# Timing Behavior Characterization of Critical Real-Time Systems through Hybrid Timing Analysis

Salvatore Barone*, Valentina Casola*, Salvatore Della Torca*† and Daniele Lombardi*
*Department of Electrical Engineering and Information Technologies, University of Naples Federico II, 80125, Naples, Italy
†Department of Management, Information and Production Engineering, Univeristy of Bergamo, 24044, Dalmine, Italy
email: {name.surname}@{unina.it, unibg.it}

2023 7th International Conference on System Reliability and Safety (ICSRS) | 979-8-3503-0605-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSRS59833.2023.10381272

***Abstract*—The spread of computing-systems, especially the real-time embedded ones, is rapidly growing in the last years, since they find usage in numerous fields of application, including, but not limited to, industry process, critical infrastructures, transportation systems, as so forth. Indeed, in these fields, precise time-constraints hold; hence, tasks need to be correct from both the functional and temporal perspectives. As for the latter, timing behavior has to be characterized, that is usually done by exploiting either static or dynamic analysis techniques, which leverage estimations based on either a model or the actual system.**

**In this paper, we foster an automated hybrid approach that allows characterizing the timing behavior of systems while introducing any alteration, i.e., relying on instruction-level tracing rather than code instrumentation for profiling purposes. Our approach is sensitive to the execution-context, – e.g., cache misses – and it allows re-using results from the development processes – e.g., unit tests. We considered a complex real-time application from the railway domain as a case study to evaluate our approach, empirically proving that it can provide a faithful characterization of systems in terms of worst-case execution time.**

***Index Terms*—Safety-Critical Systems, Real-Time Systems, Hybrid Timing Analysis, Execution-Trace Analysis**

## I. Introduction

Critical systems, i.e., those systems for which failures or malfunctions can cause environmental harms, severe economic losses, and serious injuries to people, or even their death, are increasingly common in many application fields, e.g., heavy industries, critical infrastructures, and the medicine field. Depending on the actual application field, the development of such systems has to comply with strict regulations – e.g., the CENELEC 50128 for the railway domain [1] – which aim to guarantee that systems exhibit a given Safety Integrity Level (SIL), i.e., the probability of dangerous failures per hour must be less than a specific threshold [2]. These systems typically exhibit a real-time behavior, meaning that systems have to behave correctly both from the functional and timing perspectives. Hence, an upper bound to the timing behavior of such systems, in terms of Worts-Case Execution Time (WCET), has to be measured to prove that timing requirements are met. For instance, the CENELEC 50128 recommends characterizing the system behavior in terms of predictability and WCET, for all the components of a given software system [1].

Whereas a significant number of solutions have been proposed in the scientific literature to retrieve the WCET of real-time systems, at least two challenges still hold [3]. Specifically, (i) coping with the complexity of modern computer programs, which is growing exponentially, while being able to find the longest execution-time, and (ii) how to consider the hardware and all its possible states while estimating the WCET. Since the WCET can only be estimated, as its quantification is resource-intensive [4], [5], an additional challenge regards the quality of its estimation, which can be discussed in terms of (i) *safety*, i.e., the estimate should not be smaller than the actual WCET, and (ii) *precision*, i.e., the estimate should be as close as possible to the actual value. As far as the first challenge is concerned, to cope with growing complexity, several approaches consider segments of a given program, rather than the program as a whole. Consequently, the overall WCET is estimated based on the execution time of program segments [6], [7]. For what pertains to safety and precision, when the former has to be preferred over the latter, contributions from the scientific literature attempt to consider all the possible execution states of the analyzed system, as well as formal methods, for characterization purposes [8].

The scientific literature identifies three different approaches for the WCET estimation, namely the static, dynamic and hybrid approaches. Static timing analysis approaches leverage timing models formal approaches to characterize the system, placing safety before precision, while dynamic-approaches rely on the actual execution of the program as a mean to estimate the WCET. Hybrid approaches attempt to combine the advantages of both the latter.

