

POLITECNICO DI MILANO  
MICROLAB  
DRESD PROJECT



DEVELOPMENT OF AN OS  
ARCHITECTURE-INDEPENDENT LAYER FOR  
DYNAMIC RECONFIGURATION

Tutor: Ing. Marco Domenico Santambrogio

Project Authors:  
**Ivan Beretta** , Matr.: 708233

A.A. 2006-2007

## Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Problem Description . . . . .	6
1.2	Project Goals . . . . .	7
1.3	Project Organization . . . . .	7
<b>2</b>	<b>State of the art</b>	<b>9</b>
2.1	Operating System Support . . . . .	9
2.2	Caronte Solution . . . . .	10
2.2.1	Reconfiguration controller driver . . . . .	10
2.2.2	IP-Core Manager . . . . .	11
2.3	YaRA Solution . . . . .	12
2.3.1	Kernel modules . . . . .	13
2.3.2	ROTFL Architecture . . . . .	14
2.4	Conclusions . . . . .	16
<b>3</b>	<b>Project Details</b>	<b>17</b>
3.1	First Phase: Intermediate layer definition . . . . .	17
3.1.1	Factorization of YaRA solution . . . . .	17
3.1.2	Mapping of common functionalities on the two imple- mentations . . . . .	19
3.1.3	Reconfiguration scheduler . . . . .	20
3.2	Second phase: Caronte implementation analysis . . . . .	21
3.2.1	Bootstrap from flash memory . . . . .	21
3.2.2	Bootstrap process on Avnet boards . . . . .	22
3.2.3	Avmon . . . . .	23
3.2.4	ELDK . . . . .	26
3.3	Hardware architecture . . . . .	26
3.3.1	Results . . . . .	28
<b>4</b>	<b>Future works</b>	<b>30</b>

## List of Figures

1	The Caronte solution . . . . .	12
2	YaRA solution kernel modules . . . . .	13
3	ROTFL Architecture . . . . .	15
4	Flash memory layout on Avnet Virtex II-Pro Evaluation Board	24

## List of Tables

- 1 Mapping of the layer functionalities on the existing solutions . 20
- 2 Area requirements for the architecture generated by ISE 7.1 . 28
- 3 Area requirements for the architecture generated by ISE 9.1 . 29

### **Abstract**

The increasing complexity of the hardware architectures that are used on embedded systems makes the software development process a very costly task. This is particularly true when the software application is a single, standalone application that has to deal with the management of the hardware components. The introduction of an operating system support in such systems simplifies this process, because all the user applications do not have to be aware of the underlying hardware, but they simply interact with an abstraction layer.

Aim of this project is to study an operating system support for all the dynamic reconfigurable architectures developed in the Microarchitectures Laboratory at Politecnico di Milano. Several solutions were proposed in the past, but all of them were tied to a specific architecture and, as a consequence, to a specific processor.

These solutions should be used as a starting point to define a new layer of the operating system. All the existing functionalities that are related to the dynamic reconfiguration process and are architecture-independent will be collected and used to develop a brand new layer. The defined layer should be unaware of the high-level Linux distribution that is currently used, which is necessary to make it executable on different processors. Moreover, it should be unaware of the low-level architecture on which it is executed, in order to make it more general purpose.

# 1 Introduction

This project work is focused on the definition of a general, architecture-independent layer for the operating systems that are used in dynamic reconfigurable environments. The first section of this document gives a first description of the benefits that are provided by the introduction of an operating system, with respect to an ad-hoc software application. After the problem space has been defined, the goals of this project will be clearly defined. Finally, a brief description of the work phases will be provided.

## 1.1 Problem Description

An operating system is a software applications whose task is to manage the hardware resource while offering an abstraction layer to other user applications. The main advantage of the presence of an operating system is that the software development process is highly simplified, since an user application does not have to directly interact with hardware components. This also means that an application does not deal with all the hardware details and, moreover, it can be completely unaware of the hardware on which it is executed. In this scenario, the software application performs a series of system calls to the operating systems, which handles the requests by interacting with the hardware side. Since an application only performs system calls, the same code can be easily ported on different architectures by providing different back-ends, thus improving the software compatibility.

The introduction of an operating system in an embedded system can bring the same advantages described above. However, a stand-alone software application is usually preferred in this kind of systems, and it is in charge of handling the hardware resources. Such an application may become too complex to be realized on large systems and, as previously described, it must be adapted whenever an hardware component changes.

Stand-alone software applications are particularly spreaded in dynamically reconfigurable architectures. This is mainly due to the fact that a common support for the reconfiguration process, which collects all the most common functionalities, does not exist. As a consequence, when an operating system support was introduced on a reconfigurable architecture, it usually implemented the required functionalities from scratch, with a waste of time and resources. An architecture-independent layer that collects the most common functionalities of a reconfigurable system would be particularly useful in those scenario, because it would allow the designers to simply develop a back-end for their specific architecture.

## 1.2 Project Goals

The main goal of this project is to define the boundaries of the architecture-independent middleware that was introduced in the previous paragraph. The first objective that has to be achieved to understand which functionalities should be included in the new layer is a detailed analysis of the state of the art. In particular, two existing solutions will be considered, and both were developed within the DRES D Group at the Politecnico di Milano. The comparison between the two implementations will be the starting point for the definition of the functionalities that should be provided in a dynamically-reconfigurable environment. Those functionalities will be collected in the new middleware,

The second goal of the project work is related to the physical implementation of one of the two existing solutions. In particular, the existing operating system kernel should be recovered, in such a way that it can be executed again on the same hardware architectures on which it was implemented. Then, in order to provide a support for future developments, the old hardware architecture will be replicated using the most recent releases of synthesis tools.

The steps that have been performed to achieve those goals will be described in the following section.

## 1.3 Project Organization

This project work starts with the state of the art analysis, thus with the analysis of the two solutions proposed inside the DRES D group. After this preliminary step, the first phase of the project is the definition of the boundaries of the new layer. The following two phases are focused on the physical implementation of one of the two solutions and on the creation of an hardware support for future works.

In particular, during the second phase of the project work, the main goal was the recovery of the Caronte solution that will be described in the following sections. The main purpose of this phase is to restore the original bootstrap process and to rebuild the old kernel on a given hardware architecture. Then, during the third phase of the work, the hardware architecture was replicated using the newest versions of Xilinx synthesis tools: Integrated Software Environment (ISE) and Embedded Development Kit (EDK) version 7.1 and 9.1.

