Run-time Monitoring and Trace Analysis Methodology for Component-based Embedded Systems Design Flow

Vittoriano Muttillo, Giacomo Valente, Luigi Pomante
DISIM
University of L'Aquila
L'Aquila, Italy
{vittoriano.muttillo, giacomo.valente, luigi.pomante}@univaq.it

Hector Posadas, Javier Merino, Eugenio Villar
TEISA
University of Cantabria
Santander, Cantabria, Spain
{posadash, javierm, villar}@teisa.unican.es

***Abstract*—The purpose of this paper is to introduce run time monitoring infrastructures and to analyze trace data inside a well-established component-based methodology. The goal is to show the concept among different monitoring requirements by defining a general reference architecture that can be adapted to different scenarios. Starting from design artifacts, generated by a system engineering modeling tool, and source code automatically generated from UML models, a custom Hardware monitoring sub-system infrastructure will be presented. This sub-system will be able to generate run-time artifacts for run-time verification. We will show how the framework provides round-trip support in the development chain, injecting monitoring requirements from design models down to code and its execution on the platform and trace data back to the models, where the expected behavior will then be compared with the actual behavior. This approach will be used towards optimizing design models for specific properties (e.g, for system performance), using a specific constraint approach compliant with UML standards. Industrial and custom use cases will be used to demonstrate the effectiveness of this approach in real scenarios.**

***Index Terms*—cyber-physical systems; embedded system; design methodologies; validation; monitoring; traces;**

I. Introduction

In the last years, the spread and importance of Cyber-Physical and Embedded Systems are even more increasing [1] [2], but it is still not yet possible to completely standardize and engineer their system-level design flow. Starting from an Electronic System Level (ESL) methodology, the main design issues are to model Functional/Non-Functional requirements and to validate them before implementing the system itself [3]. Designers commonly adopt one or more system-level models (e.g., block diagrams, UML, SystemC, etc.) to have a complete problem view, to perform a check on HW/SW resources allocation/binding and to validate the design by simulating the system behavior. In this scenario, SW tools to support designers to reduce costs and overall complexity of systems development are even more of fundamental importance. Most of the ESL methodologies start the design activities from an executable model of the system behavior based on a given Model of Computation (MoC), and that can be described by means of a proper specification/modeling language (e.g., Ptolemy [4], ForSyDe [5], CONTREX [6]). Moreover, the component-based design [7] approach came to the fore as a reference modeling methodology for several industrial domains (e.g., automotive, avionics). This approach relies on it through the separation of concerns w.r.t. the different system functionalities. This model is strictly related to the concept of events, while each component can produce or fire events of a different nature, using services as an exchanging method to manage data transmission (by means of component interfaces). This model is then translated into an executable system code, and the model is annotated w.r.t. the target application, while designer can change the abstraction level, and the type of analysis without loss of accuracy. In this field, requirements traceability acquires an important role during the design activities. Event tracing allows to verify and validate the input constraints at different abstraction levels, while keeping the static component model and the executed run-time application in sync became an important aspect in the design and validation. Thus, monitoring sub-systems can have an important role in the verification and validation activities, while providing information about system state and behaviours.

In such a context, this work introduces a run-time hardware monitoring infrastructure inside a component-based embedded system design flow. The goal is to extract run-time information from target platforms, while trying to get the component and the run-time model synchronised. A custom HW monitoring infrastructure for reconfigurable logic architectures will be presented, with a focus on Zynq platform and FPGA boards, and a general trace data extraction framework will be considered. Then, bare-metal and Linux APIs will provide run-time information from HW sniffers. A transformation toward Common Trace Format (CTF) will be proposed for run-time validation activities in order to check input constraints, defined using a standard timing model. Results targeting real case studies of run-time monitor for off-line verification will be presented.

The remainder of the paper is organized as follows: Section II presents the state of the art regarding monitoring

HW/SW systems used for model-based design and verification of Cyber-Physical Systems. Section III presents the general reference methodology, starting from the modeling activity down to the HW profiling sub-system. Section IV presents the reference component-based design methodology, while Section V introduces the reference HW monitoring sub-system that generates traces backward to the main design flow. Section VI presents some experimental results with focus on academic and industrial use cases. Finally, Section VII closes the paper with some conclusions and future works.

II. Preliminaries

A Cyber-Physical System (CPS) is an integration of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa [8]. In such a domain, different challenges arising from physical, communication and embedded modeling become more important for analysis and design activities into a complex design environment. Moreover, one of the most important problems is in relation to the electronic technological advancement. With the more and more constant use of FPGAs to prototype and implement SoC designs including multiple processors as either programmable soft cores or HW accelerators, it is also needed a runtime monitoring in order to cope with unpredicted and unforeseen situations in the whole design methodology, with focus on system requirements. Moreover, there is a lack in the definition of approaches that try to consider system requirements into an unified design flow, while take into account traceability and link between models and run-time execution.

In this scenario, ForSyDe (Formal System Design) [5] is a methodology for modeling and design of heterogeneous embedded multi-processors systems. The starting application is modeled by a network of processes interconnected by signals. Then, the model is refined by different design transformations into a target implementation language. Meanwhile, the work in [9] starts from three sub-models, considering a model for SW application (Platform Independent Model) on one side and a platform (Platform Description Model) on the other side, and both models are connected by a Platform Specific Model that defines the mapping of SW into HW. The tool offers different simulation and estimation outputs that drive the designer from the system-level model to the final implementation. Recently, MARTE became the de facto standard for modeling and design embedded systems and CPS, using stereotypes and transformation of models for ad-hoc external tools [10]. A lot of approaches start from MARTE specification, while system transformations allow to integrate external analysis and simulation tools. AADL transformations [11], single consistent behavioral model generation [12] or component-based methodology [13] are just some approaches proposed in literature. All these works do not considers monitoring infrastructure sub-systems fully integrated inside the whole design methodology.

However, none of these works consider monitoring infrastructure sub-systems fully integrated inside the whole design methodology. A monitoring action on a system, in a general view, provides information about its state. These ones can be processed (e.g. filtered, interpolated, etc.) to obtain indications about parameters, such as the behaviour of application under execution (workload characterization, debug action), or characterize some specific components (e.g. cache memories, buses, etc.). This is the base to apply some actions on the system under exam, such as debug, adaptation to different scenario (both software and hardware), system partitioning. Referring to information collection, two types of monitoring can be identified: hardware and software.

