A Unified Hardware/Software Monitoring Method for Reconfigurable Computing Architectures using PAPI

Leonardo Suriano*, Daniel Madroñal[†], Alfonso Rodríguez*, Eduardo Juárez[†], César Sanz[†] and Eduardo de la Torre*

*Centro de Electrónica Industrial (CEI), Universidad Politécnica de Madrid

[†]Research Center on Software Technologies and Multimedia Systems (CITSEM), Universidad Politécnica de Madrid

Email: {leonardo.suriano, daniel.madronal, alfonso.rodriguez, eduardo.juarez, cesar.sanz, eduardo.delatorre}@upm.es

Abstract—In this work, a standard and unified method for monitoring hardware accelerators in Reconfigurable Computing Architectures is proposed, based on a standard software monitoring interface.

The open source Performance Application Programming Interface (PAPI) library is commonly used in the field of High Performance Computing and aims at providing event information directly extracted from a set of Performance Monitor Counters. Important events such as data and instruction cache misses, hardware interrupts, etc. are collected to analyze and profile applications to pinpoint the contingent bottlenecks. In other words, it serves as a "Hardware Abstraction layer" for applications running in the user-space.

In this paper, its use is extended by proposing a method to target custom Performance Hardware Registers on accelerators built upon an FPGA. Furthermore, portability and standardization are discussed and the overhead associated with PAPI use is evaluated. Two hardware examples are proposed to evaluate this approach: a simple counter and an infrastructure for hardware accelerators called ARTICo³.

Index Terms—Monitoring Hardware, Performance API, PAPI, PAPIFY, ARTICo³, Hardware Abstraction

I. INTRODUCTION AND MOTIVATION

Nowadays, the world is in the era of Internet of Things (IoT) and Cyber-Physical Systems (CPSs) [1]: everybody is connected to everything. Likewise, the new generation of computer systems should be (1) portable, (2) wearable, (3) offer the highest computing power using the lesser energy possible [2] and (4) include a specific set of sensors according to the specific application. Consequently, sensing, processing, communication and energy consumption are now considered key features of such devices [3].

For this reason, during the past years, ultra-low power micro-controllers have been massively used. Nevertheless, there is still a great research interest in this direction: in 2018, Intel has proposed a complete IoT system prototype [4] featuring an ultra-low power SoC in a 14nm tri-gate CMOS technology.

However, as demonstrated in [5], new architectures, using hardware-based solutions (i.e., FPGAs in this particular case), can (1) reduce the application execution time and (2) minimize the energy consumption of the system. Furthermore, including an FPGA in a Multi-Processor System on Chip (MPSoC)

Fig. 1. Computing architecture: comparison [6].

ensures a level of flexibility and adaptivity unreachable with a custom Application Specific Integrated Circuit (ASIC): surrounding the CPUs with programmable logic supports the modification of the CPS functionalities during its lifetime according to each specific situation.

For these reasons and taking into account the analysis of Reconfigurable System on Chip in [6] (2018), it is clear that, as shown in Fig 1, "Hybrid Reconfigurable SoCs" (hereafter called Heterogeneous MPSoCs) have a fair trade-off between performance and power consumption in one side versus flexibility in the other side. Hence, they will play a central role in the evolution of the CPS.

In this context, it is easy to explain the increasing interest on reconfigurable heterogeneous devices such as the Zynq-7000 family and the Zynq UltraScale+ developed and commercialized by Xilinx. Specifically, promising results have been obtained when exploiting their capability in different research areas, e.g., image compression [7], complex telecommunication systems [8], data acquisition applications and readout systems in the field of nuclear electronics [9]. Moreover, several design methodologies for multi-HW-accelerator-based systems were proposed to fully exploit their features [10].

Likewise, the main challenge when using this kind of *Reconfigurable Computer Architecture* is to give an easy access to the device hardware resources to users that are not familiar with the underlying concepts [11]. The tasks of **hardware-abstraction** and **resource-access-standardization** are usually carried out by an Operating System (OS). This issue has been addressed by different research groups and different solutions have been proposed: a *Run-Time System Manager* (RTMS) [12] by Technical University of Crete; SPREAD [13], a *Streaming-Based Partially Reconfigurable Architecture and Programming Model* proposed by Wang et al.; FUSE [14], a *Front-end USER framework* developed in Canada at the Simon Fraser University are some of the latest frameworks and OS extensions that target reconfigurable platforms.

