Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS)

1. September 2012
Oslo, Norway

edited by
Tobias Becker

Preface

The current landscape of computing sees a continuing trend towards increasingly complex, heterogeneous and distributed systems. This does not only raise the question of how to efficiently develop applications for such systems, but also how to cope with dynamic changes in the application requirements or the system itself. Self-awareness is an emerging field of research in computing that considers systems and applications that gather and maintain information about their current state and environment, reason about their behaviour, and adapt themselves if necessary.

Reconfigurable hardware, such FPGAs, is a technology that is becoming increasingly relevant to embedded and high-performance applications. Being able to adapt the functionality through dynamic reconfiguration is an inherent benefit of reconfigurable devices. Over the past decade, significant progress has been made in tools and methods for approaches that require the exchange of a number of functional units in a pre-defined scenario. However, it is still an open question how we can harness the benefits of reconfigurable technology for systems that automatically adapt to changing requirements or environments. Recent research has investigated several so-called self-* properties such as self-modifying, self-optimising or self-healing as a means of improving flexibility, performance or reliability of applications targeting reconfigurable hardware. Self-awareness extends this line of research and includes aspects such as reasoning, learning and intelligence to a run-time adaptive system.

The Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS) was created to bring together researchers who are active in this field, present their current work, and share their concepts and visions of self-aware systems. The topics of interest for this workshop are:

- Concepts and foundations of self-aware systems.
- Architectures, control and infrastructure for self-aware systems.
- Algorithmic approaches for self-awareness.
- Engineering self-aware reconfigurable systems.
- Advanced autonomous and self-adaptive systems.
- Applications using self-awareness or self-adaptivity.

The workshop was held on 1. September 2012 in Oslo, Norway, and co-located with the 2012 International Conference on Field Programmable Logic and Applications (FPL). Of all papers submitted to this workshop, 6 were selected for regular presentations and 6 were accepted as posters. In addition, we were able to attract 3 invited speakers from industry and academia, resulting in a diverse program that covers many aspects of self-aware systems. We would like to thank all authors for submitting their work to the workshop. We would also like to thank the program committee for reviewing papers and helping with the paper selection. We gratefully acknowledge the financial support of Awareness, a FET coordination action funded by the European Commission under FP7. Special thanks go the the FPL organisers who helped us co-locating this workshop with FPL 2012.

Tobias Becker, Imperial College London
Marco Platzner, University of Paderborn
Arjun Chandra, University of Oslo

Program Committee

David Andrews	University of Arkansas
Tobias Becker	Imperial College London
Arjun Chandra	University of Oslo
Suhaib Fahmy	Nanyang Technological University
Michael Hübner	Ruhr-University of Bochum
Peter Jamieson	Miami University
Enno Lübbers	Intel
Marco Platzner	University of Paderborn
Jürgen Teich	University of Erlangen-Nürnberg
David Thomas	Imperial College London
Andy Tyrrell	University of York

Additional Reviewers

Dennl, Christopher
Schmidt, Bernhard
Todman, Tim
Wildermann, Stefan
Ziener, Daniel

Author Index

Table of Contents

INVITED TALK: AN OUTLOOK FOR SELF-AWARENESS IN COMPUTING SYSTEMS

Peter Lewis

School of Computer Science
University of Birmingham
United Kingdom
email: p.r.lewis@cs.bham.ac.uk

Talk summary

Novel computing systems are increasingly being composed of large numbers of heterogeneous components, each with potentially different goals or local perspectives, and connected in networks which change over time. Such future computing systems, from robots to personal music devices to web services, should be able to achieve advanced levels of autonomous behaviour, in order to adapt themselves at run-time and learn behaviours appropriate to changing conditions. Nevertheless, users engaging with different parts of the system still expect high performance, reliability, security and other qualities. Such systems will therefore be faced with the challenge of managing trade-offs between these conflicting goals at run-time, both at the global and at the local level, in response to changing conditions, and sometimes with humans in the loop.

In order for a system to effectively adapt itself, its ability to be self-aware becomes important. Self-awareness is concerned with the availability, collection and representation of knowledge about something, by that something. A self-aware node has knowledge of itself, permitting reasoning and intelligent decision making to support effective, autonomous adaptive behaviour. Such self-information might include its internal state, its history, its social or physical environment, its goals, or perhaps even its own way of representing or reasoning about these things.

There are several clusters of research in computer science and engineering which have used the term self-awareness explicitly. However, there is no general methodology or common framework for describing or benchmarking the self-awareness capabilities of these systems, or the benefits that self-awareness brings. In this talk, I shall survey definitions and current understanding of self-awareness in psychology, focussing on three key concepts: public and private self-awareness, different levels of self-awareness, and the emergence of self-awareness in collective systems. I will then attempt to translate these concepts from psychology to the computing domain, and show how their explicit consideration may be beneficial in the engineering of adaptive computing systems. Based on this understanding, I shall present a working definition for self-aware computing systems, with the aim of providing common ground for future discussions.

I will then describe some prospects for realising an increased capacity for self-awareness in computing systems, and what will be required in order to achieve the increased adaptivity and robustness that the vision promises. Online learning is expected to play a key role, as will techniques for decision-making in the presence of multiple objectives, representative of the conflicting goals of adaptive nodes in dynamic heterogeneous environments.

Finally, I will describe some challenges which need to be faced in both developing and applying self-aware systems. For example, how should self-aware systems learn and adapt to changing conditions at run-time, considering trade-offs both between system goals and the overheads associated with learning itself? Another critical question is that of how to formulate claims about what we can expect from self-aware systems, when deployed in uncertain and dynamic environments. As should become clear, there is still much to understand about how to incorporate self-awareness properties into computing systems.

About the speaker

Peter Lewis is a post-doctoral research fellow at the Centre of Excellence for Research in Computational Intelligence and Applications (Cercia) in the School of Computer Science at the University of Birmingham. His research is concerned with investigating algorithms and techniques for achieving self-awareness and self-expression in decentralized computational systems. Particular focuses include economics-inspired computational techniques and online learning algorithms, such as those using evolutionary computation and other nature-inspired techniques. He obtained his PhD from the University of Birmingham in 2010, on the topic of evolutionary market-based resource allocation in decentralised computational systems.

INVITED TALK: SELF-AWARENESS AND ADAPTIVE TECHNOLOGIES: ARE THEY THE FUTURE OF OPERATING SYSTEMS?

Lamia Youseff

Google, Inc.
email: lyouseff@google.com

Talk summary

Trends in multicore architectures point to an ever-increasing number of cores available on a single chip. Moore's Law predicts an exponential increase in integrated circuit density; in the past, this increase in circuit density has translated into higher single-stream performance, but recently single-stream performance has plateaued and industry has turned to adding cores to increase processor performance. In a few years, multicores have gone from esoteric to commonplace.

Given exponential scaling, it will not be long before chips with hundreds of cores are standard, with thousands of cores following close behind. This new architecture trend is providing an exciting opportunity for exploring different research directions in scaling operating systems. At the same time, a similar, independent trend can be seen in the growth of cloud computing. Rather than consolidating a large number of cores on a single chip, cloud computing consolidates multicore machines within a data center. There is much commonality between constructing OSs for clouds and multicores, such as large-scale resource management, heterogeneity, and possible lack of widespread shared memory [1]. These similarities allow operating systems to be designed for both multicore and cloud computers.

The primary question facing OS designers over the next ten years will be: *What is the correct design of OS services that will scale up to hundreds or thousands of cores?* We argue that the structure of monolithic OSs is fundamentally limited in how they can address this problem. In contrast, our work explored a new factored structure for the OS, which we dubbed *fos* for "factored operating system" [2]. The structure of fos brings scalability concerns to the forefront by decomposing an OS into services, and then parallelizing within each service. To facilitate the conscious consideration of scalability, fos system services are moved into userspace and connected via messaging. In fos, a set of cooperating system servers which implement a single system service is called a *fleet*. However, a fundamental research challenge in this design is to identify the characteristics of such a fleet. Given the unprecedented variability in demand of the system resources, the OS fleets have to deploy elastic techniques to adapt to this variability at runtime.

We argue that the OS services have to deploy elastic techniques to adapt to this variability at runtime. In this talk, we advocate for elastic OS services, illustrate their feasibility and effectiveness in meeting the variable demands through our prototype system, dubbed *elastic fos* or "*e-fos*", which provides elastic technologies for OS services in the fos operating system [3]. We furthermore showcase a prototype elastic file system service in *e-fos* and illustrate its effectivness in meeting variable demands.

About the speaker

Dr. Youseff holds a Ph.D. degree in Computer Science from the University of California, Santa Barbara and has received her post-doctoral training at MIT working at CSAIL with Professor Anant Agarwal on research in cloud computing, operating systems and next generation exascale computing paradigm. Dr. Youseff has received several awards, including the international ACM-UPE'02, the AUC presidential cup'03, XHPC best paper award'06 and IPDPS best poster award'08 at the TCPP forum. In addition, she has served as a program committee member, program co-chair and organizer to several top conferences and workshops in computing systems, including SOCC, DAC and USENIX Middleware. She also has tens of technical publications and book chapters with more than 1000 non-self citations. She is currently a research software engineer in cloud computing at Google Seattle, WA office. Dr. Youseff worked on fos and e-fos with the fos Carbon group between 2009 and 2011 when she was affiliated with CSAIL, MIT.

1. REFERENCES

[1] K. Modzelewski, J. Miller, A. Belay, N. Beckmann, C. Gruenwald III, D. Wentzlaff, L. Youseff, and A. Agarwal, "A Unified Operating System for Clouds and Manycore: fos." [Online]. Available: http://mit.dspace.org/handle/1721.1/49844

[2] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "An operating system for multicore and clouds: mechanisms and implementation," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. ACM, 2010, pp. 3–14.

[3] L. Youseff, N. Beckmann, H. Kasture, C. Gruenwald, D. Wentzlaff, and A. Agarwal, "The case for elastic operating system services in fos," in *Proc. of the 49th Annual Design Automation Conference*, ser. DAC '12. ACM, 2012, pp. 265–270.

INVITED TALK: THE CHANGE PROJECT! ENABLING TECHNOLOGIES FOR SELF-AWARE ADAPTIVE COMPUTING SYSTEMS

Marco D. Santambrogio

Dipartimento di Elettronica e Informazione, Politecnico di Milano
email: marco.santambrogio@polimi.it

Talk summary

The computing industry is at a major crossroad. Semiconductor technology offers billions of transistors on a chip, the level of integration and the speed of components have increased dramatically over the years, and advancements show no sign of abating. Post silicon technologies such as graphene electronics could offer even more industrially viable computing power within years. In recent times, unfortunately, these advancements have not resulted in a proportional increase in performance or other measures of interest to users. Even though technologies have improved, we continue to apply outdated approaches to our use of these resources, and key computer science abstractions have not changed since the 1960's. Furthermore, reconfigurable architectures and multicore processors are becoming prevalent. Therefore, the complexity of computing systems is increasing up to the point that it is no longer practical for an average programmer to balance all of the system constraints and produce applications that perform well on a variety of machines, in a variety of situations. Within this context, this is the right time for a fresh approach to the way systems are designed and used.

Imagine a revolutionary computing system that can observe its own execution and optimize its behavior with respect to the external environment, the user desiderata and the applications demands. Imagine providing users with the possibility to specify their desired goals rather than how to perform a task, along with constraints in terms of energy budget, time, and results accuracy. Imagine, further, a computing chip that performs better, according to a set of goals expressed by the user, the longer it runs an application. Self- Aware computing is a research area aimed at leveraging the new balance of resources to improve performance, utilization, reliability and programmability, overcoming the burden imposed by the increasing complexity and the associated workload of modern computing systems. Self-aware computing systems will be able to configure, heal, optimize, protect themselves and improve interaction with the user and the environment without the need for human intervention, through learning abilities that will allow them to automatically find the best way to accomplish a given goal with the resources at hand. Within this context, the need for a systematic approach to the design of architectures and systems enabling self-awareness has been motivated by some trends that have gained momentum in the past few years. On one hand there is the increasing importance of non-functional constraints in the perceived value of a digital system; features that cannot be completely translated to functionalities are getting more important. On the other hand there is the increasing structural complexity of devices, which in turn increases the number and the complexity of interacting peer elements at various levels, e.g., cores on a multicore processor, concurrent programs in a multitask operating system, number of threads within an application.

Within this context, this talk present the work that we are doing in proposing a new way of thinking and approaching computer systems that reflects 21st century demands and opportunities. During this talk, after a general presentation of the overall CHANGE project two examples, morphone and AcOS and will be presented and the relation between CHANGE and DRESD (the reconfigurable computing research project) will be discussed.

About the speaker

Marco D. Santambrogio received his laurea (M.Sc. equivalent) degree in Computer Engineering from the Politecnico di Milano in 2004, his second M. Sc. degree in Computer Science from the University of Illinois at Chicago (UIC) in 2005 and his PhD degree in Computer Engineering from the Politecnico di Milano in 2008. Dr Santambrogio was at the Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT as postdoc fellow and he is now assistant professor at Politecnico di Milano, research affiliate at MIT and adjunct professor at UIC. Marco D. Santambrogio is a member of the IEEE, the IEEE Computer Society (CS) and the IEEE Circuits and Systems Society (CAS). He has been with the Micro Architectures Laboratory at the Politecnico di Milano, where he founded the Dynamic Reconfigurability in Embedded System Design (DRESD) project in 2004 and the CHANGE project (Self-Aware and Adaptive Computing Systems) in 2010. He conducts research and teaches in the areas of reconfigurable computing, computer architecture, operating system, hardware/software codesign, embedded systems, and high performance processors and systems.

SYSTEM LEVEL SYNTHESIS FLOW FOR SELF-ADAPTIVE MULTI-MODE RECONFIGURABLE SYSTEMS

Stefan Wildermann, Felix Reimann, Daniel Ziener, Jürgen Teich

University of Erlangen-Nuremberg, Germany
{stefan.wildermann, felix.reimann, daniel.ziener, juergen.teich}@cs.fau.de

ABSTRACT

This paper presents a synthesis flow to design self-adaptive multi-mode reconfigurable systems on the system level. Such systems are able to react on environmental changes by switching operational modes through hardware reconfiguration. Thus, they can provide context-aware processing while efficiently utilizing the (constrained and restricted) system resources.

1. INTRODUCTION

Embedded systems of any kind should have low cost, be small and power efficient. This implies design constraints (regarding these objectives but also requirements like real time capabilities) and limited capacity for providing functionality on the one hand. On the other hand, many embedded systems, e.g., embedded smart cameras, are operating in unknown, highly dynamic, and often unpredictable real world environments so that a variety of complex algorithms is required for a robust operation of the system. Due to constraints, only a subset of these algorithms may be used concurrently in a configuration of the system. As a solution for this tradeoff, context-aware and resource-aware adaptation by re-organizing the configuration of algorithms at run-time can lead to a better utilization of the system resources in the presence of constraints and restrictions while retaining and possibly even optimizing the processing quality of the system. Here, reconfigurable hardware is a solution to further increase the flexibility of the system despite these constraints by offering the capability of sharing hardware resources between different configurations mutually exclusive.

2. SELF-ADAPTIVE MULTI-MODE SYSTEMS

In a more abstract view of this system model, the set $\mathcal{G} = \{G_i \mid i = 1, ..., n\}$ denotes all n algorithms which are provided by the designer. During run-time, different combinations of these algorithms can be executed on the available architecture, each representing an *operational mode* O of the system. Ideally, each possible combination of algorithms could be run in the system. However, due to aforementioned constraints, only a small subset of configuration may actually constitute feasible operation modes, and only such modes are allowed to be executed.

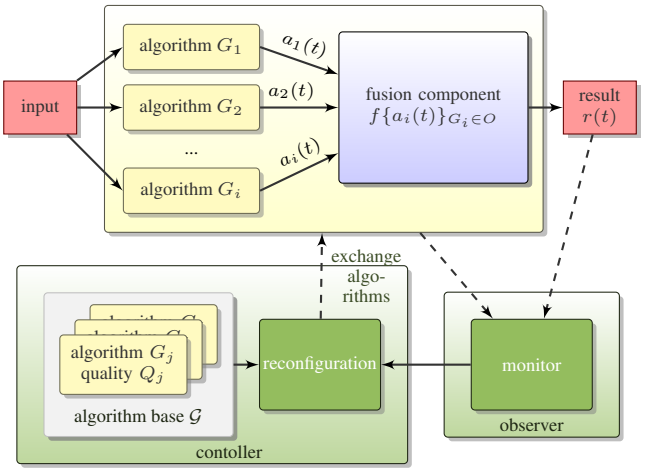


Fig. 1. Self-adaptive system architecture fusing the results of multiple algorithms and adapting this configuration when the current algorithms do not work efficiently.

Therefore, the idea is to additionally equip the multi-mode system with an autonomous *Control Mechanism (CM)* which is able to observe and control the system [1]. The purpose of the CM is to detect environmental changes and degeneration of the system's processing quality (*observe*). It can then react by modifying the system configuration through a transition to a new operational mode (*control*). The system architecture as illustrated in Fig. 1 is described next.

2.1. Self-adaptive System Architecture

The operational mode of the system at instant of time t is represented by the set of active algorithms $O(t) \subseteq \mathcal{G}$, where each algorithm $G_i \in O(t)$ calculates a result $a_i(t)$. These results are fused by a fusion function f to produce the result $r(t)$ of the overall system at time t:

$$r(t) = f\{a_i(t)\}_{G_i \in O} \tag{1}$$

The *observer* evaluates the quality of each active algorithm $G_i \in O(t)$ by an adequate quality function $\tilde{q}(a_i(t), r(t))$, which measures how good a filter is predicting the result $r(t)$:

$$\tilde{q}(a_i(t), r(t)). \tag{2}$$

The *normalized qualities* q_i are then given as

$$q_i(t) = \frac{\tilde{q}(a_i(t), r(t))}{\sum_{G_j \in O} \tilde{q}(a_j(t), r(t))} \tag{3}$$

so that the sum of all qualities sum up to 1. The number N_{eff} of algorithms which are now efficiently contributing to the system output can be calculated based on these qualities as:

$$N_{eff} = \frac{1}{\sum_{G_i \in O(t)} \left(q_i(t)\right)^2} \tag{4}$$

Furthermore, a long term estimate Q_i of the algorithms qualities is generated according to

$$Q_i = (1 - \lambda) \cdot Q_i + \lambda \cdot q_i(t). \tag{5}$$

The *controller* takes these results to test whether a reconfiguration of the system is necessary. If so, the new system configuration $O(t+1)$ has to be determined. This decision is based on a *fitness value* $Z(O)$ used for each mode O. It is calculated as the multiplied qualities of its algorithms according to

$$Z(O) = \prod_{G_i \in O \cap O(t)} q_i(t) \cdot \prod_{G_i \in O \setminus O(t)} Q_i. \tag{6}$$

The fitness value uses the actual qualities of all algorithms which are part of the current mode $O(t)$, and the estimated qualities of inactive algorithms.

Algorithm 1 outlines the decision process for reconfiguration. Adaptation is possible at the earliest θ_{mod} time steps after having performed the previous modification at time step t_{mod} (line 1). This is required to give the system some time to evaluate the quality of the new algorithms in the current context. No modification is necessary if the system efficiently manages to track an object. Therefore, adaptation is only performed if the efficiency N_{eff} is below a predefined threshold (line 2). Note that threshold $\theta_{eff}(O)$ may depend on the configuration since configurations may contain different numbers of filters. For example, in case at least 75% of the active filters should contribute to the result, the threshold would be defined as $\theta_{eff}(O(t)) = 0.75 \cdot |O(t)|$.

Now, one of two behaviors is performed:

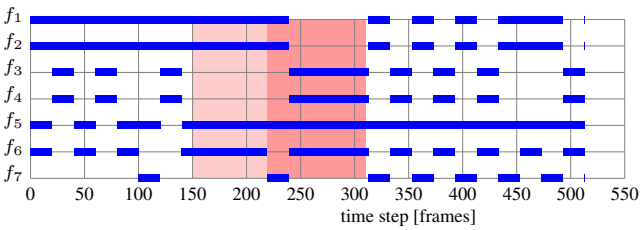
- Exploitation: With a probability of $p_{exploit}$, the controller selects that **feasible** mode $O(t+1)$ which has the maximal fitness value.
- Exploration: With a probability of $(1 - p_{exploit})$, the controller selects a **feasible** mode $O(t+1)$ randomly with probabilities proportional to their fitness values.

2.2. Smart Camera Case Study

A smart camera application from [2] serves as a case study to illustrate the behavior of systems implemented according to above system architecture. It performs person tracking based on the image processing filters illustrated in Figure 2. The system executes a subset of these filters on the same input image and fuses their results via a tracking algorithm (cf. [2]). The tracking result is used to calculate the filter qualities, indicating how good each filter has predicted this result. The qualities are used to perform the adaptation as described above.

Algorithm 1: Control mechanism for reconfiguration decision, which exchanges algorithms if necessary, either by a behavior performing exploitation or exploration.

```
1 if t − t_mod > θ_mod then
2     if N_eff < θ_eff(O(t)) then
3         generate random number rnd ∈ [0, 1];
4         if rnd ≤ p_exploit then
5             doExploitation();
6         else
7             doExploration();
8         t_mod = t;
```



(a) Gantt chart of system setup

Fig. 3. Gantt chart for a test sequence with color corruption. The person is visible in the highlighted interval and color corruption happens in the interval with darker color from frame 219.

Fig. 3 illustrates the system behavior for a image test sequence. The Gantt chart illustrates the time intervals when each filter is active. In this test sequence, no person is visible between frames 0 and 150, and the system is arbitrarily switching configurations after every $\theta_{mod} = 20$ time steps (frames) according to Algorithm 1. The person appears in the scene around frame 150, and the system is successfully tracking the person with color-based filters and edge-based filters being loaded. A color corruption happens at frame 219 where the input image is switched to gray scale. As the two color-based filters are unable to produce an output, N_{eff} falls below the threshold. The system adapts until the color filters are removed from the system and replaced by the more adequate motion-based filters. When the person leaves at frame 310, all filters fail and the system switches between configurations with after each θ_{mod} time steps.

(a) input (b) skin color in RGB (f_1) (c) skin color in YCbCr (f_2) (d) motion detection (f_3) (e) background subtraction (f_4) (f) Canny edge detection (f_5) (g) edge background (f_6) (h) Sobel edge detection (f_7)

Fig. 2. Examples of the filters used in the smart camera case study for person tracking. Filters f_1 and f_2 are *color*-based, filters f_3 and f_4 are *motion*-based, and filters f_5, f_6, and f_7 are *edge*-based.

2.3. Design Challenges

For performing the system adaptation, it is necessary to determine the feasible modes of a system. However, system synthesis in the presence of stringent design constraints (restricted bandwidth, reconfigurable hardware, processor utilization, etc.) is known to be NP-complete (cf. [3]). We therefore propose a system level design methodology since it would be too costly or even infeasible to select, verify, and optimize each configuration at run-time.

Furthermore, design constraints limit the amount of combinations of algorithms that can be implemented as feasible modes of the systems. Therefore, resource sharing becomes a key concept to increase the number of feasible operational modes: Even if not all algorithms are able to be executed concurrently, subsets of algorithms can be executed on the same resources as mutually exclusive operational modes. Of course, sharing of computational resources can be achieved by providing a schedule for each mode independently. However, through the use of reconfigurable hardware, it is also possible to share hardware resources between modes. This allows the cost and size to be decreased, while increasing the resource utilization of the system. In this work, Field Programmable Gate Array (FPGA) technology is the implementation target.

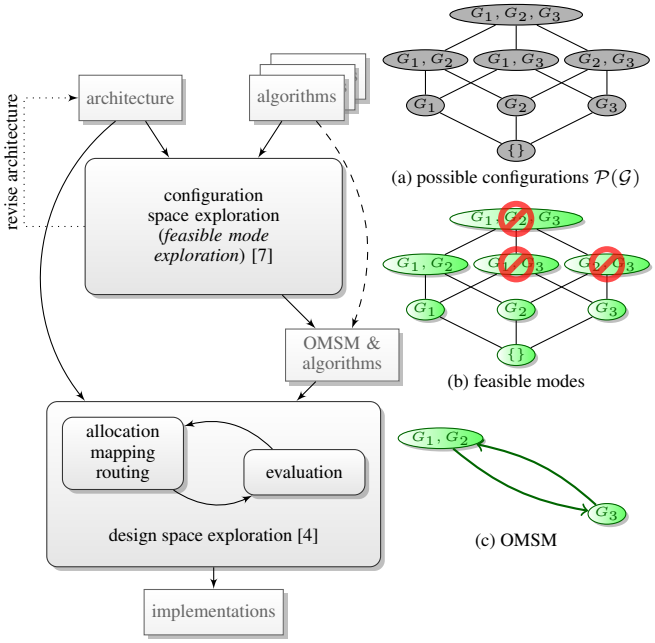
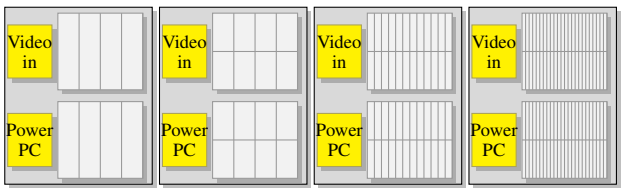


Fig. 4. System level synthesis flow. The first CSE phase identifies the feasible modes for a given specification. In the second DSE phase, the multi-mode reconfigurable system is synthesized through iterative optimization.

