Design methodology for partial dynamic reconfiguration: a new degree of freedom in the HW/SW codesign

Marco D. Santambrogio, Donatella Sciuto

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Via Ponzio 34/5
20133 Milano, Italy

{santambr,sciuto}@elet.polimi.it

ABSTRACT

Many emerging products in communication, computing and consumer electronics demand that their functionality remains flexible also *after* the system has been manufactured and that is why the reconfiguration is starting to be considered into the design flow as a new relevant degree of freedom, in which the designer can have the system autonomously modify its functionalities according to the application's changing needs. Therefore, reconfigurable devices, such as FPGAs, introduce yet another degree of freedom in the design workflow: the designer can have the system autonomously modify the functionality carried out by the IP core according to the application's changing needs while it runs. Research in this field is, indeed, being driven towards a more thorough exploitation of the reconfiguration capabilities of such devices, so as to take advantage of them not only at compile-time, i.e. at the time when the system is first deployed, but also at run-time, which allows the reconfigurable device to be reprogrammed without the rest of the system having to stop running. This paper presents emerging methodologies to design reconfigurable applications, providing, as an example the workflow defined at the Politecnico di Milano.

1. A BRIEF INTRODUCTION

The concept of reconfigurable computing has been around since the 1960s, when Gerald Estrin, [1], a computer scientist at the University of California, Los Angeles, proposed the concept of a computer consisting of a standard processor augmented by an array of *reconfigurable hardware*. The basic idea was to use the array of reconfigurable hardware to perform a specific task and once the assigned task was completed the hardware could be reconfigured to perform a new task.

Before considering the specific area of reconfigurable computing we would like to try to define what *reconfiguration* means. [2] provides a first *interesting* definition:

Definition 1.1. ***Reconfiguration:*** *The process of physically altering the location or functionality of network or system elements. Automatic configuration describes the way sophisticated networks can readjust themselves in the event of a link or device failing, enabling the network to continue operation.*

Reconfiguration is an intuitive concept that is based on two main elements: a system and a behaviour, in this document we will use behaviour or functionality as synonyms.

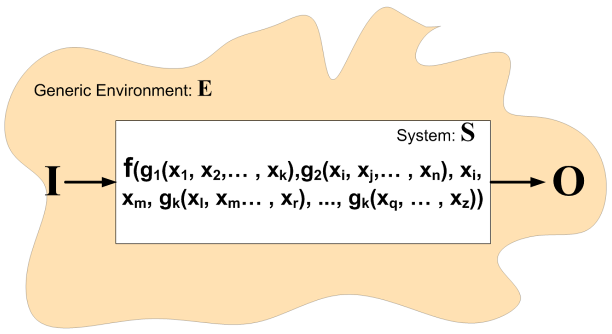
Generic Environment: E
System: S
I
O
f(g₁(x₁, x₂,… , xₖ),g₂(xᵢ, xⱼ,… , xₙ), xᵢ, xₘ, gₖ(xₗ, xₘ… , xᵣ), …, gₖ(x_q, … , x_z))

Figure 1: An overview of a generic system S

Figure 1 presents a generic system S, characterized by a functionality f, working in a environment E. S can interact with E through its interface which is composed of a set of inputs, I, and output O. According to this scenario a reconfiguration means the ability to change the original functionality f in S with a new one, i.e. g, that can also produce, but it is not necessary, a transformation $\rho : f \longrightarrow f^*$ in the I and in the O in S.

2. OVERVIEW ON RECONFIGURABLE DESIGN

In the traditional design workflow for embedded systems, the application was implemented entirely in hardware. However, due to the increasing spread and capabilities of general purpose processors suitable for use in such systems, the traditional methodology has been, over the past two decades, gradually substituted by another approach, in which a specific application can obtain better cost/performance tradeoffs by only using custom hardware for the most computationally demanding functionalities. The other components of the application are implemented as software tasks running on a general purpose CPU. In this scenario, which gives foundation to the area of Hardware-Software Co-Design, these two parts have to be designed jointly for a given application.

