Diploma thesis:
Design and Implementation of a Linux-based Dynamic Reconfiguration Task Manager in FPGAs

by

Spanakis Emmanouil

Department of Electronic and Computer Engineering

Technical University of Crete

Chania,October 2009

Advisor Professor : Prof. Apostolos Dollas

Evaluation Committee:

Professor Apostolos Dollas

Associate Professor Dionisios Pnevmatikatos

Assistant Professor Ioannis Papaefstathiou

Abstract

The advent of reconfigurable computing has benefited numerous application domains. Its inherent parallelism along with the in-the-field customization offers an unprecedented opportunity to exploit the features characterizing each application. Field Programmable Gate Arrays (FPGAs) are some of the stronger representatives of reconfigurable computing. The SRAM-based nature of many FPGA families allows for their potentially unlimited programming similar to the way the traditional memories are programmed. A technology using this inherent characteristic that has been incorporated in some specific FPGA families is called partial reconfiguration. It allows for the reprogramming of part(s) of the FPGA chip without disturbing the rest of its operation even during run-time.

FPGAs have been widely adopted in embedded systems. Partial reconfiguration technology can leverage these systems by swapping in and out task modules that are not needed to co-exist in the chip at the same time. A task module can be downloaded only when needed during the system operation, i.e. on demand, allowing for area savings and potentially low power consumption. To this direction the available hardware should be managed in an effective way regarding efficiency and ease-of-use. This can be offered by an operating system. Several embedded systems have been equipped with an OS running on a software processor, but, none to limited research has been conducted on OS that control the partial reconfiguration of the hardware. In particular, an OS can help to simplify the interaction between the software application and the IP cores in the same way a traditional OS manages the access to the I/O peripherals. The task management can be carried out by the OS running on a

processor coupled with the reconfigurable fabric within the same chip in order to keep the development complexity low and also to relieve the developer from the difficult task to deal with the details of the hardware and the complex partial reconfiguration technology.

Acknowledgements

I would like to thank my advisor, Prof. Apostolos Dollas for giving me the opportunity to work on this very interesting field of research and for his support and precious contribution throughout the elaboration of this thesis.

Also, I would like to thank Associate Professor Dionisios Pnevmatikatos and Assistant Professor Iwannis Papaefstathiou who agreed to evaluate my diploma thesis.

I would like to thank PhD student Kiprianos Papadimitriou for his continued interest, support and tutoring throughtout this diploma thesis. His valuable opinions were very important for the develpment of the final system.

Moreover, i would like to thank George Nikoloudakis for his help and contribution to my thesis.

I would like to thank all my friends for these great years we spent together and for many wonderful memories.

Most of all, I would like to thank my family for their enormous help, understanding and support throughout all these years as a student.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, an introduction is made about FPGAs, embedded operating systems, partial reconfiguration and their advantages.

1.1 Field Programmable Gate Arrays

FPGAs are an integrated circuits that can be easily configured by the end-user to execute many different applications. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC). FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, and the low non-recurring engineering costs relative to an ASIC design (not withstanding the generally higher unit cost), offer advantages for many applications. In contrast, FPGAs have generally been slower, less energy efficient and achieved less functionality than theis ASIC counterparts. However, the bridge is quickly closing due to improvements in technology of semi-conductors.

FPGAs contain programmable logic components called "configurable logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together". Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Many FPGAs contain embedded processors allowing for great flexibility. The generic structure of an FPGA is seen in Figure 1.1