In this paper, we present a novel hybrid approach for the estimation of the WCET of complex real-time systems. The program to be analyzed is decomposed into code units – i.e., set of statements – which execution time is measured while introducing no alteration to the timing behavior, relying on instruction-level tracing rather than code instrumentation. The approach is sensible to hardware states – e.g., cache-misses –, that can be injected for safety of the estimation, and it allows leveraging results from the development processes – e.g., unit or integration tests – for measurement purposes. To evaluate our approach, we consider a complex real-time application taken from the railway domain as a case study.

The remainder of the paper is organized as follows: Section II provides the reader with a brief overview of related works from the scientific literature. Section III deeply dis-

cusses our timing-characterization technique, while Section IV discusses the case study. Finally, we draw conclusions, and we outline possible future extensions in Section V.

## II. Related Works

Computing an upper bound of the execution time of a program, i.e., estimating the WCET, is a problem that research efforts have focused on for numerous years. Despite this, it is still of interest to the scientific community. Indeed, over the years, a plethora of methods and tools have been developed [3].

The scientific literature identifies three different approaches for the WCET estimation, namely the static, dynamic and hybrid approaches. Static timing analysis approaches leverage a timing model, derived from the hardware and software under investigation, and formal methods for characterization purposes; hence, they require neither running the system nor simulating it to estimate the WCET [9]. They favor safety of the estimation: they attempt to consider all the possible execution states of the analyzed system [8]. Anyway, the precision of the estimation strongly depends on the faithfulness of the model. Dynamic-approaches, on the other hand, rely on the actual execution of the program [7]; hence, they allow for a more precise estimation, giving up safety [8]. Indeed, there is no guarantee that every execution-state – e.g., cache-misses, pipeline flushes, branch misprediction, and so forth – is witnessed during simulations, as well as every execution path is taken. Hence, improper simulation approaches may result in underestimating the WCET, possible leading to a wrong characterization [10]–[12].

Hybrid approaches generally provide better trade-offs between safety and precision, since they attempt to combine both the latter approaches. The program to be analyzed is decomposed first, to bound its complexity, and the execution time of each code fragment is measured while running fragments on the actual hardware; then, the WCET is estimated through formal methods, based on the execution time of each fragment. As the measurement technique for the execution time is concerned, pioneering hybrid approaches rely on source-code instrumentation [13]. The latter adds specific code to the program under analysis, such that the execution of the code produces dump-data for runtime analysis, or component testing. Clearly, this alters the timing behavior of the system, and, consequently, resulting measurements suffer from the probe effect. Furthermore, the delays introduced by insertion or removal of code instrumentation may result in a non-functioning application, or unpredictable behavior.

Conversely, recent works leverages non-intrusive measurement techniques [14]–[17]. For instance, the authors of [18], [19] exploited execution traces to seek for performance issues in real-time tasks, the authors of [20] leveraged machine learning to analyze execution traces while targeting GNU/Linux system calls, while the authors of [14], [15] leverage execution trace analysis to estimate the WCET while exploiting the Field Programmable Gate Array (FPGA) for on-the-fly analysis.

Moreover, when combined with graph theory, execution trace analysis allows modeling complex scenarios that may arise during the execution of a system: the authors of [21] exploited a graph-based representation of dependencies between software-threads and hardware resources with the aim to detect bottlenecks, while the authors of [22] leveraged a dependency graph to analyze latencies of hardware resource usage.

While the above-mentioned works allow resolving very specific problems, our work is more generalizable and can be applied in different contexts. Indeed, it is suitable to analyze all kinds of execution trace, not only those generated from specific executions, e.g., GNU/Linux system calls. Also, both latencies of hardware resource usage and bottlenecks can be detected because it provides a complete timing characterization of the system. Finally, our technique does not require additional hardware, FPGA for instance.

