A composable monitoring system for heterogeneous embedded platforms

GIACOMO VALENTE, Università degli Studi dell'Aquila, Italy
TIZIANA FANNI, Università degli Studi di Sassari, Italy
CARLO SAU, Università degli Studi di Cagliari, Italy
TANIA DI MASCIO, Università degli Studi dell'Aquila, Italy
LUIGI POMANTE, Università degli Studi dell'Aquila, Italy
FRANCESCA PALUMBO, Università degli Studi di Sassari, Italy

Advanced computations on embedded devices are nowadays a must in any application field. Often, to cope with such a need, embedded systems designers leverage on complex heterogeneous reconfigurable platforms that offer high performance, thanks to the possibility of specializing/customizing some computing elements on board, and are usually flexible enough to be optimized at run-time. In this context, monitoring the system has gained increasing interest. Ideally, monitoring systems should be non-intrusive, serve several purposes and provide aggregated information about the behaviour of the different system components. However, current literature is not close to such ideality: for example, existing monitoring systems lack in being applicable to modern heterogeneous platforms. This work presents a hardware monitoring system, which is intended to be minimally invasive on system performance and resources, composable and capable of providing to the user homogeneous observability and transparent access to the different components of a heterogeneous computing platform, so that system metrics can be easily computed from the aggregation of the collected information. Building on a previous work, this paper is primarily focused on the extension of an existing hardware monitoring system to cover also specialized coprocessing units, and the assessment is done on a Xilinx FPGA-based System on Programmable Chip. Different explorations are presented to explain the level of customizability of the proposed hardware monitoring system, the trade-offs available to the user, and the benefits with respect to standard de-facto monitoring support made available by the targeted FPGA vendor.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: HW Monitoring, Monitoring system, Heterogeneous system, HW Reconfiguration, FPGA

ACM Reference Format:
Giacomo Valente, Tiziana Fanni, Carlo Sau, Tania Di Mascio, Luigi Pomante, and Francesca Palumbo. 2018. A composable monitoring system for heterogeneous embedded platforms. *J. ACM* 37, 4, Article 111 (August 2018), 33 pages. https://doi.org/10.1145/1122445.1122456

Authors' addresses: Giacomo Valente, Università degli Studi dell'Aquila, L'Aquila, Italy, giacomo.valente@univaq.it; Tiziana Fanni, Università degli Studi di Sassari, Cagliari, Italy, tfanni@uniss.it; Carlo Sau, Università degli Studi di Cagliari, Cagliari, Italy, carlo.sau@unica.it; Tania Di Mascio, Università degli Studi dell'Aquila, L'Aquila, Italy, tania.dimascio@univaq.it; Luigi Pomante, Università degli Studi dell'Aquila, L'Aquila, Italy, luigi.pomante@univaq.it; Francesca Palumbo, Università degli Studi di Sassari, Cagliari, Italy, fpalumbo@uniss.it.

0004-5411/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

System monitoring is typically an application- and platform-specific process, especially when dealing with embedded systems. By definition, a generic monitoring process, implemented by a proper monitoring system, has the goal of satisfying some given monitoring requirements (those for the design of a monitoring system [31]), which turns out into gathering system representative metrics [19]. Traditionally, designers have been using custom methodologies and monitoring systems to evaluate the metrics of interest, exploiting ad-hoc Hardware (HW)/Software (SW) components and infrastructures to make such metrics observable by the designer and/or by the system itself. HW monitoring systems, preferable to SW since they minimally affect the execution performance, are intended as the physical components needed in the design to gather information from system resources (e.g., cores, memories, coprocessing units. etc.). The definition of ad-hoc solutions, optimized for a given platform [6, 46] or on metrics [11, 30], typically fails in being portable to other platforms, technologies, and/or contexts.

The problem of defining effective and efficient HW monitoring systems, with small engineering effort, has become more complex in modern heterogeneous devices. To support variable workloads and to address several and concurrent functional and non-functional requirements, modern platforms tend to integrate different types of resources. Different cores and/or application-specific units are adopted, but also configurable logic is often brought on-board. Field Programmable Gate Array (FPGA) devices, traditionally employed for rapid prototyping and low-volume application purposes, are gaining momentum in production (e.g., Lattice Semiconductor FPGA in edge devices [1]). The continuous increase in the computational demand required by modern applications, coupled with flexibility requirements typical of Cyber-Physical Systems (CPSs), began to show the limits of traditional general-purpose SW-programmable platforms. Customizability upon the application needs makes FPGA-based System on a Programmable Chip (SoPC) particularly suitable to this context. Nevertheless, having available these many degrees of freedom certainly complicates the monitoring process. The monitoring system needs not only to gather information to evaluate different kinds of metrics (e.g., number of cache misses, number of clock cycles to execute HW or SW tasks) and to access different types of resources (e.g., the memory), but it also needs to be configurable enough to be customized upon designer requests, possibly with minimal effort for the user. To the best of our knowledge such a desirable monitoring infrastructure does not exist yet, despite the problem is known since a while. Back in 2013, Kornaros et al. [19] already expressed this lack, surveying the existing monitoring systems for multi-core Systems on Chip (SoCs), and trying to identify the motivations behind it. In Kornaros' work a taxonomy for the formalization of the monitoring problem was proposed and discussed. The concept of event to be monitored turned out to be not really well defined, and a lot of subjectivity was there due to the fact that *"different observers can describe the same event in different terms, and may assume different sources of the cause, or of the location, or of the time of the event monitoring in general"*. Such a lack and the need for overcoming the diversity of solutions, brought by the designer-subjectivity or by the heterogeneity of monitored metrics and platforms/components, which is particularly true in the context of heterogeneous FPGA SoPCs, are the major drivers behind this work.

In this paper, we propose a composable HW monitoring system for heterogeneous embedded platforms able to gather information from different system resources to allow gathering information from all the different components composing a traditional heterogeneous systems, which will allow afterwards for system-level metrics evaluation. In particular:

- Following a brick-based principle, different system resources at different abstraction levels are made observable by the proposed HW monitoring system. Each system resource/level is observed through distributed *sniffers* able to get the monitoring information and to aggregate

them in a common data structure easily accessible by the user (e.g., a designer, a run-time manager): sniffers are composable and, together with their collected information and their final structure, represents the monitoring system. *Composability, i.e. the possibility of observing different system resources operating at different levels of abstraction, is a key feature of the proposed approach.*

- The proposed HW monitoring system is a unifying solution that makes observable both HW and SW Intellectual Propertys (IPs). Furthermore, the collected monitoring information is accessible through Application Program Interfaces (APIs), guaranteeing that the events are properly captured and that the information is correctly propagated through the SW stack, no matter where they have been extracted from. *Homogeneous observability and access provided by the sniffers at different layers is a key feature of the proposed approach.*
- The proposed HW monitoring system is based on a configurable number of distributed sniffers, enabling a high degree of customization in terms of observed interconnections and metrics to be gathered. No matter to which resource they are attached to, sniffers have always the same internal architecture. These features enable the reuse of blocks among different sniffers by means of a library-based approach. *System observation is more straightforward. Sniffers can be re-used and easily customized to gather new information from new system parts.*
- The proposed approach is fully passive. The actions and elements needed to gather information from the system are decoupled from those using the gathered information. In this work, we focus on the Event Instance Generation, Data Capture and Data Filtering steps of the *generic monitoring process* introduced in [19]. Decoupling the passive observation action from the active reaction one allows a more generic definition of the HW monitoring system and to make it reusable for different purposes over different platforms. *The heterogeneous system, despite its complexity and target application, is made entirely observable.*

The rest of this paper is organized as follows. Section 2 describes more in details the reference scenario and, in particular, the family of targeted computing platforms. Moreover, it explores also the state of the art, highlighting the contributions/advances of this paper in the provided boxes. Section 3 illustrates our composable HW monitoring system for heterogeneous embedded platforms. Section 4 discusses the carried out assessments, before the final remarks provided in Section 5.

2 MONITORING HETEROGENEOUS SYSTEMS: CONTEXT AND BACKGROUND

The advent of SoPCs allowed for the definition of powerful embedded computing platforms that combine the high-level management capabilities of microprocessors with the high performance. Concepts as FPGA overlays and FPGA companion computers are nowadays extremely popular [9, 43]. They are addressed in many studies that target HW/SW systems [20, 32, 36] that aim to couple microprocessors with HW accelerators, which may also offer different levels of reconfigurability [14]. This introduces the need for having visibility at various levels of the computing infrastructure. Figure 1 depicts a high-level schematic overview of the target reference architecture that we consider in this work that can present one or more microprocessors, one or more coprocessors, and a memory that shares data by means of one or more interconnection buses. Here with coprocessors we intend dedicated hardware accelerators, which can be either fixed function or reconfigurable ones, loosely coupled

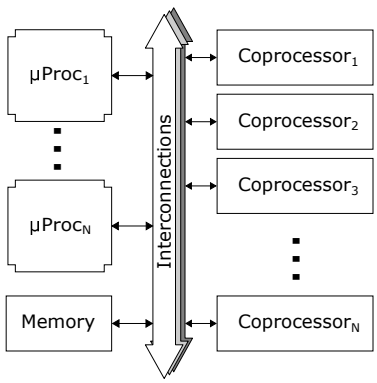

Fig. 1. Baseline reference architecture.

with the host microprocessors by means of the generic system interconnection (meaning that a unique bus line, or parallel bus lines, can be used to better handle data and control signals). A tightly coupled version of the work, where coprocessors are accessible through exclusive, more efficient connections is left to future extensions. Monitoring infrastructures are already integrated in fixed architectures [6], and HW performance counters are a built-in feature of all modern microprocessors, allowing for low-level performance analysis or tuning. Nevertheless, dealing with heterogeneity makes the problem of system observability more complex. At the time being, the state of the art around FPGA overlays and FPGA companion computers is still young [23, 43] and established design and programming solutions are not there yet [22]. One of the reasons is that the programmable part is quite always shaped according to the considered problem, and this inevitably impacts also on the adoptable HW monitoring system. A widely applicable solution embracing all of the components of the architecture, and allowing for a homogeneous observability and access to all the components, is not there yet. That is the motivation at the basis of this work. Times are mature enough to start defining a more comprehensive metric agnostic HW approach. In the rest of this section, state of the art approaches are discussed in Section 2.1, then the contributions of the proposed work in this context are summarized in Section 2.2.

2.1 Related Works

Several reasons for monitoring heterogeneous systems exist: e.g., to guarantee the correct execution of a device and the right communication among its elements, to gather statistics used in subsequent system re-design cycles, to understand run-time performance and check whether any room for optimization is there. Such a proliferation of needs led designers to cope with them with different solutions which turned out to be as heterogeneous as the systems they want to monitor. The lack of standardization and homogeneity of methodologies and purposes was analysed for the first time in 2013 by Kornaros et. al [19]. Authors tried to classify the monitoring techniques according to their purpose: Debug, Performance, Quality-of-Service, Power related, Fault Tolerance and Reliability, Security and Others. Most of the works they analysed adopt vertical approaches (as [37, 39, 40] in Table 1), but in nowadays complex and heterogeneous systems such classification may fail. Indeed, in current systems different aspects and purposes are monitored concurrently, which may imply accessing different architectural components at the same time. Kornaros et al. concluded their analysis with an open challenge: *"to understand the exact robustness, performance, and complexity characteristics of integrating diverse HW components together with SW tasks with varying behaviour"*.

> The issue of heterogeneity mastering is still there and when microprocessors are coupled to embedded reconfigurable units the problem is even more severe. Designers shall tackle both the heterogeneity of the infrastructure and of some of the elements of the infrastructure over time. Indeed, the coprocessing units may change their internal structure over time, depending on the tasks to be executed, leveraging on the reconfiguration capabilities of the FPGA.

In the following, we analyse and discuss different monitoring strategies available in literature, with the intent of classifying them in terms of monitored resources. We consider only those works that address one or more elements present in our reference architecture (see Figure 1) and we also provide information regarding the purpose at the base of their monitoring needs. To simplify the classification, we grouped Kornaros' purposes as follows: Performance (including also Profiling and Quality of Service), Debug and Verification (including also Run-time Verification, Fault Tolerance and Reliability), Power (including also Energy and Temperature), and Security. Table 1 depicts a schematic overview of the considered monitoring strategies, including the one presented in this paper.

Many works available in literature focus on microprocessor monitoring. Nelissen et al. [31] and Seo et al. [40, 41] focus on monitoring it for verification purposes. Nelissen et al. present an architecture for the implementation of a safe and reliable monitoring framework for run-time SW verification. Seo et al. in [40] propose a Trace Abstraction Layer and show a methodology that can perform verification using the Micron Automata Processors, while in [41] authors show a methodology that can drive a designer from requirements to the configuration of a hardware monitoring system for runtime verification, targeting the SW running on microprocessors. Similarly, Rambo et al. [35] adopt the Trace Abstraction Layer for verification and performance monitoring, discussing different techniques for information processing factory that make it a suitable solution for highly dependable systems. Different researches in literature developed strategies of HW profiling intended to foster SW optimization or system self-adaptation. Aldham et al. [4] presented LEAP, a HW profiler for FPGA-based embedded microprocessors, where gathered information is used to improve both performance and HW/SW partitioning. To improve this latter, Nadimpalli et al. [29] proposed another profiler capable of identifying high computational loads to be moved to HW execution. Scheipel et al. [39] present an approach for measuring execution time and events of an embedded system by integrating a dedicated performance monitoring infrastructure using HW/SW co-design techniques; on the other hand, Sadek et al. [37] propose another infrastructure called STHEM, which is a collection of utilities for heterogeneous embedded image processing platforms. STHEM features also a power measurement utility that enables programmers to correlate instantaneous power samples with concurrent HW/SW traces, gathered through the Xilinx SW debugger.