One alternative being considered is based on the technology of field programmable gate arrays, FPGAs. It should be obvious that every application would be best served by custom circuitry targeted specifically for it; and, in fact, ASICs are often made in response to special needs. But no one can afford to turn out a custom chip for every application he wants to run. As technology has improved, a market has grown up instead for versatile off-the-shelf parts that can be programmed to emulate arbitrary digital circuits in place of ASICs. FPGAs are one class of such devices, distinguished by their ability to be reprogrammed (reconfigured) any number of times. Therefore, reconfigurable devices, such as FPGAs, introduce yet another degree of freedom in the design workflow: the designer can have the system autonomously modify the functionality carried out by the IP-Core[1] according to the application's changing needs while it runs. Research in this field is, indeed, being driven towards a more thorough exploitation of the reconfiguration capabilities of such devices, so as to take advantage of them not only at compile-time, i.e. at the time when the system is first deployed, but also at run-time, which allows the reconfigurable device to be reprogrammed without the rest of the system having to stop running. During the past 6-8 years, we have seen reconfigurable logic emerge as a commodity technology comparable to memories and microprocessors. Like memories, reconfigurable logic arrays rapidly adapt to new technology since the design consists of an array of simple structures. Furthermore, again similar to memories, their design regularity allows designers to focus on adapting the key logic structures to extract the highest performance from the available technology.

The versatility and reprogrammability of FPGAs comes at a price. Only a few years ago, the algorithms that could be implemented on a single FPGA chip were fairly small. In 1995, for example, the largest FPGAs could be programmed for circuits of about $15,000$ logic gates at most. Since a fast 32-bit adder requires a couple hundreds of gates, the capabilities of such devices were somewhat bounded. More recently, though, FPGAs have reached a size where it is possibile to implement reasonable sub-pieces of an application in a single FPGA part. The incorporation of reconfigurable array logic into a microprocessor provides an alternative growth path which allows application specialization while benefiting from the full effects of commoditization. Like modern reconfigurable logic arrays, a single microprocessor design can be employed in a wide variety of applications. Application acceleration and system adaptation can be achieved by specializing the reconfigurable logic in the target system or application [3]. This has led to a new concept for computing: if a processor were to include one or more FPGA-like devices, it could in theory support a specialized application specific circuit for each program, or even for each stage of a program's execution. The unlimited reconfigurability of an FPGA permits a continuous sequence of custom circuits to be employed, each optimized for the task of the moment. Because FPGAs scale better than superscalar techniques, such designs have the potential to make better use of continuing advances in device electronics in the long term [4].

Reconfigurable systems, while providing new interesting features in the field of hardware/software co–design, also introduce new problems in their implementation and management. This is particularly true for systems that implement self partial reconfiguration [5–10]. In *partial* reconfiguration, only *portions* of the reconfigurable device are involved by the configuration change. *Dynamic* reconfiguration allows the device portions that are not directly involved in the reconfiguration to run without interruption through the reconfiguration process. A commonly adopted approach is the definition of predetermined area portions on the device, *reconfigurable slots*, in which components implementing different tasks from the specification, or *modules*, can be configured.

In the simplest scenario, that can be termed Compile Time Reconfiguration (CTR), the configuration of the FPGA is loaded at the end of the design phase, and it remains the same throughout the whole time the application is running. In order to change the configuration one has to stop the computation, reconfigure the chip resetting it, and then start the new application. CTR was for some years the only kind of reconfiguration available for FPGAs. With the evolution of technology, though, it became possible to considerably reduce the time needed for the chip reconfiguration: this made it conceivable to reconfigure the FPGA *between* different stages of its computation, since the induced time overhead could be considered acceptable. This process is called Run Time Reconfiguration (RTR), and the FPGA is said to be *dynamically reconfigurable*. RTR can be exploited by creating what has been termed *virtual hardware* [11, 12] in analogy with the concept of *virtual memory* in general computers. Consider for instance an application that is too big to fit into a particular FPGA: one can *partition* it into n smaller tasks, each one fitting on the chip. Then it is possible to load task 1 on the chip, execute it, then reconfigure the FPGA for task 2 and execute it, and so on till task n is finished. This idea is called *time partitioning*, and has been studied extensively in literature (see [13–16]).

A further improvement in FPGA technology allows modern boards to reconfigure only *some* of the logic gates, leaving the other ones unchanged. This *partial reconfiguration* is of course much faster in case only a small part of the FPGA logic needs to be changed. When both these features are available, the FPGA is called *partially dynamically reconfigurable*. Although there are several techniques to exploit partial reconfiguration (e. g. [6, 17, 18]), there are only a few aprroaches for frameworks and tools (e. g. [8, 10, 17–19]) to design dynamically reconfigurable System on Programmable Chip,SoPC (e. g. [20–22]). Examples of such frameworks are the operating systems for reconfigurable embedded platforms which have been analyzed in [23]. In [24] the authors have presented a run-time system for dynamical on-demand reconfiguration. Several research groups, [6, 25–31] have built reconfigurable computing engines to obtain high application performance at low cost by specializing the computing engine to the computation task; some preliminary results can be found in the literature, [7, 28, 32–36], but no general framework and no publicly available tools are, at the best of our knowledge, available.

