Dynamic Reconfigurability in Embedded System Design

Vincenzo Rana, Marco Santambrogio, Donatella Sciuto
Politecnico di Milano
Dipartimento di Elettronica e Informazione
Via Ponzio 34/5
20133 Milano, Italy
{rana, santambr, sciuto}@elet.polimi.it

ABSTRACT

Nowadays, dynamic reconfigurable embedded systems are widely used, since they have the capability to modify their functionalities, adding or removing components and modify interconnections among them. The basic idea behind these systems is to have the system autonomously modify its functionalities according to the application's changes. This paper describes the area of reconfigurable embedded systems presenting both architectural and methodological aspects trying to point out common features and needs. After a brief introduction, an overview of the models, of the reconfigurable architectures, and of the design methodologies will be presented.

1. INTRODUCTION

Many emerging applications in the fields of (mobile) communications, computing, and consumer electronics require flexible and evolvable functionalities after the system deployment. In order to offer better computing capabilities, high-performance commercial reconfigurable architectures provide ample reconfigurable logic and, often, have also integrated a number of fixed components, including digital signal processing (DSP) and microprocessor cores, custom hardware, and distributed memory modules. Such reconfigurable architectures, integrated with distributed memory modules, exhibit superior computing capabilities, storage capacities, and flexibility over traditional FPGAs. The successful deployment of these novel embedded systems to the market requires the identification, formalization, and implementation of concepts, methods and tools for their design, considering, together with the traditional requirements, the possibility of dynamically reconfiguring themselves at runtime, over a set of predefined behaviors [1]. Emphasis is given on correctness and dependability of joining technologies in the hardware and software domains, on the reconfigurable hardware characteristic, on the satisfaction of real-time constraints and, in general, on the exploration of the solution space, to select the most effective solutions compatible with design and market constraints. Initially thought as a tool for rapid prototyping, FPGAs have lately increased in power and flexibility, offering a new option to the hardware/software dichotomy: faster than a pure software approach, less static than traditional hardcoded solutions and with better time-to-market. The success has been so big that nowadays it is not uncommon to even directly *deploy* FPGA–based solutions. In this scenario, the configuration of the FPGA is loaded at the end of the design phase, and remains unchanged the entire application run-time (e.g., [2], [3], [4]). The reconfiguration requires the system to be stopped reconfigured and reset (therefore called Compile-Time-Reconfiguration). 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 Real Time Reconfiguration *RTR*, (e.g., [5], [6]), and the FPGA is said to be *Dynamically Reconfigurable.* RTR can be exploited by creating what has been termed *virtual hardware* [7] in analogy with the concept of *virtual memory* in general computers. However, most real–life applications are simply too large to fit in the logic available on a single chip. In such a scenario it is possible to *partition* the desired application into a subset of n smaller tasks, each one fitting on the chip. The FPGA will be reconfigured at run-time to execute the various tasks: this idea has been named *time partitioning*, and has been extensively studied in literature [8–10]. A further improvement in FPGA technology allows modern boards to reconfigure only *some* of the logic gates. This *partial reconfiguration* is much faster in the common case where only a small portion of the FPGA needs to be changed. When both these features are available, the FPGA is called *partially dynamically reconfigurable.*

The paper is organized as follows: Section 2 presents an overview of the state of the art regarding models and reconfigurable architectures while Section 3.2 provides an overall view of three design methodologies. Finally, Section 4 ends this paper.

2. MODELS AND RECONFIGURABLE ARCHITECTURES

There are different models of reconfiguration, which can be classified according to the following scheme [11]: (i) *who controls reconfiguration*, (ii) *when configuration is generated* and (iii) *which is the granularity of the reconfiguration.* The *first subdivision* is between *external* and *internal reconfiguration.* In the first case, the reconfiguration is managed by an external entity, usually a PC. Internal reconfiguration, instead, is performed by the FPGA itself; for this to be possible, the device must have a physical component dedicated to reconfiguration, such as the ICAP component in Xilinx FPGAs. For what concerns the *configuration generation*, it can be done in a complete static way, at design time, determining all possible configurations of the system. Each module must be synthesized and all possible connections between modules and the rest of the system must be considered. Other possibilities are runtime placement of pre-synthesized modules, which requires dynamic routing of interconnection signals or complete dynamic modules generation. This last option is currently impossible, since it would require runtime synthesis of modules from VHDL (or other HW description languages) code, typically requiring very long processing times. Finally, *reconfiguration can involve different granularity levels*, depending on the size of the area reconfigured. The two typical approaches are *smallbits* and *module based*: the first consists in modifying a single portion of the design, such as single Configurable Logic Blocks (CLB) or I/O blocks parameters [12], while the second involves the modification of a bigger FPGA area. The module based approach consists in creating HW components (modules) that can be added and removed from the system each time a reconfiguration is applied. This requires the reconfiguration of entire FPGA areas, generally sets of columns, since in available FPGAs configuration can only be done on per-column basis. An evolution of the module based approach is the Early Access flow [13] where a new set of constraints to design self reconfigurable architectures, using Xilinx FPGAs, is presented. Several research groups, [14–16] have built reconfigurable computing engines to obtain high application performance at low cost by specializing the computing engine to the computation task. What seems to be neglected so far is the full exploitation of the ability to partially reconfigure the FPGA at runtime; some preliminary results can be found in the literature, [17–22], but no general framework and no publicly available tools are, at the best of our knowledge, available.