3. DESIGN FLOW

The methodology illustrated in Fig. 4 contains two mandatory design phases: The first one is *configuration space exploration* (CSE). For a given specification, the power set of $\mathcal{G}$ constitutes the possible configurations (see Fig. 4(a) for an example with three algorithms). The purpose of CSE is to evaluate which of these configurations can be executed on a given reconfigurable architecture layout as feasible modes despite the constraints (see, e.g., Fig. 4(b)). We define an exploration model in [4] that captures the behavioral aspects of self-adaptive systems and the spatial and technological aspects of FPGA-based reconfigurable system-on-chip architectures. We provide models in [5] for island style reconfiguration as well as 2-dimensional module placement according to [6]. For performing CSE, a symbolic encoding of this model is formulated that specifies the restrictions and design constraints as a *Pseudo Boolean (PB) Satisfiability problem*. By applying PB solvers, this encoding can be tested for satisfiability. We then apply an algorithm called *Feasible Mode Exploration algorithm* [7] that provides a scheme to efficiently apply this test for the exploration of the configuration space. This phase can be performed repeatedly to evaluate and compare different architecture layouts and templates, as indicated in Fig. 4. Based on the result of CSE, the designer can then select the configurations of the system as well as possible transitions between them. This is expressed by an *Operational Mode State Machine (OMSM)* [8], as illustrated in Fig. 4. The OMSM specifies how the CM can then switch between modes at run-time.

The second design phase is *Design Space Exploration (DSE)*. DSE is a multi-objective optimization problem with conflicting objectives like cost, area, power consumption, and reconfiguration time. Our DSE approach [4] applies the exploration model to derive several different implementations by (a) allocating resources from the architecture template and remove those not required, which allows the size of the architecture to be further reduced, (b) mapping tasks onto the allocated resources (which also includes the 2-dimensional placement of hardware modules), and (c) routing the communication between tasks. Each implementation is



(a) **A1** (b) **A2** (c) **A3** (d) **A4**

Fig. 5. Schematic outline of architecture options **A1** to **A4**.

evaluated regarding the objectives and iteratively further refined. The result of this phase is a set of non-dominated implementations regarding the specified objectives.

4. EXPERIMENTS

We have implemented and tested our synthesis flow using the publicly available PB solver *Sat4j* [9] and the publicly available optimization framework OPT4J [10]. The flow is applied to implement a self-adaptive reconfigurable smart camera according to the case study from Section 2.2 using image filter algorithms f_1 to f_6 from Fig. 2. The target architecture is a reconfigurable system-on-chip according to [6]. It contains two partially reconfigurable regions (PR regions) for 2-dimensional module placement, as well as a CPU-subsystem using a PowerPC. Four architecture alternatives are generated, the PR regions were divided in different granularities of 4×1 tiles (**A1**), 4×2 tiles (**A2**), 14×2 tiles (**A3**), and 28×2 tiles (**A4**) as illustrated in Figure 5. Partial modules can be placed by occupying one or several contiguous of such tiles.

As the application provides six image filter algorithms, a total of $64 = 2^6$ combinations are possible. The results of CSE for the four architecture alternatives are presented in Table 1. The results show that the finer the tiling, the more efficient resource sharing can be performed, resulting in more feasible modes. Moreover, the execution time can drastically vary depending on the complexity of the exploration model as well as the number of infeasible modes[1]. It shows that such a test could hardly be performed online.

Fig.6 shows the results of performing DSE for the smart camera case study with three operational modes. The optimization objectives are to minimize 1.) the average reconfiguration time, 2.) the power consumption, and 3.) cost of the used resources. Since this is a multi-objective optimization problem, we have chosen the ϵ-dominance where low values $\epsilon \in [0, 1]$ indicate results with higher quality. The symbolic DSE (*dse*) is compared to a state-of-the-art DSE based on an Evolutionary algorithm [8] (*moea*). The results show that the proposed symbolic approach converges much faster to the optimum, while *moea* is trapped in a local optimum.

[1]The latter being the case for **A2**. Testing infeasible modes normally takes longer than testing feasible modes

Table 1. Results of CSE.

architecture	# of feasible modes	avg. execution time [min]
A1	16	0.28
A2	57	204.06
A3	62	3.47
A4	63	12.42

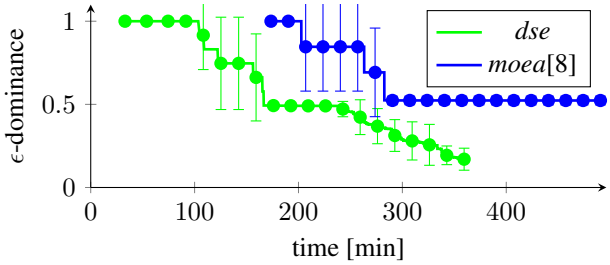


Fig. 6. Result of DSE. Average ϵ-dominance results for test case over the optimization time.

5. CONCLUSION

This paper proposes a system level design methodology for self-adaptive systems which switch between algorithmic configurations to maintain the system efficiency. The analysis, verification, and optimization of such systems at design time, as proposed by our methodology, is mandatory to guarantee feasible and highly optimized implementations.

6. REFERENCES

[1] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter, "Adaptivity and self-organisation in organic computing systems," in *Organic Computing – A Paradigm Shift for Complex Systems*, ser. Autonomic Systems. Springer Basel, 2011, vol. 1, pp. 5–37.

[2] S. Wildermann, A. Oetken, J. Teich, and Z. Salcic, "Self-organizing computer vision for robust object tracking in smart cameras," in *Proc. of ATC*, ser. LNCS. Springer-Verlag, 2010, pp. 1–16.

[3] T. Blickle, J. Teich, and L. Thiele, "System-level synthesis using evolutionary algorithms," *Design Automation for Embedded Systems*, vol. 3, pp. 23–58, 1998.

[4] S. Wildermann, F. Reimann, D. Ziener, and J. Teich, "Symbolic design space exploration for multi-mode reconfigurable systems," in *Proc. of CODES+ISSS*, 2011, pp. 129–138.

[5] S. Wildermann, J. Teich, and D. Ziener, "Unifying partitioning and placement for SAT-based exploration of heterogeneous reconfigurable SoCs," in *Proc. of FPL*, 2011, pp. 429–434.

[6] A. Oetken, S. Wildermann, J. Teich, and D. Koch, "A bus-based SoC architecture for flexible module placement on reconfigurable FPGAs," in *Proc. of FPL*, 2010, pp. 234–239.

[7] S. Wildermann, F. Reimann, J. Teich, and Z. Salcic, "Operational mode exploration for reconfigurable systems with multiple applications," in *Proc. of FPT*, 2011, pp. 1–8.

[8] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles, "Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities," *TCAD*, vol. 24, no. 2, pp. 153–169, 2005.

[9] D. Le Berre and A. Parrain, "The SAT4J library, release 2.2, system description," *JSAT*, vol. 7, pp. 59–64, 2010.

[10] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J – a modular framework for meta-heuristic optimization," in *Proc. of GECCO*, 2011, pp. 1723–1730.

HARDWARE/SOFTWARE PLATFORM FOR SELF-AWARE COMPUTE NODES

Markus Happe, Andreas Agne, Christian Plessl, Marco Platzner

University of Paderborn, Germany
email: {markus.happe, agne, christian.plessl, platzner}@uni-paderborn.de

1. INTRODUCTION

Today's design and operation principles and methods do not scale well with future reconfigurable computing systems due to an increased complexity in system architectures and applications, run-time dynamics and corresponding requirements. Hence, novel design and operation principles and methods are needed that possibly break drastically with the static ones we have built into our systems and the fixed abstraction layers we have cherished over the last decades. Thus, we propose a HW/SW platform that collects and maintains information about its state and progress which enables the system to reason about its behavior (self-awareness) and utilizes its knowledge to effectively and autonomously adapt its behavior to changing requirements (self-expression).

To enable self-awareness, our compute nodes collect information using a variety of sensors, i.e. performance counters and thermal diodes, and use internal self-awareness models that process these information. For self-awareness, online learning is crucial such that the node learns and continuously updates its models at run-time to react to changing conditions. To enable self-expression, we break with the classic design-time abstraction layers of hardware, operating system and software. In contrast, our system is able to vertically migrate functionalities between the layers at run-time to exploit trade-offs between abstraction and optimization.

This paper presents a heterogeneous multi-core architecture, that enables self-awareness and self-expression, an operating system for our proposed hardware/software platform and a novel self-expression method.

2. SELF-AWARE MULTI-CORE ARCHITECTURE

As architecture we propose a heterogeneous multi-core that consists of processors (that execute software threads) and reconfigurable hardware cores (that execute hardware threads). As prototyping platform, we use the programming model and execution environment ReconOS [1]. ReconOS extends the well-known multithreading approach to reconfigurable hardware. Here, hardware threads can access the same shared resources (i.e. shared memory, synchronization and communication primitives) like the software threads.

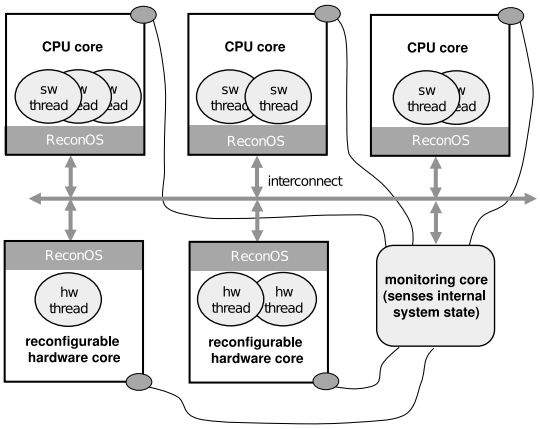


Fig. 1. Proposed heterogeneous multi-core architecture [2].

Figure 1 depicts an example architecture that consists of three processors and two (reconfigurable) hardware cores. The system contains a monitoring core that captures core-specific information. We currently support ring-oscillator based thermal sensors to capture the on-chip temperature distribution and performance counters to measure the system's performance.

3. OPERATING SYSTEM

ReconOS extends current operating systems (OS), i.e., Linux or eCos, to support reconfigurable hardware. In ReconOS, hardware threads are represented by delegate threads in software. Whenever a hardware thread makes an OS call, an interrupt is generated and the function name and its parameters are forwarded to delegate software thread. The delegate thread makes the OS call on behalf of the hardware thread and returns the results to the calling hardware thread.

Figure 2 gives an overview of the ReconOS system where the software threads interact directly with the OS kernel, while the hardware threads in the FPGA's logic are connected through operating system interfaces (OSIFs) and delegate threads. The operating system runs on one (main) processor while the other processors (workers) only execute software threads. The software on the worker processors are

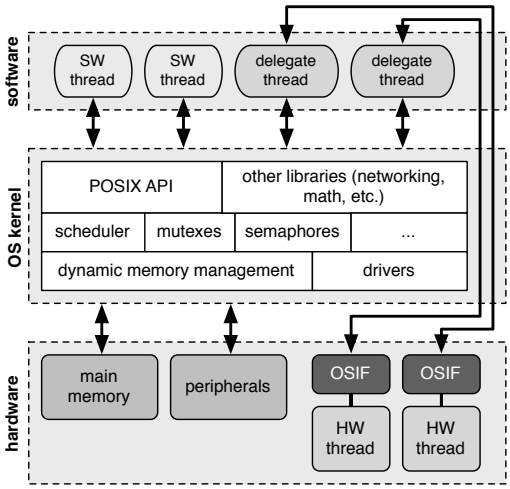


Fig. 2. Conceptual overview of the ReconOS system [1].

also represented by delegate threads on the main processor.

In extension to our previous work on ReconOS, we introduce vertical function migration as a novel self-expression method where the system can migrate threads between the software, OS and hardware layers at run-time to affect various aspects of the system such as performance, power consumption, temperature etc.

4. SELF-EXPRESSION BY THREAD MIGRATION

Migrating a thread from the main processor to a hardware core affects the performance, overall power consumption and the thermal profile of the chip. Considering heterogeneous processors similar effects can be observed for the migration between processors. Migrating a thread into the OS kernel is a special case which can lead to an improved performance if the operating system distinguishes between the a kernel space and a user space (such as Linux). In contrast to a user thread, a kernel thread can avoid internal context switches because it can directly access OS objects which results in an improved performance. For thread migration between hardware cores and between the hardware/software boundary we propose cooperative multitasking [3] where the threads have well-defined migration points and inform the OS every time they reach these points. Resuming execution form these migration points should be possible for both, the hardware and the software thread.

In [4] we demonstrated that a heterogeneous multi-core can regulate an application's performance by adding and removing software and/or hardware threads at run-time for a particle filter-based video object tracker. The performance of the application varies when a tracked object moves into the foreground or the background because this influences the computational complexity of the histogram calculation. The system measured the execution time of the individual threads in order to assess the application's performance. According to a static internal performance model, the system adapted the thread partitioning to either meet a user-defined lower performance bound or to stay within a performance budget. We currently work on a heterogeneous multi-core that learns a thermal model at run-time and performs thermal management autonomously using vertical thread migration.

5. CONCLUSION

To meet the rising dynamics, complexities and requirements of future computing systems, we propose that future systems collect and maintain information about their system state and their progress autonomously. Therefore, we have proposed a HW/SW platform that consists of heterogeneous HW/SW cores and a monitoring core which senses the current state of each core. To enable self-expression, we break with the classic design-time abstraction layers of hardware, operating system and software. In contrast, our proposed system vertically migrates threads between these layers to affect various system characteristics, such as application performance, temperature distribution and power consumption.

Acknowledgment

The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement n° 257906, from the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500) and the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

6. REFERENCES

[1] E. Lübbers and M. Platzner, "Communication and Synchronization in Multithreaded Reconfigurable Computing Systems," in *Int. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'08)*. CSREA Press, 2008.

[2] C. Plessl, M. Platzner, A. Agne, M. Happe, and E. Lübbers, "Programming Models for Reconfigurable Heterogeneous Multi-Cores," *Awareness Magazine, http://www.awareness-mag.eu*, 2012.

[3] E. Lübbers and M. Platzner, "Cooperative Multithreading in Dynamically Reconfigurable Systems," in *IEEE International Conference on Field Programmable Logic and Applications (FPL'09)*. IEEE, 2009.

[4] M. Happe, E. Lübbers, and M. Platzner, "A Self-adaptive Heterogeneous Multi-core Architecture for Embedded Real-time Video Object Tracking," *Journal on Real-Time Image Processing (JRTIP)*, 2011.

TOWARDS SELF-ADAPTATION IN RECONFIGURABLE NETWORK NODES

Ariane Keller, Daniel Borkmann, Stephan Neuhaus

Communication Systems Group
ETH Zurich
Gloriastrasse 35, 8092 Zurich
email: first.last@tik.ee.ethz.ch

1. INTRODUCTION

Today, computing nodes are everywhere, either visible as laptops and mobile phones or invisible as embedded devices in trains, home appliances or in the backbone of the Internet. The management of those devices becomes more and more difficult as the number of devices and the complexity of the applications increase. In order to cope with this complexity the idea of nodes with *self-**[1] features emerged.

In order to support self-* applications, Field Programmable Gate Arrays (FPGAs) are used as a basis to build (partially) reconfigurable hardware platforms. However, in order to use those platforms optimally, the software also needs to be redesigned. In previous work we developed the *Autonomic Network Architecture (ANA)* [1] which addresses the most important problems in the networking area: scalability, maintainability, and security. We also introduced a node architecture for ANA that is based on partially reconfigurable FPGAs and uses ReconOS [2] as its operating system [3]. The largest remaining challenge is to provide a self-aware algorithm that decides which protocols should be implemented in hardware and which in software.

In this paper we first briefly review the architecture of networking nodes (Section 2), and then tackle this problem by (i) establishing a reasonable adaptation frequency and classifying hardware/software mapping algorithms in the context of networking (Section 3), (ii) developing mechanisms to algorithmically recognize periodic traffic patterns that can be used as input to a mapping algorithm (Section 4); and (iii) developing a simulator for networked hardware/software systems that gives insights into the impact of different parameters (Section 5). The results will be used for further investigation in a self-aware hardware/software mapping algorithm for networking applications.

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n^o 257906.

[1]Where "*" can be a word such as healing, configuring, adaptation, etc.

2. NETWORKING NODE

For the rest of this paper, we assume that network nodes have the architecture shown in Figure 1; we describe this architecture as well as a prototype implementation more fully in previous work [3]. The whole system is implemented on an FPGA: the operating system runs on a CPU that is configured into the FPGA, and a configurable number of *hardware slots* are connected to the CPU over a shared bus interface. Networking blocks executing in hardware are configured into those slots, and all slots are connected with a dedicated interconnect that offers line rate communication. In hardware, packets are processed in a pipelined fashion. The hardware/software interface is implemented as a combination of shared memory and interrupts.

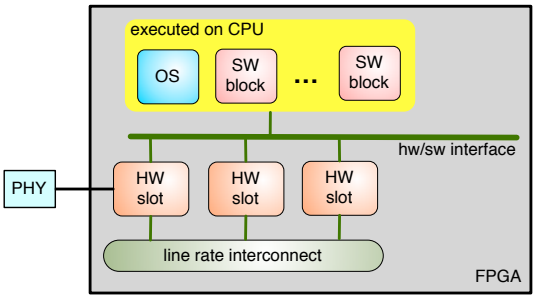


Fig. 1. Simplified node architecture.

3. ADAPTATION FREQUENCY

For the development of an adaptive system, it is crucial to determine an adequate adaptation interval. The underlying hardware limits the the maximum reconfiguration frequency, and hence the minimum time between reconfigurations, which can be computed as follows: if m is the time required to transmit sensor data to the reconfiguration application, c the time to compute the next hardware/software mapping, s_i the state relocation time in block i, r_i the reconfiguration time for block i, and n the number of blocks, the

total adaptation overhead A is then given by $A = m + c + \sum_{i=1}^{n}(s_i + r_i)$.

Adaptation overhead should be a small fraction of overall processing time so that the system spends most time processing traffic. We suggest limiting the adaptation overhead to 1% of the overall processing time. Applying these guidelines to the architecture we presented in [3], we obtain a maximum adaptation frequency of one adaptation per second. On end nodes, where traffic is user-dependent, this maximum frequency should be used so that the node remains responsive to traffic peaks. However, on nodes processing aggregated traffic (e.g., routers), a lower adaptation frequency should be used, in order to cope with seasonal effects [4] or trends [5]. This adaptation frequency could be set to one per hour.

4. ADAPTATION STRATEGIES

Regardless of the adaptation frequency, different strategies can be applied for the actual adaptation algorithm. Those strategies determine whether the actual traffic and/or learned traffic characteristics are taken into account and determine the actual optimization goal.

4.1. Traffic Dependent/Independent Algorithms

Traffic independent algorithms derive a mapping based only on the static properties of the required networking blocks. Such properties include the required FPGA area and memory footprint or the expected average benefit of a hardware implementation. The mapping only changes when new protocols are loaded. **Traffic dependent** algorithms take the actual traffic into account, e.g., by measurement of the number of packets and bytes processed by each block, the number of packets and bytes to be transmitted from hardware to software, and the utilization or energy consumption of the networking blocks.

4.2. Reactive versus Proactive Algorithms

A **reactive algorithm** makes an observation in time slot $t-1$, calculates an optimal mapping for this traffic distribution and uses this mapping in time slot t. This approach is especially useful for traffic changes introduced by a single user where the protocol and traffic mixes depend on user behavior, and which usually contain long periods of inactivity. However, we might also observe peaks in aggregated traffic, e.g., for protocols that are rarely used such as IPSec in the MAWI network [6].

A **proactive algorithm** changes the hardware/software mapping *before* the traffic mix changes. This requires knowledge from past traffic mix changes that can be applied to the current situation. On end nodes this might be a regular pattern for checking emails or storing a backup to a server. On intermediate nodes this might be the variation of network traffic that occurs on a daily basis.

We can distinguish between algorithms that *know the period* and algorithms that *learn the period*. Algorithms that know the period could assume for example that tomorrow's traffic will be much like today's. This approach is particularly useful for aggregated traffic.

End nodes may require a more sophisticated technique such as the *partial autocorrelation function* (PACF). For a given lag ℓ, the PACF is the correlation of the traffic at time $t-\ell$ with the traffic at time t, considering possible linear dependencies due to lesser lags; the significance level of this autocorrelation, i.e., the probability that the observed correlation will be this large by chance, can also be computed.

In Figure 2, a repetitive pattern is clearly visible for email traffic on an end node. In periodic signals, the PACF will be significant at lags that are multiples of the period: in the PACF for this traffic trace, shown in Figure 2, the correlations are higher then the dashed line at lags 60, 120, 180, and 240 seconds ($p = 10^{-4}$), suggesting a period of one minute. Not surprisingly, the email client on the end node checks for new emails every minute.

There are also significant ($p = 0.05$) *negative* autocorrelations just before and after these large peaks. This means that traffic just before and just after a peak is likely to be much less, and resources required for handling the periodic traffic can be freed up immediately after the peak.

In a similar analysis for a protocol without periodic traffic, the PACF did not show any significant periods.

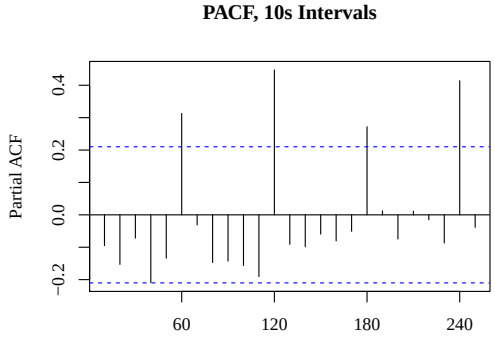


Fig. 2. Partial autocorrelation of IMAP traffic, x is lag, dashed line is significance level 10^{-4}.

4.3. Optimization Goals

Regardless of its type, an algorithm will optimize the mapping for a certain goal. Typical goals are maximizing throughput, minimizing energy, or providing sustainability (e.g., by avoiding mappings that could damage the FPGA over time). For the throughput optimization goal we can distinguish two classes of algorithms.

Maximisation of hardware benefits algorithms put those blocks into hardware that offer the largest benefit of execution in hardware when compared to execution in software. It is assumed that the algorithm has access to information on how long it takes to process a packet of a given size in software and in hardware. For a traffic-independent mapping, the blocks with the largest differences are put in hardware; for a traffic dependent mapping, the differences are multiplied by the number and length of the actual packets.

Protocol graph partitioning algorithms minimize the overhead introduced by transmitting packets between hardware and software. Therefore, they only map *connected subgraphs* of the original protocol graph to hardware. One of those blocks is the block that is physically connected to the network interface. The subgraph mapped to hardware can be selected based on several criteria. For example, we could choose the subgraph with the aggregated largest difference in execution time, or the one that minimizes the load on the hardware/software boundary. Different hardware subgraphs might introduce different loads due to the following reasons:

- one block is a firewall or an intrusion prevention system that drops packets on purpose;
- one block offers routing, sending packets from the receive code path to the transmit code path;
- the packets are dropped due to buffer overflows and hence this subgraph needs more resources.

5. EVALUATION

In order to experiment with different parameters of the underlying hardware we have developed a simulator that can process network traces and execute different algorithms on these traces.

5.1. Simulator

The simulator models the whole system consisting of software and hardware parts. Therefore it needs to simulate real hardware parallelism. We have implemented the simulator in SystemC [7] which is a set of C++ classes and macros that allows the simulation of concurrent processes.

Our simulator models the system described in Section 2, and consists of the following building blocks:

- a CPU, shared among all blocks executed in software;
- several hardware processing units, each capable of hosting a networking block;
- an interconnect between the hardware units, offering line rate communication;
- a communication bus between hardware and software with limited bandwidth;
- a monitoring framework that collects the number of packets and bytes transmitted between the blocks;
- a hardware reconfiguration interface.

The simulator differs from the actual system as follows:

- Instead of receiving real network packets from a physical device, the simulator reads a captured packet trace. For each packet the following information is stored: packet id, arrival time, packet length, protocols. For the packet trace generation we use packet traces captured by libpcap [8] or netsniff [9] that were converted with the help of the pcap decoder provided by yaf [10].
- Since the packets only contain protocol information but not the actual data, the blocks only specify a processing time per packet (to simulate header processing), a processing time per byte (to simulate payload processing), a reply rate (to simulate reliable traffic) and a drop rate (to simulate firewalls etc.).
- The monitoring framework can only obtain packet and byte counters but no physical parameters such as FPGA temperature or energy consumption.
- The reconfiguration overhead is not modelled.