Besides formal methods and measurement-based approaches, recently, probabilistic-based techniques have been proposed in the scientific literature [23]. Such techniques characterize a system without relying on a scalar value such as WCET, rather on a probability distribution, namely the pWCET distribution, which is gathered from multiple executions of the system under analysis.

However, such an approach is costly, and it requires knowing the pWCET distribution and modelling dependencies between tasks, meaning that the complexity and the computational cost grow.

## III. Investigating the temporal behavior of critical real-time systems

In this Section, we detail our hybrid technique to analyze the timing-behavior of critical real-time systems, which involves: (i) static analysis of the code of the system, aiming to test-cases generation, which is presented in Section III-A, (ii) the execution of the previous test-cases on the actual target, that is discussed in Section III-B, to collect execution traces, and (iii) a final analysis step, debated in Section III-C, which provides a graph representation of the execution flow and an accurate analysis of the execution time.

The description of each step is supplied with a running example, which is shown in Listing 1. The code-snippet, extracted from an actual code unit, has been intentionally chosen to be as brief and simple as possible, to clearly and concisely introduce our proposed technique.

### A. Static analysis

In this step, the program under-analysis is decomposed into its subunits – e.g., functions or tasks – and unit tests are automatically generated by following the approach presented in [24], in order to facilitate the WCET estimation.

Input test pattern generation resorts on the Abstract Syntax Tree (AST)-representation of units. The AST is a tree-based representation resulting from the syntax analysis step of the compilation phase, and it often serves as an intermediate representation of the program onto which compilers work through the compilation and linking steps. For instance, Figure 1

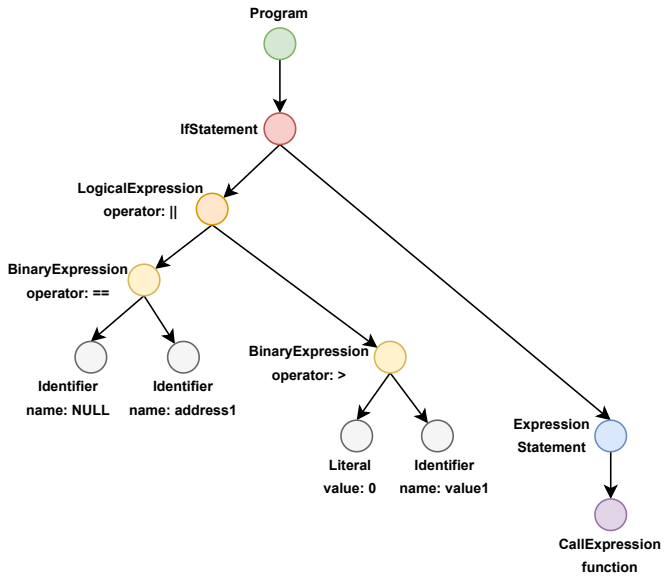


Fig. 1. AST-graph of the code in Listing 1.

```
if((NULL == address1) || (0U > variable1))
    function();
```

Listing 1. Code snippet of the running example

```
000b048:  cmp x0, #0x0
000b04c:  b.ne 000b074
000b050:  mov w4, #0x12e
000b054:  adrp x0, 0013000
  .   .   .   .   .   .
000b070:  bl 0010c6c <function>
000b074:  adrp x0, 0015000
  .   .   .   .   .   .
000b080:  cmp w0, #0x0
000b084:  b.ne 000b0ac
000b088:  mov w4, #0x12e
000b08c:  adrp x0, 0013000
  .   .   .   .   .   .
000b0a8:  bl 0010c6c <function>
000b0ac: ldr w0, [sp, #44]
```

Listing 2. ARM Assembly translation of the snipped code shown in Listing 1

represents the AST-graph referred to the code in Listing 1. As depicted, each node denotes a language construct of the analyzed code, and information pertaining to branches, loops, function-calls, and so forth. The AST furnishes only structural and content-related details of the program, yet it preserves variable-types and their declaration, the order of statements, identifiers, assignment statements, and the left and the right-hand side operands of binary operators.

