Caronte: a methodology for the implementation of partially dynamically seld-reconfiguring systems on FPGA platforms

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

Politecnico di Milano, Milano, Italy,
{donato,ferrandi,redaelli,santambr,sciuto}@elet.polimi.it,
http://micro.elet.polimi.it/

Abstract. This paper aims at introducing a complete methodology that allows to easily implement on an FPGA a system specification by exploiting the capabilities of *partial dynamic* reconfiguration provided by the modern boards. In the resulting system, which includes a set of fixed components (such as a processor and a controller) as well as some reconfigurable area (which can be allotted to different tasks running concurrently and replaced independently of one another — thus possibly hiding reconfiguration times), reconfiguration is handled *internally* by the system, without the use of external hardware. In order to meet the software requirements of complex systems, the solution is provided with a porting of a real–time GNU/Linux OS, μCLinux, which allows software processes to exploit a rich set of features, and with a Linux module that simplifies and enhances the handling of reconfiguration.

1 Introduction

To cope with changing user requirements, evolving protocols and data–coding standards, together with demands for the support of a variety of different user applications, many emerging appliances in communication, computing and consumer electronics need that their functionalities remain flexible *after* the system has been manufactured. FPGAs provide a means to meet these requirements, and have thus received increasing attention over the last years: not only they can implement arbitrary logic functions, but can also be reprogrammed an unlimited number of times during their lifetime.

Most applications running on FPGA–based systems are implemented using a single configuration per FPGA. This means that the functionality of the circuit does not change while the application is running. Such an application can be referred to as being *Compile–Time Reconfigurable* (CTR), because the entire configuration is determined at compile–time and does not change throughout system operation. Another strategy is that of implementing an application with *multiple* configurations per FPGA. In this scenario the application is divided into time–exclusive operations that need not (or cannot) operate concurrently. Each

of these operations is then implemented as a distinct configuration which can be downloaded onto the FPGA as necessary at run–time. This approach is referred to as *Run–Time Reconfiguration* (RTR) or *Dynamic Reconfiguration*.

FPGAs approaches to dynamic reconfiguration can be further divided into two categories: *small bits* and *modular based*. The former consists in changing small portions of the design in order to modify the system behavior — an example of this reconfiguration technique can be found in Xilinx XAPPs [1, 2]. The latter allows the creation of complex reconfigurable systems, composed of different IP–Cores. The Caronte methodology [3–5] describes how to create a flexible system design, where each core can be seen as a module that implements a specific functionality of the system.

Reconfiguration can be also classified in terms of *external* or *internal*. In the former scenario there exists an external entity which drives the configuration — either a PC connected to the board (for example using the JTAG controller) or some other kind of dedicated device. In this case the FPGA has a passive role, simply receiving the configuration data from the outside. With internal reconfiguration, instead, it is the *system itself* that modifies its own structure, and the code running on the local processor is communicating with the Internal Configuration Access Port (ICAP). This allows the system to run without needing to be connected to other devices, as long as it is possible to store all the necessary configuration information in the system memory. An example of such a system is the one proposed in [6].

The last generation of FPGAs, due to the high density of reconfigurable logic blocks present in the device, allow the designer to implement on them a complete system. This means that it is possible to include also a general purpose microprocessor, whether hard core or soft core. The designer, thus, must be ready to take into account also the software requirements of such a specification: in particular the processor, whether hardcore (such as the PowerPC) or softcore (MicroBlaze and Neos), typically runs a standalone executable implementing the application logic and exploiting the underlying hardware. On the other hand, though, there are scenarios that require the presence of a more complex software system to manage multiple tasks, interrupts and various system resources. This is the task typically delegated to an operating system.

There is a huge number of embedded and real–time operating systems, often built on top of a microkernel implementing basic management of interrupts and peripheral I/O. Also GNU/Linux, which is a complete operating system kernel, has been ported to architectures such as PowerPC and MicroBlaze, and adapted to support embedded systems such as development boards using Virtex–II and Virtex–II Pro FPGAs. For example, the *μClinux* project [7] contains a set of patches and extensions to the standard Linux kernel for specific hardware mounted on the most common development boards.

The Linux kernel modular architecture makes it easy to implement new modules and load or unload them dynamically in a running system.

The next section will present the state of the art of dynamic reconfigurable architectures and in section three we will propose our own methodology. Sections

four and five will show the physical implementation of the proposed reconfigurable architecture both for the hardware and the software components. Finally section seven will show some experimental results.

2 Previous work

Many implementations are now available both for CTR, such as [8–10], and for RTR [11–13].

In [14] the authors propose a new methodology to allow the platforms to hot–swap application specific modules without disturbing the operation of the rest of the system. This goal is achieved through the use of partial dynamic reconfiguration. The application has been implemented onto a Xilinx Virtex–E FPGA, and external reconfiguration is handled by an external device such as a Personal Computer, while ensuring the correct operation of those active circuits that are not being changed [15]. The reconfigurable modules are called Dynamic Hardware Plugin (DHP). A methodology is proposed to transform standard bitfiles, computed by common computer aided design tools, into new partial bitstreams that represent the DHP modules, using the PARtial BItfile Transform tool, PARBIT [16]. The PARBIT tool transforms FPGA configuration bitstreams to enable Dynamically Hardware Plugins modules in the Field–programmable Port Extender (FPX) [17]. The tool accepts as input the original bitfile, a target bitfile and some parameters given by the user, and provides as output the new bitstream, which then can be used to load a DHP module into any region of the Reprogrammable Application Device (RAD) on the FPX.