A software monitoring solution is based on the exploitation of the processors that are executing the application under exam to collect data useful for monitoring, using code instrumentation to properly manage timers among processing elements. There are various examples of software based profiling systems, that depend of the application (e.g., Gprof [14], LTTng [15]). Furthermore, software profiling necessarily introduces some overheads on execution time and, considering the sampling approach, it has some grade of statistical inaccuracy. Regarding unifying design and monitoring approaches, the work in [16] use time triggered run-time verification, that is an approach that seeks to minimize the software overhead of run-time verification for multi-core systems. During the design process, it seeks to find an optimal mapping of software components to processors cores and an optimal configuration of monitoring frequency to minimize overhead of the monitoring and verification software. Instead, EgMon [17] reduces the software overhead for runtime verification by focusing on monitoring messages transmitted between components, specifically monitoring messages broadcast over a CAN bus and verifying requirements defined using bounded metric temporal logic. EgMon uses a separate device on the CAN bus through USB connection, which reduces the performance overhead for the system but does not eliminate it, due to the added delay of the EgMon device. [18] present a runtime verification system that utilizes live sequence charts, that are similar to UML sequence diagrams and enable support modeling multiple system behaviors, conditional execution sequences, and activation timing. The approach concatenates several live sequence charts to define the possible states of the system that can be examined, and then transforms the concatenated live sequence charts into Linear Temporal Logic (LTL). Although live sequence charts can be transformed to linear temporal logic, the number of events in the resulting LTL can be explode. Also, LTL model checking provides powerful verification method, it would not be the best method to implement monitoring using hardware. Copilot [19] is a compiler-assisted approach that automatically instruments a software binary with custom verification code compiled from a requirements specification language. It generates verification code, for which the timing can be statically analyzed. While this approach reduces the effort required to verify hard real-time constraints for the system, any changes in requirements

mandate re-verifying that the hard real-time constraints are met.

On the other side, Hardware monitoring systems are based on dedicated hardware resources able to carry on the profiling action. This means that less or no source code instrumentation is needed and the software execution by the central processor unit is not altered enough, thus less or no overhead on execution time is introduced, depending on the HW monitoring system infrastructure and approach. For the same reason, Hardware solutions can guarantee the best accuracy in performance analysis. However, these solutions require a larger silicon area occupation for system implementation. Other possible disadvantages are the difficulty to correlate low-level measurements to source code performance metrics and the limited number of allocable hardware resources, that often forces to collect desired performance metrics by means of multiple tests. HW monitors also helps designers to the verification and validation of timing properties on real target platforms. In this domain, the work in [20] proposes to extract assertion branches as part of an HLS flow (to generate custom hardware circuits) from the control data flow, and hardware on-chip monitors are automatically generated. Nasar et All. [21] proposed on-chip monitor based instrumentation, enabled by custom instructions to detect events and dedicated hardware that verify system requirements specified using parametric finite state machines. Authors in [22], on the contrary, automatically synthesize requirements specified in past-time Metric Temporal Logic to hardware monitoring systems, that verifies those requirements over fixed time intervals. P2V [23], instead, is a hardware-based verification method that extends the memory access stage of a MIPS processor to implement a hardware monitoring system. The P2V approach synthesizes requirements specified using a subset of property specification language to a custom dedicated hardware component integrated with the memory access stage. Other interesting works are [24], that identifies software bugs caused by atomicity violations using hardware monitoring systems, [25], that provides a mechanism that can detect thread race by hardware monitoring systems, and [26], that illustrates the watchdog timers, useful and widely used method to recover from system failure at run-time. Other non-invasive proprietary trace-based methods from industries to collect monitoring data are the Arm CoreSight System Trace Macrocell [27], Xilinx ChipScope [28], Intel Altera SignalTap [29], but all these three require the presence of external hardware to read the traces. SnoopP [30] and Airwolf [31], otherwise, are two function-level academic profilers for software applications running on soft-core processors. The work of Seo et al. [32] presents a requirements-driven methodology enabling efficient run-time monitoring of embedded systems. Their approach extracts at run-time monitoring graph from system requirements specified using UML sequence diagrams. Non-intrusive, on-chip hardware dynamically monitors the system execution and in the event of a failure provides detailed information that can be analyzed to determine the root cause.

Another work is [33], that use a trace-based approach for tracing specific signals within a microprocessor focused on test, debug, and validation of real-time systems, while [34] use monitoring systems to run-time verify the CPS as product of the information processing factory. Furthermore, the work in [35] produce hardware monitoring systems that serve as debug infrastructure for high-level synthesis produced circuits.

In order to define a custom profiling system for embedded applications, solution based on specific metric definition and implementation of necessary parts has been considered in [36]. This technique conduced to a definition of a library of elements, to be used to compose a hardware profiling system tailored for the application [37]. These two last papers made the basis for this work, while the custom HW monitoring subsystem has been integrated within the approach presented in [13], considering several software processors (i.e., LEON3, ARM Cortex-A9, Microblaze).

Fig. 1 presents the proposed trace analysis methodology for embedded system design using the *Single-Source System Design* framework (S3D) [13] and custom HW monitors. S3D offers an integrated Eclipse environment that acts as a user interface where the designer can define and analyze the system, using different modules integrated and connected to it.

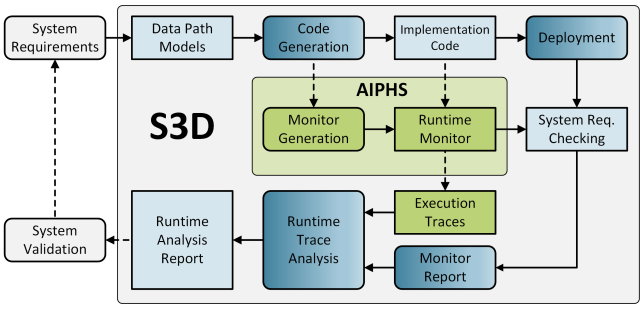

Figure 1. General Proposed Methodology.

For that purpose, the starting point is the development of a UML/MARTE model of the full system, including the specification of the application, the target platform and the corresponding resource mapping. Then, this model is used to automatically generate the executable codes. These codes combine the internal component functionality, described in C++ language, with synthesized glue code used to deploy the functional components on the selected hardware resources and interconnect these functional components with the selected communication semantics and resources. This glue code is also in charge of asking the hardware monitor to collect traces at the adequate points for later analysis. That way the user has no need to directly interact with the hardware monitor, although it can be done if the designer requires more specific information, as described below. Once the executable/simulatable binary is generated, it has to be executed, using the HW monitor to collect trace information. Finally, these traces are analyzed using S3D framework to report information about the system behavior, such as the execution times of the different services, number of calls to them, or data path analysis.