Another challenge to cope with in a CPS environment is the **self-adaptivity**, i.e., the ability to change the system behavior and to *reconfigure* itself based on both a set of environment inputs and the system status [15]. Consequently, including **self-awareness** in a CPS allows the automatic selection of an optimal configuration in terms of internal system parameters such as energy-consumption or performance [16]. In this context, Rajkumar et al. in [17] defines CPSs as “physical and engineered systems whose operations are **monitored**, coordinated, controlled and integrated by a computing and communication core”.

The aforementioned frameworks and OS extensions introduce a **custom** solution to manage generic hardware. Thus, these systems lack of unified interface to access both SW and HW performance information when a heterogeneous architecture is employed, as in the case of the next generation of CPS.

In this line, a commonly used High Performance Computing (HPC) open-source library called Performance Application Programming Interface (PAPI) [18] provides unified method to access hardware performance information through a set of Performance Monitoring Counters (PMCs). Additionally, an extended version of this library called PAPIFY [19] simplifies the use of PAPI with an extra abstraction layer, which unifies the processing element (PE) monitoring configuration.

In this paper, a PAPI component to include FPGA monitoring for the available PEs (and also for an hardware architecture called ARTICo³) is developed and tested. With this solution, the existing OS-based Resource HW/SW Managers will have performance information available through a standard well-known library to monitor both software and hardware resources. To evaluate this new approach, two use cases (i.e. two hardware examples) are proposed and tested and monitoring overhead is evaluated.

The rest of the paper is structured as follows: an analysis of the role of PAPI is discussed in Section II. An overview of the tools, the device and the hardware components is given in Section III; in Section IV, the technical details of the custom-hardware/PAPI/PAPIFY integration are reported and, after a brief description of the examples implemented in Section V, the results are present and discussed in Section VI. Finally, in the conclusion, the main achievements and the motivation beyond the work are depicted.

II. BACKGROUND AND STATE OF THE ART

In this section, the role of PAPI is analyzed when targeting heterogeneous devices and platforms that include also an FPGA. Its use is investigated and the differences with the work here proposed are highlighted. Moreover, the importance of monitoring Hardware Performance Counters is discussed and another approach (similar to the one presented in this paper) is examined.

The Performance API (commonly called PAPI library) has provided, for more than a decade, low-level cross-platform access to hardware performance counters on the most modern CPUs and GPUs and its use was extended, also, to measure and report power and energy values. In fact, in [20], the author describes, in detail, the types of energy and power readings available thanks to the extended use of PAPI. The paper is a great example on how the use of a standard existing library can bring benefits when designing an application: in addition to the CPU performance counters, GPU counters and many other advanced PAPI features natively supported, new energy and power “Events” can be evaluated with no extra effort. In this respect the work proposed in this paper should be considered: through the use of the same API, new devices can be targeted including the use of generic hardware accelerators (not still taken into account from Weaver et al. in [20]).

Likewise, in [21], the use of PAPI to monitor energy and power is discussed. In that work, the effort of McCraw et al. is to extend the use of PAPI to support power monitoring capabilities for various platforms and, specifically, for the Intel Xeon Phi and Blue Gene/Q. Also, the integration of PAPI in PARSEC (a data-flow task-based runtime [22]) is discussed but, here, compatibility with hardware accelerators is not proved and, neither, discussed.

An example on how to monitor hardware registers on the programmable logic of an FPGA is present in [23]: *SnoopP* was introduced as a non-intrusive and real time profiling tool for **soft-core processors** and its use was proven with a MicroBlaze on a Xilinx Virtex II FPGA. Anyway, a compatibility with other **hard-processors** is here not shown. It means that, when using heterogeneous platform that includes, for instance, ARM processors, other profiling tools need to be used in addition to SnoopP.

Besides, the importance of Hardware Monitors is highlighted, also, in [24]. In this paper, the authors present the so called **Hardware Performance Monitoring Infrastructure** (HwPMI): a set of hardware monitors (hardware cores) to be inserted in a general HDL design and, also, a set of software tools to manage them. The purpose is to *profile a hardware design*. The use of custom hardware counters for monitoring other kind of *Events* (different from the ones they already have included in the **HwPMI**) is not described. Also, the set of functions to retrieve data from the hardware needs to be integrated in the particular developed application.