In the following sections those steps will be described in details. In particular, Section 2 describes the state of the art analysis process, while Section 3 focuses on the actual contribution of this work, which is the layer definition,

and on the work performed on the physical implementation. Finally, Section 4 introduces some possible future works.

## 2 State of the art

The second section of this document is a summary of the state of the art analysis. In particular, the analysis is focused on two projects developed inside the DRESD group of the Politecnico di Milano. Aim of those works was to provide an operating system support for Caronte and YaRA, two reconfigurable architectures that were developed inside the group.

As it was introduced in the previous section, the ultimate goal of an operating system support is to increase the level of abstraction, and thus to make the software development process easier. Without this kind of support, every embedded system requires its own standalone application, that is able to exploit the hardware resources. Obviously, such applications lack of generality and reusability, because they are tied to the specific hardware implementation. Instead, an operating system increases the generality of the software part, which does not have to handle the hardware directly.

The two works that are considered here face a common problem, which is the absence of a standard support for dynamic reconfiguration in the existing operating systems. Thus, Linux was chosen as a starting point, and new modules were developed and added to the standard kernel. Those modules will be deeply described in the following paragraphs.

### 2.1 Operating System Support

Modern embedded systems, and in particular the dynamically reconfigurable ones, are created using a large number of hardware modules, that go from a general-purpose processor to several special-purpose units. As a consequence, the software that is executed by the processor is in charge of control all the modules, and of using them properly. In order to use them, the application must call the functions exported by the libraries, which are provided by the module designer. This approach is deprecated because the application has to work at different levels of abstraction, and it must be aware of the specific drivers that are used by each hardware core.

Aim of the operating system, as said before, is to handle all the hardware issues, while the software application should be implemented as a userspace process. Thus, the operating system should provide all the features that are offered by all the common distributions, such as a process scheduling and inter-process communication. All of those features are implemented, for instance, by a common Linux distribution, which was chosen as a starting point to develop the operating system support. The choice was due to several reasons, such as the availability of the source code and the modular structure of the kernel.

The standard Linux kernel does not have a native support for dynamic reconfiguration, but it can be included by adding several modules to the kernel. The first step to provide such a support is to add a driver for the reconfiguration controller that is provided by the FPGA vendor, in such a way that it can be used as a standard peripheral. The second step is to let the system handle the registration of the cores that are currently placed on the FPGA, while the user application only access them through the standard driver mechanism. Those solutions were implemented in the first work that is described in the following paragraphs.

The solution that has been just described still requires the software application to know a lot about the reconfiguration process. In fact, it is in charge of deciding when a new module has to be placed on the FPGA and, as a consequence, of providing the correct partial bitstream to the reconfiguration controller. The second approach that is described in this document introduces a new level of abstraction, that is used to provide a common interface which is independent of the specific reconfiguration technique.

## 2.2 Caronte Solution

Aim of this paragraph is to describe the first implementation of an operating system support for Caronte[1], a reconfigurable architecture developed by the DRESD project. The proposed solution starts from a uCLinux, which is a porting of the Linux kernel for systems without Memory Management Unit, and a bootloader distributed by Avnet, the board manufacturer. Aim of the work is to introduce a reconfiguration controller driver and a core manager, which is called IP-Core Manager.

### 2.2.1 Reconfiguration controller driver

The reconfiguration controller is a dedicated hardware that manages the FPGA physical reconfiguration process. It is strictly tied to a specific FPGA, since each vendor provides its own mechanism, and different controllers may be used to reconfigure the same FPGA. The described work was implemented on a Xilinx VirtexII-Pro FPPA, which uses the Internal Configuration Access Port (ICAP) to execute the dynamic reconfiguration. In this case, the hardware controller is called ICAP Controller and it is an OPB-slave core provided by Xilinx.

In order to provide an interface between the reconfiguration controller and the user application, a standard device driver was developed and included in the kernel. Then, the controller is accessed from userspace by using the device-node mechanism, that is a standard of the Linux kernel. A device-node

is a special file (in the implementation, the reconfigurable controller driver is called `/dev/icap`) which represents a physical resource but is handled using such primitives as `open`, `close`, `read`, `write`, and `ioctl`.

A driver is basically a collection of functions, which are written according to the kernel specification. Thus, the reconfiguration controller driver is required to implement two functions that are used to perform initialization and shut-down operations, and that are called `icap_init()` and `icap_cleanup()`, respectively. Moreover, it has to implement the standard system calls, that are listed above.

Once the driver is added to the kernel, the user-application is only required to provide the partial bitstream that has to be mapped on the FPGA to the `/dev/icap` device node. However, this is also a limitation of this solution, because it requires the user application to be aware of the cores that are currently mapped on the FPGA and of all the details related to the bitstreams. For instance, the user application has to know where a partial bitstream is stored as well as where it is mapped on the FPGA. A complete framework that is able to perform the caching, the allocation and the positioning of the cores would be preferred, because it would provide a further abstraction layer to the userspace processes.

### 2.2.2 IP-Core Manager

The second innovation introduced by the described solution is the IP-Core Manager (IPCM). This module is a sort of middleware between the kernel and the cores that are dynamically mapped on the FPGA, and it provides a sort of Plug-and-Play mechanism for the hardware cores. Moreover, it makes the development of a device driver easier by exporting a smaller interface than the kernel one. This module is also located at a higher level of abstraction with respect to the reconfiguration controller driver, because it dynamically loads some device drivers, but it is not a driver itself.

The first task that has to be accomplished by the IPCM is the recognition of the addition and the removal of a reconfigurable device. In order to perform this operation, the module is connected to an hardware core, called HW-IPCM. Once a new core is recognized by the HW-IPCM, an interrupt is sent to the IPCM, which fetches the core informations from its hardware counterpart. Once the core type is known, the IPCM checks whether the corresponding driver is already loaded in the kernel and, otherwise, it is loaded. Then, the new core is added in an internal list that is managed by the IPCM, and it reserves the required I/O address space. Finally, the IPCM is also in charge of calling the initialization routine of the driver which was just loaded.