5.2. Results

We have implemented four different mapping algorithms in the simulator that was configured with three hardware slots. To obtain a realistic protocol mix, packet length and packet interarrival times, we captured a packet trace on a notebook in a university network and included all packets that were either broadcast packets or packets addressed to the collecting node into the trace.

We have evaluated the following four algorithms:

- **SW only**: put only those elements in hardware that do not have a software implementation. Put the others into software.
- **Maximum Benefit**: put those elements into hardware that benefit most from a hardware implementation (depending on the number of packets and bytes processed).
- **Minimum Bandwidth**: minimize the number of packets to be sent from hardware to software.
- **Hardware Cluster**: put those blocks in hardware that offer the best hardware benefit and that are connected (in order to avoid sending packets unnecessarily between hardware and software).

We evaluated those algorithms in the following systems:

- **Exp A.** Ethernet is implemented in hardware only, ARP, IPv4, and icmp are implemented in hardware and in software, all the other protocols are implemented in software only. There is no cost associated with sending packets from hardware to software.
- **Exp B.** Additionally, TCP, UDP and tls get a hardware implementation. There is no cost accociated with sending packets from hardware to software.
- **Exp C.** Same as B, but now there is a cost for sending packets between hardware and software, which is proportional to the packet length.

Figure 3 shows one particular hardware/software mapping of the Maximum Benefit algorithm for the Exp B sce-

nario. Nodes represent protocol blocks and edges represent the number of packets sent between the blocks. The colored nodes represent the nodes to be mapped to hardware.

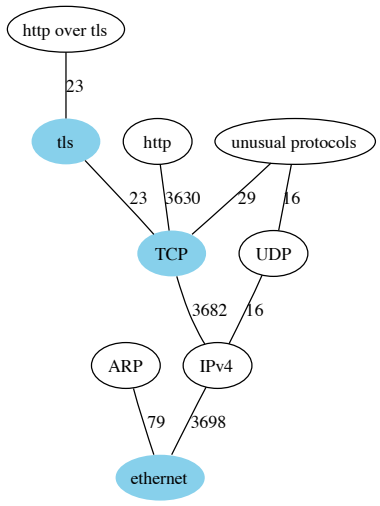


Fig. 3. One particular hardware/software mapping of the Maximum Benefit algorithm for the Exp B scenario.

Table 1 shows the results of the four algorithms in the three different scenarios. However, instead of looking at the individual results, we focus on the *differences* between the algorithms in the different scenarios.

For the SW Only and the Minimum Bandwidth algorithm, the performance does only depend slightly on the scenario. For the Maximum Benefit and the Hardware Cluster algorithm, the performance increases significantly, when more protocols have a hardware implementation. As expected, the Maximum Benefit algorithm offers the best performance. When we introduce a cost for crossing the hardware/software boundary the performance decreases again; however, it decreases less for the Hardware Cluster algorithm so that this algorithm offers now the best performance.

It is also interesting to see that the Hardware Cluster algorithm requires about half the number of reconfigurations than the Maximum Benefit algorithm. This is especially interesting since in the actual hardware, reconfiguration requires time, in which a slot cannot process packets.

From those results we learn that finding the most efficient algorithm heavily depends on the actual parameters and in a real live scenario probably a combination of the different algorithms is required.

6. CONCLUSION AND FUTURE WORK

We have presented an application for self-awareness in reconfigurable computing systems, namely the mapping of network protocols to either hardware or software. To this end an adaptation algorithm is required that takes both network traffic characteristics as well as hardware characteristics into account. We have presented a first classification of such algorithms and developed a simulator that can be used to obtain initial performance results. As a next step we will develop new algorithms for the hardware/software mapping.

Table 1. Performance Comparison

	Exp A	Exp B	Exp C
SW Only Algorithm			
different configurations	1	1	1
reconfigurations	0	0	0
packet drop rate	32.9 %	32.9	33.5%
Maximum Benefit Algorithm			
different configurations	3	9	9
reconfigurations	41	2266	2271
packet drop rate	32.9%	2.7%	12.1%
Minimum Bandwidth Algorithm			
different configurations	2	2	2
reconfigurations	1	1	1
packet drop rate	32.9%	32.9%	33.5%
Hardware Cluster Algorithm			
different configurations	3	4	4
reconfigurations	39	1006	1006
packet drop rate	33.7%	4.8%	11.1%

7. REFERENCES

[1] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May, "The autonomic network architecture (ANA)," *Selected Areas in Communications, IEEE Journal on*, vol. 28, no. 1, pp. 4 –14, Jan. 2010.

[2] E. Lübbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads," *IEEE Int. Conf. on Field Programmable Logic and Applications*, 2007.

[3] A. Keller, B. Plattner, E. Lübbers, M. Platzner, and C. Plessl, "Reconfigurable nodes for future networks," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, dec. 2010, pp. 357 – 361.

[4] K. Thompson, G. J. Miller, and R. Wilder, "Wide-area internet traffic patterns and characteristics," *IEEE Network*, vol. 11, pp. 10–23, 1997.

[5] S. McCreary and kc claffy, "Trends in wide area ip traffic patterns - a view from ames internet exchange," in *ITC Specialist Seminar*, Monterey, CA, Sep 2000.

[6] "MAWI Working Group Traffic Archive," http://mawi.wide.ad.jp/mawi/ (Dec 11).

[7] "SystemC," http://www.systemc.org (Dec 11).

[8] "tcpdump and libpcap," http://www.tcpdump.org/ (Aug 12).

[9] "netsniff-ng," http://www.netsniff-ng.org (Aug 12).

[10] C. M. Inacio and B. Trammell, "Yaf: yet another flowmeter," in *Proceedings of LISA'10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

ON DESIGNING SELF-AWARE RECONFIGURABLE PLATFORMS

K. Siozios[1], H. Sidiropoulos[1], D. Diamantopoulos[1], P. Figuli[2], D. Soudris[1], M. Hübner[3] and J. Becker[2]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[3] Ruhr-University of Bochum, Germany

ABSTRACT

Existing application domains exhibit variations in terms of complexity, performance and power consumption, whereas their efficient implementation onto general-purpose FPGAs is not always a viable solution. In this paper we introduce a framework for designing self-aware reconfigurable platforms. Rather than similar approaches, our solution having a template of architectures, instantiates the most suitable FPGA platform at run-time, depending onto the application's requirements. Experimental results with various applications prove the effectiveness of such an approach, as compared to perform application mapping onto a conventional FPGA architecture.

1. INTRODUCTION

Reconfigurable platforms and more specifically Field-Programmable Gate Arrays (FPGAs) have become the implementation medium for the vast majority of modern digital circuits. This make the FPGA paradigm to grow in importance, as there is a continuous demand towards faster, smaller, cheaper and lower-energy devices.

Apart from the flexibility introduced by reconfigurable architectures, usually such devices exhibit limitations posed by application's inherent properties. For this purpose, FPGA vendors provide a variety of devices for each family, each of which has different properties (e.g. LUT size, number of LUTs per slice, amount of routing resources, etc), whereas the selection of proper FPGA device for a specific application becomes a challenging issue. Additionally, it is common that different applications, even from the same domain, might lead to considerable performance variations whenever they are mapped onto a single device.

In order to alleviate the problem of selecting the most suitable FPGA architecture for a given benchmark, throughout this paper we introduce a novel framework for designing self-aware reconfigurable platforms. The architectural properties of these platforms are tunable and customizable at run-time, depending on the constraints posed by the target application.

Different approaches have been used for determining the most suitable parameters for target FPGA. Among others brute force and simulated annealing are two candidate approaches, which are mostly applied at design time, due to their increased computational complexity. However, throughout this paper, we incorporate an approach with significant lower complexity, which is based on neural network.

A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. Among others, neural networks provide non-linearity, are universal approximatiors (can approximate input–output functions to any desired degree of accuracy, given an adequate computational complexity) and they are adaptable (adjustable synaptic weights and network topology, can adapt to its operating environment and track statistical variations).

In contrast to relevant approaches aiming to perform architecture-level exploration with the usage of neural networks [1] [2], our framework instantiates also the derived (optimal) Virtual FPGA and then performs application implementation onto this platform under the selected design criteria. In order to support these tasks, apart from the introduced neural network, we incorporate also our former Virtual FPGA architectural [3], as well as the CAD flow for application mapping onto the target architecture [4] [8] [9].

The paper is structured in the following manner: Section 2 describes the target architecture, whereas the proposed methodology for supporting the self-aware reconfigurable platforms is discussed in Section 3. Qualitative and quantitative results that prove the effectiveness of introduced solution can be found in Section 4. Finally, conclusions are summarized in Section 5.

2. TARGET ARCHITECTURE PLATFORM

Fig. 1 gives an abstract view of our target platform consisted of multiple heterogeneous instantiations of Virtual FPGAs (V-FPGA), each of which is based on Xilinx Virtex devices. The term heterogeneous refer to the

possibility each of the employed Virtual FPGAs to have different properties. Typical parameters for differentiation are the size are the LUT size, number of LUTs per slice, number of routing channels, etc. By appropriately handling these parameters, it is possible to design either a high-performance, or low-power, FPGA platform.

Apart from the Virtual FPGA the target platform includes also a microprocessor core, a configuration controller, some of-chip memories and busses for on-chip communication and configuration. While the microprocessor is well suited for control oriented tasks and interfacing, the virtual FPGA cores add the advantage of efficient parallel data processing to the system. On-chip communication is realized by AMBA busses. The microprocessor can access the virtual FPGA cores by the AMBA APB bus, since each core acts as an APB slave. The microprocessor also communicates with the configuration controller by the AMBA APB bus. It specifies which configuration the configuration controller should load into a certain virtual FPGA core. All configurations are stored inside an external non-volatile memory. The configuration controller can access the memory controller by the AMBA AHB bus. This requires a bus arbiter as there are two masters on the AMBA AHB, the microprocessor and the configuration controller. Additional details about the architecture of Virtual FPGA can be found in [3].

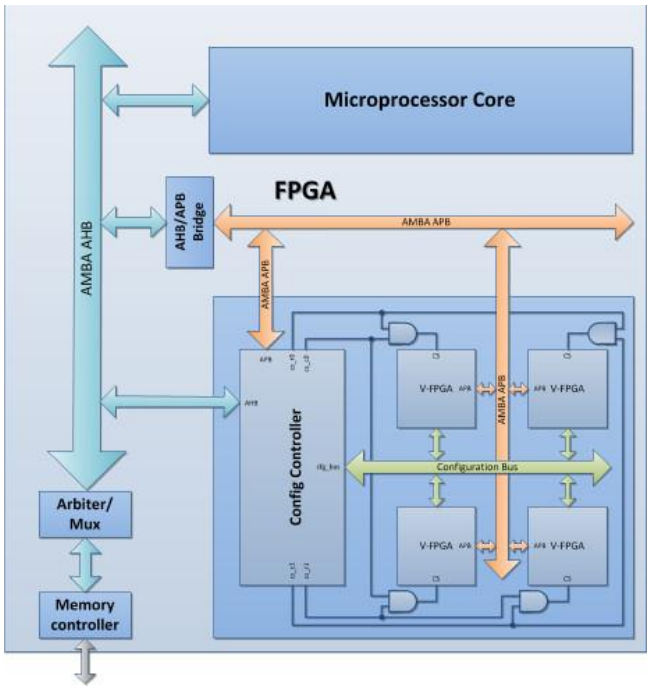


Fig. 1: Schematic view of the system architecture [3].

The platform described in Fig. 1 was developed at generic VHDL and it is mapped onto logic cells within a physical FPGA board. Next section describes in detail the proposed methodology for supporting both the customization of these Virtual FPGAs at run-time based on the application's requirements, as well as the application mapping onto the derived platforms with the usage of a Just-in-Time compilation framework.

3. PROPOSED SOLUTION

The proposed methodology for performing application mapping onto self-aware reconfigurable platforms is depicted in Fig. 2. As input to this framework we use the synthesized application's description.

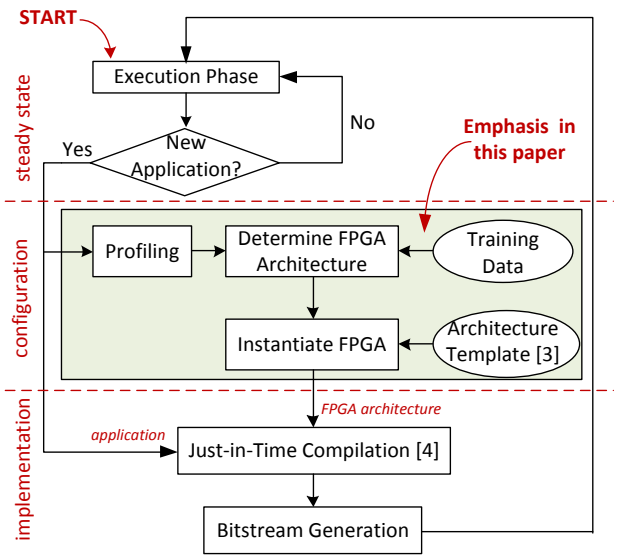


Fig. 2: Proposed methodology for supporting self-aware reconfigurable platforms

Whenever a new application has to be mapped onto the reconfigurable platform, the proper Virtual FPGA is instantiated. In order to derive the architecture's parameters of this device, we profile the application's netlist to extract a number of application-oriented parameters. For the scopes of this paper, the following parameters per benchmark are profiled: (*i*) number of nodes, (*ii*) number of edges, (*iii*) number of primary inputs/outputs, (*iv*) average and maximum fanout.

The profiling results provide a first estimate about the desired architecture for the target FPGA device. Different approaches might be used in order to appropriately fuse these results (e.g. brute force, simulated annealing, genetic algorithms, etc). However, all of them impose mentionable run-time overhead, which is not affordable

especially for systems that have to be executed at run-time.

For this purpose, our methodology incorporates a fast yet accurate multi-objective optimization technique based on neural network. A neural network is a system that learns to map a function from an input vector to an output vector. It consists on a set of simple units which are called artificial neurons. Each neuron has an internal state which depends on its own input vector. From this state the neuron maps an output that is sent to other units through parallel connections. Each connection has a synaptic weight that multiplies the signal travelling through it. So, the final output of the network is a function of the inputs and the synaptic weights of the neural network.

Such a neural network with enough elements, called neurons, can fit any data with arbitrary accuracy. The outcome from neural network is the most suitable architectural parameters in terms of LUT size, number of LUTs per slice, number of routing channels, etc. These parameters are handled by our framework in order to automatically instantiate the desired Virtual FPGA platform. The key differentiation of this solution, as compared to the rest approaches discussed previously, affects the significant lower run-time overhead for deriving architectural parameters of target Virtual FPGA.

A critical task for deriving this optimized architecture affects the proper training of neural network with a representative set of benchmarks. For this purpose, at design-time, we trained the employed neural network with a variety of applications from different benchmark suites (MCNC, LGSynth93, QUIP). Hence, we can almost safely guarantee that the architectural properties derived from our solution are close enough to the optimum device FPGA.

Then, we perform technology mapping, placement and routing (P&R) with the usage of proposed JIT compilation framework. Since a number of additional tasks might be already mapped onto the reconfiguration device, our JIT framework preserves that application's functionalities are mapped onto available (empty) hardware resources. Having as input our framework this information, JIT can perform task implementation (P&R) only with non-utilized resources.

We have to mention that our Virtual FPGA enables such a fine-grain reconfiguration (e.g. routing wires and/or logic blocks inside a single slice). Additionally, it supports a read-back technique for retrieving the current state of configuration data.

The outcome from JIT framework contains all the necessary information in order to compute (*i*) the partial bitstream file for the new task and (*ii*) the resources over the target architecture where this task has to be allocated. Finally, the computed bitstream file configures the Virtual FPGA with the new task. Additional details about how we apply this technique can be found in [3] [6] [7].

The task of customizing Virtual FPGA and generating the properly partial reconfiguration data with the usage of JIT framework is software-supported by an open-source toolset, named 2-D MEANDER [8] [9]. Even though one might expect that running technology mapping and application's P&R will introduce mentionable overheads in execution time and the quality of derived results, we have appropriately tuned these tools in order to improve significantly the run-time overhead without penalties in terms of maximum operation frequency and power consumption, since it is possible to be executed sufficiently even at the embedded processor. Additionally, the JIT framework is expected to introduce the minimum possible fragmentation, since it does not require contiguous area of empty (non-utilized) hardware resources.

4. EXPERIMENTAL RESULTS

For exploration purposes, the employed neural network was modeled in Matlab, whereas after determining the optimal parameters for training (e.g. number of neurons), the network was also developed in C++. This allows integrating the developed network with our MEANDER design framework (depicted in Fig.2).

A critical parameter that affects the efficiency of designed neural network is the regression. Fig. 3 summarizes the metrics of this parameter, as we vary the number of hidden neurons and epochs (determine the maximum number of iterations for training).

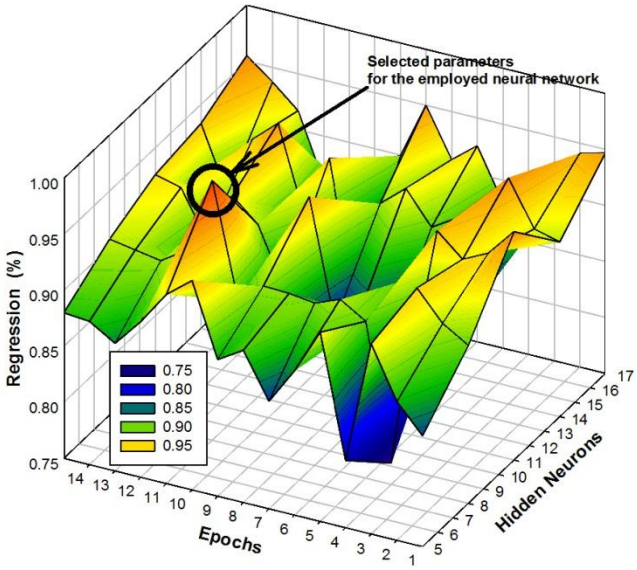


Fig. 3: Exploration results for designing neural network

Based on this graph, we can conclude that mentionable variation in this parameter is possible among the alternative solutions. This also imposes that in order to design an efficient neural network, careful study is required for determining both the number of epochs, as well as the hidden neurons. Specifically, from our exhaustive exploration depicted in Fig. 3, we found that the optimal solution retrieves whenever the number of epochs and hidden neurons are set equal to 12 and 7, respectively.

Another important parameter that quantifies the efficiency of the derived neural network affects its error. This parameter is computed by finding the error between the network's output and the target value over all the example pairs (targets - outputs).

The output of this analysis is summarized in Fig. 4. Specifically, this figure plots the error histogram for the selected neural network. The red color lines denote the optimal solution retrieved after *brute-force* exploration. Regarding our experimental setup, we performed almost 200,000 runs of VPR tool [5] for deriving the results marked with red color.

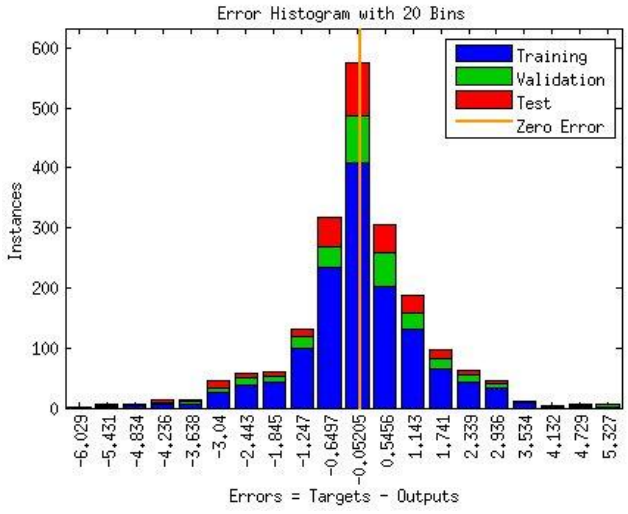


Fig. 4: Error histogram for the employed neural network.

Based on Fig. 4, we can conclude that our proposed network leads to considerable lower execution time compared to the *brute-force* approach, with an almost negligible penalty in selections of architectural parameters.

Note that due to lack of space, the results in this paper affect only the minimization of Power×Delay product. However, apart from this experimental setup, we have already retrieved the corresponding training data for delay, power, area, or any potential combination among them.

5. CONCLUSION

A framework for supporting application mapping onto self-aware reconfigurable platforms was proposed. The introduced solution based on neural networks can achieve sufficient tuning of architectural parameters under delay, power and area metrics.

6. REFERENCES

[1] A. Gomperts, A. Ukil, F. Zurfluh, "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications" , IEEE Trans. on Industrial Informatics, vol. 7, no. 1, pp. 78 – 89.

[2] Z. Marrakchi, H. Mrabet, U. Farooq and H. Mehrez, "FPGA Interconnect Topologies Exploration," International Journal of Reconfigurable Computing, vol. 2009, Article ID 259837, 13 pages, 2009. doi:10.1155/2009/259837.

[3] M. Hubner, et.al., "A Heterogeneous Multicore System on Chip with Run-Time Reconfigurable Virtual FPGA Architecture", Proc. of Reconfigurable Architectures Workshop (RAW), pp. 143-149, May 2011, USA.

[4] H. Sidiropoulos, K. Siozios, P. Figuli, D. Soudris and M. Hubner, "On Supporting Efficient Partial Reconfiguration with Just-In-Time Compilation", Proc. of Reconfigurable Architectures Workshop (RAW), May 2012, China.

[5] V. Betz, J. Rose, and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs", Kluwer Academic Publishers, February 1999.

[6] K. Siozios, D. Soudris and A. Thanailakis, "A Novel Allocation Methodology for Partial and Dynamic Bitstream Generation of FPGA Architectures", Journal of Circuits, Systems, and Computers (JCSC), Vol. 19, No. 3, pp. 701-717, May 2010.

[7] P. Figuli, M. Hübner, R. Girardey, F. Bapp, T. Bruckschlögl, F. Thoma, J. Henkel and J. Becker, "A heterogeneous SoC architecture with embedded virtual FPGA cores and runtime Core Fusion", NASA/ESA Conf. on Adaptive Hardware and Systems (AHS), pp. 96-103, 2011.

[8] Online: http://proteras.microlab.ntua.gr

[9] K. Siozios, K. Tatas, G. Koutroumpezis, D. Soudris, and A. Thanailakis, "An Integrated Framework for Architecture Level Exploration of Reconfigurable Platform", 15th International Conference on Field Programmable Logic and Applications (FPL), pp. 658-661, 2005.

IDENTIFYING OPPORTUNITIES FOR DYNAMIC CIRCUIT SPECIALIZATION

Tom Davidson, Karel Bruneel, Dirk Stroobandt

ELIS CSL, HES group
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
email: [name].[surname]@elis.ugent.be

1. INTRODUCTION

This work describes the identification of designs that benefit from a Dynamic Circuit Specialization (DCS) implementation on FPGAs. In DCS, the circuit is specialized for slowly changing inputs, called parameters. For certain applications or cores, a DCS implementation is faster and smaller than the original implementation. DCS implementations can benefit from the possibility in modern FPGAs to reconfigure specific configuration bits at run-time. This means DCS can be used to specialize an FPGA circuit during the run-time of the FPGA.

Initially, in DCS, it is assumed that the parameters are constant. Constant propagation then allows the specialization of the circuit by partially evaluating the circuit. This specialized circuit is smaller and faster than the original circuit but is only correct for one specific parameter value. Of course, the parameters will not stay constant for ever, eventually they will change. This is solved using the run-time reconfiguration capabilities of the FPGA. A DCS system has both an FPGA and a configuration manager. The configuration manager is responsible for generating the specialized circuit and reconfiguring the FPGA.

In an implemented DCS system, the FPGA contains a specialized circuit for the current parameter value. The FPGA is working normally until the value of a parameter changes. Using the new parameter value, a new specialized circuit is generated by the configuration manager and the FPGA is reconfigured with this new circuit. During this process the FPGA is halted. Once the reconfiguration has finished, the FPGA is working normally again. Until the next parameter change, then the specialization process starts over. The time and resources needed to generate the new circuit and to reconfigure the FPGA are overheads DCS introduces. The time of one specialization is called the *single specialization overhead*.

Such a DCS system could be implemented in multiple ways. In [1] an efficient method for DCS, developed by Ghent University, is described. It includes an FPGA tool flow adapted for DCS, which will be used in the following sections. For the moment, it only implements the reconfiguration of LUT truth tables, and not the routing infrastructure. Only a select number of LUTs are run-time reconfigured, these LUTS are called TLUTs.