In [18] the hardware subsystem of the reconfiguration control infrastructure sits on the on–chip peripheral bus (OPB). The microprocessor, PowerPC or MicroBlaze, communicates with this peripheral over the OPB bus. The hardware peripheral is designed to provide a lightweight solution to reconfiguration. It employs a read/modify/write strategy. At any time, only one frame of data is considered. In this way no external memory is not needed to store a complete copy of the configuration memory. The program installed on the processor requests a specific frame, then the control logic of the peripheral uses the ICAP to do a readback and loads the configuration data into a dual–port block RAM. One block RAM can hold an XC2V8000 data frame easily. When the read–back is complete, the processor program directly modifies the configuration data stored in the BRAM. Finally, the ICAP is used to write the modified configuration data back to the device. The software subsystem is implemented using a layered approach. This solution allows a change in the implementation of the lower layers without affecting the upper layers, and proved useful for debugging. There are functions for downloading partial bitstreams stored in the external memory, for copying regions of configuration memory, and pasting it to a new location [18].

In [19], the authors considered reconfigurable computing as a close combination of hardware cores and of the run–time instruction set of a general purpose processor. The classification of core types is generally accepted to be split into three classes [20]: Hard cores, Firm cores and Soft cores. In [21], 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. For example, an adder core can be produced, and then parameterized at run–time for different operand widths. 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 [21]. 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 fashion to respond to the changing computational demands of the application. A problem of this methodology, though, is that the RTPs are targeted only to a single device family and there is no information about the communication channel between RTPs and about how they solve the physical reconfiguration problem. To control the mapping of cores at application run–time onto the programmable device, a management mechanism is required.

Our aim in this work is threefold. First of all, we show a novel implementation of *internal partial dynamic* reconfiguration requiring only tools that are already widely used for FPGA–based systems in order to be implemented. Secondly, we propose a new methodology that introduces the partial dynamic reconfiguration degree of freedom directly *in the design phase*. Lastly, we build an innovative modular Linux driver that greatly simplifies the software handling of reconfiguration, allowing the programmer to concentrate on a *hierarchical* view of the system to be implemented.

3 The proposed methodology

In this section we introduce a new design methodology for the implementation of a dynamic reconfigurable system using a common FPGA, through the combination of different design flows and using a development tool such as EDK, Embedded Development Kit, produced by Xilinx Inc. The proposed methodology could be applied within any specific device just porting it to a different design technology. In order to show the possibility of implementing the *reconfiguration design flow* we decided to use the Xilinx tools but it could be easily ported to be reused for different systems that can achieve embedded dynamic reconfiguration. One of EDK most important features is the possibility of developing complete systems, integrating both the software and the hardware components of the design in a single tool. In fact, EDK provides developers with a rich set of design tools, such as XPS (Xilinx Platform Studio), gcc, XST (Xilinx synthesizer), and a wide selection of standard peripherals required to build systems with embedded processors using the MicroBlaze softcore processor or/and the IBM PowerPC CPU [22]. The proposed methodology aims at introducing dynamic reconfiguration in the hardware part of the system, without increasing the complexity of the implementation, simply by changing the tools employed [4, 5]. In this way the implementation can be easily mapped on a standard FPGA. The Caronte Flow [3, 4] is mainly composed of three phases:

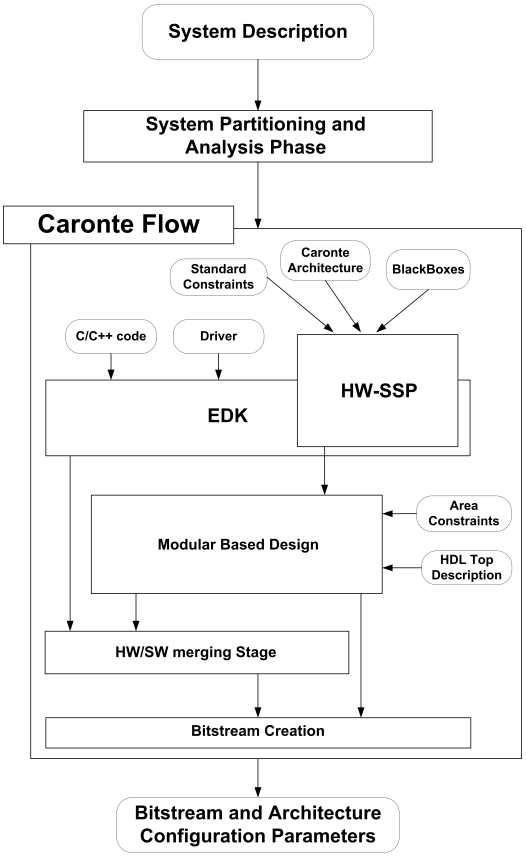

Fig. 1. Reconfiguration Design Methodology Flow.

HW–SSP Phase The HardWare Static System Photo Phase identifies a set of EDK system descriptions that will be (partially) dynamically reconfigured at run–time. These functional blocks are called *BlackBox cores* and will be described in Section 4.2.

Design Phase Aim of this phase is to collect all the information needed to compute all the bitstreams to physically implement the embedded reconfiguration of the FPGA.

It solves three different problems:

- Identify the structure of each reconfigurable block by providing a specific implementation for each of them. This phase is based on the Xilinx Modular Based Design approach;
- Identify, using the *Floorplanner* tool provided in the ISE tool chain, the area of each reconfigurable component of the system;
- Solve the communication problem between reconfigurable modules, by introducing *Bus Macros* that allow signals to cross over a partial reconfiguration boundary.

Bitstream Creation Phase This phase creates all the bitstreams needed to implement the system description onto an FPGA through the dynamic embedded reconfiguration.

Figure 1 shows the described methodology and how it can be included into the standard FPGA flow.