The IPCM hides all the differences between the cores to the kernel, because it registers a single major number. As a consequence, the system calls that are directed to the reconfigurable cores are actually sent to the IPCM, which forwards the request to the correct driver implementation according to the value of the minor number. Obviously, this module lacks of scalability, because it becomes inefficient as long as the number of connected cores increases. Moreover, the specific implementation allows to register at most 16 different core types, and then at most 16 cores for each type that was declared before.

The complete structure of the Caronte solution, which is composed by the IPCM and the Reconfiguration Controller Driver, is show in Figure 1.

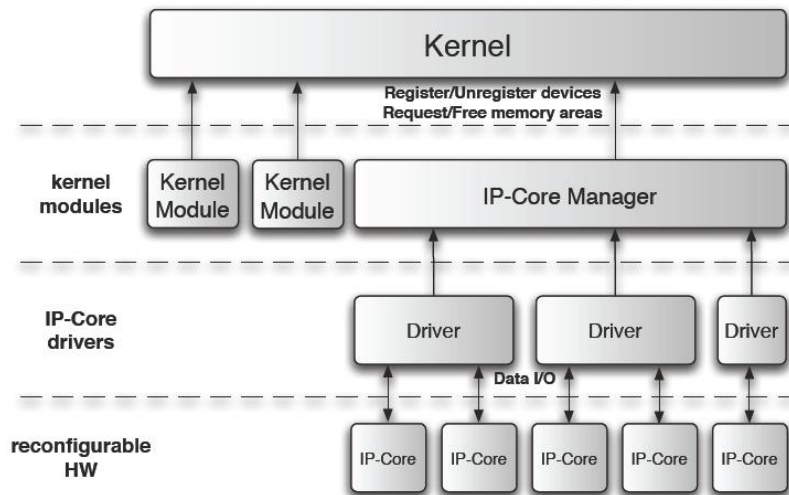


Figure 1: The Caronte solution

### 2.3 YaRA Solution

Aim of this paragraph is to describe an evolution of the operating system support that was introduced on the Caronte architecture. This solution is part of a more complex work[2], that defines a complete framework that leads to the creation of a partial dynamic reconfigurable system, starting from an high-level specification. The defined flow is called BE-DRES, and it is based on YaRA, which is the evolution of the Caronte architecture.

A phase of the BE-DRES flow is the generation of an operating system that is able to support dynamic reconfiguration. This step, that is called DRES-SW, is very important for the purpose of this document, because the

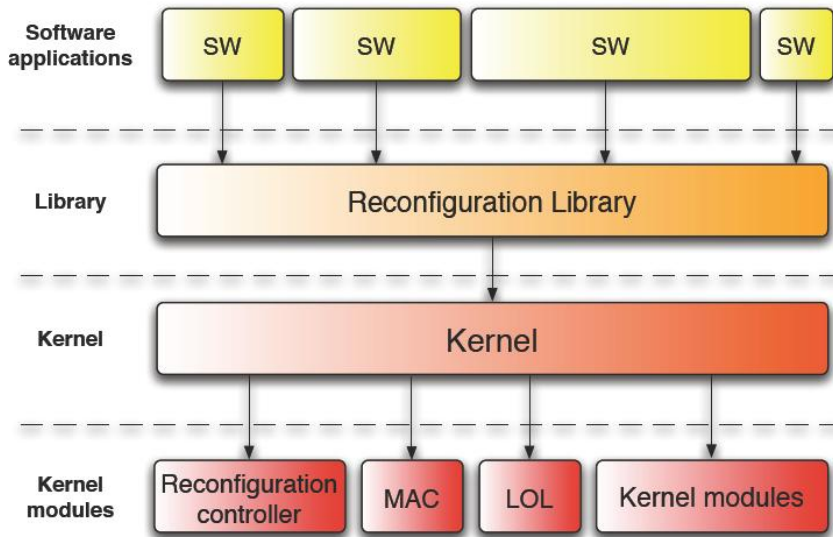


Figure 2: YaRA solution kernel modules

whole structure of the Caronte solution was modified to make the software development easier. The following paragraphs will show the main innovations that were introduced.

### 2.3.1 Kernel modules

The YaRA solution is a rather complex architecture, which is based on four modules that are added to the Linux kernel. Three of those modules are low-level ones, while the other one is basically a library, which is directly used by userspace processes. The purpose of those elements is close to the features that were included in the Caronte solution, and they are briefly described below. The complete structure of this solution, instead, is show in Figure 2.

**Reconfigurator controller kernel module** The reconfiguration controller kernel module is a software interface for the reconfiguration controller device. The most important difference between this solution and the Reconfiguration controller driver described in section 2.2.1 is the Direct Memory Access (DMA) support. DMA support simplify the reconfiguration process, because the user task does not have to send the whole partial bitstream to the controller. The application is only asked to set three registers of the controller controller, specifying the bitstream base address in the SDRAM memory, the bitstream size and the command to execute. Once the command register is

set, the controller accesses the SDRAM memory at the specified address, and then it copies the bitstream, leaving the processor free during the transfer.

**MAC kernel module** The Medium Access Control (MAC) kernel module is the part of the operating system that is in charge of manage the communication between the processor and the peripherals. In order to accomplish this aim, tha MAC kernel module sets the correct address space for each peripheral, handling the dynamic connection of a new core.

**LOL kernel module** The Load On Linux (LOL) kernel module is used to manage device registration and unregistration. In order to achieve this purpose, each time a device driver is loaded, it invokes a function on the LOL kernel module to inform it of the new addition. The LOL module is in charge of registering the major number of the new device and of making it accessibleby the userspace applications.

**Reconfiguration Library** The reconfiguration library is a C library that is directly used by userspace tasks to exploit the previous modules. In particular, this library is implemented in order to simplify the access to some standard operation (e.g. `read`, `write`, and `ioctl`) on the reconfiguration controller device and the MAC module. The ultimate goal is to simplify the whole reconfiguration process by allowing user applications to complete a reconfiguration process and to assign an address to the new cores by invoking only few functions.

### 2.3.2 ROTFL Architecture

The kernel architecture described in the previous section provides enough support to reconfigure part of the system by specifying the correct bitstream and by assigning the correct address space to the new cores. The further step that is implemented by the YaRA solution is a centralized support for the whole reconfiguration process. This module allows the userspace task to request for a particular device and, as a response, the name of such device is provided to the task. As a consequence, the software application does not know anything about the low-level details, such as whether a device is already on the FPGA or it has to be dynamically added.