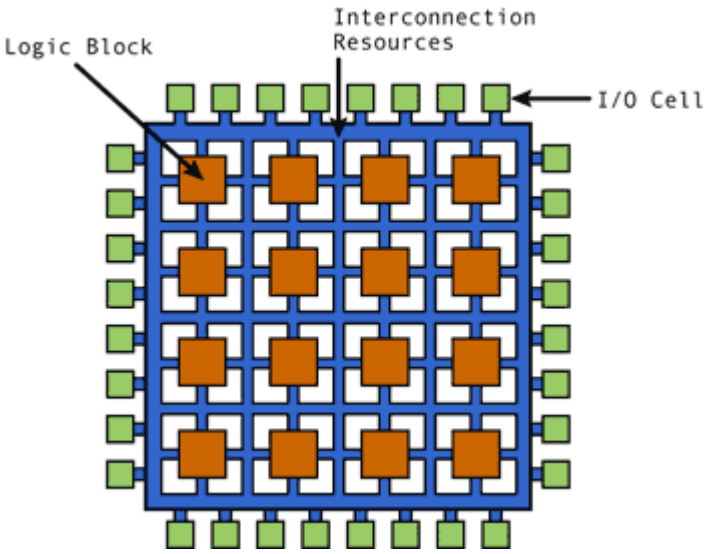

Figure 1.1: Internal structure of an FPGA

Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy and a growing range of other domains.

1.2 Embedded Operating Systems

An operating system for embedded computer systems undertake tasks such as memory management, task scheduling, hardware device interaction, handling filesystems and data, and allows for user-system interaction through a graphical or command line interface. Such operating systems are especially designed to be compact and efficient, excluding many functions, provided by traditional operating systems, or providing alternatives with reduced functionality. They are also possible to integrate real time support, making in order to run real-time applications. There is a wide variety of embedded operating systems [14]. For the purpose of this thesis, the embedded Linux was used due to its widespread use [9].

Embedded systems generally have in general limited resources available; they incorporate much less RAM and secondary storage than desktop computers and are likely to use flash memory instead of a hard drive. Embedded Linux can be configured and optimized to target specific hardware configurations and usage situations. These optimizations can include reducing the number of device drivers and software applications as well as offering alternatives, thus reducing the kernel size and making it more lightweight. Some of the advantages of using Linux in embedded systems are listed below [36]:

- The open source, universal and Posix[1]-compliant nature of Linux, offers the capability of developing highly customized and highly portable applications, with reduced development and deployment costs and without having to "lock-in" to a single proprietary supplier. Development and debugging can be performed with GNU C and C++ compilers and the GNU debugger. The GNU debugger can be used in various connection schemes in combination with the Chipscope Pro Logic Analyzer [5] to offer full debug visibility.

- Offers a range of powerful capabilities including memory protection, processes, threads and extensive networking facilities

- Dynamically linked device drivers may not be covered by GPL [22] and so, their source code need not be disclosed with binary versions of the driver. Moreover, Glibc[2] is licensed under lesser GPL [23]. As most applications link to Glibc to access the required kernel functionality, they are free from GPL licensing requirements. This flexible licensing model enables the mixing of proprietary device drivers and application code with the Linux kernel and libraries, which opens up a wider set of applications to use on Linux.

- It provides the capability of dynamically loading driver modules on demand. This allows for a small kernel which boots faster but still offers the required functionality needed at the time.

- Existing device drivers for hardware peripheral IP cores, developed by Xilinx,

[1]Posix is the name of a family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system

[2]Glibc is an important system call library used by most applications to provide an interface to the kernel

can be easily adapted to higher OS functions due to their modular and generic properties as shown in Figure 1.2