The Caronte Flow accepts as input the result of a previous partitioning and analysis phase [4]. Whatever the reason for creating a dynamic hardware configuration may be, there are common implementation issues: the system description must be partitioned in a fixed set of components that will be dynamically mapped onto a partitioned architecture. For this purpose, both the FPGA physical area and the initial system description have to be divided into several parts to provide the correct starting point for a dynamic reconfigurable design suitable to the system description provided. This first phase identifies all the processing elements of the description that will be mapped onto the corresponding part of the FPGA, as shown in Figure 2.

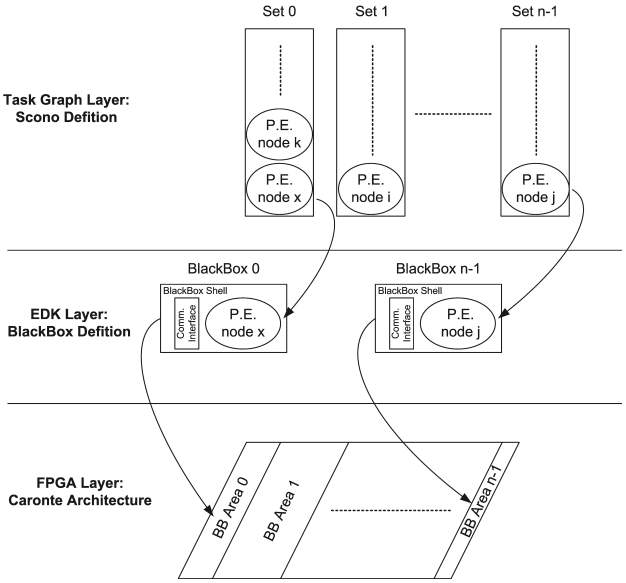

Fig. 2. Partitioning layers.

3.1 HW–SSP Phase

The input of the Caronte Flow is composed of a special set of EDK Cores, the BlackBox elements, described in Section 4.2, that are used by the HW–SSP phase to create all the HW Static System Photos, as shown in Figure 3.

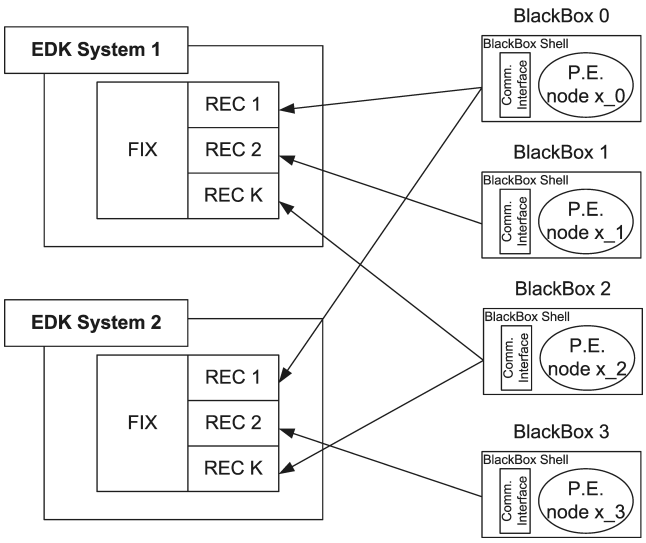

Fig. 3. HW–SSP definition.

An HW–SSP is an EDK system based on the Caronte architecture, described in Section 4. This architecture contains a fixed part and several reconfigurable blocks, named BlackBoxes. The application moves from an HW–SSP to another by reconfiguring the BlackBoxes and by leaving the fixed part unchanged. The idea is to consider the system in time as a sequence of static photos. All those HW–SSP *share* the static part of the system, which is used to implement the embedded reconfiguration of the other components, as shown in Figure 4. Finally, the EDK output is used as input for the next phase.

3.2 Design Phase

The idea is to implement a specific reconfiguration–oriented environment that, starting from a system description provided by EDK and using the *Modular Based Design* (MBD) generates all the bitstream for the final system implementation. To obtain all the HW–SSPs needed by the MBD the designer will use a part of the EDK implementation chain, starting from the design phase to the VHDL generation one. The produced VHDL descriptions must take into account the dynamic nature of the system: the main issues are raised by the communication channel between modules. In order to allow communication among dynamic modules a special bus, the BUS *Macro*, has to be introduced into the design description. Each time a partial reconfiguration is performed, the bus macro is used to establish unchanging routing channels between modules, guaranteeing correct connections.
The synthesis provided by EDK does not take into account the placement of components into specific FPGA areas; for our purpose this is indeed necessary,

since the FPGA is partitioned in fixed and reconfigurable areas. To accomplish this task, the Floorplanner, a tool contained in the ISE Xilinx package, can be used. The Floorplanner provides an easy way to constrain the placement of every component of a project onto a specific area of the physical architecture. When the FPGA is partially reconfigured, the configuration bitstream, called partial bitstream, contains data only for the area to be reconfigured. Partial bitstreams are computed as the logical difference between two complete configuration bitstreams. This means that, without constraining the components placement, it is impossible to guarantee that the partial bitstream between two configurations will affect only the desired area.

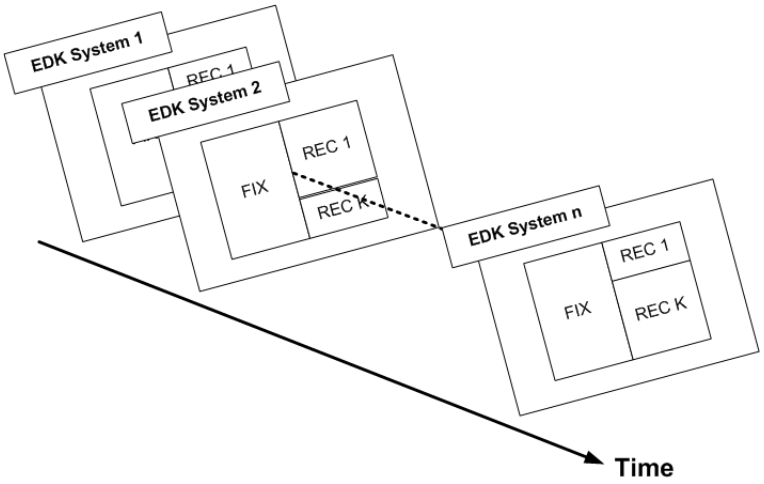

Fig. 4. HW–SSP point of view: the system execution.

