Operating system support for dynamically reconfigurable SoC architectures

Alberto Donato Fabrizio Ferrandi Marco Santambrogio Donatella Sciuto
Politecnico di Milano
Dipartimento di Elettronica e Informazione
Via Ponzio 34/5
20133 Milano, Italy
{donato,ferrandi,santambr,sciuto}@elet.polimi.it

ABSTRACT
The advantages and the flexibility introduced into the hardware implementation by partial dynamic reconfiguration has rapidly changed the design flow of embedded systems. Although nowadays it is common to deal with systems characterized by a dynamic architecture able to manage and to adapt themselves to extremely different working scenarios, it is not so easy to provide such a flexibility also into the software part of these systems. In order to cope with this problem we developed 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. This methodology can be applied to different architectures providing a powerful and flexible software solution and, at the same time, it can be easily customized to respond to specific behaviors and requirements.

1. INTRODUCTION
Reconfigurable devices, such as modern FPGAs provide advanced capabilities which allow the creation of embedded systems on single chips (SoC). One of the most challenging opportunities provided by this kind of devices is the ability to dynamically and partially reconfigure themselves, which consists in the modification of a portion of the circuitry mapped on the FPGA while the system is running, while maintaining the functionality of the rest of the system. This ability allows development of flexible systems that can deal with changes in the requirements, standards and operational conditions.

The Caronte methodology and architecture [1–3] exploits the described FPGA capabilities to create a complex reconfigurable system which allows dynamic placement and configuration of hardware computational modules (called Blackboxes) on a reconfigurable portion of an FPGA with an embedded general purpose processor. These modules are connected to the system through a standard bus, so that the application code running on the processor can exchange data with them.

In [4], the authors present a different approach to dynamic reconfigurable embedded system, which consists in mapping a network of processors (Xilinx MicroBlaze soft-cores) on an FPGA. In the architecture proposed in this paper, software reconfiguration is considered, since different tasks are mapped on the processors when a particular functionality is needed; the reconfiguration is managed by a run-time system which also runs on a processor. This approach has been chosen to avoid some limitations in the design of hardware reconfigurable modules, such as the definition of fixed routing communication channels, which reduce the system scalability since they are bound to a specific FPGA; changing the target device would require some modifications in the system design.

A mixed hardware/software approach to reconfigurability is presented in [5], where the proposed reconfigurable system executes multiple applications on top of a common API (Application Programming Interface). The API provides the services necessary to the applications (similarly to an operating system) hiding their low level implementation, which can be done in software (running on a general purpose processor) or in hardware by a reconfigurable core. Dynamic reconfiguration, in this case can be a matter of changing the software running on the processor or mapping a new core in a free zone of the device or replacing another one. The paper also discusses the possibility of providing a given function both in hardware and software with multiple implementations which differ in resource requirements (memory used, reconfigurable cells, computation time), using specific algorithms to select the solution which best fits a particular situation.

This paper proposes an approach to the dynamic reconfiguration which has common points with many of the previously described solutions, since it allows the exploitation of reconfigurable hardware resources (fIP-Cores) from multiple software applications which run on a complete multitasking general purpose operating system. It also provides support for the addition and removal of Cores within the operating system, which makes it possible to add at runtime both new peripherals and resources and the software needed to manage them with no need of stopping and restarting the whole system.

2. THE RECONFIGURABLE ARCHITECTURE
The software architecture proposed in this paper aims at providing complete support for dynamic reconfiguration and exploitation of FPGA-mapped devices within an operating system environment. The Linux kernel has been chosen as
development platform, mainly for source code availability, number of supported hardware platforms (including soft and hard cores such as Xilinx MicroBlaze and IBM PowerPC 405) and for its modular architecture, which provides the possibility of extending kernel functionalities and drivers in a flexible way. The discussed system is based on the Caronte hardware architecture, presented in [1–3]. The Caronte architecture has been implemented on a Xilinx Virtex-II Pro FPGA using the embedded PowerPC 405 processor to run the software code and the ICAP (Internal Configuration Access Port) component included in the device for the partial reconfiguration. The software architecture is made of two main components: the manager for the dynamic partial reconfiguration, which offers reconfiguration access from user space, interfacing with the hardware which provides reconfiguration and the infrastructure for the management of FPGA-mapped devices, which require specific drivers to be accessed from the operating system and user space processes.

