Fitting Software Execution-Time Exceedance into a Residual Random Fault in ISO-26262

Irune Agirre, Francisco J. Cazorla, Jaume Abella, Carles Hernandez, Enrico Mezzetti, Mikel Azkarate-askasua, and Tullio Vardanega

***Abstract*—Car manufacturers relentlessly replace or augment the functionality of mechanical subsystems with electronic components. Most such subsystems (e.g., steer-by-wire) are safety related, hence subject to regulation. ISO-26262, the dominant standard for road vehicles, regards software faults as *systematic*, while differentiating hardware faults between systematic and *random*. The analysis of systematic faults entails rigorous processes and qualitative considerations. The increasing complexity of modern on-board computers, however, questions the very notion of treating the violation of execution-time envelopes for software programs as a systematic fault. Modern hardware in fact reduces the user's ability to delve deep enough into the fabric of hardware-software interaction to gage its extent of contribution to worst-case execution time (WCET). Changing the nature of the WCET-analysis problem may help address that challenge effectively. To this end, we propose a solution that should allow ISO-26262 to quantify the likelihood of execution-time exceedance events, relating it to target failure metrics employed in support of certification arguments, similarly to random faults in hardware. To this end, we inject randomization in the timing behavior of the computer hardware to relieve the user from the need to control hard-to-reach low-level parts, and use Measurement-Based Probabilistic Timing Analysis (MBPTA) to quantify, constructively, the failure rates resulting from the likelihood of execution-time exceedance events.**

***Index Terms*—Execution-time exceedance, safety certification, measurement-based probabilistic timing analysis (MBPTA), automotive real-time systems**

I. Introduction and Motivation

An increasing variety of functions in modern cars are controlled by electrical and/or electronic (E/E) subsystems; for instance, active/passive safety and driver assistance. For quantity, complexity, and use, those functions make the safety of E/E systems an increasingly important and complex matter.

ISO-26262 [27] is the functional safety standard of reference for the automotive domain. ISO-26262 is an adaptation of the broader IEC-61508 safety standard, which has been similarly adapted to nuclear plants, industrial machinery, railway, and other application domains (see Figure 1).

ISO-26262 seeks to preserve systems' safety by sustaining *safety goals* (SG) that prevent hazardous situations due to E/E malfunction. To this end, ISO-26262 (much like the parent IEC-61508) defines procedures for the management of deterministic design faults (i.e., *systematic* faults) and unpredictable hardware faults (i.e., *random* faults). The ISO-26262 tenet is that systematic faults can be either avoided by adopting prevention measures throughout the development process, or controlled at run time by safety mechanisms such as diverse redundancy. ISO-26262 uses cognizant assessment, based on judgment from practical experience, to guarantee that the contribution of systematic faults to SG violation is kept acceptably low by assuring coverage of all requirements of the standard. Conversely, random faults can only be controlled at run time: ISO-26262 requires their likelihood of occurrence to be quantified and assessed against reference values, asserting with sufficiently high confidence that the *residual risk* of SG violation falls below tolerable rates.

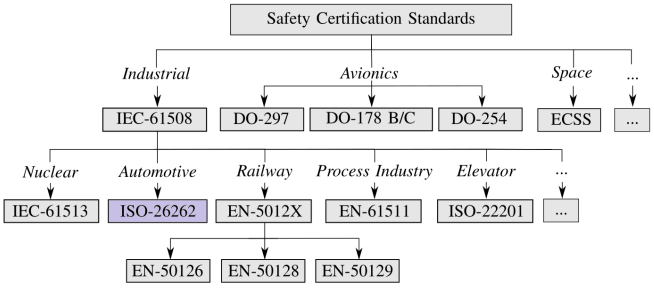


Fig. 1: Safety standards in different application domains and those inheriting from IEC-61508 (including ISO-26262)

Motivation. While for hardware parts[1], the standard contemplates both systematic and random hardware faults, software faults are all deemed systematic. Yet, software has functional and non-functional traits, which may give rise to different fault trees, ill-fit for the homogeneous treatment prescribed by ISO-26262. This problem becomes apparent for *execution-time exceedance* events (i.e., the violation of worst-case execution-time, WCET, boundaries), which is a non-functional trait, evidently involving software *and* hardware concerns. An incorrect (optimistic) WCET estimation may be the root cause of a possible deadline violation and thus of a timing failure. For instance, the system design may assign a task an insufficient execution-time allowance, and this under-provision may go unnoticed because the established boundary value is only exceeded when rare circumstances of hardware/-

I. Agirre and M. Azkarate-askasua are with the department of Dependable Embedded Systems, IK4-IKERLAN, Mondragón 20500, Spain (e-mail: iagirre@ikerlan.es; mazkarateaskasua@ikerlan.es).

F. Cazorla, J. Abella, C. Hernandez and E. Mezzetti are with the CAOS group, Barcelona Supercomputing Center, Barcelona 08034, Spain (e-mail: francisco.cazorla@bsc.es; jaume.abella@bsc.es; carles.hernandez@bsc.es; enrico.mezzetti@bsc.es). F. Cazorla is also with IIIA-CSIC, Barcelona, Spain

T. Vardanega is with the department of Mathematics, Universita degli Studi di Padova, Padova 35122, Italy (e-mail: tullio.vardanega@math.unipd.it).

[1] In this paper we focus on the functional safety of the computer subsystems in cars, using the term *hardware* to refer to embedded computers within the automotive E/E; likewise we use *software* to refer to applications.

software interaction happen, either undocumented or unknown, or exceedingly hard for the user to reproduce during WCET analysis. Determining the WCET of a software program is a very difficult task indeed, as the programs' execution time varies much beyond user control (and, sometimes, also comprehension). This strenuous task is being made significantly harder by the massive increase in complexity of the hardware and software of modern automotive systems. Postulating that such execution-time violations can all be prevented by standard procedures defined for systematic faults is becoming increasingly prohibitive, yielding unsatisfactory ratios of effort vs. quality of outcome. The following observations manifest the magnitude of the problem:

① While the software embedded in cars already totals hundreds of millions lines of code [17], the computational needs of novel functionalities such as Advanced Driver Assistance are projected to increase by 100x in the next decade [10]. This trend reflects the centrality of software to a rising proportion of the competitive value of the vehicle.

② Those performance needs can only be met with high-performance processors that include multi- and many-core components, with deep cache hierarchies and high-end GPUs (like in the NVIDIA DrivePX [2], RENESAS R-Car H3 [4], QUALCOMM Snapdragon 820 Processor [3], and the Intel Go [1]), with massively increasing hardware complexity.

A string of increasingly powerful WCET analysis techniques for safety-critical systems has been put forward by the research community over the last two decades [45], [7], and commercial tooling exists that implements (some of) them [42], [44]. Yet, the most part of those techniques only really applies to a small subset of relatively simple and highly critical (ASIL-D) sub-systems running on simple and well-understood processor architectures, thus only covering a fraction of the needs. For most subsystems, therefore, the common industrial practice to upper-bound the execution time of real-time software programs uses high-water mark (HWM) measurements and adds a safety margin to them to account for unobserved behavior. With this practice, the confidence in the resulting estimates rests on the user ability to: (i) understand the hardware internals well enough to capture the major sources of execution-time variability, and (ii) construct test cases that serve for WCET determination effectively. This knowledge, together with the addition of a conservative margin, sustains the argument that the risk of missing out relevant situations in the analysis is sufficiently low. Whilst this approach may seem inadequate in comparison to state-of-the-art static analysis solutions for other than low-criticality parts, evidence of (cautious) use of measurement-based methods exists for DO-178C-certified avionics software at the highest criticality [34].

Emerging systems imperil the current timing analysis practices, by challenging the user ability to understand deep enough the sources of jitter in the hardware internals, and to control them. The former weakness hinders the determination of how hardware-software interactions affect timing; the latter impairs the creation of effective analysis scenarios. In those circumstances, the risk of execution-time exceedance events can be made "sufficiently" low by using either inordinately large margins (hence renouncing resource efficiency) or lower margins with less support evidence (hence increasing risk). Either prospect faces the user with a dire conundrum.