The described support is implemented by the ROTFL Architecture, which is explained in this paragraph. The main advandage of this architecture is the addition of another high-level support, that makes the userspace applications completly unaware of the underlying architecture. The structure of

the ROTFL Architecture is divided into three parts: a library, a daemon and a repository, that are described below. The complete structure is shown in Figure 3.

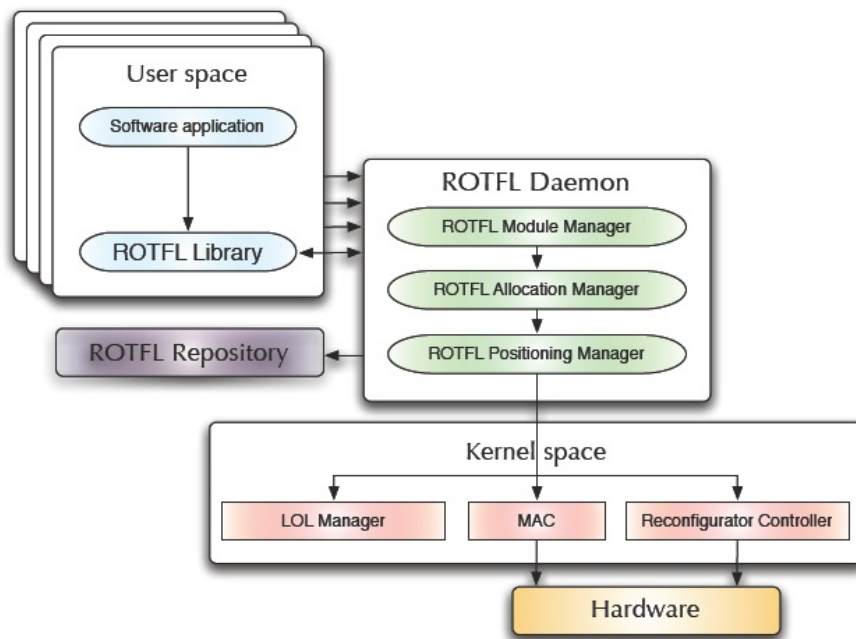


Figure 3: ROTFL Architecture

**ROTFL Library** The ROTFL Library is an intermediate layer between the userspace tasks and the ROTFL daemon, and it is the only part of the ROTFL architecture that is directly accessed by the software applications. It exports a set of functions that allow the user process to require the addition or the removal of a core by simply specifying its name. All the requests lead to the creation of a dedicated socket communication with the daemon, in such a way that the task is unaware of it.

**ROTFL Daemon** The ROTFL daemon is a kernel-space process that is organized as a hierarchy of sub-modules. Each sub-module, that is generically called "manager", tries to accomplish its task by itself and, if it is unable to do so, it forwards the request to the next manager. The module manager keeps a list of all the available devices and, if the userspace applications request a module that is already in the list, the pointer to the existing instance is returned. Otherwise, the device has to be loaded in the FPGA, so the request

is forwarded to the allocation manager. This second manager is able to find an FPGA area where the new module can be placed, and then it forwards the location to the positioning manager. The positioning manager performs a query on the ROTFL Repository, in order to find the correct bitstream. Then, it communicates with all the kernel modules that were described in section 2.3.1, performing all the reconfiguration tasks.

**ROTFL Repository** The ROTFL Repository is simply in charge of providing informations about the modules to the ROTFL Daemon when it requests a particular core.

## 2.4 Conclusions

The state of the art analysis that was performed in this section shows that the YaRA solution, which was proposed after the Caronte solution, has a very modular structure. In particular, each logical function of the system, as for instance the device driver loading or the address space assignment, is performed by a single module. Thus, this structure can be used as a starting point to extract the common functionalities of an intermediate layer. The Caronte solution can be used to validate the results obtained from the analysis of the YaRA solution, because those abstract functionalities should be found also in the Caronte structure, even though that the implementation would be slightly different.

The YaRA solution is not a good starting point for the implementation analysis, instead, because it was implemented on a multi-FPGA board called RAPTOR2000. This project work, on the other hand, will be implemented on a single-FPGA board including a Virtex II - Pro VP7. This is the same platform that was used by the Caronte solution, and thus that implementation will be recovered.

## 3 Project Details

This section describes the three phases of this project work. The first paragraph describes the real innovative contribution that is proposed, which is the layer definition. The YaRA solution has been chosen as a starting point to understand which functionalities are required in dynamically-reconfigurable systems. Then, the other implementation, which is the Caronte one, will be recovered. In particular, the second paragraph will show the recovery of the bootstrap process, given a pre-synthesized hardware architecture. Then, the hardware architecture replication will be described in the third paragraph, in which some numeric results will also be shown.

### 3.1 First Phase: Intermediate layer definition

Aim of this paragraph is to provide a first definition of the new intermediate and architecture-independent layer that should be introduced in the operating system support for a dynamic-reconfigurable environment. The definition is a mixture of existing features that were implemented in the past, that need to be factorized, and of new solutions that should be added to enrich the reconfiguration support.

The first paragraph shows how current features should be collected to provide an architecture-independent support, while the second paragraph describes how they are mapped on the two existing solutions. Then, a possible new feature that should be developed in the future is described in the third paragraph.

#### 3.1.1 Factorization of YaRA solution

The existing implementations offer a set of functionalities that can be included in an architecture-independent layer. In particular, the YaRA solution is a more suitable starting point than the Caronte solution, because it is more structured and modular, even though it includes the same features. The modules included in the YaRA solution will be now analyzed, trying to understand which features can be collected in the intermediate layer.

**Reconfigurable Controller Driver** The reconfigurable controller driver is per se an architecture-dependent component, because different controllers can be used according to what the FPGA vendor provides and to what kind of buses are used. For instance, on Xilinx FPGAs the Internal Configuration

Access Port (ICAP) is used as a reconfiguration mean, but different controllers for this port are available, both for PLB and OPB bus. Thus, the driver is a back-end that must be provided by the architecture designer, and the intermediate layer can export at most an interface to it.