Despite the heterogeneity of their purposes, these microprocessors monitoring systems do not address the other components of the adopted reference architecture. Moreover, to the best of our understanding, the kind of monitored events and used monitors would make them hardly tuneable to gather different information from other components in a composable manner.

Monitoring the microprocessor is not sufficient to offer a complete system observability. Patrigeon et al. [34] present an FPGA-based platform, instrumented with monitors, for real-time evaluation of Ultra Low Power SoCs. The microprocessor interacts with the monitors as with common peripherals, connected by means of the Advanced High-performance Bus lite system. The monitors gather events related to the memory execution and are enabled, disabled, and reset through SW function calls. Doyle et al. [13, 25] present ABACUS, a HW framework that leverages on the FPGA reconfigurable fabric to monitor the workload execution. To foster ABACUS portability to different architectures and the extension with profiling units, microarchitecture independent metrics are supported. Also, a trade-off analysis between application performance and amount of data gathered is performed.The work of Valente et al. [27, 44], apart from the fact that is able to monitor also the interconnection, introduces also the flexibility in monitoring through the definition of a custom profiling system for embedded applications: their work split the monitor design in a number of sub-blocks (stored in a library), each one implementing a part of a generic profiling action.

All the above-mentioned works use monitoring for SW performance understanding only, and the considered reference architecture is still not a heterogeneous one. Nevertheless, Doyle et al. and Valente et al., with a library of customizable monitors, moved some steps towards the concepts of a wider applicability of the monitoring components to different resources/architectures.

Similarly to memories, also the communication elements have been addressed as monitoring object. Kyung et al. [26] presented the first monitor for the Advanced eXtensible Interface (AXI) bus, and Xilinx provides a dedicated monitor for AXI4 connections, the AXI Performance Monitor [46]. It can be added to the design to measure major transaction-related metrics.

These communication-oriented solutions are limited by construction, being highly component-specific, but are certainly complementary to others.

The ARM Coresight [6] is a commercial solution for the debug and trace of complex SoCs. It eases the monitoring of the data infrastructure, providing a library of modular components, for the monitoring of CPU, memory and communication elements, generalizing the access to the monitors.

One of the most interesting aspects of the ARM Coresight approach is that data gathering, from the SW perspective, is made homogeneous despite the monitored components. This is an appealing aspect, that goes towards the same attempt of access homogenization that we are pursuing in this work. Nevertheless, as it should be expected, the Coresight is conceived for a fixed infrastructure, and no (re-)configurable coprocessing units are taken into consideration.

The advent of FPGA-based SoCs opened up a plethora of new opportunities in the computing domain and, from the monitoring perspective, it introduced coprocessors as new elements to be monitored. SW monitoring could serve to identify parts of the application that benefit from a HW implementation [4, 29] and some works address FPGA-based platforms [25, 34, 44], despite they do not give a comprehensive monitoring for such systems. To enable observability, FPGA vendors, such as Xilinx and Intel, are making available solutions to monitor SoC [5, 45, 46]. Besides AXI Performance Monitor, intended for buses, Xilinx provides also the System Integrated Logic Analyzer (ILA) [45] while Intel has the similar Signal Tap Logic Analyzer [5], both for debug purposes.

In all these cases, the internal signals and interfaces are monitored, requiring high HW expertise since no API calls are available. Moreover, these techniques are conceived for the programmable logic only. They could be combined with other approaches, but fail in providing a widely re-usable and applicable solution. Finally, they are vendor-specific. To address the problem of monitoring from a more general perspective, the level of abstraction has to be risen, and vendor-specific approaches are not generally suitable for this.

To raise the level of abstraction and offer a user-friendly monitoring system suitable for SW developers, Goeders et al. [16] presented a High Level Synthesis (HLS) tool for the instrumentation of monitors for debugging. Their work has been demonstrated on the LegUp HLS [10] and, starting from a C++ description, both systems and monitors are derived. Hammouda et al. [17] provide an HLS tool that, starting from a C description, automatically generates the HW system instrumented with monitors for run-time verification. To provide a SW developer-friendly approach, and to monitor also coprocessing units, Fanni et al. [15, 24] extended the Performance Application Programming Interface [33], developing a configurable component for reading monitors within an application-specific coprocessor. They also propose a toolchain that instruments the HW coprocessors, generates the configuration file for the component (compliant with the Performance Application Programming Interface), and provides the APIs to initiate and read the monitors in the HW coprocessors, as the ones normally present on every CPU.

HLS-based approaches are capable of overcoming the problem of low level details management. They could potentially be extended to cover all the elements of our reference architecture. Nevertheless, to the best of our knowledge, such extensions have not been developed yet.

By looking at the monitoring issue from a high level perspective, the work of Lee et al. has to be mentioned. They proposed in [21] a system-level observation framework for the run-time analysis and verification of both SW and HW tasks without perturbing the system execution. In this work, a concrete step ahead towards a formalization of the monitoring process and the definition of a generalizable monitoring method is made. Monitors are built upon sets of customizable observation probes that can be adapted to monitor both microprocessors and coprocessing units. This work

draws attention on the connection between the goal of the monitoring, which in the paper is debugging, and the low-level event types. Moreover, the event-generation process from "what is monitored" is considered as a fundamental activity in building the monitoring infrastructure.

Lee's work represents a step ahead in the generalization of a monitoring action for run-time verification, even if it focuses only on microprocessor and coprocessor (no details about the types of supported coprocessors are provided). It defines the HW/SW observation interfaces to gather the data from the coprocessing elements, but not the details on how events are captured.

To the best of our knowledge, only two literature works seemed to be capable of targeting all the components of the adopted reference architecture. Both Najem et al. [30] and Zoni et al. [11] are focused on power monitoring only, exploiting machine learning techniques, applied over the Register Transfer Level (RTL) description of the system.

These methodologies address all the components of the reference architecture as the present work is doing, but portability towards other monitoring purposes seems to be impractical and would require a complete re-engineering of the discussed methodologies to enable a completely different set of metrics/statistics gathering. Basically, to the best of our understanding, despite these works are capable of feeding back at system level power-related information, the authors never tried to generalize the monitoring system to define a broader applicable methodology out of the engineering work done for power estimations.

2.2 Key Remarks and Summary of Contributions

Based on our studies we derived a list of features that an ideal monitoring system should present:

(1) It should be *minimally invasive* with respect to resources (with a low overhead on the programmable logic), to the nominal behaviour (execution should be not slowed down) and to the memory occupancy of the monitored system.
(2) It should be applicable to monitor functionalities on SW and HW in a seamless manner, which is why the proposed HW monitoring system intends to provide *homogeneous observability*.
(3) It should be flexible enough to gather information from different system components to evaluate different metrics. The HW monitoring system should be easily extendable and re-usable, and built to facilitate *heterogeneous information aggregation*.
(4) From the SW developer perspective, the access to the monitoring infrastructure should be as simple as an API call, no matter where the monitored task is executed. Lower level details on how events are observed and captured and on the involved components should be transparent to the user, thus providing *homogeneous access* to the monitoring process and data.
(5) It should be coupled to a framework/tool to enable a fast and possibly automatic integration of the monitoring infrastructure within the target system and within the executing code. Indeed, a *user friendly* design environment has to be provided.

This list is quite aligned with other lists of desiderata for monitoring systems in literature [18]. Nevertheless, the considered works (Table 1) lack in trying to embrace cross-purpose aspects and mix observability and controllability of the passive and active monitoring activities that we are trying to decouple. Moreover, the heterogeneity of the components has not been addressed.

The proposed HW monitoring system for heterogeneous FPGA-based SoPCs targets all the components of the reference architecture in Figure 1. The monitoring components are built in a similar manner, thanks to the customization capabilities of the monitors that can be specialized for the events to be captured, but the APIs used to report the information at system level are the same. Moreover, designers can play with different customization options trading off monitoring overhead (in terms of resources, time and memory) versus the observation capabilities according to

Table 1. Related Works Classification: *µProc*, *Mem.*, *Comm.* and *Copr.* are respectively Microprocessor, Memory, Communication and Coprocessor. [**a** This paper extends previous works of Valente et al. [27, 44], where microprocessor, memory and interconnection monitoring capabilities have already been demonstrated. Nevertheless, monitoring capabilities of the proposed system are proved for microprocessor, coprocessor and communication (see Transaction Level within the coprocessor).

b Quantitative results comparison with respect to this work is presented in Section 4. [c] *D&V* stands for Debug and Verification.]

References	Monitored Platform Element				Purpose			
	µProc	*Mem.*	*Comm.*	*Copr.*	*Perfor.*	*D&V*[c]	*Power*	*Security*
State of the Art Academic Solutions								
Nelissen et al. [31]	X					X		
Seo et al. [40]	X					X		
Seo et al. [41]	X					X		
Rambo et al. [35]	X				X	X		
Aldham et al. [4]	X				X			
Nadimpalli et al. [29]	X				X			
Scheipel et al. [39]	X				X			
Sadek et al. [37]	X						X	
Patrigeon et al. [34]		X			X		X	
Doyle et al. [13, 25]	X	X			X			
Valente et al. [27, 44]	X	X	X		X			
Kyung et al. [26]			X		X			
Goeders et al. [16]		X	X	X		X		
Hammouda et al [17]		X		X		X		X
Fanni et. al [15, 24]	X			X	X	X		
Lee et al. [21]	X			X		X		
Najem et al. [30]	X	X	X	X			X	
Zoni et al. [11]	X	X	X	X			X	
State of the Art Commercial Solutions								
ARM Coresight[6]	X	X	X		X	X		
Xilinx AXI PERF Mon [46][b]			X		X	X		
Xilinx ILA [45][b]				X		X		
Altera SignalTap [5]				X		X		
[**THIS WORK**]	X[a]	X[a]	X[a]	X	X	X	X	X

the addressed context. Our approach, as we already said, can certainly be considered cross-purpose since it focuses on the first two stages of the monitoring process formalized by Kornaros in [19], where the emphasis is not on metrics computation, but on gathering events to compute them.

It is worth noting that we are proposing a monitoring system targeting FPGA-based SoPCs that can be applied also to the monitoring of hardwired components. Indeed, the proposed monitor requires to access lines at RTL. When dealing with the monitoring of hardwired components, there can be two situations:

- the system manufacturer includes the proposed monitoring scheme in its RTL schematic during its design phases, finally merging it in the hardwired component;
- the lines to be monitored are accessible from an FPGA part present in the same SoC where the hardwired component lies.

In our work, we implement our monitor only on FPGAs because, except for layout activities and manufacturing, the rest of the flow is the same of hardwired components.

3 THE PROPOSED MONITORING SYSTEM

This section presents the paper contributions in detail. The first two subsections present the monitoring system and its features. In particular they:

(1) discuss the development of a HW monitoring system enforcing composability and homogeneous observability, by designing it as assembled by modules able to implement the generic monitoring process defined by Kornaros et al. [19]. Leveraging on the broadness of Kornaros' process, we ensure a wider applicability of the proposed solution;
(2) review our previous work, AdaptIve Profiling Hardware Sub-system (AIPHS) [27, 44], originally applicable to microprocessors, interconnections, and memory, making it compliant to the newly proposed HW monitoring system;
(3) present a comprehensive HW monitoring system, covering all the elements of the reference heterogeneous processing architecture, including the coprocessors.

The last subsection provides an example of implementation, presenting the design of the proposed monitoring system for a class of coprocessors existing in literature.

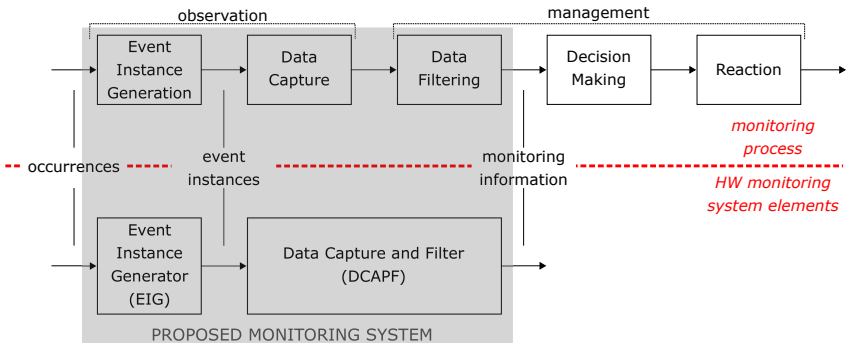

Fig. 2. The steps of a generic monitoring process [19], organized in two categories, and the corresponding HW blocks of the proposed monitoring system that implement them.