Previous papers have shown that this method for DCS can achieve significant area reductions in a number of hardware designs. In [1], an adaptive 16-TAP FIR-filter is implemented using 56% less area. It uses only 1301 LUTs, while the size of the original implementation was 2999 LUTs. The resulting DCS implementation is also 27% faster than the original implementation. The single specialisation overhead is 166μs. This is the design used for the profiler run-time measurements in Section 3. The results for larger FIR-filters are similar. The same publication also shows a 66% LUT reduction for Ternary Content-Addressable Memories. Both of these results, and the adapted FPGA tool flow, were verified by building these DCS implementations on an actual FPGA, the Xilinx Virtex II Pro.

[2] shows that a number of key-based encryption algorithms also see a significant area reduction, 20.6% for AES. 27.8% for Triple DES and a 72.7% reduction for the rc6-algorithm. Finally, in [3] this method for DCS is shown to achieve similar results as hand-crafted run-time reconfigurable implementations of a Network Intrusion Detection System (NIDS). This paper also presents improvements to make the NIDS implementation fully run-time reconfigurable, using this DCS method.

In this paper, we present a profiling tool to aid the designer in analysing the feasibility of a DCS implementation (Section 3). It automatically provides a functional density estimate (see Section 2) for the most interesting DCS implementations.

2. DCS IDENTIFICATION OF A DESIGN

Determining whether or not a certain design will benefit from a DCS implementation is a difficult task for the designer. First, it requires the designer to be familiar with the exact DCS-method. Secondly, in order to find a good parameter selection, insight into the dynamic behaviour of the

signals in the design is required. This is typically not something that is important in the normal design process. In addition, there are no tools that allow the designer to analyse the dynamic behaviour of large groups of signals. VHDL and Verilog Simlators do exist, but they focus on verifying the behaviour of a limited number of signals. Even after the parameter selection it is very difficult to predict the gains of a DCS implementation without actually implementing it. To solve this problem, we first decided on a metric that allows an easier comparison between implementations. This metric is the functional density, and is explained in detail below. The next section presents a profiler that uses this metric to automatically analyse an existing implementation.

To identify designs that benefit from DCS, different implementations of the design have to be compared. A good measure for this comparison is the functional density (FD) [4]. It is the number of computations per unit of area and per unit of time (Equation 1). T_{FPGA} is the complete execution time of the FPGA. N is the number of operations, and A_{FPGA} is the number of LUTs in the implementation.

$$FD_{DCS} = \frac{N}{A_{FPGA} \times T_{FPGA}} \tag{1}$$

The number of operations, N, can always be expressed as the number of clock cycles times a correction factor (C). In the case of the FIR-filter, where one input sample is processed every clock cycle, N can be redefined as exactly the number of clock cycles.

DCS implementations where the benefits are high will occupy less area and therefore have a higher FD. On the other hand, if DCS introduces a large time overhead, the total execution time will increase, while the number of operations stays the same, leading to a lower FD. A design benefits from DCS if a DCS implementation with a higher FD that the original implementation can be found.

In the DCS implementation a number of signals will be selected as parameters. We call this the parameter set. The most exhaustive way to find the best DCS-implementation is to calculate the FD for all possible parameter sets. However, this would take a prohibitively long time, because (i) the number of signals in complex designs is very high and (ii) calculating the FD itself requires up to hours for complex designs.

To address (i), the most interesting parameters for DCS are identified based on Equation 2. This equation expresses the FD as a function of the average single specialization overhead ($\hat{T}_{SST}$) and the average time the FPGA is working for a single parameter value ($\hat{T}_{FPGA}$).

$$FD_{DCS} = \frac{1}{\frac{\hat{T}_{SST}}{\hat{T}_{FPGA}} + 1} \frac{N}{A_{FPGA} \times T_{FPGA}} \tag{2}$$

The first part of Equation 2 expresses the degradation of the FD, caused by the single specialization overhead. It is clear that the degradation will be small if the single specialization time is much smaller than the average time for each parameter value. The influence of the degradation can be seen clearly in Figure 1. This figure shows how the functional density is dependent on the average time between parameter changes ($\hat{T}_{FPGA}$). Looking more closely at the first part of Equation 2, this means only signals for which $\hat{T}_{SST}$ is (much) smaller than $\hat{T}_{FPGA}$ will have a low degradation. In other words, only signals for which the time between transitions ($\hat{T}_{FPGA}$) is much longer that the overhead for a single reconfiguration ($\hat{T}_{SST}$) are interesting parameter candidates. To reduce the number of signals under consideration, signals for which $\hat{T}_{SST}$ is larger than $\hat{T}_{FPGA}$ are ignored. A good value for $\hat{T}_{SST}$ is discussed in Section 3.

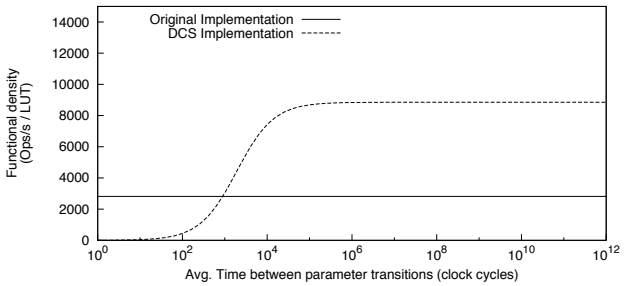


Fig. 1. Influence of the degradation on the functional density of a DCS-implementation. Compared to the functional density of a generic implementation.

As for (ii, the long time needed to calculate the FD), the FD has to be determined for each of the remaining parameter candidates. Calculating the FD exactly requires running the full FPGA tool flow, which can take hours for complex designs [5]. Most time is spent in the last two steps, placement and routing. In addition, both of these steps scale very badly with increasing circuit size, since both are NP-complete problems.

However, the FD can also be estimated, before placement and routing. For this estimation, Equation 3, a rewritten version of Equation 2, is used. Where, $\hat{T}_O$ is the average time between parameter changes in the original implementation. T^1_{DCS} and T^1_O are the clock periods of the DCS and the original implementation, respectively. All these variables can be estimated based on the FPGA tool flow before placement and routing. A more detailed discussion in Section 3.

$$FD_{DCS} = \frac{C}{\frac{\hat{T}_{SST}}{\hat{T}_O \frac{T^1_{DCS}}{T^1_O}} + 1} \frac{1}{A_{FPGA} \times T^1_{DCS}} \tag{3}$$

3. PROFILER

To help the designer analyse the different DCS implementations of designs based on the FD metric, an automatic profiling tool was developed. It requires an RTL description of the design and a test bench with realistic input data. The profiler uses a two-step approach. In the first step a number of parameter candidates are selected from all the signals in the design. In the second step, the functional density is estimated for each of those candidates. Both steps are discussed in more detail below.

Selecting the parmeter candidates: First, the profiler runs the test bench through a simulator to gather data on the dynamic behaviour of all signals. This data is then analysed in order to remove all signals for which $\hat{T}_O$ is smaller than the chosen $\hat{T}_{SST}$. This is the criterion explained in the previous section. However, because the exact overhead of the DCS-implementation is not known yet, a minimal value for $\hat{T}_{SST}$ was chosen. This minimal $\hat{T}_{SST}$ is the time needed to run-time reconfigure one single LUT. This removes all the signals that have a large FD degradation even with the smallest possible overhead. All the remaining signals, the parameter candidates, are given to the next step of the profiler, which will estimate the FD for each candidate.

Functional Density estimate: This FD estimate is based on Equation 3. Here, $\hat{T}_O$ is collected from the analysis of the dynamic signal behaviour in the previous step. All the other variables (A_{FPGA}, $\hat{T}_{SST}$,T^1_{DCS} and T^1_O) are collected by running an abbreviated FPGA tool flow for DCS [1], without placement and routing. This flow is run for each parameter candidate. After this flow has finished, A_{FPGA} is known exactly and $\hat{T}_{SST}$, T^1_{DCS} and T^1_O can be estimated. The details of these estimates are discussed below.

3.1. Single Specialization Time ($\hat{T}_{SST}$)

The single specialization time is the run-time overhead introduced by DCS. It has two parts: the time needed for generating a new circuit ($T_{generation}$) and the time needed for the actual reconfiguration of the FPGA ($T_{reconfiguration}$).

The FPGA tool flow presented in [1] uses Boolean functions to express how the TLUT truth tables are dependent on the parameter values. The new circuit is generated by evaluating these Boolean functions for the new parameter values. This *generation time* is estimated using the number of Boolean operations ($BoolOps$) and the chosen computation unit (K). K represents the average total overhead of one Boolean operation. It is determined by running the complete run-time reconfiguration flow for a large design multiple times, while each time measuring only the time required for the Boolean evaluation.

$$T_{generation} = BoolOps \times K \tag{4}$$

This computation unit is generally also the configuration manager. A good option is the embedded CPU in a lot of modern FPGAs. For the Xilinx Virtex II Pro FPGA, this is the PowerPC 405. In that case, K is 3.32 clock cycles.

The *reconfiguration time* is dependent on the chosen reconfiguration method. [1] proposes two methods, one method using the HWICAP and one using the Shift Register LUT (SRL) capability of Xilinx FPGA's.

The HWICAP is the standard configuration interface provided by Xilinx. In this case, the FPGA is reconfigured frame by frame. To estimate the *reconfiguration time* we estimate number of frames that needs to be reconfigured, assuming the TLUTs are spread out randomly over the total number of LUTs. The reconfiguration time is then Equation 5. For the Virtex II Pro, a single frame is reconfigured in 98.23 μs.

$$T^{HWICAP}_{reconfiguration} = E[\#frames] \times T_{frame} \tag{5}$$

The second method for run-time reconfiguration uses the Shift-Register LUT capabilities, present in some modern FPGAs. This allows the TLUTs to be combined in one or more shift registers chains. This method of reconfiguration is much faster because each chain can be reconfigured in parallel and only the actual truth table bits are sent. A HWICAP frame carries a lot more overhead. The SRL chains are clocked at the design speed. For the above information, the *reconfiguration time* using SRLs can be estimated easily. It uses the number of TLUTS, the number of chains and the clock speed of the DCS implementation (Equation 6).

$$T^{SRL}_{reconfiguration} = \frac{\#TLUTS \times 16 \times T^1_{DCS}}{\#chains} \tag{6}$$

3.2. Clock periods (T^1_{DCS},T^1_O)

The clock periods, T^1_{DCS}, of the DCS implementation, and T^1_O, of the original implementation, are estimated by the number of LUTs in the longest path of the mapping result of each implementation. This depth is then multiplied with a worst-case estimate of one complete LUT delay for the target FPGA. To get the complete clock period estimation a pre and post delay are added. It is assumed that the longest path will be from FF to FF, not from I/O Block to I/O Block. For the Virtex II Pro, the LUT gate delay is 0.275 ns and the LUT net delay is 0.575. The pre and post delays are 0.886 ns and 0.208 ns respectively.

$$T^1_{estimate} = T_{pre} + depth \times T_{LUTdelay} + T_{post} \tag{7}$$

3.3. Profiling time

Using FD estimates instead of exact FD calculations, reduces the execution time of the profiler significantly. As discussed earlier, the most time intensive parts of the FPGA tool flow are the placement and the routing [5]. Both of these steps are avoided by using the estimates. The impact of using the estimates is shown in the execution time measurements described below. These experiments were done for a Virtex II Pro (xc2vp30-7ff1152), the profiler was run on a computer with 8 GBs of RAM and an Intel Core2 Quad Q9650 (3GHz, 1333MHz, 12MB).

Two adaptive FIR filters, a 16 TAP and a 32 TAP version, and corresponding test benches were analysed by the profiler. The profiler was run two times for each FIR filter, one run used the FD estimates, the other the exact FD calculation. In each case the execution time for each parameter candidate was measured. We will discuss the average execution time per parameter candidates for all cases.

Table 1. The average execution time for one parameter candidate

	FD Estimate	Exact FD
16 TAP filter	36.12 s	107.12 s
32 TAP filter	41.67 s	248.10 s

The FD estimate for one parameter candidate in a 16 TAP adaptive FIR filter requires 36.12 seconds. Calculating the FD exactly instead of estimating it requires an extra 71 seconds. In that case, the total analysis time for this filter would increase from 10.23 minutes to 30.35 minutes. This extra time is the time needed for the placement and the routing of the design.

In addition, because the placement is an NP-complete problem, it scales badly with increasing circuit size [5]. E.g. for a 32 TAP adaptive FIR filter, the time needed for the exact calculation of the FD is already 248.10 seconds for each parameter candidate, while estimating the FD still only requires 41.67 seconds, a difference of 206.43 seconds. So, even though the size of the circuit has only doubled, the time for placement and routing has increased by 2.9x. The 32 TAP adaptive FIR filter still only uses 15% of the Virtex II Pro area. This effect will be even more pronounced in designs that use a larger FPGA area.

We are currently preparing an extensive discussion on the accuracy of the FD estimates. However, it is already clear that if the FD estimates predict a significant gain, then the exact, calculated, FDs will also show a significant gain.

4. CONCLUSION

This paper shows how to identify designs that benefit from DCS implementations, using the functional density (FD) as a metric. In addition, a profiler that implements this metric is presented. It automatically analyses the quality of the most interesting DCS implementations of a given design. This allows the designer to determine more easily whether a certain core benefits from a DCS implementation or not. To reduce the execution time of the profiler an FD estimate is used instead of an exact calculation.

This profiler is the first step towards a completely automatic tool flow for DCS. This flow would allow the implementation of DCS systems without any intervention of the designer. Additionally, self-aware reconfigurable systems could also use this automatic flow to analyse their own dynamic behaviour, which could lead to optimal DCS implementations. Off course, this kind of flow would, most probably, require running the full FPGA tool flow at runtime. A process which can take up to hours for complex systems. However, dependent on the practical implementation of these self aware systems, this kind of elaborate self-analysis would only be done very infrequently.

Acknowledgements

This research was supported by both the IWT and the European Commission in the context of the FP7 FASTER project (#287804).

5. REFERENCES

[1] K. Bruneel, "Efficient circuit specialization for dynamic reconfiguration of FPGAs," Ph.D. dissertation, Ghent University, 2011.

[2] T. Davidson, F. Abouelella, K. Bruneel, and D. Stroobandt, "Dynamic circuit specialisation for key-based encryption algorithms and DNA alignment," *International Journal of Reconfigurable Computing*, 2012.

[3] T. Davidson, M. Merlier, K. Bruneel, and D. Stroobandt, "A dynamically reconfigurable pattern matcher for regular expressions on fpga," in *ParCo2011*.

[4] A. M. Dehon, *Reconfigurable architectures for general-purpose computing*. Massachusetts Institute of Technology, 1996.

[5] C. Ababei, "Speeding up FPGA placement via partitioning and multithreading," *International Journal of Reconfigurable Computing*, 2009.

A DYNAMICALLY TUNED FINITE DIFFERENCE METHOD FOR RECONFIGURABLE SYSTEMS

Xinyu Niu and Wayne Luk

Dept. of Computing, School of Engineering
Imperial College London, UK
Email: {nx210, wl}@doc.ic.ac.uk

ABSTRACT

This paper introduces a novel adaptive method applicable to all algorithms based on finite difference method. The algorithm is tuned adaptively and incrementally in terms of data presentation and computation structure. Computational accuracy is dynamically predicted and controlled, and hardware resource consumption is analysed to explore run-time potential of target applications. The design flow involves algorithm update, precision estimation, hardware analysis, runtime scheduling and dynamic learning.

1. INTRODUCTION

Run-time reconfigurability of reconfigurable systems is capable is tuning applications while running. By optimising hardware implementations according to specific requirements during run-time, the performance of reconfigurable systems can be pushed forward. Effective as the technique is, existing algorithms and applications tend to be static, and current runtime reconfiguration methods are limited to applications with varying properties [1, 2].

Bit-level optimisation works as an important design technique in the field of reconfigurable computing. By customising the data presentation to algorithm characteristics and user requirements, significant area saving can be achieved, which in turn improves system concurrency and thus increases system throughput. Existing tools to perform bit-level optimisation are limited to either static precision analyses [3, 4, 5] or Monte-Carlo methods [6]. The concept of run-time reconfiguration is missing from current tools.

In this paper, we aim to introduce an approach to dynamically tune algorithms based on finite difference method. Our approach actively generates runtime design space from target algorithms, and the error propagation is dynamically controlled.

The major contributions of this work include:

- An adaptive approach to control computational error due to reduced data presentation. Instead of passively estimated, the error propagation is actively controlled by tuning the parameters.

- A novel methodology to turn static algorithms into dynamic implementations. The runtime properties of algorithms and reconfigurability of systems are explored to adaptively improve system performance.

2. BACKGROUND

2.1. Precision Analysis

As a FPGA-exclusive optimisation technique, bit-width optimisation has been widely used in the field of custom computing. Various algorithm presentations and precision analysis methods were proposed to generate circuits with guaranteed accuracy [3, 4, 5]. These work depend on passive analyse of the target algorithms, the capacity of tuning data presentation during runtime is not explored. A runtime compensation method was proposed in [6]. However, this method is limited to Monte-Carlo methods, where error is bounded within one path and only the final result matters.

2.2. Runtime Reconfiguration

Runtime reconfiguration is an emerging area to improve system performance during runtime. Given runtime information is properly utilised, the implemented operators can be further optimised during specific time slots. The slowly varying properties of input data were captured in [1] to implement arithmetic operators with constant input. The algorithm parameters were dynamically approximated to optimal constants for the operators in [2], to further reduce the upper bound of implemented operators. Besides customising implementations, the tuning process also impacts the computational precision. However, the interactions have not been explored.

2.3. Finite Difference Method

Finite difference method is a widely used numerical method to approximate solutions to differential equations. The approximation error depends on the step size and the approximation order.

$$\frac{\partial u^2(a)}{\partial x^2} \approx \frac{\alpha \cdot u(a-x) + \beta \cdot u(a) + \gamma \cdot u(a+x)}{x^2} \quad (1)$$

This static approximation process can be dynamically accomplished, with coefficients actively varied. The potential benefits include reduced computational effort, as well as opportunity to dynamically tune the algorithm.

3. MOTIVATION

The run-time potential of application depends the varieties of the application in time dimension. Previous work are limited to applications [7, 8, 9, 10] with varying properties. The proposed approach exploit the run-time potential of applications by actively tuning application configurations. Our aim is to show that, with proper run-time design methods, applications with static properties can explore reconfigurability to improve system

The explored properties in current approach include data presentation and constant coefficients. The data presentation involves achieving optimal bit-width optimisation for arithmetic operations, while varied coefficients impact resource usage and error propagation. The interaction between the updated properties and system performance is formulated to incrementally tune the target applications.

In reconfigurable computing, data presentation refers to bit-width optimisation. The data are presented in customised formats to reduce resource consumption. As a consequence, generated results differ from the results of original presentation. If the original results are assumed as accurate, inaccuracy is introduced at the time when data presentation is varied, and propagates through the computational space, as shown in Figure 1. By dynamically introducing and compensating variable inaccuracy in different time slots, an optimal run-time data presentation can be achieved.

Besides data presentation, algorithm optimisation during run-time involves reconstructing the target algorithms according to dynamic requirements of applications. Figure 2 demonstrates the structure of a one-dimensional finite difference method. Propagation of generated results between time steps can be dynamically controlled, by varying the mapping constant. For algorithms based on finite difference method, coefficients are decided by approximation order O, stencil size S, step size in time dt and step size in space ds. The computational process for a one-dimensional finite difference method is shown in Figure 2.

$$(\alpha, \beta, \gamma ...) \Leftarrow f(O, S, dt, ds) \quad (2)$$

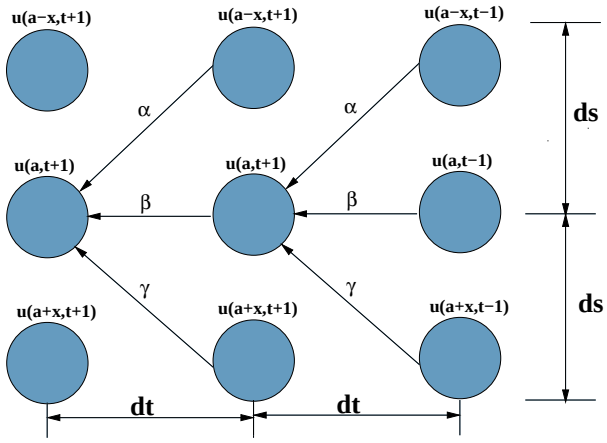


Fig. 2. Structure of one-dimensional finite difference method in time.

With dynamic data presentation and algorithm structures in different time steps, the target algorithms can be dynamically tuned.

4. ADAPTIVE APPROACH

The proposed approach is presented in Figure 3. It works as an iterative method to adapt the target algorithm into dynamically optimised operators. The algorithm is constantly tuned as computation goes through involved grid space. Tuned coefficients are fed into a precision estimator to predict accumulated errors. The precision model analyses accuracy of the specific data presentation and constant values, while the hardware model calculates resource usage of the configuration. Two analytical models cooperate to estimate the benefit of possible reconfiguration opportunities. Target accuracy and available hardware resource work as constraints for possible configurations.

The error for a specific point accumulates from neighbouring points in space and previous calculations in time. Affine Arithmetic [11] (AA) was proposed to estimate the computation range and precision. It can be used here to estimate the dynamic precision.

$$x_{re} = x + 2^{-MA-1} \cdot \lambda, \lambda \in [-1, 1] \quad (3)$$

$$E_i = 2^{-MA_i-1} \cdot \lambda_i \quad (4)$$

$$E_{j,t} = \sum_{i=0}^{n} \lambda_{<i,t-1>} \cdot E_{<i,t-1>} \quad (5)$$

where x_{re} denotes the data presented with reduced precision, and MA is the mantissa size. Computation error introduced in a specific time step is expressed as E_i, while the error propagation is formulated as Eq 5. E_i is estimated with data presentation, and the propagation is controlled with dynamic constants. Therefore, the impacts of varied algorithm

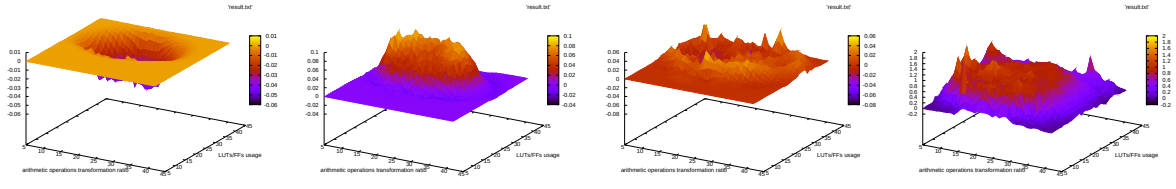


Fig. 1. Error propagation in computation space and time.

on computation accuracy can be adaptively predicted.

The hardware analytical model is built to capture the dynamic optimisation opportunities as design configurations are mapped onto reconfigurable fabrics. Bit-width optimisation impacts resource usage of arithmetic operations linearly. In the meanwhile, the varied constants construct the upper bound of resource usage. As resource consumption goes down with data presentation, system performance can be increased after a reconfiguration operation. For a reconfigurable area with upper bound resource consumption, multiple time steps can be mapped into it with reduced data presentation. As shown in Figure 2, the two time steps can be accomplished with data streamed one time, given the available resource can accommodate the dynamically optimised circuits.

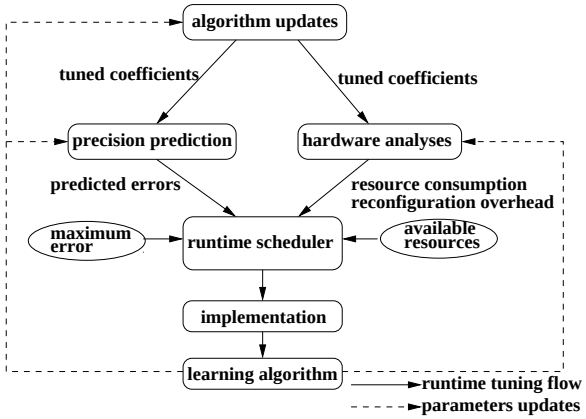


Fig. 3. Tuning process of the proposed approach.

Analysed data are fed into a runtime scheduler to decide whether the current configuration needs to be reconfigured. For the implemented circuits, data are sampled back into a learning algorithm to update module parameters, closing the tuning process.

5. CONCLUSION

In this paper, we present an adaptive approach to explore runtime properties of algorithms based on finite difference method. The computational error is dynamically controlled and the arithmetic operators are adaptively optimised. Work in progress and future work include expanding modules in the turning process, exploring more runtime properties and building various applications to evaluate the proposed approach.

6. ACKNOWLEDGEMENT

This work was supported in part by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 248976, 257906 and 287804, by the HiPEAC NoE, by Maxeler University Program, and by Xilinx.