Measurement-Based Probabilistic Timing Analysis, nicked MBPTA [8], proposes a set of techniques that require applying small and sustainable changes in the hardware design (or alternatively in dedicated runtime libraries) to cause the system to exhibit a probabilistic – hence probabilistically analyzable – timing behavior. In this way, MBPTA provides by-construction evidence to quantify the probability of execution-time exceedance events. Earlier work describes how to design *MBPTA-friendly* hardware and software platforms [30], [31] such that execution-time exceedance occurs with an (arbitrarily low) probability. Both hardware [30] and software [31] implementations of MBPTA support have proven viable even with complex hardware designs (i.e., multicore processors with multi-level cache hierarchies), in space [26] and automotive platforms [28], with successful evaluation in industrial case studies [19], [24], [28]. So far however, there is lack of understanding of how the probabilistic treatment of execution-time exceedance events can be understood by safety certification standards in general, and ISO-26262 in particular. In this regard, this paper seeks to answer the following research question:

How does the approach of quantifying the probability of occurrence of execution-time exceedance events fit the scope and intent of ISO-26262?

Contribution. To address this research question, this work analyzes ISO-26262 and its treatment of faults, describing how probabilistic execution-time analysis solutions can satisfy the ISO-26262 prescriptions and how quantitative evidence can be obtained to support certification arguments. Our contention is that to tackle this challenge satisfactorily we should change the nature of the WCET-analysis problem: safety standards should be enabled to allow sound quantification of the execution-time exceedance rate (or its likelihood of occurrence), in relation with target failure metrics associated with SG. This approach would be akin to established practice for hardware random faults in ISO-26262, and would no longer require leaning on qualitative cognizant experience, scarcely available with new hardware, as in current practice for the treatment of systematic faults.

Accordingly, in the remainder of this paper: we survey the management of systematic and random faults in ISO-26262 (Section II); we show how an asymmetric treatment of software faults that addresses execution-time exceedance probabilistically, fits in the safety life cycle defined in ISO-26262 and how it can be extended to IEC-61508 (Section III); we present the concept of MBPTA as a solution to quantify execution-time exceedance rates, and examine the feasibility of applying it to ISO-26262 compliant automotive applications for conformance to the standard intent and prescriptions and for cost of hardware and software modifications (Section IV); we provide evidence of the viability of the proposed approach with an automotive case study targeting the AURIX [43], a multicore processor candidate for use in automotive systems

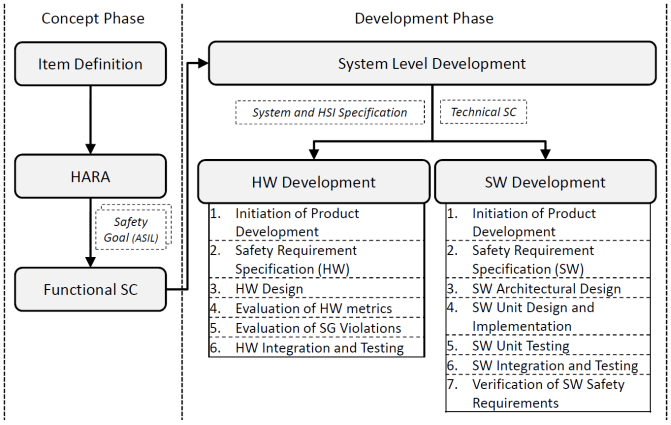


Fig. 2: Schematic view of ISO-26262 concept and development phases.

(Section V). Finally, in Section VI, we draw the main conclusions from this work.

II. Hardware and Software Faults in ISO-26262

ISO-26262 requires the user to provide evidence of the absence of unreasonable risk due to hazards caused by the malfunction of E/E systems. For the management of functional safety, ISO-26262 includes a concept definition phase, system, hardware and software development processes, and production and operation measures. Figure 2 depicts the ISO-26262 workflow for the concept and development phases, which are the focus of this work.

Concept Phase. For each item to be developed and certified, the *Hazard Analysis and Risk Assessment* (HARA) step defines the set of hazardous events caused by item's malfunction under specific operational situations. Safety experts classify the hazardous events at different integrity levels – called *Automotive Safety Integrity Levels* (ASIL) – based on their severity, probability of exposure and controllability. The ASIL levels range from A to D, with D being the most restrictive. Overall, this step formulates the safety goals and associated ASILs for each hazardous event.

For each safety goal, the *functional safety concept* (functional SC) defines the *safety measures* to be implemented in the item. Rather than the technical implementation details, the functional SC describes the functional *safety requirements* to achieve the safety goal. *Safety measures* include activities for the avoidance of systematic faults and technical *safety mechanisms* to detect and control errors caused by systematic and random hardware faults. Whenever a safety mechanism detects an error, an action shall be taken as defined in the functional SC. In the application domain of ISO-26262, this action typically seeks to achieve or maintain a *safe state*, in which no unreasonable level of risk is known to exist. If the system has a safe state, then it is categorized as *fail safe*.

Development Phase. Here, the *technical SC* elicits technical requirements from the functional SC requirements, which determine how the hardware and software parts should implement the functional SC to achieve the stated SG. The ASIL of each SG determines the set of safety requirements assigned to each part. In this way, the stringency of the design is determined by the properties of the possible hazardous events that the parts may influence. At that point, the hardware and the software parts of the system are developed in accord with the technical SC.

A. Hardware faults in ISO-26262

ISO-26262 provides quantitative techniques for assessing the safety mechanisms, and the residual risk of violating SG.

The hardware development process (see Figure 2) involves: ① determining and planning functional safety activities in the product initiation phase, ② deriving the hardware safety requirements from the technical SC; ③ designing hardware components and ④ their interconnect at architectural level, and each component in detail, factoring safety requirements in them (i.e., with provisions for fault tolerance); ⑤ evaluating the hardware mechanisms designated to handle faults; ⑥ integrating and verifying the hardware architecture against system specification.

Steps ④ and ⑤ include a quantitative analysis of safety mechanisms and residual risk: the hardware architectural metrics defined in step ④ evaluate the effectiveness of the hardware architecture and the implemented safety mechanisms against the fault handling requirements; step ⑤ requires evaluating whether the residual risk of safety goal violations is acceptable (i.e., sufficiently low).

ISO-26262 acknowledges that safety techniques cannot achieve full coverage for all types of faults and allows diagnostic coverages even below 90% for the highest-criticality applications. The system may therefore be exposed to uncovered faults, which results in residual risk that needs to be assessed. Faults can be classified into: safely-ignorable faults (i.e., multiple-point perceived or detected faults) that are regarded as irrelevant since their effects become "visible" before they can do harm, or they are simply harmless; and non safely-ignorable faults (i.e., single-point faults that are not covered by safety mechanisms, residual faults that may escape safety mechanisms and multiple-point latent faults) that are critical, as they may lead to SG violation.

ISO-26262 addresses non safely-ignorable faults by defining the *single-point fault metric* (SPFM) that determines the item's robustness to single-point and residual faults by either design or safety mechanisms, and the *latent fault metric* (LFM) that determines the item's robustness to latent faults by either design or safety mechanisms or driver logic diagnosing the fault before SG violation. The pass/fail reference figures (Table I(a)) defined for these metrics range between 90% and 99% for single-point faults and between 60% and 90% for latent ones, depending on the target ASIL level.

To assess whether the residual risk is acceptable, strict values are imposed on the allowed failure rates. In one of the methods described in ISO-26262, *failure rate classes* (FRC) 1 to 5 are defined with different target rates. Table I(b) describes the maximum FRC for hardware parts depending on the *diagnostic coverage* achieved for the hardware faults and the target ASIL. For instance, an ASIL-D SG requires proving residual failure rate $\leq 10^{-7}$ (FRC 4) when the diagnostic coverage is above 99.9%. Lower failure rates are required

TABLE I: Target values for hardware quantification metrics [27].

ASIL	Single-Point Fault Metric	Latent Fault Metric	Diagnostic Coverage (Residual Faults)			
			≥ 99,9 %	≥ 99 %	≥ 90 %	< 90 %
D	≥ 99 %	≥ 90 %	FRC 4 (10^{-7})	FRC 3 (10^{-8})	FRC 2 (10^{-9})	FRC 1 (10^{-10}) *
C	≥ 97 %	≥ 80 %	FRC 5 (10^{-6})	FRC 4 (10^{-7})	FRC 3 (10^{-8})	FRC 2 (10^{-9}) *
B	≥ 90 %	≥ 60 %	FRC 5 (10^{-6})	FRC 4 (10^{-7})	FRC 3 (10^{-8})	FRC 2 (10^{-9})
(a)			(b)			

(*) + Dedicated Measures

if the diagnostic coverage is lower, with higher failure rates allowed if the ASIL level is lower (e.g., C or B).

Overall, the quantitative assessment of random hardware faults provides evidence of whether the resulting design meets its assigned safety requirements.