To build our monitoring system, we refer to the steps of a generic monitoring process, identified by Kornaros et al. [19], and reported in Figure 2. This figure highlights the contribution of the paper, which is the monitoring system, and the corresponding steps of the generic monitoring process it takes care of. The *Event Instance Generation* step takes as inputs the low-level signals describing occurrences happening on a platform and provides as output event instances, as defined in [31]. The *Data Capture* transforms the event instances into a meaningful representation that, after an opportune filtering in the *Data Filtering* step, can be aggregated to compute metrics (monitoring information) that satisfy given monitoring requirements. The monitoring information can be stored as result or forwarded to the *Decision Making* step, that in turn can decide whether to trigger a *Reaction* or not. To better depict the context of our work with respect to the generic monitoring process steps, we identify the two following categories:

- *Observation*: it includes the steps that extract information from executing targets, and organize them in metrics (monitoring information). The steps involved in this category are fully passive, meaning that the information is simply extracted but not used.
- *Management*: it includes the steps that, basing on the monitoring information, manage the system by applying countermeasures in case of deviations from given objectives. Then, the steps involved in this category can be considered active, since the extracted information is translated into decisions and actions over the system.

The proposed HW monitoring system mainly implements the steps part of the observation category and, for this, it can be efficiently used to support system-level methodologies that drive designers from requirements to ad-hoc monitoring systems (e.g., [41, 42]).

3.1 The Monitoring Process Implementation

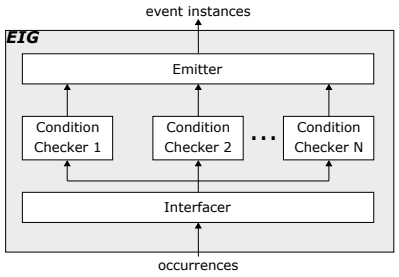

Fig. 3. EIG HW block diagram.

The *Event Instance Generation* step involves the generation of event instances when the low-level occurrences in a defined place of the target platform verify specific conditions. We implement this step by means of an Event Instance Generator (EIG) block. An EIG is composed of an Interfacer for the low-level occurrences, one or more Condition Checkers to evaluate if a specific condition is verified, and one Emitter to generate event instances (see Figure 3). The *Data Capture* step involves the management of event instances to obtain monitoring information. We implement this step and the *Data Filtering* step, with a Data CAPturer and Filter (DCAPF) block *for each metric*. A DCAPF is composed of one or more Extractors to read event instances and format them in a suitable way to be aggregated, one or more Filters of the formatted data, and one Aggregator to exploit the filtered data to evaluate a metric (see Figure 4). The EIG and one or more DCAPF (depending on how many metrics are evaluated) are part of a *sniffer*, a HW component that takes as inputs low-level occurrences and outputs monitoring information.

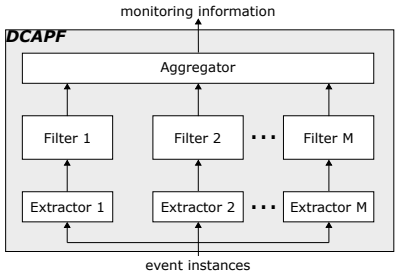

Fig. 4. DCAPF HW block diagram.

The monitoring system is composed of distributed sniffers that make observable different system resources, at different abstraction levels. Each sniffer provides, as output, monitoring information following a given structure that is independent from the place where the monitoring information is extracted.

The level of customization provided through EIG and DCAPF within each sniffer, and their derivation from a generic monitoring process, offers the possibility to cover the different metrics that can be associated to functional objectives of monitoring considered in [19]. Furthermore, the splitting of the monitor in EIG and DCAPFs allows a reuse of internal sniffer blocks among different monitoring systems.

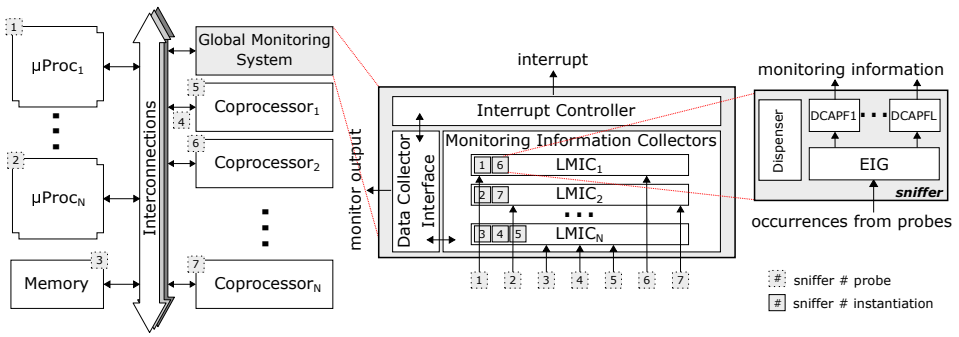

Fig. 5. HW block diagram of the complete monitoring system applied to the considered reference architecture.

The proposed sniffers deliver information to a Local Monitoring Information Collector (LMIC), as indicated in Figure 5, that controls the sniffers in a given local area, enabling or disabling them, and collecting their results.

LMICs are interfaced with the Interconnection through a Data Collector Interface (DCI) block that allows a homogeneous access to monitoring information coming from all the sniffers; to this end, the proposed monitoring system offers APIs to access monitoring information that can be used by any external host. It is worth noting that all the LMICs offer a unique shared register space. In order to efficiently bridging the steps of observation and management categories of the monitoring process (i.e., to avoid the polling of results), the proposed global monitoring system also offers the possibility to trigger an interrupt when thresholds are exceeded (by means of an Interrupt Controller, see Figure 5).

To clarify the implemented process in the creation of a sniffer, i.e., how to manage the usage of EIG and DCAPF, let us suppose that we are interested on monitoring all the writing transactions, between the addresses 0x0 and 0xF, happening on a given BUSx that connects memory mapped peripherals to multiple masters. The monitoring system will contain a single sniffer composed of an EIG and one DCAPF. In turn, the EIG will contain:

- an Interfacer, physically connected to the control and address lines of BUSx, to bring occurrences of the bus as input of Condition Checker;
- one Condition Checker, to verify whether the control lines in BUSx indicate a write. When the write happens, it is signalled to the Emitter;
- an Emitter to emit an event instance containing the address of the happened write and a sniffer identifier, when receiving the signalling from the Condition Checker,.

On the other hand, the DCAPF will contain:

- an Extractor, to receive event instances and extract useful data for the measure;
- one Filter, to check the addresses of the writes and keep the ones in the range of 0x0 - 0xF;
- an Aggregator, to aggregate the measures to get the monitoring information.

We have reviewed our previous work, AIPHS [27, 44], to make it compliant with the new proposed monitoring system. AIPHS is a HW monitoring system that observes the performance of a system by evaluating some metrics, and that is applicable to microprocessors, memories, and interconnections. Similarly to the proposed monitoring system, AIPHS is constituted of distributed sniffers controlled and read by a central element. Each sniffer is composed of an adapter, a nucleus, and an interface to receive commands and to send results. AIPHS is organized in a library fashion, in particular, there are three libraries containing elements: LIB_ADAPT for the adapter, LIB_NUCLEUS for the nucleus, and LIB_GM for the interface.

3.1.1 Implementation and Programmability Details. In this section we provide a focus on implementation details, highlighting the configuration options, for the proposed monitoring solution blocks: the global monitor, the sniffers, and the DCAPF. Along with the descriptions, we also discuss which parts have been taken from AIPHS library, and which ones are new.

The global monitor interacts, through an interface connection, with a host that controls the monitoring process and makes usage of monitoring information. The global monitor is shown in the middle of Figure 5. The DCI receives the interface connection from host as input and propagates some control lines and initialization values to the different LMICs. Also, the DCI receives the sniffers results and makes them accessible through the interface connection toward the host. The DCI, depending on how it is configured, is able to either be accessed (in a slave fashion) or to automatically write (in a master fashion) monitoring information to an external memory. At least one LMIC must be present in the system, with at least one sniffer inside it. Three sets of registers

are available for each LMIC, namely the control, the initialization, and the results registers (see Figure 6.b). The control and initialization registers are organized as reported in Figure 6.a. The register on the top is the control one, that has the two least-significant bits assigned, respectively, to *run* and *soft-reset*. Moving to the most significative bits of the register, there are two programming bits associated to each sniffer; while, moving to bottom, there is an initialization register for each sniffer for the initialization of the value of the filtering range of the DCAPF (as detailed later in the section). To match with the number of sniffers inside the LMIC both the number of initialization registers as well as the number of bits used in the control register can be customized. Supposing that the control register is equal to 32 bit size, up to 15 sniffers can be connected to the LMIC. Results registers are accessible both by sniffers (to write monitoring information) and by the host through the interface connection (to read the monitoring information). Those registers are customizable in number and size, and each sniffer has its own private area to write its results. The ensemble of the LMIC registers represents the DCI register space, shown in the right-side of Figure 6.b. The global monitor and the DCI internal construction shall be manually performed by the users, but we provide a library of IP-cores to facilitate this action: in particular, we included in the new library different IP-cores already available from LIB_GM of AIPHS, adding more IP-cores to provide the DCI with the capability of writing monitoring information toward a destination memory in a master fashion. In addition, the DCI now implements the management of *Catch* and *Wack* signals (detailed in the next paragraph), in order to better separate the communication among sniffers and the entities accessing (e.g., a microprocessor) the monitoring information.

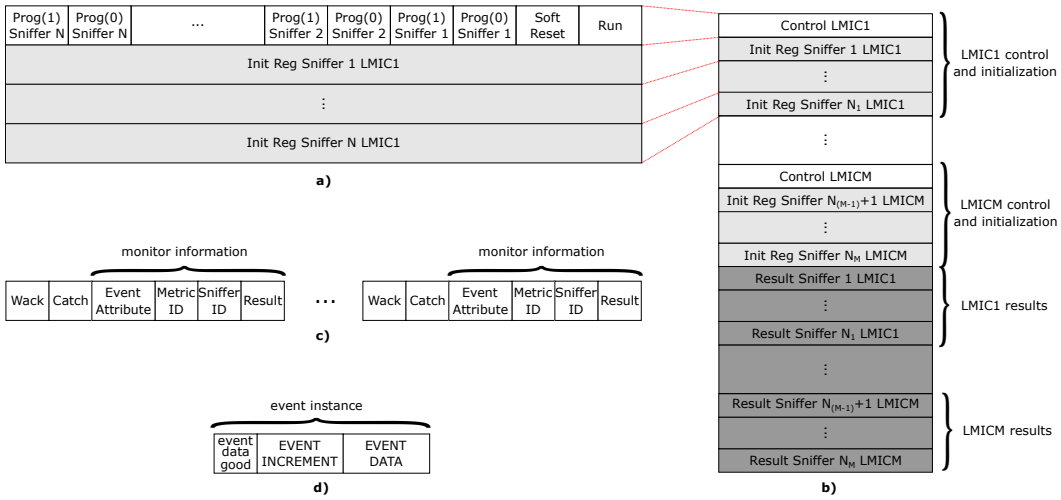

Fig. 6. Data organization in the proposed monitoring system: a) control and initialization registers of the LMICs; b) DCI register space; c) structure of the sniffer output; d) structure of an event instance.

The sniffer is shown in the right side of Figure5. It receives the occurrences as input, together with the control and initialization bus, and provides different monitoring information as output (depending on the number of DCAPFs), together with the *Catch* and *Wack* signals. The structure of the sniffer output is shown in Figure 6.c: each sniffer is able to output some monitoring information, associated to one or more DCAPFs. In particular, for each DCAPF, the sniffer outputs three information: *Wack* signal, *Catch* signal, and monitoring information. The latter is further divided in Event Attribute, that refers to attributes associated to the monitored events, Metric ID and Sniffer ID, that refer to the ID of the evaluated metric and the sniffer that provides it, respectively, and

Result, that refers to the metric value. *Catch* is high when the monitoring information is ready to be stored in the register of the DCI, while *Wack* is high when the monitoring information is ready to be accessed from the host. The necessity of two distinct phases, managed by *Catch* and *Wack* signals, comes from the fact that there are cases where the monitoring information is written in multiple steps, and only at the end it can be accessed. The programming bits for each sniffer have the following meaning:

$$PROG = \begin{cases} 00, & \text{IDLE - the sniffer is not controlled neither by run nor soft-reset} \\ 01, & \text{INIT - the sniffer initializes its internal DCAPF} \\ 10, & \text{FILTERING - the sniffer works using its internal filters} \\ 11, & \text{NO-FILTERING - the sniffer works without using its internal filters} \end{cases}$$

When the *run* signal (coming from the LSB of Figure 6.a) is high, all the sniffers in FILTERING or NO-FILTERING mode connected to the corresponding LMIC start working. When the *soft-reset* signal (again coming from control register of Figure 6.a) is high, all the sniffers not in IDLE mode are internally reset. The EIG receives the occurrences as input and produces the event instances as output, as described above. Event instances have a well-defined structure, reported in Figure 6.d, where EVENT DATA represents the value associated to the monitored event, EVENT INCREMENT is the increment associated to the monitored event, and *event_data_good*, when high, ensures the validity of the event instance. The ratio behind the event instance structure is to produce an interconnection independent interface that provides both the EVENT DATA and a weight to count it (EVENT INCREMENT). The EIG strongly depends on the monitored interconnections (i.e., where the sniffer is connected and which kind of lines need to be monitored). The development of EIG internal blocks is left to users, and to support them we provide libraries with interfacers, condition checkers, and emitters IP-cores. In particular, we include in the new library different IP-cores already available from LIB_ADAPT of AIPHS, adding new IP-cores to interface with AXI4-Full bus and to extract information from burst-based communication buses. Furthermore, as explained in the next section, we add also IP-cores to monitor coprocessors. The description of sniffer internal blocks ends with the dispenser, that receives some control lines (PROG, *run*, and *soft-reset*) and the initialization value associated to the sniffer, and propagates it to the DCAPF inside the sniffer. Each DCAPF requires two initialization values: due to the fact that the initialization register is unique for the sniffer, in order to initialize all the DCAPFs is required a number of writes that is two times the number of DCAPFs.