We generate input patterns by searching for specific nodes within the AST, e.g., those corresponding to function definitions. Then, we search for *branches* and *jumps* – e.g., *if-then* statements – *parameter-declaration*, *variable-declaration*, *constant-declaration*, and *literal* nodes, to collect parameters and variables involved in conditions and decisions. The latter are identified by searching for *binary-operator* nodes within the AST, and decomposed. Then, based on the binary operator defining the condition and its operands, a proper input assignment is generated to make the outcome either *true* or *false*. Kindly note that the discussed test-case generation procedure is clearly discretionary: if the concerned system has already been tested, test-cases generated during the development life-cycle can be reused, with no need to generate new ones.

Let consider, for instance, the `address1 == NULL` condition from the Listing 1: to make the outcome either *true* or *false*, the input assignments `address1 = NULL` and `address1 = value` are generated, with $\texttt{value} \neq \texttt{NULL}$ being plausible for the executed code.

After suitable input assignments for conditions have been determined, a minimum set of test cases is selected for each of the decision within the concerned unit, while taking into consideration the Modified Condition/Decision Coverage (MC/DC) metric, which is the common choice for measuring test coverage in safety critical environments [25]. It requires that: (i) every entry/exit point in the program is invoked at least once; (ii) every decision in the program takes all possible outcomes at least once; (iii) every condition in a decision takes all possible outcomes at least once; (iv) each condition in a decision shows to affect the outcome of the involved decision independently.

Higher the coverage while utilizing the MC/DC metric, higher the path coverage, as well as the probability of the longest lasting execution path to be considered during our analysis. Besides, to correctly observe all the executed instructions which contribute to the execution-time, it is required to analyze the program while considering object-code. As an example, Listing 2 shows the assembled code for Listing 1. It is worth noticing that, by evaluating the second condition of the `if-then` statement of the Listing 1, more Assembly instructions will be executed. This is due to the short-circuit logic being exploited by compilers to optimize the execution time of computer programs.

### B. Program execution and tracing

Once test cases are available, either automatically generated or resulting from the development process, they are executed while targeting the actual hardware, and a non-intrusive debugging is exploited to collect execution traces. Non-intrusive debugging alters neither the source-code nor the temporal behavior of the unit under test, while simultaneously allowing to recreate preconditions for each of the test case by introducing very precise modifications to the value of variables during the execution. Besides, it allows altering the hardware states: for instance, the cache can be invalidated to cause a cache-miss, benefitting the precision of the estimated WCET [11].

Once test-cases are performed, timestamped execution-traces are extracted by using an off-chip tracing technique [26] – that resort to an external device to store the trace, rather than the on-chip ones – allowing to analyze the system for a longer time.

Extracted execution traces are composed of several records, each produced while observing the occurrence of events in

the system. It is up to the trace-source what event generates a record in the trace. The Embedded Trace Macrocell (ETM) of the ARM CoreSight™, which is embedded into almost every ARM CPU, for instance, generates records when *jumps* or *branches* instructions are executed [27]. Kindly note that, to be suitable for our technique, the above-mentioned records in the execution trace have to include at least the address and the timestamp of an occurred event. Specifically, the timestamp is generally expressed in terms of the tracer clock-cycles, measured as the time elapsed from the tracing-start to the event-occurrence.

Before undergoing the analysis, execution traces have to be parsed, record per record, to map them with the source code. The parsing operation retrieves symbols – e.g., the name of a function – at the source-code level, by leveraging the memory-addresses within the trace record. Such knowledge is exploited in the very next phase, i.e., during the execution-trace analysis.

### C. Execution-trace analysis

Once tests have been executed and execution traces collected, we exploit them to build the control-flow graph (CFG) of the concerned software, i.e., a graph-based representation of all paths that might be traversed through a program during its execution. We annotate the CFG with the execution time for each event of interest as the difference between its timestamp, and that of a previous event in the execution trace, the execution trace itself to which such execution time refers, and the corresponding test being executed. All of these annotations will serve the subsequent analysis. Kindly note that the same given path can be identified in multiple different execution traces, and that it can exhibit different execution times, as we already observed while discussing Listing 2.

