I**ntegrity **L**evel* (ASIL). It ranges from *ASIL A* as its lowest safety class to *ASIL D* with the highest safety requirements. The ASIL rating can be applied to a wide field within the automotive development, but we focus only on hard *real-time operating systems* (RTOS) with preemptive scheduling. One part of the functional safety assessment within an RTOS relies on the timing constraints of the running task set. For a functional analysis assisting the functional safety assessment, the important timing information of a task is the period p, *worst-case execution time* (WCET) c^+, and the jitter j as shown in Fig. 2.
One further parameter is the task's *worst-case response time* (WCRT) r^+, which is the complete duration between task release and completion, including periods where the task was preempted. We define the period p, jitter j, WCET c^+, and WCRT r^+ as the *task statistics.*
Obtaining the task statistics is called monitoring and comes with two main issues. The first and most critical issue is the intrusiveness.

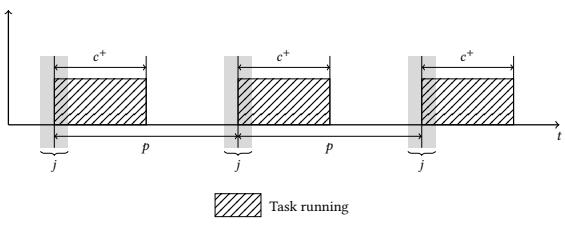


Figure 2: Commonly used task model and its parameters.

Usually, the task statistics are measured by instrumentation. For example, the instrumented RTOS hits different watchpoints implemented in software which are then used together with a timestamp to calculate the statistics. The downside of software instrumentation is that the system's behavior is altered by the instrumentation code, also known as the *probe effect* [11]. The second issue is the lack of a standardized monitoring interface. If every RTOS had the same monitoring interface, which can be used to retrieve the task statistics, the solution would be easy. However, this is not the case. Hence, every RTOS needs tailored software to enable monitoring. A general solution to solve this problem does not exist and therefore leads to errors made by the developers, who want to verify their task set.

Many modern *System on Chip* (SoC) devices come with a feature called hardware tracing. We can utilize hardware tracing along with trace data analysis to solve both problems. By tracing an SoC like the *Infineon AURIX*, a large number of trace messages can be acquired. We call these trace messages the *trace data*. Trace data contains information about the execution history of the SoC which includes for example executed instructions, jumps, calls, data operations and many more. Even though hardware tracing is very powerful to observe a system, it can be difficult to analyze due to a large number of trace messages. Because many trace messages are generated, the device-internal memory available for hardware tracing somehow limits the duration that can be traced. Device external tracing hardware enables longer tracing durations but is very expensive. Therefore, we want to avoid the use of such hardware, since not every developer has access to it.

With NITRO we want to show how trace data can be used to detect and monitor the running task set of an RTOS. Also, the methodology we propose works RTOS-independent and needs no error-prone user configuration. Moreover, with the RTOS independence we do not need a special interface for monitoring and hence enable it also in the post-development phase where not all information might be available. Especially when IP components are used for the development of a real-time system, necessary information may not be given to properly observe and validate the system. NITRO greatly improves the handling of such systems for developers. Other use cases for NITRO is the reverse engineering and debugging of legacy or unknown systems.

The general idea behind NITRO is that by nature every RTOS already has variables that can be used for the task activity analysis. An RTOS has to keep record of the currently running task and this has to be stored somewhere within the RTOS. We call this variable *Context Switch Descriptor* (CSD) and use it to distinguish between tasks and other RTOS related code to identify the task set. After the identification of the task set, the CSD is also used for the monitoring. Because we do not instrument the RTOS on our own and use hardware tracing to retrieve the needed information, NITRO is working completely non-intrusive. The work-flow of NITRO can be seen in Fig. 1.

To conclude our new methodology, we developed a proof of concept of NITRO and executed experiments based on ErikaOS [9, 10], which is OSEK/VDX compliant [22], FreeRTOS [1] and an AUTOSAR [4] implementation from *ARCCORE* (now *Vector Informatik*).

2 RELATED WORK

As far as we are aware, little research has been done on the topic of task monitoring within hard real-time systems as explained in this section. However, we could not find much research done on automatically detecting the tasks to enable ease of use, except the proposed approach by Iegorov *et al.* [13].

A commonly used approach today is the instrumentation of the RTOS or its tasks. An example of software instrumentation can be found in [6]. The advantage for the developers is the simplicity that comes along with software instrumentation. The downside is the probe effect [11] which may alter the system behavior while monitoring and therefore can lead to false results. A much more sophisticated example of code instrumentation is Feathertrace by Brandenburg *et al.* [5]. Feathertrace is very lightweight and has almost no impact on the timing behavior of the observed system.

Similar research compared to ours was done by Rufino *et al.* [24]. Their approach, called *NORTH*, focuses on runtime verification and observation of real-time systems. They advance the state-of-the-art of runtime verification by using temporal logic along with an FPGA for a completely non-intrusive device observation.

Further research on real-time monitoring was done by Decker *et al.* [7, 8]. The authors developed a tool, called *RETOM*, which utilizes *TeSSLa*, proposed by Leucker *et al.* [17]. The stream processing defined in TeSSLa is processed by an FPGA, which also provides sufficient processing power to enable an online analysis.

Both approaches are powerful in terms of monitoring and runtime verification. They have in common with NITRO, that they are non-intrusive. The downside is that they need expensive hardware as well as a complex configuration. NITRO has automatic detection of system information which is needed for monitoring while only utilizing cost-effective on-chip hardware for detection and monitoring.

As mentioned previously, a novel task mining approach was proposed by Iegorov *et al.* in [13]. They use system traces consisting of timestamps together with task identifiers to distinguish between periodic and aperiodic tasks appearing in the system under test. If a task is considered to be periodic, its response time and period are calculated. They developed a tool named PeTaMi and used it along with two real-world example traces to verify their approach. The downside is, that the task identifiers are needed beforehand. However, this approach could be used along with NITRO to further lower the needed bandwidth while monitoring as explained later in section 5.

For OSEK compliant operating systems, a debugging standard called ***O**SEK **R**un **T**ime **I**nterface* (ORTI) [23] was developed to enable debugging and monitoring. The OSEK shortcomings are the missing support of multi-core, the needed user interaction and the limited support by OS vendors. ORTI was used as the starting point for ARTI [3], which shall address AUTOSAR based RTOSes on multi-core devices.