The DCAPF is shown in the middle of Figure 7. It receives control and initialization information, together with event instances, and produces monitoring information as output, together with *Catch* and *Wack* signals. With reference to Figure 7, the Event Monitor and Time Monitor are two blocks that represent two different Extractors. Both of them also contain the filters, as shown in the same Figure 7, on the left and on the right side. The Init DCAPF block stores the initialization values that represent the low and top of the range where filters work. The Data Gating block can be used if it is necessary to perform some gating actions on event instances, while the Aggregator block has been already discussed above.

The Event Monitor and Time Monitor are both shown in Figure 7, on the left and right side respectively. The Event monitor is able to count the number of event instances: depending on the presence or not of the filter, it also checks whether the EVENT DATA fits inside a range delimited by INF and SUP input values (they are included in the range). The Event Capture block ensures to add the event instance with the corresponding weight (EVENT INCREMENT) to the total event count (done with a counter). The Time Monitor has the same blocks of the Event Monitor, apart for Time

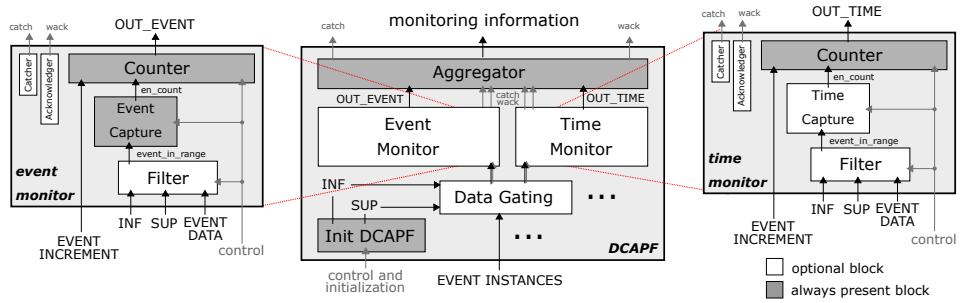

Fig. 7. DCAPF internal structure with a focus on time and event monitors.

Capture, that is able to measure the time spent within a range of values (selected using the Filter block). Finally, the Catcher and Acknowledger blocks provide the *Catch* and *Wack* signals. The development of DCAPF shall be made by users, even though leveraging on provided libraries with extractors, filters, and aggregators. In particular, the available extractors are the Time Monitor and the Event Monitor, discussed above, that have been taken from the LIB_NUCLEUS of AIPHS and evolved toward a more reusable and configurable structure. In addition, both IP-cores optionally contain also catchers and acknowledgers.

The proposed monitoring system is highly configurable, thanks to the proposed libraries of IP-cores and their level of configurability. The IP-cores are provided through a VHDL description, and they can be configured using VHDL generics: they are all stored inside a VHDL configuration file, named *config.vhd*. Some of these generics are reported here below to build an example sniffer here named *WOT* (0 means that the block is not instantiated, 1 vice versa):

```
-- from the LSB, configure data gating block, time monitor, and event monitor
constant DCAPF_WOT_CONFIG: unsigned(4 downto 0) := "000";
-- from the LSB, configure filter, time capture block, catcher, and acknowledger
constant TIMEMON_WOT_CONFIG: unsigned(4 downto 0) := "0000";
-- size of the counter inside the event monitor
constant size_WOT_count_out_event: integer := 64;
-- size of the counter inside the time monitor
constant size_WOT_count_out_time: integer := 64;
```

APIs to use the monitoring system after its implementation are also available. An example of them is reported here below (the APIs details can be accessed in the jointer open-source repository [2]):

```
// initialize the selected DCAPF of the selected sniffer
void jointer_initialize(parameters)
// run sniffers of selected LMIC (works on FILTERING/NO-FILTERING sniffers)
void jointer_run(parameters)
// reset sniffers of selected LMIC (works on not IDLE sniffers)
void jointer_reset(parameters)
// print all the result registers of the selected LMICs
void jointer_print_reg(parameters)
```

3.2 Extension to cover coprocessors

To extend the applicability of the proposed monitoring system to coprocessors, we leverage on a high-level architecture of a coprocessor, which internal architecture is divided in three main parts: *Transaction Handler*, *I/O Manager* and *Data Cruncher*. The Transaction Handler is responsible of interfacing each coprocessor with the interconnection backbone it is attached to. The I/O Manager is responsible of transferring the input data to the computing core of the coprocessor, which is the Data Cruncher, and to collect from it the results that will be transferred back to the host microprocessor through the Transaction Handler. Such a generic coprocessor architecture is quite common and it is adopted by different FPGA based computing heterogeneous platforms that embed coprocessing units [20, 32, 36]. In such architectures three monitoring levels are possible:

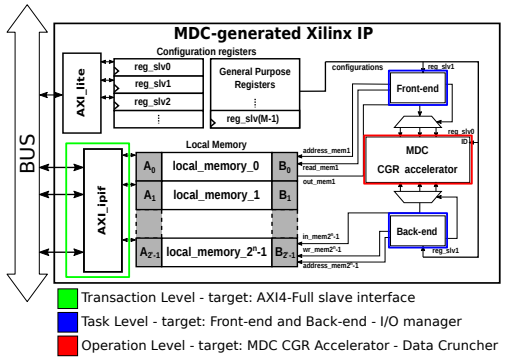

Fig. 8. MDC-based coprocessor, highlighted blocks are associated to different monitoring levels.

- *transaction level* - it applies to the logic within the Transaction Handler and is referred to the monitoring of all the HW elements inside the coprocessor that manage the transactions to interact, through the Interconnection, with other components of the reference architecture of Figure 1. As an example, the logic necessary to handle memory-mapped or stream-based transactions towards/from the microprocessors is monitored at transaction-level.
- *task level* - it applies to the logic within the I/O Manager and is referred to the monitoring of all the HW elements inside the coprocessor that manage the start, the stop, and the I/O data of the executed HW-task. As an example, data movement from the coprocessor internal memory to/from the Data Cruncher are monitored at task-level.
- *operation level* - it applies to the logic within the Data Cruncher and is referred to the monitoring of all the HW elements, at any granularity, inside the coprocessor that perform the computations associated to the HW-task. As an example, the individual functional units within the computing core of the coprocessor can be monitored at operation-level to check whether they represent a bottleneck for the computation by monitoring their busy/idle time.

By properly configuring the sniffers for the given coprocessors, we introduce a sniffer for each of the three levels that depends on (i) the occurrence to be observed and (ii) the metrics to be evaluated. For instance, for a HW monitoring system able to evaluate four different metrics, associated to a monitor for debugging [19] of the HW-tasks executed on a coprocessor, the metrics are expressed as follows:

- $METRIC_1^D$ - total size of data transfers;
- $METRIC_2^D$ - number of internal custom events referred to a generic computation;
- $METRIC_3^D$ - time spent to perform a data transfer;
- $METRIC_4^D$ - time spent to perform a computation;

The four metrics can be computed by collecting the following event instances:

- for $METRIC_1^D$: a data transfer and the related amount of bytes is needed (transaction level);
- for $METRIC_2^D$ an event inside the Data Cruncher is needed (operation level);

- for $METRIC_3^D$ start and end of a data transfer are needed (transaction level);
- for $METRIC_4^D$ start and end of a computation within the coprocessor are needed (task level).

The four metrics can be computed by means of the three sniffers for the coprocessor monitoring.

If multiple clock regions are present in the system, very common situation in heterogenous platforms, the proposed approach is capable of collecting, storing and retrieving monitoring information in a consistent way. Coprocessors, which are one of the possible source of events/information in the proposed monitoring solution, usually run at different frequencies than microprocessors, that can act, for example, as monitoring information retrievers. According to the reference architecture depicted in Figure 5, sniffers connected to coprocessors extract event occurrences and extract monitoring information from them at the same frequency they are generated. Such information is then properly collected and stored by LMICs, and made available to the microprocessor through DCI, which can access to this latter at its own frequency. Thus, information collection and retrieving are completely decoupled, and can run at different speeds without problems, also leveraging on the possibility of adopting an interrupt mediated communication if reactivity is crucial.

In the next section, the design of the three sniffers for a class of coprocessors available from literature is reported, in order to highlight the reuse of components inside each sniffer.

3.3 Implementation over a real coprocessor infrastructure

This section provides an example of implementation, over a microprocessor-coprocessor system, of the described HW monitoring system extension presented in Section 3.2.

The class of coprocessors for which we implement the monitoring system is represented by coprocessors provided by the Multi-Dataflow Composer (MDC) tool, a dataflow to HW tool that automatically generates Coarse Grained Reconfigurable (CGR) HW designs [38]. Dataflows are oriented graphs whose nodes, the actors, are accounted for data processing, while the edges are point to point buffered communication channels. MDC takes as input dataflow network(s) specifications, combines them if they are more than one, and generates a ready-to-use IP. Depending on the number of input dataflows, MDC generates a non-reconfigurable (one dataflow) or CGR (more than one dataflow) coprocessor. In this second case, different dataflow specifications are merged together and different functionalities are enabled by multiplexing resources in time.

MDC has already been proven to be effective for the automatic instrumentation of the generated HW custom monitors. In the previous work of Fanni et al. [15], MDC-compliant coprocessors have been instrumented to be monitored, as already mentioned in Table 1. The HW block diagram of an MDC-generated coprocessor for Xilinx oriented platforms is reported in Figure 8, where we highlight the Transaction Handler, the I/O blue, and the Data Cruncher with green, black, and red colours, respectively. The coprocessor has an assigned data-set in memory and, when required by the host microprocessor, it streams data through a DMA, performs the computation within the Data Cruncher, and writes back results direct to memory. MDC-compliant coprocessors are interfaced with the rest of the computing platform through an AXI4 bus [7]. Therefore, the interface with the host microprocessor and with the on board memory is an AXI4 Slave Interface, representing the Transaction Handler block. Moreover, MDC-compliant coprocessors present a front-end and a back-end that, according to the chosen dataflow-based actor to actor communication protocol, are responsible of inputs management and outputs retrieval. Front-end and back-end, representing together the I/O manager, can be monitored to get task level information. HW tasks are executed within the MDC CGR accelerator that acts as the Data Cruncher.

In order to perform the monitoring process to compute the four metrics of the example reported at the end of the previous subsection, we designed the three sniffers capable of:

- collecting the total number of written and read bytes through the AXI4 Slave interface, monitoring transaction level information;
- measuring the time between the start and the end of a computation, monitoring task level information;
- collecting the total number of defined events, monitoring operation level information.

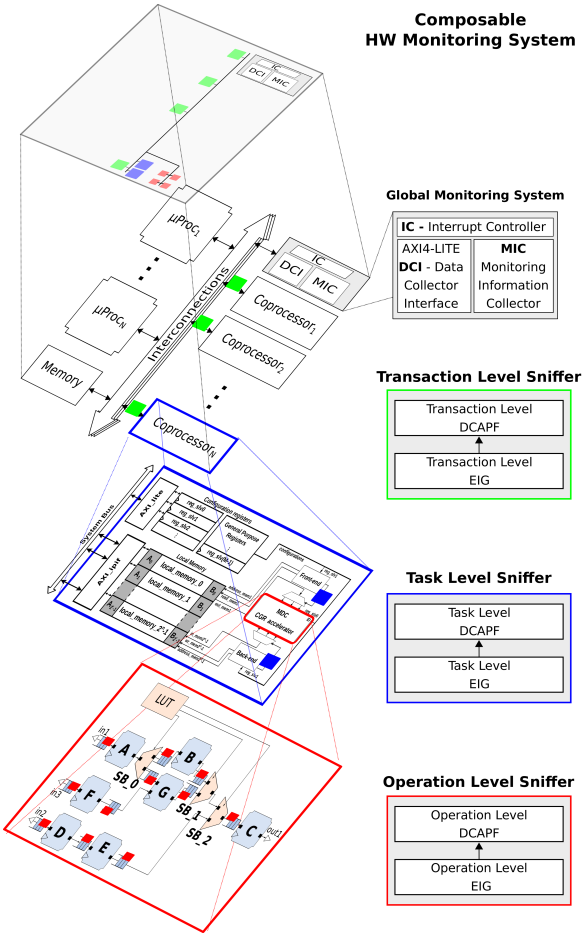

Fig. 9. Example of the proposed HW Monitoring System.

All the sniffers deliver their monitoring information to a single LMIC, while the DCI is an AXI4 Lite slave interface [7]. The host can access resulting monitoring information through a dedicated AXI4 Lite bus.

The transaction level sniffer has an EIG that takes as input low level AXI4 bus occurrences and provides as output event instances containing the address accessed for writing and reading as EVENT DATA, together with the associated number of bytes related to write/read bursts as EVENT INCREMENT. The sniffer has one DCAPF that accumulates the number of total written/read bytes that are performed on a predefined range. The block contains the Init DCAPF, an Event Monitor, and the Aggregator. In turn, the Event Monitor contains the Filter, the Event Capture, the Counter, and the Catcher.

The task level sniffer has an EIG that takes as input low-level coprocessor transactions associated to the Front-End and Back-End, and provides as output event instances containing the indication of a start of computation and end of computation (both as EVENT DATA). The sniffer has one DCAPF that measures the time between the start and the end of the computation, and provides that metric as output. The block contains the Init DCAPF, a Time Monitor, and an Aggregator. In turn, the Time Monitor contains only the Time Capture and the Counter.