Once all the execution traces related to the same code unit have been analyzed, our technique identifies which path requires the longest execution time by traversing the annotated CFG. Figure 2, for instance, shows results for Listing 1: the longest lasting path is highlighted in red, and the annotation provides information about the execution traces (and the corresponding test being executed) to which such execution time has been observed.

## IV. Evaluation

For evaluation purposes, we resort to a real-world safety-critical software from the railway domain. The considered software implements part of the functionalities of the European Rail Traffic Management System/European Train Control System (ERMTS/ETCS) standard, and it consists of many periodic real-time tasks, whose execution flow, albeit quite complex, can be summarized as tasks either behaving as *producers*, or as *consumers*. The considered software monitors the location of the train on the rail route. It consists of different processing units: *Pos* is the one assigned to calculate the precise position, based on the odometric information and that from the eurobalises, respectively, obtained by interfacing with the *Odo* and *Btm* units. The latter perform decoding and pre-processing operations of the information coming from

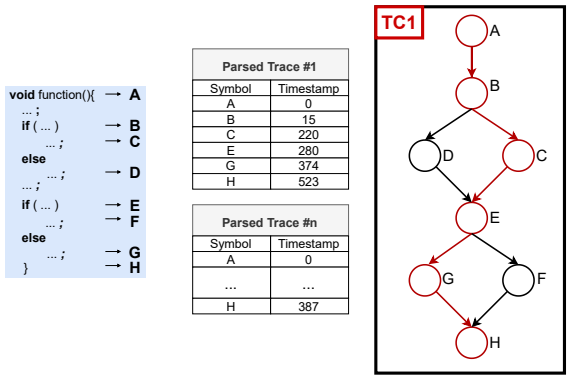


Fig. 2. Exemplary application of our technique. The figure shows a unit – to which the code in Listing 1 belongs – (on the left side), two example of parsed trace (center) and the result on the CFG of the unit (on the right side). The longest path, whose execution time is computed by exploiting its parsed execution-trace, is highlighted in red.

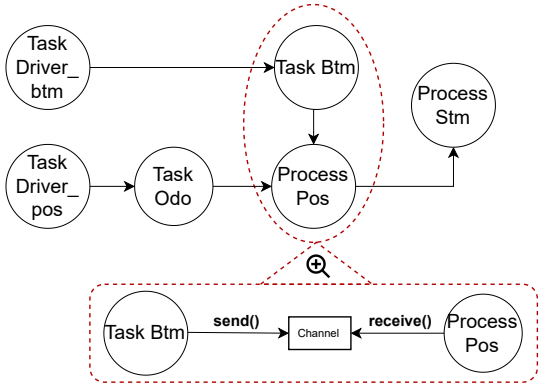


Fig. 3. An application from the ERMTS/ETCS standard.

the *Driver_Odo* and *Driver_Btm* tasks, which implement the device driver functions.

The whole system operates under memory segregation constraints, and any communication between software units is arbitrated by the real-time operating system (RTOS) through message-passing inter-process comunications (IPCs). The RTOS also orchestrate communications between tasks and the underlying hardware.

The mentioned software runs on four ARM® Cortex-A53, and it is two-out-of-two (2oo2) redundant, meaning that the software is executed by two replicas, which receive the same inputs and are expected to provide the same output. The deployment to specific cores is statically defined by the vendor, to guarantee a higher system predictability.

Given the complexity, we consider only a subset of the mentioned software, as shown in Figure 3, to make the discussion plain. Specifically, we focus on the communication between *Task Btm* and *Task Pos*, which behave as producer and as consumer, respectively. Both are periodic tasks, with $500\mu s$ period, rather they exhibit different scheduling priority, with *Task Pos* having higher priority than *Task Btm*.