A similar approach to the one proposed in this paper was presented in 2017 by Wagner et al. in [25]: the effort of the authors is in extending the use of **OpenMP Tools**

Interface (known as **OMPT**) for supporting and targeting accelerators. OMPT was first introduced in [26] as “an application programming interface (API) to support construction of portable, efficient, and vendor-neutral performance tools” and it “enables performance tools to gather useful performance information from applications with low overhead”. With the extension, the authors target also FPGAs and not only homogeneous/heterogeneous platforms, allowing insightful analysis by retrieving detailed performance information about the execution of the accelerated tasks. The use of the library is restricted to applications OpenMP-compliant.

Differently, in this paper, the method proposed uses PAPI, which is an abstraction layer commonly employed to interface hardware performance counters of CPUs, GPUs, network or memory controllers [27], but also of power and temperature monitors, as it is previously discussed in this section. Its use is not restricted to any particular application and it is employed by many profiling toolkits that will be discussed in Section III.

III. MATERIAL AND METHODS

In this section the technologies employed are described: (1) the target platform, (2) the PAPI library and its PAPIFY extension and, (3) the ARTiCo³ hardware architecture.

A. The device: UltraScale+ / Heterogeneous platforms

In order to demonstrate the method proposed, an UltraScale+ chip developed by Xilinx is selected as the target platform. In Fig 2 the main components are highlighted: this device is equipped with a quad-core ARM[®]Cortex-A53, with a dual-core Cortex-R5 real-time processor, a Mali-400 MP2 Graphics Processing Unit (GPU) and, also, with a 16nm FinFET+ programmable logic. The Programmable Logic (PL) hosts a set of System Logic Cells, Block Memory, UltraRAM, DSP and many other resources and here, thanks to the use of Vivado, we can build HW accelerators for boosting SW applications. For a detailed description of the device and for a complete list of SW and HW documentation, please, refer to [28].

Additionally, for the purpose of the paper, an OS GNU/Linux-based is developed to run on the quad-core and to manage the transaction between the user-space and the hardware. Also, the examples proposed were tested on the Pynq Board featuring a Zynq XC7Z020 chip.

B. The open source library: PAPI library and PAPIFY extension

The Performance Application Programming Interface (PAPI) library focuses on providing a standard API to easily access hardware monitoring information. Although this library can be used as a standalone tool for system analysis, it is also usually employed as a middleware in profiling, tracing and sampling toolkits like Vampir [29], HPCToolKit [30] and Score-P [31].

This API is divided into two layers: first, a platform-independent layer that provides an unified hardware monitoring interface; secondly, a lower, platform-dependent layer to

Fig. 2. A simplified block diagram of the UltraScale+ MPSoC.

deal with the specific characteristics of the platform, which is transparent for the user. Additionally, in [19], a new abstraction layer called PAPIFY is added on top of the other ones. This extra layer focuses on easing the configuration process and the usage of PAPI library when several PAPI components, which are usually associated to different types of PEs, are defined in the same architecture.

In order to use PAPIFY, a set of functions has been included within the *eventLib* library. In this context, the performance information can be accessed at runtime by including three stages in the application code: (1) a configuration step where *configure_papify()* transparently initializes both the PAPI component and the events to be monitored and associates them to the specific PE; (2) *event_start()* and *event_start_papify_timing()* trigger the monitoring of both events and timing, respectively; (3) *event_stop()* and *event_stop_papify_timing()* finish the monitoring and store the results.

C. The hardware architecture: ARTiCo³

In one of the examples proposed, the ARTiCo³ architecture [32] is used. ARTiCo³ permits to manage and set, dynamically, reconfigurable hardware accelerators making use of the Dynamic and Partial Reconfiguration (DPR). Its use has been already demonstrated in the Cyber-Physical System field where a fair trade-off among performance, dependability and energy consumption is achieved thanks to a runtime adaptable bus architecture [33]. In this paper we are not focusing the attention on the benefits/disadvantages that this hardware structure brings and (1) further details on the implementations, (2) software APIs, (3) applications developed and (4) performance/power-consumption measurements are out of the scope of this work and are remanded to the bibliography [32] [33]. However, in Fig 3, a stylized view of the whole system is given, where it is possible to identify the Processing System of the UltraScale+ (with the inclusion of two main physical cores) and how it is connected to the Programmable Logic (PL). Furthermore, the ARTiCo³ infrastructure and the hardware accelerators are highlighted to show that the monitors to be interfaced with PAPI are located inside the shuffler and they collect the information concerning every dynamic slot.