In comparison to the above work, the advantage of NITRO is, that it is generic and not limited to one RTOS standard.

Other work on automatic and non-intrusive real-time system analysis was done by Li *et al.* [18, 19]. It shows the possibilities of hardware tracing based analysis.

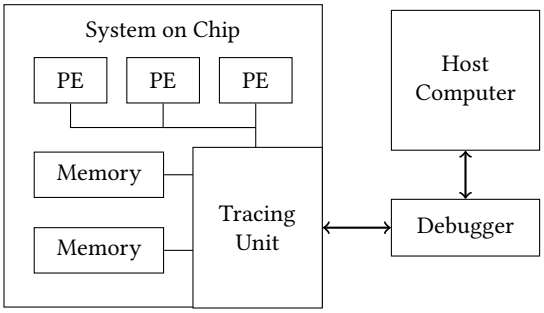


Figure 3: Typical setup for hardware tracing with the tracing unit attached to the memories and *Processing Elements* (PE).

3 HARDWARE TRACING

Before we explain NITRO in detail, we give a brief overview of the hardware tracing facilities of the Infineon AURIX and their utilization.

Many modern *System-On-Chip* (SoC) devices support hardware tracing. With hardware tracing it is possible to retrieve detailed information of the current state of an SoC and hence what the SoC is executing over time. The data we collect by tracing contains, for example, the current address of the *Instruction Pointer* (IP), address and data of memory accesses, taken branches, called functions, bus operations and many more. We call the collected data of the SoC the *trace data*. A typical setup for hardware tracing is depicted in Fig. 3.

Because modern SoCs not only provide hardware tracing but also operate at high speed, the amount of generated trace data per second by these devices is huge and so is the needed bandwidth of the tracing interface that is used to retrieve the trace data. This bandwidth can easily exceed several hundreds of megabytes per second which may be strongly limited by the used hardware and therefore can incur several issues. At first, due to the huge amount of data, it can hardly be used by the developer itself and hence has to be processed or at least pre-processed by a computer. If the amount of data is too high, even a high-end desktop computer can come into trouble while processing. As a second issue, the tracing interface itself may cause problems because its bandwidth is limited and hence not all trace data can be retrieved in real-time. The device has to be halted or we lose trace data (we call this loss a *gap* in the trace data), neither is desired.

There are two solutions we can take to not halt the system or lose trace data:

- increase the available tracing interface bandwidth
- lower the needed bandwidth

The first solution is easy to apply but is also expensive and may not be affordable for every developer because special debugging hardware is needed. This special hardware provides a high-speed interface to the device like the Xilinx *Aurora Gigabit Trace* [26] along with a huge external memory as trace data storage. Even though every tracing capable SoC comes with an internal buffer for trace data, this external memory is multiple times larger than the internal one. This solution is powerful but costly.

The second solution is to lower the amount of trace data generated and hence reduce the needed bandwidth. By lowering the needed bandwidth, we can make reasonable use of the SoC's internal trace data buffer. Since no special hardware is needed here, it is affordable for most developers but comes with limitations in terms of observability. However, we want to support a broad community of engineers and thus use the latter solution and show that our algorithm is applicable even with these limitations.

To reduce the amount of trace data being generated, we configure the tracing unit of the SoC to only record trace data that is needed by NITRO. Depending on the tracing unit that is used inside of the SoC, we can, for example, set different filters, triggers, and observation points as can be seen in the next subsection.

The advantage of hardware tracing is, that it is completely non-intrusive and hence does not alter the behavior of the system under test. NITRO only utilizes the non-intrusive hardware tracing interface and runs completely off-chip on a local machine.

Since our proof-of-concept is implemented for an Infineon AURIX [14], we want to give a brief introduction of the built-in tracing unit called *Multi-Core Debugging Solution* (MCDS). Since the AURIX is a multi-core SoC, the MCDS can observe different cores at the same time. However, we used only one core of the AURIX for our tests as can be seen in section 5. This introduction will cover the parts we utilize for NITRO. For more detailed information on the MCDS see [25]. It should be possible with only little effort to implement NITRO along with other tracing interfaces like ARM CoreSight [2] or Nexus [12].

3.1 Program-Trace

The *Program Trace Unit* (PTU) is one of the two main parts of the MCDS. It is connected to the different cores of the SoC and hence can observe the instructions that a specific core is executing. It provides three different modes for the program tracing, providing a different degree of detail. On the contrary, the more details the selected program tracing mode provides, the more trace data is generated and thus a higher bandwidth of the tracing interface is needed.

The first mode is the *Compact Function Trace*. It only generates a trace message if a function of a program is invoked or returns to the calling function. This mode provides little details but also generates only a few trace messages and hence is the most space-efficient way of program tracing.

As the second program trace mode, the *Flow Trace* can be used. It generates a trace message not only on function calls and returns but also on program discontinuities like `if`-statements or loops. This mode is a trade-off between program flow details and the memory consumption of the trace messages. As the last mode, the *Instruction Trace* is used as the opposite of the *Compact Function Trace* in terms of details and space efficiency. It generates a trace message for every instruction executed by the observed core. It provides the same details as the *Flow Trace* but with more information on the timing behavior since each trace message has its timestamp. The three different trace modes are depicted in Tab. 1-3.

The timestamps for each trace message can be relative, absolute or disabled. If the timestamps are disabled, the *Flow Trace* and

Instruction Trace are equal. Timestamps will be covered later in more detail.

3.2 Data-Trace

The *Data Trace Unit* (DTU) as the second main part of the MCDS is used to observe the data reads and writes of a specific core. The DTU can be set up in different ways depending on the needs and generates a trace message each time a data operation takes place. The DTU can generate messages at write and/or read operations containing only the address or the address and data value in combination.

3.3 Qualifiers

Each DTU and PTU comes with two qualifiers that act as a filter for trace messages. Each qualifier consists of two addresses (address range) and the qualifier type.
For the PTU, two main qualifier types exist: *in-range-qualifier* and *out-of-range-qualifier*. If the type is set to in-range, the PTU will only generate program trace messages if the address of the executed instruction is within the address range that is set for the qualifier. On the contrary, if the out-of-range type is used, the PTU will only generate program trace messages if the address of the executed instruction is not in the address range that is set for the qualifier.
The DTU also provides the in-range and out-of-range qualifier types. The address range that is set for the DTU qualifier is compared to the memory address of the data operation observed. If for example the in-range type is used and any data value is written to an address within the specified address range of the qualifier, the DTU will generate a data-trace message and vice versa for the out-of-range qualifier type.
In addition to these qualifier types, the DTU also provides a third type called the *program-qualifier*. If the type is set to program qualification, the DTU only generates data-trace messages as long as they appear within the address range specified for the PTU qualifier. If for example the PTU in-range qualifier uses the address range for the function `rnd_func()` and the qualifier type of the DTU is set to the program qualification, the DTU will only generate data-trace messages as long as the data operations take place within the range of `rnd_func()`. If the DTU uses this program qualifier, no address range has to be specified for the DTU qualifier since it is bound to the PTU qualifier. These qualifiers can be used to dramatically lower the amount of generated trace data and hence lower the needed trace memory and bandwidth.