7. REFERENCES

[1] K. Bruneel, F. Abouelella, and D. Stroobandt, “Automatically mapping applications to a self-reconfiguring platform,” in *DATE*, 2009, pp. 964–969.

[2] Q. Jin, T. Becker, W. Luk, and D. B. Thomas, “Optimising explicit finite difference option pricing for dynamic constant reconfiguration,” in *FPL*, 2012.

[3] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong, “Bit-level optimization for high-level synthesis and fpga-based acceleration,” in *FPGA*, 2010, pp. 59–68.

[4] D.-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, “Accuracy-guaranteed bit-width optimization,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, 2006.

[5] S. Bayliss and G. A. Constantinides, “Optimizing sdram bandwidth for custom fpga loop accelerators,” in *FPGA*, 2012, pp. 195–204.

[6] G. C. T. Chow, A. H. T. Tse, Q. Jin, W. Luk, P. H. W. Leong, and D. B. Thomas, “A mixed precision monte carlo methodology for reconfigurable accelerator systems,” in *FPGA*, 2012, pp. 57–66.

[7] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *Proc. FPL*, 2006.

[8] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Proc. DATE*, 2009.

[9] F. Nava *et al.*, "Applying dynamic reconfiguration in the mobile robotics domain: A case study on computer vision algorithms," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 4, no. 29, 2010.

[10] D. Koch and J. Torresen, "FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proc. FPGA*, 2011.

[11] J. Stolfi and L. Figueiredo, "Self-validated numerical methods and applications," *Brazilian Mathematics Colloquium Monograph*, 1997.

DYNAMIC DATA-PATH SELF-RECONFIGURATION OF A VLIW-SIMD SOFT-PROCESSOR ARCHITECTURE

Guillermo Payá-Vayá, Roman Burg, and Holger Blume

Institute of Microelectronic Systems, Leibniz Universität Hannover
Appelstr. 4, 30167, Hannover, Germany,
emails: {guipava,burg,blume}@ims.uni-hannover.de

ABSTRACT

Dynamic Partial Reconfiguration (DPR) can be used on Xilinx FPGA devices (by using the internal ICAP unit) to increase the processing performance during run-time and to reduce the hardware cost requirements of a specific system. This paper presents a dynamically self-reconfigurable VLIW-SIMD soft-processor architecture, that takes advantage of the fast reconfiguration of small functional units (FU) instead of hardware accelerators. The proposed soft-processor implements a DMA-ICAP controller to control the dynamic reconfiguration process of its reconfigurable FUs. The implementation of this DMA-ICAP controller together with the software program control are the main topic of this paper. For video-based processing algorithms running under real-time conditions (i.e., 30fps), specialized FUs with equivalent hardware cost of a typical 64-bit SIMD-arithmetic unit can sequentially be reconfigured up to 720 times per frame. Therefore, the use of reconfigurable specialized FUs allows a fine-grained acceleration of intra-frame processing tasks.

1. INTRODUCTION

Over the past years, embedded systems have expanded to cover a wide variety of applications, ranging from portable multimedia devices to sensor networks and medical imaging systems. Stringent computing performance and power-consumption requirements in combination with the increasing demand for low cost and time-to-market products make the research field of embedded systems challenging.

In order to meet the mentioned design goals, specialization is required. This can be done by inserting special instructions to a processor or by implementing dedicated hardware macros. One implementation form, which is frequently used, is an application specific integrated circuit (ASIC). Unfortunately, the functionality, flexibility and processing performance of the resulting ASIC remain fixed after the production. This means that in order to adapt these architectures to future changes, the architecture eventually must be physically re-implemented, resulting in long time-to-market and high production costs.

Instead of implementing ”static” ASICs, the TUKUTURI project at the Institute of Microelectronic Systems studies the use of reconfigurable devices to avoid the necessity of completely re-implementing the embedded multimedia processing system physically every time a new enhanced functionality is required. Therefore, dynamic partial reconfiguration (DPR) [1] is used. DPR is provided by commercial Xilinx FPGA devices and can be used to specialize the instruction-set and some parts of the architecture to efficiently execute any kind of multimedia tasks. The time required to reconfigure a specific region of an FPGA directly depends on the size of this region and highly influences the reachable processing performance. For example, in video-based processing algorithms like those used in driver assistance systems [2] (depending on the required reconfiguration time) DPR can be used during inter- or/and intra-frame processing. A usual real-time constraint in this kind of systems is to process at least 25 frames per second (fps), i.e., one frame in less than 40 ms.

Most of the publications, concerning DPR, proposed architectures based on reconfigurable hardware accelerators attached to an OCP bus, which can only be used for inter-frame processing acceleration [1]. Only the hardware demonstrator presented in [2] can perform at least two reconfigurations of sequentially working hardware accelerators within 40 ms, allowing intra-frame processing. In order to efficiently accelerate intra-frame processing, the size of the hardware accelerators should be reduced to enable a speed-up of the reconfiguration process.

This paper presents a reconfigurable VLIW-SIMD soft-processor architecture that can dynamically self-reconfigure the data-path instead of hardware accelerators, performing fine-grain acceleration at the application program code execution level. Section 2 introduces the generic VLIW-SIMD soft-processor architecture, called TUKUTURI. The DMA-ICAP controller used for dynamic partial reconfiguration is described in Section 3. In Section 4, an evaluation of the time required to reconfigure different examples of functional units is presented. Finally, conclusions are presented in Section 5.

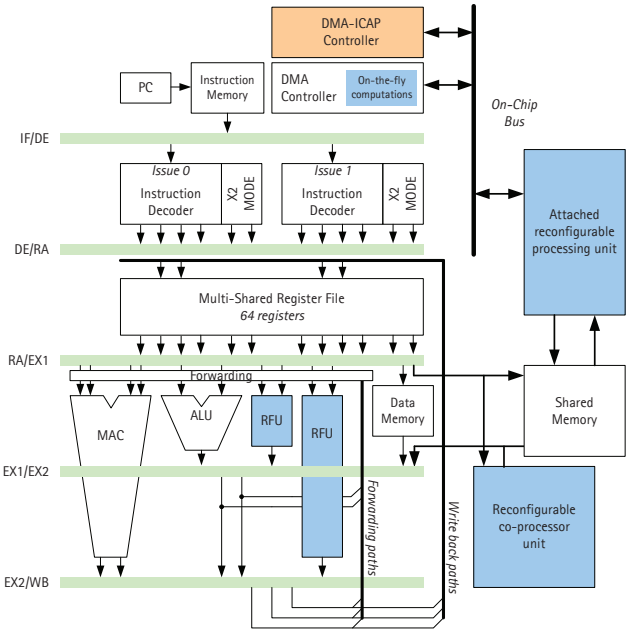


Fig. 1. Simplified pipeline scheme of the generic SIMD-VLIW processor architecture (TUKUTURI). The blue shaded areas identify the proposed reconfigurable hardware elements.

2. A GENERIC VLIW-SIMD SOFT-PROCESSOR ARCHITECTURE

In previous works carried out at the Institute of Microelectronic Systems in the research project RAPANUI [3], a comprehensive analysis of the architectural design alternatives of application-specific VLIW-SIMD processors for multimedia applications was performed. For that, a comprehensive design space exploration environment based on a generic VLIW-SIMD architecture template was implemented. This environment includes a configurable pipeline architecture simulator, an enhanced assembly code compiler, and a parameterized VHDL implementation of the architecture template. By using this environment, the architecture template can be optimized in terms of performance and hardware cost for a set of multimedia applications. The RAPANUI project has demonstrated that the combination of new enhanced hardware architecture mechanisms and the corresponding assembly code compiler improvements plays an important role to overcome the architectural bottlenecks.

In this work, a new soft-processor architecture, called TUKUTURI (see Figure 1), is proposed based on the mentioned VLIW-SIMD template. The novelty lays in the use of a commercial FPGA device as the destination target platform. The generic VLIW-SIMD processor template was specially designed for efficient processing of macroblocks, commonly used in video coding algorithms, and comprises a flexible data-path controlled by a dual issue-slot VLIW. The control-path is initially divided into 5 basic pipeline stages, but the number of execution stages can be modified to increase the operation clock frequency, since the critical path is located in these stages (when implementing the VLIW-SIMD processor design on a Virtex-5 FPGA). All the SIMD-FUs internally work with 64-bit wide operands, which can be split up to perform the same operation with different data sizes. Finally, it is worth mentioning that the implementation of a basic TUKUTURI configuration based on 5 pipeline stages on a Virtex-5 FPGA can reach more than 100 MHz.

A two-level (re-)configuration strategy is implemented to efficiently use the FPGA resources. First, the TUKUTURI architecture can be optimized to efficiently process a particular application, taking into account the optimal size and number of reconfigurable FUs and reconfigurable co-processor units. Therefore, every time this application is about to be executed, a ***static reconfiguration process*** is performed, programming the whole or a part of the FPGA device before starting the application execution. This level of reconfiguration allows to reuse the system for future applications with totally different processing characteristics, allowing to adapt the complete TUKUTURI architecture to a new application. This mechanism introduces a significant time penalty during run-time, because of the time required to reconfigure those FPGA partitions or the whole FPGA (e.g., a VIRTEX-5 LX330 requires up to 26 ms to be completely reconfigured) and also because of the static reconfiguration (i.e., no computation can be performed on the FPGA during the reconfiguration process). However, the optimization of the TUKUTUTI soft-processor on the architectural level (e.g., number of RFU or static FUs, ...) can significantly increase the processing performance.

Second, ***dynamic partial reconfiguration*** is also supported to insert in run-time new complex instructions or co-processors in the architecture. The proposed RFUs are highly coupled to the pipeline structure of the TUKUTURI architecture, benefiting from the data forwarding-path. The effective use of the forwarding mechanism is crucial for increasing the reachable processing performance. Dynamic reconfiguration will also be supported in other parts of the TUKUTURI architecture, such as the DMA controller and the register file structure. For example, it is planed that the DMA controller supports the use of simple data transformations that will be performed on-the-fly while accessing data from the external memory, e.g. a realignment process proposed in [4].

3. THE DMA-ICAP CONTROLLER

DPR is supported by the TUKUTURI processor by means of RFUs and (if required) reconfigurable co-processor units.

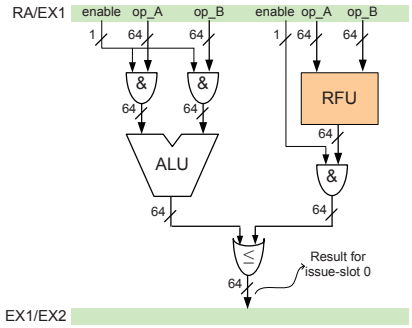


Fig. 2. Simplified scheme of a part of the TUKUTURI execution stage.

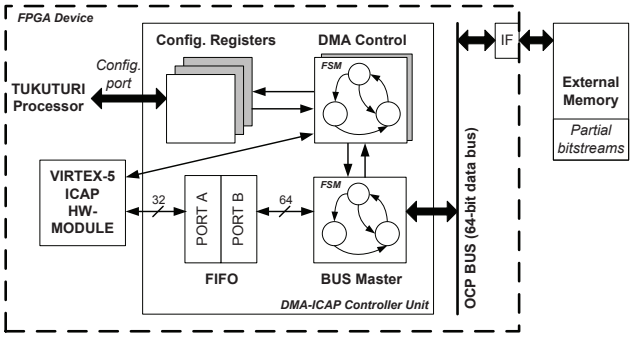


Fig. 3. Block diagram of the DMA-ICAP controller interconnected with the TUKUTURI processor and an external memory, where the partial bitstreams are stored.

The RFU coupling mechanism is tighter than the other mechanisms based on hardware accelerators. Moreover, it does not only implement a direct connection between the RFUs and the register file but also supports data forwarding. During the reconfiguration process, the RFU being reconfigured can not be used by the TUKUTURI processor and the outputs of this unit are not propagated, as usual in any pipeline processor. This mechanism is sufficient to isolate the RFU from the rest of the processor architecture.

In Figure 2, the technique used to isolate the RFU is shown. The main concept is that each FU or RFU should generate a zero output if not used. On the one hand, an AND-gate logic is used in non-reconfigurable FUs to produce a zero result in case this FU is not used (i.e., enable signal is zero). This technique was used before in ASIC implementation to reduce the switching activity of the FUs and, therefore, the dynamic power consumption. In FPGA implementation, during the logic synthesis, these AND-gates are distributed inside the FU logic with a marginal increase of the hardware resources. On the other hand, an AND-gate logic is used to control the results of the RFUs. In contrast to non-reconfigurable FUs, this logic will not be distributed inside the RFU logic during the FPGA logic synthesis and it is used to isolate the RFU during its reconfiguration. Finally, an OR-gate logic is used to generate the result operand for each issue-slot.

It is worth mentioning that an efficient FPGA implementation of the SIMD-FUs is crucial for reducing the hardware resources requirements and, therefore, decreases the number of cycles required for reconfiguring the unit [5].

DPR is performed by using the internal ICAP [6] module available in the Xilinx Virtex FPGA family. The TUKUTURI architecture implements a DMA-ICAP concept (see Figure 3) that allows the processor core to read the information required for reconfiguring the FPGA device (i.e., partial bitstreams) from an external memory. The transmission of the partial bitstream and the following reconfiguration process is initiated by the TUKUTURI processor by programming the configuration registers (see Table 1).

Figure 4 shows an example of an assembler code that indicates how the partial reconfiguration works. This assembler code is executed on the TUKUTURI processor to program the DMA-ICAP controller, which performs the reconfiguration of a RFU during the execution of a subroutine that does not use this RFU.

The assembler code can be summarized in the following points: In lines 2 and 3, two *STOREIL* are used to configure the DMA-ICAP. In this example, the partial bitstream is 0x480C bytes large, as written in BLOCK_SIZE. Then, the address position where the partial bitstream is located (in the external memory) is stored in EXT_ADDR. This STOREIL initiates automatically a block-transfer of the indicated partial bitstream to the ICAP unit. In line 6, a subroutine call and return is performed. This subroutine is located in *L_SUBROUTINE_A*. Register *V0R31* is used to return back to this program position after the subroutine execution. Finally, in lines 9 to 14, a WAIT loop is used to check if the DMA-ICAP controller has finished the reconfiguration process (i.e., the partial bitstream transfer). For that, the STATUS_FLAG is read and compared with the immediate value 0x1. These two operations will be executed until the result of the comparison is zero. The STATUS_FLAG register is automatically initialized to 0 after programming the DMA-ICAP controller (i.e., writing in EXT_ADDR), and then it turns to 1 when the partial bitstream has been transfered completely (i.e, BLOCK_SIZE bytes were transfered).

It is desired that the reconfiguration process is performed in background by the DMA-ICAP controller, allowing the processor core to execute other tasks simultaneously. In Figure 4, *L_SUBROUTINE_A* is executed at the same time that the DMA-ICAP controller performs the reconfiguration. Therefore, it is the task of the software engineer to optimally schedule the reconfiguration process of a new RFU together with the execution of a RFU-independent subroutines.

The DMA-ICAP controller also implements internally a dual true port FIFO with different data width ports (a 32-bit data port connected to the ICAP module and a 64-bit data

Table 1. DMA-ICAP Configuration Registers.

Parameter	Description
EXT_ADDR	External memory base address where the partial bitstream is located
BLOCK_SIZE	Size in bytes of the partial bitstream
STATUS_FLAG	Flag that indicates if the DMA-ICAP is still performing the transfer

```
1  // Reconfigure RFU1 by programming the DMA-ICAP
2  STOREIL   0x211, #0x480C  // Write to BLOCK_SIZE
3  STOREIL   0x210, #0x600000 // Write to EXT_ADDR
4
5  // Execute a subroutine that does not use RFU1
6  JLR      V0R31, L_SUBROUTINE_A
7
8  // Check Reconfiguration before using RFU1
9  :L_WAIT
10 LOAD      V0R0, 0x212       // Read from STATUS_FLAG
11 SUBICS_8  V0R1,V0R0,#0x1    // V0R1=V0R0-0x1
12                             // store status flags
13                             // (e.g. zero)
14 BSR       L_WAIT, #0b00000001, #COND_ZERO
15                             // Jump to L_WAIT
16                             // if subword 0 was zero
```

Fig. 4. Example of an assembler code that uses the DMA-ICAP to load the RFU1 and checks if the reconfiguration process has finished.

port connected to the OCP bus via a BUS master). This configuration uses the FIFO as a buffer, allowing to reach the maximum available reconfiguration speed even if the external memory is partially busy. It is worth mentioning that the overall system works with more than 100 MHz. Therefore, for this system clock frequency, the DMA-ICAP module, that also works with 100 MHz, only requires one 64-bit data every two system cycles (i.e., one 32-bit data every system cycle) to feed the ICAP module continuously with 32-bit data from the partial bitstream, reaching the maximum bandwidth of 400 MB/s.

4. EVALUATION

In case of processing a video sequence (e.g., for video-based driver assistance systems or video coding purposes) under real-time conditions (i.e., 30 fps), the TUKUTURI architecture can sequentially reconfigure a typical 64-bit SIMD arithmetic unit up to 720 times per frame (see Table 2). Moreover, more complex units, such as the disparity map unit presented in [7], can be reconfigured up to 98 times per frame, also allowing the use of DPR in intra-frame video processing. This FU computes the position of the minimum 8-bit value on two vector registers, each one with 32 values. However, there is a technological limitation on Xilinx FPGA devices. For example, on Virtex-5 FPGA devices, only one reconfiguration can be performed simultaneously, although these devices have two ICAP units.

Table 2. Time required to reconfigure different representative SIMD-Functional Units on a Xilinx Virtex-5 LX330.

SIMD Unit	LUTs[1]	RB[2]	Cycles[3]	RPF[4]
ClipMaxMin	170	2	~ 3080	~ 1082
Arithmetic	293	3	~ 4620	~ 721
Shift-and-Round	1504	12	~ 17920	~ 186
Disparity-Map[5]	2835	22	~ 33870	~ 98

[1]Look-up-Tables. [2]Virtex-5 Reconfigurable Blocks. [3]Number of cycles required to reconfigure each SIMD-FU. These cycles are measured on a CHIPit emulation system using the DMA-ICAP and the TUKUTURI soft-processor at 100MHz. [4]Number of times that each unit can be reconfigured per frame for a 30 frame per second system constraint. [5]Implementation of a complex SIMD disparity map unit presented in [7] as *DISP_X4*.

5. CONCLUSION

In dynamically reconfigurable soft-processors for video signal processing, the size of the reconfigurable modules constraints the maximum number of reconfigurations per frame while processing a frame under real-time conditions (e.g., 30 fps). The evaluation presented in this paper shows that the use of specialized reconfiguration functional units can be used for speeding-up intra-frame processing due to their fast reconfiguration.

6. REFERENCES

[1] K. Papadimitriou, *et al.*, "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 36:1–36:24, Dec. 2011.

[2] C. Claus, *et al.*, "Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems," in *Reconfigurable Computing: Architectures, Tools and Applications*, 2010, vol. 5992, pp. 55–67.

[3] G. Payá-Vayá, "Design and Analysis of a Generic VLIW Processor for Multimedia Applications," Ph.D. dissertation, Leibniz Universität Hannover, Feb 2011.

[4] G. Payá-Vayá, *et al.*, "An Enhanced DMA Controller in SIMD Processors for Video Applications," in *Proc. of the 22nd Int. Conf. on Architecture of Computing Systems (ARCS09)*, vol. 5455, 2009, pp. 159–170.

[5] S. Nolting, *et al.*, "Optimizing VLIW-SIMD Processor Architectures for FPGA Implementation," in *ICT.OPEN 2011 Conference*, November 2011.

[6] *Virtex-5 FPGA Configuration User Guide*, Ds202 (v5.3) ed., Xilinix Inc., May 2010.

[7] G. Payá-Vayá, *et al.*, "VLIW Architecture Optimization for an Efficent Computation of Stereoscopic Video Applications," in *Proc. of the 1st Int. Conf. on Green Circuits and Systems (ICGCS2010)*, Jun. 2010, pp. 457–462.

COMPUTING SYSTEMS BASED ON ADAPTIVE RECONFIGURABLE ARCHITECTURES

Andrea Cazzaniga, Filippo Sironi, Davide B. Bartolini, Marco D. Santambrogio

Dipartimento di Elettronica e Informazione, Politecnico di Milano

{acazzaniga,sironi,bartolini,santambrogio}@elet.polimi.it

ABSTRACT

Imagine further a computing system that performs better according to a user's preferred goal the longer it runs an application. Such an architecture will enable, for example, a hand-held radio or a cell phone that can run cooler the longer the connection time. Moreover, Systems on a Chip (SoC) can draw various benefits, such as adaptability and efficient acceleration of compute-intensive tasks from the inclusion of reconfigurable hardware as a system component. Dynamic reconfiguration capabilities of current reconfigurable devices create an additional dimension in the temporal domain. During the design space exploration phase, overheads associated with reconfiguration and hardware/software interfacing need to be evaluated carefully in order to harvest the full potential of dynamic reconfiguration. Self-aware computer systems will be capable of adapting their behavior and resources thousands of times a second to automatically find the best way to accomplish a given goal despite changing environmental conditions and demands. In this work we present an attempt in presenting the key enabling technologies to realize such self-aware runtime system that can gain benefits from the presented paradigm.

1. INTRODUCTION

Reconfiguration capabilities and hardware-software codesign techniques are becoming just elements of a more complex scenario. The need for a systematic approach to the design of new architectures and systems enabling self-awareness is motivated by some trends that have gained momentum in the past few years. Research is pushing forward, looking for complex heterogeneous, reconfigurable multi-core architectures. In order to overcome the limits deriving by the increasing complexity and the associated workload to maintain such complex infrastructure, one possibility is to adopt self-adaptive and autonomic computing systems [1]. A self-adaptive and autonomic computing system is a system able to configure, heal, optimize and protect itself without the need for human intervention. Different companies, i.e., IBM [2, 3], Oracle [4], and Intel [5] have invested in this research, creating several products characterized by a self-adaptive behavior. However, a lot of work still needs to be performed in defining effective self-adaptive and autonomic architectures in the embedded system domain.

On one hand there is the increasing importance of non-functional constraints: in the *perceived value* of a digital system, features that are not completely reducible to the functionalities are getting ever more important. Two famous examples of such non-functional constraints are power consumption and reliability, but there are many other potential dimensions, that lie at the border of what can be called functionality, that impact user experience of a digital system or device; examples can be results accuracy, like in different audio and video qualities for a multimedia device, or efficiency in understanding human signals in interactions (as already happens, for instance, in speech recognition software)). Meeting such constraints (or optimizing the associated figures) is getting more and more difficult, mainly because of the exponential increase of environmental interactions and conditions in which devices are required to operate.

On the other hand, devices structure evolution tends towards forms of complexity characterized by the increase in number and of complexness of interacting "peer" elements, at various levels (e.g.: cores on a multicore processor, concurrent programs in a multitask operating system, number of threads within a single application). Meeting non functional constraints requires, most of the times, a coordination among all those elements, for any possible working condition. It is evident that statically foreseeing, at design time, the actions that must be taken in order to maximize non-functional constraint satisfaction for all the possible scenarios is already way beyond feasibility. Think of the simplest problem that control engineering faces since a long time: controlling the temperature of a room to stay stable at a given value, within acceptable bounds. Room temperature can be determined or influenced by a plethora of different factors: outside weather, windows being open or closed, the presence of persons inside the room and so on. Knowing all such factors in advance is of course impossible. The conceptual solution developed was the closed loop control: the system reacts to deviations from the goal (differences between the temperature set and that measured) with actions somehow proportional to that distance (injecting thermal power

in the room).

The user of the control system just sets the goal, then the system dynamically and automatically reacts, adapting itself to the new conditions. This control task example can be used as a metaphor for the motivations towards implementation of the self-aware adaptive systems that are the focus of this project: as the temperature controller exploits information on its state and on the environment to pursue a goal that is dependent on a set of factors non foreseeable at design time, so should be able to do, on a much higher, behavioral level, embedded systems.

2. CONTEXT DEFINITION

Resources such as quantities of transistors and memory, the level of integration and the speed of components have increased dramatically over the years. Even though the technologies have improved, we continue to apply outdated approaches to our use of these resources. Within this context, imagine an interaction capability of digital systems by which designers and users can specify their desired goals rather than how to perform a task, along with constraints in terms of an energy budget, time, or simply a preference for an approximate answer over an exact answer. Imagine further a computing chip that performs better according to a user's preferred goal the longer it runs an application. Such an architecture will enable, for example, a handheld radio or a cell phone that can run cooler the longer the connection time. Or, a system that can perform reliably and continuously in a range of environments by tolerating hard and transient failures through self healing. Self-aware computer systems will be capable of adapting their behavior and resources thousands of times a second to automatically find the best way to accomplish a given goal despite changing environmental conditions and demands. Such a capability would benefit a broad spectrum of computer systems from embedded systems to supercomputers and is particularly useful for meeting power, performance, and resource-metering challenges in mobile computing [6, 7], grid and cloud computing [8, 9, 10], multicore computing [11, 12, 13], networks [14, 15], self-healing systems [16, 17, 18], complex distributed Internet services [19, 20, 21], distributed system [22], operating systems [23, 24, 25, 3, 26], and adaptive and dynamic compilation environments [27, 28].