Fig. 3. Stylized view of the whole system with the ARTICo³ hardware architecture.

IV. INTERFACING HW WITH PAPI AND PAPIFY

In this section, the proposed method is discussed and explained step by step using a naive example. On the other hand, a brief description of the hardware used is necessary to fully understand the procedure.

A. Targeting Physical Addresses from the User-space

A simple 32-bits counter was created in VHDL and nestled in an AXI4-lite compliant wrapper (it permits to communicate with the standard bus protocol of the quad-core in the PS). Fig 4 shows that the hardware component includes four slave registers where, in order to start/stop the counting and retrieve the data, AXI4-lite transactions are required. In red, the useful bits of the slave registers are highlighted while the white ones are unused (they exist for a full compatibility of the AXI4-lite protocol).

Fig. 4. Schematic view of the 32 bit counter interface with PAPI.

In order to quantify the overhead due to the use of PAPI and PAPIFY, a register access time test is required at this point, where no additionally libraries are involved.

Differently from the approach in [14] where Linux Device Drivers were developed every time new accelerators/HW-

registers were created, Fig 5 depicts a block diagram explaining the procedure to access Physical Addresses from the User-space. Specifically, the steps to manage HW components are: (1) direct mapping of user-space virtual addresses to HW accelerators physical addresses using `mmap()` [34]; (2) command/data writing into HW accelerators using virtual addresses obtained with `mmap()`.

Fig. 5. The use of `mmap()` for managing physical addresses.

B. Component in PAPI

PAPI provides a long list of Components [35] that gives the possibility of reading PMCs of the most important architectures: ARM CPUs, Intel CPUs, AMD CPUs but also on NVIDIA GPUs, InfiniBand, etc.

However, when new hardware accelerators are designed, the necessity of developing a new component arises [27]. Likewise, the essential functionalities of the basic FPGA hardware together with their corresponding PAPI functions are the following ones:

```
PAPI_start(Ev) // writes the bit 0 of slv_reg0
PAPI_stop(Ev) // writes the bit 0 of slv_reg0
PAPI_reset(Ev) // writes the bit 0 of slv_reg1
PAPI_read(Ev) // reads the whole slv_reg3
```

where **Ev** argument of the functions stands for **Event**.

C. Connecting HW with PAPI and PAPIFY

Profiling an application using PAPI can be a tough task and, as explained in IV, PAPIFY aims at easing this process. Specifically, the whole PAPI configuration setup is now transparent and the setting/monitoring/collecting-data can be summarized in three instructions. Fig 6 depicts the schema of the approach.

In short, the steps to follow are:

- 1) to create hardware counters (PMCs) on the programmable logic. Every counter will be in charge of counting "Events". The designer is, so, completely free to monitor signal and transaction useful in his own design;
- 2) to give access from the user-space to physical addresses corresponding to the created PMCs;

Fig. 6. PAPIFY usage.

- 3) to create a PAPI component corresponding to the PMCs. The PAPI library will manage internally and autonomously the access to the hardware resources.

The use of PAPIFY is optional and aims at easing PAPI usage. By including this new abstraction layer, the configuration step is simplified from tens of function calls to only one (*configure_papify()*), independently of the number of events.

To sum up, in the Fig 7, a schematic overview of the entire system with the hardware-software connection is given. The *Adaptation Engine* is in charge of reconfigure the hardware if and when necessary on the *Adaptation Fabric* (that is a Zynq Ultrascale+ in our example). The Linux-Based OS running in the device is in charge of retrieving data by using PAPI and PAPIFY. The measurements are, so, available to other profiling SW in charge to model and evaluate the behavior of the system.

V. EXAMPLES IMPLEMENTED

In order to quantify the overhead due to the use of extra abstraction layers, test-applications were created: several **read** operations were sequentially performed using three methods:

- 1) a pointer to virtual memory to directly access hardware registers thanks to the use of *mmap()* described in Section IV;

```

//...
gettimeofday(&t0, NULL);
for(i=0;i<N_readings;i++){
    output_vector[i] = *ptr_to_hw;
}
gettimeofday(&tf, NULL);
//...