Table 1: Example of the *Compact Function Trace* mode

Time	Operation	Source	Target
0 ns	IP CALL	sysInterrupt	tickHandler
6 ns	IP CALL	tickHandler	schedAndDispatch
70 ns	IP RET	schedAndDispatch	tickHandler
71 ns	IP RET	tickHandler	sysInterrupt

Table 2: Example of the *Flow Trace* mode

Time	Operation
	MOV.AA A15, A4 LD.A A4, [A4], 0x14 LD.W D4, [A15], 0x18
9 ns	CALL 0xFFFFFFB8
	JNZ D4, 0xC LD.W D2, [A4], 0x3C EXTR.U D2, D2, 0x1, 0x1
21 ns	RET

Table 3: Example of the *Instruction Trace* mode

Time	Operation
8 ns	MOV.AA A15, A4
	LD.A A4, [A4], 0x14 LD.W D4, [A15], 0x18
9 ns	CALL 0xFFFFFFB8
11 ns	JNZ D4, 0xC
	LD.W D2, [A4], 0x3C
20 ns	EXTR.U D2, D2, 0x1, 0x1
21 ns	RET

3.4 Triggers

The MCDS comes with a small internal memory that is used for trace recording. Usually, if trace recording is started, all the generated trace data is stored in this small memory until it is full and the trace recording automatically stops. In general, this is fine to catch a snapshot of the system under test, but in some cases, we are interested to observe the execution of a specific function or want to see what is happening before or after a specific function. In this case, it is cumbersome to trace again and again until we get the desired trace data snapshot of the system.
To cope with this, the MCDS comes with certain trigger mechanisms. If a trigger is used and trace recording is started, the internal trace memory is used as a ring buffer. The generated trace data is stored in this ring buffer while older trace data in it is overwritten. As soon as a trigger is hit, the trace is ongoing until a certain trigger position is reached and then the trace recording is stopped.
The trigger position is the position of the trace message in the recorded trace data that has fired the trigger. In general, 5 trigger positions can be used: *start*, *30%*, *60%*, *90%* and *40 bytes before end*. If for example the trigger position is set to *start*, the trace recording will continue after the trigger was hit until the complete ring buffer is overwritten once. The trace message that fired the trigger is now at the *start* of the recorded trace data. If the trigger position is set to *40 bytes before end* and the trigger is hit, the trace recording will stop after 40 additional bytes of recorded trace data. The triggering trace message is now almost at the end of the recorded trace data.
To fire a trigger, the DTU, as well as the PTU, can be used along with a specified address range. This address range can be set up like the address range for the qualifiers and with the same types:

in-range and *out-of-range*. If for example the in-range trigger of the PTU is set to the function `rnd_func()`, the trigger is fired as soon as the *Instruction Pointer* (IP) of the observed core hits the address range of `rnd_func()`. On the DTU the trigger is fired as soon as a data operation takes place within the specified address range of the trigger. The DTU trigger can also be bound to a certain data value that is read or written. In that way, the trigger can be for example fired if a specific data value is written to a specific memory address.

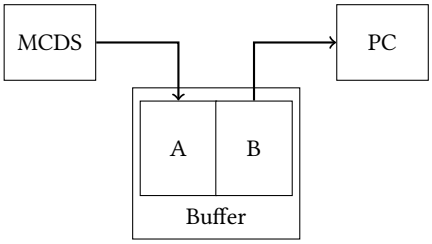
3.5 Timestamps

The last important setting of the MCDS to cover is the timestamp setting. The timestamp settings look trivial in the first place but can make all the difference for the trace configuration. The timestamps generated by the MCDS can be one of two types: a *tick message* or a *timestamp message*.

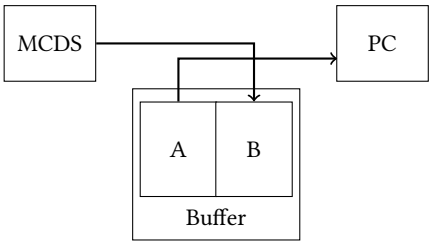
Usually, only tick messages are generated and attached to the different trace messages generated by the MCDS. In that way, we get the information for every trace message, how many ticks passed since the last message arrived. These messages can be summed up for absolute time information instead of only relative time information. The downside here is, that even if there are no trace messages generated due to the qualifier settings, the MCDS still generates tick messages so we do not lose any information regarding the absolute time that has passed. Hence, if we use tick messages for the timing information, the MCDS will generate tick messages during the complete trace recording. Even though the tick messages consist only of a few bytes, they can consume most of the trace memory available, which dramatically limits the overall recording time and increases the needed bandwidth.

The timestamp messages in comparison to the tick messages contain the complete absolute time and hence do not need to be generated continuously. The timestamp messages consist of more bytes and would consume the trace memory even faster if many trace messages are generated, therefore usually tick messages are used. However, the advantage of the timestamps is, that we can also bind them on the triggers used for trace recording control. We can, for example, disable the time information and just let generate an absolute timestamp if a certain value is written to a specific memory address. This can be a good compromise between time information and trace memory consumption as we will see later in section 5.

With these settings we can control our tracing unit (MCDS) in a sophisticated way: *we only trace what we need*. This helps to dramatically reduce the needed internal trace memory and hence we can trace for a much longer time. Moreover, since we lower the bandwidth of the trace data that is generated, we can utilize the internal trace buffer as a double buffer to get continuous traces. The *Device Access Server* (DAS, [15]) interface we are using to control and retrieve the trace data can handle almost up to 2 MB/s of bandwidth to the host computer. If the generated trace data is below this bandwidth, we can fill half of the internal buffer with trace data while the other half of the internal buffer is read and its trace data is transferred to the host computer. If all data is transmitted, the buffer is swapped and the other half is filled with the trace data while the other half is again read. This can be done continuously to get a continuous trace without expensive tool hardware as depicted in Fig. 4. If the bandwidth of the generated trace data exceeds the maximum bandwidth of the DAS interface, we can either suspend the device or suspend the tracing. The first would be intrusive and hence we do not want to suspend the device. The second solution will give us a trace data stream that contains gaps. Since we get a dedicated trace message for each gap, we can also correctly handle the trace data so we will not face any problems due to gaps. Moreover, with the settings that we will use later for the task monitoring, we decrease the needed bandwidth down to a point where no gaps appear.