**Medium Access Control** The MAC module is able to reserve an address space for a dynamically loaded module. Even though that the implementation of this module is related to a physical MAC device, the ability of allocating an address space at runtime is a feature that must be included in a system that aims to deal with different modules instantiated during the execution.

**Load On Linux** The LOL module is used to manage the dynamic registration of devices, which also includes the ability of loading the proper driver. This is also the key feature of the IPCM module that was implemented in the DRES D solution. Of course, since this feature does not depend on the specific hardware, it is immediate to include it in the architecture-independent layer.

**Reconfiguration Library** The reconfiguration library is an user-space library that exports a simple interface to the user applications that allows them to interact with the Reconfigurable Controller module and with the MAC module. As for the LOL module, also the reconfiguration library does not rely on a specific hardware, so in principle it can be part of the intermediate layer. However, a deeper investigation is needed to understand how much a specific reconfigurable controller influences the interface to the user applications.

**ROTFL Architecture** The ROTFL architecture is used to perform a sort of module caching system, which hides to the user applications the request of a new module, the interactions with it and its removal. In the YaRA implementation, those tasks are done using a highly centralized approach, because a daemon is used to receive the requests from user applications. In the Caronte solution, on the other hand, this feature is not implemented. The caching mechanism provided by the ROTFL architecture is an essential part of the new intermediate layer, because it also handles bitstream relocation and it guarantees hardware reuse.

Each module that has been described above tries to satisfy a particular functionality. Similar functionalities are also solved by the Caronte solution, even though that some of them are actually provided by the same module.

In the next paragraph, all those functionalities will be exposed, and then it will be possible to understand how they are implemented in the two existing solutions.

### 3.1.2 Mapping of common functionalities on the two implementations

The analysis that was carried out in the previous paragraph shows that there are some functionalities that are provided both by the YaRA and the Caronte solutions. As a consequence, it is immediate to conclude that they should be part of a new intermediate layer.

In particular, five possible features that an architecture-independent layer should provide are listed below. A summary of this analysis is reported in Table 1.

**Reconfiguration controller support** A reconfiguration controller driver is required to interact with the physical device that performs the reconfiguration process. It is implemented in both the YaRA solution and the Caronte solution (in which is known as "ICAP controller") as a device driver. Obviously, since it is a device driver, it is mainly a back-end of the specific architecture. However, it should be possible to define a common interface in order to simplify the driver implementation and to make the access to the device uniform.

**Dynamic address space assignment** The dynamic address space assignment is required whenever a new IP-Core is allocated on the hardware architecture, otherwise the processor would not be able to communicate with the new module. This feature also relies on a hardware counterpart, as it happens in the MAC module in the YaRA solution. In the Caronte solution, this functionality is provided by the IPCM module, which also communicates with an hardware IPCM core.

**Device registration and driver loading** Each time a new physical core is loaded in the hardware architecture, the corresponding device driver has to be loaded in the operating system kernel, and the device should be registered. This phase only deals with device drivers, thus it does not need to interact with an hardware module. In the YaRA solution, this functionality is provided by the LOL module, while in the Caronte solution it is performed by the IPCM module.

Feature	Caronte solution	YaRA solution
Reconfiguration controller support	ICAP device driver	Reconfiguration controller driver
Dynamic address space assignment	IPCM module	MAC module
Device registration and driver loading	IPCM module	LOL module
Interface to user applications	Direct interaction with modules	Reconfiguration library
Module management	Not implemented	ROTFL Architecture

Table 1: Mapping of the layer functionalities on the existing solutions

**Interface to user applications** Since the main purpose of an operating system is to simplify the software development process, a set of functions to exploit dynamic reconfiguration should be exported to the userspace applications. This is exactly the purpose of the Reconfiguration Library that is implemented in the YaRA solution. However, this functionality is not strictly required in order to provide a support for the whole reconfiguration process, and for this reason the Caronte solution does not implement it. In this scenario, userspace applications are required to directly interact with the device driver.

**Module management** The module management functionality includes all the advanced, high-level functionalities that are performed by the ROTFL Architecture, which is part of the YaRA solution. Those features include module allocation and removal, module placement and relocation, and module caching, which were not implemented in the Caronte solution. This layer provides a further abstraction level to userspace application, and thus it should be considered in the future because of the large number of feature that can be added. A possible extension, which is the reconfiguration scheduler, is described in the following paragraph.

### 3.1.3 Reconfiguration scheduler

The ROTFL architecture, which was described in the previous paragraph, reacts to a request by looking for the specific module and, whenever it is not instantiated, it uses the correct bitstream to load the module on a free area of the FPGA.

A new possible solution that can be included in the intermediate layer is a sort of scheduler, which can optimize the number of required reconfigurations

and which can hide the latency due to the reconfiguration phase. Up to now, this kind of mechanism is not included neither in the Caronte or in the YaRA solution, but it can be implemented later. Of course, since it does not deal with a specific architecture, but it only knows which modules are available and which ones can be instantiated, it can be considered as architecture-independent.

## 3.2 Second phase: Caronte implementation analysis

The second phase of the project work, which is described in this section, was performed on the existing implementation of operating system support for the Caronte architecture.

The Caronte solution is the only implementation that will be considered during this phase, because it was originally implemented on Xilinx VirtexII-Pro FPGAs, which are the same that are used in this project. The YaRA solution, instead, was developed on a multi-FPGA architecture, and its porting on a single-FPGA system will be possible once the hardware has been upgraded using up-to-date tools.

The following paragraphs will provide a description the recovery of the original bootstrap process, which is required to load the kernel of the Caronte solution. Then, a brief description of the cross-compilation process will be provided to understand how the kernel image has been built from scratch.

### 3.2.1 Bootstrap from flash memory

The first issue of the implementation is the bootstrap of the operating system, which is the process that loads a kernel into memory, in such a way that it is executable by the processor. Since the kernel image is too large to be stored on the RAM Blocks (BRAMs) of the FPGA, and since other kinds of memories are not persistent (which is the case of SRAM and SDRAM memories), it is usually stored on flash memory. However, in order to be executed, the kernel needs to be moved from the flash memory to a faster memory, such as the SDRAM, which guarantees a good tradeoff between size and speed. The steps that are required to transfer the kernel from flash to SDRAM memory are here described.