B. Software faults in ISO-26262

ISO-26262 holds a deterministic view of software faults and classifies them all as systematic. Moreover, ISO-26262 assumes that all systematic faults have to be prevented, tolerated or removed at some stage of the development process. It is for this reason that their contribution to the residual risk is not contemplated. The software development process is similar to the hardware one (Figure 2), except that it does not include quantitative analysis.

In practice, however, process-oriented solutions cannot provide positive evidence of the lack of residual faults, especially in the face of the increasing complexity of modern software functions, and the intricate interactions that they may have with advanced hardware. Interestingly, some authors [41] argue that the software complexity combined with that of the associated development process cause faults to be randomly scattered across the program code.

Qualitative analysis is meant to prevent faults in the development phase, not to predict their occurrence during operation. For qualitative assessment, software variants exist [38], [35] of state-of-the-art techniques that apply to hardware components, such as Fault Tree Analysis and Failure Mode and Effects Analysis. The main focus at this level would be on process-level issues, to make sure that all discrepancies between program behavior and functional specification are intercepted. Proactive techniques, such as software fault injection or workload generators can be leveraged to further increase the test coverage and reduce the risk of residual faults. All the above techniques, however, suffer from the limitations that the quality of their outcomes depends on the user's ability to achieve sufficient test coverage[2].

The objective of quantitative analysis, instead, is to predict the occurrence of residual faults. ISO-26262 introduces quantitative assessment for random hardware faults, to quantify the risk of residual faults and to determine whether it is below the assigned threshold. Our contention here is that the same should be done for software: means should be provided to reason on the probability of residual software faults (whose presence is bound to stem from the increasing complexity of the system), and to relate that probability to given thresholds.

[2]Note that the analysis of non-functional failures has its own metrics and analysis techniques (including timing and schedulability analysis).

The metric to use to quantify the risk of residual software faults depends on the specific property, either functional or non-functional, for which the risk needs to be quantified. From the functional/implementation standpoint, a lot of effort has been devoted to study and predict the occurrence of software faults as a ground for reasoning on software reliability. Both deterministic or probabilistic models have been proposed. Deterministic models build on characteristics of the program's code (e.g., Halstead's delivered bugs metric [25] or McCabe's cyclomatic complexity [36]) complementary to those suggested by best practice and guidelines for software implementation. Probabilistic models instead relate the occurrence of faults in a function to its frequency of execution or, inversely, to the number of tests executed on it [39]. Probabilistic models generally extrapolate the information collected during the test campaign to predict the occurrence of faults during operation. These models derive reliability predictions from trends observed in failure data. Relevant techniques include Failure Rate, Fault Count models, or the Software Reliability Growth Models [22].

For non-functional properties, such as, e.g., the program's timing behavior, the metrics of interest tend to relate to the test quality and the (test) coverage achieved during development. While a quantitative approach may be needed to assess the residual risk of various types of software faults, in the sequel we focus on execution-time exceedance events, where a software unit exceeds its assigned budget during operation.

III. The Case for Execution-time Exceedance Rates

ISO-26262 requires establishing upper-bounds on the execution time of real-time tasks. The resulting WCET estimates allow deciding how to schedule tasks at run time, thereby assuring the overall feasibility of system's execution. The provided WCET values should be tight, to avoid waste of processor resources. The WCET values should also be consistent with the SG requirements. It is commonly held that any WCET estimate overrun necessarily causes a system-level failure. Yet, this is a misconception since existing safety mechanisms may factor in the execution-time exceedance's impact on the SG, and prevent its escalation into a *timing failure*.

Whereas an execution-time exceedance may not compromise system safety, the timing behavior of software functions should still be characterized to assure proper functioning of the system. It is therefore crucial to assess the quality of the provided WCET estimates to assure that they tightly upper-bound the application's timing behavior under any possible execution scenario. Unfortunately, as noted, the increasing complexity of modern computing platforms threatens the soundness of *qualitative* assessment of timing correctness, and may allow execution-time exceedance situations to escape prevention.

Execution-time exceedance may result, for instance, from the combination of specific task interleaving, initial cache states, interrupt arrival patterns, DRAM refresh operations, whose sources are often too remote from the user reach and too difficult to control and prevent. Accordingly, we contend that execution-time exceedance events should be treated by ISO-26262 similarly to random hardware faults, and the concept

of residual risk should apply to the former too, in conjunction with quantification means as proposed in this paper.

A. Timing analysis challenges on complex systems

With increasingly complex hardware and software, the WCET bounds obtained with traditional means are subject to unquantifiable risk arising from the limitations of the analysis process and the exceeding hardness of the verification procedures. Two main WCET-analysis paradigms have been used so far in industry [45]: static timing analysis (STA) and measurement-based timing analysis (MBTA). Those paradigms and their hybrid variants have been reviewed critically in [7], concluding that, in spite of occasional successes in industrial applications, none of them can be claimed to be effective in the general case and even less so against the relentless increase in complexity of new-generation systems.

While STA is generally held as scientifically sound, confidence in the results of it critically depends on the availability of a detailed and trustworthy timing model of the computing platform underneath the application. Sadly, the latter is increasingly rare, as IP restrictions frequently ban that information off public documentation. Hence, for future complex hardware and software systems, STA may become untenable, as obtaining the information needed for it may become too hard or altogether impossible. Evidence of this trend emerges from recent avionics and automotive reports, where the industrial teams and their STA tool providers have been compelled to resort to measurement-based analysis to derive timing bounds for multicore processor architectures like the NXP P4080 [37], Texas Instrument TMS320C6678 [33], and ARM-based SABRE Lite [13]. Industrial pragmatism, therefore, continues to regard MBTA as the most practicable timing analysis approach even for safety-related real-time systems, which explains STA's weaker penetration [45].

MBTA requires identifying the main sources of execution-time jitter, to activate them during analysis. While being far from trivial, this identification is a much easier job than building or acquiring the detailed timing model required by STA, and can be performed by first reviewing processor specifications to identify those resources and then using specialized programs called micro-kernels [37][40] that place a predetermined load on the desired processor resource(s) to quantify their impact on timing. For MBTA, uncertainty stems from the inherent difficulty in mimicking, during analysis, all of the execution conditions – especially those of jittery processor resources – that can arise during operation. Deriving reliable WCET estimates on complex hardware requires that low-level architectural features, which can contribute to significant execution-time variations (e.g., cache placement), are factored in the measurement runs taken during analysis so that the observed execution times can be considered *representative* of those that can arise during operation. As complex hardware architectures may have a huge number of potential states with bearing on execution-time jitter, it is not realistically possible to fully explore them during analysis. Hence, by construction, MBTA cannot exclude that residual execution-time exceedance events may occur during operation, reflecting circumstances not covered during analysis.

Common industrial practice to address uncertainties in WCET analysis requires adding conservative safety margins (often starting at 20%) to the computed WCET value. Any such number however evidently lacks scientific grounding and simply rests on engineering judgment. Consequently, this practice may yield either ineffective use of the available resources (due to WCET over-estimation) or higher risk of execution-time exceedance events (owing to WCET under-estimation), as a result of insufficient quality in the computed bound. Moreover, this practice does not scale to more complex hardware and software. Already on a relatively simple 4-core processor, in fact, small variations in execution conditions have been shown to cause either tiny (e.g., below 10%) or huge slowdowns (e.g., up to 20x) [23]. Appropriate means are therefore needed to produce tight WCET estimates that can be related to a quantified (and arbitrarily low) risk of execution-time exceedance.

B. Probabilistic WCET distribution

To address this challenge, we build on timing analysis solutions that yield probabilistic distributions of the execution-time behavior of application tasks (nicked probabilistic WCET, pWCET), instead of a single-valued WCET. The pWCET distribution, illustrated in the right side of Figure 3, represents the probability that a task may exceed the assigned budget envelope at run time. Cutting the tail of it at the desired probability of exceedance (10^{-10} on the Y axis, per run or hour of operation) projects onto an execution-time value (7 on the X axis) that may serve as the WCET budget at that level of assurance. Hence, the pWCET provides means to statistically quantify the likelihood of execution-time exceedance accurately.

C. Fitting pWCET into the Safety Life Cycle

Interestingly, the notion of pWCET distribution follows ISO-26262's philosophy for the handling of random hardware faults and applies it to the timing domain. Returning to the ISO-26262 life cycle depicted in Figure 2, with focus on timing-related requirements, we now describe how an approach delivering a pWCET curve can fit in the software development process defined in the standard, which we illustrate in Figure 3.

