Automatic Instrumentation of Dataflow Applications using PAPI

D. Madroñal[1], A. Morvan[2], R. Lazcano[1], R. Salvador[1], K. Desnos[2], E. Juárez[1], C. Sanz[1]
[1]Universidad Politecnica de Madrid - Research Center on Software Technologies and Multimedia Systems
{daniel.madronal,raquel.lazcano,ruben.salvador,eduardo.juarez,cesar.sanz}@upm.es
[2]Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes, France
{anmorvan,kdesnos}@insa-rennes.fr

ABSTRACT

The widening of the complexity-productivity gap witnessed in the last years is becoming unaffordable from the application development point of view. New design methods try to automate most designers tasks in order to bridge this gap. In addition, new Models of Computation (MoC), as those dataflow-based, ease the expression of parallelism within applications and lead to higher productivity.

Rapid prototyping design tools offer fast estimations of the soundness of design choices. A key step when prototyping an application is to have representative performance indicators to estimate the validity of the design choices. Such indicators can be obtained using hardware information through the Performance API (PAPI).

In this work, PAPI and a dataflow MoC are integrated within a Y-chart design flow. The implementation takes the form of a dedicated automatic code generation scheme within the Preesm tool. Preliminary results show that depending on the complexity of the application, the computation time overhead due to monitoring varies from being almost negligible to more than 50%. Also, on top of offering accurate hardware performance indicators, the extracted values can be combined to estimate power or energy consumption.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; **Data flow languages**; • **Hardware** → *Platform power issues*;

KEYWORDS

Dataflow Model of Computation, Code Instrumentation, Performance Monitoring Counter, Performance API, Automatic Code Generation

1 INTRODUCTION

During the last few decades there has been an ever increasing widening of the gap between platform complexity and application productivity, as shown for instance in the case of Cyber-Physical Systems [6]. On the one hand, modern architectures constantly grow in heterogeneity and number of Processing Elements (PE). Secondly, autonomous and multimedia applications also have a constant increase in both algorithmic complexity and computational power requirements while demanding low energy consumption.

In this context, evaluating which is the best architecture for a specific application becomes a challenging task. This is due to the peculiarities associated to each platform, e.g., efficient workload distribution, parallelization strategies, bottlenecks, memory accesses, etc. Dealing with these aspects with today's methods is reaching the frontiers of what can currently be achieved. Indeed, it requires a lot of time for experienced developers to achieve an implementation that meets the required non-functional criteria. Regarding this situation, the Y-chart design strategy [4] is gaining momentum among the paradigms to bridge this productivity gap. Specifically, this strategy is based on separating application and architecture concerns and merge them under a set of developer-defined constraints.

To carry out this Y-chart design methodology, dataflow Models of Computation (MoC), Models of Architecture (MoA), programming methodologies and associated design tools have appeared to explore new, improved design flows. In this way, design space exploration by rapid prototyping of applications on these complex architectures is being tackled through a set of different design automation tools like compiler parallelization techniques, code generation, deadlock analysis, task scheduling, etc. However, tools based on Y-chart design, such as ORCC [14] or Preesm [8], usually provide a generic solution following a predefined methodology for any application.

In order to evaluate the quality of automatic deployments, it is necessary to analyze the execution of the application on the target platform. Doing so requires having a deep understanding of the specific characteristics of the architecture. Thankfully, an abstraction layer exists that exposes an uniform interface to access hardware Performance Monitoring Counters (PMC). The Performance Application Programming Interface (PAPI) offers a common architecture-independent layer coupled with an architecture-specific layer to cope with the individual characteristics of each architecture. Using PAPI, the PMCs can be accessed to profile information such as memory usage, code parallelization, workload distribution, I/O utilization, etc. Additionally, some other parameters can be estimated combining this information, such as power or energy [9, 10]. Having these performance indicators would contribute not only to designers' productivity but also to achieve an iterative design flow. Likewise, Spider [3], a runtime manager based on Preesm, could benefit from this information to expand its reconfiguration criteria.