S3D allows to generate normal or trace executables, depending on designer specifications. The introduction of sequence diagrams permits to define the event chain instrumentation path, while specific instrumentation points are added at interfacing level, when services are required or provided by components at run-time. LTTng tracing application has been used as SW monitors inside an embedded system platforms, while the introduction of HW monitors guarantees lower system interference. In the next paragraph the S3D methodology and approach and the HW monitoring sub-system will be described more in details.

III. S3D Framework

As described in previous section, S3D proposes a UML/MARTE modeling methodology to completely specify the system under development. The main idea of this methodology is to propose a single-source modeling approach that supports capturing all the relevant information in a single site thus avoiding duplication of design information. As a result, the automatic generation certain design elements, such as glue codes, makefiles, or simulator configurations, can be performed. This framework enables the analysis of the system under development at different abstraction levels:

- Native Functional Emulation: source code can be compiled for the specific host platform, while event-chains are analyzed at functional level
- Behavioral Simulated Execution: behavioral source code can be compiled for a specific emulator (i.e., VIPPE [38]) and simulated in the host environment with target platform models (in our case ARM and SPARC-V8 architectures). Two different type of simulations are allowed:
  - Performance Model Execution, with all the features and functionalities implemented in each component;
  - Workload Model Execution, without effective code but only with virtual component workloads given by external entities (i.e., WCET, task periods, and deadlines);
- SW Synthesis Execution: the source code can be compiled for the specific target environment and executed on real platforms.

The methodology is component-based both for the description of the target platform and for the application. The definition of the target platform includes the specification of processors, memories, physical communication channels, etc, including their internal characteristics to support the different abstraction levels. The definition of complex systems, including multiple nodes, is also possible.

The modeling of the application also uses components as the main modeling primitive, following a client/server approach. Components communicate among them through ports, and ports contain interfaces, which define the communication methods. These communication interfaces (and methods) can be required or provided by the components, depending if a component acts at this point as a client or a server of this service. As a result, the system application is conceived as a hierarchical network of components, as shown in Fig. 2.

IV. General Methodology Description

Moreover, component models also include complete information about their internal functionality. In principle, no restrictions are imposed to the way the functionality is specified. However, as a preferred solution, it is specified using an action language (i.e., C++). It is important to note that these files must contain platform-independent code. The functional code must avoid the use of system calls or minimize them as much as possible. Communication, concurrency and synchronization details are specified as parameters of ports and interfaces in the UML model, being this information automatically implemented in the glue code used to deploy and connect application components, instead of including it in the associated functional codes. This separation of functionality and communication provides two advantages. First, it enables using the same functional code for the different abstraction levels supported. Secondly, it improves component reuse, as component codes do not restrict the way each component will interact with other components, enabling its use in different projects.

A. Data-path Modeling

Interfaces and ports provide the possibility of specifying communication semantics at the component level. It is possible to define if a service call will be synchronous or asynchronous, if multiple calls can be executed sequentially or must be protected, if its execution must be immediate or can be stored in a queue, the definition of timeouts, retries, etc. As a result, it is possible to define the model of computation (MoC) under each component will operate, including MoCs such as Kahn Process Network (KPN), Synchronous Data Flow (SDF), Timed Data Flow (TDF), Synchronous Reactive (SR), etc. Typically, these MoCs are defined to ensure that the resulting system has certain characteristics. For example, in KPN, no input data is lost during the computation process, while if we use a set of periodic processes communicated with shared variables, it is very likely that some data will be overwritten on the intermediate variables before used. Typically, these MoCs are proposed to be used by all the elements in the system. However, this homogeneous solution is not always the most efficient approach, especially on large systems. The problem then is to define how the system behaves when combining components running under different MoC.

To analyze so, our proposal is to define the most important data paths followed within the system (typically from input to output) in order to analyze its behavior. That way, the trace collection marks added by the glue code generator and the trace analysis tools integrated in S3D can help designers to optimize the system. These data paths are defined as the list of services each data must cross from the input to the output, or between two internal services of the system. For example, we can consider that input data A is used by service 1 to generate data B, which is used by service 2 to generate data C, which again is used by service 3 to generate data D, and so until the output. The modeling of these data paths in UML is done as shown in Fig. 3. As it can be seen, first, each component involved in the data path is specified as a

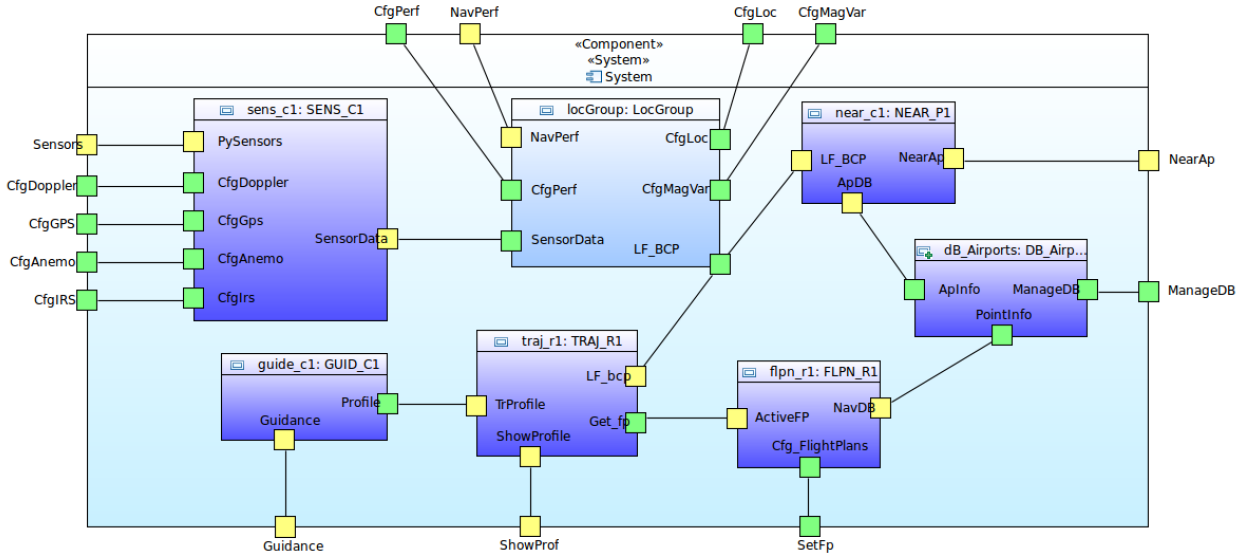

Figure 2. Reference Industrial (Avionic) Application Model.