In the *concept phase* (not shown in Figure 3), the functional SC should be extended to also consider the possibility of execution-time exceedance events that can propagate into timing failures and, accordingly, define adequate safety protection measures against them (e.g., watchdog timer).

In the software *development phase*, ISO-26262 includes timing-related requirements in three different phases of the software V-model (sketched in Figure 3). First, during software safety specification phase ①, it requires system designers to specify the time budgets of critical software. Then, the software architectural design ② shall consider the time upper-bounds to dimension the system. If an approach delivering a pWCET distribution instead of a single-valued WCET is used, the designer needs to identify the appropriate probability of exceedance ④ to determine the corresponding WCET from

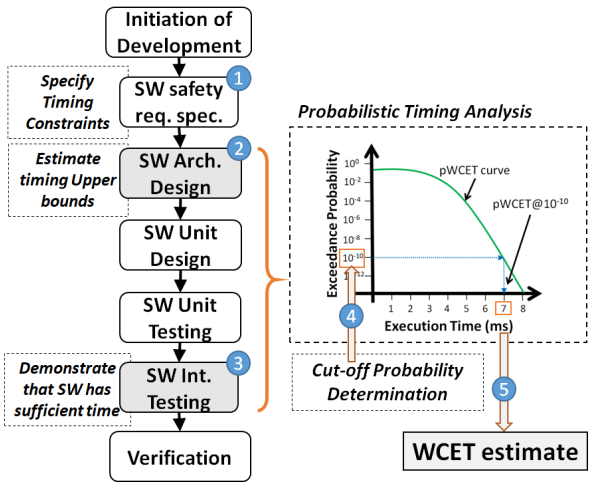


Fig. 3: Sketch of how pWCET fits in ISO-26262 software development process

the pWCET curve ⑤. To this end, the cut-off exceedance probability (or allowed execution-time exceedance rate) shall be evaluated together with the diagnostic coverage for timing errors and the ASIL of the SG. In other words, the standard should provide target metrics for the combination of these three factors as done for random hardware faults in Table I.

For integration testing ③, ISO-26262 requires providing evidence that the software is allocated enough time to complete its functionality. The pWCET distribution allows associating the assigned budget envelope to the corresponding probability of exceedance.

This approach advocates abandoning the current practice of adding a safety margin to the WCET estimate and assuming – on expert judgment only – that it will never be exceeded, and therefore exposing to an unquantified risk of execution-time exceedance. In contrast with that, the pWCET improves the soundness of the verification process by providing a quantitative upper-bound to the risk of execution-time exceedance estimated with a sound approach. To this end, however, it is of vital importance that the timing analysis technique meets the property of guaranteeing that the delivered pWCET distribution is representative of the worst-case timing behavior that may occur during operation.

D. Software failure rate classes and diagnosis coverage

We now describe failure rate classes (FRC) and diagnostic coverage for execution-time exceedance events, so that they can be used as for hardware random faults.

Failure Rate Classes. The pWCET distribution allows selecting the acceptable rate of execution-time exceedance, normally associated with *a single run* of the task. By multiplying this value by the task's execution frequency per hour, we determine the execution-time exceedance rate per hour for the task. For instance, in order to assure an execution-time exceedance rate per hour of, e.g., 10^{-9}, for a program executed 10^3 times per hour, the user should cut the pWCET tail at the 10^{-12} exceedance threshold, which would yield a 7.7 ms WCET value in Figure 3. In this way, it is probabilistically guaranteed that the accumulated execution-time exceedance rate of all instances of the program executed per hour is below 10^{-9}. This reasoning matches random hardware metrics as defined in Table I. Similarly to the hardware case, the particular probability to choose comes from the ASIL level assigned to the software element.

Diagnostic Coverage. The standard suggests the usage of watchdog timers to detect the consequences that a fault in a hardware component may have in the program schedule (e.g., missed, delayed, or too close activations of the program). In this scenario, the standards of interest categorize the diagnostic coverage achievable by watchdogs for errors in the control logic of processing units as either low (60%) or medium (90%). Accordingly, watchdogs can also detect (possibly with a high, >99%, diagnostic coverage) execution-time exceedance events in the operational system. On the occurrence of such an event, the safety mechanisms in place detect the error and instigate action to remove the residual risk of SG violation. While advising the usage of an external monitoring facility (e.g., watchdog) for error detection at the software architectural level (which correlates to software faults categorized as systematic), ISO-26262 does not explicitly allude to the achievable diagnostic coverage of mechanisms against execution-time exceedance.

For fail-safe systems, the system should be moved to a safe state every time a diagnostic mechanism detects an execution-time exceedance. In that manner, the SG would be preserved at the expense of making some functionality (or the entire system) unavailable. As a result, the degree of diagnostic coverage that the safety mechanisms provide for timing faults should be taken into account when quantifying the residual failure rate (as it is the case for hardware, see Table I(b)). Whereas safety is not affected in fail-safe systems in the event of an execution-time exceedance (assuming that high diagnostic coverage mechanisms are in place), system availability is instrumental for the end user since an unavailable system does not deliver the expected functionality. Arguably, therefore, solutions that yield reliable pWCET distributions can improve the design process by allowing the user to assess the availability of fail-safe systems from a timing perspective.

For fail-operational systems, which need to stay operational to preserve safety, the event of an execution-time exceedance should activate the use of appropriate forms of redundancy or diversity, so that the occasional failure of one unit does not stop the (safe) operation of the entire system. Whenever this solution is not possible, having high diagnostic coverage against timing faults is not sufficient to preserve safety and a sufficiently low cut-off probability needs to be chosen to ensure that the contribution of execution-time exceedance to the residual risk is kept correspondingly low.

E. From ISO-26262 to IEC-61508

The IEC-61508 meta- (or parent-) standard differs from ISO-26262 only slightly. The latter refines some definitions for the life-cycle phases and provides additional requirements for the safety requirement specification of the hardware. IEC-61508 does not organize the life cycle around concept and

development phases explicitly, but rather fragments it into smaller units that match the activities defined within the ISO-26262 workflow. Those activities include, for instance, the *Hazard and Risk Analysis* (which ISO-26262 names *Hazard Analysis and Risk Assessment*) and the *Overall safety requirements* and *Allocation* (which ISO-26262 places in the *Functional Safety Concept*), where the SIL level, ranging 1 to 4, is computed. As a rule of thumb, the highest ASIL level in ISO-26262 (ASIL-D) matches on certification ambition a SIL-3 in IEC-61508. The safety requirement specification concerning random hardware faults is lighter in IEC-61508 than in ISO-26262. The hardware concept and development involve the same steps in the meta-standard, but the derivation of hardware faults neither includes latent faults nor multiple-point faults, which simplifies the calculations. Regarding software faults, the approach is identical in both standards: software faults are considered systematic and qualitative measures are recommended for fault avoidance, such us WCET analysis to assure temporal independence among software elements.

Like ISO-26262, IEC-61508 determines the requirements for avoiding or controlling systematic faults based on expert judgment from practical experience. IEC-61508 states that "*the probability of occurrence of systematic faults cannot in general be quantified*". To exemplify this difficulty, IEC-61508 reasoning observes that the effects of systematic faults manifesting at run time, depend on the moment of the life cycle in which they were introduced, and the effectiveness of the prevention measures (e.g., structured programming) in place, which are both difficult to quantify sensibly. However, IEC-61508 allows considering that the target failure reduction for a safety function is achieved by demonstrating compliance to all requirements of the standard. In this regard, the standard introduces the concept of *Systematic Capability*, which is equivalent to the SIL, but only considers systematic faults. In addition to systematic fault reduction or prevention in the design, the standard does also define mechanisms to control the run-time errors arising from systematic faults (e.g., diverse software redundancy).

Overall, IEC-61508 retains the notion of random hardware faults and proposes a qualitative approach for software faults that may not scale well against increasingly complex systems. Arguably, therefore, all the application domains covered by the IEC-61508 umbrella might equally benefit from incorporating an execution-time exceedance quantification approach, much like the automotive domain would do via ISO-26262 following the solution presented in this paper.

IV. MBPTA: CONCEPT AND APPLICATION

As a particular probabilistic timing analysis solution, we build on the MBTA variant proposed in [8], [30], [29], called Measurement-Based *Probabilistic* Timing Analysis (MBPTA). MBPTA yields a reliable pWCET distribution while guaranteeing, by construction, that the delivered pWCET is an upper-bound of the execution conditions that may occur at system operation: it therefore fits the ISO-26262 exceedance rate quantification approach presented in Section III.