4 The Hardware Architecture

This section describes the proposed model of dynamic reconfiguration under the GNU/Linux operating system environment, using a board equipped with a Xilinx Virtex–II Pro FPGA with a PowerPC 405 processor and a Linux distribution based on the μClinux kernel.
The core of the architecture is the PPC405 processor which implements both the controller and the scheduler for the given system implementation. Figure 5 presents the complete architecture, showing both the fixed and reconfigurable sides.

Both from the hardware and the software point of view, 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. 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.

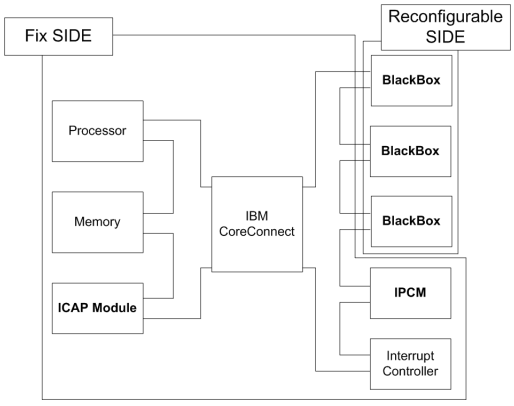

Fig. 5. The Architecture Overview

4.1 The Fixed Architecture of Caronte

The body of the Caronte architecture is the real physical implementation of the fixed part. It is basically a Von Neumann architecture composed of six classes of components:

- **ICAP**, used to read/write a configuration from/to the BRAM to/from a specific BlackBox;
- **IP-Core Manager, IPCM**, this hardware module is a sort of bridge between the SW side of the architecture, the kernel of the operating system, and the HW side, the BlackBoxes;
- **Memory**, used to store all the partial bitstream data information;
- **Buses**, used to implement the architectural communication infrastructure. It is possible to identify two different kind of busses:
  - The IBM *CoreConnect technology*, that represents the 90% of the entire communication system of the architecture;

- The *bus macro technology*, which provides a fixed *bus* of inter–design communication. Each time partial reconfiguration is performed, the bus macro is used to establish unchanging routing channels between modules, guaranteeing correct connections.

– **PPC405 Processor**, used to provide the physical support for the *executable code*;
– **Interrupt Controller**, used by the PPC405 processor and the BlackBoxes to dialog one to each other.

4.2 The reconfigurable side: the blackboxes

A BlackBox is a reconfiguration core, mainly defined by a processing element of the starting system description, which is set into a fixed known portion of the FPGA that can be completely reconfigured without interfering with the execution of the remaining part of the FPGA. Therefore, a BlackBox can be considered as a shell for processing elements. The BlackBox includes not only the logic implementing the component functionalities, but also the communication channel interface between the node and the system. This interface allows the node to send data directly on the communication channel or to temporally store a fixed number of data in an internal communication spooler, which is used during the reconfiguration action.
EDK defines as a component any part of an EDK architectural design such as a bus, or a peripheral or even a processor. A BlackBox can be considered as an EDK component, although this is a simplified way of thinking of a BlackBox. The main difference is that a BlackBox is not a static component mapped onto the FPGA, as any classical EDK component. It can be considered as a virtual shell used to contain different processing elements of the system description that need to be mapped onto the FPGA. In order to be able to implement a partial reconfiguration of a portion of the FPGA it is important to know which is the portion that has to be reconfigured. The Xilinx Platform Studio Tool of EDK, used to create FPGAs architectures, offers an automatic synthesis engine that generates a real project implementation by arranging each logic unit in a standard way. A BlackBox provides the interfaces needed by the VHDL description of a processing element to dialog with all the other components of the architecture, such as the CoreConnect bus, the processor, the interrupt controller and the other blackboxes. The BlackBox is shown in Figure 6.

During reconfiguration the Processing Element node logic will be modified, while the communication interface and IP Interconnect (IPIC) between the node logic and the interface will remain the same. This means that a BlackBox is constituted by two VHDL, Verilog or EDIF files, the first one containing the *architecture–dependent* logic interface and the second one the processing element hardware description.

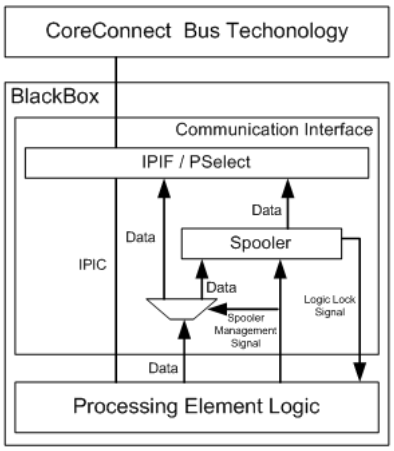

Fig. 6. A BlackBox overview

5 The Software Architecture

The software side of the Caronte architecture consists of a scheduler for dynamic computation of the execution times, and a controller which manages the reconfiguration process. Those components run as user processes under the GNU/Linux operating system. To deal with the underlying hardware, such as the ICAP module and the reconfigurable IP–Cores, a driver system have been introduced, based on the standard Linux kernel modules system.