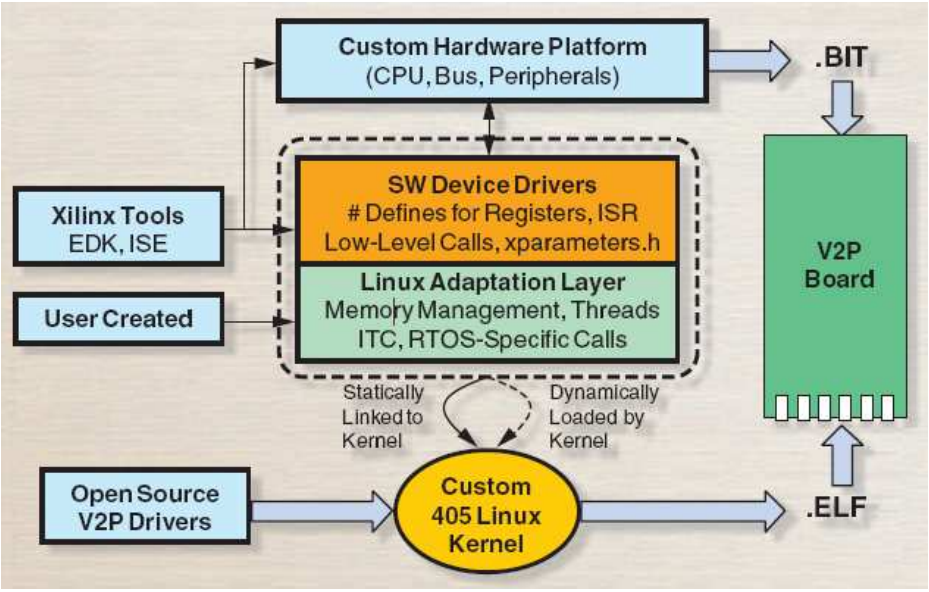

Figure 1.2: Linux device drivers can be built on top of lower-level drivers provided by Xilinx

1.3 Partial Reconfiguration

Partial Reconfiguration (PR) provides the ability to reconfigure selected areas of an FPGA. This can be done either when the design is operational and the device is active (known as dynamic partial reconfiguration), or when the device is inactive(known as static partial reconfiguration). PR gives the ability to adapt hardware algorithms, share the same hardware amongst different applications, increase resource utilization, and to update hardware remotely.

Some of the benefits of PR are:

- **Increased System Performance:** PR provides support for different versions of hardware designs, ready to be instantiated at any time. Thus, it is possible to use optimized versions of a design for different situations, while the rest of the system remains unaffected and continues to run without performance loss.

- **Reduced Power Consumption:** Power consumption can be reduced by simply loading a less power-consuming version of the design, or a blank bitstream when the particular device is not needed, therefore reducing static leakages.

- **Adaptability:** PR-supporting systems can adapt to changes in their environment, their input data or their mission specifications. This capability makes the system more efficient as compared to a generic one, which cannot be optimal for a number of different situations.

- **Self Test and upgradability:** Self test components can be instantiated on demand to check the integrity of the system and, if deemed necessary, the faulty module can be reconfigured to resume its correct operation. Hardware updates are also simplified, without the need for re-configuring the entire device. This can be performed either locally or remotely.

- **Hardware Virtualization:** PR provides the user with a number of functions ready to be used, that can be either present in the hardware or ready to be instantiated. This capability allows for smaller and cheaper FPGAs to

be used, as hardware modules can be swapped-in to fit the user's needs.

Many Xilinx FPGAs support partial reconfiguration, from the Virtex 5 to the low-end Spartan 3/E family. For the purpose of this thesis, we used a XUP Virtex-II PRO FPGA based board from Digilent. Xilinx supports two basic styles of partial reconfiguration, module-based and difference-based. In this present thesis, we used the module-based approach.

1.3.1 Module Based Partial Reconfiguration

Module-based partial reconfiguration uses modular design concepts to reconfigure large blocks of logic. These distinct blocks are known as partially reconfigurable modules (PRM). These modules are loaded in special regions, defined statically, called partially reconfigurable regions (PRR). Partial bitstreams, which are created during system design for each PRM, are used to reconfigure the region and a blank bitstream is used to recant any previous configuration. A default PRM is used during system synthesis and initial configuration. This design flow introduces a number of limitations listed below:

- Communication access to the PRR is performed through special pre-defined static buses called Bus Macros. They ensure that after the PR operation is completed, the PRR will still be accessible by the rest of the system. All signals coming in or out from a PRR, must go through a bus macro, with the exception of clock signals. They are placed on the boundary of the PRR, either from the left or right size, and have a fixed direction of signal(right-to-left or left-to-right). They can be enabled or disabled (thus temporarily disconnecting

the PRR from the rest of the system) by a BusMacro enable signal.

- The size of the PRR depends on the size of the largest module to be configured in it. This limits available resources for the rest of the system, due to the fact that even when small modules are used in the PRR, the unused resources are still binded.

- The height of the PRR occupies the entire length of the columns in the Virtex-II PRO. So it is impossible to implement more than one PRR in the same column.