The operation level sniffer has an EIG that takes as input low-level coprocessor transactions associated to the CGR area, and provides as output event instances containing the number of events occurrences (with the EVENT INCREMENT) associated to specific events. The sniffer has multiple DCAPF, each one performing the counting of specific events. Each block contains the Init DCAPF, an Event Monitor, and an Aggregator. In turn, the Event Monitor contains the Event Capture and the Counter.

The final monitoring system, applied to MDC-based coprocessors, is reported in Figure 9. In this work, the implementation of the proposed HW monitoring system for coprocessors is partially

automated. In particular, the coprocessor has been automatically generated and integrated in the microprocessor-coprocessor system using the MDC tool, while a Python script allows to modify the monitor configuration through a GUI, and, taking also into account the *config.vhd* file, to integrate the described sniffers, modifying the existing microprocessor-coprocessor system. Further details are reported in Section 4.2.3.

4 ASSESSMENT

In this section, we are going to assess the proposed monitoring system for the processing and coprocessing units considering a Xilinx Zynq-7000 XC7Z020 SoPC [47] (the coprocessing unit is compliant with the reference architecture presented in Figure 1, as discussed in Section 4.1). The assessment considers the dual-core ARM microprocessor and MDC-compliant coprocessors, and is performed in terms of provided degrees of freedom and customization possibilities on a simple fixed function coprocessor (see Section 4.2), and in terms of delivered benefits with respect to literature solutions on a more complex system with microprocessor and reconfigurable coprocessor (see Section 4.3).

4.1 Target Architecture and Experimental Setup

Figure 10 illustrates as the adopted device maps the elements of the reference architecture presented in Figure 1. In particular, the microprocessors are the two ARM Cortex-A9 cores, and the coprocessor is an MDC generated IP, as illustrated in Figure 8, connected to the microprocessors by means of a communication infrastructure that involves bus and Direct Memory Access engines (not represented for the sake of simplicity). The MDC CGR accelerator (please, refer to Figure 8) is the coprocessor Data Cruncher that, in this paper, can execute two applications, the *Selective Accumulations* (see Section 4.2) or the *Edge Detection* (see Section 4.3) one. Please note that, the main memory (Data Mem in Figure 10), where data to be processed and results are stored, for Resources and Dynamic Power data has been substituted with an AXI accessible *BRAM* memory. In all the other cases, it is an external DRAM. The monitoring system is composed of four sniffers, three for the MDC-based coprocessor, described in the previous section, and one for the ARM microprocessor. In particular, the sniffer for ARM is a sniffer that is able to observe a microprocessor that sends commands to an AXI4-Lite slave component. The AXI4-Lite slave component is represented by our monitoring system: when the microprocessor writes a value VAL in a specific memory location, the sniffer gets a timestamp and store a monitoring information constituted of VAL + TIMESTAMP in a dedicated memory. The dedicated memory is a BRAM (Tst Mem in Figure 10), and the microprocessor can access the monitoring information stored in BRAM again using the AXI4-Lite bus. A similar sniffer has been used in one of our previous works [28]: here, we reuse it by slightly changing the EIG block. All the sniffers are controlled by a single LMIC, while the DCI contains sixteen registers (each with a size of 32-bit). One register is left for the control associated to LMIC, while four are left for initialization of the four sniffers. The remaining eleven registers are used for the storage of monitoring information. In particular, the monitoring information related to the MDC-coprocessor are accessible by the host directly in the DCI registers, while the monitoring information related to the monitor for ARM are further stored in Tst Mem. The communication between the host (here, the ARM core microprocessor) and the global monitor is a dedicated AXI4-Lite bus. The host can access, using the same bus, to monitoring information stored in Tst Mem. Further details for the sniffers for MDC-based coprocessor are reported in the following:

- transaction level sniffer: collects the total number of written bytes by the coprocessor to the external memory. The size of the counter inside the Event Monitor is 23-bit. The sniffer writes its results to one of the DCI registers;

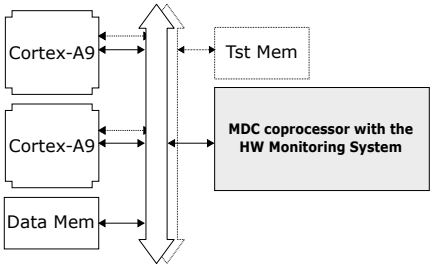

Fig. 10. Mapping of the Xilinx Zynq-7000 XC7Z020 over the reference architecture of Figure 1.

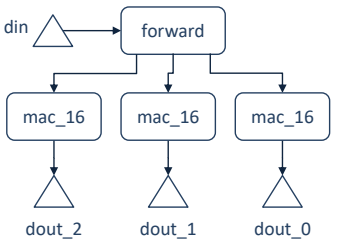

Fig. 11. Dataflow model of the *Selective Accumulations* application.

- task level sniffer: measures the duration, in terms of clock cycles, of the computation associated to the HW-task performed by the coprocessor. The size of the counter inside the Time Monitor is 53-bit size. The sniffer writes its results to two of the DCI registers, splitting the least and most significant parts of the result;
- operation level sniffer: collects the total number of selected events inside the Data Cruncher. The information on which are the selected events to be monitored, together with size of monitoring information, is reported below (since they vary from test to test).

The experiments reported hereafter have been carried out with the Xilinx Vivado design suite. Resources occupation data come from a full implementation of the considered designs under test, while the power consumption data have been obtained through Vivado power estimations by considering the real switching activity of the system, gathered during post-implementation functional simulations. Execution latency and memory footprint data have been measured during real executions on board. To reduce the statistical error, each latency number is computed as the average among 10 executions of the same design and configuration. Memory footprint numbers come from the actual memory occupancy of the SW part of the considered applications and from the amount of data due to HW-task execution and monitoring. The proposed monitoring system is designed in a way that it does not contain the system critical path (i.e., the addition of the monitoring system in the design does not impact in the maximum system clock frequency).

4.2 Exploration of the Monitoring Solutions

For a complete exploration of the possible monitoring solutions enabled by the proposed monitoring system, we here focus on the monitor customized to contain only the sniffer for MDC-based coprocessor, and we adopt an application that executes *Selective Accumulations* of input data to build a simple fixed function coprocessor. Figure 11 illustrates the dataflow representation of that application, which is composed of two actors:

- $forward$ is responsible of data selection: it forwards data on a different output depending on the same data value (output 0 if input is bigger than 5, output 1 if input is between 0 and 4, output 2 if input is smaller than 0);
- *mac*_16 is in charge of multiplying the input for a given constant and of accumulating together the result of 16 subsequent products (one different instance of such actor is connected to each output of $forward$).

The resulting HW coprocessor is the object of the assessment for the proposed monitoring.

4.2.1 Designs Under Test. In this exploration, we compare 10 different versions of the design under test (see Table 2). Different monitoring solutions are evaluated in different combinations of

Table 2. Designs under test for the *Selective Accumulations* HW coprocessor involved in the exploration of the monitoring solutions (*trans.* and *oper.* are respectively *transaction* and *operation*).

Design	Enabled Monitor Level			Operation Level Configuration		
	trans.	*task*	*oper.*	#events	#bits per event	#regs
Y0	-	-	-	-	-	-
Y1	x	-	-	-	-	-
Y2	-	x	-	-	-	-
Y3	-	-	x	1	5	1
Y4	-	-	x	2	10	1
Y5	-	-	x	3	20	3
Y6	x	x	-	-	-	-
Y7	x	-	x	2	10	1
Y8	-	x	x	2	10	1
Y9	x	x	x	2	10	1

sniffers and in different configurations to explore the possibilities available to the user to customize the monitoring system. Column *Enabled Monitor Level* illustrates the different combinations of sniffers (*transaction*, *task* and *operation* levels) that have been compared (*Y0* is the design without the monitoring system). Column *Operation Level Configuration* gives the details of the operation monitoring level. Column *#events* reports the number of different Data Cruncher events the sniffer is able to monitor. For each event, there is a dedicated DCAPF within the sniffer: column *#bits per event* reports the counter size of the Counter inside each Event Monitor contained in the DCAPF. Column *#regs* reports the register mapping. Please, notice that the adopted transaction level and operation level sniffers are a subset of the possible configurations. Indeed, the transaction level monitoring can be applied for different aspects of AXI transactions (e.g., number of read bytes or different address ranges), while the operation level monitoring can be applied for whatever event the designer needs to keep trace of within the HW coprocessor Data Cruncher. On the contrary, at the moment, besides counting events or monitoring time, the task level sniffer cannot be configured in a different manner.

Table 3. Resources and power of the *Selective Accumulations* designs. *Exec.* refers to power consumption during coprocessor execution, while *Transf.* refers to power consumption during data transfer periods.

Design	Resources						Dynamic Power [mW]			
	LUT	*LUT%*	*FF*	*FF%*	*BRAM*	*BRAM%*	*Exec.*	*Exec.%*	*Transf.*	*Transf.%*
Y0	3397	-	2864	-	6	-	24	-	25	-
Y1	3675	+8.18	3163	+10.44	6	+0.00	25	+4.17	26	+4.00
Y2	3497	+2.94	3092	+7.96	6	+0.00	25	+4.17	26	+4.00
Y3	3478	+2.38	3043	+6.25	6	+0.00	26	+8.33	26	+4.00
Y4	3491	+2.77	3058	+6.77	6	+0.00	25	+4.17	26	+4.00
Y5	3516	+3.50	3098	+8.17	6	+0.00	25	+4.17	26	+4.00
Y6	3702	+8.98	3217	+12.33	6	+0.00	27	+12.50	28	+12.00
Y7	3690	+8.63	3183	+11.14	6	+0.00	25	+4.17	27	+8.00
Y8	3512	+3.39	3112	+8.66	6	+0.00	26	+8.33	26	+4.00
Y9	3718	+9.45	3237	+13.02	6	+0.00	26	+8.33	28	+12.00

4.2.2 Quantitative Results. Table 3 depicts resource occupancy and power consumption results of the considered designs. Since the monitors are register-based, they present a cost in terms of *LUT*

Table 4. Latency and memory footprint of the *Selective Accumulations* designs. *Instr.* is related to the whole executable file and *Data* is to the real disk occupancy of the data for processing and monitoring.

Design	Latency [us]				Memory [B]			
	Tot.	*Tot.%*	*Proc.*	*Mon.*	*Instr.*	*Instr.%*	*Data*	*Data%*
Y0	10.981	—	—	—	229988	—	204	—
Y1	14.356	30.73	10.839	4.617	235104	2.22	207	1.29
Y2	14.736	34.20	10.804	5.128	235112	2.23	211	3.25
Y3	14.161	28.96	10.597	4.692	235108	2.23	205	0.31
Y4	14.831	35.06	10.853	5.207	235172	2.25	207	1.23
Y5	15.122	37.71	10.806	5.365	235236	2.28	212	3.68
Y6	15.083	37.36	10.861	5.278	235172	2.25	213	4.53
Y7	15.172	38.17	10.813	5.593	235236	2.28	209	2.51
Y8	15.08	37.33	10.567	5.685	235240	2.28	213	4.47
Y9	15.338	39.68	10.574	5.945	235304	2.31	216	5.76

and *FF*, without any impact on the *BRAM* (we remark that we are considering only the monitor of MDC-based coprocessor here). This overhead is always under the 10% for *LUT* and 13% for *FF*. The lowest monitoring overhead is achieved by the *Y3* design, which employs only the operation level monitoring with 1 event (column *#events*) and a 5 bits counter (column *#bits per events*). The transaction level monitoring is the most resource hungry solution, since it embeds all the logic necessary for AXI protocol decoding and monitoring. Of course, when more than one monitoring level is enabled, the amount of overhead resources grows, up to the case where all the monitoring levels are employed (*Y9*).

Dealing with power numbers, a preliminary clarification is needed: the resolution of the power estimation is 1 mW, so that differences lying below this minimum value are lost. Power consumption presents only slight variations among different designs, variations sometimes even not expected, such as for *Y6* and *Y9* where this latter is consuming less than the former, which however demands more resources. It is mainly due to different synthesis and implementation choices that lead to optimizations under different goals: resource and/or timing for *Y6*, power for *Y9*. In terms of data transfer power (see *Transf.* column in the table), *Y6* and *Y9* have the same behaviour. Please note that power results almost follow the same trend of the resource ones, since *Y6* and *Y9* are still the most resource demanding solutions. Overall, the power overhead is small, since it is lower than 12.5% for both *Exec.* (coprocessor execution) and *Transf.* (data transfer) periods.

Table 4 depicts numbers gathered from real execution of the designs under test on the evaluation board. Since the considered application is quite simple and small, the monitoring overhead on the execution latency has a clearly visible weight, causing always about 30% additional time on the Total duration (*Tot.%*). Again, *Y9* presents the highest overhead in terms of latency: all the three monitoring levels have to be configured and *216 B* of monitoring information have to be retrieved. In this case, the overhead is close to 40% of the overall execution latency. The most lightweight solution is, instead, *Y3*, where the minimal operation level monitoring is enabled. Looking at the *Proc.* and *Mon.* columns, showing processing and monitoring contribution on the overall execution latency, it is clear how the processing latency is always almost the same, with small variations due to statistical effects. The monitoring latency, instead, is following the monitor complexity for each design, and it is in line with resource and power numbers depicted in Table 3.