```

- 2) *PAPI_read(EventSet)*: the standard PAPI function created for the designed hardware;

```

//...
gettimeofday(&t0, NULL);
for(i=0;i<N_readings;i++){
    PAPI_read(EventSet, values);
}

```

```

}
gettimeofday(&tf, NULL);
//...

```

- 3) *event_read(papify_actions_Read)*: the PAPIFY function;

```

//...
gettimeofday(&t0, NULL);
for(i=0;i<N_readings;i++){
    event_read(papify_actions_Read, 0);
}
gettimeofday(&tf, NULL);
//...

```

In the case of ARTICo³ architecture, two PMCs were designed in VHDL:

- 1) error counters;
- 2) latency of execution (time window between the **start** signal and the end of the hardware execution).

Also, the ARTICo³ architecture is accompanied by a Run-time support: a set of software functions to manage the underlying hardware. Two new function has been included to have access to the PMCs:

- 1) *hw_get_pmc_errors(SLOT)*;
- 2) *hw_get_pmc_cycles(SLOT)*;

where *SLOT* is the number of the specific hardware accelerator. Similarly, three methods were used to evaluate the overhead (here the snippet code only for one of the two registers is reported: the method is equivalent in the two cases):

- 1) the lowest functions of the Run-time to have access to the hardware registers;

```

//...
gettimeofday(&t0, NULL);
for(i=0;i<N_readings;i++){
    output_value = hw_get_pmc_cycles(0);
}
gettimeofday(&tf, NULL);
//...

```

- 2) *PAPI_read(EventSet)*: the PAPI function created for the designed hardware; here the code snippet is exactly the same (the benefits of the abstraction and *code reuse* are clear). The main difference with the previous example is in the PAPI configuration code.

- 3) *event_read(papify_actions_Read)*: the equivalent PAPIFY instruction; again the code snippet is missing due to the exact correspondence with the previous one. In this case, the configuration code is missing and hidden by PAPIFY. The advantage of the *Abstraction* is, again, highlighted.

VI. RESULTS

In this section, the values measured using the code reported in the section V are analyzed and discussed.

Fig. 7. Schematic of the connections between the PMCs and the PAPI component

A. Results report

In table I, the elapsed times with different number of repetition of the `for` loop are shown. In this way it is possible to prove the accuracy of the measurement: in fact, it was noted that, with less than one hundred repetitions, the `read` transactions are so fast that the precision of the function `gettimeofday()` is not enough.

TABLE I
32-BIT COUNTER ACCESSING USING DIFFERENT FUNCTIONS

Measurements			
HW	Function	Reps	Time (ms)
Counter 32 bits	low level	1000	0.211
		10000	2.101
		100000	21.100
	PAPI	1000	0.361
		10000	3.644
		100000	36.154
	PAPIFY	1000	0.3710
		10000	3.7260
		100000	37.647

The overheads are computed dividing the average of the elapsed time per access in table I by the low level access time and are reported in the following table:

TABLE II
OVERHEAD ESTIMATION FOR COUNTER 32 BITS

Function	Access Time (ns)	Overhead
HW_pointer	221	x 1
PAPI_read(EventSet)	361	x 1,71
event_read(papify_actions)	371	x 1,77

The same procedure has been repeated for the ARTICo³ architecture and the table III and table IV summarize the results:

TABLE III
TIME NECESSARY FOR READING VALUES FROM ARTICo³ USING DIFFERENT FUNCTIONS

Measurements			
HW	Function	Reps	Time (ms)
ARTICo ³	low level	1000	0.242
		10000	2.428
		100000	24.244
	PAPI	1000	0.495
		10000	4.130
		100000	41.086
	PAPIFY	1000	0.475
		10000	4.247
		100000	42.104

TABLE IV
OVERHEAD ESTIMATION FOR THE ARTICo³ ARCHITECTURE

Function	Access Time (ns)	Overhead
hw_get_pmc_cycles(SLOT);	242	x 1
PAPI_read(EventSet)	413	x 1,70
event_read(papify_actions)	424	x 1,75

B. Results discussion

As expected, the overhead in the accessing times due to the use of PAPI and PAPIFY to read the registers is shown. This time deceleration is the price for the portability of the application and the monitoring. Likewise, using PAPI and

PAPIFY guarantees potential compatibility with SW profiling tools.