In the Garp project of the UC Berkeley, the FPGA is recast as a slave computational unit located on the same die as the processor [14]. Garp has been designed to fit into an ordinary processing environment that includes structured programs, libraries, context switches, virtual memory and multiple users. Athanas and Silverman introduce the PRISM (Processor Reconfiguration through Instruction-Set Metamorphosis) architecture which couples a programmable element with a microprocessor [15]. From each application, new processor *instructions* are synthesized in the reconfigurable element which are designed to accelerate the application.

The Washington University in the WUGS project, Washington University Gigabit Switch, has proposed a new methodology proposed in [19] that allows the platforms, by means of partial dynamic reconfiguration, to hot–swap application specific modules, called Dynamic Hardware Plugin (DHP). The presented application has been implemented onto a Xilinx Virtex-E FPGA. The PARBIT tool [21] transforms FPGA configuration bitstreams, representing the DHP modules, to enable them in the Field- programmable Port Extender, FPX, [20]. The tool accepts as input the original bitfile, a target bitfile and parameters given by the user. The output is a new bitstream that can load a DHP module into any region of the Reprogrammable Application Device, RAD on the FPX. The flexibility of this approach is limited by the constraints imposed by the predefined size and location of the reserved areas on the FPGA where the DHP's must be placed, reducing the overall logic utilization factor when DHP's with different sizes are employed, and by the need of an external reconfiguration device.

Reconfigurable computing can be considered as a close combination of hardware cores and of the run-time instruction set of a general purpose processor [23]. The classification of core types is generally accepted to be split into three classes [24]: Hard cores, Firm cores and Soft cores. In [25], a new class of cores called run-time parameterizable (RTP) has been introduced. RTP cores allow a single core to be computed and customized at run-time. The core produces all the required configuration data to define the logic and the routing. The possibility of determining limited amounts of routing at run-time is also dealt with in [25]. An innovation of this approach consists in considering the RTP cores as a specific example of a reconfigurable core, placed on the programmable device in a dynamic manner to respond to the changing computational demands of the application. The problem of this methodology is that the RTP cores are targeted only to a single device family and there is no information about the communication channel between RTP cores and on how they solve the physical reconfiguration problem.

3. DESIGN METHODOLOGIES

To develop a reconfigurable system it is possible to build an ad-hoc solution or to follow a generalized design flow. The first choice implies a considerable investment in terms of both time and efforts requested to build a specific solution for the given problem, while the second one allows to exploit the re-use of knowledge, cores and software to reach more rapidly a good solution to the same problem.

3.1 Adriatic and RECONF2

Aim of the *ADRIATIC* [26] project is to define a methodology able to guide the Codesign of reconfigurable SoC, with particular attention to cores situated in the wireless communication application domain. The first phase is the system specification, in which the functionality of the system can be described by using a high-level language program.This executable specification can be used to accomplish the following tasks: *generation* of the test-bench; *partitioning* of the application to specify which part of the system will be implemented in hardware (either static or dynamically reconfigurable hardware); *accurate definition* of the application domain and of the designer knowledge. To derive the final architecture from the input specification, the dynamically reconfigurable hardware has to be identified; each dynamically reconfigurable hardware block can be considered as a hardware block that can be scheduled for a given time interval. The partitioning phase defines, for each part of the system, the type of implementation: hardware, software or reconfigurable hardware block. To help in this decision, some general guidelines have been developed. In the mapping phase the functionalities defined by the executable specification are modified to obtain thorough simulation results. The *ADRIATIC* flow is a solution that can be easily applied to the system-level of a design. In this phase, in fact, it is possible to draw benefits from the general rules that guide the partitioning and from the mapping phase. However, no details are provided on the following phases, that take place at RTL level, thus there are some implementation problems that cannot find a solution within the *ADRIATIC* flow.

The *RECONF2* [27] aim is to allow implementation of adaptive system architectures by developing a complete design environment to exploit the benefits of dynamic reconfigurable FPGAs (real-time image processing or signal processing applications).The *RECONF2* builds a set of partial bitstreams representing different features and then use this collection to partially reconfigure the FPGA when needed; the reconfiguration task can be under the control of the FPGA itself or through the use of an external controller. A set of tools and associated methodologies have been developed to accomplish the following tasks: automatic or manual partitioning of a conventional design; specification of the dynamic constraints; verification of the dynamic implementation through dynamic simulations in all steps of the design flow; automatic generation of the configuration controller core for VHDL or C implementation; dynamic floorplanning management and guidelines for modular back-end implementation. It is possible to use as input for this flow a conventional VHDL static description of the application or multiple descriptions of a given VHDL entity, to enable dynamic switching between two architectures sharing the same interfaces and area on the FPGA. The steps that characterize this approach are the partitioning of the design code, the verification of the dynamic behavior and the generation of the configuration controller. The main limitation of the *RECONF2* solution is that it does not provide the possibility of integrating the system with both a hardware and a software part, since both the partitioned application and the reconfiguration controller are implemented in hardware.

3.2 The Earendil methodology

The *Earendil* 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 [1] and the YaRA (Yet another Reconfigurable Architecture) architecture, but it can be easily adapted to different architectural and SoC solutions, i.e. the RAPTOR2000 system [28]. In particular the Caronte and the YaRA solutions consist of two distinct parts: a **fixed part** containing all the components that always have to be present 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. The *Earendil* design flow consists mainly of three phases: **High Level Reconfiguration (HLR)**, **Validation (VAL)** and **Low Level Reconfiguration (LLR)**. Aim of *HLR* is to analyze the input specification in order to find a feasible representation, produced by a first partitioning phase, that can be used to perform the HW/SW 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. Finally, the computed cover is placed and scheduled onto the given device. On the opposite, the goal of *VAL* 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. Finally, the last step that has to be performed is *LLR*, introduced in Section 3.2.1, which goal is the definition of an automatic generation of the low-level implementation of the final solution that has to be physically deployed on the target device and that realizes the original specification.

3.2.1 Low Level Reconfiguration

Aim of the *Low-Level Reconfiguration* phase is to generate the low-level implementation of the desired system in order to make it possible to physically configure the target device to realize the original specification. A diagram showing the whole LLR process is presented in Figure 1. To develop

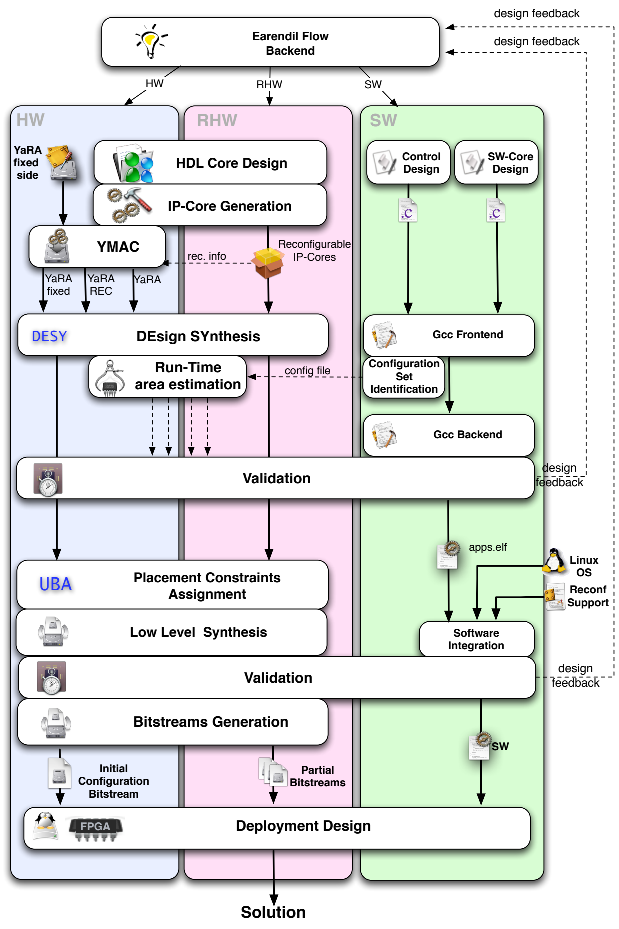

Figure 1: LLR design flow overview

the system it is necessary to split LLR 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 the High-level Description Language (HDL) *Core Design* and the *IP-Core Generation*, 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. After these steps, the fixed 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 fixed part of the architecture during the *DEsign SYnthesis* phase, while the fixed components will be synthesized together with the fixed 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 fixed 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, that extends a standard OS making it able both to perform reconfigurations of the reprogrammable device and to manage the reconfigurable hardware as well as the fixed hardware, in order to allow components runtime plugin. 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. CONCLUDING REMARKS

Embedded partial dynamic reconfiguration, due to its internal nature, can be used to realize embedded systems without involving any other device. The successful deployment of such complex and reconfigurable embedded systems to the market requires the identification, formalization, and implementation of concepts, methods, and tools for embedded software design that are able to ease the development of software components and the implementation of the system architecture. In this paper we presented an overview on the comprehensive work that has been done in the area of reconfigurable embedded systems, describing both the architectural and the methodological aspects of such systems.

5. REFERENCES

[1] Alberto Donato, Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. Exploiting partial dynamic reconfiguration for soc design of complex application on fpga platforms. In *IFIP VLSI-SOC 2005*, pages 179–184, 2005.

[2] F.Furtek. A field-programmable gate array for systolic computing. pages 183–199. Research on Integrated Systems: Proc. of the 1993 Symposium, G. Boriello and C. Ebeling, 1993.

[3] D. T. Hoang. Searching genetic databases on splash2. pages 185–191. Proc. of IEEE Workshop on FPGAs for Custom Computing Machines, D.A. Buell and K.L. Pocek, 1993.

[4] B.K. Fawcett. Applications of reconfigurable logic. pages 57–69. More FPGAs: Proc. of the 1993 International workshop on field-programmable logic and applications, W. Moore and W. Luk, 1993.

[5] P.C. French and R.W.Taylor. A self–reconfiguring processor. pages 50–59. Proc. of IEEE Workshop on FPGAs for Custom Computing Machine, D.A. Buell and K.L. Pocek, 1993.

[6] P. Lysaught, J. Stockwood, J. Law, and D. Girma. *Artificial neural network implementation on a fine–grainde FPGA*. R. Hartenstein and M.Z. Servit, 1994.

[7] W. Fornaciari and V. Piuri. Virtual fpgas: Some steps behind the physical barrier. Parallel and Distributed Processing (IPPS/SPDP'98 Workshop Proceedings), 1998.

[8] Manish Handa, Rajesh Radhakrishnan, Madhubanti Mukherjee, and Ranga Vemuri. A fast macro based compilation methodology for partially reconfigurable fpga designs. In *VLSI Design*, pages 91–, 2003.

[9] 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.

[10] Rafael Maestre, Fadi J. Kurdahi, Milagros Fernandez, Roman Hermida, Nader Bagherzadeh, and Hartej Singh. A formal approach to context scheduling for multicontext reconfigurable architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):173–185, 2001.

[11] John Williams and Neil Bergmann. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In Toomas P. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 163–169. CSREA Press, June 2004.

[12] Vince Ech, Punit Kalra, Rick LeBlanc, and Jim McManus. In-Circuit Partial Reconfiguration of RocketIO Attributes. Technical Report XAPP662, Xilinx Inc., January 2003.

[13] Xlinx. Early access partial reconfiguration guide. In *UG208*. Xlinx, 2006.

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

[15] H. F. Silverman. Processor reconfiguration through instruction–set metamorphosis. *IEEE Computer*, 1993.

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

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

[18] Xilinx. *The Programmable Gate Array Databook*. Xilinx.

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

[20] 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*, January 2000.

[21] 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.

[22] Mihai Budiu Srihari Cadambi Matt Moe Seth Copen Goldstein, Herman Schmit and R. Reed Taylor. Piperench: A reconfigurable architecture and compiler. In *Computer*, volume 33, pages 70–77. ACM Press, 2000.

[23] G. Brebner. A virtual hardware operating system for the xilinix xc6200. pages 327–336. IEEE Symposium on Field Programmable Logic and Applications, 1996.

[24] J. Case, N. Gupta, L.J. Mitta, and D. Ridgeway. *Design methodologies for core–based FPGA design*. Xilinx Inc., 1997.

[25] S. Guccione and D.Levi. Run–time parameterizable cores. pages 215–222. IEEE Symposium on Filed Programmable Logic and Application, 1999.

[26] Antti Pelkonen, Kostas Masselos, and Miroslav Cupék. System-level modeling of dynamically reconfigurable hardware with systemc. In *PDPS '03: Proc. of the 17th Int. Symposium on Parallel and Distributed Processing, Washington, DC, USA*. IEEE Computer Society, 2003.

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

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