Hereafter, we report a detailed analysis made on the producer unit, i.e, `send()` function, whose source-code is shown in Listing 3. As mentioned, the RTOS – whose source-code

```
send_status_t send(uint32_t channel_id,
        const uint8_t * const sendbuff,
        const uint32_t size)
{
    int32_t channel_index = -1;
    send_status_t send_status = send_error;
    rtos_return_code_t retcode;
    channel_index = get_channel(channel_id);
    ASSERT_GTE(channel_index, 0);
    ASSERT_WITHIN(size, 1, MAX_SIZE);
    ASSERT_NOT_NULL(sendbuff);
    retcode = rtos_send(channel_index, sendbuff, size);
    send_status = send_error_handler(retcode, size);
    return send_status;
}
```

Listing 3. Code of the function `send()`. The function is invoked by the producer task and take as input the id of the channel used to send the message, the address of buffer from what the message is colected, and its size. First, the function retrieve from the id the channel and asserts its existence. After asserting that the size of the message is correctly bounded and the address of the buffer is not a null pointer, the message is sent to task consumer and the result of such operation is compared to the one obtained by the other reply of the redundant architecture.

is not accessible – orchestrates communications. Specifically, it handles operations such as retrieving the channel from the *channel_id*, message-exchange through the channel and error-handling.

Concerning execution trace recording, we exploit unit test-cases for the `send()` unit that have already been specified and implemented during V&V, according to the MC/DC metric. The execution of such tests and the timing behavior of the system are observed by relying on the ARM ETMv4 [26], and on the Lauterbach PowerTrace II [28]. The latter allows us to execute test-cases, and to alter both inputs and variables without altering the software. In addition, it provides the mean to modify the hardware state, allowing the behavior of the software to be evaluated starting from different preconditions, e.g., under cache miss or cache hit. In this manner, we can cover any possible execution path in the considered software, defining specific execution scenarios to cover that path.

Once a trace is captured, it is first parsed and then analyzed to compute execution times of the `send()` function, and to find the longest lasting execution path. Figure 4 reports the CFG of the concerned function, while Table I shows the path covered by each test-case, according to Figure 4, and the execution times of each path, expressed in terms of clock cycle. As the reader can observe, the first test-case, and hence the path it covers, requires more times to execute than the others. Therefore, the concerned path is highlighted in the CFG, as shown in Figure 5.

The results shown previously can be further exploited to perform a schedulability analysis.

Our methodolgy proves both disclosed tasks complete their operations within their period. Indeed:

- *Task Btm* only calls the `send()` function, which take at most 36442 clock cycles to execute in case of cache-hit and 99661 in case of cache-miss, i.e., $36.44\mu s$ and $99.66\mu s$;
- *Task Pos*, by calling twice the `receive()` function, receive from both *Odo* and *Btm* tasks, and then compute the speed of the train. The above-mentioned function has a WCET of 48563 clock cycles to execute in case of cache-hit and 145702 in case of cache-miss, i.e., $48.56\mu s$ and $148.70\mu s$. Finally, the computation of the train-position is performed by calling a new function – which analysis are not reported in this paper – requires at most $60.23\mu s$ and $101.45\mu s$ in case of cache-hit and cache-miss, respectively.

| Test Case | Path Covered | Execution Time Cache-Hit | Execution Time Cache-Miss |
|---|---|---|---|
| 1 | start → A → B → D → F → H | 36442 | 99661 |
| 2 | start → A → c | 6596 | 49871 |
| 3 | start → A → B → E | 7891 | 52871 |
| 4 | start → A → B → D → G | 8931 | 57513 |

TABLE I
EXECUTION TIMES OF THE FUNCTION `SEND()` EXPRESSED IN CLOCK-CYCLES.