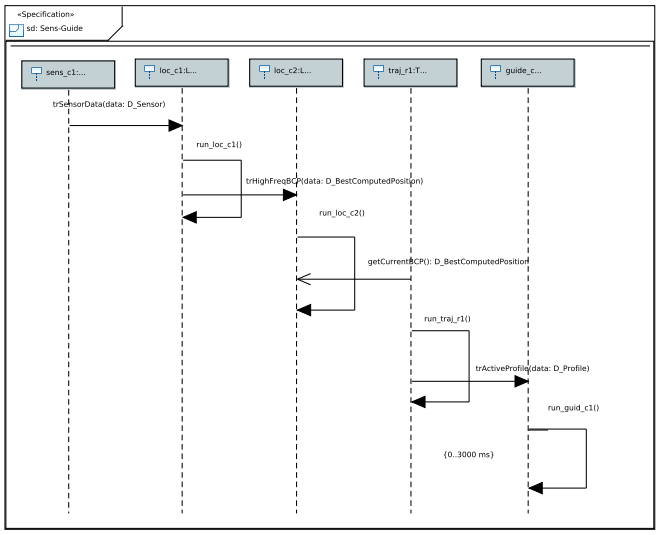

Figure 3. Sens-Guide Data Path Example w.r.t. Application in Fig. 2.

lifeline. Then the services used in the path are specified upside down. There are three types of services: services requested by the component containing the data from other components (e.g., "trSensorData" in Fig. 3), services executed within a component (e.g., "run_loc_c1"), and services from other components that want to read the data, (e.g., "getCurentBCP").

This information is used by S3D to analyze the execution traces obtained by the HW/SW monitor as requested by the marks included in the automatically generated glue code. These traces can report information of data-path latency, the amount of input data collected or output data generated, the internal data lost, the amount of times the same datum has been used by a certain service, or how many times the path has been executed with completely new data (data not used by each service before). Additionally, it is possible to specify constraints, such as the max latency limit of 3000 ms defined in Fig. 3. These parameters will be shown in the result section of this paper.

V. Hardware Monitoring

This section focuses on monitoring methodologies based on ad-hoc hardware mechanisms in order to avoid possible distortion of the system behavior due to the monitoring action, thus satisfying unobtrusive monitoring requirements. We integrate a framework, named *AIPHS*, acronym of ***Adapt**I**ve** **P**rofiling **H**ardware **S**ub-System* [36] [37], that can be used to compose hardware monitoring sub-systems to monitor different metrics at run-time.

A. AIPHS Monitoring

AIPHS is basically conceived to support designers on the development of On-Chip Monitoring Sub-Systems (OCMSs) able to satisfy given Monitorability Requirements, namely requirements about the possibility to observe the behaviour of a system with the goal of evaluating metrics. It is a flexible framework that targets SoCs implemented on Field Programmable Gate Arrays (FPGAs), or on Integrated Circuits (ICs) exploiting some reconfigurable logics. OCMSs developed with AIPHS can generate logs for timing performance measurements on targets with multi-core processors, running bare-metal and Linux based applications (e.g., logs for WCET analysis). AIPHS works internally by exploiting a generalization of the concept of monitoring among different OCMSs, by defining a general reference architecture that can be adapted to different applications. The monitor targets embedded systems architectures, as shown in Fig. 4.

The HW architecture is composed of on-chip and off-chip areas. The former can contain different cores, with possible caches, plus optional single-purpose processors (not shown

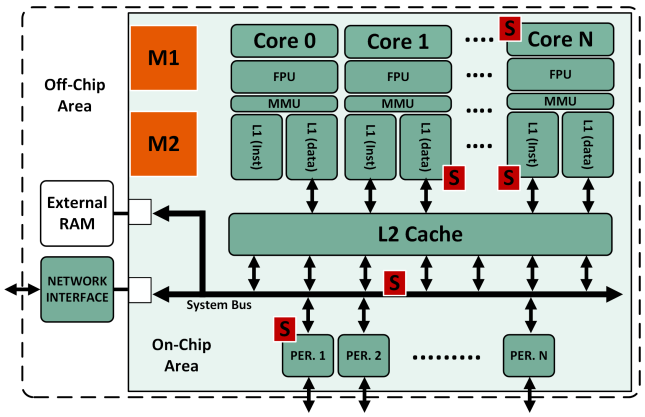

Figure 4. General view of a digital embedded system architecture with the introduction of an On-Chip Monitoring Sub-System by means of AIPHS.

in Fig. 4). Cores and single purpose processors can share different peripherals and controllers, indicated as {'PER. 1', ···, 'PER. N'}. Off-chip area can contain memories (such as RAM or Flash) and Network interfaces. The red elements indicate the on-chip monitoring sub-system: it is composed of two global monitors (M1 and M2) and five sniffers (S). Sniffers are used to monitor the system bus, the peripherals, the cache memories and the different cores.

Fig. 5 shows the connection in more detail. The monitoring data are collected during application execution, and temporarily stored in a memory buffer contained within each sniffer. At the end of execution, the global monitor unit collects data from each sniffer in order to communicate them via an external communication interface. In Fig. 5, *Sniffer 1* monitors the On-chip local memory accesses, while *Sniffer 2* monitors the system bus and *Sniffer 3* monitors the secondary bus. Each sniffer has a *Profiling Data Bus*, through which it is initialized, and a *Log Bus*, to communicate results to the global monitor.

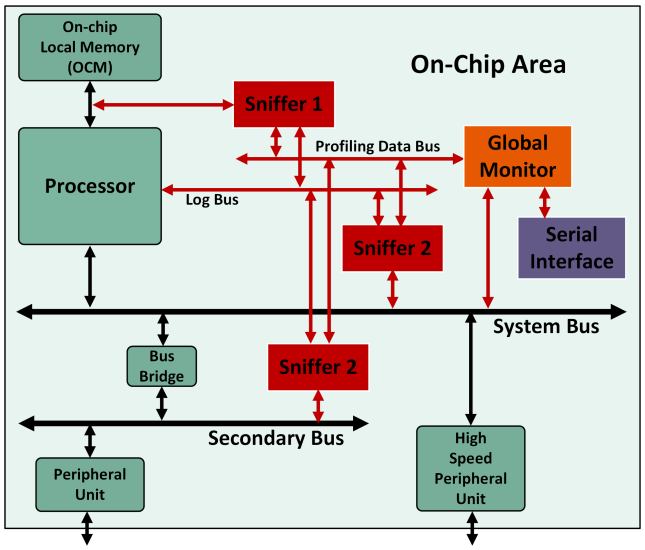

Figure 5. Connection of Sniffers and Global Monitor.