(a) Buffer A is filled while buffer B is emptied.



(b) Buffer B is filled while buffer A is emptied.

Figure 4: The two alternating states of the trace buffer usage for continuous tracing.

As this section has covered the basics of hardware tracing as well as a brief introduction to the Infineon AURIX tracing interface, the next section will proceed with the general idea of NITRO and its methodology.

4 METHODOLOGY

NITRO is based on the assumption that every RTOS comes with variables that are mandatory for the RTOS to manage and maintain the different tasks. There can be different redundant variables, e.g. explicit ones with a number representing the active task and implicit ones like a pointer to the control structure or the stack storage of the active task. Any of these variables can be used for NITRO, the only prerequisite is the biunique relationship with the active task. As mentioned before, we call the detected variable the *Context Switch Descriptor* (CSD).

To find a CSD, NITRO executes two phases. The first phase performs one long trace (up to several seconds) of the system to get a list of all *software functions* (SWF) that are executed. This phase is called the *probing phase* and it also builds the *Dynamic Call Graph* (DCG). With the DCG, certain SWFs are selected that could be tasks. The second phase (*CSD phase*) is then to identify a valid CSD by performing multiple short traces (in the range of milliseconds). The

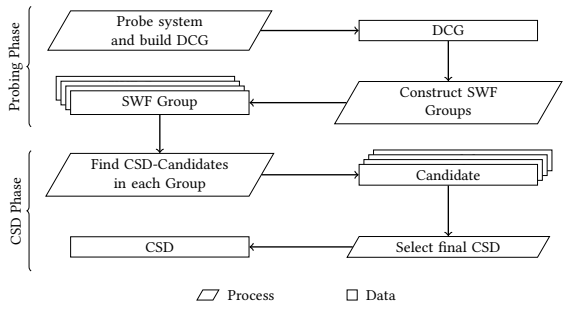


Figure 5: Workflow of NITRO.

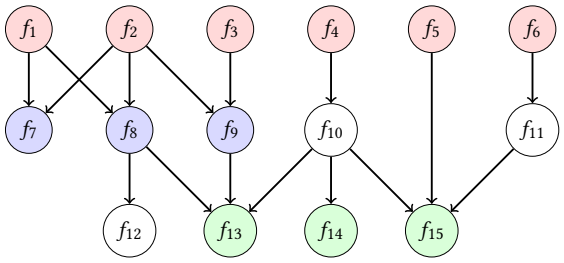


Figure 6: A dynamic call graph with the functions f_1 to f_{15} and three possible task sets (red, blue and green).

complete workflow is depicted in Fig. 5 and is explained next.

4.1 Probing Phase

To find the task set, we first construct different groups of SWFs of which one group or a subset of a group may be the system's task set. We will use the DCG, built upon hardware trace data, to construct these groups. The DCG is a directed graph describing the relation between different SWFs. It is used to determine the functions invoked by a given function. The DCG is called dynamic because it is built by observing the system with hardware tracing and not by a static analysis of its executable.

To build the DCG we trace function calls and returns on the device. Since we do not need any timing information for this, we disable the timestamps of the tracing unit. With these settings, we achieve a comparatively low trace data bandwidth and hence can perform a long continuous trace without expensive tracing hardware. To reduce the overall runtime of NITRO we have chosen a recording time of five seconds since we were running numerous tests to have sound experiments. However, this recording time can easily be raised to minutes to ensure that all tasks are covered in the recorded trace data of a real system (as can be seen later in section 5.2). After the trace data has been acquired, we can build the DCG since we get a trace message for each function call appearing on the system. The DCG is constructed by creating a node for each function appearing in the trace data and creating a directed edge from function f to f' where f invokes f', visible by a call instruction in the trace data. An example of a DCG can be seen in Fig. 6.

After constructing the DCG, we use it to compose groups of SWFs which could be the task set of the system. One group is always constructed by grouping all functions that have no parent nodes (red nodes in Fig. 6). This group is constructed since it might be the case that the tasks are not directly called and hence may not appear as called in the trace data. Further groups are composed by looking at the direct children of each node. Since it is most likely that all tasks are directly called by the same RTOS related function (e.g.: a dummy function within the dispatcher), the children of each node are considered to be the task set as long as the number of children is above or equal to a predefined threshold. We used a threshold of three since it is unlikely that an RTOS is used along with only two tasks. Lowering the threshold increases the runtime of NITRO since more groups are constructed and tested. In Fig. 6, the three SWF groups which pass these rules are colored.

All three operating systems we have tested fulfilled these assumptions (see section 5), even though the scheduler and dispatcher were also called sporadically by a task termination function.

4.2 CSD Phase

4.2.1 Collecting the Trace Data. The least minimum of functionality an RTOS needs is the scheduling and dispatching of tasks. Usually, an RTOS has a system tick which runs periodically. This system tick runs the scheduler to decide which task is executed next or if the current task can continue its execution. After the scheduler made its decision, the dispatcher performs a context switch if necessary. At a context switch, the dispatcher saves the context of the current task to the memory and then initializes or restores the context of the next scheduled task. The context of a task usually consists of RTOS related metadata and hence we will most likely find a CSD in the scheduling or dispatching function, which makes use of the context. Since the scheduler and dispatcher have to be executed beforehand, we are interested in the code that is executed right before a task starts or continues execution. In consideration that an ISR is the only way to interrupt the system, we assume that the scheduler and dispatcher are executed (or at least started) within an ISR. To get this particular trace data appearing before the investigated SWF executes, we set up a memory-efficient trace configuration as follows:

We will only trace data operations together with their timestamp. Instead of a continuous trace, we use a trigger that is set to the address range of the investigated SWF. The internal trace memory is used as a circular buffer, while the trigger ensures that the trace is stopped as soon as code within the investigated SWF is executed. If we start the trace with these settings, we retrieve trace data that contains all write operations right before the assumed task's execution. The hardware tracing also provides information regarding the start and end of ISRs. An example of a trace with these settings can be seen in Tab. 4.