1.3.2 Difference Based Partial Reconfiguration

Difference-based partial reconfiguration is a method for making small changes in FPGA design such as changing LUT equations and Block Ram (BRAM) contents. Application of this method demands the modification of the device layout and routing through the use of low-level software (FPGA Editor). The result is a single partial bitstream that contains only the differences between layouts. This allows for fast reconfiguration of the device but introduces two limitations:

- Difficulty to change routing of the design.

- Limited range of applications due to the fact that it applies to simple designs, with minor differences.

1.3.3 Applications of Partial Reconfiguration

Several application domains are being studied in order to explore the benefits gained from their implementation with partial reconfiguration technology. PR is the cornerstone for power-efficient and cost-effective software-defined radios (SDRs) [21]. Through the Joint Tactical Radio System (JTRS) Program, SDRs are becoming a reality for the defense industries as an effective and necessary tool for communication. SDRs satisfy the JTRS standard by having both a software-reprogrammable operating environment and the ability to support multiple channels and networks simultaneously. Current implementations of open Software Communications Architecture (SCA) enabled SDR modems with multiple channels require multiple sets of processing resources and a dedicated set of hardware for each channel. The more channels SDR must support, the more dedicated resources are needed. This has a great impact on space, weight, power consumption, and cost savings. With partial reconfiguration technology, the ability to implement an SDR modem using shared resources, in order to support multiple waveforms, is realized.

Another usage is in the mitigation and recovery from single-event upsets (SEU). In-orbit, space-based, and extra-terrestrial applications have a high probability of experiencing SEUs. By performing partial reconfiguration, in conjunction with readback[3], a system can detect and repair SEUs in the configuration memory without disrupting its operations or completely reconfiguring the FPGA.

[3]Readback is the process of reading the internal configuration memory data to verify that current configuration data is correct

Implementation of cryptographic systems, can also benefit from PR. It gives the ability to support a range of cryptographic algorithms, without the need for having all of them integrated in the design. The system can swap in and out algorithms on demand, to respond to many different variables. This allows for smaller and inexpensive FPGAs to be used, as well as simpler designs, which results in reduced cost.

1.4 Combining Embedded Linux and Partial Reconfiguration

By combining the capabilities of Embedded Linux and Partial Reconfiguration, the latter is elevated from being low-level and complex to being easily expressed and automated. As a result:

- Reconfiguration operation is controlled with simple commands i.e. "open", "write" and "close". This relieves the researcher developing an application with PR technology, from dealing with the details of the complex PR technology.
- The ability to dynamically load device drivers on demand, combined with the partial reconfiguration capability of the Virtex-II Pro FPGA, allows for features like "hot-swapping" devices at run time.
- Access to the reconfiguration bitstreams is simplified, regardless of whether they are located in a local memory-based file, or, in a remote network filesystem.
- Complete operations may be performed by chaining multiple Linux tools and

utilities to implement network functionalities such as bitstream compression and encryption, managing multiple FPGAs, downloading bitstreams from a server and so on.

Chapter 2

Related Work

There exist several projects that employed an Operating System to leverage applications implemented on a partially reconfigurable system. In this chapter we describe these projects we continue with the previous work that has been conducted in the Microprocessor and Hardware Laboratory of the Technical University of Crete.

2.1 Related work on OS

John Williams [37] provided an abstraction layer for the Xilinx Internal Configuration Access Port (ICAP), and showed some ways in which it can be used to implement dynamic self-reconfiguring systems. He integrated the device by using the standard device driver architecture and developed a simple character-based device driver, which implements the read(), write() and ioctl() system calls. Also he demonstrated the power and ease-of-use of the linux shell.

Cryptography Algorithm	Slices	Throughput (Mbps)				Bitstream Size (Bytes)	Reconfiguration Time (msec)
		Xilkernel		uClinux			
		SW	HW	SW	HW		
Des	499	0.24	3.52	0.24	1.12	62896	318
Triple-Des	815	0.08	2.09	0.08	1.09	65116	323
RC4	614	0.95	3.22	1.18	2.10	70832	347

Table 2.1: Evaluation of a Dynamically Reconfigurable Cryptography platform

Lagger et. al [28] developed a dynamically reconfigurable platform for cryptography. The supported algorithms were DES, Triple DES and RC4. He deployed the Xilinx Xilkernel environment and the uCLinux port for the Microblaze, on a Xilinx Virtex-II 1000 FPGA. Execution times of the algorithms (implemented as both a software program running on the processor and a hardware co-processor) were measured and compared as a result of this project. These are presented in table 2.1.