The monitoring sub-system is highly customizable, both in the sniffer and the global monitor parts. Each sniffer is internally composed of a *Nucleus*, a *Global Monitor Processing Interface* (MPI) and a *Target Adapter*, as shown in Fig. 6. The nucleus is based on elements contained in a VHDL library called *LIB_NUCLEUS*. The nucleus takes as input a set of interconnection independent signals, and writes output on a set of registers, that can be readable by global monitors, and that are configurable from the point of view of number and size. Such registers represent the storage space for raw information. Currently, *LIB_NUCLEUS* allows to use either an *Event Monitor Unit* or a *Time Monitor Unit* (or both), that respectively count events instances happened on a specified monitored area and take timestamps related to specified event instances. The latter has been used in this paper. The MPI is able to communicate with global monitors: the block can be composed with elements contained in *LIB_MPIN*, that is composed of a set of on-chip bus interfaces. Finally, the target adapter takes input from interconnections and feeds the nucleus with interconnection independent signals: this block allows the development of on-chip monitoring sub-systems that target different scenarios, while remaining the same nucleus and MPI. Fig. 6 shows how to compose a monitoring system using AIPHS.

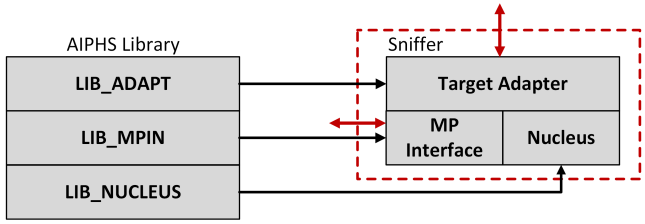

Figure 6. Use of AIPHS VHDL Library to Develop on-chip Monitoring Sub-Systems with the Internal Composition of each Sniffer.

B. AIPHS Application Programming Interface

In order to monitor execution code that runs on embedded platforms without SW tracing application, reducing also system overheads, in this work it is necessary to correlate low-level measurements (taken with HW profiling system) with high-level execution. Let assume that the interest is to measure the execution time of components in the application. To perform this, HW monitors have been introduced to monitor application running on different processors (e.g., LEON3, ARM, Microblaze). These monitor sub-systems can be managed with parallel external debugger tools or inside the same system environment by means of specific APIs offered to manage basic monitor functionalities. Fig. 7 presents the proposed AIPHS API interfaces for monitors configuration, management and data extraction.

The APIs are organized as a 3-tier layered library. The *AIPHS Platform Independent Interface* (AIPHS_PIIF) exposes the function calls to exploit the monitoring operations offered by the HW profiling sub-system. This includes monitors initialization, code instrumentation and data collection. This

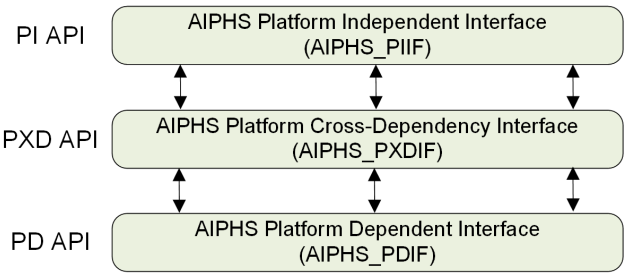

Figure 7. AIPHS SW Interfacing API.

layer allows to guarantee the portability of such kind of monitoring solution to different platform architectures, without changing the design or instrumentation methods. The *AIPHS Platform Dependent Interface* (AIPHS_PDIF) contains the function interfaces dependent on the implementation of HW profiling sub-system. In this layer, different solutions have been realized:

- Special character device driver that allow the library functions to write specific initialization and control frames in profiling configuration registers has been developed (in the prototype case, all registers are mapped in delimited address area).
- Specific user space drivers that allows to directly access memory-mapped devices without writing or introducing new drivers inside the final platform. This solution reduces overheads, but decrease security and fault-tolerance features.
- Specific user space functions able to write direct in memory. This solution bypass kernel space services and routine, and introduce unpredictable run-time behaviors at run-time, depending on OS functionalities.

Finally, the *AIPHS Platform Cross-Dependency Interface* (AIPHS_PXDIF) that connect the PI and PD layer has been introduced to offers system portability on several different platforms.

C. Trace Backward Generation, Injection and Analysis

The last part of this work concerns the trace collection and the adopted timing model to check input requirements. The HW monitors have been limited to only extract information about timing application behaviors, while future works will consider the possibility to use other profiling features (e.g., cache inference, bus contention, memory access). Fig. 8 shows the adopted approach to compile the source code generated from the input component model, introducing the S3D library (used to link high-level UML models to run-time application) and the LibAIPHS (used to manage HW monitors and collect trace data).

Source code and instrumentation are automatically generated from S3D tool, while libraries have been modified to properly manage and use AIPHS, this latter able to profile the application at run-time and generate tracing data from executions, as presented in Section IV. The reference selected trace format is the *Common Trace Format* (CTF), a binary

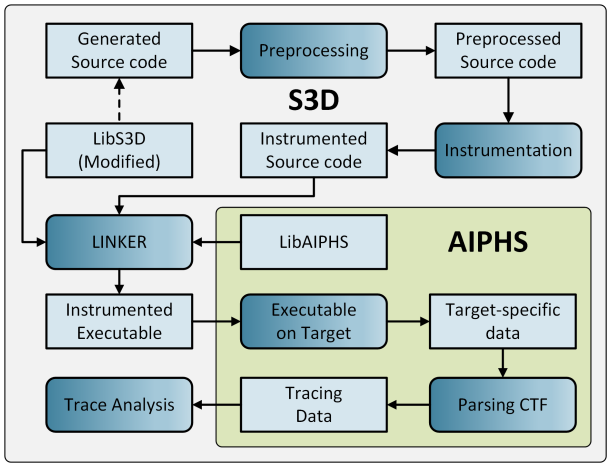

Figure 8. Proposed Code Generation, Profiling and Trace Generation approach.

trace format where headers, contexts, and event fields written in binary files are described using a declarative language called the *Trace Stream Description Language* (TSDL) [39]. Babeltrace has been used as a trace conversion application and library which is able to read and write traces. The traces are in the following form:

[Date:Time] (Time interval) User Trace_generator : function_trace : { **cpu_id = value** }, { **function_id = value, threadId = value, time = value** }

Finally, the trace analysis step involves the injection of collected data back to the data path model, to check input constraints at system level. The S3D-AIPHS whole system framework is presented in Fig. 9. This framework allows to trace, log, monitor and test system behavior at run-time.