2.1 Partial dynamic reconfiguration support in Operating System

Many embedded systems including a general purpose processor (whether a soft-core or a hard-core one) and reconfigurable logic run a standalone application on the processor, which implements both the low-level interface with the underlying hardware and the application code. This approach, however, is not very flexible, since it is necessary to develop or include in each application the functions that manage the hardware, and only a single application can be run on the processor. In our methodology, the use of a complete operating system, such as the Linux kernel, is a major advantage, since the hardware management functions are all demanded to the operating system, and user processes can access the devices using an easier interface. Moreover, many different applications can be run at the same time on the processor thanks to the multitasking provided by the operating system and can share the resources mapped on the FPGA. The µcLinux kernel, which is a modified version of the standard Linux kernel for processors without Memory Management Unit (MMU), does not have any kind of support for the ICAP device. For this reason, the first step has been the development of a Linux kernel module implementing a driver for the ICAP peripheral integrated in the Xilinx Virtex-II and Virtex-II Pro FPGAs. The Linux operating system allows user space programs to access devices via special files, located under the /dev directory. Each device is identified by a couple of numbers, indicating the driver managing the device (called “major” number) and the id of the specific device (called “minor” number). Devices are classified into two families, “character” and “block” devices, based on the kind of access they support. When a code running in kernel space registers a major number, all access requests to the corresponding devices are directed to it. For this reason the driver must implement handlers for various system calls: open, close, read, write, and so on. This is done by the ICAP kernel module on startup (the default major number used is 120). The driver also reserves the memory-mapped address space corresponding to the location of the ICAP device (as shown in Figure 1); the base address can be specified as a parameter when loading the module. At this point creating a device file with major number 120 and minor 0, called for instance /dev/icap, allows user space processes to access the device and perform reconfigurations.

![Figure 1: ICAP Linux kernel module structure and registrations process](image)

2.1.1 icap driver system calls

The ICAP driver module implements three system calls, besides the basic open and close functions:

- **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 doing `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 to the function is a number indicating the kind 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, 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.

2.1.2 Information on the driver status

The ICAP kernel module uses the standard Linux /proc pseudo-filesystem to provide information on the status of
the driver. This file system appears as composed of normal files and directories, but when read or write operations are performed to one file, special functions are triggered, which can do any kind of action: usually reading a file results in getting information on devices status, while writing sets or performed to one file, special functions are triggered, which files and directories, but when read or write operations are to the driver. This file system appears as composed of normal files and directories, but when read or write operations are

**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.

The module is designed to be able to handle multiple ICAP devices, and the actual number is specified at compile time. Current FPGAs contain only one physical ICAP component, but multiple devices support can be useful if the hardware architecture contains more than one reconfigurable device (e.g. FPGAs arrays). If more than one device is used, the *devices* directory contains a file for each device. The ICAP driver can be used both for small bits and module based reconfiguration, as long as a partial bitstream is available for download. The main difference is in that small bits reconfiguration consists usually in modifying little configuration details in mapped peripherals or IP-Cores, not affecting the rest of the system; in the module based one, instead, 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.

3. THE MODULAR SYSTEM FOR RECONFIGURABLE DEVICES

Dynamic mapping and unmapping of IP-Core devices requires, from the software point of view, a corresponding runtime loading and unloading of specific code which must be able to manage the peripherals and offer the functionalities provided by them to the user space processes. The basic idea of the proposed infrastructure is to create an hot-plug mechanism, where new peripherals announce themselves, allowing automatic loading of the necessary software drivers within the Linux kernel. From the hardware side, it’s necessary to have a controller that collects information on the newly added IP-Cores, passing them to the software that manages the dynamic loading of the drivers. The informations mainly consist of the Core type, which allows selection of the proper driver, and I/O memory range.

3.1 Modular software architecture

The structure of the software architecture is in some way specular to the hardware one of Caronte, since it implements dynamic reconfigurability as the ability to load and unload drivers for the IP-Cores mapped on the FPGA upon request, during normal system execution. As already discussed, module based partial reconfiguration implies addition or removal of IP-Cores peripherals, which results in changes in the availability of system resources and functionalities, with deeper implications on the entire system than in small bits reconfiguration. The proposed architecture extends the one presented in [1], introducing a software layer that acts as an interface between the operating system and, as a consequence, the user space, and the IP-Cores, using specific drivers. The core of this modular architecture is a software manager, called IP-Core Manager (IPCM), which performs management of the lower-level IP-Core drivers, linking them to the Linux kernel in order to make IP-Core devices available to the user space.

### 3.1.1 The IP–Core Manager

The IPCM structure exploits the Linux kernel modularity by creating a hierarchical structure, which consists of the kernel, the IPCM itself and the IP-Core low-level drivers. From the kernel point of view, the IPCM is a normal module which registers a major number (by default 121) in character devices used to access all the IP-Core devices. The IPCM also requests to the kernel a range in the address space to be used by the device drivers to communicate with the IP-Cores through the memory mapping mechanism.