When the system starts, a minimal program code, which is called "bootloader", is executed by the general purpose processor (in this case, a PowerPC processor). The bootloader code is small enough to store it directly on BRAMs, because it only has to access the flash memory at a fixed address and copy a memory block to the SDRAM memory. The block is supposed to contain the executable image of another program, which is called "boot-

manager””. So, when the bootmanager is loaded, the bootloader sets the program counter to the first SDRAM address it wrote, so the bootmanager can start its execution.

The bootmanager is more complex than the bootloader, since it has to load the whole kernel image into SDRAM memory and it has to initiate it. Moreover, the bootmanager is an interactive program that allows the user to provide inputs using a command line interface, while the bootloader is only able to print a welcome message on the serial interface and on the leds. If no input comes, the bootmanager simply loads the kernel and a file system image (which is called RAM disk) into SDRAM memory, by copying a given number of flash blocks. Then, the kernel is executed, and the bootstrap process completes.

### 3.2.2 Bootstrap process on Avnet boards

The Virtex II-Pro FPGA that is used in this project is part of the Virtex II-Pro Evaluation Board, which is distributed by Avnet. Since the board is stackable and it does not include either the SRAM and the flash memories, the Communication/Memory Module was added. This module also adds an Ethernet interface that will be used later to program the flash memory.

While the bootloader program was written inside the DRES D group[3], the boot manager and in general the whole bootstrap process are tied to the implementation provided by Avnet, that is described in this paragraph.

The Avnet implementation[4] relies on a 16MB flash memory that is located on the Communication/Memory module, whose address space ranges from 0xE4000000 to 0xE4FFFFFF. All flash blocks are 256KB in size, and there are 64 of them. The organization of this space is shown in Figure 4. The bootmanager is located at 0xE4F80000, and the bootloader is implemented to access that address. Moreover, the bootmanager is aware that the kernel image is stored from address 0xE4000000 and, since its size is not fixed, twelve memory blocks are always copied to SDRAM memory.

The board includes 64MB of SDRAM memory whose address space ranges from 0x00000000 to 0x03FFFFFF. This space is used to store data loaded from flash memory because of two reasons: first of all, the SDRAM is fast enough to be used as an instruction memory. Moreover, since flash and SRAM memories are both on the Communication/Memory module, they share the same address and data channels, and thus the transfer would be harder to implement. That aspect will be recalled later, when the hardware architecture will be analyzed. In the original Avnet solution, SDRAM memory was only used to store the kernel image, while the bootmanager was loaded in a 1MB bank of SRAM memory, whose address space ranges be-

tween 0xD8000000 and 0xD80FFFFFF. This is no longer true in the DRES D bootloader, since also the bootmanager is stored in SDRAM memory.

Because of the address space described above, the bootstrap process on the Avnet Virtex II-Pro Evaluation Board can be summarized as follows:

- The DRES D bootloader accesses the flash memory at address 0xE4F80000 and copies one block (256KB) to address 0x00800000 in SDRAM memory.
- Once the copy is completed, the bootloader jumps to 0x00800000 and the bootmanager is executed.
- The bootmanager accesses the flash memory at the base address 0xE4000000 and copies twelve blocks to SDRAM memory, starting from address 0x00000000.
- The kernel can be executed from SDRAM memory.

Up to now, no caching techniques are actually used in the architecture, but they can be easily implemented to improve performance. It is also important to notice that the flash memory controller that is synthesized on the hardware architecture does not allow the user application to directly write on flash blocks. Thus, the flash memory can be seen as a read-only memory, and it can be written using ad-hoc techniques.

In the following paragraphs a technique to write on flash memory based on Avnet tools will be shown. This procedure was actually used during the project, since neither the bootmanager and a kernel image were originally available on flash memory.

### 3.2.3 Avmon

A possible solution to write data on flash memory is Avmon[5], which is a program distributed by Avnet. It is compiled to work on a standard architecture, which includes the memory structure described in the previous paragraph, and some additional features, as for instance an Ethernet controller. The bitstream of the architecture is also pre-synthesized, so it can be directly used to program the FPGA and to execute Avmon. Aim of this paragraph is to describe what Avmon is and which steps should be performed to use it properly to store both a kernel image or the bootmanager.

Avmon is a command line interface that interacts with the user, providing a set of tools to test memory regions, as well as configure the Ethernet profile of the FPGA and jump to a specific address. It also allow the user to write on the flash memory by removing the write-lock on it, writing one or more