In this paper, hardware instrumentation using PAPI is integrated within the Y-chart design strategy supported by Preesm. Specifically, a new Preesm code generator is presented, where C code is instrumented automatically for x86 architectures. This work is a starting point to achieve an architecture-independent monitoring tool for dataflow applications developed with the Preesm tool-chain. This new code generator focuses on helping developers to analyze the performance of their dataflow applications in both compile and execution time. The aim to do so is to increase their productivity and efficiency. Specifically, this paper presents a prototype for characterizing dataflow applications and studies the restrictions that should be evaluated to support real-time PMC monitoring.

The rest of the paper is organized as follows: both PAPI and Preesm are detailed in Section 2 together with their integration.

After that, the results obtained during the assessment of the Papify-Preesm code generator are presented in Section 3 and the main conclusions and future work discussed in Section 4.

2 MATERIALS AND METHODS

In this section, the tools used to automatically generate instrumented C code for PAPI using Preesm are presented. Specifically, this section is divided in three main parts: (i) description of PAPI; (ii) Preesm main aspects; (iii) integration of both tools as a new code generator called *Papify-Preesm*.

2.1 *Performance API* (PAPI)

PAPI aims at providing a standard API focused on easing the access to hardware monitoring information [13] through a set of PMCs. Even though PAPI can be used as a standalone tool for system and application analysis, it has been widely employed as a middleware component in profiling, tracing and sampling toolkits such as HPCToolkit [1], Vampir [5] and Score-P [11].

With the arrival and expansion of multi-core processors and heterogeneous platforms, PAPI has been divided into two layers: on the one hand, an upper layer, which is platform-independent, providing a standard hardware monitoring interface; on the other hand, a lower, platform-dependent layer, which is transparent for the user, configured at compile time to automatically deal with the specific characteristics of each architecture [13].

Additionally, PAPI can be linked with several independent components at compile time, each of them representing a different resource within the same platform (e.g. a GPU and an x86 processor). Likewise, even when a heterogeneous platform is taken into account, the different hardware resources can be accessed through the same interface. Consequently, the PMC information can be obtained from a set of software and hardware resources such as CPUs, GPUs, memory or user defined components [2].

2.2 PREESM

Preesm is an open-source rapid prototyping tool [8] that works with three inputs: a dataflow graph defining the application; a System-Level Architecture Model (S-LAM) describing the target architecture; and a scenario including a set of parameters and constraints to link both of them. To deploy the algorithm over the target architecture, Preesm maps and schedules automatically the dataflow specification over the available PEs, e.g, over the available CPU cores in a multicore environment as the one used in this work.

Applications in Preesm are specified using the Parameterized and Interfaced Synchronous Dataflow (PiSDF) [8] MoC, an extension of the Synchronous Dataflow (SDF) [7], where computations are represented by nodes, called *actors*, and communications occur through *First In, First Out* data queues (FIFOs). PiSDF extends SDF by introducing consistent graph hierarchy using interfaces, parameterized FIFO sizes and runtime reconfiguration [3].

Likewise, S-LAM [8] describes parallel architectures as a set of PEs transmitting data through a set of communication nodes and data links. By doing so, it supports the definition of SW, HW or heterogeneous platforms [12] connected through different levels of granularity (i.e. Ethernet, shared memory, etc).

The join point of the Y-chart design flow is the scenario. It relates both the application (PiSDF) and the architecture (S-LAM). Additionally, it provides user defined information to drive the automatic steps of the flow, e.g., actor timing information or actor ↔ PE affinity. Using this information, Preesm schedules, maps and simulates the execution of the application and generates a compilable code in a language supported by the architecture, thus providing both metrics for system design and a prototype for testing, respectively.

It should be noted that, to the best of the authors knowledge, not only the decoupling between application and architecture design, but also its static nature and deadlock-free execution makes Preesm a suitable method for an architecture-independent strategy compared to others like, for example, ORCC [14].

2.3 Papify-PREESM

In order to integrate PMC monitoring based on PAPI library into Preesm code generation, Papify-Preesm has been developed. This code generator aims at instrumenting C code automatically, including both timing and event monitoring using PAPI capabilities, hereafter called *papification*. Papify-Preesm includes function calls for monitoring each actor individually. These extra functions have been developed and included in a new library called *eventLib*, adding a new level of abstraction to PAPI. As a result, the user is able to decide whether to characterize each actor or not in design time. During the execution, a *csv* file is generated for each instrumented actor storing both timing and PMC values.