Due to capabilities described above, FPGAs can be used to create hardware/software platforms that keep their flexibility at runtime, allowing the development of SoPCs. Modern FPGAs can also contain a general-purpose processor,

[1]An Ip-Core is defined as a core described using a HD Language (i.e., VHDL or verilog) combined with its communication infrastructure (i.e. the bus interface)

which can be both a physical CPU embedded in the FPGA fabric, or a soft core, mapped to a part of the FPGA. In both scenarios there is a software (SW) application running on the processor (or multiple processors) which realizes some of the system functionalities, usually acting also as a controller for the hardware (HW) components and interfacing with them. The SW part of a reconfigurable system can be either a standalone code, dealing directly with HW at a low level, or a complete operating system, including multiprocessing and resource scheduling.

A standalone code is usually an application which uses SW libraries exporting functions to interface with HW components. This approach can be acceptable for small systems, involving few components and configurations, but as soon as the complexity of the system increases, it becomes more difficult to develop a complete application dealing with all those aspects. The use of an operating system allows more flexibility on both sides, since it is possible to implement the SW part as one or multiple *userspace* processes, introducing complex inter-process communication systems and scheduling techniques. HW management is performed by the OS, offering the processes an interface to access system peripherals at a higher level of abstraction. The counterpart for this added flexibility is the necessity to add support to a standard operating system for reconfiguration-specific HW and for reconfigurable components. This means that the HW and SW parts of the system must be designed in order to allow the creation of a reconfigurable architecture.

3. THE PROPOSED METHODOLOGY

Aim of this work is to define a methodology and design flow for reconfigurable embedded systems. The proposed methodology aims at defining a specification-to-bistream and autonomous design flow based on, where possible, standard tools. The idea behind the proposed methodology is based on the assumption that it is desirable to implement a flow that can output a set of configuration bitstreams used to configure and, if necessary, partially reconfigure a standard FPGA to realize the desired system.

One of the main strengths of the proposed methodology is its low-level architectural independence. In fact it has been developed using both the Caronte [37–39] and the YaRA (Yet another Reconfigurable Architecture) architecture (developed at Politecnico di Milano), but it can be easily adapted to different architectural and SoC solutions, i.e. the RAPTOR2000 system [40]. In particular both the Caronte and the YaRA solutions share the same structure that consists of two distinct parts: a **static/fixed part** containing all the components that have to be present permanently in the final system or that are used very frequently; a **reconfigurable part** used to hold the reconfigurable components that are dynamically plugged into the system during the computation phase. Figure 2 presents the physical implementation of these architectures, in particular the YaRA solution. Starting from the bottom (figure 2 refers to YaRA implemented over a Virtex II Pro VP7 with a single reconfigurable module) the first layer contains BRAM memories and Power-PC 405, while the second layer contains TBUFs and TBUFLINE. The upper one contains SLICEs and Switch Matrices and consequently all user logic, but also the CoreConnect components are implemented at this level. Finally,

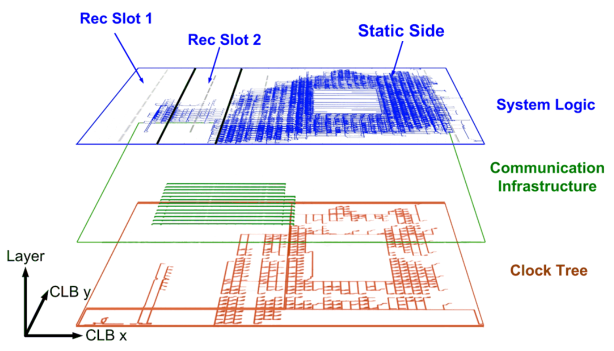

Figure 2: YaRA architecture at the FPGA level

the last layer is the clock level that is, according to Xilinx Documentation [17, 41], routed at a different level with respect to the other signals. This figure explains why a reconfiguration process involving a module does not abort communication between other modules: it is due to the fact that the communications go through TBUFLINEs that are not involved and modified by the reconfiguration process.

Finally, the software side can be developed either as a standalone application or with the support of an Operating System, such as Linux. Even if the standalone approach can be optimized for each particular scenario to improve timing performance, it reduces the flexibility of the whole system, so it is usually desirable to employ a standard Operating System. Under this assumption, the standard Linux OS has been enhanced with the addition of support for reconfigurable hardware modules ([21]), nevertheless it is possible to extend this work to support other Operating Systems as well. This solution also helps to increase code reuse.

The proposed design flow [42, 43] consists mainly of three phases, as shown in Figure 3:

- **High Level Reconfiguration, HLR**
  The goal of *High-Level Reconfiguration* is to analyze the input specification in order to find a feasible representation, produced by a first partitioning (cores / functionalities identification) phase, that can be used to perform the hardware/software codesign. In the currently implemented framework, cores are identified by extraction of isomorphic templates used to generate a set of feasible covers of the original specification. Then, the computed cover is placed and scheduled onto the given device.