The basic functions of the IPCM are the following:

**IP-Cores registration/deregistration**: to recognize mapping and unmapping events, 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 receives from the controller the information on which devices have been deconfigured and which added. An IP-Core is basically 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-Cores drivers are also implemented as Linux kernel modules, but they don’t need to be loaded manually. 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 also exports a function that registers a structure containing data on the driver; the driver, during the loading phase, provides the IPCM with all the necessary data (driver id, name, list of implemented system calls) invoking the function exported by the manager. In this way, the IPCM can keep 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 provide an interface to access the IP-Cores from the user space. A unique character device major number is associated with the IP-Cores; the IPCM uses the minor number to identify the different IP-Cores instances. 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 possibilities. 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 the correct driver for the ip-Cores, both by their type and by an unique identifier, 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 has been adopted. This means that there can be up to 16 drivers, each one managing up to 16 ip-Cores.

3.1.2 Driver modularity

Since the ip-Cores all use a memory mapping mechanism to communicate with the operating system, the drivers managing them will have many similarities, mainly in the initialization and shutdown functions. The main differences in the drivers, instead, will be in the functions performing reads and writes with the device and interrupt management. 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. This simplifies the writing of ip-Core drivers, as the stub contains the implementation of all 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 deal with all programming details of the Linux kernel or the internal structure of the ipcm, but has to implement only the specific functions complyng to a simplified interface. The stub is responsible to perform the linking of the driver system calls with the corresponding ones in the kernel and to manage the interfacing with the ipcm.

4. TEST AND RESULTS

The software architecture has been tested using the Caronte architecture described in [1,2]. As shown in the previous section, in order to use an operating system such as Linux on a self dynamic reconfigurable embedded system it is necessary to adapt also the hardware system architecture. The new architecture which has been implemented has been extended with some IP-Cores such as the Ethernet, the Flash and the SDRAM interfaces to be able to exploit all the operating system capabilities. The system has been tested on a Xilinx FPGA, the Virtex II Pro xc2vp20 chip, and the preliminary test showed that the free reconfigurable space left from the fixed part of the architecture was too small to be used to map reconfigurable components. In this scenario our attention was focused on determining which part of the architecture, from the fixed part of it, could be omitted in order to obtain a better usage of the space available on the FPGA. The results of our exploration shows that one of the biggest component of the original architecture is the Ethernet Controller. After the definition of the basic system architecture supporting the Linux operating system, the comparison between this new architecture and the classic one is reported in Table 1.

The Table shows that introducing an operating system able to support also the self dynamic reconfiguration is not so expensive for what concerns the amount of the hardware resources usage. The set of tests has been implemented using the same architecture as the one proposed for the Virtex II Pro xc2vp7 but using a Xilinx FPGA, the Virtex II Pro xc2vp20, that is characterized by a greater number of hardware resources and a bigger reconfigurable area. On this new FPGA we were able to implement a self dynamic reconfigurable system using an operating system with better performances from reconfiguration time point of view. All the embedded partial reconfiguration, EPR, has been implemented on the the Virtex II Pro xc2vp20 FPGA and the results are shown in Table 2.

We made different tests using the proposed software solution and the Caronte architecture and the reconfiguration performance has been estimated in about 1.5 MBytes/s transfer rate.

5. REFERENCES


| Table 1: Caronte classic Vs Linux Caronte on the Xilinx XC2VP7 FPGA |
|-------------------------|------------------|------------------|------------------|------------------|------------------|
| Resource               | Caronte Classic | Caronte Linux   | Total available  |
| Slice Flip Flops        | 1843  | 18%   | 2369  | 24%   | 9856  |
| 4-input LUTs            | 1727  | 17%   | 2173  | 22%   | 9856  |
| Occupied Slices         | 1818  | 36%   | 2262  | 45%   | 4928  |
| Bonded IOBs            | 107   | 27%   | 168   | 42%   | 396   |
| Block RAM              | 32    | 72%   | 32    | 72%   | 44    |
| DCMS                   | 2     | 50%   | 2     | 50%   | 4     |

<table>
<thead>
<tr>
<th>Table 2: EPR performances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
</tr>
<tr>
<td>RC0</td>
</tr>
<tr>
<td>RC1</td>
</tr>
<tr>
<td>RC2</td>
</tr>
<tr>
<td>RC3</td>
</tr>
<tr>
<td>RC4</td>
</tr>
<tr>
<td>RC5</td>
</tr>
<tr>
<td>RC6</td>
</tr>
<tr>
<td>RC7</td>
</tr>
</tbody>
</table>