Moreover, it is important to note the difference between TABLE I and TABLE III where the absolute values of timing access are reported: using the ARTICO³ runtime function for retrieving data from the registers adds a little overhead more. The reason lies in the hardware pointer-calls: they are not performed directly by the PAPI-Component but it *delegates* the task to the ARTICO³'s functions.

VII. CONCLUSIONS

In this paper, a unified and standard strategy to interface Hardware Performance Registers is proposed. The Performance API is extended to Reconfigurable Computing Architectures targeting, also, hardware accelerators. Using a standard open-source library (to monitor what is happening in the Hardware Accelerators located in the FPGA) ensures a potential compatibility with a myriad of SW profiling tools such as Vampir [29], HPCToolKit [30] and Score-P [31]. Also, as discussed in section I, many research groups have proposed frameworks and OS extensions to manage SW and HW PEs in heterogeneous platforms. The aim of the work proposed in this paper is to give a common profiling instrument for these Heterogeneous MPSoCs. This further *Abstraction Layer* is also important in self-aware CPSs where a runtime manager needs to reconfigure the system based on the platform monitoring information (as well as all the inputs from the external physical world).

Specifically, a detailed description on the steps to include the HW monitoring as a PAPI component is given by using a naive VHDL module. Additionally, the method is tested using the ARTICO³ hardware architecture running on the Zynq Ultrascale+ MPSoC device developed by Xilinx.

As a result, it is shown that the overhead associated to the use of PAPI needs to be taken into account. Finally, the additional overhead of PAPIFY is negligible, so its use is encouraged to ease the utilization of PAPI in terms of code simplicity and intuitiveness.

ACKNOWLEDGMENT

This work has received funding from the EU Commissions H2020 Program under grant agreement No 732105.

Leonardo Suriano and Daniel Madroñal hold predoctoral contracts under RR01/2015 (Programa Propio) by Universidad Politécnica de Madrid.

The authors would like to thank the Spanish Ministry of Education, Culture and Sport for its support under the FPU grant program.