Given Tab. 4 as the recorded trace data, NITRO iterates over every write operation from `Trig CPU IP` at time 0 up to `ISR_START` at time −240 (this timestamp varies for different traces). While iterating, every write operation where its address is the first appearance in the trace data (closest to the trigger), is stored. Every write operation with an address that was already observed and stored in this single trace, is ignored. At last, we summarize the resulting write operations of this single trace and call it a fragment of the SWF assumed to be a task. We will trace multiple of these fragments for each SWF of a group.

To recapitulate this briefly, NITRO records trace data multiple times to acquire multiple trace fragments for each SWF of each group. If we, for example, investigate a group with a size of six and collect three fragments for each SWF, we get $6 \cdot 3 = 18$ different trace fragments by tracing the system 18 times. It is important to know that the different groups are handled independently. The algorithm to identify a valid CSD is only used on the trace fragments of exactly one group as explained next.

4.2.2 Finding the CSD. As we have retrieved the trace fragments for all SWFs of a group, NITRO uses these to identify a CSD. For the identification, every write operation in every fragment is compared to all other write operations of the other fragments. Every address that is observed in the fragments is tested if it fulfills certain properties to be a valid CSD:

(1) The address appears in every fragment of the investigated SWF
(2) The data value written to this address,
   (a) is the same in all fragments as long as the fragments belong to the same SWF
   (b) is uniquely assigned to one SWF and hence does not appear in any fragment of another SWF

If an address fulfills all properties, it is a CSD candidate. The address of each candidate is stored as well as the data values that were written in front of each SWF. Since it might be the case that not all SWFs of the investigated group are tasks, the number of SWFs, where the address was found within the fragments, is stored as the candidate's coverage. As the last parameter, the time distance between the candidate's write operation and the actual SWF execution is stored. We call this the candidate's time-value.
After all fragments of all SWFs within each group were processed, it is most likely to retrieve multiple candidates that could be used as the CSD, of which we have to choose one as the final CSD. This selection is based on the coverage of the candidate. The candidate with the highest coverage is used as a CSD since it is very unlikely that a non-RTOS related variable with a high coverage fulfills all CSD properties. If multiple candidates share the same highest coverage, we could select any of these as the final CSD. In this case, we select the CSD with the lowest time-value without a specific reason.

Table 4: Example of a trace fragment. The write operations to identify a suitable CSD (yellow) are found between the start and end of the fragment (cyan).

Rel. Time	Data	Operation	Address	Symbol / Label
−240	0B88	STATE		ISR_START
−186	A9617FF3	W32	70000240	Timer
−160	0001BE87	W32	70000068	xTickCount
−141	700000D4	W32	700008B4	ucHeap
−116	7000020C	W32	7000034C	ucHeap
−115	00000001	W32	7000020C	pxReadyTasksList
−83	0B08	STATE		
−66	00870EA0	W32	700006D8	ucHeap
−39	70000338	W32	70000220	pxCurrentTCB
−38	0000000F	W32	70000064	uxTopReadyPriority
−5	0300	STATE		ISR_END
0		Trigger		

4.3 False-Positive Classification of Tasks

Since the classification of the tasks relies on assumptions made upon the RTOS, there is still a chance that a group of SWFs could be classified as a task set even though it is not a task set – a false-positive classification. However, to classify a group of SWFs as a task set, they have to fulfill three criteria according to phase one and two, explained previously:

(1) All SWFs are called by the same SWF or have no parent SWF
(2) An ISR takes place before the respective SWFs are called
(3) A suitable CSD is written prior to the SWF execution

With these criteria in mind, the respective SWF of a group of SWFs needs to be started/executed by either an ISR or within a true-positive task of the task set.
A true-positive task most likely calls several other SWFs while being executed. However, to fulfill the criterion of a suitable CSD, the true-positive task has to maintain such a variable, which is not very likely. The only rational use-case we could imagine is the use of a state-machine within a true-positive task as shown in Listing 1.

Listing 1: State-machine within a task causing a false-positive classification.

```
void task() {
  x = next_state();
  switch (x) {
    case 1: func_1();
            break;
    case 2: func_2();
            break;
    case 3: func_3();
            break;
  }
}
```

However, a false-positive classification of such a state-machine comes with two further limitations:

(1) The state-machine variable x has to be written before the switch-case statement, otherwise, the x cannot be identified as a suitable CSD
(2) The number of SWFs used within the state-machine needs to be equal or higher than the real task-set

Assuming that all conditions are fulfilled, we would still retrieve two SWFs groups classified as task sets: the true-positive and false-positive task set. If the state-machine, classified as the false-positive task set, is executed within a true-positive task, the false-positive task set can easily be dropped since we can identify a *parent task set* executing the false-positive task set. The identified parent task set is then classified as the true-positive task set. We prepared several task sets fulfilling these criteria to prove that this false-positive classification can be solved.
However, for the case that such a state-machine is implemented aside from the RTOS in another ISR, we found no solution yet to solve this. But since such an implementation would be somehow a

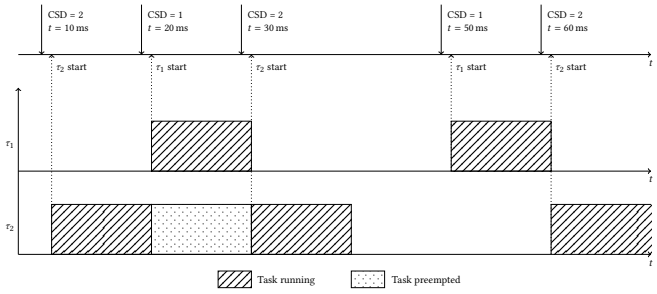


Figure 7: The trace data from a time axis perspective and the corresponding result of the task instances.

contradiction to the usage of an RTOS, we do not expect that this case occurs on real systems.

5 EXPERIMENTS

We developed three different experiments to demonstrate the accuracy and correctness of NITRO. The first experiment is based on a simple but highly reproducible task code to show the accuracy that is achieved by the exploitation of RTOS mechanisms, as NITRO does. The second experiment is similar to the first but focuses on the task set identification on systems where complex code is used within the tasks. This second experiment is then followed by the third and last experiment which covers the corner case that can cause false-positives as discussed in section 4.3.
For all experiments, we set up an automatic test environment based on Java on the tool PC side along with FreeRTOS [1], ErikaOS [9] and an AUTOSAR [4] based OS. All three operating systems execute their task sets as rate monotonic scheduled and fully preemptive.