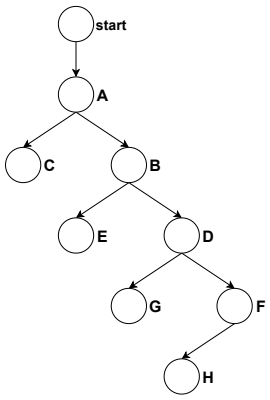


Fig. 4. CFG of the function `send()`. Each node is labelled with a symbol represented the concerend branch or jump.

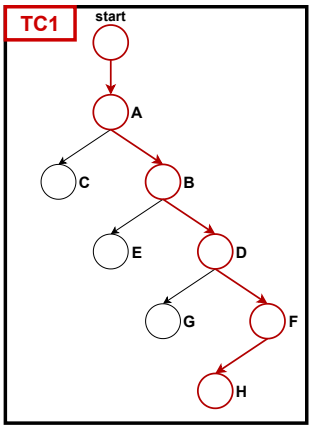


Fig. 5. Result of the applied methodology.

## V. CONCLUSION

In this paper, we discuss an automated hybrid approach that allows characterizing the timing behavior of safety critical systems while introducing any alteration, i.e., relying on instruction-level tracing rather than code instrumentation for profiling purposes. The approach allows re-using unit tests resulting from the development processes to stimulate the concerned software, aiming at collecting execution traces. The latter are exploited to build the control-flow graph of the concerned software, which is a graph-based representation of all paths that might be traversed through a program during its execution, on which we perform our execution time analysis.

We considered a complex real-time application from the railway domain as a case study to evaluate our approach, empirically proving that it can provide a faithful characterization of systems in terms of worst-case execution time.

The advantages provided by our approach can be summarized as follows: (i) it automatically generates test-cases or reuses them, if any, (ii) it does not counterfeit the temporal behavior of the system under analysis, (iii) it is context-sensitive, since it automatically forces certain hardware states, e.g, cache-miss, (iv) it allows estimating either the WCET of all units in the program or the entire program itself.

## REFERENCES


[1] Comité européen de normalisation en électronique et en électrotechnique, "Railway applications - Communication, signalling and processing systems Software for railway control and protection systems," Comité européen de normalisation en électronique et en électrotechnique, Tech. Rep., 2011.

[2] R. Carbone, S. Barone, M. Barbareschi, and V. Casola, "Scrum for Safety: Agile Development in Safety-Critical Software Systems," in *Quality of Information and Communications Technology*, A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, and R. Pérez-Castillo, Eds. Springer International Publishing, 2021.

[3] V. P. Kozyrev, "Estimation of the execution time in real-time systems," *Programming and Computer Software*, vol. 42, no. 1, pp. 41–48, Jan. 2016. [Online]. Available: https://doi.org/10.1134/S0361768816010059

[4] H. Li, P. De Meulenaere, K. Vanherpen, S. Mercelis, and P. Hellinckx, "A hybrid timing analysis method based on the isolation of software code block," *Internet of Things*, vol. 11, p. 100230, Sep. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660520300639

[5] A. Bucaioni, E. Ferko, and H. Lönn, "Trace-based Timing Analysis of Automotive Software Systems: an Experience Report," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Oct. 2021, pp. 254–263.

[6] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-Based Timing Analysis," in *Leveraging Applications of Formal Methods, Verification and Validation*, ser. Communications in Computer and Information Science, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer, 2008, pp. 430–444.

[7] N. Merriam, P. Gliwa, and I. Broster, "Measurement and tracing methods for timing analysis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 1, pp. 9–28, Feb. 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0266-6

[8] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner, "Improving the Confidence in Measurement-Based Timing Analysis," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Mar. 2011, pp. 144–151, iSSN: 2375-5261.

[9] A. Löfwenmark and S. Nadjm-Tehrani, "Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective," *Journal of Systems Architecture*, vol. 87, pp. 1–11, Jun. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762117304903

[10] M. Zolda, S. Bünte, and R. Kirner, "Context-Sensitivity in IPET for Measurement-Based Timing Analysis," in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer, 2010, pp. 487–490.