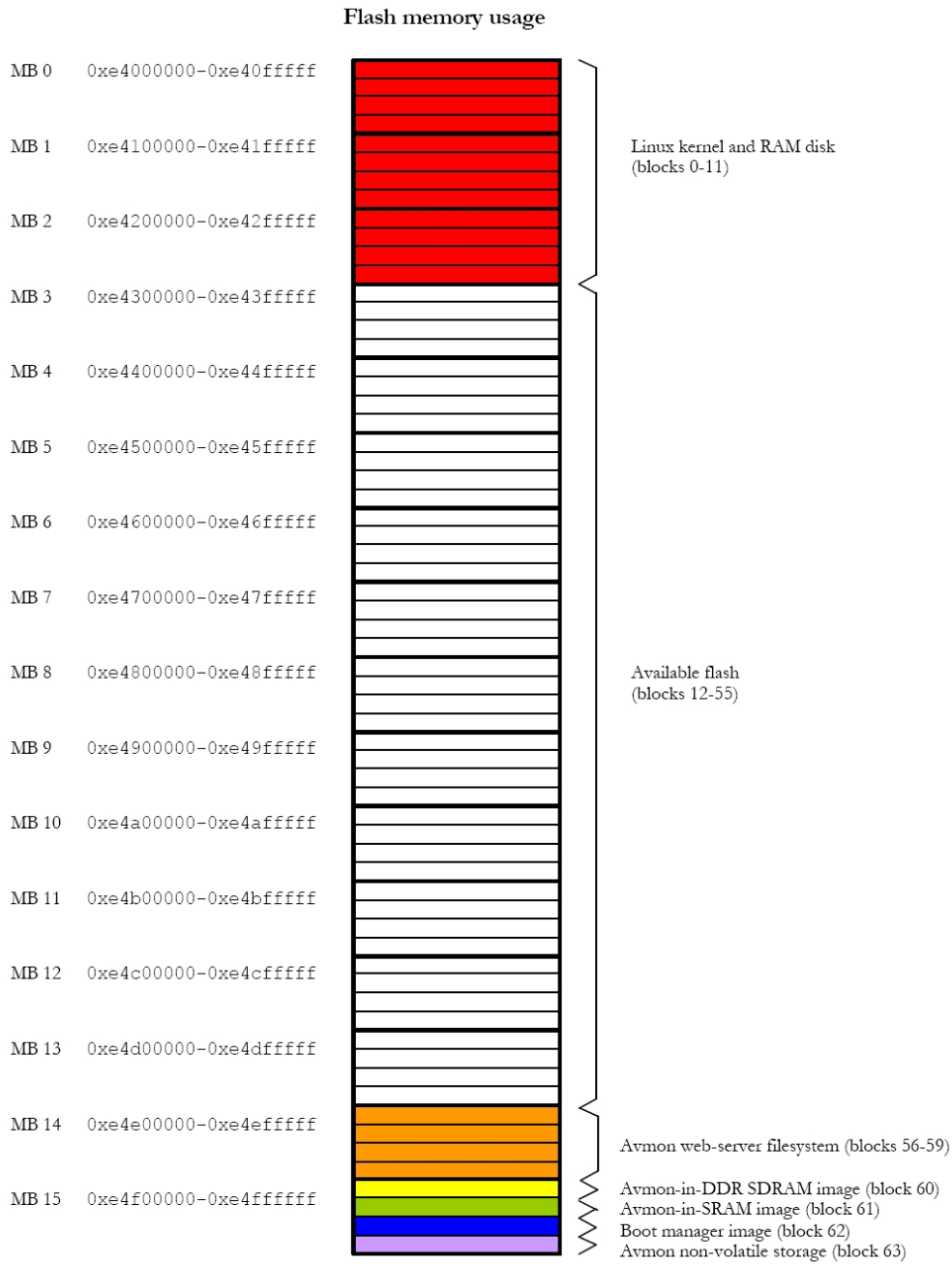


Figure 4: Flash memory layout on Avnet Virtex II-Pro Evaluation Board

block, and then restore the lock. Thus, the user is supposed to write data on a volatile memory, and then to invoke the proper command to transfer it to the flash.

In order to make Avmon work on an FPGA, it must be downloaded manually, since it is too large to be directly written on BRAMs. Then, while it is running on the processor, the bootmanager or the kernel images have to be stored on a memory. In this project, the first task was performed using the serial interface, while the second one was performed by means of the Ethernet interface. The steps that are necessary to achieve those tasks are listed below:

- The FPGA must be programmed using the provided bitstream, in such a way that the memory layout is the same as in Figure 4. The configuration is done using the JTAG interface, which must be released after the bistream download, since it is required by the next step.
- A debugger must be used to download the Avmon image on the SDRAM memory. Xilinx provides a debug interface called 'XMD': once it is executed, two commands have to be invoked by the user, as shown below.

```
XMD% ppc
XMD% dow {Path_to_executable_file}/avmon_dds.elf
XMD% con 0x00100000
```

The first command is used to establish a connection with the PowerPC processor that is included in the architecture, while the second one performs the download of the executable image of Avmon. The third command forces the processor to continue the execution from the base address of the Avmon image. There is no need to specify the memory address where the image should be store, since it is included in the header of the *.elf* file.

- Avmon is now executing, so the image of either the kernel or the bootmanager should be stored in such a memory as the SDRAM, which provides enough space to complete the task. In order to perform this operation, the Ethernet interface is usually preferred, so a connection must be established between the host computer and the FPGA by setting IP addresses properly. Once the network is configured, it is simple to download the image (which can be either an *.elf* or a *.mem* file) by means of a command line script provided by Avnet. The script is also able to invoke the *flash copy* command that stores the image in SDRAM memory on flash. Of course, the script must be modified

according to the IP address of the FPGA, to the size of the image that has to be downloaded, and to the flash memory address where the image should be stored.

Once those steps are completed, the bootmanager and the kernel are persistently stored on flash memory, and can be loaded at any time by the bootloader. However, as said before, the hardware architecture that was used to execute Avmon was pre-synthesized, while it would be preferred to have a set of projects that can be modified and synthesized by the user, using different versions of the same tools. Later in this document a description of the main issues to deal with while trying to achieve this goal is provided.

#### 3.2.4 ELDK

The bootstrap recovery process that was described in the previous paragraphs was performed using an existing executable kernel image, which was compiled for the same hardware architecture that has been used. Another step of the project work is to build the same kernel from scratch, and then to add the two additional module that were implemented in the Caronte solution: the ICAP device driver and the IPCM module.

In order to rebuild the kernel image, a software development environment called *Embedded Linux Development Kit* (ELDK) [6] was used. This environment provides a cross-compiler for the PowerPC processor used in the hardware architecture, which can be used to compile the linux kernel on a generic platform. The provided version of ELDK also contained a modified version of the uCLinux kernel and a set of device drivers that easily allow the user to compile the kernel for an Avnet Virtex II - Pro board.

While the rebuilding process of the original kernel is quite straightforward, the linking of the two additional modules could not be performed at compile-time. The only way to add them to the kernel is by downloading them on the FPGA using a TFP Ethernet connection, similarly to what is described in the previous paragraphs. Once the modules have been downloaded on the FPGA, they can be easily linked to the kernel by means of a shell script that was provided with the original implementation.

### 3.3 Hardware architecture

The existing architectures, on which the DRESD operating system solution was implemented and Avmon was executed, lack of portability and reusability. They were indeed created and synthesized using Xilinx ISE and EDK 6.1, and they are no longer supported by recent versions, i.e. 9.1 and 8.2.

Thus, it is necessary to build them from scratch using up-to-date tools, trying to preserve the main properties of the original version, such as the address space and the Ethernet connection. This paragraph summarizes which cores have been included in the architecture, with some additional details, and the main problems that the design has to deal with.