Table 4 depicts also memory footprint split in instruction (*Instr.*) and data (*Data*), to separate the overhead of the monitoring solutions drivers and of the additional monitoring data. From these entries it is clear that the memory footprint overhead is very limited, being always under 2.5% and

5.8% respectively for instructions and data memory. The trend is in line with the amount of events to be monitored and with the amount of monitored data to be gathered. So that, *Y9* is the most memory demanding solution under both the considered metrics, while *Y1* is the less demanding solution for instruction memory (contrarily to *Y3* and *Y4*, all adopting one single register for mapping counters, *Y1* does not require data masking since the counter width, 32 bit, is the same of the data bus) and *Y3* is the less demanding solution for data memory, requiring only one more byte than *Y0* design.

Despite its simplicity, the *Selective Accumulations* test case gives an idea of the little invasiveness of the proposed monitoring system, especially in terms of resources, power and memory footprint. Execution time overhead is, instead, not negligible in general for this specific test case. A more realistic idea of the invasiveness of the proposed monitoring system will be given in Section 4.3, where a real application is considered.

4.2.3 Qualitative Results. It is worth to discuss also qualitatively the exploration made on the *Selective Accumulations* application. In particular, referring specifically to the chosen implementation flow based primarily on the MDC tool, the considered monitoring solutions can be evaluated also in terms of *designer effort* and required *skills*. As reported in Section 3.3, the implementation of the proposed HW monitoring system is partially automated through an external script, providing a graphical user interface, that allows the instrumentation of both transaction and task level monitoring of the generated coprocessor. This means that the user does not need a specific knowledge and understanding of HW design to place sniffers in these levels: it is sufficient to understand the monitoring solution register mapping, that is anyway hidden by the monitor drivers, but that can be useful for advanced actions and optimizations on the code. Dealing with the operation level sniffer, the discussion is a bit different since the automation is only partial. In fact, this monitoring level is strongly custom, meaning that it depends on the specific application constituting the computing core of the HW coprocessor, and it offers the possibility of tracing any internal signal. Making the selected signals available at the interface and accessible by the monitoring logic is not an automated step, and the users have to modify the RTL code by hand. Nevertheless, the monitoring logic has to be only partially modified, and automation scripts for monitoring up to four events are already available, even if small modifications are required when the Data Cruncher changes or when more than four events have to be monitored. Please, note that customization and freedom in monitoring is colliding with lowering effort and skills required to the designer. Indeed, in order to be able to choose which signals and which events have to be monitored inside the coprocessor Data Cruncher, users have to know where and how to put their hands in the code. As it will be discussed in the conclusion, future automation activities are foreseen to support also a more designer friendly usage of the operation level sniffers in MDC-compliant coprocessors, to avoid such an extra effort and required low level skills.

Finally, the proposed monitoring system brings also benefits in terms of usability. Indeed, having a global monitoring system with different internal LMICs, that in turn aggregate all the data coming from all the sniffers associated to all the monitored levels and not only the ones related to coprocessors (see Section 3), implies that the user has a unique and uniform access point for monitored data, regardless of their source, size and kind. This improves usability, on the one hand, while facilitates, on the other, the perspective implementation of the active steps of the generic monitoring process envisioned by Kornaros et al. [19]. Unquestionably, multi-objective run-time optimization strategies could certainly be enabled within a system run-time manager if the metrics are grouped together within the global monitor.

As a summary, the design automation and usability of the proposed approach can be illustrated as follow:

(1) user shall model application(s) as dataflow(s) to feed the MDC tool and shall provide Hardware Description Language (HDL) descriptions for the involved actors/modules (HLS can be exploited to provide high level descriptions, e.g., C codes, to automatically generate the corresponding HDL instead of manually define each block);
(2) MDC automatically generates the corresponding loosely coupled fixed point or reconfigurable (according to the number of provided dataflow models of the applications) coprocessor, drivers for its easy usage, and scripts for generating the complete microprocessor-coprocessor system within Vivado environment;
(3) a script takes the MDC generated platform, inserts the proposed monitoring system for the coprocessor, on the three available levels according to the user preferences, and generates drivers for it. Considering the operation level, if any event has to be monitored, the user shall manually bring those events outside the coprocessor and shall modify the input to the provided script accordingly;
(4) user shall develop the software application using the available drivers and APIs (available both for the coprocessor task delegation and management and for the monitor management).

4.3 Benefits of the Monitoring Solutions

In this section, we adopt a real application involving *Edge Detection* algorithms [12] to assess the benefits of the proposed monitoring system applied to a system involving a microprocessor and a complex reconfigurable coprocessor. In particular, discrete first-order differentiation *Edge Detection* algorithms are considered. These algorithms detect the edges of a certain object according to the difference of colour intensity among neighbour pixels, that is the gradient of colour variations (G) in a certain neighbourhood of pixels. Such gradient is obtained as the sum of the horizontal and vertical gradients, resulting from the convolution of two-dimensional kernels with the source image. In this case, we adopt two different algorithms, *Sobel* and *Roberts*, which employ convolution kernels with different sizes, $3x3$ and $2x2$ pixels respectively. Such algorithms provide different trade-offs in terms of detection power and efficiency (resources, power and execution time). *Sobel* is capable of detecting more edges than *Roberts*, but requires more resources, power and time; while *Roberts* is simpler, thus adopting less resources, power and time, but it detects less edges.

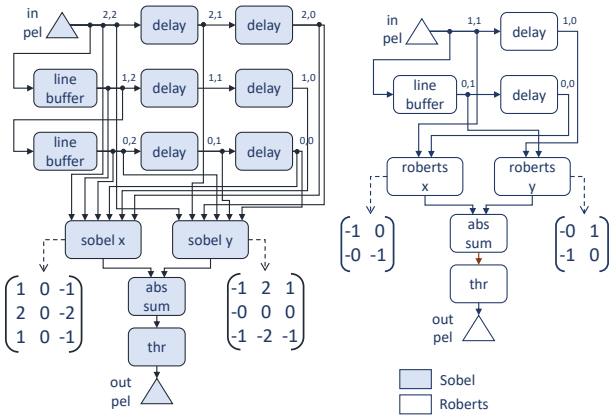

Fig. 12. Graphs representation of the *Sobel* and *Roberts* edge detectors.

Figure 12 illustrates the dataflow representations of the *Sobel* and *Roberts* edge detectors:

- *line buffer* actors to memorize one previous row within the image;
- *delay* actors to memorize one previous pixel within a row;
- *sobel x/y* and *roberts x/y* actors to perform the convolution of the specific algorithm kernel with the $3x3$ and $2x2$, for *Sobel* and *Roberts* respectively, image portion reconstructed by *line buffer* and *delay* actors, thus calculating horizontal and vertical gradients;

- *abs sum* actor to perform addition of horizontal and vertical gradients to obtain the overall gradient for the specific pixel position;
- *thr* actor to actually decide if an edge is detected or not in the specific pixel position: if the gradient is above a certain threshold (80), an edge is present and the output will be saturated to the maximum value (255), while if the gradient is below the threshold, there is not any edge and the output is forced to the minimum value (0).

The two dataflow models of *Sobel* and *Roberts* have been combined together by the MDC tool, the resulting CGR coprocessor is the object of the assessment for the proposed monitoring.

4.3.1 Designs Under Test. Differently from the exploration of the possible monitoring solutions made in Section 4.2, here we focus only on three monitoring configurations, each allowing for a different level of knowledge of the system. To understand the benefits of the proposed generalized monitoring solution, a design combining accelerator and microprocessor monitoring has been also analysed. The considered configuration summary is the following.

- $D0$ designs without the monitoring system;
- $D1$ designs provide transaction level monitoring;
- $D2$ designs provide transaction level and task level monitoring;
- $D3$ designs provide transaction level, task level and operation level monitoring.
- $D4$ designs provide all the previous coprocessor monitoring levels, along with the microprocessor software monitoring.

Table 5. Designs under test for the *Edge Detection* coprocessor.

Design	Enabled Monitor				Monitor Solution			
	$\mu Proc$	Coprocessor			$\mu Proc$	Coprocessor		
		trans.	*task*	*oper.*		*trans.*	*task*	*oper.*
$D0$	-	-	-	-	-	-	-	-
$D1_a$	-	x	-	-	-	[46]	-	-
$D1_b$	-	x	-	-	-	proposed	-	-
$D2_a$	-	x	x	-	-	[46]	[45]	-
$D2_b$	-	x	x	-	-	proposed	proposed	-
$D3_a$	-	x	x	x	-	[46]	[45]	[45]
$D3_b$	-	x	x	x	-	proposed	proposed	proposed
$D4_a$	x	x	x	x	built-in [47]	[46]	[45]	[45]
$D4_b$	x	x	x	x	proposed	proposed	proposed	proposed

The proposed HW monitoring system adoption ($D1_b$, $D2_b$, $D3_b$ and $D4_b$) is compared with state of the art monitoring systems widely adopted on FPGA platforms ($D1_a$, $D2_a$, $D3_a$ and $D4_a$)) (see Table 5). In particular, such monitoring systems are based on available Xilinx IPs within the considered Vivado Design Suite environment and on the ARM core built-in global timer/counter [47]. The transaction level monitoring, acting on the system interconnection (system bus, hereinafter), is compared with the Xilinx AXI Performance Monitor [46], which is the official Xilinx IP for monitoring AXI buses. This IP offers basically the same possibilities of the proposed transaction level monitoring solution, allowing the profiling of different events related to the AXI protocol, such as counting reads and writes or configuring the considered address range. The task level and operation level monitoring are compared with the Xilinx Integrated Logic Analyzer (ILA) [45]. This IP, differently from AXI Performance Monitor, allows custom monitoring of any signal on the system netlist (thus after RTL synthesis of the design). In such a way, it is possible to monitor both Front-End/Back-End signals (task level monitoring) and MDC CGR coprocessor Data Cruncher level

signals (operation level monitoring). Please notice that such terms of comparison are a forced choice for the considered target device, since they are the monitoring/debug instruments provided directly by the vendor. Despite this, if AXI Performance Monitor is specifically conceived for monitoring purposes, the ILA is instead intended for logic level debug through signals waveform observation, thus its adoption for monitoring is only for comparison purposes. It has to be highlighted that these are not Xilinx specific issues, since considering different vendors (e.g., Intel), similar instruments with basically specular capabilities and limitations are available (e.g., Signal Tap Logic Analyzer [5]). The monitoring of the microprocessor, in this case capable of taking and storing timestamps, is compared to the ARM core built-in timer/counter which is an instrument always available (no additional resources are needed) on the hardwired microprocessor of the targeted SoC.

4.3.2 Quantitative Results. Table 6 illustrates the resource and power consumption numbers of the considered designs under test. Firstly, it is possible to see how for all the designs containing the proposed monitoring system for coprocessors ($D1_b$, $D2_b$ and $D3_b$), the resource overhead is limited to up to 9% for *LUT* and 10% for *FF*, while *BRAM* are not affected by monitoring. Comparing this overhead with the one provided by Xilinx state of the art monitoring solutions ($D1_a$, $D2_a$ and $D3_a$), we can appreciate the lightweight impact of our proposed monitoring system. Indeed, in this case the overhead goes from about 30% to more than 60% in terms of *LUT*, and from about 40% to 90% for *FF*. Moreover, *BRAM* are also increased in all the Xilinx monitoring solutions. Comparing designs with the same monitoring capabilities, the overhead increase, going from the proposed monitoring system to the Xilinx one, is always extremely clear, and it is maximum in the $D3$ designs where it goes from 9.11% to 62.54% for *LUT*, from 10.18% to 89.87% for *FF* and from 0% to 75% for *BRAM*. Overall, the resource occupancy data demonstrates that, for the given monitoring capabilities, the proposed monitoring system is more efficient than common state of the art solutions. When microprocessor monitoring is also enabled, resource overhead of the proposed solution ($D4_b$) is increased under all the considered metrics (more than 20% for both LUT and FF), including BRAM which are doubled with respect to $D0$. The additional logic required to monitor the ARM core indeed involves also a memory to store timestamps. Despite this considerable amount of additional resources, the proposed design is smaller than the corresponding state of the art solution ($D4_a$), but for the BRAM. Please consider that $D4_a$ resources instantiated in the programmable logic and the resulting power behavior are the same of $D3_a$, since the adopted state of the art solution for microprocessor monitoring is, as said, hardwired in the processing system and thus not accounted here. However, to store timestamps with ARM built-in timer/counter, users have to write custom software and adopt the available ARM on-chip memory, which is also not accounted here.

Table 6. Resources and power of the considered *Edge Detection* designs. In brackets is reported the percentage variation with respect to $D0$. *Power numbers come from measures where ILA is not configured.