- **Validation, VAL**
  Aim of the *validation* phase is to drive the refinement cycle of the system design. Using the information provided by this phase, it is possible to modify the decisions taken in the first part of the flow to improve the development process.

- **LLR** (Low Level Reconfiguration)
  The last step that has to be performed is the *low-level reconfiguration* phase. Goal of this step is the definition of an automatic generation of the low-level implementation of the final solution that has to be physi-

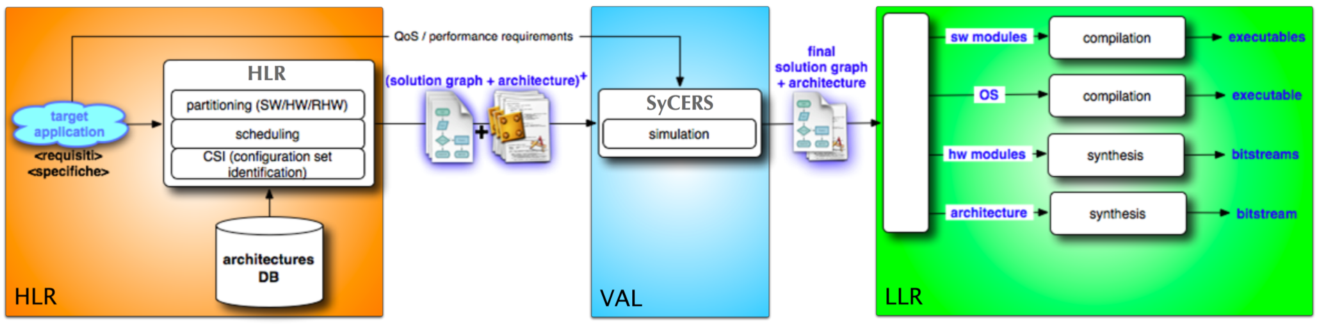

Figure 3: The proposed design flow.

cally deployed on the target device and that realizes the original specification.

3.1 High Level Reconfiguration

In this section we introduce the *High Level Reconfiguration* (HLR) component of the flow: the definition of a methodology and workflow to partition [44,45] a system specification into a task graph (in which every task is to be treated as a reconfigurable module) – tailored to the implementation on a partially dynamically reconfigurable architecture – and to schedule it on the same reconfigurable architecture. Figure 4 offers a bird-eye view of its structure, showing the different stages from the original high-level specification to the partitioned system (i.e., cores which are ready to be scheduled for configuration and execution).

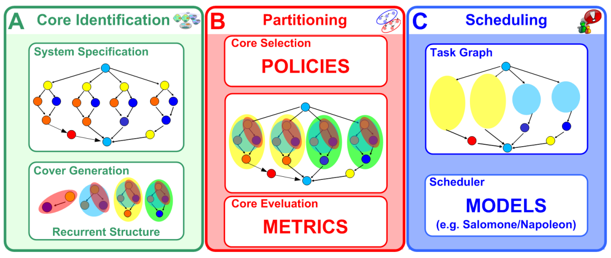

Figure 4: Overview of the High-Level Reconfiguration flow.

The first step, A in Figure 4, is the **core identification** phase. The input of this step is the original specification, whose analysis results in *cores*, i.e. groups of operations that, reconfigured together as configurable modules, have *optimal* sizes (they almost fill an integer number of columns).

The second part of HLR, B in Figure 4, is the **Partitioning** phase [46]. Using the previously computed set of cores as its input, this phase produces a set of feasible covers of the original graph of the specification, following a given policy. In fact, the previous core identification phase might or might not produce a cover of the graph, i.e. there may be parts of the specification which are not included in the union of the identified cores – or there may be alternative collections of cores to choose from. This possibility is represented by the *demultiplexer-like* block in Figure 4: depending on the chosen policy, the output might be ready to be passed on to the scheduling and allocation phase, or not. If not, we take the top arrow, which leads to three phases: Evaluation, Core Choice and Partitioning.

During the *Evaluation* phase, we have a collection of possible cores to choose from to implement the reconfigurable system. The decision to implement the cores as static or reconfigurable hardware modules is left to the later scheduling and allocation phase. During the *Core Choice* phase, instead, using the data from the Evaluation phase, the cores to be implemented in the system are determined, according to different possible policies. At this point, the cores have been selected. However, there might still be parts of the specification which are not included in the union of the chosen cores. We thus have the *Partitioning* phase, which copes with this problem and takes into account the whole specification by creating new subsets which include the still uncovered operations.