Tracing and logging information are available to be used for analysis of the system performance and behavior. The framework offers APIs to monitor the system, the trace information are visible during run-time and are made available for offline analysis after the execution of applications. In the *User Space* it is possible to run considered instrumented applications, automatically generated from S3D environment. *Tuning and Policies* module is intended for testing AIPHS_PIIF API. The *Monitor/Analysis/Log App* is an application able to extract monitoring information (offline or at run-time). This SW module can be an application running on a isolated core or can be an external SW able to analyze application chain events. All the SW modules request services from an *Operating System Abstract Layer*. Our work focus on Linux-based OSs, where Linux API, GNU C Library and Run-time Library, System Call Interface and OS kernel (glibc, stdlibc++, libgomp, Linux kernel version $\geq$ 4.9-1.0) are the minimal basic elements requested. Inside the library layer, the S3D and AIPHS libraries were included to offer traceability respect to inputs constraints and design issues. The *Device Call Abstract Layer* is used for the device driver calls. Finally, the *Hardware Abstraction*

Layer offers the direct link to the considered HW System. In our work we consider LEON3 SMP and ARM Cortex-A9 with AIPHS monitoring sub-systems for timing behavioral sniffing.

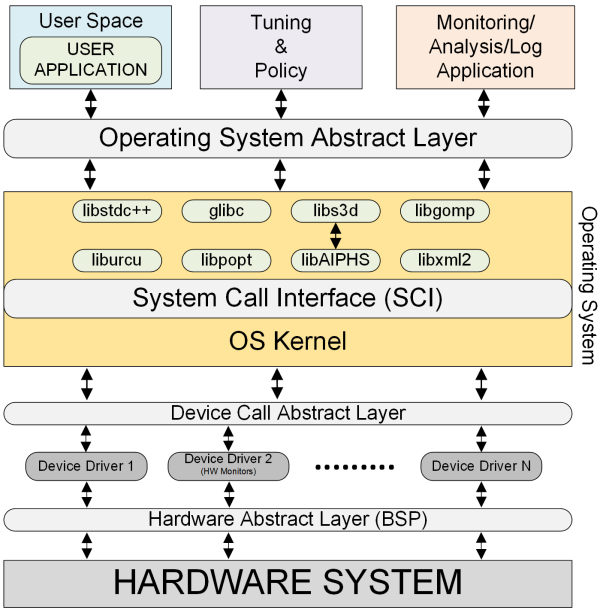

Figure 9. General Platform Validation Architecture.

VI. Experimental Results

This work focuses on the integration of a monitoring sub-system (i.e., AIPHS) inside a component-based design framework (i.e., S3D). The monitoring system has been used to check use case input constraints respect to event chains, considering workload models and different event scenarios. S3D tool and environment has been used to introduce run-time trace extraction functionalities using HW/SW monitoring tools, while compile the application with library able to manage AIPHS monitors, as shown in Fig. 8.

In this work the AIPHS library has been adapted to work on Zynq-7000 and Virtex-6 boards, and has enabled the monitoring of the run-time execution of applications executing on different processors (i.e., ARM Cortex-A9, LEON3). In order to implement and use AIPHS VHDL and SW libraries under these platforms, a set of tools, host operating systems and specific drivers to communicate with the development boards have been used. These are the prerequisites to test, modify and execute correctly the AIPHS functionality in every possible software scenario, for LEON3 (offered by Gaisler) and ARM (offered by Xilinx) platforms.

On Virtex-6 the AIPHS monitors have been introduced for a dual-core LEON3 architecture, where Gaisler offers an open-source environment configured to synthesize SPARC-V8 platform from scratch. A different approach has been used for Zynq-7000 board, where a memory mapped monitor that is able to collect timestamps of the ARM processors associating them with IDs has been realized. To test and evaluate the result, an academic and an industrial case study have been used. The former is the academic use case taken from [40] [41], the latter is a reference industrial case study based on an avionic application developed by Thales France in the context of MegaM@Rt^2 European project [42], where the model is shown in Fig. 2. Fig. 10 presents the timing behavioral tracing results using HW monitors for the academic use case. The application has been executed on dual-core platform (LEON3 in our case), while the x axis is the logical time event instant associated to the component service activation (channel calls for data exchange). From this diagram it is possible to extract information about event chains, instead we are interested to check timing constraints defined at system-level (as shown in Fig. 3). For this purpose, it is possible to create sequence diagrams associated to S3D component-models and directly link them with a trace analyzer tool.

Table I presents the trace analysis results associated to the academic use case running on dual-core LEON3 (i.e., Fir16-GCD, Fir8-GCD, Fir8E-GCDE), and industrial avionic application running on dual-core ARM platform (i.e., Sens-Guide, Sens-Near). Event chains are different in each application, in terms of workloads and data paths. In the case of the academic use case, data paths from input to output for several FIR implementation have been used. In the industrial use case, data paths from the ”sensor” input to the ”guiding” and ”near airport” outputs have been analyzed. From this analysis it is possible to determine whether the application requires design modeling changes or not, in case that requested time is not always fulfilled. This is the case for both applications tested.

Table I
Trace Analysis Results (in ms).

Name	Use Case	Request	Max	Min	Mean	Received	Valid	Lost
Fir16-GCD	Academic	40	57	15	29.3	26	92.30%	7.70%
Fir8-GCD	Academic	40	44	30	35.1	19	94.73%	5.27%
Fir8E-GCDE	Academic	40	40	33	36.2	5	100.00%	0.00%
Sens-Guide	Industrial	3000	2785	455	1607.2	20115	100.00%	0.00%
Sens-Near	Industrial	500	591	169	372.64	39728	96.95%	3.05%

Table II presents the HW monitors area overheads. It is worth noting that the overhead is under 1.50% of the total area occupation. This is due to the restricted number of sniffers used to monitor the processor execution (the minimum number of sniffers to evaluate the performance of this system under test was 2, equal to the number of cores analyzed), but also related to the AIPHS modular optimized infrastructure. This reduced FPGA area overhead ensures also bounded energy/power consumption in case the monitoring sub-system will be released also in the final platform for online run-time analysis.