3. RUNTIME SELF-AWARE SUPPORT

The operating system is in charge of choosing at runtime between the set of possible implementations (a software one or one of the available hardware implementations) according to different criteria, such as the available area (set of resources) on the FPGA, input data type and dimension, functionalities already implemented and available as hardware components. The runtime decision of the most suitable implementation (software or reconfigurable hardware) due to runtime conditions, allows this work to be considered as an attempt to the define a *self-aware computing system*. The operating system answers a request for a functionality by choosing a runtime the best implementation. *Best* does not mean the optimal solution but the one that can guarantee the best performance considering all the runtime conditions in which it has to be executed. Considering the scenario where an hardware solution is chosen as the best implementation, the corresponding hardware module has to be loaded by configuring the IP-Core on the FPGA and by creating a communication channel between the module and the software application in a transparent way. As a consequence, the IP-Core becomes accessible from the userspace when the control is returned to the user application. The online adaptability of the overall system is implemented in the OS by means of kernel modules implementing a closed control loop, called *Self-Aware Support* in Figure 1, and an Adaptive library. The *Self-Aware support* kernel extension, located between the userspace and the physical architecture, performs the online adaptation of the system, providing a common interface for software applications and hardware developers. Each software application communicates with the kernel using the API of the reconfiguration library, which allows also the access to the hardware component that physically implements specific functionalities, once they have been configured on the FPGA by the operating system.

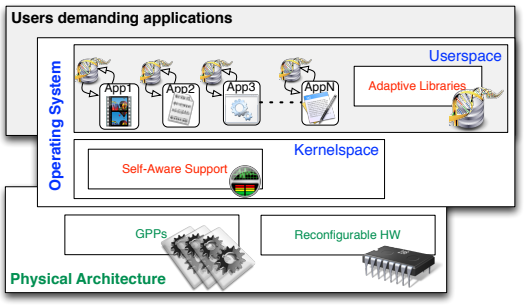


Fig. 1. The overview of a Self-Aware systems where the operating system is in charge on managing the online adaptation of the applications and of the underline architecture.

4. EXPERIMENTAL RESULTS

We designed a self-aware implementation [29] of the GNU/Linux operating system able to monitor itself to take autonomous decisions on the best implementation for the demanded functionalities. Each software application, also named *process*, can issue one or more system calls in order to require a specific functionality, which may be available either as a classical software library, as an adaptive software, or as hardware

IP-Cores, or all of them. The operating system is in charge of choosing among the software or the hardware implementation according to different criteria, such as the amount of free area on the FPGA, or the dimension/number of data that has to be processed.

The case study that we would like to present, belongs to the cryptographic application domain. A cryptographic reconfigurable architecture, implementing the *Data Encryption Standard*, has been designed to evaluate the performance of the run-time decision of the best implementation for any demanded task. The proposed case study, as shown in Figure 2, compares the performance of different implementations of the DES algorithm. The FPGA-based solutions have

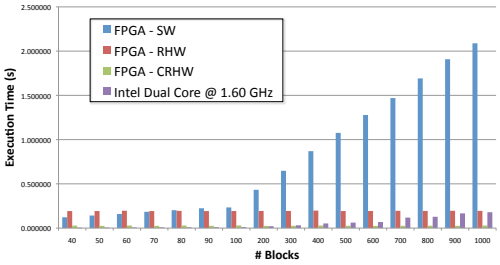


Fig. 2. Performance, in execution time, of the different implementations of the DES algorithm.

been implemented on a Xilinx Virtex-II Pro working with at 100MHz, while the data regarding the software solution has been taken using an Intel Pentium Dual Core working at 1.60GHz with Linux (kernel 2.6.27). Three different FPGA implementations have been implemented:

- SW: the DES algorithms has been executed in software on the processor on the FPGA;
- RHW: the algorithm has been implemented as a reconfigurable component and finally;
- CRHW: the reconfigurable component was already configured on the FPGA and ready to be used.

To optimize the execution time of a functionality, it is important for the operating system to be able to choose the best implementation at runtime. Therefore, the OS has not only to be able to understand on which *scenario* of the graph shown in Figure 2 it is working, but to foresee the impact of its decision on future calls. This will lead the OS to choose the most appropriate implementation for the demanded task, that may not lead to the best performance to that specific call, but that may provide better performance to the next ones.

In a scenario were we have enough area on the FPGA to configure the HW implementation of the DES algorithm, for a call on at least 300[1] blocks, it is not always the best decision to go for the Intel Dual Core solution even if we do not have the core algorithm already implemented as an HW IP-Core. To explain this situation we can consider the scenario characterized by two calls of the DES algorithm, the first one on 1000 blocks and the second one on 400. As shown in Figure 2, the best implementation, when the HW IP-Core has not been already configured on the FPGA, is the Dual Core one. Within this scenario, where the system has no knowledge of future calls (it is not aware of the fact that after the 1000 call it will serve a 400 one), the OS will always choose the Dual Core implementation of the DES. This is the solution already implemented in literature in different works [30, 31]. On the contrary, considering the history of the previous calls, the performance information of all the possible implementations, and the probability of receiving a certain call, our system will choose the reconfigurable HW solution (reconfiguration of the IP-Core and its execution) for the first call, since it will be payback for each consecutive call on at least 400 blocks. Table 1 presents the comparison between the two different approaches.

5. REFERENCES

[1] P. Dini, "Internet, grid, self-adaptability and beyond: are we ready?" *Database and Expert Systems Applications, 2004. Proceedings. 15th International Workshop on*, pp. 782–788, Aug.-3 Sept. 2004.

[2] IBM Inc., "IBM autonomic computing website," 2009. [Online]. Available: http://www.research.ibm.com/autonomic/

[3] O. Krieger, M. Auslander, B. Rosenburg, R. W. J. W., Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: building a complete operating system," *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pp. 133–145, 2006.

[4] Oracle, "Automatic workload repository (awr) in oracle database 10g," Website, http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php.

[5] Intel, "Reliability, availability, and serviceability for the always-on enterprise. the enhanced ras capabilities of intel processor-based server platforms simplify 24x7 business solutions," Online document, www.intel.com/assets/pdf/whitepaper/ras.pdf, 2005.

[6] E. F. by the European Union's 7th Framework Program, "Exposing the features in ip version six protocols that can be exploited extended for the purposes of designing/building autonomic networks and services," 2009. [Online]. Available: http://www.efipsans.org

[7] M. M. Masters, "Exploring usability in mobile autonomic networks," in *MobileHCI '08: Proceedings of the 10th international conference on Human computer interaction with*

[1] Which is the point where the CRHW implementation outperforms the Intel Dual Core one

Table 1. Different possible exe., for the same sequence of inputs

First call, #Blocks: 1000	**Second call, #Blocks: 400**	**Overall Execution TIme** (s)
Intel Core Duo: 0.179162 s	Intel Core Duo: 0.052382 s	**0.231544**
RHW: 0.194679 s	CRHW: 0.025710 s	**0.220389**

mobile devices and services. New York, NY, USA: ACM, 2008, pp. 549–550.

[8] J. Buisson, F. André, and J. L. Pazat, "Dynamic adaptation for grid computing," *Lecture Notes in Computer Science. Advances in Grid Computing - EGC*, pp. 538–547, 2005.

[9] S. S. Vadhiyar and J. J. Dongarra, "Self adaptivity in grid computing," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 235–257, 2005.

[10] P. Reinecke and K. Wolter, "Adaptivity metric and performance for restart strategies in web services reliable messaging," in *WOSP '08: Proceedings of the 7th International Workshop on Software and Performance*. ACM, 2008, pp. 201–212.

[11] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Processor power reduction via single-isa heterogeneous multi-core architectures," *Computer Architecture Letters*, vol. 2, no. 1, pp. 2–2, January-December 2003.

[12] B. Sprunt, "Pentium 4 performance-monitoring features," *IEEE Micro*, vol. 22, no. 4, pp. 72–82, Jul/Aug 2002.

[13] R. Azimi, M. Stumm, and R. W. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *ICS '05: Proceedings of the 19th Inter. Conf. on Supercomputing*, 2005, pp. 101–110.

[14] HAGGLE, "A european union funded project in situated and autonomic communications," 2007. [Online]. Available: http://www.haggleproject.org

[15] ANA, "Autonomic network architecture," 2009. [Online]. Available: http://www.ana-project.org

[16] C. M. Garcia-Arellano, S. Lightstone, G. Lohman, V. Markl, and A.Storm, "A self-managing relational database server: Examples from IBM's DB2 universal database for linux unix and windows," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 36, no. 3, pp. 365– 376, 2006.

[17] J. Appavoo, K. Hui, M. Stumm, R. W. Wisniewski, D. D. Silva, O. Krieger, and C. A. N. Soules, "An infrastructure for multiprocessor run-time adaptation," in *WOSS '02: Proceedings of the first Workshop on Self-healing Systems*. New York, NY, USA: ACM, 2002, pp. 3–8.

[18] D. Breitgand, M. Goldstein, E. Henis, O. Shehory, and Y. Weinsberg, "Panacea towards a self-healing development framework," in *Integrated Network Management*. IEEE, 2007, pp. 169–178.

[19] E. U. F. P. CASCADAS, "Component-ware for autonomic situation-aware communications, and dynamically adaptable services," 2009. [Online]. Available: http://www.cascadas-project.org

[20] A. Fox, E. Kiciman, and D. Patterson, "Combining statistical monitoring and predictable recovery for self-management," in *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. New York, NY, USA: ACM, 2004, pp. 49–53.

[21] J. Strassner, S.-S. Kim, and J. W.-K. Hong, "The design of an autonomic communication element to manage future internet services," in *APNOMS*, ser. Lecture Notes in Computer Science, C. S. Hong, T. Tonouchi, Y. Ma, and C.-S. Chao, Eds., vol. 5787. Springer, 2009, pp. 122–132.

[22] A. Quiroz, N. Gnanasambandam, M. Parashar, and N. Sharma, "Robust clustering analysis for the management of self-monitoring distributed systems," *Cluster Computing*, vol. 12, no. 1, pp. 73–85, 2009.

[23] J. Vetter and P. Worley, "Asserting performance expectations," in *Supercomputing, ACM/IEEE 2002 Conference*, Nov. 2002, pp. 33–33.

[24] M. Caporuscio, A. Di Marco, and P. Inverardi, "Run-time performance management of the siena publish/subscribe middleware," in *WOSP '05: Proc. of the 5th Inter. Work. on Software and performance*, 2005, pp. 65–74.

[25] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski, "Performance and environment monitoring for continuous program optimization," *IBM J. Res. Dev.*, vol. 50, no. 2/3, pp. 239–248, 2006.

[26] S. Oberthür, C. Böke, and B. Griese, "Dynamic online reconfiguration for customizable and self-optimizing operating systems," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2005, pp. 335–338.

[27] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *PPoPP '05: Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2005, pp. 277–288.

[28] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," in *Conf. on Programming Language Design and Implementation*, Jun 2009.

[29] M. D. Santambrogio, "From reconfigurable architectures to self-adaptive autonomic systems," *IEEE International Conference on Computational Science and Engineering*, pp. 926 – 931, 2009.

[30] M. D. Santambrogio, I. Beretta, V. Rana, and D. Sciuto, "On-line task management for a reconfigurable cryptographic architecture," in *IEEE International Symposium on Parallel and Distributed Processing, 2009. IPDPS 2009*, May 2009.

[31] V. Sima and K. Bertels, "Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform," in *IEEE International Symposium on Parallel and Distributed Processing*, May 2009.

A LOW OVERHEAD TEMPERATURE SENSOR FOR SELF-AWARE RECONFIGURABLE PLATFORMS

Rizwan Syed, Wenfeng Zhao, Yajun Ha and Bharadwaj Veeravalli

Department of Electrical and Computer Engineering
National University of Singapore
Singapore 117576
email: {*syed.rizwan,wenfengzhao,elehy,elebv*}@nus.edu.sg

ABSTRACT

To enable the self-awareness in reconfigurable platforms, FPGAs will require sensors to measure various physical quantities. A physical quantity such as the temperature profile of an FPGA die allows the platform to perform thermal management by dynamically relocating workloads. Previous research works have already demonstrated such self-aware and adaptive reconfigurable platforms. However, the relative high resource utilizations by their temperature sensors make their proposals less useful. We develop a low overhead temperature sensor for reconfigurable platforms that utilizes only 7 Flip Flops, 16 6-LUTs and 7 SRL32E. This is 52% less than that of the state-of-the-art. The resolution of our temperature sensor is 0.5°C with a sampling period of only 1ms and the accuracy is ±0.5°C using two-point calibration. We use the sensor in our reconfigurable platform to demonstrate its effectiveness.

1. INTRODUCTION

Due to the feature size shrink of transistors, integrated circuits are able to achieve a much higher density to realize a complex and high performance system in a single chip. However, their ever increasing unit area power densities also lead to thermal reliability issues. This fact holds true for high-performance CPUs, GPUs, ASICs and reconfigurable devices like FPGAs [1], and the thermal management is critical for these platforms to ensure both the correct functionality and the longer life-time. Temperature sensors based on band-gap voltage and ADCs have been reported for commercial CPUs. They are integrated and placed in the same CPU die for dynamic thermal monitoring and hot-spot profiling.

Temperature sensors are also needed for FPGA platforms to generate a die temperature profile and realize the thermal monitoring for self-awareness. However, off-the-shelf FPGAs provide only a small and fixed number of temperature sensors for the die temperature sensing, and they are incapable of thermal profiling across the whole die and in flexible locations. As a result, hot spots cannot be effectively monitored. ASIC implementations of all-digital temperature sensors are reported in [2, 3]. These temperature sensors can be categorized into either the time domain (temperature dependent delay) or the frequency domain (temperature dependent frequencies) sensors. However, either of them can be implemented in FPGA platforms. The design trade-offs of temperature sensors on FPGAs should be determined among the resource utilization, accuracy, resolution and self-heating. To measure the die temperature profile, these sensors should be instantiated in large numbers across the entire die. As a result, each sensor should have a low resource utilization.

In this work, we propose a low overhead temperature sensor which can be instantiated in large numbers to measure the temperature profile of an FPGA die. The sensor uses 52% less resource than that of the state-of-the-art. It achieves a resolution of 0.5°C with a sampling period of only 1ms, and an accuracy of ±0.5°C using two-point calibration.

2. RELATED WORK

Some previous research works focused on thermal-aware application management [4, 5, 6, 7]. However, the actual performance and area requirements of the sensor have not been discussed. Other research works have been done to develop and demonstrate temperature sensors on FPGAs. However, due to their overheads and performance limitations, they were practically less useful. A fully digital time-domain temperature sensor [2] was demonstrated on FPGA using 140 logic elements. Later, they improved their design using a retriggerable ring oscillator [3], but their design was not optimized for FPGAs as it was targeted at ASICs. Other works [8] have also tried to improve the performance of their temperature sensors by using different techniques. However, their area overheads are significant. We demonstrate a low overhead time domain temperature sensor using a retriggerable ring oscillator which offers the comparable accuracy

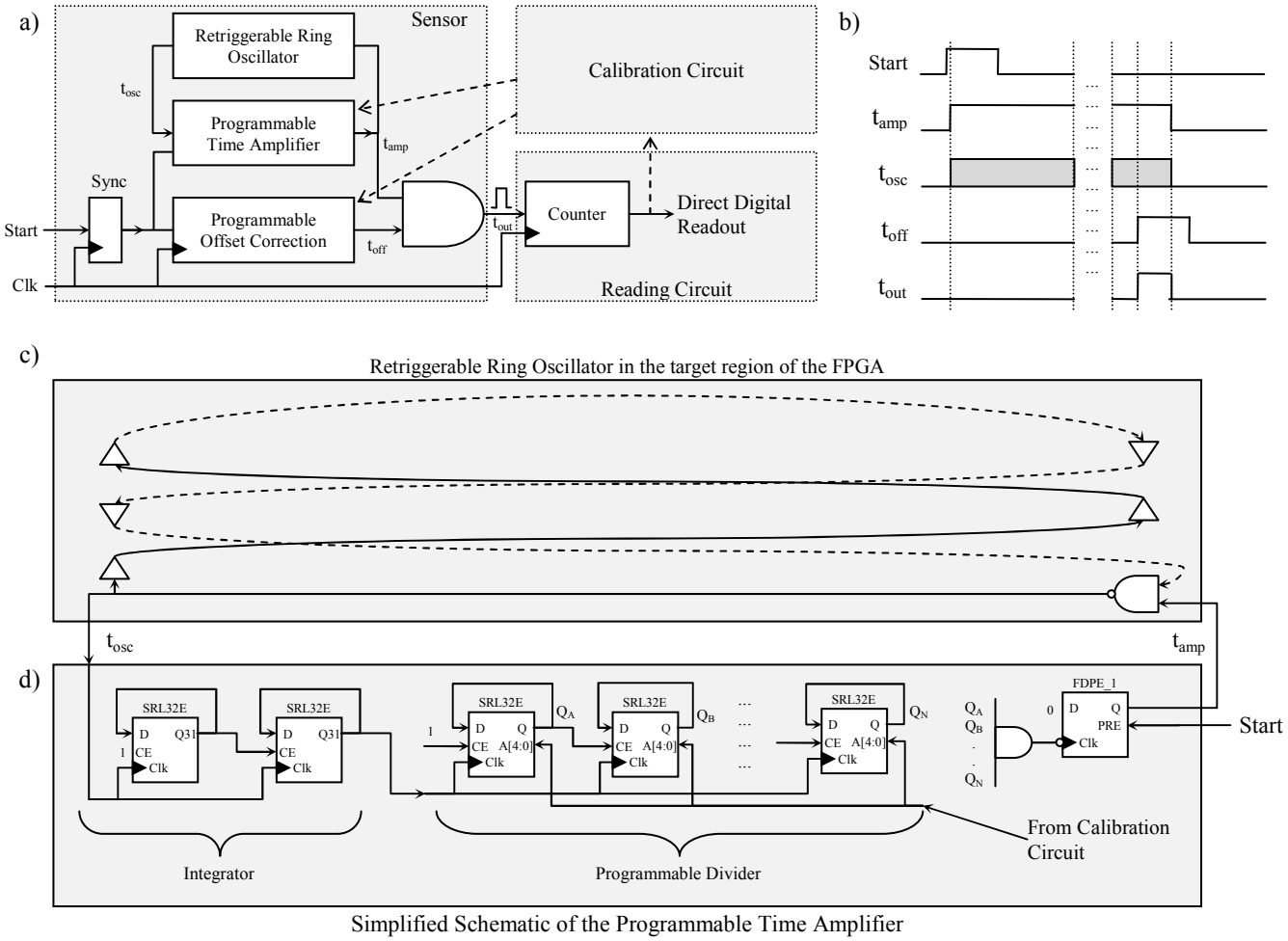


Fig. 1. (a) Block Diagram of the Proposed Sensor, (b) Timing Diagram of the Temperature Sensor, (c) Layout of the Retriggerable Ring Oscillator in the target region of the FPGA, (d) Simplified Schematic of the Programmable Time Amplifier

and the improved sampling performance. It is modular and features direct digital readout in the calibrated unit.

3. CIRCUIT DESCRIPTION

In this section, we present our temperature sensor. The block diagram of our temperature sensor is shown in Fig.1. The complete design can be divided into three sections: (1) Main Sensor, (2) Reading Circuit, and (3) Calibration Circuit. Each component is explained in the following subsections.

3.1. Main Sensor

The main sensor measures temperature by calculating the increase in the period of the retriggerable ring oscillator. The actual increase in the period is very small and cannot be measured directly. Therefore it is amplified using a programmable divider (Programmable Time Amplifier). The delay is accumulated over time during the division. As shown in Fig.1, t_{amp} enables the ring oscillator and programmable divider divides the oscillation t_{osc}. Signal t_{amp} is corrected using the programmable offset correction circuit to convert the extended period into a pulse that is directly proportional to the temperature with zero bias. In other words, at 0°C, the output is just the starting pulse. The pulse width of the output of a calibrated temperature sensor will be:

$$t_d = T \times t_{Clk} \times \alpha \times \beta + t_{Clk} \quad (1)$$

where, t_d is the pulse width of the output, T is the temperature in the desired unit, t_{Clk} is the period of the reference clock, α is the inverse of accuracy in the desired unit, and $beta$ is the integration factor. It can be seen from the equation that the pulse width will be 1 clock period wider than the actual value. This is because a starting pulse is always added to the beginning of the output pulse to enable the synchronization at the reading circuit.

Unlike previous work, the retriggerable ring oscillator in our sensor uses the transport delay in signal through the routing fabric of the FPGA as shown in Fig. 1(c). The placement of each LUT is constrained to be near the periphery of the region such that the routing distance between each is as large as possible. The exact number of buffers in the oscillator is a system parameter and can be adjusted for different sized regions. The placement of LUTs and routing is matched among different sensors in similar sized region through placement and directed routing constraints. However, it was observed that significant jitter was present in the period of the ring oscillator. This decreased the accuracy of the sensor significantly. To reduce the effect of the jitter, the output pulse from the ring oscillator was accumulated by using a fixed divider refer to as the integrator in Fig.1(d). The division factor is referred to as integration factor in equation 1. The extended versions of the pulse is again extended by programmable divisor (Programmable Time Amplifier). This dividing factor depends upon the resolution and calibrated unit. The factor is set such that unit increase (w.r.t the resolution and accuracy of measurement) in pulse length is exactly one clock period (t_{Clk}). In other words, if the resolution is high then the divisor will be larger. This implies a larger conversion time and larger area requirements for the programmable divider. Therefore, there is a trade-off between the conversion time and resolution. Offset correction is performed by using a masking pulse that is only high between minimum and maximum measurable temperature w.r.t the output of the ring oscillator. In this case, the pulse was high only between 0°C and 80°C. The implementation use an efficient implementation of dividers and pulse generators using cascaded shift registers as shown in Fig.1.

3.2. Reading Circuit

The reading circuit only consists of a simple up-counter that is enabled by the output pulse of the temperature and thus measure the pulse width of the output pulse. The count of the counter is a direct binary readout of the temperature in the selected unit during the calibration. The counter is cleared at the rising edge of the output pulse. This automatic synchronization allows us to interface many temperature sensors by multiplexing the input of the reading circuit.

3.3. Calibration Circuit

The calibration circuit is used to read the digital readout across the operating temperature range and load the corrected gain and offsets coefficients into each sensor. The corrected divisors and offsets are calculated based on the readout from the sensor and the actual reading. This circuit is only required during calibration and can be removed by hard coding the coefficients. The procedure used to calibrate the sensor is mentioned in the Section 4.

4. CALIBRATION

The actual oscillating frequency of the ring oscillator increases non-linearly with increase in temperature over a large temperature range. However, for small ranges, the relationship can be approximated using a straight line. This allows us to use a simple counter as a reading circuit. For better accuracy, two point method can be used which corrects both the gain and offset coefficients. For simpler calibration, one point calibration can be used which only corrects the offset at the given temperature. Since the gain is not calibrated, therefore, the accuracy over the selected temperature range may be reduced.

For two-point calibration, the temperature of the FPGA is controlled using a temperature controlled oven. Two temperatures are selected on the basis of the operating range of the FPGA and the readout of the sensors at those temperature are recorded using default gain and offset coefficients. The actual operating frequencies of the ring oscillators in each sensor can be calculated using the readouts at each temperature. Then we can calculate the correct values of the gain and offset coefficients and load it using the calibration circuit. After verification, the calibration circuit can be completely removed and the gain and offset coefficients can be hard coded in HDL. The placement and routing of the ring oscillators are recorded in a constraint file. This allows the calibration to be preserved for that FPGA. Because of process variation among different FPGAs of the same type, the oscillating frequency of the ring oscillator is not the same and therefore, the calculated coefficient are only accurate for that FPGA. Therefore, calibration is required for different FPGAs. To simplify the calibration process, only one-point calibration can be performed by calculating the offset coefficient using the calibrated coefficients (of another FPGA of the same type) as default. Since the change in gain coefficient is generally not large from one FPGA to another (of the same type), therefore the accuracy will not be badly affected.

5. EXPERIMENTAL RESULTS

We used Xilinx Virtex 6 board ML605 for the implementation. The ring oscillator of the sensor used 10-LUTs. The sensors were calibrated using the on-die temperature sensor in a temperature controlled oven from 35°C to 70°C. ChipScope Pro Virtual I/O was use to measure the readout and load corrected divisors and offsets. We achieved an accuracy of ± 0.5°C using two-point calibration on two FPGAs and an integration factor of 64. We compare in Table 1 the resource utilization and the performance of our sensor with that of the related works.