The hardware architecture reflects the original implementations structure, and is composed by the following elements:

- A PowerPC 405 processor and its communication infrastructure, which includes a PLB bus for fast peripherals, an OPB bus for slow peripherals, and a bridge between the two.
- A 64MB DDR-SDRAM memory, which is connected as a slave on the PLB bus with an address space that ranges between 0x00000000 and 0x03FFFFFF.
- A 1MB SRAM memory, which is controlled by a generic External Memory Controller (*emc*) core connected as a slave to the PLB bus. It is organized as a single block and its address space starts at 0xD8000000 and ends at 0xD80FFFFFF.
- A 16MB Flash memory, which is also controlled by an *emc* core with a slave connection to the PLB bus. It is also organized as a single block memory, whose address space ranges between 0xE4000000 and 0xE4FFFFFF.
- An Ethernet controller called 'emac', which is a proprietary core distributed with Xilinx ISE under an evaluation licence.
- Some general purpose input/output peripherals, such as LEDs and display, that are typically used to show that the system works correctly. They are also useful to perform small-bit dynamic reconfiguration, because a single bit modification in the original bitstream can turn on or off one of the LEDs.

SRAM and flash memory are both controlled by the *emc* core. However, since the Communication/Memory Module lacks of an exhaustive documentation, the parameters required by the up-to-date core (2.00 version) cannot be found. Thus, the old core (1.10.a version), which was used in the original architecture and whose parameters are known, can be imported in the project.

The architecture that is described above should be created using Xilinx EDK, but the standard EDK flow, which is called 'XST', cannot be used to

Resource	Used	Available	%
Slices	4926	4928	99%
Flip-Flops	5217	9856	52%
4-Input LUTs	6974	9856	70%

Table 2: Area requirements for the architecture generated by ISE 7.1

synthesize the bitstream. This is due to the fact that the SRAM and the flash memory shares the same data and address pins, and XST cannot deal with this situation. Thus, the only synthesis flow that can be used is the ISE flow. In this way, a top-level VHDL file can be created, and a multiplexer for each shared signal can be added.

Another important issue that comes out during this phase is the lack of a device driver support for the newest core versions. This makes it impossible to rebuild a complete kernel image, and thus to port the Caronte solution on this architectures. As a consequence, the results of this phase are several hardware architectures, some of which does not have a working operating system support, yet.

### 3.3.1 Results

Different hardware architectures were synthesized during this project work by means of different versions of Xilinx tools. As described in the previous paragraph, the newest versions lack of a device driver support, thus a kernel image could not be compiled for them. In particular, the hardware architectures that are generated using 8.2 and 9.1 versions of Xilinx ISE and EDK introduces too many updated cores that are not backward-compatible, thus making the kernel building not feasible. On the other hand, version 7.1 generated an hardware architecture that was fully compatible with the existing device drivers.

In this section, some synthesis results are provided. Because of the issues that are listed above, a comparison will be performed among the architecture generated by ISE 7.1 (which is compatible with the existing device drivers) and the one generated by ISE 9.1 (which does not have an valid operating system support). The synthesis summaries are reported in Table 2 and Table 3, which show how many resources are required by each architecture.

The two tables show how both the hardware architectures barely fit in the FPGA logic cells. In particular, there are two main reasons that can explain the high area requirements. The first one is the Ethernet controller IP-Core, which uses approximately 40% of the total number of slices. The second explanation is due to the presence of a fixed top-level design, which

Resource	Used	Available	%
Slices	5318	4928	107%
Flip-Flops	5724	9856	58%
4-Input LUTs	6993	9856	70%

Table 3: Area requirements for the architecture generated by ISE 9.1

limits any improvement effort.

Actually, the hardware architecture that was synthesized using ISE 9.1 is too large for the VP7 FPGA. This problem may be solved in the future by means of a more strict set of constraints. However, a reasonable conclusion is that the resulting architecture would still be too large, and it cannot be used for a module-based dynamic-reconfigurable architecture, which requires a large amount of area for reconfigurable modules. A similar problem has already been described in [1], and a solution was found by removing the Ethernet controller and by using a larger FPGA, such as Virtex II - Pro VP20.

## 4 Future works

The previous sections described all the steps that have been performed during the project work. This final section shows some possible future works that can be done starting from the results that have been obtained.

Several possible hints for future works have already been proposed in the document. First of all, the main goal of this project is the definition of the boundaries of a new architecture-independent middleware, but not its implementation. The implementation phase is left as a future work, because it will introduce some additional issues that must be addressed. For instance, it requires a deep analysis of the existing source codes, in order to understand whether they can be reused, instead of implementing the middleware from scratch. The YaRA solution can be used as a starting point, even though that the code should be adapted to extract the architecture-independent features.

Another possible future work that has already been described in this document is the addition of some additional features that are related to the module management. In particular, a reconfiguration scheduler should be added to improve performances by hiding the reconfiguration latencies.

Other possible works are related to the physical implementation, and in particular to the generation of hardware architectures using up-to-date synthesis tools. In particular, it would be useful to retrieve updated device drivers for the newest core releases. Since the driver upgrading is a common problem, it could be useful to define a simplified interface to develop new device drivers for reconfigurable cores. A sort of common stub should improve the driver portability on different hardware architectures.

Finally, in order to provide a unified hardware/software codesign flow, the new layer should be integrated in an existing synthesis flow, such as Caronte. In the ideal scenario, after the hardware platform has been synthesized, the corresponding operating system kernel should be automatically generated using an architecture-independent support and a minimal architecture back-end.

## References

- [1] A. Donato, "A software platform to support dynamically reconfigurable systems-on-chip under the gnu/linux operating system," Master Thesis, Politecnico di Milano, July 2005.
- [2] V. Rana, "A novel methodology for dynamically reconfigurable embedded systems design," Master Thesis, Politecnico di Milano, September 2006.
- [3] V. Frascino, "Implementazione di un sistema di gestione per un ip-core in ambiente gnu/linux embedded: Infrared data association," Master's thesis, Politecnico di Milano, 2005.
- [4] *Xilinx Virtex-II Pro Evaluation Board Support Package*, Rev 1.0 ed., Avnet, Inc., October 2003.
- [5] *A user's guide to Avmon*, Rev 1.0 ed., Avnet, Inc., June 2003.
- [6] *DENX Embedded Linux Development Kit*, Rev. 2.0 ed., DENX Software Engineering, November 2002. [Online]. Available: [www.denx.de](http://www.denx.de)