The generated graph, called *Task Dependency Graph* (TDG), is provided as input to the following phase, i.e. the **scheduling phase** (C in Figure 4), which generates the complete schedule, driven by predefined policies [44,45,47].

3.2 Validation

The *Validation* phase defines a novel approach to simulate and validate dynamically reconfigurable IP-Core based architectures. The proposed framework, SyCERS ([48]), is based on the SystemC class library and targets the design of architectures for embedded systems, including reconfigurable ones. The use of the SystemC library enables the designer to specify a system in its hardware and software parts with just one language. SystemC enables the co-development software and hardware, so it is possible to describe a complete hardware/software system in a single specification.

The SyCERS framework is built on top of the SystemC library and allows the specification of both architecture and system models. Classes implementing architectures have to be derived from a set of common interfaces provided in the framework. In addition, an architecture is built as a C++ library, so it can be imported in any system design. The system models use the interfaces implemented in the architecture to access the hardware resources; therefore a model can be tested in any architecture implementing the same interface set. A system model can be also parameterized so that resources, like numbers of available reconfigurable blocks and memory size, can be changed before execution. This parameterized model allows testing a system on various solutions. Finally, the SystemC model of the system, can be executed in the standard OSCI simulator or even in the ModelSim commercial simulator [49].

Once a system has been modeled in SystemC and simulated in SyCERS it is possible to obtain several pieces of information useful to the designer in deciding which is the best solution to implement his/her embedded system via the final phase, the Low-Level Reconfiguration.

3.3 Low Level Reconfiguration

Aim of the *Low-Level Reconfiguration* phase is to generate the low-level implementation of the desired system necessary to physically configure the target device to realize the original specification.

To develop the system it is necessary to split the Low-Level Reconfiguration in three parts: the hardware, the reconfigurable and the software sides. On one hand the first steps that have to be performed in the hardware and reconfigurable hardware sides are *Core Design* and the *IP-Core Generation* [50, 51], in which the core functionalities of the original specification are translated in a hardware description language and extended with a communication infrastructure that makes it possible to interface them with a bus-based communication channel. The resulting IP-Core structure is represented in Figure 5 After these steps, the static

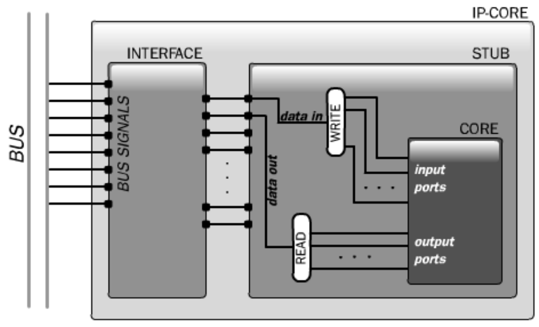

Figure 5: IP-Core final structure.

components of the system are used to realize the YaRA architecture, while the reconfigurable components are handled in a different way, as reconfigurable IP-Cores; in other words they will be kept separated from the static part of the architecture during the *DEsign SYnthesis* phase, while the static components will be synthesized together with the static part of the architecture. On the other hand, in the software part there is the need to develop, in addition to a control application that is able to manage the reconfiguration tasks, also a set of drivers to handle both the reconfigurable and the static components of the system. All these software applications are compiled for the processor of the target system. The compiled software is then integrated, in the *Software Integration* phase, with the bootloader, with the Linux OS and with the Linux Reconfiguration Support [52, 53], that extends a standard OS with the capability of both performing reconfigurations of the reprogrammable device and managing the reconfigurable hardware as well as the static hardware, in order to allow runtime plug-in of components. The following step is the *Bitstreams Generation* which is necessary to obtain the bitstreams that will be used to configure and to partially reconfigure the reprogrammable device. Finally, the last step of the LLR process is the *Deployment Design* phase, that aims at creating the final solution, that consists of the initial configuration bitstream, the partial bitstreams, the software part (bootloader, OS, Reconfiguration Support, drivers and controller) and the deployment information that will be used to physically configure the target device.

4. ADAPTIVE COMPUTING FOR THE SOFTWARE COMPONENT

The proposed methodology has been applied to the problem of the identification of the software and the hardware components of a complex dynamically reconfigurable SoC, introducing an adaptive computation approach. Adaptivity implies that due to input changes the output of the system is updated only re-evaluating those portions of the program affected by the changes.

As dynamic reconfiguration introduces more flexibility into the hardware side, adaptive computation could introduce different behaviors into the software side. Combining these two techniques together enables a new design scenario in which hardware and software are moving closer towards each other reshaping the overlapping gray space. Aim of the work proposed in [54] was to identify the best trade-off considering application-specific features in software, which can lend itself to software-based acceleration and lead to a revision of the view that certain computationally intensive tasks can only be accelerated through hardware.