5.1 The "Caronte Software"

In a first implementation, the Caronte software was realized as a *standalone* system, while now it has been integrated in an embedded version of the Linux operating system, moving it to a userspace process. The Caronte architecture allows the mapping of each processing element according to placement information. The estimated times for different reconfigurations are computed statically, but actual times can differ from those calculated. For this reason, the processor also runs a dynamic scheduler, which takes into account modifications from the original schedule.
The controller stands in a time watching state, controlling that the running time of each BlackBox meets its statically computed time.

In case the BlackBox running time exceeds the statically estimated time, the controller informs the scheduler that the run time of the BlackBox is greater than the estimated one. According to the information provided by the controller, the scheduler updates the processing element time information and computes a new schedule on the graph by following a list–based approach, in order to identify the new critical path and reorder the processing elements accordingly. The Caronte scheduler can be split in the following phases:

– **Controller Information Checking Phase**: stores the information provided by the controller;
– **New Time Computation Phase**: estimates a new execution time for the processing element given by the controller;
– **New Critical Path Computation Phase**: computes the new critical path and changes the Critical_Path, and Scheduled_Critical_Path variable values.

After informing the scheduler the controller returns in its time watching state, waiting for a new event.

Anytime a BlackBox execution terminates within its estimated time, a reconfigurable action has to be performed. At the end of its execution the BlackBox informs the controller of this event. During reconfiguration the controller, that knows which is the next node to be mapped on this BlackBox, downloads from the memory to the BRAM the correct configuration bistream, as shown in Figure 7.

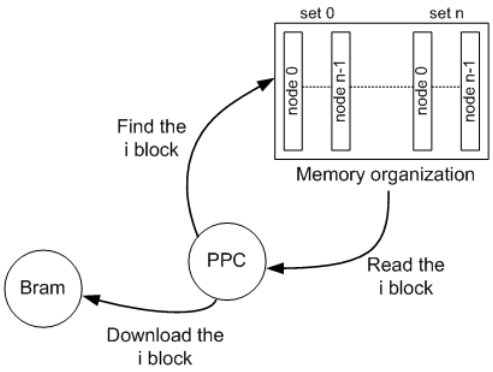

Fig. 7. Download the new configuration bitstream on the BRAM

At the same time the controller informs all the BlackBoxes (BBs) which can be disturbed by the reconfiguration action to activate their spooler communication system. At this point the PPC405 allows the ICAP to reconfigure the BlackBox with the new bitstream. Finally, when the new BlackBox has been mapped and starts its computation, the ICAP informs the processor that the reconfiguration action has successfully completed.
At this point, the controller enables the normal communications for the BBs that have been stopped.

5.2 Software support to dynamic reconfiguration

Stand alone code running on the FPGA needs to deal at a low level with hardware, including the ICAP. This means that creating a reconfigurable system has strong

implications also from the software point of view. If dynamic reconfiguration is desired, the application must implement functions both addressed to the system purpose (the actual computation) and to interface with the ICAP.

The actual Caronte architecture is based on GNU/Linux operating system, which is a complete multitasking operating system. The operating system considers the reconfiguration process as an autonomous thread of computation.
For this reason, the software side of Caronte (the scheduler and the controller) and the functions which deal and manage the hardware are separated. In this case, the application code runs as a user process in the system; this means that it does not have direct and low level access to the hardware, but it has to pass all the requests through operating system calls (read, write, etc...). Therefore, as far as reconfiguration is concerned, the OS itself must take care of the communication with the ICAP, by exporting an interface for user processes.

Since the μClinux kernel does not have any kind of support for ICAP, we developed a Linux kernel module implementing a driver for the ICAP peripheral. Linux operating system allows userspace programs to access devices via special files, located under the `/dev` directory. Each device is assigned a couple of numbers as id, indicating the driver managing the device (the "major" number) and the id of the specific device (the "minor" number); furthermore, they are also distinguished in "character" and "block" devices, based on the kind of access they support. When a kernel driver registers a major number, all access requests to the corresponding devices are directed to it, and hence it must implement handlers for various system calls: open, close, read, write, and so on. The ICAP module, on startup, registers a character device major number (by default 120) and reserves the memory–mapped address space corresponding to the ICAP device (as shown in Figure 8); 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 with major number 120 and minor 0, for example `/dev/icap`, that processes can access to execute reconfiguration.

System Calls There are currently three system calls, besides the open and the close operations, implemented by the driver:

write when a process requires reconfiguration, it simply writes the partial bitstream to the ICAP device; this can also be done manually by a user using standard Unix commands, for example `cat diff.bit > /dev/icap`. The reconfiguration does not take place immediately; instead configuration data is stored in a memory buffer until a specific request is issued through `ioctl`: in this way it is always possible to change the data stored by simply rewriting a new bitstream onto the device.

read reading from the ICAP device allows a user process to access the data stored in the memory buffer. The read operation allows reading a fragment or the entire bitstream loaded in the memory buffer.

ioctl this system call is generally for device control, to get or set configuration parameters and to interact with it in a more general way than allowed by read and write. When performing an ioctl call, the only required argument

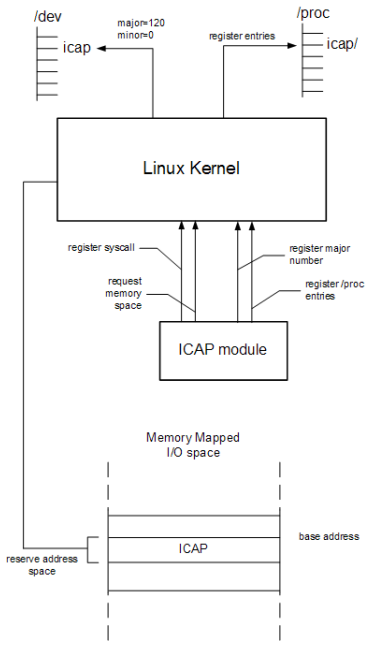

Fig. 8. ICAP Linux kernel module structure and registrations process

to the function is a number indicating the type of operation requested.
In the driver, two ioctl operations are allowed: the first is used to discard configuration data from the memory buffer, the second starts the partial reconfiguration, provided that a valid bitstream has been loaded into memory. In the latter case, as shown in Figure 9, the operation is performed by sending the bitstream, byte by byte, to the base address of the ICAP component. After the reconfiguration has been completed, the driver prints a message in the kernel log with the time used for the operation.