REFERENCES

- [1] F. Garibaldi and E. Rebecchi, "Cyber-physical system," 2018.
- [2] S. Vinco, L. Bottaccioli, E. Patti, A. Acquaviva, and M. Poncino, "A compact pv panel model for cyber-physical systems in smart cities," 2018.
- [3] M. Alioto, E. Sánchez-Sinencio, and A. Sangiovanni-Vincentelli, "Guest editorial special issue on circuits and systems for the internet of things—from sensing to sensemaking," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2221–2225, 2017.
- [4] T. Karnik, D. Kurian, P. Aseron, R. Dorrance, E. Alpman, A. Nicoara, R. Popov, L. Azarenkov, M. Moiseev, L. Zhao *et al.*, "A cm-scale self-powered intelligent and secure iot edge mote featuring an ultra-low-power soc in 14nm tri-gate cmos," in *Solid-State Circuits Conference (ISSCC), 2018 IEEE International*. IEEE, 2018, pp. 46–48.
- [5] J. Valverde, A. Otero, M. Lopez, J. Portilla, E. De La Torre, and T. Riesgo, "Using sram based fpgas for power-aware high performance wireless sensor networks," *Sensors*, vol. 12, no. 3, pp. 2667–2692, 2012.
- [6] H. K. Nguyen and T. X. Tran, "A survey on reconfigurable system-on-chips," *REV Journal on Electronics and Communications*, 2018.
- [7] P. J. Edavoor, S. Raveendran, and A. D. Rahulkar, "Implementation of adaptive image compression algorithm using varying bit-length daubechies wavelet coefficient with three-level encryption on zynq 7000," in *Embedded Computing and System Design (ISED), 2017 7th International Symposium on*. IEEE, 2017, pp. 1–6.
- [8] E. Setiawan, M. M. Latin, V. A. Mardiana, and T. Adiono, "Implementation of baseband transmitter design based on qpsk modulation on zynq-7000 all-programmable system-on-chip," in *Electronics and Smart Devices (ISED), 2017 International Symposium on*. IEEE, 2017, pp. 138–143.
- [9] T. Xue, J. Zhu, G. Gong, L. Wei, Y. Luo, and J. Li, "The design and data-throughput performance of readout module based on zynq soc," *IEEE Transactions on Nuclear Science*, vol. PP, no. 99, pp. 1–1, 2018.
- [10] L. Suriano, A. Rodriguez, K. Desnos, M. Pelcat, and E. de la Torre, "Analysis of a heterogeneous multi-core, multi-hw-accelerator-based system designed using preesm and sdsoc," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2017 12th International Symposium on*. IEEE, 2017, pp. 1–7.
- [11] M. Eckert, D. Meyer, J. Haase, and B. Klauer, "Operating system concepts for reconfigurable computing: review and survey," *International Journal of Reconfigurable Computing*, vol. 2016, 2016.
- [12] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, "Hardware task scheduling for partially reconfigurable fpgas," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 487–498.
- [13] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "Spread: A streaming-based partially reconfigurable architecture and programming model," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2179–2192, 2013.
- [14] A. Ismail and L. Shannon, "Fuse: Front-end user framework for o/s abstraction of hardware accelerators," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 170–177.
- [15] J. Otto, B. Vogel-Heuser, and O. Niggemann, "Automatic parameter estimation for reusable software components of modular and reconfigurable cyber-physical production systems in the domain of discrete manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 275–282, 2018.
- [16] J. Preden, "Generating situation awareness in cyber-physical systems: Creation and exchange of situational information," in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2014, pp. 1–3.
- [17] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Proceedings of the 47th design automation conference*. ACM, 2010, pp. 731–736.
- [18] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.
- [19] D. Madroñal, A. Morvan, R. Lazcano, R. Salvador, K. Desnos, E. Juarez, and C. Sanz, "Automatic instrumentation of dataflow applications using papi," in *CF '18: Computing Frontiers Conference*. ACM, 2018.
- [20] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 262–268.
- [21] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, "Power monitoring with papi for extreme scale architectures and dataflow-based programming models," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*. IEEE, 2014, pp. 385–391.
- [22] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: a data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2017, p. 6.

- [23] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/software codesign," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. ACM, 2004, pp. 190–199.
- [24] A. G. Schmidt, N. Steiner, M. French, and R. Sass, "Hwpmi: an extensible performance monitoring infrastructure for improving hardware design and productivity on fpgas," *International Journal of Reconfigurable Computing*, vol. 2012, p. 2, 2012.
- [25] M. Wagner, G. Llort, A. Filgueras, D. Jiménez-González, H. Servat, X. Teruel, E. Mercadal, C. Álvarez, J. Giménez, X. Martorell *et al.*, "Monitoring heterogeneous applications with the openmp tools interface," in *Tools for High Performance Computing 2016*. Springer, 2017, pp. 41–57.
- [26] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copt, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *International Workshop on OpenMP*. Springer, 2013, pp. 171–185.
- [27] "Papi-c component developer's manual," http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Component_Developers_Manual, 2018, [Online].
- [28] Xilinx, "Zynq ultrascale+ device technical reference manual, ug1085," Dec 2017.
- [29] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," *Tools for High Performance Computing*, pp. 139–155, 2008.
- [30] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
- [31] M. Schlütter, B. Mohr, L. Morin, P. Philippen, and M. Geimer, "Profiling hybrid hmpp applications with score-p on heterogeneous hardware," in *International Conference on Parallel Computing*, no. FZJ-2014-01861. Jülich Supercomputing Center, 2014.
- [32] A. Rodriguez, J. Valverde, and E. de la Torre, "Design of opencl-compatible multithreaded hardware accelerators with dynamic support for embedded fpgas," in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*. IEEE, 2015, pp. 1–7.
- [33] J. Valverde, A. Rodriguez, J. Camarero, A. Otero, J. Portilla, E. de la Torre, and T. Riesgo, "A dynamically adaptable bus architecture for trading-off among performance, consumption and dependability in cyber-physical systems," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 2014, pp. 1–4.
- [34] "Linux programmer's manual," <http://man7.org/linux/man-pages/man2/mmap.2.html>, 2018, [Online].
- [35] "Papi components list," http://icl.cs.utk.edu/papi/docs/d3/d76/group__papi__components.html, 2018, [Online].