MBPTA acknowledges that the control that the user can exercise on the application's timing behavior during analysis necessarily leverages high-level metrics such as software code coverage, but has increasingly less means to address low-level hardware aspects (e.g., bus occupancies, placement of program's code/data in cache) comprehensively. Hence, MBPTA relieves the user from the latter burden by introducing some platform modifications.

The application of MBPTA rests on the premise that the computing platforms that enable its use [30], [31] modify the timing behavior of selected jittery resources so that the execution-time measurements collected during analysis either match or upper-bound probabilistically the timing behavior that may occur during operation. In that manner, the obtained pWCET distribution is warranted to capture any extreme behavior that may occur at operation, and it is produced without burdening the user with the need to comprehend all system states relevant to execution-time analysis.

If hardware support is provided to enable the use of MBPTA, the processor vendor is the party in charge of singling out jittery resources and of designing MBPTA-compliance around them appropriately. Interestingly, using MBPTA, the processor vendor would not need to build a timing model of its processor, or granting access to all details of the hardware design as STA requires. All it would be required of the vendor is to design processor resources that can be explicitly and individually configured to feature the desired forms of MBPTA conformance, and to document them in public user manuals.

Conversely, if hardware support for MBPTA were scarce or inexistent, the user would have to identify the sources of execution-time variation building on the processor specifications, and apply software solutions to reach MBPTA compliance. In general, the resources that cause the largest jitter (e.g., cache memories, interconnection networks, memory controllers) are easy to identify. Missing out some sources of jitter, while not desirable, is not particularly harmful as long as the jitter that they may produce is not larger than the cumulative effect of the execution-time variation produced by the other known sources.

To handle jittery resources, MBPTA defines two main techniques, implemented in either hardware [30] or software [31], which we present below.

A. Time upper-bounding

This technique forces selected jittery hardware resources to work at their highest latency during analysis. In that manner, the operation conditions cannot lead to higher execution times from them and hence, a single run suffices to capture their worst-case operation-time behavior. The hardware resources that best fit the use of this technique are those whose extended variation in timing behavior depends on elements that the hardware cannot discriminate efficiently [18], [26].

The Floating Point Unit (FPU) provides an illustrative example of this kind of resources. The latency of FP operations depends on the operands, outside of the hardware's own control. For instance, multiplying any value by 0.0 may incur shorter latency than multiplying any pair of not-null parameters. Hence, for the analysis of even the simplest sequential program that included FP operations, capturing the full extent

of latencies that it might incur would require enumerating all of the executed FP operations and their respective operands, which is unduly onerous and likely to involve laborious debugging. On top of that, the user would also need to determine whether the distribution of the FP operations and operands observed during analysis is representative of what may occur at operation, which is even harder, if at all possible. Instead, MBPTA's prescription to force the FP unit to work at its highest latency (per operation type) during analysis relieves the user from the burden of controlling the impact that each FP operation incurs on program execution time.

Original FP units allow serving the result and releasing as soon as the current operation finalizes. To implement the said technique, the hardware default is modified by deactivating the immediate-release check, so that all operations take maximum latency regardless of the input operands. The hardware feature that allows enforcing the highest latency can be enabled or disabled by setting the corresponding configuration register accordingly, so that it can be kept enabled during analysis and disabled during operation. In that manner, operation-time behavior may experience shorter, but never longer, latency.

Time upper-bounding also applies to other resources such as, e.g., the number of arbitrated contenders on the shared bus that connect cores to a shared L2 cache [18], [26]. For a program running on a core, the contentions suffered during operation depend on the software being run on the other cores. This information is exceedingly difficult to determine during analysis even for the strictest of static scheduling scenarios, since the arrival time of bus requests from contenders may change across different execution paths and cache hit/miss patterns. To address this challenge, a simple modification to the hardware arbiter is applied [26] to cause arbitration to occur across all potential contenders regardless of whether they have pending requests or not, keeping the bus busy for the longest request latency after selection. Selectively disabling this feature during operation allows the program to experience fewer stalls than contemplated for WCET analysis.

B. Time randomization

This technique causes the response time of some jittery resources to exhibit a probabilistic behavior that also holds during operation. Accordingly, a representative distribution of the impact that jittery resources may cause on execution time can emerge after a statistically-significant number of observation runs. For instance, randomizing the placement and replacement of objects in cache memories, allows using execution-time measurements to model cache behavior probabilistically. Such randomization makes cache conflicts independent of the memory location of program objects, which relieves the user from the need to control memory placement. In Integrated Modular Architecture systems as used in avionics [5] and automotive [12], individual software applications are often subcontracted to different providers. As a result, the integration of the system progresses incrementally, requiring to assess at every step of integration that the new build conforms with the specification, for functional and non-functional requirements. However, as applications get integrated into the system binary, their memory placement and cache layout may vary [21], invalidating the WCET estimates computed previously. This phenomenon defers timing verification to the latest stages of integration, where the (binary) image is near final, which in turn makes timing faults much more costly to handle than during earlier phases of development.

Randomized caches mimic the behavior of multiple software integrations, which allows the WCET estimates computed in earlier development stages to hold across the whole process of integration as well as during operation.

To date, time randomization has been implemented in processor resources whose timing behavior depends on the structural dependencies created by the hardware design. Randomization helps remove those dependencies, which have no bearing on the program semantics. For instance, whether two addresses compete for the same cache space depends on how they are mapped to cache lines. And cache mapping can be randomized to make conflicts occur probabilistically. To ensure that the observations made during analysis represent (probabilistically) the timing events that may occur during operation, such randomization must be kept enabled at all times, with no distinction between analysis and operation. Random placement and random replacement have been successfully implemented in hardware [18], [26].

This technique is evidently superior to the "time upper-bounding" alternative of modifying the cache hardware to respond with the highest (miss) latency during analysis, owing to the massive performance decay incurred by the latter.

With little difficulty, time randomization has also been applied to the bus arbiter, changing the way it chooses which core is granted access to the shared L2 cache. Bus arbiters therefore have two types of modifications: time upper-bounding to determine the number of contenders and the latency with which the bus is released; and time randomization to choose which core is granted access to the shared L2 cache.

All hardware parts that use time randomization require a hardware source of randomness. An exemplary Pseudo-Random Number Generator (PRNG) has been implemented to that end [9], with a degree of randomness that has passed the most stringent cryptographic tests. The cited publication shows that the hardware cost of its implementation is low, also because a single PRNG can be shared across multiple resources. The PRNG has also been proven compatible with high safety integrity levels. If time randomization is to be implemented in software, a software implementation of the very same PRNG algorithm can also be used.

C. Probabilistic analysis

The execution time of the program 'inflated' by time upper-bounding and randomization results in an analysis-time distribution (ATD) that upper-bounds the operation-time distribution (OTD) by construction. Figure 4 illustrates this notion. The dotted line depicts the empirical complementary cumulative distribution (ECCDF) of the OTD, and the dashed line the ECCDF of the ATD. A sound use of *probabilistic analysis* (such as, e.g., Extreme Value Theory, EVT) uses a sample of ATD values – no less than a hundred, and typically

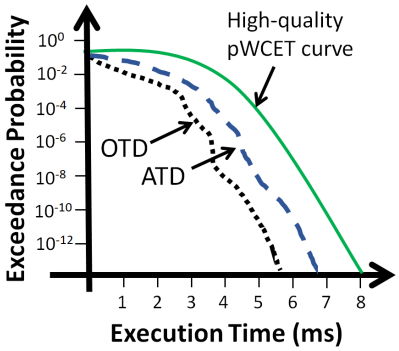


Fig. 4: Example results of MBPTA application

up to two thousands, which keeps the MBPTA overhead low – to derive a high-quality pWCET distribution that upper-bounds the ATD (and hence the OTD) [8]. For EVT to be applicable, the observed execution times must correspond to independent and identically distributed (i.i.d.) random variables, which means that each measurement observation must belong to the same execution-time distribution. Satisfying this requirement has been proven doable with simple-enough procedures [14].

The MBPTA process collects execution-time samples from the ATD, earning MBPTA conformance thanks to the hardware modifications discussed earlier, and to a measurement collection process that controls the initial conditions of the experiment [14]. The analysis procedure may determine that the sample fails to meet the eligibility criteria for the application of EVT or detect that it cannot be upper-bounded by exponential tail distributions, which is required to ensure tightness. These situations are addressed by enlarging the sample size. It is known, in fact, that increasingly larger samples from an i.i.d. random variable with a guarantee finite bound will eventually be proven statistically i.i.d., and also converge, more tightly, to either exponential or light tails, the former always upper-bounding the latter. In our analogy, the program's execution-time observations are the random outcomes of that variable, and the program itself has bounded duration in conformance with well-established real-time coding practice. At that point, the larger the sample size, the tighter the pWCET. Accordingly, MBPTA users should collect large – yet affordable – samples, below 2,000 measurements on average [8].