Finally, Table III presents the timing overhead associated to the use of HW (AIPHS) or SW (LTTng) monitoring sub-systems. The LEON3 (FPGA) and ARM (Zynq) column show the reduced overhead introduced by AIPHS sub-system using the approach described in this paper. This overhead is noticeably less to the one associated to LTTng application. Future works will integrate the new AIPHS version managed by an external Global Monitor Application (e.g., debugger, host demons, DMA and HW dedicated systems) to introduce no overhead inside the whole system environment, while

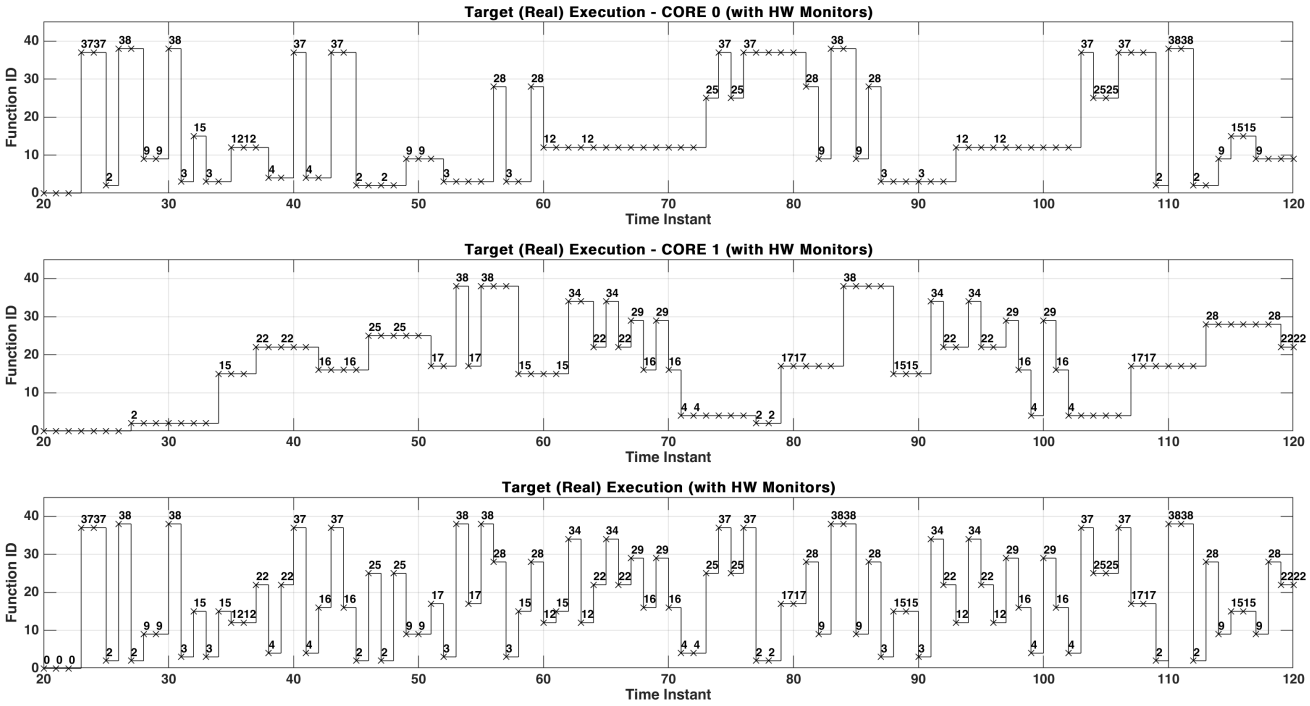

Figure 10. Reference Application Profiling using HW Monitors on Multicore.

Table II
HW MONITOR AREA OVERHEAD.

Logic elements	LEON3	LEON3 + AIPHS	ARM + AIPHS
Slice Registers	6526 (2.70%)	8335 (3.45%)	795 (0.75%)
Flip Flops	6526 (2.70%)	8335 (3.45%)	795 (0.75%)
Slice LUTs	15498 (41.13%)	15931 (42.28%)	607 (1.14%)
Logic LUTs	15444 (40.98%)	15877 (42.13%)	605 (1.14%)
Memory LUTs	54 (0.14%)	58 (0.16%)	2 (0.01%)
Block RAM Tile	34 (8.17%)	34 (8.17%)	0 (0%)
DSPs	8 (1.04%)	8 (1.04%)	0 (0%)

solving the problem of linking the high-level component-based application model with the run-time timing samples without SW instrumentation or system perturbations.

Table III
HW/SW MONITOR TIMING OVERHEAD.

Monitors	x86 (Host)	x86 (Simulation)	LEON3	ARM
AIPHS	N.A.	N.A.	2.66%	0.43%
LTTng	9.13%	12.40%	N.A.	15.27%

VII. CONCLUSION

This work presented an integrated run-time monitor subsystem inside a component-based embedded system design flow. The whole approach allows to extract timing trace from the system run-time execution, while linking them to the different abstraction level design activities needed to realize and deploy complex CPS and CPSoS. Final results show that the approach is useful for system verification and validation, while introducing bounded overheads. Future works will enhance the integrated approach by reducing or eliminating area and timing overheads, using external debugging or tracing applications, and will propose a fully integrated tool and framework offering the possibility to select the specific monitoring sub-systems respect to input designer monitoring requirements.

ACKNOWLEDGMENT

This work has been partially funded by the EU and the Spanish MICINN through the ECSEL-RIA 2017-737494 MegaMart2, the ECSEL-JU 2018-826610 COMP4DRONES and the FEDER/Ministerio de Ciencia, Innovación y Universidades - Agencia Estatal de Investigación TEC2017-86722-C4-3-R PLATINO projects.

REFERENCES

[1] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli, "Modeling cyber–physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012.

[2] Y. Z. Lun, A. D'Innocenzo, F. Smarra, I. Malavolta, and M. D. D. Benedetto, "State of the art of cyber-physical systems security: An automatic control perspective," *Journal of Systems and Software*, vol. 149, pp. 174 – 216, 2019.

[3] V. Muttillo, G. Valente, F. Federici, L. Pomante, M. Faccio, C. Tieri, and S. Ferri, "A design methodology for soft-core platforms on fpga with smp linux, openmp support, and distributed hardware profiling system," *EURASIP Journal on Embedded Systems*, vol. 2016, pp. 1–14, 9 2017.

[4] E. Lee, "The past, present and future of cyber-physical systems: A focus on models," *Sensors (Basel, Switzerland)*, vol. 15, pp. 4837–4869, 03 2015.

[5] I. Sander and A. Jantsch, "System modeling and transformational design refinement in forsyde [formal system design]," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, 2004.

[6] K. Grüttner, R. Görgen, S. Schreiner, F. Herrera, P. Peñil, J. Medina, E. Villar, G. Palermo, W. Fornaciari, C. Brandolese, D. Gadioli, E. Vitali, D. Zoni, S. Bocchio, L. Ceva, P. Azzoni, M. Poncino, S. Vinco, E. Macii, S. Cusenza, J. Favaro, R. Valencia, I. Sander, K. Rosvall, N. Khalilzad, and D. Quaglia, "Contrex: Design of embedded mixed-criticality control systems under consideration of extra-functional properties," *Microprocessors and Microsystems*, vol. 51, pp. 39 – 55, 2017.

[7] E. A. Lee and A. Sangiovanni-Vincentelli, "Component-based design for the future," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–5, March 2011.