5.1 Accuracy Experiment

With the first experiment, we want to show the accuracy of the RTOS exploitation NITRO uses. This experiment executes a number of tests with randomized task sets. These task sets are then identified by NITRO and subsequently monitored for one minute. The monitoring is done by non-intrusive hardware tracing and explained next. The monitoring is then followed by the test generation and evaluation. Finally, we discuss the measurement results and their accuracy.

5.1.1 Monitoring. To monitor the identified tasks, we trace the writes of the CSD to determine which task is executing. The tracing unit of the Infineon AURIX is capable of filtering the write operations and only generate data and timestamp messages when the CSD is written. With this, we can drastically lower the bandwidth and hence continuously monitor the task set without expensive tracing hardware. To determine if the task starts or continues execution, we look at the function trace (program addresses of calls and returns). With this setup, we get the necessary trace data to calculate the task set statistics.
Between two write operations on the CSD, we can see which task was executing for how long. The task can be identified by the biunique value written on the CSD at the first of the two write operations. Since this information is only sufficient to calculate a task's utilization, we also look at the program addresses that are executed between the two write operations, to determine whether the task is starting or continuing. If the first address of a task is executed between the two write operations, it is the task start. Otherwise, the task continues execution after a preemption. The end of a task is always the last time it was executing before a new task instance started. An example of this measurement method can be seen in Fig. 7, where one instance of each task can be measured.
Instead of using a program trace to identify the task's release and completion time, the approach of Iegorov *et al.* [13] could be applied here as previously mentioned in section 2. Only the timestamped trace of the CSD writes are then needed to monitor the task set. Since the program trace can be disabled, the tracing bandwidth will further decrease which enables NITRO for even slower tracing interfaces or multi-core monitoring.
This method for measuring the task statistics is very trace-memory- and bandwidth-efficient and hence we can use it for continuous traces. Since we only get timestamps on CSD write operations, this leads to some minor inaccuracy because we do not get the exact timestamp of the task execution start and end. Also, the runtime of RTOS related routines (scheduler, dispatcher, etc.) cannot be distinguished and hence is added to the task's runtime further impairing the measurement accuracy. With the Infineon AURIX, it is possible to trace ISR entry and exit points together with a timestamp. This information can be used to correct the task statistics. However, since the used tracing libraries did not support this, we could not implement this even though the on-chip tracing hardware is supporting it.

5.1.2 Test Generation. For the task set generation, we roll a random number of tasks $n \in [5, 9]$. Since we are using rate monotonic scheduling, we can calculate the upper bound of the task set utilization with the formula of Liu and Layland [21] to ensure that the generated task set will be feasible:

$$U_{ub} = n(2^{1/n} - 1)$$

With U_{ub} as the upper bound, we roll a random task set utilization $U \in [\max(0.5, U_{ub} - 0.2), U_{ub}]$. The utilization U is evenly distributed on all tasks of the task set with $u = \frac{U}{n}$. Each task also gets a unique and randomized period p assigned. As last, for every task a random number $m \in [3, 6]$ of sub-functions is rolled and the utilization u of the tasks is evenly distributed on these. The sub-functions itself implement the utilization as a `for`-loop executing `NOP` instructions. At a first glance, it may look insufficient to rely only on `NOP` instructions for the task load, but actually, it makes no difference in how the task load is implemented, due to the methodology that is used to detect different tasks. Since NITRO uses a trigger on the start address of a task function, the code executed within the task is never traced and processed, and therefore cannot affect task detection at all. However, complex task code leads to other difficulties as described later in section 5.2.
After the task set has been created, the code for its implementation is automatically generated, compiled and flashed.
The first RTOS we have used, ErikaOS, is a feature-rich OSEK compliant operating system. The tasks in ErikaOS are implemented as normal SWFs that are triggered by an OSEK timer event. As soon as a task is triggered, it runs once. As the second operating system, we used an AUTOSAR OS implementation of ARCCORE, similar

to ErikaOS. The tasks are again implemented as normal SWFs triggered by a timer event. The third operating system is FreeRTOS which is very lightweight compared to ErikaOS and AUTOSAR. It only provides the bare minimum of operating system features and does not implement a certain standard. The tasks in FreeRTOS are implemented as threads, meaning that each task is an SWF with an infinite loop. At the end of the loop, a delay function of the RTOS is called, which invokes the scheduler and dispatcher. All three operating systems also have a periodic system tick that runs the scheduler and dispatcher.

5.1.3 Test Evaluation. After setting up the test environment, NITRO starts its trace-based analysis to identify and monitor the task set. To validate the measurement results, three parameters of each measured task are compared to the parameters used for their generation:

(1) the number of tasks n' that were found
(2) the period p'_τ of each task τ
(3) the execution time c'_τ of each task τ

The comparison of the number of tasks n' is trivial since we just compare them to n, which is the number of generated tasks.
As the next test criteria, with the measured period p'_τ and the generated period p_τ of task τ, we can calculate the observational error in percent of the task set Γ:

$$E_p = \frac{1}{|\Gamma|} \sum_{\tau \in \Gamma} \left| \frac{p'_\tau}{p_\tau} - 1 \right| \cdot 100\%$$

The same calculation is done for the execution time c_τ that can be derived by the utilization u_τ and period p_τ used for the generation of task τ and hence for the complete task set Γ:

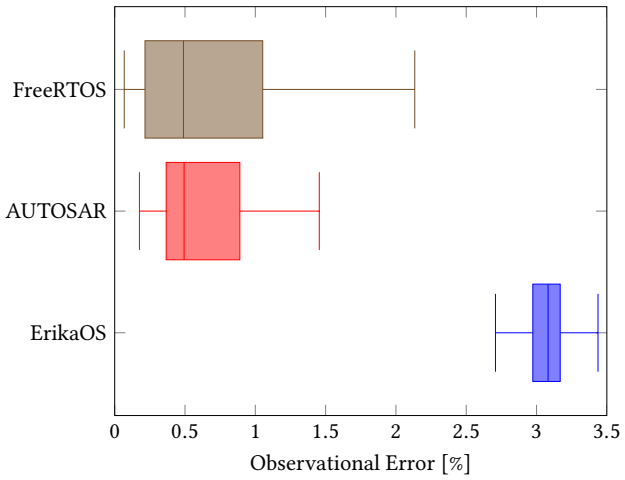
$$E_c = \frac{1}{|\Gamma|} \sum_{\tau \in \Gamma} \left| \frac{c'_\tau}{p_\tau \cdot u_\tau} - 1 \right| \cdot 100\%$$

with c'_τ as the measured average execution time of task τ.