Brebner [29] was one of the first to propose an operating system approach for partially reconfigurable hardware. He introduced the term of Swappable Logic Units (SLU), which can be described as position-independent tasks that are swapped in and out by the operating system.

Kosciuszkiewicz et. al [33] worked on the run-time management of reconfigurable hardware tasks under the supervision of a Linux OS. His implementation consists of a master Microblaze processor, running the operating system, with 8 Picoblaze co-processors as the reconfigurable resources. The proposed system offers transparent integration of reconfigurable resources within the software design and execution flow.

New applications to be executed can either bind a co-processor or create a software process, allowing for legacy applications to benefit from hardware acceleration.

Steiger et. al [31] discuss design issues for reconfigurable hardware operating systems and the problem of online scheduling hard real-time tasks to partially reconfigurable devices. He developed two online scheduling heuristics for the two area models under consideration[1].

Santambrogio et. al [34] proposed a methodology for the design of dynamically reconfigurable systems in which the reconfiguration management is completely assigned to an Operating System reconfiguration support. They present two extentions to a Linux kernel regarding reconfiguration support and reconfiguration management and present experimental results from a prototype implementation.

Berkeley University developed an extended Linux kernel that treats FPGA resources as native computational resources on reconfigurable computers such as BEE2, called BORPH [1]. It provides integral operating system supports for FPGA designs, such as the ability for an FPGA design to read/write to the standard Linux file system. A user process in BORPH, can either be a software program running on a processor, or a hardware design running on a FPGA, also called "hardware process". BORPH uses regions of FPGA fabric, called hardware regions, as computation units to spawn hardware processes.

[1] 1D and 2D. In 1D, tasks can be allocated anywhere along the horizontal device dimension, while in 2D allows for allocations anywhere on the hardware task area

2.2 Related work in our lab

Anifantis [27] conducted an experimental analysis on the reconfiguration delays in a Virtex-II Pro FPGA. He followed the Difference-Based design flow and provided the break-down delays to reconfigure a single frame. His results verified the theoritical values provided by the manufacturer.

Effraimidis [30] implemented an autonomous, partially reconfigurable, genetic algorithm system, supporting the change of the objective function at run-time. This allows for the theoritical support of a potentially unlimited number of objective functions with the ability to change them during run-time.

G. Nikoloudakis [32] implemented a partially reconfigurable cryptography system, supporting multiple algorithms, on a Virtex-II Pro FPGA. He provided several measurements of the reconfiguration time. The cryptographic cores used in that project, are employed in this thesis as well.

Chapter 3

Dynamic Reconfiguration of FPGAs

In this chapter, we describe the way partial reconfiguration is supported and executed on the Virtex-II Pro FPGAalong with some useful applications of this technology.

3.1 Xilinx Virtex-II Pro FPGA

The FPGA platform used in this thesis was the XUPV2P board by Digilent [8] with a Xilinx Virtex-II Pro FPGA. The FPGA is organized as a column based array of logic elements. The basic element is the configurable logic block (CLB) which contains look-up tables (LUT) as the basic function generators, flip-flops, multiplexers and gates. The periphery of the FPGA is occupied by input/output blocks (IOB) which are responsible for managing the FPGA pins. Except for CLBs and IOBs, the FPGA contains several specialized circuits such as Block SelectRAM (BRAM) resources, multiplier blocks and Digital Clock Manager (DCM) modules. The interconnections

between these circuits is accomplished by programmable routing resources organized in a hierarchical Global Routing Matrix (GRM), which is controlled by programmable routing switches. Since the FPGA is a two dimensional array, addressing the logic elements can be accomplished by using the Cartesian coordinates with two axes, the horizontal X and the vertical Y. The beginning of the axes is located at the lower left corner of the FPGA (X=0, Y=0). A simple diagram of the internal architecture of the FPGA is shown in figure 3.1

Configuration of the FPGA is accomplished by programming the logic elements and their interconnections. This information is stored in the configuration memory. It is organized in vertical configuration frames of one bit width whereas the length depends on the device. The frame is the smallest configuration unit and affects elements throughout the entire length of the FPGA. It has 1756 frames and each frame consists of 6592 bits.