We propose a new methodology, based on the Adaptive Programming [55] technique, to evaluate and subsequently perform the hardware and software partitioning for a SoC that employs dynamically reconfigurable hardware and software programmable cores. The adaptive computation concept which we utilize in our realization of the software partitions allows the evaluation of the performance of software execution as a non-static entity. Adaptive computing defines a relationship between the input and output of an application with respect to the input changes [55, 56]. An adaptive program responds to input changes by updating its output, only re-evaluating those portions of the program affected by the change. Adaptive programming is particularly beneficial in situations where input changes lead to relatively small changes in the output. In some cases one cannot avoid a complete re-computation of the output, but in many cases the results of the previous computation may be re-used to obtain the updated output more quickly than a complete re-evaluation. Previous studies of purely software-based systems [57], indicated encouraging performance improvements. For example, the execution time of the main procedures used in computational geometry algorithms have been reduced by up to 250 times.

In such a context, the main innovation of our technique lies primarily in the way we view and evaluate the software partition. The basic philosophy is the following. If the input to a program is not expected to change significantly over different executions, one can exploit this by introducing the *self-adjusting* property into the program such that those computations which do not change across different input sets can be reused instead of being re-executed. This concept has been introduced to exploit application specific properties in purely software-based systems in order to accelerate execution time by up to three orders of magnitude for various applications [54, 56, 57]. We aim at adapting this paradigm into a mixed hardware and software design flow for reconfigurable SoCs. Our goal is to develop a new performance model and an associated evaluation metric to identify application specific input behavior thereby differentiating between various levels of performance across different portions of software modules. This general performance model is then embedded along with hardware performance models into our proposed environment, which will yield a highly flexible means to evaluate the performance impact of different partitioning and allocation decisions.

5. FUTURE TRENDS

The inherent advantages of hardware over analogous software solutions (computational speed, inherent parallelism, device size) would make it possible to apply reconfigurable and adaptive computing to systems such as: biomedical implants (think of an artificial art control), telecommunications (think of adaptive intelligent routers), intelligent nanorobot control, artificial audio and vision, intelligent transducers at bio-electronic interfaces.

It is possible to envision for reconfigurable technologies the possibility to move from the prototyping and very specialized low-volume arenas to the implementation of **real-world** systems capable of adapting their behavior and resources thousands times a second, according to the surrounding environment evolution. This capability would widen the horizons of embedded digital system applications. Some of the main challenges towards such scenarios would be:

1. to enable distributed self-reconfiguration capabilities of digital devices;
2. the implementation of distributed training algorithms over such architectures;
3. to specify and formulate application solutions in terms of such computing paradigm.

At the moment, potential benefits of massively reconfigurable digital systems in real-life applications is far beyond sight, therefore the definition of novel architecture can be considered as key point for the future of this research area. Potentially, continuously trained devices could be implemented with reconfigurable logic technologies, allowing small devices to stabilize, with proper actions, physical parameters depending on huge sets of factors; with proper training procedures implemented in a distributed way over the whole device, behavior could be specified as the maximization of desired target functions.

Reconfiguration would not be provided from the outside as an input to the device, but would be computed autonomously by the device itself, according to the target behavior and to the environment. This would make such devices particolarly suited for applications of pervasive computing/control of any kind, e.g., medical pilot plants, neurological control systems, adaptive communication infrastructures.

6. REFERENCES

[1] G. Estrin. Organization of computer systems–the fixed plus variable structure computer. *Proc. Western Joint Computer Conf., Western Joint Computer Conference, New York*, pages 33–40, April 1960.

[2] Netted Automation GmbH. The net is the automation. *http://www.nettedautomation.com*, 1994.

[3] A. DeHon. *DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century.*

[4] J. R. Hauser. Augmenting a microprocessor with reconfigurable hardware. In *A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy.*

[5] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A Self-reconfiguring Platform. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Proceedings of the Field Programmable Logic and Application, 13th International Conference, FPL 2003*, volume 2778 of *Lecture Notes in Computer Science*. Springer, 2003.

[6] H. Kalte, M. Porrmann, and U. Rückert. System-on-programmable-chip approach enabling online fine-grained 1D-placement. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*, Santa Fé, New Mexico, 2004. IEEE Computer Society.

[7] David E. Taylor, John W Lockwood, and Sarang Dharmapurikar. Generalized rad module interface specification of the field programmable port extender (fpx). *Washington University, Department of Computer Science. Version 2.0, Technical Report.*

[8] Edson L. Horta and John W. Lockwood. *Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs.* 2004.