[11] S. Stattelmann and F. Martin, "On the Use of Context Information for Precise Measurement-Based Execution Time Estimation," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 64–76, iSSN: 2190-6807. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2826

[12] S. Law and I. Bate, "Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2016, pp. 189–199, iSSN: 2159-3833.

[13] A. Betts, N. Merriam, and G. Bernat, "Hybrid measurement-based WCET analysis at the source level using object-level traces," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 54–63, iSSN: 2190-6807. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2825

[14] B. Dreyer, C. Hochberger, S. Wegener, and A. Weiss, "Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation," in *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, ser. OpenAccess Series in Informatics (OASIcs), F. J. Cazorla, Ed., vol. 47. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 45–54, iSSN: 2190-6807. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/5255

[15] B. Dreyer, C. Hochberger, A. Lange, S. Wegener, and A. Weiss, "Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 4:1–4:11, iSSN: 2190-6807. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6897

[16] O. Iegorov, "Data Mining Approach to Temporal Debugging of Embedded Streaming Applications," Theses, Université Grenoble Alpes, Apr. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/tel-01321286

[17] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, "TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis," in *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, ser. OpenAccess Series in Informatics (OASIcs), S. Altmeyer, Ed., vol. 72. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 1:1–1:11, iSSN: 2190-6807. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10766

[18] M. Côté and M. R. Dagenais, "Problem Detection in Real-Time Systems by Trace Analysis," *Advances in Computer Engineering*, vol. 2016, pp. 1–12, Jan. 2016. [Online]. Available: https://doi.org/10.1155/2016/9467181

[19] M. Hendriks, J. Verriet, T. Basten, B. Theelen, M. Brassé, and L. Somers, "Analyzing execution traces: critical-path analysis and distance analysis," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 4, pp. 487–510, Aug. 2017. [Online]. Available: http://link.springer.com/10.1007/s10009-016-0436-z

[20] Q. Fournier, D. Aloise, S. V. Azhari, and F. Tetreault, "On Improving Deep Learning Trace Analysis with System Call Arguments," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, May 2021, pp. 120–130. [Online]. Available: https://doi.org/10.1109/MSR52588.2021.00025

[21] N. Ezzati-Jivan, Q. Fournier, M. R. Dagenais, and A. Hamou-Lhadj, "DepGraph: Localizing Performance Bottlenecks in Multi-Core Applications Using Waiting Dependency Graphs and Software Tracing," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2020.

[22] M. Zoor, L. Apvrille, and R. Pacalet, "Execution Trace Analysis for a Precise Understanding of Latency Violations," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Oct. 2021, pp. 123–133. [Online]. Available: https://doi.org/10.1109/MODELS50736.2021.00021

[23] R. I. Davis and L. Cucu-Grosjean, "A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems," *Leibniz Transactions on Embedded Systems*, vol. 6, no. 1, pp. 03:1–03:60, May 2019, number: 1. [Online]. Available: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v006-i001-a003

[24] M. Barbareschi, S. Barone, V. Casola, S. Della Torca, and D. Lombardi, "Automatic Test Generation to Improve Scrum for Safety Agile Methodology," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23, 2023, pp. 1–6.

[25] K. J. Hayhurst, "A Practical Tutorial on Modified Condition/Decision Coverage," DIANE Publishing, Tech. Rep., 2001, google-Books-ID: aqMz3xtU6HsC.

[26] ARM, "ARM - ETMv4 Architecture Specification," Arm Limited, Cambridge, Tech. Rep., 2020. [Online]. Available: https://developer.arm.com/documentation/ihi0064/h

[27] ——, "CoreSight Components Technical Reference Manual," ARM, Tech. Rep., 2009. [Online]. Available: https://developer.arm.com/documentation/ddi0314/h/

[28] Lauterbach GmbH, "LAUTERBACH DEVELOPMENT TOOLS." [Online]. Available: https://www.lauterbach.com/frames.html?home.html