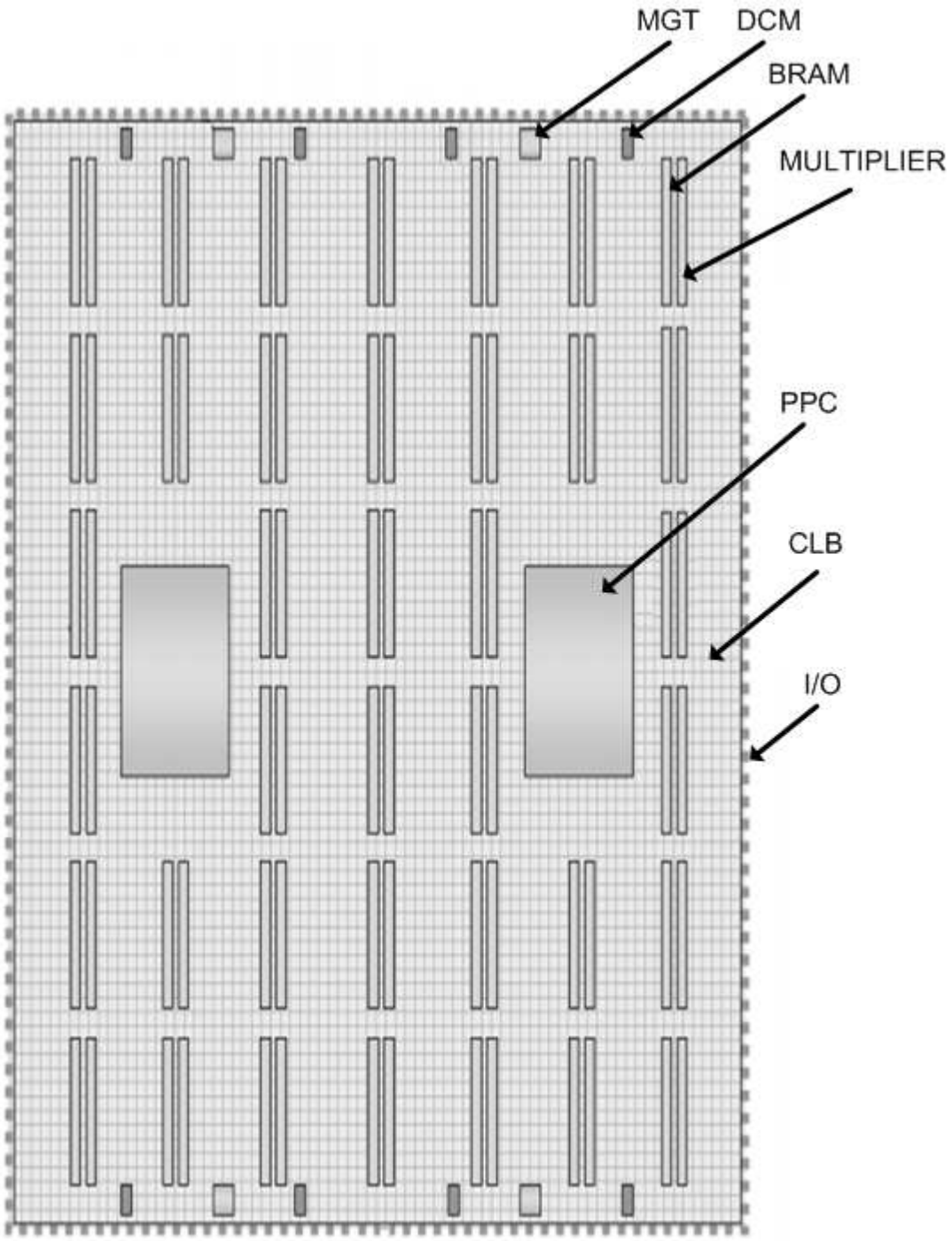

Figure 3.1: Internal Architecture of the Virtex-II Pro FPGA

3.1.1 Internal Configuration Access Port

The internal configuration access port (ICAP) is used to perform partial reconfiguration of the FPGA. It allows for reading and writing the configuration data of the FPGA and can only perform partial reconfiguration. The input and output ports of the ICAP are one byte wide, while the maximum frequency of operation is 100MHz. In previous publications of the vendor a lowest frequency (66MHz) was reported in the best case (Busy signal deactivated). The interface of the ICAP is displayed in Figure 3.2

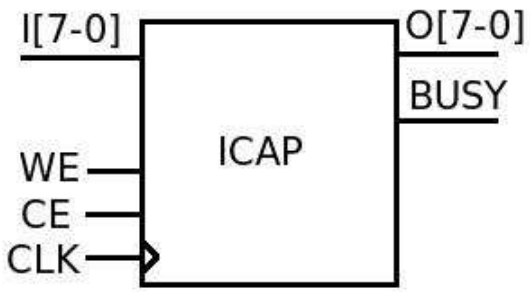

Figure 3.2: ICAP interface

In order to allow access to the ICAP by an embedded microprocessor (either Microblaze or PowerPC), the vendor's HWICAP IP core is used, allowing a user to modify the FPGA circuit structure by writing software programs. The HWICAP core is provided with EDK. This core is an OPB slave peripheral which instantiates the ICAP discussed previously. It includes a 18KBits BRAM out of which, only 2KB can be used as a temporary configuration buffer, due to addressing limitations. Modification of the configuration data is performed at frame level, because a frame is the smallest addressable segment in which the FPGA allows configuration data to be read and written. Reconfiguration is performed using a read-modify-write mechanism. To modify an FPGA circuit, the peripheral determines the configuration frames that must be modified, and then reads each frame one at a time into one Block RAM attached to the