The procedure to instrument each actor during the code generation (CodeGen) is shown in CodeGen Template 1. If the actor is being *papified*, a set of extra function calls are included during code generation. In this case, *actorData* will store the individual monitoring information related to the *papification* of each actor. The specific information is the following: (1) the *listOfEvents* being monitored during the execution; (2) the PE executing the actor (x86 processor, GPU, etc.), which isolates the actor monitoring to a specific core; (3) the values obtained for each PAPI event being monitored; (4) the timings, i.e., start and stop times.

```
for every actor within the dataflow specification do
    if actor_papified then
        configure_papification(actorData, listOfEvents, PE);
        event_start(actorData);
        event_start_PAPI_timing(actorData);
        actor_execution();
        event_stop_PAPI_timing(actorData);
        event_stop(actorData);
        event_write_file(actorData);
    else
        actor_execution();
    end
end
```

CodeGen Template 1: Papify-Preesm Code Instrumentation

As mentioned before, all the functions required to monitor actor execution have been developed together with the aforementioned *eventLib* library. The resulting extra abstraction layer aims at unifying the procedure of monitoring each actor independently, hence, to configure the instrumentation in terms of the hardware resource that is executing the actor (the so-called PAPI components) and/or the specific events being monitored. This is a preliminary step where both heterogeneous platforms (including several PAPI components) and some Key Performance Indicators (KPI), such as energy consumption estimation, will be supported [9].

3 RESULTS

Relying on PAPI monitoring correctness, this section gathers the performance results of the Papify-PREESM code generator while working with different use case applications. Additionally, the overhead associated to the *papification* of the application is also evaluated. The results obtained are for a quad-core Intel Core i5-4440 processor running at 3.10 GHz with 8 GB of RAM memory.

3.1 Application Monitoring

In the experiments, two applications widely utilized by PREESM developers are tested, Sobel filter [12] and Stereo matching [8]. The former features simple actors (in terms of computational complexity) while the latter has complex ones. This allows to characterize the monitoring overhead for different actor granularities.

In this work, the set of experiments used to evaluate the system behavior is based on varying the number of PAPI events being monitored from 0 to 8. Specifically, Figure 1 gathers the *Throughput*, i.e., executions per second, associated to each experiment. *C execution*, which refers to the standard C code execution generated by PREESM, is compared with two *Papify-PREESM* configurations: *timing*, in which PAPI just times the execution (0 events); and *N events*, the configurations monitoring from 1 to 8 PAPI events.

The experiments show that the behavior of the application with complex actors (Stereo matching) is not affected by PAPI monitoring and its average performance remains constant for every configuration. On the other hand, for simple actors (Sobel) the performance is hugely reduced when the number of monitored events increases. Specifically, as can be seen in Figure 1, when compared with the C executions, the maximum performance loss for the Stereo matching application is 5% (for the 1-Core and 8 Events experiment) while the Sobel performance loss reaches a 67% for the 4-Core and 8 Events experiment. It should be pointed out that the minimum values of Stereo matching 3-4 Events are outliers and represent less than a 0.1% of the performed experiments.

3.2 Overhead Study

In order to understand the performance loss cause, a deeper study of the *papification* has been performed. Specifically, the extra functions execution time has been measured for the 1-Core configuration of both applications. As this study focuses on computing the overhead associated to the instrumentation, only the sequential execution of the application will be evaluated. This case will provide more realistic results as only sequential kernel requests will be involved.

During this study the `event_write_file()` function has not been taken into account, as it is considered part of the prototype version of the *Papify-PREESM* code generator, which will be changed for a lighter approach. Nevertheless, its contribution can be considered negligible when compared to the rest of the *papification* function set. Likewise, both `event_start_PAPI_timing()` and `event_stop_PAPI_timing()` execution times are also insignificant. Consequently, the overhead is mainly associated to the PAPI event monitoring, i.e., `event_start()` and `event_stop()` functions. Specifically, the time contribution of the last ones are 100 times larger (above the μs) than the one associated to the others.