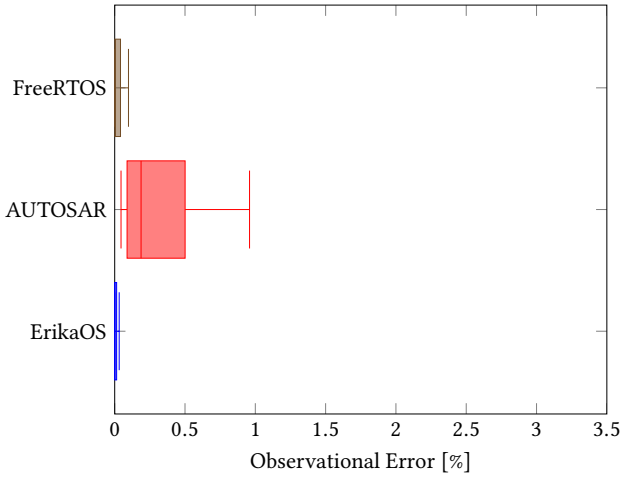
5.1.4 Results. To have reasonable results, we generated 50 randomized task sets. Therefore we executed 150 tests in total. NITRO was able to identify all task sets correctly for every RTOS. The average observational error of the measured execution times and periods of each task set (E_c and E_p) is below 5%, as can be seen in Fig. 8a and 8b.
It is conspicuous that the observational error on the measured execution time E_c for ErikaOS is higher than for AUTOSAR and FreeRTOS. This is because the period of the system tick in ErikaOS is just 0.1 ms. Since the execution time of the system tick is included in the measured execution time of a task, it has a clear impact on the measurement if the task has a long execution time. For example, if we have a task with an execution time of 10 ms, it covers the execution of 100 system ticks. If the execution time for one system tick is equal to 0.005 ms, the measured execution time of the task increases to $10\,\text{ms} + 100 \cdot 0.005\,\text{ms} = 10.5\,\text{ms}$ which results in an observational error e_p of

$$e_p = \left| \frac{10.5\,\text{ms}}{10\,\text{ms}} - 1 \right| \cdot 100\,\% = 5\,\%.$$

If the period and execution time of the system tick is known, an error term can be calculated to correct the measured values and further decrease the observational error. However, since NITRO only identifies and measures the tasks, it is not possible from the perspective of NITRO to calculate such an error term. Another problem of NITRO is the measurement impairment caused by non-RTOS software routines that are executed within ISRs. These routines would further decrease the accuracy of the implemented measurement method. As mentioned before, this can be solved, but we were limited by the used tracing libraries.



(a) Observational errors E_c on the measured execution time of the tested task sets.



(b) Observational errors E_p on the measured periods of the tested task sets.

Figure 8: Observational errors resulted from our tests.

5.2 Complex-System Experiment

The second experiment we have developed complements the first experiment by demonstrating that NITRO is also able to identify task sets consisting of complex code rather than only nop-operations. The code executed by the different tasks is taken from the *Powerstone Benchmark Suite* [20], which implements 13 different functions as can be seen in Tab. 5.
The task sets are generated in the same manner as described in

section 5.1, but instead of using loops with nop-operations, we assigned one of the functions to each task that can be generated. Thus, every generated task executes exactly one of the functions shown in Tab. 5 while every function is only executed by that respective task.
The main challenge we noticed on this test is that the probing phase may result in an incomplete call-graph due to gaps appearing while tracing. The gaps appear since the system now consists of way more functions that are called. However, this problem can easily be solved by increasing the trace duration for the probing phase or by repeating the probing phase multiple times. By extending the probing phase, a complete call-graph can be reconstructed even though there may be gaps in the trace data. Due to memory efficiency in the probing phase, we decided to increase the number of traces that are performed for probing and kept the trace duration constant at 60 s per trace.
Another issue that appeared while dealing with more complex code was, that there is a chance where the start of an ISR cannot be found within a fragment for the CSD identification. This is due to a higher activity of data read and write operations caused by the complex code. NITRO usually discards these fragments, which can then result in an incomplete task set after a CSD was found. However, it is possible to relax the conditions here, by letting fragments without an ISR pass to the next step, identifying the CSD. With this relaxation, we could easily deal with this issue and all task sets were identified properly. Anyway, this relaxation should only be used if needed since we expect that it may cause other issues. The conditions under which this relaxation can be done are not known yet but we plan to investigate this in the future.
For this experiment, we executed 50 tests for each RTOS. The outcome was that all tasks could easily be found by NITRO even with complex code involved. We also ran tests with task sets consisting of periodic and aperiodic tasks. Since we use triggers while tracing for the task identification, as explained in section 4, NITRO was also able to identify aperiodic tasks as expected.

Table 5: The different functions of the *Powerstone Benchmark Suite* [20].

Function	Description
`adpcm`	Voice Encoding
`bcnt`	Bit Manipulation
`blit`	Graphics Application
`crc`	Cyclic Redundancy Check
`engine`	Engine Controller
`fir`	Finite Impulse Response Filtering
`g3fax`	Fax Application
`huff`	Huffman Encoding
`jpeg`	JPEG Compression
`pocsag`	Asynchronous Protocol for Pagers
`qurt`	Quadratic Equation (Zero Search)
`ucbqsort`	Quick Sort
`v42`	Modem Application

5.3 False-Positive Experiment

As already explained in section 4.3, there is a chance that the code within a task or ISR favors the detection of false-positive task sets. Therefore we implemented two systems that can cause false-positives. The first system uses a state machine within a task. The variable of the state machine is changed at the beginning of the task – thus the state variable is a viable CSD. The second system we have implemented consists of a task set which runs different functions of the Powerstone Benchmark Suite. Along with this task set, a non-RTOS ISR runs periodically. With each execution of this ISR, a state-machine favoring false-positive classification is executed.
For the first test, the functions of the state-machine were identified as the task set – which is a false-positive. As explained previously in section 4.3, this is exactly what we expected. However, we could solve this issue easily by adding the constraint, that the identified task set is not maintained by a function that is part of another identified and viable task set. As we rerun the test, the state-machine was not identified as a task set anymore.
The result of the second test, where the state-machine resides in a non-RTOS related ISR, showed that the state-machine was detected as the task set – being a false-positive. However, we could not find a way to cope with this case yet but we are quite optimistic to do so in future work. Aside from solving this issue, the question of how probable this case is, in reality, remains unanswered as of now.