[9] Andreas Weisensee and Darran Nathan. A self-reconfigurable computing platform hardware architecture, 2004.

[10] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in virtex fpgas. *Computers and Digital Techniques, IEE Proceedings-*, 153(3):157–164, 2006.

[11] Xiao ping Ling and Hideharu Amano. Performance evaluation of wasmii: a data driven computer on a virtual hardware. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE*, volume 694 of *Lecture Notes in Computer Science*, pages 610–621. Springer, 1993.

[12] W. Fornaciari and V. Piuri. Virtual fpga s: Some steps behind the physical barriers. In *IPPS/SPDP Workshops*, pages 7–12, 1998.

[13] João M. P. Cardoso. On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures. *IEEE Trans. Computers*, 52(10):1362–1375, 2003.

[14] J.M.P. Cardoso and H.C. Neto. Compilation for fpga-based reconfigurable hardware. *IEEE Design & Test of Computers*, 20(2):65–75, 2003.

[15] M. Kaul, R. Vemuri, S. Govindarqjan, and I. Ouaiss. An automated temporal partitioning and loop fission approach for fpga based reconfigurable synthesis of dsp applications. In *DAC '99*, pages 616–622. IEEE Computer Society, 1999.

[16] Joo M. P. Cardoso. Loop dissevering: A technique for temporally partitioning loops in dynamically reconfigurable computing platforms. In *IPDPS '03*, page 181.2. IEEE Computer Society, 2003.

[17] Xilinx Inc. Two Flows of Partial Reconfiguration: Module Based or Difference Based. Technical Report XAPP290, Xilinx Inc., November 2003.

[18] Xilinx Inc. *Early Access Partial Reconfiguration Guide*. Xilinx Inc., 2006.

[19] Gerard Habay Philippe Butel and Alain Rachet. Managing partial dynamic reconfiguration in virtex-ii pro fpgas. *Xcell Journal Online*, 2004.

[20] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A Self-reconfiguring Platform. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Proceedings of the Field Programmable Logic and Application, 13th International Conference, FPL 2003*, volume 2778 of *Lecture Notes in Computer Science*. Springer, 2003.

[21] Alberto Donato, Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. Operating system support for dynamically reconfigurable soc architectures. In *IEEE-SOCC*, 2005.

[22] Ryan J. Fong, Scott J. Harper, and Peter M. Athanas. A versatile framework for fpga field updates: An application of partial self-reconfiguation. *rsp*, 00:117, 2003.

[23] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Computers*, 53(11):1393–1407, 2004.

[24] Michael Ullmann, Michael Hübner, Björn Grimm, and Jürgen Becker. An fpga run-time system for dynamical on-demand reconfiguration. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[25] Jason Miller David Wentzlaff Fae Ghodrat Ben Greenwald Henry Hoffman Paul Johnson Jae-Wook Lee Walter Lee Albert Ma Arvind Saraf Mark Seneski Nathan Shnidman Volker Strumpen Matt Frank Saman Amarasinghe Michael Bedford Taylor, Jason Kim and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs.

[26] J. R. Hauser and J. Wawrzynek. *Garp: A MIPS Processor with a Reconfigurable Coprocessor*. IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[27] H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, March 1993.

[28] Mihai Budiu Srihari Cadambi Matt Moe Seth Copen Goldstein, Herman Schmit and R. Reed Taylor. Piperench: A reconfigurable architecture and compiler. In *Computer*.

[29] Guangming Lu Nader Bagherzadeh Fadi J. Kurdahi Eliseu M. C. Filho Ming-Hau Lee, Hartej Singh and Vladimir Castro Alves. Design and implementation of the morphosys reconfigurable computing processor. In *J. VLSI Signal Process. Syst.*

[30] Markus Koester, Heiko Kalte, and Mario Porrmann. Task placement for heterogeneous reconfigurable architectures. In *Proceedings of the IEEE 2005 Conference on Field-Programmable Technology (FPT'05)*, 2005.

[31] Heiko Kalte and Mario Porrmann. REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. In *Proc. of the ACM International Conference on Computing Frontiers*, 2006.

[32] John R. Hauser Timothy J. Callahan and John Wawrzynek. *The garp architecture and c compiler*, volume 33. Computer, 2000.

[33] Xilinx Inc. *The Programmable Gate Array Databook*. Xilinx Inc.

[34] Edson L. Horta, John W. Lockwood, and David Parlour. Dynamic hardware plugins in an fpga with partial run-time reconfigurtion. pages 844–848, 1993.

[35] Eliseu Filho Rafael Maestre Ming-Hau Lee Fadi Kurdahi Hartej Singh, Guangming Lu and Nader Bagherzadeh. Morphosys: case study of a reconfigurable computing system targeting multimedia applications. In *Proceedings of the 37th conference on Design automation (DAC00)*. ACM Press.