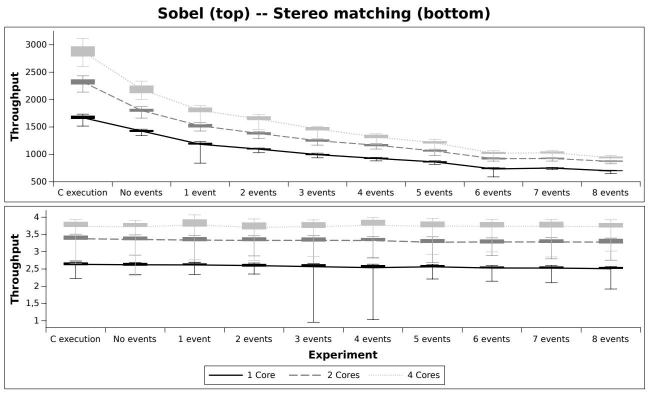

Figure 1: Throughput comparison for Sobel (top) and Stereo matching (bottom) applications and 1-Core (solid line), 2-Core (dashed line) and 4-Core (dotted line) configurations

Table 1 gathers the times (in μs) obtained during this study organized by the number of events being monitored (first column) and the application (first row). Additionally, three values are displayed for each experiment: (i) the average computation time associated to each `event_start/stop` set of instructions execution is shown under the label PT (*PAPI Time*); (ii) TET gathers the *Total Execution Time* of each configuration and is obtained as the inverse of the *Throughput* value; (iii) the total time dedicated to monitoring in each execution, hereafter PTT, which stands for *PAPI Total Time*. It should be noted that PTT values are computed as PT multiplied by the number of actors of each application, which are 12 and 94 for Sobel and Stereo matching, respectively.

Comparing both scenarios, it can be observed that PT times are larger in the Stereo matching case, which means that context switching for PAPI kernel requests are slower in this situation due to the higher actor complexity. Likewise, for both applications, it can be seen that the complexity increment associated to larger event sets also increases the time required to perform kernel requests. Furthermore, the figure associated to PTT is also larger for the Stereo matching application, which means that PAPI monitoring consumes more time per execution in this scenario.

Nevertheless, analyzing Table 1 TET column, it can be deduced that the low complexity of the Sobel application actors implies a TET of around 1ms, which is less than a 1% of the one required for Stereo matching. Consequently, as shown in figure 2, the overhead associated to monitor the Sobel application (black line) in real-time, which is computed as PTT divided by TET of each experiment, in the worst case (monitoring 8 Events), reaches a 45.83% in this case while for the stereo matching (red line) reaches only a 4.27%.

Therefore, both actor complexity and the monitoring strategy should be evaluated together so as to reach an acceptable approach where the instrumentation becomes precise and efficient. Specifically, two preliminary strategies are proposed: (i) grouping actors with low complexity and instrumenting them as a whole; (ii) increase the granularity of the instrumentation and monitor each core.

Table 1: Execution time (µs) associated to PAPI monitoring for Sobel and Stereo matching applications and monitoring 1 to 8 events. *PAPI Time* (PT) measures single executions of `event_start/stop()` instructions. TET the *Total Execution Time* of the experiment. PTT the *PAPI Total Time* dedicated to execute PAPI instructions

	Sobel			Stereo matching		
Events	PT	TET	PTT	PT	TET	PTT
1	7.81	844.07	93.72	19.50	381,679.39	1,833.00
2	14.24	912.21	170.88	32.28	384,615.38	3,034.32
3	19.66	1,004.24	235.92	48.83	387,596.90	4,590.02
4	25.19	1,078.85	302.28	67.13	393,700.79	6,310.22
5	32.41	1,158.84	388.92	89.71	390,625.00	8,432.74
6	46.18	1,360.56	554.16	146.10	395,256.92	13,733.40
7	46.91	1,332.05	562.92	144.27	395,768.61	13,561.38
8	54.55	1,428.39	654.60	180.80	398,406.37	16,995.20

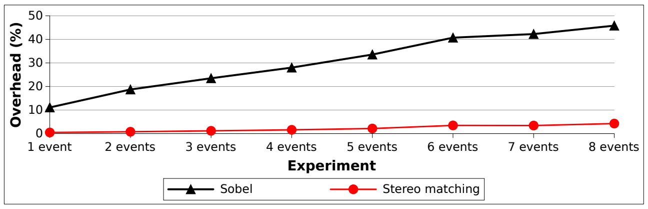

Figure 2: Overhead in % for Sobel (triangles) and Stereo matching (circles) applications for 1 to 8 events experiments

In future works, these strategies will be addressed and compared with the one presented in here in terms of accuracy and overhead.