We further build an evaluation platform based on the AARP [9] on Xilinx ML605 Board [10]. The AARP platform allows us to instantiate applications dynamically in the

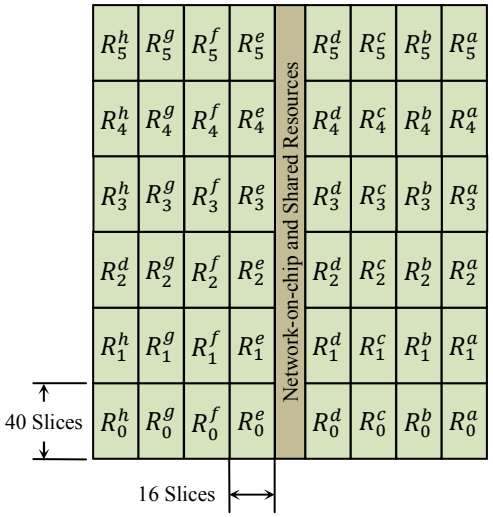


Fig. 2. Floorplan of Implementation

Table 1. Comparative Study

	#LE	Resolution	Accuracy	Samples/sec
[2]	140	0.06	-1.5 to 0.8	3000
[3]	48	0.13	-0.7 to 0.6	4400
Our Sensor	23*[24†]	0.5	0.5	1000

* Resource Usage on Virtex 5/6, Spartan 6, and Series 7 FPGAs: 07 FFs, 16 6-LUTs, 07 SRL32Es packed into 23 Equivalent LEs

† Resource Usage on Virtex 4 and Spartan 3 FPGAs: 07 FFs, 15 4-LUTs, 09 SRL16Es, 22 MUXFX, 03 MUXF5 packed into 24 Equivalent LEs

FPGA. The floorplan of the implementation on FPGA is shown in Fig.2. As shown in the Fig.2, there are $8 \times 6 (= 48)$ regions available. We develop a small dummy application using only one region and embed our developed sensor in it. We instantiate the dummy application in all the regions and calibrate the sensors using the one-point calibration using the initial coefficients from the first experiment. We achieve an accuracy of 0.5°C for the operating temperature range from 35°C to 70°C. We are able to successfully measure the temperature profile of the reconfigurable regions of the FPGA. This information will be used by the system manager for hot spot detection and thermal management through dynamic frequency scaling and task scheduling/relocation in a future work.

6. CONCLUSION

We present a low overhead time domain temperature sensor to enable self-awareness for reconfigurable platforms. The sensor uses 52% less resource than that of the state-of-the-art. Its resolution is 0.5°C with a sampling period of only 1ms. Its accuracy is $\pm$0.5°C using two-point calibration.

7. ACKNOWLEDGEMENT

The authors acknowledge the funding support from Singapore A*Star (grant no 1122804010).

8. REFERENCES

[1] P. Mangalagiri, S. Bae, R. Krishnan, Y. Xie, and V. Narayanan, "Thermal-aware reliability analysis for platform fpgas," in *ICCAD 2008 IEEE*, nov. 2008, pp. 722 –727.

[2] P. Chen, M.-C. Shie, Z.-Y. Zheng, Z.-F. Zheng, and C.-Y. Chu, "A fully digital time-domain smart temperature sensor realized with 140 fpga logic elements," *IEEE Trans. Circuits Syst. I*, vol. 54, no. 12, pp. 2661 –2668, dec. 2007.

[3] P. Chen, S.-C. Chen, Y.-S. Shen, and Y.-J. Peng, "All-digital time-domain smart temperature sensor with an inter-batch inaccuracy of -0.7C-0.6C after one-point calibration," *IEEE Trans. Circuits Syst. I*, vol. 58, no. 5, pp. 913 –920, may 2011.

[4] P. Jones, J. Moscola, Y. Cho, and J. Lockwood, "Adaptive thermoregulation for applications on reconfigurable devices," in *FPL 2007 Intl. Conf.*, aug. 2007, pp. 246 –253.

[5] P. H. Jones, Y. H. Cho, and J. W. Lockwood, "Dynamically optimizing fpga applications by monitoring temperature and workloads," in *VLSID 2007 Intl. Conf.*, jan. 2007, pp. 391 –400.

[6] D. Atienza and E. Martinez, "Inducing thermal-awareness in multicore systems using networks-on-chip," in *ISVLSI 2009 IEEE CS Symp.*, may 2009, pp. 187 –192.

[7] X. Zhang, W. Jouini, P. Leray, and J. Palicot, "Temperature-power consumption relationship and hot-spot migration for fpga-based system," in *GreenCom 2010 IEEE/ACM Intl. Conf.*, dec. 2010, pp. 392 –397.

[8] Z. Chen, R. Nagesh, A. Reddy, and P. Schaumont, "Increasing the sensitivity of on-chip digital thermal sensors with pre-filtering," in *VLSI 2009 IEEE CS Symp.*, may 2009, pp. 304 –309.

[9] R. Syed, Y. Ha, and B. Veeravalli, "A low overhead abstract architecture for fpga resource management," in *HEART 2012 Intl. Workshop.*, Jun 2012.

[10] Xilinx, "Xilinx ML605 User Guide UG534," 2008.

THE SERVICES COORDINATOR: ORCHESTRATING THE BEHAVIOR OF INDEPENDENT ADAPTIVE SYSTEMS

Jacopo Panerati †, *Martina Maggio* ‡, *Matteo Carminati* †, *Marco Triverio* †, *Marco D. Santambrogio* †

† Dipartimento di Elettronica e Informazione, Politecnico di Milano

‡ Automatic Control, Lund University

{jacopo.panerati,marco.triverio}@mail.polimi.it, martina.maggio@control.lth.se

{mcarminati,santambrogio}@elet.polimi.it

ABSTRACT

Nowadays the complexity of computing systems is skyrocketing. Programmers have to deal with extremely powerful computing systems that take time and considerable skills to be instructed to perform at their best. This work analyzes the stated problem and proposes a simple, yet powerful mechanism for optimizing performance through the coordination of the interaction of multiple, independent adaptive systems called *services*. In this scenario we developed the Services Coordinator, a system-centralized *decision engine* based on reinforcement learning. The Services Coordinator gathers information about the performance goals of the system and can either turn services *on* or *off*. The Services Coordinator analyzes the runtime impact of services and of their autonomous decision policies, looking for a combination of services that makes it possible to reach the given goals. The experiments that have been carried out show the ability of the Services Coordinator to adapt to changing conditions, confirming the validity and the flexibility of the followed approach.

1. INTRODUCTION

The power and the complexity of computing systems are evolving and increasing at an unprecedented rate. On one hand, the advantages of highly-parallel systems could benefit an enormous variety of fields. On the other hand, the growing complexity is making it unfeasible for the average programmer to weight all the constraints and optimize the system for a wide range of machines and scenarios [1]. Even though technologies have improved, making a system perform at its best is a non-trivial task. The burden on programmers is noticeable and many research efforts were spent in addressing this issue. Clearly, it is not feasible to rely on human intervention to tune a system: conditions change constantly, rapidly, and unpredictably. It would be desirable to have the system automatically *adapt* to the mutating environment [2].

A commonly shared opinion is the need for new paradigms to be explored and for new frameworks to be developed. Among those, self-adaptive systems seem to be the answer to most of the problems previously described [2]. Self-Aware Adaptive computing systems adapt behavior and resources to automatically find the best way to accomplish a given goal despite changing environmental conditions and demands. Therefore, this kind of system needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions.

This work presents a solution able to adjust itself during execution due to a simple, yet powerful mechanism for coordinating the interaction of multiple, existing adaptive systems, each of them with its own decision engine. The system used in this paper considers current computing and operating systems augmented through the usage of a modular ecosystems of software components called *services*, capable of actuating a change on the system. The proposed solution can be used within different architectures, from mobile devices (e.g. mobile phones), to desktops, servers and laptops. In this scenario we developed a system-centralized *decision engine*, called Services Coordinator and based on reinforcement learning. The Services Coordinator gathers all the information about system performance and is able to decide which measures to enact, orchestrating the different elements that commit changes to the environment. The Services Coordinator is fully integrated in a standard Linux operating system, aiming at making it a Self-Aware Adaptive computing system, and co-exists with other independent and adaptive decision making entities, which reside in services. The Services Coordinator turns services either *on* or *off* as necessary to meet the performance goals of the system. We validated the framework with preliminary tests, aimed at evaluating the approach and the feasibility of the solution.

The remainder of this paper is organized as follows. Section 2 presents the context of adaptive operating systems. Section 3 introduces and describes in detail the proposed methodology and the machine learning-based decision engine. In Section 4 our experimental results are presented. Finally, Section 5 concludes the paper.

2. CONTEXT DEFINITION

The goal of the Services Coordinator is to enhance the Operating System (OS) with a novel adaptivity layer at run time. A common Linux distribution is chosen here as the target OS to show the wide applicability of the approach in both designing a completely new OS and complementing existing code. In literature, some projects exist that are explicitly

trying to enable self-awareness and adaptiveness in a newly designed OS. Space limitations allow only a brief description of main contributions in the area: *K42* [3] and *Sefos* [4].

K42 is a research kernel designed for cache-coherent 64-bit multiprocessor and NUMA systems. Among its declared goals there is allowing applications to customize the OS behavior in how the OS is managing the resources devoted to them. Another stressed goal is letting the system adapt to changing workload characteristics. To achieve these goals, the overall structure of K42 is based on the object-oriented paradigm and on a modular design based on microkernels. K42 proposes a layered architecture based on kernels, servers, and user-level libraries for applications developing. The servers marshal all the operating system functionalities, therefore introducing adaptation at the OS level means simply modifying the behavior of these servers. It is noticeable that however, a central, coordinating entity is not included in this structure.

SElf-aware Factored Operating System (Sefos) is a self-aware OS specifically designed for scalability on many-cores architectures. It is based on the *fos* [5] operating system, and integrated with *SEEC* [6] (SElf-awarE Computational model), the purpose of the integration being the introduction of the self-adaptation layer. To do this, a typical decision loop is implemented: the autonomic system executes and monitors itself using sensors. The system is able to react to the sensed conditions, taking decisions and acting to guarantee application performance. The adopted monitoring interface is Application Heartbeats [7, 8]. There is a decision engine acting on the system to set the values of each decision parameter. However, when multiple decisions are taken at the same time, it is not clear how the coordination between the different actuation mechanisms available in the system takes place. Notice also that Sefos relies on trusted actuators: no security checks are performed on the taken decision, thus malicious entities cannot be detected and deactivated.

Similarly to Sefos, the adaptive system the Services Coordinator lives in, implements a decision loop. However the Services Coordinator represents a new entity: it is a system-centralized decision engine, learning how to enable and disable the available adaptation mechanisms in order to meet the high level goals of the system.

3. PROPOSED METHODOLOGY

To better understand the importance and the rationale behind the Services Coordinator, the general scenario[1] that we are addressing is briefly presented. It is made of applications, processes and monitored processes, services and the Services Coordinator, in a system structured as shown in Figure 1:

- *Applications* – Pieces of software written to accomplish a specific task.
- *Processes* – Instances of an application.
- *Monitored Processes* – Processes that are making one or more entities of the system aware of their performance goals and actual progresses. To do this, the Application Heartbeats framework [8, 7] is used.
- *Services* – A service represents a component capable of performing changes on one or more applications or on the whole system. Services are enabled or disabled by our decision engine; when enabled, they are autonomous components that can decide and act on the system, for example reading the performance signals of the applications and/or computing some reaction to changes in the external environment.
- *Services Coordinator* – It is the object the focus of this document is on. It is aware of monitored processes and services, gathers data about applications performance levels and acts enabling or disabling services. Its action policy is based on a machine-learning technique.

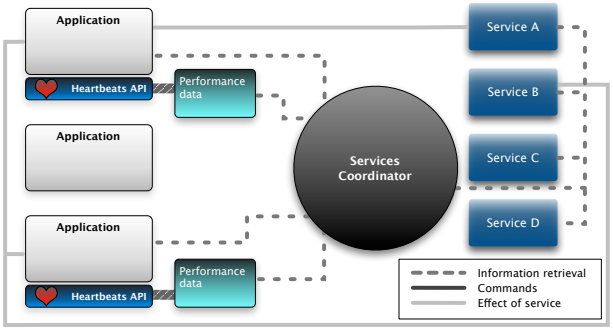


Fig. 1. The Services Coordinator architecture.

In this context, the applications set their performance goals and the *Services Coordinator* orchestrates the available services boosting (or reducing) performance in order to make it possible to achieve the given goals. This scenario features the Services Coordinator as a central element that has a global vision of the system.

We believe this approach has the power to achieve, if possible, a global optimum. We are conscious of the theoretical fragility of the proposed architecture, the Services Coordinator representing a *single point of failure*. This condition has been partially mitigated by our implementation, which allows each enabled component of the framework to

[1]The description of the overall approach with details of each component, presenting a set of experiments to prove its effectiveness, is out of the scope of this paper. This paper is focusing its attention of the Services Coordinator, the key component at the center of the decision loop.

run even in case of failure of one or more other components. However, the research is still ongoing on this topic and more experiments have to be carried out to evaluate different solutions.

3.1. Orchestrating the services

The Services Coordinator stands at the center of the decision loop and exploits the awareness given by the available observation mechanisms (in this case Application Heartbeats) to elaborate a plan for future behavior. The aim is to tune performance in order to make each monitored process achieve its performance goals. In particular, the Services Coordinator exploits the information coming from the monitor in order to determine the available services and the monitored applications; constantly verifies the availability of new or old services and the presence of new or old monitored processes; gathers the information coming from the monitored processes; analyzes the performance-related data in order to understand whether to enact a correction policy; decides which services to enable or disable and communicates them these decisions.

The decision policy that drives the Services Coordinator is based on *machine learning* techniques, which were proved powerful tools for managing the increasing complexity of computing systems [9, 10, 7]. We implemented *R-learning*, a *reinforcement learning* algorithm [11]. In general, Reinforcement Learning augments a system with the possibility to learn from experience through the use of a *reward signal* that drives the learning process. In particular, the algorithm calculates a signal that is a synthesis of the current state (and of its performance characteristics, such as whether the applications are in the desired performance range); it then selects actions attempting to maximize the given reward. A Reinforcement Learning technique is needed since the Services Coordinator has no a-priori knowledge about the action performed by each service in the system. Within this context, it has to discover the effects of each independent adaptive system enabling and disabling it. Furthermore, Reinforcement Learning provides an efficient method to build knowledge from experience.

Once a strategy has been decided by the Services Coordinator, it must be enacted. In our frameworks the actuators are called *services*. There are many kinds of services that might be available and that might affect the system in different ways: some might affect performance, accuracy, or both; some might impact on the whole system while others might target one or more applications. In the following we focus on a couple of services but many others could be envisioned (e.g., lock mechanisms [7], memory allocation and frequency scaling [6]). The two services we address are core allocation and priority adjusting. Both these optimizations are enabled on a per-application basis. It is worth stressing that this means that a service per application could be activated.

In our design of the Services Coordinator we paid attention to some of its characteristics. First, the interface between the Services Coordinator and the services is extremely simple. Second, there is no need to model a service – something that might prove truly difficult given the heterogeneity of the computing systems on which the framework could run. Moreover, the Services Coordinator does not need to be updated or restarted when new services are plugged in, to the extent of self-configuration. All services embed a decision making mechanism that is independent from the Services Coordinator: the Services Coordinator only enables or disables services. When a conscious service is enabled its own loop-based decision mechanisms is activated.

4. PRELIMINARY RESULTS

The experimental evaluation was able to enable/disable a multi-application version of the *core-allocator* presented in [6] and a newly developed service named *priority adjuster*. Note that the proposed methodology is general and may be applied with any other service that implements the Services API. The monitored processes were instances of x264, possibly with different parameters, a different number of threads, and different requirements. In this scenario, x264 was run to see its behavior during an uncontrolled execution (neither the Services Coordinator or any of the services are running). The heart rate is bound between 80 and 90 heartbeats per second. On average, with the given parameters and the given number of cores, it signals 85.49 heartbeats per second.

4.1. Overhead

Previous research has shown that the Application Heartbeats framework has a very limited overhead (e.g., only circa 4% on an application encrypting and decrypting through the DES algorithm). We have then analyzed the Services Coordinator and the *Services Application Programming Interface API* in order to quantify the overhead of the framework. In particular, we run an Application Heartbeats instrumented version of x264[2] without the Services Coordinator and with the Services Coordinator and the *core allocator* on an x86-64 Intel Core i7-870 processor[3] and compared the results. The same experiment has been repeated 10 times and then averaged. The Services Coordinator-enabled version has an overhead of circa 5%, an encouraging result.

4.2. Approach validation

In this subsection we present different tests, carried out on the same machine used for the overhead tests. The experimental evaluation was able to enable/disable a multi-application version of the *core-allocator* presented in [6] and a newly developed service named *priority adjuster*. Note that the

[2]x264 is an open-source application for encoding video streams into the H.264/MPEG-4 AVC format.

[3]Clock speed: 2.93 GHz, 4 GB of SDRAM DDR3-1333, NVIDIA GeForce GT 240 graphic card, OS: Ubuntu 9.10.

proposed methodology is general and may be applied with any other service that implements the Services API. The monitored processes were instances of x264, possibly with different parameters, a different number of threads, and different requirements. In this scenario, x264 was run to see its behavior during an uncontrolled execution (neither the Services Coordinator or any of the services are running). The heart rate is bound between 80 and 90 heartbeats per second. On average, with the given parameters and the given number of cores, it signals 85.49 heartbeats per second.

4.3. Different performance goals

This experiment tests the reaction of the system in a multi-application and multi-service domain: a second application is started around 30 seconds from the beginning of the experiment. Both the applications are instances of x264, with different performance goals (desired heart rate of 20 to 30 heartbeats per second for the first application and 30 to 50 for the second one). The system behavior is shown in Figure

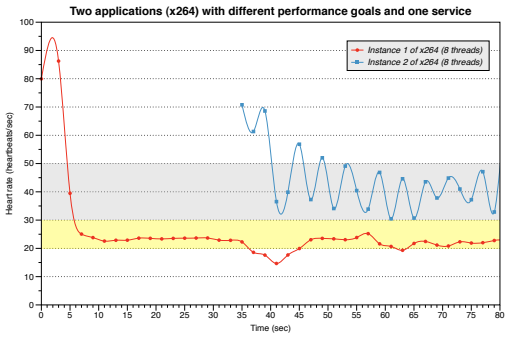


Fig. 2. Two instances of x264 with different performance goals. Desired heart rate ranges are in yellow and gray.

2. When the second x264 instance is introduced the Services Coordinator explores the new state-action space entering a suboptimal condition for the first application. It is however able to recognize the best action to be taken almost immediately, allowing both instances to meet their goals.

4.4. Stress test

This test has been run with eight x264 monitored instances, each of them trying to achieve the same performance goals, their heart rate being between 5 and 10 heartbeats per second. In this experiment eight services can be activated and deactivated, each one being the core allocator for a specific x264 process. Figure 3 shows the results for this test, where the solution space is made up of 256 possible configurations. The time needed to learn the system reactions is intuitively higher than in the previous experiments; nevertheless this test confirms the flexibility of the algorithm. Moreover, the Services Coordinator reaction is fast enough to drive all the eight applications to their desired performance level.

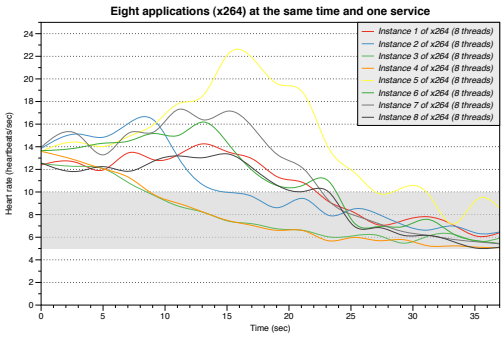


Fig. 3. Eight instances of x264.

5. CONCLUSIONS

In this paper we presented a component, called Services Coordinator, that acts as a decision engine in a self-aware architecture capable of performing optimizations on itself, and adapting to unpredictable, unknown, and unfavorable conditions. The Services Coordinator is the central entity that gathers all the information coming from applications and decides which actions to perform in order to make the processes reach the desired goals. It implements a decision algorithm called *R-learning*, allowing the component to learn from experience and to optimize the performance of the whole system. The introduction of such central entity, based on machine learning, represents the prior contribution of this paper with respect to the works described in literature. Without any prior information on the services, it is shown that the Services Coordinator is able to learn and dynamically improve the quality of the system.

6. REFERENCES

[1] P. Horn, "Autonomic computing: Ibm perspective on the state of information technology," in *IBM Germany Scientific Symposium Series*, 2001.

[2] M. Salehie *et al.*, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.

[3] J. Appavoo, *et al.*, "K42 Overview," http://www.research.ibm.com/K42/white-papers/Overview.pdf, August 2002.

[4] Various Authors, "The mit angstrom project: Universal technologies for exascale computing (project home page)," http://projects.csail.mit.edu/angstrom/index.html.

[5] D. Wentzlaff, *et al.*, "An operating system for multicore and clouds: mechanisms and implementation," in *SoCC*, 2010, pp. 3–14.

[6] H. Hoffmann, *et al.*, "Seec: A framework for self-aware computing," MIT CSAIL, Tech. Rep., October 2010.

[7] J. Eastep, *et al.*, "Smartlocks: lock acquisition scheduling for self-aware synchronization," in *7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809079

[8] H. Hoffmann, *et al.*, "Application heartbeats for software performance and health," *15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 347–348, January 2010.

[9] E. Ipek, *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 39–50. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.21

[10] R. Bitirgen, *et al.*, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *41st annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 318–329. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2008.4771801

[11] R. S. Sutton *et al.*, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 1054–1054, 1998.

DESIGN TOOLS FOR SELF-AWARE SYSTEMS ON FPGAS

Dirk Koch, Christian Beckhoff, Alexander Wold, and Jim Torresen

Department of Informatics,
University of Oslo (Norway)
email: {koch, jimtoer}@ifi.uio.no

ABSTRACT

To fully exploit the capabilities of run-time reconfigurable FPGAs in self-aware systems, design tools are required that exceed the capabilities of present vendor design tools. Such tools must allow the implementation of scaleable reconfigurable systems with various different partial modules that might be loaded to different positions of the device at run-time. This comprises several complex tasks, including floorplanning, communication architecture synthesis, physical constraints generation, and the physical implementation all the way down to the final bitstream generation. In this paper, we present how our GOAHEAD framework helps in implementing self-aware systems with a minimum of user interaction.

1. INTRODUCTION

Partial reconfiguration of FPGAs is a key technology in the implementation of self-aware systems that are capable of adapting behavior and structure of hardware at run-time. For example, reconfigurable modules might be relocated to compensate for device defects or a variable number of accelerator modules might be instantiated in order to adapt to varying compute demands. In general, such self-adaptations require that various modules (each with different resource requirements) can be placed freely and multiple times on the fabric while being able to communicate with the dynamically placed modules.

However, from the FPGA vendor side, there is only weak support for implementing such flexible self-adaptive systems. For example, following the latest partial design flow from Xilinx [1] still does not permit relocation of modules on the FPGA fabric. This means that in a scenario with, for example, 10 possible module placement positions and 5 different modules, it requires 50 individual place & route steps for the modules and consequently 50 partial configuration bitstreams. Moreover, all these physical implementation steps have to be carried out again after changes in the static part of the system. Note that these restrictions also apply for the PR tools from Altera [2].

A further drawback of the vendor tools is that they do not permit sharing a reconfigurable region by multiple modules at the same time and only one module can be placed exclusively into a reconfigurable region. For instance, a large module cannot be replaced by multiple smaller ones. Consequently, the vendor tools neither scale with the complexity required for implementing advanced self-adaptive systems nor do they allow for the implementation of systems that exploit the full flexibility available in an FPGA.

Besides the vendor tools from Xilinx, there exist a few academic approaches to implementing reconfigurable systems on FPGAs. For example, OpenPR [3] allows for the implementation of relocatable modules resulting in a more scalable flow than what is available from the FPGA vendors. However, having multiple modules in a reconfigurable region or the crossing of static routing through a reconfigurable regions is not supported. The tool ReCoBus-Builder [4] includes synthesis capabilities of communication architectures required to integrate multiple modules simultaneously in a reconfigurable region, but the tool only supports older devices. The following sections introduce how our new tool GOAHEAD [5], can be used for building self-aware systems.