Design	Resources			Dynamic Power [mW]		
	LUT (%)	*FF* (%)	*BRAM* (%)	*Sobel* (%)	*Roberts* (%)	*Transf.* (%)
$D0$	3807 (-)	4017 (-)	2 (-)	45 (-)	22 (-)	22 (-)
$D1_a$	4934 (+29.60)	5698 (+41.85)	3 (+50.00)	47 (+4.44)	25 (+13.64)	26 (+18.18)
$D1_b$	4081 (+7.20)	4316 (+7.44)	2 (+0.00)	44 (-2.22)	21 (-4.55)	22 (+0.00)
$D2_a$*	6017 (+58.05)	7413 (+84.54)	3.5 (+75.00)	55 (+22.22)	32 (+45.45)	33 (+50.00)
$D2_b$	4116 (+8.12)	4370 (+8.79)	2 (+0.00)	45 (+0.00)	22 (+0.00)	23 (+4.55)
$D3_a$*	6188 (+62.54)	7627 (+89.87)	3.5 (+75.00)	56 (+24.44)	33 (+50.00)	34 (+54.55)
$D3_b$	4154 (+9.11)	4426 (+10.18)	2 (+0.00)	46 (+2.22)	22 (+0.00)	23 (+4.55)
$D4_a$*	6188 (+62.54)	7627 (+89.87)	3.5 (+75.00)	56 (+24.44)	33 (+50.00)	34 (+54.55)
$D4_b$	4694 (+23.30)	5031 (+25.24)	4 (+100.00)	48 (+6.67)	25 (+13.64)	26 (+18.18)

Power numbers in Table 6 have the same trend of resource occupancy ones. The only design showing a different behaviour is $D1_b$ which seems to save power with respect to $D0$ during both applications execution, but not during data transfers. Such design is the one capable of monitoring the system bus, thus, as expected, its overhead is related more to the data transfer period. However, such overhead is quite low, and the resulting power consumed by the $D1_b$ while transferring data is the same of $D0$. Differently, the measured power saving related to execution is due to the extremely similar values which difference is close to the power estimation resolution (1 mW). Small synthesis and implementation differences in these designs may cause differences in switching activity (e.g., due to the presence of pending flipping signals that, being not isolated, can drive unused resources and waste power) and, in turn, have different rounding in power estimation. Apart from the saving observed in $D1_b$, the power data are as expected. The two coprocessor configurations differ in terms of power consumption (*Sobel* consumes about twice the power of *Roberts*).

Comparing the state of the art Xilinx coprocessor solutions ($D1_a$, $D2_a$ and $D3_a$) and the proposed ones ($D1_b$, $D2_b$ and $D3_b$), these latter are always more efficient than the former also in terms of power. While for $D1$, when only AXI Performance Monitor is enabled on the state of the art solution, the increment of overhead is limited (at maximum 18.18%), in the other cases, when also ILA is employed, the increment of overhead is stronger (always more than 22.22%, with a peak of 54.55%). Please note that in power numbers related to ILA ($D2_a$ and $D3_a$), the ILA IP has not been configured to actually measure the monitored signals/events (the ILA is free running) since it has not been possible to simulate it (the configuration of the ILA is performed through the JTAG interface that is not visible in post-implementation simulation). If microprocessor monitoring is enabled ($D4_b$), power consumption increases in all the considered configurations and periods (*Sobel*, *Roberts* and *Transf*), with a maximum overhead with respect to $D0$ of 18.18% in this last case. Nevertheless, the proposed solution is always better than the corresponding state of the art one, $D4_a$, in terms of power dissipation.

Table 7. Latency and memory footprint of the *Edge Detection* designs. *Instr.* is related to the whole executable file and *Data* is to the real disk occupancy of the data for processing and monitoring. *A fair measure of the considered metrics for such designs has not been possible. ** Only timestamp monitoring overhead is considered.

Design	Latency *Sobel*		Latency *Roberts*		Memory [B]			
					Instr.	*Instr.%*	*Data*	*Data%*
	Tot. [ms]	*Tot.%*	*Tot.* [ms]	*Tot.%*				
$D0$	502.208	–	482.666	–	1295112	–	2097152	–
	Mon. [us]	*Tot.%*	*Mon.* [us]	*Tot.%*				
$D1_a$	28.382	0.0059	28.338	0.0056	1365724	+5.45	2097156	+0.0002
$D1_b$	4.232	0.0009	4.153	0.0008	1302012	+0.53	2097155	+0.0001
$D2_a$*	–	–	–	–	–	–	–	–
$D2_b$	4.795	0.0010	4.783	0.0010	1302152	+0.54	2097161	+0.0004
$D3_a$*	–	–	–	–	–	–	–	–
$D3_b$	6.187	0.0012	6.17	0.0013	1302408	+0.56	2097168	+0.0008
$D4_a$*	–	–	–	–	–	–	–	–
$D4_b$	9.290	0.0018	9.27	0.0019	1309796	+1.14	2097168	+0.0008
$D4_{a256}$*	230.279**	0.0459	229.641**	0.0476	–	–	–	–
$D4_{b256}$	137.867**	0.0275	138.454**	0.0287	1309840	+1.14	2099216	0.0984

Table 7 reports execution measures of the considered designs on the target device. Dealing with a real *Edge Detection* application, which requires several runs of the coprocessor to complete,

the overall execution latency is now extremely bigger (hundreds of ms) than monitoring one (tens of us). It does not make anymore sense to compare total execution latency among different designs, since the differences are flattened by the statistical fluctuations. For this reason, the total execution time is given only for $D0$, that is the design without any monitoring capability. For the remaining designs only the latency due to the monitoring is reported, together with the percentage of this latter over the total $D0$ execution latency (*Tot* column). Please note that, for state of the art monitoring solutions ($D1_a$, $D2_a$, $D3_a$ and $D4_a$), it has been possible to provide fair numbers only in the $D1_a$ case, where the AXI Performance Monitor is adopted alone, and partially in the $D4_a$ one, as we will explain afterwards. In the other cases, where also ILA is adopted, a fair comparison under Table 7 related metrics has not been possible. Indeed, ILA is configured through the JTAG connection and uses dedicated *BRAM* modules, thus execution latency and instruction related memory footprint is not measurable. Looking at numbers, as expected, the monitoring latency overhead grows together with the monitoring capabilities. Differently from results reported in Section 4.2, the overhead on the total execution time is quite limited (always around 0.001%) demonstrating the little invasiveness of the proposed monitoring system, also in terms of latency, when applied to a real test case. The only fully measurable state of the art case, $D1_a$, is more time consuming than the corresponding proposed solution, $D1_b$ and than the other proposed solutions (with different monitoring capabilities). However, the resulting overhead is still negligible due to the size of the application, being not more than 0.006% of the total $D0$ execution latency. The adoption of microprocessor monitoring on the top of the coprocessor one, in form of timestamps, causes a certain overhead also in terms of latency. In particular, the monitoring configuration and finalization ($D4_b$) is still negligible with respect to the whole execution latency. Timestamps overhead is, instead application/user dependent, since they can be taken several times according to the specific purpose and needs. As an example, 256 timestamps have been taken during the execution of the application, one for each coprocessor call and just after microprocessor prepares input before such calls. Monitor overhead due to the timestamps alone has been measured and compared with the overall execution ($D0$) for the proposed ($D4_{b256}$) and for the state of the art ($D4_{a256}$) solution. Timestamps monitoring overhead is now heavier, but sill under 0.05% in all cases. Looking a bit more in detail, there is a clear difference between $D4_{a256}$ and $D4_{b256}$, with the proposed solution that is overperforming the state of the art one. In fact, $D4_{b256}$ takes the timestamp with a single write operation, while $D4_{b256}$ implements a more complex process, resulting in higher overhead (almost doubled) in terms of latency.

Table 7 also reports results related to memory footprint. In this case, a unique value for *Sobel* and *Roberts* is provided since such metrics do not change going from one detector to the other one. As for latency numbers, only $D1_a$ literature solution has been profiled since for the other state of the art monitoring designs, $D2_a$, $D3_a$ and $D4_a$, a fair comparison has not been possible. In terms of instruction memory footprint (*Instr* column), the proposed monitoring system requires a very low overhead that is always lower than 1.2%, meaning that the drivers necessary to configure and manage monitors are extremely lightweight. Here, as for the other considered metrics, monitoring the microprocessor ($D4_b$ and $D4_{b256}$) requires a bigger overhead (more or less doubled) than monitoring coprocessor alone with different levels ($D1_b$, $D2_b$ and $D3_b$). Considering the unique state of the art monitoring solution presenting memory footprint data, $D1_a$, it is clear that such overhead is bigger, being about 5% of the $D0$ program memory occupation. Focusing on data memory footprint such difference between state of the art and proposed monitoring system is no more present since the overhead is always negligible (less than 0.001%) in all the considered designs, due to the fact that the application data is huge. Anyway, fixing the monitoring capabilities to the transaction level monitoring, the proposed monitoring system ($D1_b$) is overperforming the state of the art ($D1_a$) by saving one byte.

4.3.3 Qualitative Results. A qualitative analysis of the benefits provided by the adoption of the proposed monitoring system is extremely important, especially due to the fact that the Xilinx ILA [45] does not allow a fair comparison for certain metrics evaluated in Section 4.3.2. Again, the aspects under which the qualitative analysis is going to be conducted are user effort and required skills. Dealing with the effort, as already discussed in Section 4.2.3 for the exploration, the proposed monitoring system requires almost no effort for the transaction level and task level monitoring while, at the moment, the operation level monitoring requires the user to manually bring out from the Data Cruncher of the coprocessor the signals to be monitored. The monitoring of the microprocessor, being not automated so far, needs a certain design effort and skills. In particular, the users have to insert monitors manually on the Vivado Block Design of the system, so that they have to know the architecture and the same monitors. This process will be automated in the future by means of the same script already available for most of coprocessor monitoring features, and all the updates will be uploaded in the provided open-source repository [2]. From the programming side, users have only to take timestamps wherever needed and to retrieve data afterwards. Such operations are possible through proper APIs which have the same shape of the coprocessor related ones, thus ensuring transparent and generic access to the monitors for the targeted heterogeneous platform.

At transaction level, the state of the art AXI Performance Monitor solution requires the users have to manually introduce and configure the IP in the Vivado project. This task can be easily automated, as for the proposed monitoring system, and its usage is facilitated by Xilinx drivers that require effort and skills similar to the ones required by the proposed monitoring system. The situation changes when considering task level and operation level monitoring, that at the state of the art are possible only with the ILA IP. Such IP has to be inserted in the netlist of the synthesized design. Users have to manually choose which signals have to be monitored in the netlist. This is similar to what is requested by the operation level monitoring in the proposed solution, with the huge difference that in this latter case the signals should be identified on the RTL code rather than in the netlist. Understanding and editing the netlist to find the right signals is not trivial since the design can change drastically after synthesis, and signals can be renamed and split, so that HW design skills and knowledge of the synthesis process are necessary to properly identify signals. For instance, the 8 bit signal *out_pel* going from the Data Cruncher to local memories inside the coprocessor IP in the $D3_b$ design has been renamed and split into the 1 bit signals *i_back_end_out_pel_n_0*, *i_back_end_out_pel_n_4*, *i_back_end_out_pel_n_5* and *i_back_end_out_pel_n_6*, while there is not trace of the remaining 4 bits of the signal in the netlist. Once signals are chosen, Vivado offers an automated wizard to insert the ILA IP and link it with the selected signals. Focusing on the considered monitor for microprocessor, the ARM built-in timer/counter, it is hardwired in the microprocessor chip, so that there is no need to insert it by hand. Moreover, it is extremely easy to be used, since Xilinx already provide drivers to properly access and exploit it, as occurs for the AXI Performance Monitor. Thus, also in this case, the effort required to the user for learning how to use it is similar to the one of the proposed solution.

In terms of monitor usage, it is important to highlight as the ILA has a completely different interface with respect to the AXI Performance monitor. Such interface, based on JTAG connection, makes it difficult to manage monitoring from the same SW running on the host core, as occurs for AXI Performance Monitor and for the proposed monitoring system. Moreover, the ILA is only able to store samples of data of the monitored signals for a certain time period and according to a predefined clock period. This means that, if users want to keep trace of the number of events or the duration of a certain signal, they have to post process data provided by ILA, while the proposed monitoring system delivers the final events or time count ready to be used. Please consider that, even if drivers are provided by Xilinx for both transaction coprocessor monitoring

level (AXI Performance Monitor) and microprocessor monitoring (ARM built-in timer/counter), they have multiple interfaces compared to our single interface. In summary, the considered literature solutions are not sufficient to provide monitoring for heterogeneous systems involving dedicated coprocessors, since they require a great design effort and specific digital HW design skills to the user. The proposed monitoring system, on the contrary, demonstrated to be a better choice under the considered effort and required skills aspects.

4.4 On the Exploitation of the Monitoring Infrastructure for Different Purposes

As discussed in Section 1, the specific monitoring purpose is not directly addressed by the proposed work, which is purpose-independent since focuses only on Event Instance Generation, Data Capture and Data Filtering steps of the *generic monitoring process* introduced by Kornaros et al. [19]. As the metrics are available, what to do with them depends on the purpose of the monitoring and the process from passive becomes active with the Decision Making and Reaction steps. This section discusses how the proposed HW monitoring system could be effective for different purposes. In particular, we discuss below how the proposed work is capable of extracting the same metrics of two works available in literature used for different purposes, namely monitor for debug and verification and monitor for security.

The work of Lee et al. [21] is an example of monitor for debug and verification, as also reported in Section 2.1: authors highlight the type of events to perform a debug, and propose observation interfaces to monitor those events at low-level with hardware circuitry. It is worth noting that their measure are part of two categories: either counting the number of events, or measure the time between two events. Our monitoring system can extract all the events proposed by authors: furthermore, the distinction between EIG and DCAPF allows to configure only two different DCAPFs, one with Event Monitor and one with Time Monitor, and to reuse them in different sniffers. Depending on the monitored interconnections, sniffers with different EIG blocks can be built.