MBPTA promotes a paradigm shift with respect to traditional, deterministic (i.e., single-valued) WCET analysis. The relation of MBPTA with its deterministic counterpart is straightforward: MBPTA's main constituents (*time upper-bounding* and *randomization*) specifically address the representativity concerns that afflict standard measurement-based approaches, and threaten to become insurmountable with increasingly complex systems. Relating MBPTA to STA is much harder instead, as those two techniques build on largely different (and mostly incompatible) assumptions [6]. The correctness and the precision of either of them depend on whether and to what extent their assumptions are guaranteed to hold. See [7] for a detailed analysis of those assumptions and how they relate to hardware and software complexity.

D. MBPTA: industrial viability

MBPTA's viability for industrial use in safety-related systems relates to the cost of the required hardware or software changes, and how the approach can be fitted in the overall ISO-26262 safety life cycle as discussed in Section III-C.

The latter question leverages the need to step up the guidelines of current safety standards to increasing complexity of new-generation processors. This has been done, for instance, in the avionics domain, where CAST32 [15] and the accompanying CAST-32A [16] address the use of multicore processors. Arguably, this game-changing scenario should ease the task of incorporating MBPTA related changes.

The MBPTA requirements on the computing platform, if implemented at hardware level, have been shown affordable, first by implementation in architectural simulators, then at RTL level in FPGA, and finally in off-the-shelf products [18]. Implementing randomization has been surprisingly non-intrusive. We illustrate this for two cases. Bus protocols like AMBA [11] (one of the most, if not the most, used), do not define any particular arbitration policy. This situation allows adding random arbitration policies with no impact on the protocol specification. The same happens for cache placement and replacement. While the latter is already supported in many processors, adding the former requires combining the address being accessed with a hardware- (or software-) generated random seed [9], changed across runs, to map the address to a random cache set. This change causes the timing behavior of cache conflict scenarios that are probabilistically relevant – those whose timing behavior can only be exceeded with negligible probability – to be close to average behavior which, in turn, is very close to the typical behavior on conventional hardware designs.

At software level, randomization has been implemented as a pass in the LLVM compiler [29] or as a source-to-source translator developed in an approach called TASA [31]. Both solutions leverage the fact that the way in which functions and data (locals and globals) are placed in the source code and the binary determines their address in memory. By randomly allocating them and adding padding space among them keeps the program functionality unchanged and attains similar randomized timing to that obtained with hardware-implemented random placement. As opposed to the hardware and LLVM-based software solutions, which attain randomization at program run granularity, the TASA approach applies randomization on a per-binary basis. As a result, the probability of exceedance determined by the use of TASA is equivalent to the execution-time exceedance probability of all systems with the same randomly-generated binary. For the hardware and LLVM-based software randomization cases instead, the obtained probability is per run of the program, and therefore has to be multiplied by its rate. Time upper-bounding at software level is managed off-line, by monitoring relevant events during the analysis-time measurements (through Performance Monitoring Counters, PMC) and by padding execution-time observations so that their impact on the program's execution time is deterministically upper-bounded. For instance, reading a PMC that returns the quantity of FP operations executed by

a program, allows computing a padding equivalent to each FP operation taking the highest latency. Similarly, bus jitter can be deterministically upper-bounded by monitoring the number of bus access requests with PMCs and applying a contention model that assumes worst-case overlap among them [20].

Both hardware and software randomization and upper-bounding solutions have proven effective for performance in various platforms and with negligible implementation costs: ≈1% additional hardware for a 4-core processor for the space domain [26], and a pre-process compiler step for TASA source-to-source transformations in the automotive domain [31]. Moreover, pWCET estimates have been shown to be typically within 20% of the HWM on conventional (time-deterministic) platforms used as reference for industrial applications in the space, avionics and automotive domains [19], [24], [28]. This evidence proves that time randomization and upper-bounding do not incur untenable pessimism.

V. Experimental Support Evidence

To sustain our contention, we discuss an exemplary application of MBPTA in an automotive case study targeting the AURIX TC277 [43]. We show that, even on a processor whose hardware design expressly seeks maximum determinism, the execution-time behaviour of applications running on it suffers jitter created by resources that may be hard, if at all possible, for the user to control. We do *not* aim to present a full WCET analysis method for the TC277: our intent is just to show that the execution-time jitter of hard-to-predict resources like the cache – a definite and massive asset of future automotive processors [2][4][3][1] – can be handled with MBPTA.

A. Application case

The AURIX TC277 comprises three cores (plus two additional ones that operate in lockstep mode): one energy-efficient core and two performance-efficient ones. We focus on the latter, which embed high-performance jittery resources such as caches and dynamic branch predictors. All cores are equipped with local scratchpad memories and caches, for both instruction and data, and are connected via a crossbar to a common 'memory system' comprising a shared SRAM, and program/data Flash memories.

The application we consider is an Automotive Cruise Control System (ACCS), whose functional code was automatically generated from a Simulink model, and a CONCERTO[3] model for its architectural specification. The application was run on a customized version of ERIKA Enterprise[4], which implements an OSEK/VDX compliant personality. The originating Simulink model comprised in excess of 200 blocks, which corresponded to about 3,000 lines of C code. After a transformation process that flattened the Simulink block hierarchy, optimally re-grouped the blocks by compatible rates, and re-generated source code accordingly, the application was embedded in three real-time ERIKA tasks: (i) Signal Acquisition, (ii) Monitoring, and (iii) Speed Controller, as shown in Figure 5.

[3]CONCERTO, ARTEMIS JU, http://www.concerto-project.org/.

[4]Erika Enterprise RTOS, http://erika.tuxfamily.org/drupal/.

A fourth task, Status Update, was added to the application to close the simulation loop in the experiment by stubbing and interconnecting all input and output ports.

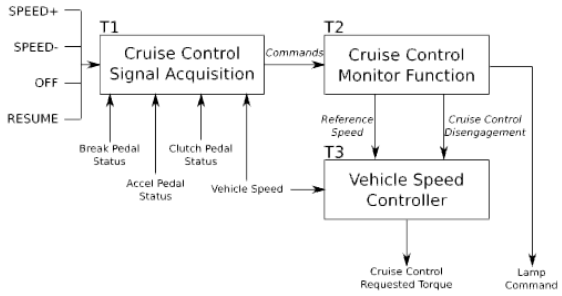


Fig. 5: Block diagram of the case-study application.

For the purposes of this paper, we discuss the impact of the instruction cache layout on the application's execution-time behavior, to showcase its relevance as a source of jitter, and demonstrate how MBPTA can capture its contribution in the analysis process. To this end, we deployed the application on the processor such that part of the code was stored (and cached from) the program Flash memory segment. Private stack and data were located in the local scratchpads while local shared data was mapped to the shared SRAM. The instruction cache size in the performance-efficient cores is 16KB, with 32B cache lines. Other sources of variability and combinations thereof are not considered here. How to jointly account for them is discussed in [20].

B. Instruction cache jitter

To analyze the impact of cache layout on execution-time jitter, we used TASA, a technique that applies software randomization to off-the-shelf caches to make their response time probabilistically analysable. As explained in Section IV, TASA performs source-to-source program code transformations where the relative location of functions, stack frames, and global variables is randomized by reordering and padding the source file so that each resultant binary incurs a randomly different cache placement. Thus, by studying the timing behavior of a statistically significant number of binaries, the impact of caches can be accounted for probabilistically. In our experiments, we generated 1,000 distinct binaries with TASA, all with identical functionality, and each with different stack and global data allocations to memory positions. Simple ad-hoc scripts were needed to automate this process: they invoke the TASA pre-process pass and compile the output of it to produce one binary; this process is repeated as many times as needed, varying the random seed so that the required number of binaries is obtained. Each such binary needs to be run once for the purposes of MBPTA. The computational cost of this process is proportional to two characteristics of the software being considered: (1) its size and complexity, and (2) its execution time. The former determines the cost of generating the binary, largely dominated by compile time, much more complex than the TASA pre-processing step. The latter determines the cost to execute each binary. In our particular

case, generating the binary and executing it took around 5 seconds altogether (the most part for the compilation), which serialized in less than 1.5 hours for all binaries. Of course, binary generation and execution can be parallelized in multiple instances: as the process is fully independent per binary, the turn-around time would decrease roughly linearly with the degree of parallelization. The remainder of the MBPTA process (acquiring the collected execution times, and producing the pWCET distribution) takes just a few seconds.

While the cost of this process for a single program is rather low, it would increase linearly for applications that include multiple programs assigned to a criticality level that requires evidence of bounded execution-time. However, each such program could be analyzed separately, thus allowing the analysis process to proceed independently (and perhaps with internal parallelism). In general, devoting around 1-2 hours of computational cost (less if parallelized) to the timing analysis of each program, with minimal user intervention, should be affordable even for complex applications.

Figure 6 reports the execution-time variability observed for the four application tasks – for the same program path –, as determined by the different randomly-generated program layouts. For the system under analysis, the observed variability, which may incorporate the effects of other sources of execution-time jitter, ranged up to approximately 5%. In all cases, the HWM was quite distant from the observed average and mode[5]. The impact of time upper-bounded jittery resources, computed off-line based on PMC measurements, should then be added to these execution-time observations before obtaining the pWCET distribution. This kind of variability is not explored by state-of-the-art WCET analysis procedures, measurement-based and static alike. Even more critical is the fact that traditional measurement-based techniques ***do not support constructing arguments*** on whether and to what extent the effect of jittery resources has been captured at analysis time. Next, we show how MBPTA can consider these effects in the determination of pWCET bounds.

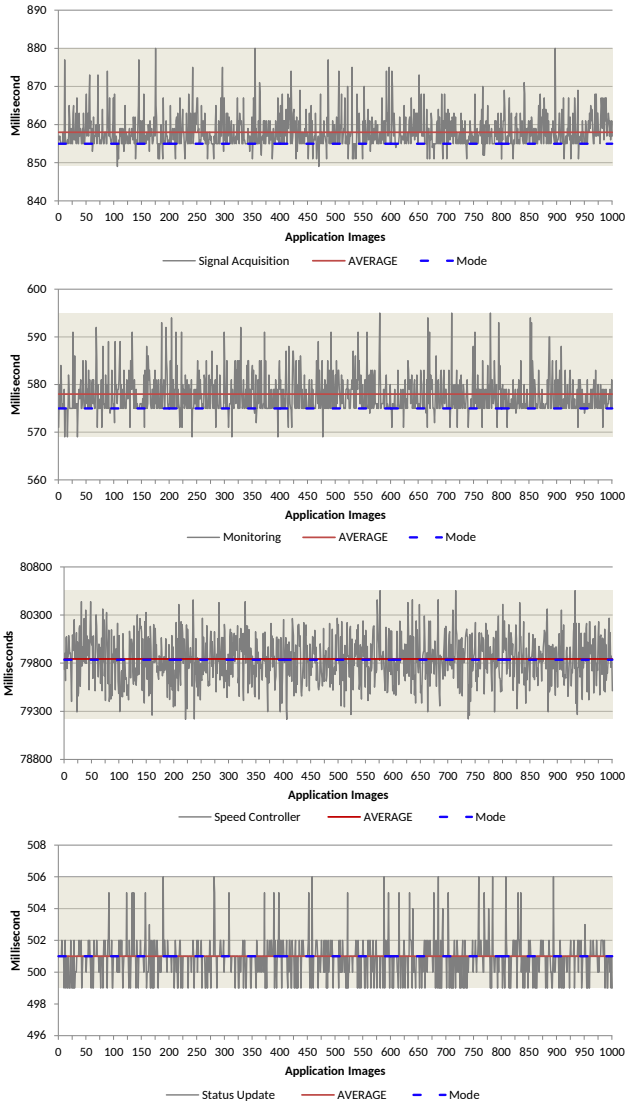


Fig. 6: Uncontrolled variability induced by program layout.

C. Application of MBPTA

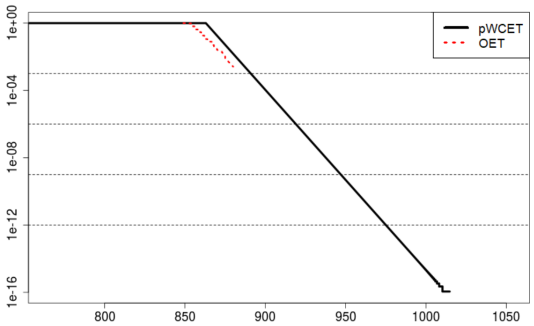
We applied MBPTA to the four tasks of the case-study application. According to the TASA prescriptions, we collected timing measurements for each such task by executing the same set of 1,000 randomly-generated binaries of the application. The collected observations successfully passed the statistical i.i.d. tests (a pre-condition to apply statistical analysis), which allowed using them as input to the subsequent probabilistic analysis process. To the latter end, we used the MBPTA-CV [8] method, which applies Extreme Value Theory [32] by automatically selecting the distribution parameters that best fit the maxima of the observed execution times. In all cases, 1,000 measurements were sufficient for MBPTA to converge: adding additional observations for each application would *not* change the resulting pWCET distributions shown in Figure 7.

The red dotted line in Figure 7 plots the observed execution times (OET), in the form of Complementary Cumulative Distribution Function (CCDF), to show that the pWCET curves (solid black lines) always tightly upper-bound the observed data. The pWCET bounds for the analyzed functions at relevant exceedance thresholds are reported against the HWM in Table II. The application of MBPTA-CV to the automotive functions led to extremely tight results as, when compared to their respective HWM, the predicted pWCET bounds are always below the reference 20% margin. The low distances for higher exceedance thresholds can be partially ascribed to the overall high predictability of the execution platform.

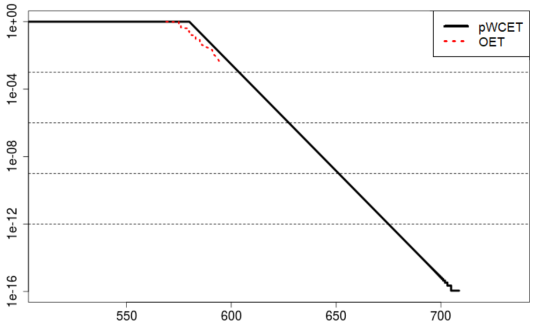
An intuitive but wrong conclusion here might be that the 20% margin is a reliable figure in the general case. In fact, our experiments show that, limited to the processor considered in this paper, and focusing only on the instruction cache, a conservative margin at 20% would be conservatively pessimistic and therefore sound. Yet, for other processor architectures, with complexity similar to technology used in the automotive domain [2], [4], [3], [1], that margin would be optimistic instead, hence unsound [24].

In actual fact, the slope of the pWCET distribution, hence the margin above the highest observed value for the acceptable exceedance probability, depends on the particular

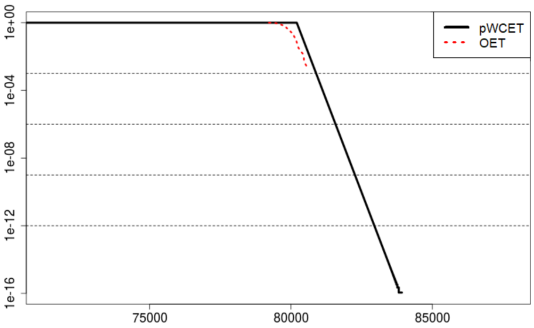
[5]The mode of a data sample is the most frequent value.



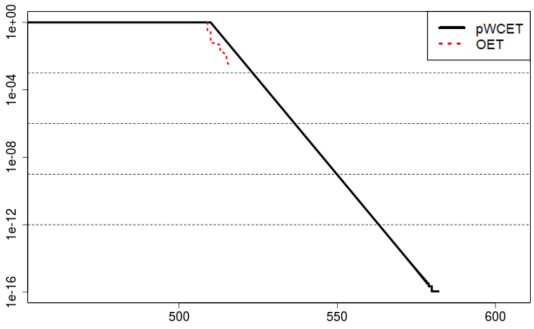
(a) Signal Acquisition



(b) Monitoring



(c) Speed Control



(d) Update Status

Fig. 7: The pWCET distributions computed by MBPTA for the four application tasks.

characteristics of the program under analysis and how it uses the underlying processor hardware. Such margin (or the pWCET value itself) is guaranteed not to be exceeded with a particular probability (e.g., 10^{-12} per run). That would be the exceedance probability if *all* the sources of jitter that have been upper-bounded, always caused their highest latency. This does not happen in the general case; how often it may, cannot be told beforehand as it depends on the input-dependent behavior of the application during operation. In this situation, the MBPTA method allows the user to strictly upper-bound the residual risk of execution-time exceedance. Conversely, time-deterministic approaches building on a margin set on expert judgment for a particular platform, hardly scale to other (arbitrarily complex) platforms and also do not provide means to assess the residual risk.

TABLE II: pWCET bounds at relevant exceedance thresholds (in processor cycles).

Test	HWM	10^{-3}	%	10^{-6}	%	10^{-9}	%	10^{-12}	%
`Signal Acq.`	880	891	**1.2**	919	**4.4**	947	**7.6**	975	**10.8**
`Monitoring`	595	603	**1.3**	627	**5.4**	651	**9.4**	674	**13.3**
`Speed Control`	80554	80888	**0.4**	81574	**1.3**	82260	**2.1**	82946	**3.0**
`Status Update`	506	511	**1.0**	521	**3.0**	531	**4.9**	541	**6.9**

VI. Conclusions

ISO-26262 classifies hardware faults as either systematic or random, while it considers all software faults to be systematic. The unrelented demand for newer value-added functionalities for computer-based systems requires the use of increasingly complex hardware and software. This trend challenges the viability of exhaustive analysis and prevention for all types of systematic faults as prescribed by the standard. This threat is especially true for the timing behavior of software applications, as the fabric of new systems denies users the ability to capture all sources of execution-time variations and to create the test scenarios needed to estimate the residual risk of failure. Recent timing analysis techniques that deliver WCET estimates with an associated probability of exceedance have the potential to overcome this limitation. However, how the quantification of the likelihood of execution time exceedance events fits the scope and intent of safety standards such as ISO-26262 is still an open research question. In this paper, we address this question by proposing ISO-26262 adaptations to assess the residual risk associated to exceeding the timing budget assigned to a software program, in analogy to what is done for random hardware faults. This approach relies on MBPTA, which delivers a probabilistic WCET bound that serves the purpose of upper-bounding residual risk. We exemplify this approach with a particular incarnation of MBPTA, which transparently applies time randomization to selected hardware or software elements of the computing platform, in this manner relieving the user from the burden of controlling the impact of low-level hardware elements on software execution time. This proposal is presented in the context of the ISO-26262 software development process and the treatment of random hardware faults in the safety life cycle, with the intent of promoting the acceptance of execution-time exceedance rate quantification in the standard.

Acknowledgment

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence. Jaume Abella has been partially supported by the Ministry of Economy and

Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717. Carles Hernández is jointly funded by the Spanish Ministry of Economy and Competitiveness and FEDER funds through grant TIN2014-60404-JIN. Enrico Mezzetti has been partially supported by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva-Incorporación postdoctoral fellowship number IJCI-2016-27396. This work used proceeds of the CONCERTO project (ARTEMIS-JU grant nr. 333053), which we gratefully acknowledge: Intecs SpA, lead of CONCERTO, provided the sources of the automotive application, and the University of Padova the build automation for the AURIX target.

REFERENCES

[1] Intel GO Automated Driving Solution Product Brief. https://www.intel.es/content/dam/www/public/us/en/documents/platform-briefs/go-automated-accelerated-product-brief.pdf.

[2] NVIDIA DRIVE PX. Scalable supercomputer for autonomous driving. http://www.nvidia.com/object/drive-px.html.

[3] QUALCOMM Snapdragon 820 Automotive Processor. https://www.qualcomm.com/products/snapdragon/processors/820-automotive.

[4] RENESAS R-Car H3. https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html.

[5] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.

[6] J. Abella, D. Hardy, I. Puaut, E. Quiñones, and F. J. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[7] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-Askasua, J. Perez, E. Mezzetti, and T. Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015.

[8] J. Abella, M. Padilla, J. Del Castillo, and F. J. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4):72:1–72:29, June 2017.

[9] I. Agirre, M. Azkarate-askasua, C. Hernandez, J. Abella, J. Perez, T. Vardanega, and F. J. Cazorla. IEC-61508 SIL 3 Compliant Pseudo-Random Number Generators for Probabilistic Timing Analysis. In *Euromicro Conference on Digital System Design (DSD)*, 2015.

[10] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php, 2015.

[11] ARM Ltd. AMBA open specifications. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[12] AUTOSAR. *Technical Overview V2.0.1*, 2006.

[13] A. Blin, C. Courtaud, J. Sopena, J. Lawall, and G. Muller. Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System. In *28th ECRTS*, 2016.

[14] F. J. Cazorla, T. Vardanega, E. Quiñones, and J. Abella. Upper-bounding Program Execution Time with Extreme Value Theory. In *WCET Workshop*, 2013.

[15] Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical report, CAST-32, May 2014.

[16] Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical report, CAST-32A, November 2016.

[17] R.N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.

[18] COBHAM. LEON3 Processor. Probabilistic platform. http://www.gaisler.com/index.php/products/processors/leon3.

[19] F. Cros, L. Kosmidis, F. Wartel, D. Morales, J. Abella, I. Broster, and F. J. Cazorla. Dynamic software randomisation: Lessons learned from an aerospace case study. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 103–108, March 2017.

[20] Enrique Díaz, Mikel Fernández, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. *MC2: Multicore and Cache Analysis via Deterministic and Probabilistic Jitter Bounding*, pages 102–118. Springer International Publishing, 2017.

[21] E. Mezzetti and T. Vardanega. A rapid cache-aware procedure positioning optimization to favor incremental development. In *RTAS*, 2013.

[22] William Farr. Handbook of software reliability engineering. chapter Software Reliability Modeling Survey, pages 71–117. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[23] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla. Assessing the suitability of the NGMP multi-core processor in the space domain. In *EMSOFT*, 2012.

[24] M. Fernandez, D. Morales, L. Kosmidis, A. Bardizbanyan, I. Broster, C. Hernandez, E. Quiñones, J. Abella, F. Cazorla, P. Machado, and L. Fossati. Probabilistic timing analysis on time-randomized platforms for the space domain. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 738–739, March 2017.

[25] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[26] C. Hernández, J. Abella, F. J. Cazorla, A. Bardizbanyan, J. Andersson, F. Cros, and F. Wartel. Design and Implementation of a Time Predictable Processor: Evaluation with a Space Case Study. In *29th ECRTS*, 2017.

[27] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[28] L. Kosmidis, D. Compagnin, D. Morales, E. Mezzetti, E. Qui nones, J. Abella, T. Vardanega, and F. J. Cazorla. Measurement-Based Timing Analysis of the AURIX Caches. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2016.

[29] L. Kosmidis, C. Curtsinger, E. Qui nones, J. Abella, E. Berger, and F. J. Cazorla. Probabilistic Timing Analysis on Conventional Cache Designs. In *2013 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 603–606, 2013.

[30] L. Kosmidis, E. Quiñones, J. Abella, T. Vardanega, C. Hernandez, A. Gianarro, I. Broster, and F. J. Cazorla. Fitting processor architectures for measurement-based probabilistic timing analysis. *Microprocessors and Microsystems*, 47:287 – 302, 2016.

[31] L. Kosmidis, R. Vargas, D. Morales, E. Quiñones, J. Abella, and F. J. Cazorla. TASA: Toolchain-agnostic Static Software Randomisation for Critical Real-time Systems. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 59:1–59:8, New York, NY, USA, 2016. ACM.

[32] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.

[33] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *22nd RTNS*, 2014.

[34] S. Law and I. Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 189–199, 2016.

[35] N. G. Leveson and P. R. Harvey. Software fault tree analysis. *Journal of Systems and Software*, 3(2):173–181, 1983.

[36] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[37] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

[38] H. Pentti and H. Atte. Failure mode and effects analysis of software-based automation systems. In *VTT Industrial Systems, STUK-YTO-TR 190*, page 190, 2002.

[39] H. Pham. *System Software Reliability*. Springer Series in Reliability Engineering. Springer-Verlag London, 2006.

[40] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical Environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, 2012.

[41] P. H. Seong. *Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[42] https://www.absint.com/ait/. *aiT WCET Analyzers*. AbsInt Angewandte Informatik GmbH.

[43] http://www.ehitex.de/application-kits/infineon/2531/aurix-application-kit-tc277-tft. *AURIX Application Kit TC277 TFT*. hitex.

[44] http://wwww.rapitasystems.com. *RVS SUite*. Rapita Systems ltd.

[45] Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.