`/proc` filesystem interface The ICAP kernel module uses the standard Linux *proc* pseudo–filesystem to give information on the status of the driver. This filesystem, from a user point of view, is composed of normal files and directories, but reading or writing files actually triggers functions that can do any kind of action: usually reading a file results in getting information on devices status, while writing sets or modifies some parameters.
On module initialization, the `/proc/icap` directory is created; here the following files can be found:

`info:` the file contains information on the ICAP device, such as device id, address range in memory–mapped space and amount of memory buffer used.

`status:` reading this file will send a command to the ICAP which will result in reading the FPGA status register, containing flags reporting information on the status of the device and configuration mode.

`devices/0:` when a valid bitstream is loaded in the memory buffer, this file contains a human–readable dump of the information contained in the bitstream header, such as design filename, target part, creation time and date.

The module is designed to provide the capability of handling multiple ICAP devices (the actual number is specified at compile time), although current FPGAs contain only one physical ICAP component. If more than one device is used, the `devices` directory contains a file for each device.

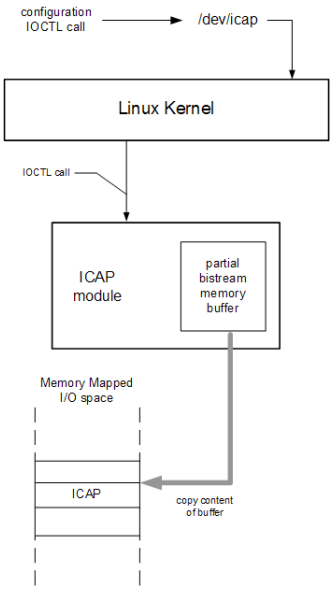

Fig. 9. Partial reconfiguration process with ICAP kernel module

The driver described can be used for both kinds of reconfiguration, small bits or module based, as long as a partial bitstream is available for download. Yet, if the small bits reconfiguration consists usually in modifying little configuration details in mapped peripherals or IP–Cores, not affecting the rest of the system, when one or more IP–Cores are added or removed, new features will be available while others may no longer be. This means that the operating system must cope with these changes and manage those resources, making them available to user processes.

6 The reconfiguration system

The architecture described in this section aims at creating an integrated hardware/software reconfigurable system where IP–Cores can be loaded and unloaded while the system is running, based on the required functionalities and on the area physically available on the FPGA. The idea is to create an hot–plug mechanism, where new peripherals announce themselves, allowing automatic loading of the corresponding software driver.

From the hardware side, it is necessary to have a controller that collects the information on the newly added IP–Cores, passing them to the software that manages the dynamic loading of the drivers. The information mainly consist of the Core type, which allows selection of the proper driver, and the I/O memory range.

The software side, instead consists of a core module that interfaces with the hardware controller and loads the specific drivers.

This hardware/software architecture has been implemented on an Avnet *Virtex*-II *Pro Evaluation Board*, connected to a *Communications/Memory Modules*, also produced by Avnet. The board integrates a Xilinx Virtex-II FPGA with an embedded PowerPc 405 processor, used to run the software part of the system, various kinds of RAM memory, Flash (where the operating system image is stored) and many additional components such as communication ports (ethernet, serial, ...) and general purpose I/O connectors.

6.1 Modular software architecture

The structure of the software component of the architecture is in some way specular to the hardware counterpart, implementing the dynamic reconfigurability as the possibility of loading and unloading at runtime drivers for the IP–Core mapped on the FPGA. As already discussed, addition and removal of IP–Cores results in changes in resources availability, which has deeper implications on system functionalities than small bits reconfiguration. The proposed architecture extends the one presented in [6], introducing a software layer that interfaces the operating system and, as a consequence, the userspace, with IP–Cores, through specific drivers.

Similarly to the hardware controller, there must be a software manager, called IP–*Core Manager* (IPCM), which acts as a layer between the kernel and the lower–level IP–Core drivers.

The IP–Core Manager The IPCM architecture exploits the Linux kernel modularity, creating a hierarchical structure among the kernel, the IPCM itself and the IP–Core drivers, as shown in Figure 10. From the kernel point of view, it is a standard module which registers a major number (by default 121) among character devices that will be used to access all the IP–Core devices. The IPCM requests to the kernel an address space to be assigned to the registered IP–Cores, allowing them to use memory mapping to communicate with the drivers.