Our experiments have shown that NITRO works as expected and can probably be used along with many other real-time operating systems. Also, the identification and monitoring of the task set runs off-chip and therefore is along with the hardware tracing completely non-intrusive.

6 CONCLUSION

With NITRO we have shown that it is possible under certain assumptions, to automatically identify the task set from hardware trace data for a real-time operating system. This enables many previously costly and complex performance measurements for developers. For instance, the exact runtime of a function, which is executed within a low priority task with many preemptions, can only be measured when the task set is known. The approach was tested with a proof of concept tool with three different real-time operating systems and 50 task set configurations per system. The tests have shown that NITRO applies to various operating systems and that it can tackle the problem of a missing standard interface for RTOS monitoring. Further on the information retrieved by NITRO is sufficient to monitor the task set in a non-intrusive manner without the need for expensive tracing hardware. We also minimized user errors by avoiding RTOS specific software instrumentation and complex tool configurations. NITRO also enables task monitoring for real-time operating systems, where not all information is known or given due to IP components or legacy systems.
As of now, NITRO was only tested with single-core software configurations. However, we do not see a fundamental issue in applying it for multi-core setups which are used for hard real-time systems, due to the nature of operating systems used there.

REFERENCES

[1] Amazon Web Services, Inc. 2020. FreeRTOS – Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. http://www.freertos.org/

[2] ARM Limited. 2020. CoreSight Debug and Trace. https://developer.arm.com/ip-products/system-ip/coresight-debug-and-trace

[3] ASAM e.V. 2020. ASAM Run-Time Interface (ARTI). https://www.asam.net/project-detail/asam-run-time-interface-arti/

[4] AUTOSAR. 2020. AUTOSAR – Enabling Innovation. https://www.autosar.org/

[5] Björn B. Brandenburg and James H. Anderson. 2007. Feather-Trace: A Light-Weight Event Tracing Toolkit. *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)* (2007), 19–28.

[6] Sarah E Chodrow, Farnam Jahanian, and Marc Donner. 1991. Run-Time Monitoring of Real-Time Systems. *Real-Time Systems Symposium (RTSS)* (1991), 74–83. https://doi.org/10.1109/REAL.1991.160360

[7] Normann Decker, Boris Dreyer, Philip Gottschling, Christian Hochberger, Alexander Lange, Martin Leucker, Torben Scheffel, Simon Wegener, and Alexander Weiss. 2018. Online analysis of debug trace data for embedded systems. *Design Automation and Test in Europe (DATE)* (2018), 851–856. https://doi.org/10.23919/DATE.2018.8342124

[8] Normann Decker, Philip Gottschling, Christian Hochberger, Martin Leucker, Torben Scheffel, Malte Schmitz, and Alexander Weiss. 2017. Rapidly adjustable non-intrusive online monitoring for multi-core systems. *Formal Methods: Foundations and Applications* (2017), 179–196. https://doi.org/10.1007/978-3-319-70848-5_12

[9] ERIKA Enterprise. 2012. ERIKA Enterprise | Open Source RTOS OSEK/VDX Kernel. http://erika.tuxfamily.org/drupal/

[10] Paolo Gai, Enrico Bini, Giuseppe Lipari, Marco Di Natale, and Luca Albeni. 2000. Architecture For A Portable Open Source Real Time Kernel Environment. In *Real-Time Linux Workshop and Hand's on Real-Time Linux (RTLWS)*.

[11] Jason Gait. 1986. A probe effect in concurrent programs. *Software: Practice and Experience* (1986), 225–233. https://doi.org/10.1002/spe.4380160304

[12] IEEE- Industry Standards and Technology Organization (IEEE-ISTO). 2003. *The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface*. IEEE-ISTO. http://www.arbitrarytechnology.com/files/Download/ieee-5001nexusprotocol.pdf

[13] Oleg Iegorov, Reinier Torres, and Sebastian Fischmeister. 2017. Periodic Task Mining in Embedded System Traces. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Pittsburgh, Pennsylvania, 331–340. https://doi.org/10.1109/RTAS.2017.5

[14] Infineon Technologies AG. 2020. 32-bit AURIX™ Microcontroller based on TriCore™. https://www.infineon.com/AURIX

[15] Infineon Technologies AG. 2020. DAS – Infineon Technologies. https://www.infineon.com/DAS

[16] International Organization for Standardization (ISO). 2011. Road vehicles – Functional Safety Standard.

[17] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. 2018. TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams. *Annual ACM Symposium on Applied Computing (SAC)* (2018), 1925–1933. https://doi.org/10.1145/3167132.3167338

[18] Lin Li and Albrecht Mayer. 2016. Trace-based Analysis Methodology of Program Flash Contention in Embedded Multicore Systems. *Design Automation and Test in Europe (DATE)* (2016), 199–204. https://doi.org/10.3850/9783981537079_0442

[19] Lin Li, Philipp Wagner, Albrecht Mayer, Thomas Wild, and Andreas Herkersdorf. 2017. A non-intrusive, operating system independent spinlock profiler for embedded multicore systems. *Design Automation and Test in Europe (DATE)* (2017), 322–325. https://doi.org/10.23919/DATE.2017.7927009

[20] Qingan Li. 2014. The Powerstone Benchmark. https://github.com/li-qingan/powerstone

[21] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* (1973), 46–61. https://doi.org/10.1145/321738.321743

[22] OSEK. 2005. OSEK/VDX – Operating System. (2005). https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf

[23] OSEK. 2005. OSEK/VDX – OSEK Run Time Interface (ORTI) – Part A: Language Specification. (2005).

[24] José Rufino, António Casimiro, Antónia Lopes, Frank Singhoff, Stéphane Rubini, Valérie Anne Nicolas, Mounir Lallali, Mourad Dridi, Jalil Boukhobza, and Lyes Allache. 2018. NORTH – Non-intrusive observation and Runtime Verification of cyber-physical systems. *Ada User Journal* (2018).

[25] Neal Stollon. 2011. *Infineon Multicore Debug Solution*. Springer, Boston, MA. 219–230 pages. https://doi.org/10.1007/978-1-4419-7563-8_14

[26] Xilinx, Inc. 2020. Aurora 64B/66B. https://www.xilinx.com/products/intellectual-property/aurora64b66b.html