Finally, concerning the values obtained from the actor timing, it should be noted that the actor execution time remains stable for every experiment. In consequence, for characterization purposes, this approach allows developers to obtain a suitable estimation of the system behavior if no instrumentation was included.

4 CONCLUSION AND FUTURE WORK

In this work, the Papify-Preesm code generation has been presented as a proof-of-concept where the integration of a rapid prototyping tool such as Preesm and PAPI monitoring has been carried out. By doing so, an automatic instrumented code generator has been included within Preesm Y-chart design flow. Papify-Preesm enables transparent actor timing and hardware resource usage profiling for developers. Moreover, it has been demonstrated that monitoring dataflow applications with complex actors is also transparent in terms of execution time (overhead below 5%).

However, the overhead for actors with a small execution time can reach more than 50%. This is an issue that will be addressed in the future by exploring new monitoring strategies. Additionally, instrumentation of heterogeneous architectures where both FPGA and ARM/x86 components work together will be studied within Papify-Preesm. Finally, the development of a real-time resource manager will be evaluated. Using Spider [3] will then allow system reconfiguration at runtime according to the information obtained from the PMC values. This resource manager could work with hardware information obtained directly from PAPI events or estimating KPI based on them, e.g., power and energy consumption.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation program under Grant No.: 732105 (CERBERO project).

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.
[2] Azzam Haidar, Heike Jagode, Asim YarKhan, Phil Vaccaro, Stanimire Tomov, and Jack Dongarra. 2017. Power-aware computing: Measurement, control, and performance analysis for Intel Xeon Phi. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE.* IEEE, 1–7.
[3] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-Francois Nezan, and Slaheddine Aridhi. 2014. Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in.* IEEE, 167–171.
[4] Bart Kienhuis, Ed F Deprettere, Pieter Van der Wolf, and Kees Vissers. 2001. A methodology to design programmable embedded systems. In *International Workshop on Embedded Computer Systems.* Springer, 18–37.
[5] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. 2008. The vampir performance analysis tool-set. *Tools for High Performance Computing* (2008), 139–155.
[6] Edward A Lee. 2008. Cyber physical systems: Design challenges. In *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on.* IEEE, 363–369.
[7] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. DOI: http://dx.doi.org/10.1109/PROC.1987.13876
[8] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. 2014. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in.* 36–40. DOI: http://dx.doi.org/10.1109/EDERC.2014.6924354
[9] Rong Ren, Eduardo Juarez, Cesar Sanz, Mickael Raulet, and Fernando Pescador. 2014. Energy estimation models for video decoders: reconfigurable video coding-CAL case-study. *IET Computers & Digital Techniques* 9, 1 (2014), 3–15.
[10] Rong Ren, Jianguo Wei, Eduardo Juarez, Matias Garrido, Cesar Sanz, and Fernando Pescador. 2013. A PMC-driven methodology for energy estimation in RVC-CAL video codec specifications. *Signal Processing: Image Communication* 28, 10 (2013), 1303–1314.
[11] Marc Schlütter, Bernd Mohr, Laurent Morin, Peter Philippen, and Markus Geimer. 2014. Profiling Hybrid HMPP Applications with Score-P on Heterogeneous Hardware. In *International Conference on Parallel Computing.* Jülich Supercomputing Center.
[12] Leonardo Suriano, Alfonso Rodriguez, Karol Desnos, Maxime Pelcat, and Eduardo de la Torre. 2017. 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, 1–7.
[13] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009.* Springer, 157–173.
[14] Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet. 2013. Orcc: Multimedia Development Made Easy. In *Proceedings of the 21st ACM International Conference on Multimedia (MM '13).* ACM, 863–866. DOI: http://dx.doi.org/10.1145/2502081.2502231