The basic functions of the IPCM are the following:

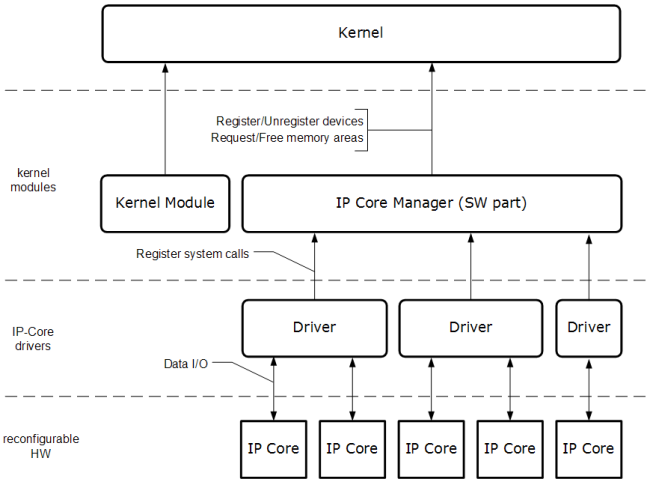

Fig. 10. Linux kernel and IPCM modules hierarchy

IP–Cores registration/deregistration: to accomplish the task, the IPCM needs to interface with the hardware controller; upon each partial reconfiguration, the controller sends an interrupt request to the IPCM. In the interrupt service routine, the IPCM gets from the controller the information on which devices have been deconfigured and which added. An IP–Core is essentially identified by its type (a numeric identifier) that defines the driver to be loaded for its management, and by its address space (base address and range), which must be in the address range registered by the IPCM.

specific drivers load/unload: IP–Core drivers are implemented as Linux kernel modules, but they don't need to be loaded manually; instead, each time a core is loaded for which a driver is not already present, the IPCM automatically loads it.

Besides loading the driver, the manager exports a function that registers a structure containing data on the driver; the driver, during the loading phase, provides the IPCM all the necessary data (driver id, name, list of implemented system calls) invoking the function exported by the manager. In this way, the IPCM maintains an updated list of all registered drivers; each driver data structure also contains the list of the IP–Cores managed by the driver.

system calls management: other than providing registration and deregistration capabilities, the module must also allow the use of the IP–Core from the userspace. A unique character device major number is associated with the IP–Cores; the IPCM uses the minor number to identify the different IP–Cores. Since this identifier is currently implemented in Linux with an unsigned 8–bit wide integer, this allows up to 256 different IP–Cores to be registered, which is a fairly large number for current FPGA capabilities.

When a system call is issued for a device, the IPCM delivers this request to the correct driver which implements this call for the specific underlying

hardware. To be able to distinguish IP–Cores both by their type and by a unique identier, we adopted the rule to consider the 4 most significant bits of the device minor number as identifier of the device type (indicating the associated driver), and the other 4 bits as device identifier within the driver. This means that there can be up to 16 drivers, each managing 16 IP–Cores.

Driver modularity Since the IP–Cores all use memory mapping to communicate, the drivers managing them will be very similar, the main difference being the functions performing reads and writes with the device and interrupts. According to this observation, a hierarchical architecture has been implemented to manage the driver creation and implementation.

The proposed solution has been implemented as a sort of *stub*, as shown in Figure 11. This simplifies the writing of IP–Core drivers, as the stub contains the implementation of functions common to all drivers, such as module initialization and shutdown, registration and deregistration with the IPCM. The main aim of this process is to hide as much as possible the Linux kernel programming interface, so that a user wanting to write a driver for an IP–Core does not need to know all kernel programming details or the internal structure of the IPCM, but has only to implement the specific functions complying to a simplified interface, while the stub performs the linking with the corresponding system calls and interfacing with the IPCM.

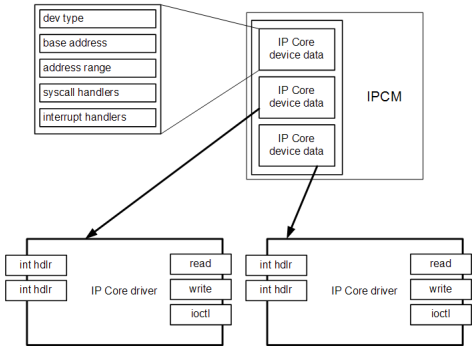

Fig. 11. IP–Core Manager and drivers structure

7 Test and results

The Caronte flow has been applied to the AES (Rijndael) algorithm to test Caronte architectural features, such as the possibility of storing the reconfiguration data on the board without external resources.

The first phase of the analysis and partitioning of the system description has been applied to the AES algorithms to obtain a first HW/SW codesign solution of the entire system to test the proposed methodology. After that step we further partitioned the hardware description of the system to obtain all the processing elements needed as input by the Caronte flow.

We decided to adapt our execution model to be able to justify the reconfiguration approach using a model similar to the one proposed in [23]. The idea 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, as shown in Figure 12.

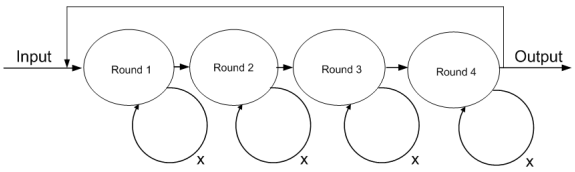

Fig. 12. Execution model.

Let us show the details of the methodology on the Rijndael example. 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*, organized as a 4×4 matrix of 8 bit elements.
After the sets identification phase [3], it is possible to identify all the processing elements and so all the BlackBox cores. Having all the cores means that we are now ready to define all the HW–SSPs for the algorithm. According to this scenario the Caronte architecture chosen for the AES application is composed of two BlackBoxes, BB_1 and BB_2, and of the Caronte Core, which in turn is made up by all the static parts previously described. In this case we obtain the four different HW–SSP that are shown in Table 1. Figure 13 shows a sample execution

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

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

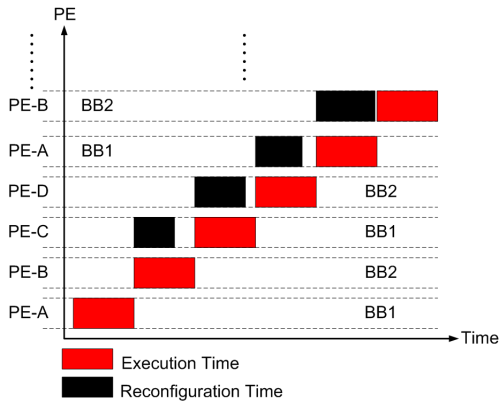

Fig. 13. AES Caronte execution.