2. IMPLEMENTING SELF-AWARE SYSTEMS WITH GOAHEAD

Implementing self-aware systems using partial reconfiguration on FPGAs involves several complex tasks and deep knowledge about self-aware strategies, as well as knowledge on how to build and manage reconfigurable systems. The tool GOAHEAD [5] automates and assists in the latter issues. GOAHEAD provides the following features:

- *Floorplanning* in manual and automatic mode. This is the process of defining reconfigurable regions on an FPGA for hosting dynamically loadable modules.
- *Communication architecture synthesis* for island style and slot-based reconfigurable systems. This is the process of binding signals for the communication with the partial modules to physical wires on the fabric.
- *Physical constraints generation* for place and route.
- *Design rule checking and verification*. GOAHEAD can create netlists for simulation, timing verification,

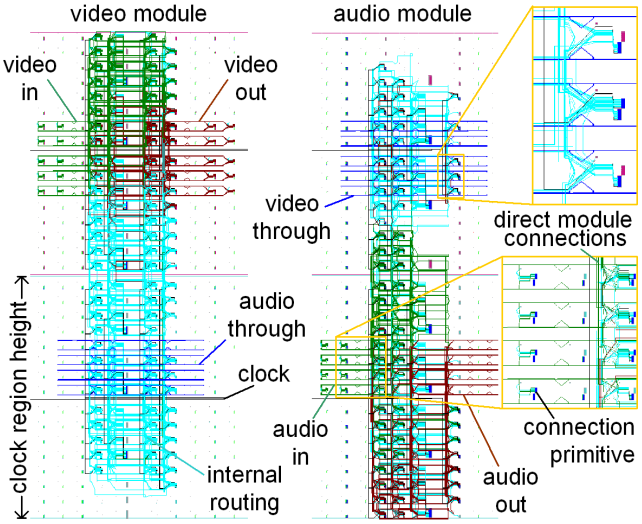


Fig. 1. Audio and video module with stitchable interfaces.

and full bitstream generation of any combination of modules that might occur during system operation.

- *Bitstream assembly* of the static and all partial module configuration bitstreams.

The communication architecture synthesis can create module interfaces that allow stitching various modules together in an arbitrary manner. It is also possible to route signals through the region of a partial module (e.g., for crossing the reconfigurable region), while still being able to relocate modules to different positions on the FPGA (as long as the resource footprint matches). See [5] for more details on module relocation.

Figure 1 shows an example of two stitchable modules. While the left module accesses the video stream while routing through the audio stream, respectively, the right module accesses the audio stream and routes through the video stream. Both modules provide connections that permit direct connections between adjacently placed modules. However, a system might provide *route through modules* in order to bridge a gap between two placed modules. The compatibility of the interfaces is ensured by constraining signals to matching wires on both sides of the module.

The order of stitching the modules is reversible, as long as the underlying resource footprint matches. Because the modules work on different data (i.e., audio and video data), it is possible to stitch the audio module either left or right beside the video module while still providing exactly the same functionality. In the case of multiple modules working on the same data stream, two modes are supported: 1) in *multicast mode* the input stream is tapped and sent directly to the next module which permits an arbitrary placement order of modules along a stream, while 2) in *read-modify-write mode* the input data is processed and the result is streamed to adjacent modules. In the later case, the data dependency results in an placement order along the data stream that has to be followed.

For high performance, the communication can be pipelined and clock rates of more than 300 MHz are possible on Xilinx Virtex-6 FPGAs. Note that GOAHEAD can include pipeline registers into route through channels that permit, for example, to partially load a reconfigurable video module without interfering the audio stream or vice versa. This permits stitching together a large number of modules without dropping the throughput on the channels.

As Opposed to the partial design flows from the FPGA vendors Xilinx and Altera, GOAHEAD does not need connection primitives on the signal paths from or to a partial module (called proxy logic [1] by Xilinx). This removes in particular for small modules (e.g., instruction set extensions for CPUs) the logic overhead and the additional latency of the connection primitives. As shown in the right zoomed box in Figure 1, input signals are routed directly to the module. In the GOAHEAD design flow, connection primitives are placed temporarily *outside* the module. However, by cutting out the module, when generating the partial configuration bitsream, the connection primitive is completely removed from the system.

3. CONCLUSIONS

Our novel tool GOAHEAD provides distinguished features for implementing reconfigurable systems that are not available in the PR tools from the FPGA vendors. The tool is available from [6]. We spent much energy on making it easy to use. Through our effort, we hope to stimulate research on self-aware and self-adaptive systems using FPGAs.

Acknowledgment

This work is supported by the Norwegian Research Council founded project *Context Switching Reconfigurable Hardware for Communication Systems* (COSRECOS) [6], under grant 191156V30.

4. REFERENCES

[1] Xilinx Inc., "Partial Reconfiguration User Guide," 2011, rel 13.2.

[2] Mark Bourgeault, "Alteras Partial Reconfiguration Flow," 2011, available online: `http://www.eecg.utoronto.ca/~jayar/FPGAseminar/FPGA_Bourgeault_June23_2011.pdf`.

[3] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *Proc. of the IEEE Int. Symp. on Parallel and Distr. Processing Works. (IPDPSW)*, 2011, pp. 228–235.

[4] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder– a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs," in *Proc. of Int. Conf. on Field-Progr. Logic and Applications (FPL)*, Sept. 2008, pp. 119–124.

[5] D. Koch, C. Beckhoff, and T. Jim, "GoAhead: A Partial Reconfiguration Framework for Xilinx FPGAs," in *20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Apr. 2012.

[6] "Project website: *Context Switching Reconfigurable Hardware for Communication Systems*." [Online]. Available: http://www.mn.uio.no/ifi/english/research/projects/cosrecos/

AN INFRASTRUCTURE TO INSTRUMENT APPLICATIONS AND MEASURE PERFORMANCE IN SELF-ADAPTIVE COMPUTING

Davide B. Bartolini [†], *Filippo Sironi* [†‡], *Donatella Sciuto* [†], *Marco D. Santambrogio* [†‡]

[†] Dipartimento di Elettronica e Informazione, Politecnico di Milano

[‡] Computer Science and Artificial Intelligence Laboratory, Massachussets Institute of Technology

{bartolini,sironi,sciuto,santambrogio}@elet.polimi.it, {sironi,santa}@csail.mit.edu

ABSTRACT

The shift of mainstream computing architectures to the parallel paradigm, together with the increasing demand for functional and non-functional requirements for modern applications result in a heavy burden for developers and administrators when trying to design and tune computing systems. One of the possibilities for lightening this burden comes from research on runtime self-management and adaptation, which aims at automatizing the runtime management of computing infrastructures. The availability of accurate and appropriate system status information (i.e., self-awareness) is crucial for self-adaptive systems to be functional and useful, and can be achieved through runtime measures provided by *monitors*.

This paper illustrates recent advances in the development of an infrastructure for monitoring applications' throughput called *Heart Rate Monitor* (HRM). Its design and structure are illustrated and a showcase of its capabilities is provided. HRM introduces novel features for a runtime monitor, allowing versatile instrumentation and exposing rich runtime information. These characteristics make HRM an enabling technology for advanced adaptation techniques.

1. INTRODCUTION

The turn of computer architectures from the well understood, single-core structure to multiple (possibly heterogeneous) processing elements is pervasive. This change has been dictated by physical (i.e., inability to increase the clock frequency) and architectural (i.e., diminishing performance returns from efforts in further optimizing the individual processors' internal architecture) constraints [1]. To survive its commitment to exponential performance improvements, the computer industry changed its strategy, leading to the multicore era. However, new issues are arising, as modern multicore processors hit the power wall, being thus constrained to make use of (i.e., switch) only a part of their transistors at the same time and leading to the phenomenon called *dark silicon* [2].

In the single-core era, faster processors provided software performance improvements and applications experienced the so-called "free lunch", with free-of-charge speedups just by switching to the next-generation CPU. The new parallel course in computer architectures, despite being due to architectural causes, carries the side effect of ending the "free lunch", posing a considerable burden of improving performance on software developers. The demands for efficient and reliable parallel software sums up to the already considerable bulk of expertise software developers need to successfully cope with requirements for computing performance, functionality, reliability, and constraints satisfaction due to today's IT. Moreover, computational resources must be carefully managed to avoid hitting power and thermal limits, while respecting Service Lever Agreements (SLAs). This situation leads to an increased need of pushing as much of the system management as possible into computing systems themselves, making autonomic computing a possible breakthrough for IT success [3].

Respecting SLAs employing the least amount of resources is one of the goals of *Autonomic Computing* [3]. Such systems are required to monitor themselves and the environment, detect significant changes, decide a chain of actions, and actuate them [4]. The activity of gathering runtime information (referred to as either *observe* or *monitor* phase) is crucial, and the availability of accurate and appropriate status information can determine the efficacy of the system.

This paper focuses on this phase, presenting an active monitoring [5] infrastructure to observe applications' throughput: the *Heart Rate Monitor* (HRM). HRM is designed to be versatile and provide rich information accounting for simplicity, usability, and functionality. HRM has been successfully employed as a building block in a previous work: the *Metronome* [6] framework, demonstrating its utility in gathering relevant runtime information used to provide performance-awareness in an experimental autonomic operating system. The present paper shifts the focus from the actuation phase to the observation phase and on HRM, providing more details regarding its design principles and implementation, describing novel features which were missing in early revisions of HRM.

2. DESIGN AND IMPLEMENTATION

Throughput is one of the most used metrics for characterizing applications' performance. For instance, the performance of a web server can be characterized in terms of requests served within each time unit (i.e. $\frac{requests}{second}$) while a video encoder can express its performance using the encoding frame rate (i.e., $\frac{frames}{second}$). Being able to access accu-

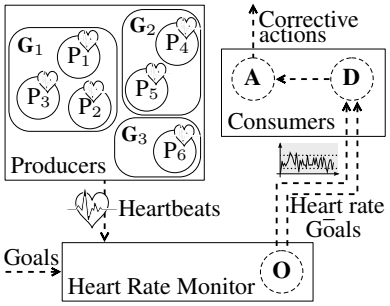


Fig. 1. Black box view of the Heart Rate Monitor. The inputs are heartbeats emitted by instrumented *producers*, organized in *groups*, and goals set by the users. HRM outputs heart rate measures to *consumers*, creating an ODA adaptation loop.

rate and comprehensive information about the throughput of mission-critical applications and to set meaningful performance goals in terms of high-level well understood metrics can enable the system to enact adaptation of resources allocation in order to match SLAs. HRM is designed exactly for this reason: it lets software developers instrument the resource-demanding section (called the *kernel*, or *hotspot*) of the application to emit a *heartbeat* per unit of work done and provides throughput measures in terms of a *heart rate* [6]. Moreover, HRM allows expressing goals in terms of a min–max heart rate window, which directly maps to an application-specific goal.

The advantages of HRM compared to similar solutions (e.g., Application Heartbeats [7, 8]) lie in functionality and efficiency. HRM is functionally superior since it grants both the user and the kernel-space the permission to access information (supporting both the Linux kernel [9] and the FreeBSD operating system [10]). Moreover, HRM supports any kind of parallelization model (i.e., multi-threading and multi-processing, spawning/waiting, pooling, pipelining, etc.). In terms of efficiency, HRM is very low-overhead thanks to its distributed and asynchronous design. In addition, HRM has been recently extended, without sensibly increasing its overhead, with the ability to provide heart rate measurements on multiple windows at the same time. The availability of such information poses a challenge for research, calling for more intelligent adaptation policies able to understand the correct time scale to consider and to take proactive actions to match applications' goals.

2.1. Black Box View

We can consider HRM as a black box implementing a producer/consumer model similar to the one employed by the *Performance and Environment Monitoring* (PEM) [11]: *producers* emit *heartbeats* for signaling progress and *consumers* access the *heart rates* computed by the monitor. HRM acts as an interface, collecting heartbeats, transforming them in throughput measurements, and making them available, realizing the *observation* phase of the *Observe, Decide, Act* (ODA) adaptation loop. Figure 1 represents this black box view, highlighting the flow of information from producers to consumers through, going through HRM, which enables the realization of the ODA adaptation loop.

To provide flexibility and be useful in current and future parallel systems, HRM must support monitoring any kind of parallel workload (i.e., multi-threaded, multi-processed, or any feasible mix of the two); this is attained by defining *monitoring groups* (marked as $\mathbf{G}_i$ in Figure 1). A *group* is a set of tasks cooperating for a certain activity (e.g., encoding video frames) and it constitutes the atomic monitoring entity.

Throughput measurements are computed as heart rates, i.e., for each group, the summation of the heartbeats emitted by all the producers over the elapsed time. Clearly, for such a measurements, the considered time horizon matters: considering the whole execution time provides a smoothed average, while considering a shorter time span discards the old history and allows to better highlight short-term trends. For this reason, HRM provides both a *global* and a *window* heart rate, allowing tuning the focus on longer or shorter-term trends as required by the specific monitoring context. Moreover, several measures on windows of different size (i.e. moving averages on different horizons) can be accessed at the same time to highlight different trends and providing richer information to consumers.

HRM allows for a simple yet general way of setting a desired value for the heart rate of a group through two parameters: a *minimum* and a *maximum* heart rate, defining the desired throughput range; moreover, it is possible to tie the goal to a specific window heart rate. For instance, dealing with a video encoder, the minimum heart rate could be set to the minimum frame rate to guarantee the desired QoS (e.g., 30 $\frac{frames}{second}$), the maximum could be set to a value over which no sensible benefit would be achieved and the goal could be tied to a certain time horizon according to how much buffering space is available for the encoded video.

Interaction with HRM is provided through a simple API implemented by *libhrm*: instrumenting an application needs as little as adding a couple of calls for attaching to a group and emitting heartbeats. Consumers are provided with a simple and powerful API, which was extended to support the new features. Details on the API are skipped here due to space constraints.

2.2. Under the Hood

The implementation of HRM has been partitioned between user and kernel-space. Partitioning the implementation lowers the overhead due to heartbeats emission while allowing the design of both user and kernel-space adaptation policies (i.e., consumers). The user-space partition is essentially an implementation of *libhrm*. The management logic, which handles grouping and logging, is implemented in kernel-space. Communication among different address spaces is

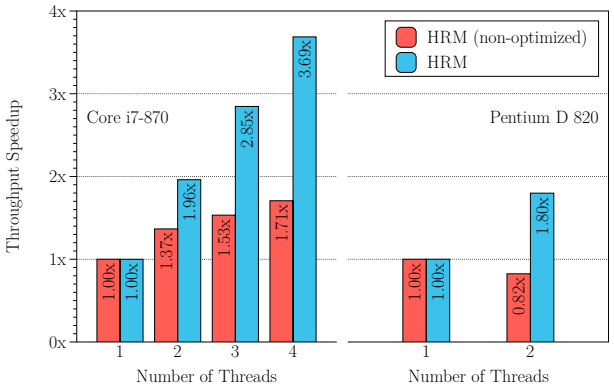


Fig. 2. Throughput speedup in emitting heartbeats when scaling the number of concurrent producers per group.

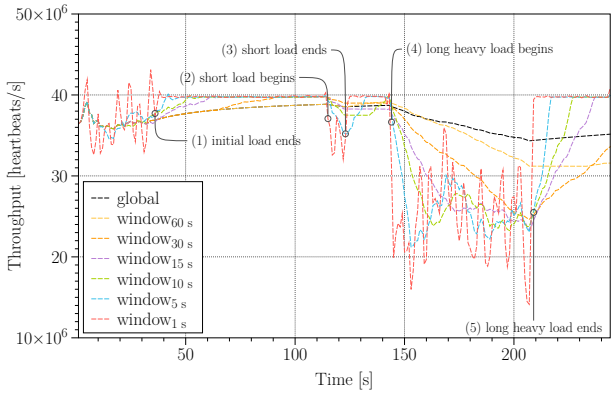


Fig. 3. Global and six different window heart rates of an ad-hoc application showing different performance trends.

enabled through shared memory, which grants low-overhead accesses from each side. For each group, the kernel allocates memory to store the information; memory segments are mapped in the address space of producers and user-space consumers upon group attachment. This way, all the entities of the group can access (with proper read/write privileges) the information. Sharing memory among different address spaces (and even within the same address space) is a delicate practice. Requiring carefully laid out data structure to achieve high efficiency. Since the most frequent operation within a group is emitting a heartbeat, the associated code path must be thoroughly optimized. This is done with a mapreduce-like approach, decoupling heartbeats emission and data computation (i.e., heart rates). Each producer receives counter within group memory to store the amount of heartbeats generated. Heartbeats emission becomes as quick as the increment of an atomic integer. Snapshots of the emitted heartbeat counts for all the active windows are periodically (with tunable period, defaulting at $100ms$) made available to allow on-demand heart rates computations. This way, heart rates are represented as floating point numbers in user-space and integer numbers in kernel-space, where floating point computation is discouraged. When a consumer asks for the global heart rate, such measurement is computed according to Equation 1. The window heart rate is otherwise computed according to Equation 2.

$$ghr_g(t) = \frac{\sum_i cnt_i(t)}{t - t_0} \quad (1)$$

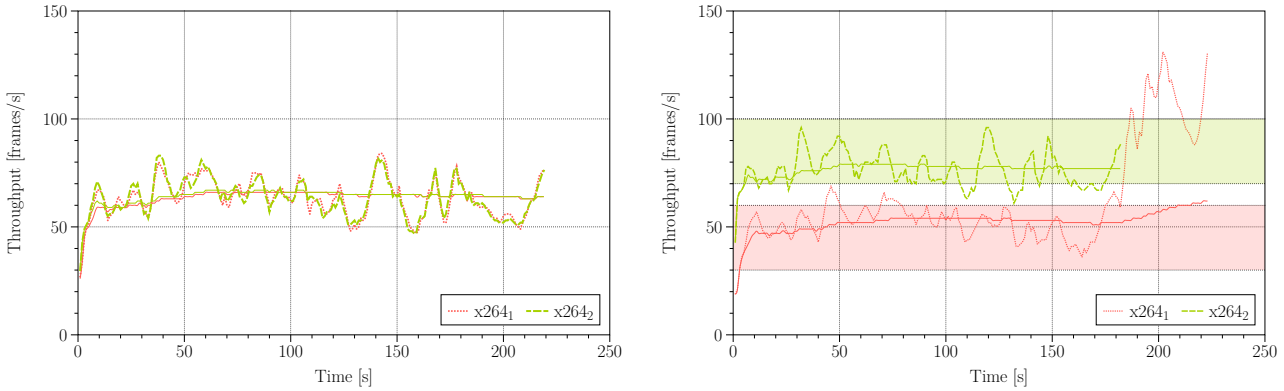
$$whr_g(t) = \frac{\sum_i cnt_i(t) - cnt_i(t - t_w)}{t - t_w} \quad (2)$$

In the formulae, t indicates the current timestamp, t_0 is the time at which the group was created, cnt_i are the counters associated with each of the group's producers, and t_w indicates the timestamp at the beginning of the window. To compute the window heart rates, HRM uses a circular buffer to store, at each accounting period, a snapshot of the current overall heartbeats count for the group and the timestamp.

This approach requires a careful implementation to avoid pitfalls resulting in poor performance. The memory location of each counter must be cache line-aligned to avoid false sharing, which would cause useless cache coherency traffic [12]. Figure 2 shows the speedup on throughput that can be achieved going scaling the number of producers emitting heartbeats in a tight loop for the same group. The test compares the optimized (final) revision with the non-optimized (i.e., non cache-friendly) revision of HRM on different processors. On the left side, the test is run with one to four concurrent producers executing on a quad-core Intel Core i7-870 processor with the Intel Hyper-Threading Technology disabled; both the non-optimized revision and the optimized revision of HRM scale. However, the latter scales almost linearly with the number of producers since it avoids false sharing. The On the right side, the same test is run with one to two concurrent producers executing on a dual-core Intel Pentium D 820 processor; the optimized revision of HRM scales almost linearly while the non-optimized revision of HRM shows a slow down when two producers emit heartbeats together for the same group. The slow down is due to the false sharing problem, which causes a notable performance decrease due to the off-chip (i.e., through the northbridge), inefficient cache coherency protocol of the Intel Pentium D processor.

3. SHOWCASE AND CASE STUDY

HRM has been employed within the *Metronome* [6] framework to measure applications' throughput, information that is later used to adapt process scheduling. Previous work proved the *efficiency* of HRM compared to the reference implementation of Application Heartbeats [13]. This is an incremental work extending the *functionality* of HRM with multiple window heart rates, a novel feature not yet adopted in similar contexts. Figure 3 presents a showcase of this capability: a 4-threaded microbenchmark is run to emit heartbeats as fast as possible on a quad-core Intel Core-i7 870 processor while many workloads differing in both duration and intensity run simultaneously. CPU-bounded workloads are simulated through the *cpuburn* utility. HRM is used with the multi-window capability to highlight performance



(a) Unmanaged instances of x264.

(b) Managed instances of x264; goals set at $30-60$ and $70-100\,\frac{frames}{s}$.

Fig. 4. Global and window heart rates for the x264 instances scheduled by the CFS (a) and by the adaptive scheduler (b).

trends and hence workloads' phases. In absence of additional workloads, the microbenchmark peaks at about $40 \times 10^6\,\frac{heartbeats}{s}$. The traces on the plot track the global heart rate and six different moving averages of size $\{1, 5, 10, 15, 30, 60\}s$[1]. The execution presents six phases: initially, up to the point marked (1), there is a light additional load which then terminates, letting the benchmark reach its peak performance up to point (2), when another external load is started. At point (3) the second load terminates and the microbenchmark goes back to its peak throughput but, at point (4), a heavier and longer-lasting load is applied up to point (5). It can be noticed from the plot how measurements on different time horizons highlight different trends: short windows give a prompt feedback when changes happen; however, they tend to be noisy when the execution is regular.

3.1. Adaptive Performance-Aware Scheduling

We evaluated the enhanced revision of HRM within the *Metronome* [6] framework. HRM monitors two 4-threaded instances of the *x264* application [14] encoding the *Big Buck Bunny* full HD movie [15]. The test platform is a quad-core Intel Core-i7 870 processor with the Intel Hyper-Threading Technology disabled running a modified version of Debian GNU/Linux [9]. The window size has been empirically set to $5s$ after an experimental evaluation using the multi-window capability of the latest revision of HRM. Figure 4 shows the results of this experiment: since the two instances are exactly the same, they have almost overlapping performance when scheduled by the Completely Fair Scheduler (CFS), as shown in Figure 4(a). The experiment consists in setting two different high-level performance goals for the two instances and let the adaptation policy implemented within the *Metronome* framework dynamically allocate processor time to match the performance goals. Figure 4(b) shows the managed case: the performance goals (i.e., the red and green shaded areas), are $30-60\,\frac{frames}{second}$ and $70-100\,\frac{frames}{second}$ and the two instances of *x264* are driven towards meeting their SLAs. The throughput of the slower application receives a sudden speedup when the other terminates, since it is now the only application in execution and the maximum heart rate is considered as a *soft bound* on the QoS, and not as a performance cap.

[1]Note that when there is not enough data to compute a window heart rate over its full size, the measure is still provided using the available data.

4. CONCLUSIONS

HRM proved to be a flexible, efficient, and scalable throughput monitor and was employed for realizing adaptive computing. This paper offers a detailed description of the careful design of HRM, which allows to provide very small overhead, and provides a showcase of a novel feature: the availability of measurements on multiple tunable moving averages. We believe that this feature could be exploited by smarter adaptation policies able to leverage this richer status information in order to take more effective adaptation decisions.

5. REFERENCES

[1] S. H. Fuller *et al.*, "Computing Performance: Game Over or Next Level?" *Computer*, vol. 44, no. 1, 2011.

[2] H. Esmaeilzadeh, *et al.*, "Dark silicon and the end of multicore scaling," in *Proc. of ISCA*, 2011.

[3] J. O. Kephart *et al.*, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003.

[4] M. Salehie *et al.*, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, 2009.

[5] M. C. Huebscher *et al.*, "A survey of Autonomic Computing: Degrees, Models, and Applications," *ACM Comput. Surv.*, vol. 40, no. 3, 2008.

[6] F. Sironi, *et al.*, "Metronome: Operating System Level Performance Management via Self-Adaptive Computing," in *Proc. of DAC*, 2012.

[7] H. Hoffmann, *et al.*, "Application Heartbeats for Software Performance and Health," in *Proc. of PPoPP*, 2010.

[8] ——, "Application Heartbeats: a Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *Proc. of the ICAC*, 2010.

[9] "The Linux Kernel," http://www.kernel.org/.

[10] "The FreeBSD Project," http://www.freebsd.org/.

[11] C. Cascaval, *et al.*, "Performance and Environment Monitoring for Continuous Program Optimization," *IBM J. Res. Dev.*, vol. 50, no. 2/3, 2006.

[12] M. M. K. Martin, *et al.*, "Why On-Chip Cache Coherence is Here to Stay," *Commun. ACM*, vol. 55, no. 7, 2012.

[13] "Application Heartbeats," http://code.google.com/p/heartbeats/.

[14] "x264 - H.264/ACV encoder," http://www.videolan.org/developers/x264.html.

[15] "Big Buck Bunny," http://www.bigbuckbunny.org/.