[8] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. The MIT Press, 2nd ed., 2016.

[9] H. Posadas, P. Peñil, A. Nicolás, and E. Villar, "Automatic synthesis of communication and concurrency for exploring component-based system implementations considering uml channel semantics," *J. Syst. Archit.*, vol. 61, p. 341–360, Sept. 2015.

[10] F. Mallet, E. Villar, and F. Herrera, *MARTE for CPS and CPSoS*, pp. 81–108. Singapore: Springer Singapore, 2017.

[11] M. Faugere, T. Bourbeau, R. d. Simone, and S. Gerard, "Marte: Also an uml profile for modeling aadl applications," in *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pp. 359–364, 2007.

[12] J. DeAntoni and F. Mallet, "Timesquare: Treat your models with logical time," in *Objects, Models, Components, Patterns* (C. A. Furia and S. Nanz, eds.), (Berlin, Heidelberg), pp. 34–41, Springer Berlin Heidelberg, 2012.

[13] F. Herrera, J. Medina, and E. Villar, *Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach*, pp. 141–185. Dordrecht: Springer Netherlands, 2017.

[14] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, (New York, NY, USA), p. 120–126, Association for Computing Machinery, 1982.

[15] *LTTng: an open source tracing framework for Linux*, 2020 (accessed: 18.04.2020). https://lttng.org/.

[16] S. Navabpour, B. Bonakdarpour, and S. Fischmeister, "Time-triggered runtime verification of component-based multi-core systems," in *Runtime Verification* (E. Bartocci and R. Majumdar, eds.), (Cham), pp. 153–168, Springer International Publishing, 2015.

[17] A. Kane, O. Chowdhury, A. Datta, and P. Koopman, "A case study on runtime monitoring of an autonomous research vehicle (arv) system," in *Runtime Verification* (E. Bartocci and R. Majumdar, eds.), (Cham), pp. 102–117, Springer International Publishing, 2015.

[18] M. Chai and B.-H. Schlingloff, "Monitoring systems with extended live sequence charts," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), (Cham), pp. 48–63, Springer International Publishing, 2014.

[19] L. Pike, S. Niller, and N. Wegmann, "Runtime verification for ultra-critical systems," in *Runtime Verification* (S. Khurshid and K. Sen, eds.), (Berlin, Heidelberg), pp. 310–324, Springer Berlin Heidelberg, 2012.

[20] M. B. Hammouda, P. Coussy, and L. Lagadec, "A unified design flow to automatically generate on-chip monitors during high-level synthesis of hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 384–397, 2017.

[21] A. Nassar, F. J. Kurdahi, and W. Elsharkasy, "Nuva: Architectural support for runtime verification of parametric specifications over multicores," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 137–146, 2015.

[22] T. Reinbacher, M. Függer, and J. Brauer, "Real-time runtime verification on chip," in *Runtime Verification* (S. Qadeer and S. Tasiran, eds.), (Berlin, Heidelberg), pp. 110–125, Springer Berlin Heidelberg, 2013.

[23] H. Lu and A. Forin, "The design and implementation of p2v, an architecture for zero-overhead online verification of software programs," Tech. Rep. MSR-TR-2007-99, Microsoft, August 2007.

[24] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access-interleaving invariants," *IEEE Micro*, vol. 27, no. 1, pp. 26–35, 2007.

[25] P. Zhou, R. Teodorescu, and Y. Zhou, "Hard: Hardware-assisted lockset-based race detection," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 121–132, 2007.

[26] N. Murphy, "Watchdog timers," in *Embedded Systems Programming*, pp. 112–124, 2000.

[27] *Arm CoreSight System Trace Macrocell*, 2020 (accessed: 18.04.2020). https://developer.arm.com/.

[28] *ChipScope Integrated Logic Analyzer (ILA)*, 2020 (accessed: 18.04.2020). https://www.xilinx.com/products/intellectual-property/chipscope_ila.html.

[29] *Quartus SignalTap*, 2020 (accessed: 18.04.2020). https://www.intel.com/.

[30] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/software codesign," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, (New York, NY, USA), p. 190–199, Association for Computing Machinery, 2004.

[31] J. G. Tong and M. A. S. Khalid, "Profiling tools for fpga-based embedded systems: survey and quantitative comparison," *Journal of Computers*, pp. 1–14, 2008.

[32] M. Seo and R. Lysecky, "Non-intrusive in-situ requirements monitoring of embedded system," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, aug 2018.

[33] R. Fryer, "Fpga based cpu instrumentation for hard real-time embedded system testing," *SIGBED Rev.*, vol. 2, p. 39–42, Apr. 2005.

[34] E. A. Rambo, T. Kadeed, R. Ernst, M. Seo, F. Kurdahi, B. Donyanavard, C. B. de Melo, B. Maity, K. Moazzemi, K. Stewart, S. Yi, A. M. Rahmani, N. Dutt, F. Maurer, N. A. V. Doan, A. Surhonne, T. Wild, and A. Herkersdorf, "The information processing factory: A paradigm for life cycle management of dependable systems," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion*, CODES/ISSS '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[35] J. Goeders and S. J. E. Wilton, "Signal-tracing techniques for in-system fpga debugging of high-level synthesis circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 83–96, 2017.

[36] A. Moro, F. Federici, G. Valente, L. Pomante, M. Faccio, and V. Muttillo, "Hardware performance sniffers for embedded systems profiling," in *2015 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES)*, pp. 29–34, 2015.

[37] G. Valente, V. Muttillo, L. Pomante, F. Federici, M. Faccio, A. Moro, S. Ferri, and C. Tieri, "A flexible profiling sub-system for reconfigurable logic architectures," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 373–376, 2016.

[38] *virtual Platform Parallel Performance Evaluation*, 2020 (accessed: 18.04.2020). http://vippe.teisa.unican.es/.

[39] *The Common Trace Format*, 2020 (accessed: 18.04.2020). https://diamon.org/ctf/.

[40] L. Pomante, V. Muttillo, M. Santic, and P. Serri, "Systemc-based electronic system-level design space exploration environment for dedicated heterogeneous multi-processor systems," *Microprocessors and Microsystems*, vol. 72, p. 102898, 2020.

[41] D. Ciambrone, V. Muttillo, L. Pomante, and G. Valente, "Hepsim: An esl hw/sw co-simulator/analysis tool for heterogeneous parallel embedded systems," in *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, 2018.

[42] *MegaM@Rt2: Industrial case studies that will benefit of MegaM@rt2*, 2020 (accessed: 18.04.2020). https://megamart2-ecsel.eu/use-cases/.