The reconfiguration time for the first two BlackBoxes of the AES algorithm, A and B, is not shown in Figure 13 since these two components are mapped as the start–up configuration of the entire FPGA.

Let us show the details of the methodology on a second example: the MD5 algorithm. The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit *fingerprint* or *message digest* of the input.
The methodology allows the identification of *two* BlackBoxes, BB_1 and BB_2. Also the Caronte Core, composed of the processor, the memory, the ICAP module and all the other static parts previously described, is included in the design. In this case we obtain *six* different HW–SSP that are shown in Table 2.

Table 2. 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-E	PE-D
5	Caronte Core	PE-E	PE-F
6	Caronte Core	Empty	PE-F

The access time to the memory, where all the difference bitstreams are stored, has been obtained via a timing test: writing 32 bits of data takes 0.135μs, while reading the same amount of data requires 0.020μs. Without considering the first configuration bitstream, which implies a complete configuration of the FPGA, the comparison between the external reconfiguration and the embedded one are shown in Table 3.

Table 3. Embedded Vs External Reconfiguration

Action	External Rec.	Embedded Rec.
Rec. Time C block	14.558s	15.152ms
Rec. Time D block	14.597s	15.305ms
Rec. Time E block	14.560s	15.223ms
Rec. Time F block	15.482s	15.837ms

Also in this example the reconfiguration time for the first two BlackBoxes (A and B) are not shown, as already said, because they are part of the starting up configuration of the entire FPGA.

The results for both the architectures are shown in Table 4.

Table 4. Tests

Input	#RECs	$\frac{\mu(Rec.Times)}{\#BBs}$	$\frac{\mu(Exe.Times)}{\#BBs}$
MD5	5	15.379ms	16.765ms
AES	4	12.405ms	13.672ms

Column 2 lists the number of reconfigurations, RECs, that have to be performed in order to implement the complete architecture, while columns 3, and 4 list the average of the embedded reconfiguration time and of the RECs execution time, respectively.

8 Concluding Remarks

Preliminary results show that the Caronte methodology, implementing a module–oriented approach based on an EDK system description, provides an effective and low cost approach to the partial dynamic reconfiguration problem. Its strength lies both on introducing the partial dynamic reconfiguration degree of freedom at *design time*, and on the use of widely available tools. Also, the Linux driver we have developed allows a simplified (and yet flexible and hierarchical) software interface to hardware reconfiguration.

We are now working on a new version of the IPCM module that embeds the IPCM, the ICAP and the Interrupt controller in just *one* module. This new module will provide a single access point for the reconfiguration action both for the HW and the SW side of the architecture, and will hence guarantee less area overhead on the FPGA.

We are also developing an automated version of the entire flow (addressing problems such as task scheduling and task partitioning, which are now only semi–automated) able to define all reconfiguration bitstreams, transforming the input description into VHDL code that will define the core of each BlackBox and hence producing all the HW–SSP's.

References

1. Derek R. Curd. Dynamic Reconfiguration of RocketIO MGT Attributes. Technical Report XAPP660, Xilinx Inc., February 2004.
2. Vince Ech, Punit Kalra, Rick LeBlanc, and Jim McManus. In-Circuit Partial Reconfiguration of RocketIO Attributes. Technical Report XAPP662, Xilinx Inc., January 2003.
3. 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.
4. Marco D. Santambrogio. Dynamic reconfigurability in embedded system design — a model for the dynamic reconfiguration. Master's thesis, University of Illinois at Chicago, 2004. http://www.micro.elet.polimi.it/people/santa.
5. 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.
6. 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.
7. Arcturus Networks Inc. μclinux, Embedded Linux/Microcontroller Project. In *www.uclinux.org*.
8. D. T. Hoang. Searching genetic databases on splash 2. pages 185–191. Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, D.A. Buell and K.L. Pocek, 1993.
9. D. A. Buell. A splash 2 tutorial. technical report src–tr–92–087. *Supercomputing Research Center*, December 1992.
10. D. T. Hoang. Searching genetic databases on splash2. pages 185–191. Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, D.A. Buell and K.L. Pocek, 1993.
11. D. Ross, O. Vellacott, and M. Turner. An fpga–based hardware accelerator for image processing. pages 299–306. More FPGAs: Proceedings of the 1993 International workshop on field–programmable logic and applications, W. Moore and W. Luk, 1993.
12. 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.
13. P.C. French and R.W.Taylor. A self–reconfiguring processor. pages 50–59. Proceedings of IEEE Workshop on FPGAs for Custom Computing Machine, D.A. Buell and K.L. Pocek, 1993.
14. Edson L. Horta, John W. Lockwood, and David Parlour. Dynamic hardware plugins in an fpga with partial run–time reconfigurtion. pages 844–848, 1993.
15. S. Tapp. Configuration quick start guidelines. *XAPP151*, July 2003.
16. 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.
17. 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.

18. B. Blodget, S. McMillan, and P. Lysaght. A lightweight approach for embedded reconfiguration of fpgas. 1991.
19. G. Brebner. A virtual hardware operating system for the xilinix xc6200. pages 327–336. IEEE Symposium on Field Programmable Logic and Applications, 1996.
20. J. Case, N. Gupta, L.J. Mitta, and D. Ridgeway. *Design methodologies for core–based FPGA design*. Xilinx Inc., 1997.
21. S. Guccione and D.Levi. Run–time parameterizable cores. pages 215–222. IEEE Symposium on Filed Programmable Logic and Application, 1999.
22. Xilinx Inc. *Embedded Development Kit EDK 6.2i*. Xilinx Inc., 2004.
23. 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.