The work of Arora et al. [8] is an example of monitor for security for microprocessors; in particular, authors propose a framework to build a HW monitoring system able to identify unintended behaviours by performing a check of the application (i) inter-procedural control flow, (ii) intra-procedural control flow, and (iii) instruction stream. Their monitoring system extracts the program counter to make a comparison with some expected values (for (i) and (ii)), and extracts the instruction register value to compute a hash function and compare, again, with expected values. Our monitoring system can perform the same extraction activity, with a proper configuration of filters inside DCAPF, making data available for the analysis and, additionally, is not limited to work with microprocessors, but can target also other system components.

Focusing on the proposed experimental activities, considering the different combinations of events monitored in the *Selective Accumulations* example (see Table 2 in Section 4.2), for instance the transaction level monitor can be used for:

- debug and verification - by playing with the DCAPF initialization values, it could be possible to identify any faulty transfers;
- security - to detect potentially malicious data transfers.

The task level monitor can be used, for example, for:

- performance - by measuring coprocessor computation time, something that is typically not available without a proper HW instrumentation of the coprocessor, useful to understand the impact of coprocessor data management and coprocessor computation time to better optimize the system components;

- debug and verification - to serve as watchdog, checking if the execution completes within a given deadline.

The operation level monitor can be used, for example, for:

- performance - by monitoring events giving hints on how data is flowing within the datapath;
- debug and verification - by detecting event occurrences not in line with expected ones;
- power - by linking events to coprocessor power models to estimate the related consumption.

Dealing with the examples provided in this paper, we can discuss how the different provided monitoring levels can be particularly useful in terms of prospective purposes. Considering the more realistic *Edge Detection* application, in which different combinations of events are monitored (see Table 5 in Section 4.3), we can provide different examples of monitoring for debug and verification related to fault detection and identification. The considered HW coprocessor system can be affected by different faults, ranging from data transfer (affecting the system bus) to computational ones (internal to the coprocessor). Moreover, also the microprocessor can be affected by faults especially considering the CPSs, which are connected, distributed and adaptive. In order to have detection and identification capabilities on such faults, the three envisioned monitoring levels can be effective in different ways:

- $D0$: this design does not provide any monitoring feature, so that it is not possible to know where are, which are and, sometimes, if there are faults;
- $D1$: this design, providing bus level monitoring, makes it possible to detect and to identify, thanks to the programmability of the adopted monitors, faults related to data transfers, e.g., user selects a wrong input data location;
- $D2$: the monitoring in this design enables additional features with respect to $D1$, being able to detect and identify faults on data transfer, but also to detect faults on the coprocessor computation, e.g., accelerator stall;
- $D3$: this design delivers the maximum coprocessor monitoring possibilities, making it possible not only to detect and identify data transfer faults and detect computation faults, but also to identify these latter, e.g., accelerator stall caused by wrong configuration for the given input image size, thanks to the view opened on the accelerator computing core signals of interest.
- $D4$: this design enables monitoring on the heterogeneous platform by adding on the top of the coprocessor monitors of $D3$ a monitor for the microprocessor, which delegates operations to the coprocessor and prepares data to be processed: if these latter are corrupted or faulty, this monitor can detect the fault and, depending on how and where timestamps are placed, identify it.

5 CONCLUSIONS AND FUTURE WORKS

Complex heterogeneous reconfigurable embedded computing platforms are currently widely adopted, being capable of providing performance together with flexibility. However, heterogeneity has raised several challenges for the designers: one of them is related to system monitoring which is often fundamental to ensure efficiency and adaptivity required to modern CPSs. Monitoring heterogeneous platforms means dealing with a variety of components ranging from general purpose cores, passing through interconnections, data moving and storing modules to custom coprocessing units. Providing a unified, homogeneously observable, composable, customizable and minimally invasive monitoring system is not trivial, and state of the art still lacks in providing an extensively adopted solution. In this work, we proposed a HW monitoring system limited to the passive part of the monitoring process, from low level signals to metrics. It has been extensively assessed, firstly, on a simple test case by considering all the possible system configurations and customization possibilities, and then on a real-world image processing application comparing the proposed solutions with

state of the art ones. Results show that our solution is flexible enough to offer to users a wide set of monitoring possibilities, at the price of a little overhead in terms of resources, power consumption, execution time and memory footprint. Moreover, thanks to the adopted tooling support, it requires less engineering effort and HW expertise.

In the next future, we plan to trace a line for a comprehensive HW monitoring system where all the components of a heterogeneous platform are monitored together in a unified, homogeneously observable, composable, customizable and little invasive way. Active parts of the monitoring process, taking as input metrics and performing decision making and reaction steps, will be also tackled in the evolution of the work. Coprocessors support will be also completed by adding tightly coupled accelerators, besides loosely coupled ones which are currently addressed. Moreover, design automation is going to be completed and refined. At the moment, designer effort and skills are still required concerning coprocessors operation level monitoring and processors monitoring, this latter to be added manually on Vivado Block Design. Dealing with the former, we envision to abstract low level details and allow users to choose signals of interest to be monitored directly on the application model. The latter, instead, will be embedded in the provided script already supporting most of coprocessor monitoring stuff.

ACKNOWLEDGMENTS

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 826610 (COMP4DRONES) and H2020-ECSEL-2017-2-783162 (FitOptiVis) [3]. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Austria, Belgium, Czech Republic, France, Italy, Latvia, Netherlands.

REFERENCES

[1] 2012-05. An FPGA "Companion" in Smartphone Design - A Lattice Semiconductor White Paper. Document ID 47335.
[2] 2020. Jointer Open-source repository. https://github.com/alkalir/jointer.git
[3] Zaid Al-Ars et al. 2019. The FitOptiVis ECSEL project: highly efficient distributed embedded image/video processing in cyber-physical systems. In *Conf. on Computing Frontiers*. 333–338.
[4] M. Aldham et al. 2011. Low-cost hardware profiling of run-time and energy in FPGA embedded processors. In *Conference on Application-specific Systems, Architectures and Processors*. 61–68.
[5] Altera. 2013-11. Design Debugging Using the SignalTap II Logic Analyzer. Quartus II Handbook v.13.1. Vol. 3: Verification.
[6] ARM. 2013-08. White Paper: CoreSight Technical Introduction, A quickstart for designers. ARM-EPM-039795.
[7] ARM. 2020. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. https://developer.arm.com/documentation/ihi0022/e/
[8] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. 2005. Secure embedded processing through hardware-assisted run-time monitoring. In *Design, Automation and Test in Europe*. 178–183 Vol. 1. https://doi.org/10.1109/DATE.2005.266
[9] Alexander Brant and Guy G. F. Lemieux. 2012. ZUMA: An Open FPGA Overlay Architecture. In *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 93–96.
[10] Andrew Canis et al. 2013. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Trans. on Embedded Computing Systems (TECS)* 13 (09 2013).
[11] Davide Zoni and and others. 2018. PowerTap: All-digital power meter modeling for run-time power monitoring. *Microprocessors and Microsystems* 63 (2018), 128 – 139.
[12] ER Davies. 1984. Circularity — a new principle underlying the design of accurate edge orientation operators. *Image and Vision Computing* 2, 3 (1984), 134–142.
[13] N. C. Doyle et al. 2017. Performance impacts and limitations of hardware memory access trace collection. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 506–511.
[14] Tiziana Fanni et al. 2018. Multi-Grain Reconfiguration for Advanced Adaptivity in Cyber-Physical Systems. In *Conference on ReConFigurable Computing and FPGAs*. IEEE, 1–8.
[15] T. Fanni et al. 2019. Run-time Performance Monitoring of Heterogenous Hw/Sw Platforms Using PAPI. In *Workshop on FPGAs for Software Programmers*. 1–10.

[16] J. Goeders and S. J. E. Wilton. 2017. Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 36, 1 (2017), 83–96.

[17] M. B. Hammouda et al. 2017. A Unified Design Flow to Automatically Generate On-Chip Monitors During High-Level Synthesis of Hardware Accelerators. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 36, 3 (2017), 384–397.

[18] Lizy Kurian John and Lieven Eeckhout. 2006. *Performance Evaluation and Benchmarking*. Taylor & Francis Group - CRC Press, Boca Raton.

[19] Georgios Kornaros and Dionisios Pnevmatikatos. 2013. A Survey and Taxonomy of On-Chip Monitoring of Multicore Systems-on-Chip. *ACM Trans. Des. Autom. Electron. Syst.* 18, 2, Article 17 (April 2013), 38 pages.

[20] Andreas Kurth et al. 2017. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. *CoRR* abs/1712.06497 (2017). arXiv:1712.06497

[21] Jong Chul Lee and Roman Lysecky. 2015. System-Level Observation Framework for Non-Intrusive Runtime Monitoring of Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.* 20, 3, Article 42 (June 2015), 27 pages.

[22] Xiangwei Li et al. 2018. FPGA Overlays: Hardware-based Computing for the Masses. In *Conference On Advances in Computing, Electronics and Electrical Technology*.

[23] Xiangwei Li and Douglas L. Maskell. 2019. Time-Multiplexed FPGA Overlay Architectures: A Survey. *ACM Trans. Design Autom. Electr. Syst.* 24, 5 (2019), 54:1–54:19.

[24] Daniel Madroñal and Tiziana Fanni. 2019. Run-Time Performance Monitoring of Hardware Accelerators: POSTER. In *Conference on Computing Frontiers*. 289–291.

[25] E. Matthews et al. 2010. A configurable framework for investigating workload execution. In *2010 International Conference on Field-Programmable Technology*. 409–412.

[26] Hyun min Kyung et al. 2010. Design and implementation of Performance Analysis Unit (PAU) for AXI-based multi-core System on Chip (SOC). *Microprocessors and Microsystems* 34, 2 (2010), 102 – 116.

[27] A. Moro et al. 2015. Hardware performance sniffers for embedded systems profiling. In *Workshop on Intelligent Solutions in Embedded Systems*. 29–34.

[28] V. Muttillo, G. Valente, L. Pomante, H. Posadas, J. Merino, and E. Villar. 2020. Run-time Monitoring and Trace Analysis Methodology for Component-based Embedded Systems Design Flow. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 117–125. https://doi.org/10.1109/DSD51259.2020.00029

[29] P. K. Nadimpalli and S. K. Roy. 2016. An efficient FPGA-based function profiler for embedded system applications. In *Symposium on VLSI Design and Test*. 1–6.

[30] M. Najem et al. 2017. A Design-Time Method for Building Cost-Effective Run-Time Power Monitoring. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 36, 7 (2017), 1153–1166.

[31] Geoffrey Nelissen et al. 2015. A Novel Run-Time Monitoring Architecture for Safe and Efficient Inline Monitoring. In *Reliable Software Technologies – Ada-Europe 2015*. Springer International Publishing, Cham, 66–82.

[32] Francesca Palumbo et al. 2019. Hardware/Software Self-adaptation in CPS: The CERBERO Project Approach. In *Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (2019-01-01). Cham, 416–428.

[33] PAPI. 2020. Performance API. http://icl.utk.edu/papi/

[34] G. Patrigeon et al. 2018. FPGA-Based Platform for Fast Accurate Evaluation of Ultra Low Power SoC. In *Symposium on Power and Timing Modeling, Optimization and Simulation*. 123–128.

[35] E. A. Rambo et al. 2019. The Information Processing Factory: A Paradigm for Life Cycle Management of Dependable Systems. In *Conference on Hardware/Software Codesign and System Synthesis*. 1–10.

[36] Alfonso Rodríguez et al. 2018. FPGA-Based High-Performance Embedded Systems for Adaptive Edge Computing in Cyber-Physical Systems: The ARTICo^3 Framework. *Sensors* 18, 6 (2018), 1877.

[37] Sadek, Ahmad and others. 2018. Supporting Utilities for Heterogeneous Embedded Image Processing Platforms (STHEM): An Overview. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Springer International Publishing, Cham, 737–749.

[38] C. Sau et al. 2016. Automated Design Flow for Multi-Functional Dataflow-Based Platforms. *J. Sign. Process. Syst.* 85, 1 (Oct. 2016), 143–165.

[39] T. Scheipel et al. 2017. System-Aware Performance Monitoring Unit for RISC-V Architectures. In *Conference on Digital System Design*. 86–93.

[40] Minjun Seo and Fadi Kurdahi. 2019. Efficient Tracing Methodology Using Automata Processor. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 80 (Oct. 2019), 18 pages.

[41] Minjun Seo and Roman Lysecky. [n.d.]. Non-Intrusive In-Situ Requirements Monitoring of Embedded System. *ACM Transactions on Design Automation of Electronic Systems* 23, 5 year = 2018, issn = 1084-4309, ([n. d.]).

[42] Minjun Seo and Roman Lysecky. 2018. Work-in-Progress: Runtime Requirements Monitoring for State-based Hardware. In *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[43] Anuj Vaishnav et al. 2018. A Survey on FPGA Virtualization. In *Conference on Field Programmable Logic and Applications*. IEEE Computer Society, 131–138.
[44] G. Valente et al. 2016. A Flexible Profiling Sub-System for Reconfigurable Logic Architectures. In *Conference on Parallel, Distributed, and Network-Based Processing*. 373–376.
[45] Xilinx. 2017-06-7. System Integrated Logic Analyzer v1.0, LogiCORE IP Product Guide, PG261.
[46] Xilinx. 2017-10-4. AXI Performance Monitor v5.0, LogiCORE IP Product Guide, PG037.
[47] Xilinx. 2020. Zynq7000 SoC Technical Reference Manual. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf