Caronte: a complete methodology for the implementation of partially dynamically self–reconfiguring systems on FPGA platforms

Alberto Donato Fabrizio Ferrandi Massimo Redaelli
Marco D. Santambrogio Donatella Sciuto

Politecnico di Milano, {donato,ferrandi,redaelli,santambr,sciuto}@elet.polimi.it

It is common nowadays to employ FPGAs, not only as a means of rapidly prototyping and testing dedicated solutions, but also as a platform on which to implement actual production systems. The latter case comes in two flavors: Compile Time Reconfigurable systems, in which the configuration of the FPGA is done once and is never changed, or Run Time Reconfigurable systems, where the configuration of the chip is modified during the execution [6].

Although modern FPGAs allow the designer to modify dynamically even only portions of the chip, to this date there is a lack of satisfying design methodologies that using only non–proprietary widely available tools make it possible to optimally implement a high–level specification into a partially dynamically reconfigurable system.

The aim of this work is to propose a methodology for solving this problem. The main features of the *Caronte methodology* [1, 2] (which targets a board equipped with a Xilinx Virtex–II Pro FPGA with a PowerPC 405 processor) are: 1. full exploitation of *partial dynamic* reconfiguration; 2. the reconfiguration is *internal* (on the FPGA a microprocessor handles the reconfiguration through the ICAP module — thus reducing the reconfiguration times; e.g. [4]); 3. a real–time unix–like operating system helps the management of complex systems with multiple tasks, and simplifies reconfiguration through an optimized device driver.

The Hardware Architecture. Let us first describe the architecture of the solution resulting from the Caronte Methodology.

In order to manage reconfiguration internally it is necessary to always have a processing element running on the chip that communicates with the ICAP port. This means that it will be necessary to have a part of the FPGA which always remains the same during runtime (the *fixed side*, managing reconfiguration) while the rest of the available area is free for dynamic reconfiguration (the *reconfigurable side*).

The reconfigurable side can at any instant be viewed as the collection of a certain number of independent functionalities which are mapped on the chip as need be. Hence the area of the reconfigurable part is divided in rectangular boxes, all sharing a minimal interface that allows them to interact with the rest of the system, such as the IBM CoreConnect bus. These boxes are called *BlackBox*es. Aside from that, each BlackBox has also a *processing layer* part which can be reconfigured to various tasks, always retaining the communication functionalities offered by the communication layer. The *bus macro technology* is used to establish unchanging routing channels between modules. From the implementation point of view, this means that each BlackBox is in fact an EDK component made up of two VHDL, Verilog or EDIF files, the first one containing the *architecture–dependent* logic interface and the second one the *processing element* hardware description.

As for the fixed side is made up of six components: **ICAP**, used to read/write a configuration from/to the BRAM; **IP-Core Manager** (IPCM), a layer between the kernel of the operating system the BlackBoxes; **Memory**, used to manage all the partial bitstream data information; **Buses**, used to implement the architectural communication infrastructure (mainly with IBM *CoreConnect technology*); **PPC405 Processor**, used to provide the physical support for the *executable code* (including reconfiguration control); **Interrupt Controller**, used by the PPC405 processor and the BlackBoxes to dialog with one another.

The starting point for our work has been the *Board Support Package* (BSP) supplied by the board producer, Avnet Inc. The hardware support consists of a project to use with Xilinx design tools, EDK and ISE, including most of the physical hardware components of the board, such as processor, system buses (OPB and PLB), flash and RAM memory, Ethernet controller and serial port. The Avent BSP also contains the *Embedded Linux Development Kit* (ELDK), a package including tools for cross–development such as the gcc compiler for PowerPC and MicroBlaze architectures and the μClinux kernel [5]. ELDK can run on any Linux distribution on x86 machines. Both ELDK and the kernel have been modified by Avnet to include kernel support for specific hardware of the board (Ethernet, flash, leds) and some scripts to download the kernel image to the board using a network connection.

The Software Architecture. In order for the target system to be responsive to external inputs or to allow the execution of unbounded loops, it is necessary to run a controller and a scheduler on board. In the Caronte Architecture these processes are responsible for the correct execution of the code and the (un)loading of the BlackBoxes, and are represented by user processes of the operating system. The scheduler is activated in two cases.

On one hand, the actual run–time can be greater than the statically computed one. In this case the scheduler computes a new schedule with a list–based approach, using estimates on the new processing times and then finding a new critical path.

On the other hand, every time a BlackBox ends its execution a reconfiguration is necessary. In this case the controller starts to download the new configuration in the BRAM, af-

ter having informed all the BlackBoxes that can be affected by the reconfiguration process (thus enabling their spooler communication system). When the ICAP has finished reconfiguring the BlackBox, the controller re–enables normal communication.

On top of this architecture runs a port of a real–time GNU/Linux OS, μClinux [5], that allows the processes to exploit advanced functionalities. Since the μClinux kernel does not have any support for ICAP, we developed a Linux kernel module implementing an ICAP driver. The ICAP module at startup registers a character device major number and reserves the memory–mapped address space corresponding to the ICAP device (the base address can be specified as a parameter when loading the module). At this point it is possible to a create a device file, for example `/dev/icap`, that processes can access to execute reconfiguration. Three operations (besides `open` and `close`) are supported: `write` stores a partial bitstream data in a buffer; `read` reads the data in the buffer; `ioctl` allows 1. to discard the data in the buffer, or 2. to start the reconfiguration process.

The module also creates a directory (`/proc/icap`) containing files that give information about the driver status. The directory contains three files: `info` (with info about the linux device), `status` (reporting info from FPGA flag registers), and `device/` (a directory containing n files of the form `0`, `1`, ..., each referring to an ICAP device and showing a human–readable dump of info from the bitstream header, if present).

To mimic in software the hardware architecture, the OS accesses each BlackBox through a driver: the IP–core drivers, which are standard driver modules. These drivers are automatically loaded and unloaded by another kernel module, the IP–core Manager (IPCM), when the BlackBoxes are reconfigured. This is achieved by the controller sending an interrupt request at every reconfiguration. The IPCM also acts as an interface that all other user processes have to call in order to access the functionalities of the IP–cores.

The modularity of the approach can be further underlined since the IP–core drivers are developed hierarchically, using *stubs* according to their general functionalities. For instance there is a *network interface stub*, which is a specialization of the *generic stub*, and so on.

The Methodology. We can only briefly hint at the Caronte methodology here [1, 2]. It is composed of three phases.

In the *Hardware Static System Photo Phase* (HW–SSP) all the needed (global) states of the FPGA are computed as a combination of the fixed part and a certain number of IP–cores loaded in the BlackBoxes (each state is a "photo" of the system). The input for this phase is a partitioned system specification.

In the *Design Phase* all the information needed to compute all the bitstreams to physically implement the embedded reconfiguration of the FPGA are collected. In this phase the structure of each reconfigurable block is identified and all the placement and communication problems are solved.

Lastly, the *Bitstream Creation Phase* creates the bitstreams to be loaded on the FPGA.

Test and results. The Caronte flow has been applied to the AES (Rijndael) algorithm.

The idea (similarly to what was suggested in [3]) is to iterate the execution of each BlackBox a certain number of times, and in such a way to obtain blocks whose running time is comparable to the reconfiguration time of other BlackBoxes, thus hiding reconfiguration overhead.

The Rijndael algorithm is a succession of 4 basic operations that are iterated many times. These operations are performed on a 128 bit block, called *state*. After the sets identification phase [1], it is possible to identify all the processing elements and hence all the BlackBox cores — in the AES case, the application is composed of two BlackBoxes, BB_1 and BB_2 (and of the fixed Caronte Core). Now we can define all the needed HW–SSPs. In this case we obtain the four different HW–SSP that are shown in Table 1.

Table 1. HW–SSP Description

HW–SSP	Fix Module	BB_1	BB_2
0	Empty	Empty	Empty
1	Caronte Core	PE-A	PE-B
2	Caronte Core	PE-C	PE-B
3	Caronte Core	PE-C	PE-D
4	Caronte Core	PE-D	PE-A

Figure 1 shows a sample execution of the AES algorithm where the reconfiguration of a BlackBox has been hidden by the execution of an already mapped one.

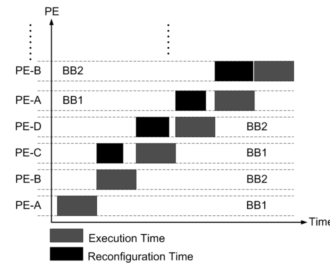

Figure 1: AES Caronte execution.

1. REFERENCES

[1] Marco D. Santambrogio. A methodology for dynamic reconfigurability in embedded system design. Master's thesis, Politecnico di Milano, 2004. http://www.micro.elet.polimi.it/people/santa.

[2] Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. A Design Methodology for Dynamic Reconfiguration: The Caronte Architecture. In *The 12th Reconfigurable Architectures Workshop (RAW 2005)*, 2005.

[3] R. Maestra, F.J. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, 9(6):858–873, December 2001.

[4] 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*. CSREA Press, 2004.

[5] Arcturus Networks Inc. μclinux, Embedded Linux/Microcontroller Project. In *www.uclinux.org*.

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