[36] Edson Horta and John W. Lockwood. Parbit: A tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas). *Washington University, Department of Computer Science, Technical Report WUCS-01-13*, July 2001.

[37] F. Ferrandi, M. D. Santambrogio, and D. Sciuto. A design methodology for dynamic reconfiguration: The caronte architecture. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Reconfigurable Architecture Workshop - RAW*, page 163, 2005.

[38] A. Donato, F. Ferrandi, M. Redaelli, M. D. Santambrogio, and D. Sciuto. Caronte: a complete methodology to implement partially dynamically self-reconfiguring embedded systems on modern fpga. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 321–322, 2005.

[39] A. Donato, F. Ferrandi, M. Redaelli, M. D. Santambrogio, and D. Sciuto. Exploiting partial dynamic reconfiguration for soc design of complex application on fpga platforms. In *13th International Conference on Very Large Scale Integration*, pages 179–184, 2005.

[40] Mario Porrmann Heiko Kalte and Ulrich Ruckert. A prototyping platform for dynamically reconfigurable system on chip designs. In Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC), 2002.

[41] *Virtex-II Pro Data Sheet Virtex-II Pro^{TM} Platform FPGA Data Sheet*. Xilinx Inc., 2003.

[42] M. D. Santambrogio and D. Sciuto. Partial dynamic

reconfiguration: the caronte approach. a new degree of freedom in the hw/sw codesign. In *16th International Conference on Field Programmable Logic and Applications*, pages 945–946, August 2006.

[43] V. Rana, M. D. Santambrogio, and D. Sciuto. Dynamic reconfigurability in embedded system design. In *IEEE International Symposium on Circuits and Systems*, pages 2734–2737, May 2007.

[44] F. Ferrandi, M. Redaelli, M. D. Santambrogio, and D. Sciuto. Solving the coloring problem to schedule on partially dynamically reconfigurable hardware. In *13th International Conference on Very Large Scale Integration*, pages 97–102, 2005.

[45] M. Giorgetta, M. D. Santambrogio, P. Spoletini, and D. Sciuto. A graph-coloring approach to the allocation and tasks scheduling for reconfigurable architectures. In *14th IFIP International Conference on Very Large Scale Integration - IFIP VLSI-SOC*, October 2006.

[46] M. Giani, M. Redaelli, M. D. Santambrogio, and D. Sciuto. Task partitioning for the scheduling on reconfigurable systems driven by specification self-similarity. In Toomas P. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 78–84. CSREA Press, June 2007.

[47] F. Redaelli, M. D. Santambrogio, and D. Sciuto. Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems. In *DATE 2008*, 2008.

[48] C. Amicucci, F. Ferrandi, M. Santambrogio, and D. Sciuto. Sycers: a systemc design exploration framework for soc reconfigurable architecture. In *The 2006 International Conference on Engineering of Reconfigurable Systems & Algorithm, June 26-29, Las Vegas - Nevada, USA*, June 2006.

[49] Mentor Graphics Corporation. *ModelSim User's Manual*. 2007.

[50] F. Ferrandi, G. Ferrara, R. Palazzo, V. Rana, and M. D. Santambrogio. Vhdl to fpga automatic ipcore generation: A case study on xilinx design flow. In *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06) - Reconfigurable Architecture Workshop - RAW*, April 2006.

[51] M. Murgida, A. Panella, V. Rana, M. D. Santambrogio, and D. Sciuto. Fast ip-core generation in a partial dynamic reconfiguration workflow. In *14th IFIP International Conference on Very Large Scale Integration - IFIP VLSI-SOC*, October 2006.

[52] A. Donato, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. Operating system support for dynamically reconfigurable soc architectures. In *7th International Symposium on System-on-Chip*, pages 235–238, 2005.

[53] V. Rana, M. D. Santambrogio, D. Sciuto, B. Kettelhoit, M. Koester, M. Porrmann, and U. Ruckert. Partial dynamic reconfiguration in a multi-fpga clustered architecture based on linux. In *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS'07) - Reconfigurable Architecture Workshop - RAW*, pages 1–8, March 2007.

[54] M. D. Santambrogio, V. Rana, S. Ogrenci Memik, and D. Sciuto. A novel soc design methodology combining adaptive software and reconfigurable hardware. In *25th International Conference on Computer-Aided Design*, page To appear, November 2007.

[55] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL*, pages 247–259, 2002.

[56] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *POPL*, pages 14–25, 2003.

[57] Umut A. Acar. Self-adjusting computation. In *PhD Thesis, School of Computer Science, Carnegie Mellon University*, May 2005.