ICAP. The contents of each frame are modified and then written back to the ICAP. This is repeated for each frame. Although a single CLB, LUT or flip-flop can be modified, the underlying mechanism requires that the full column be read into Block RAM. This implies that other logic in the same column can be modified. In most cases, this effect can be ignored. When the frame is written back to the configuration memory the sections of the column that were not modified are written with the same data. Because the FPGA memory cells have glitchless transitions, when rewritten, the unmodified logic will continue to operate uninterrupted.

3.2 Process of Dynamic Reconfiguration

3.2.1 Implementation Strategies

There are different ways to implement dynamic reconfiguration of an FPGA, based on where the bitstreams are stored and who initiates the process. The possible ways are summarised below:

- **Exo-Reconfigurable:** In this case, reconfiguration is initiated by an external host. The bitstreams are located on the host and are transferred to the FPGA either through the JTAG or the SelectMAP external interface. This method of reconfiguration is depicted in Figure 3.3

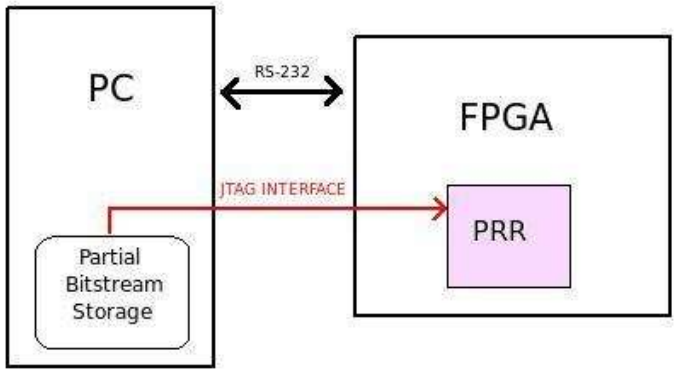

Figure 3.3: Exo-Reconfigurable

- **Endo-Reconfigurable:** In this case, the decision to initiate the reconfiguration and the reconfiguration process itself are controlled internally by the system. The partial bitstreams are stored in a Compact Flash or other external memory, and a processor reads them and performs the reconfiguration process. This method of reconfiguration is depicted in Figure 3.4

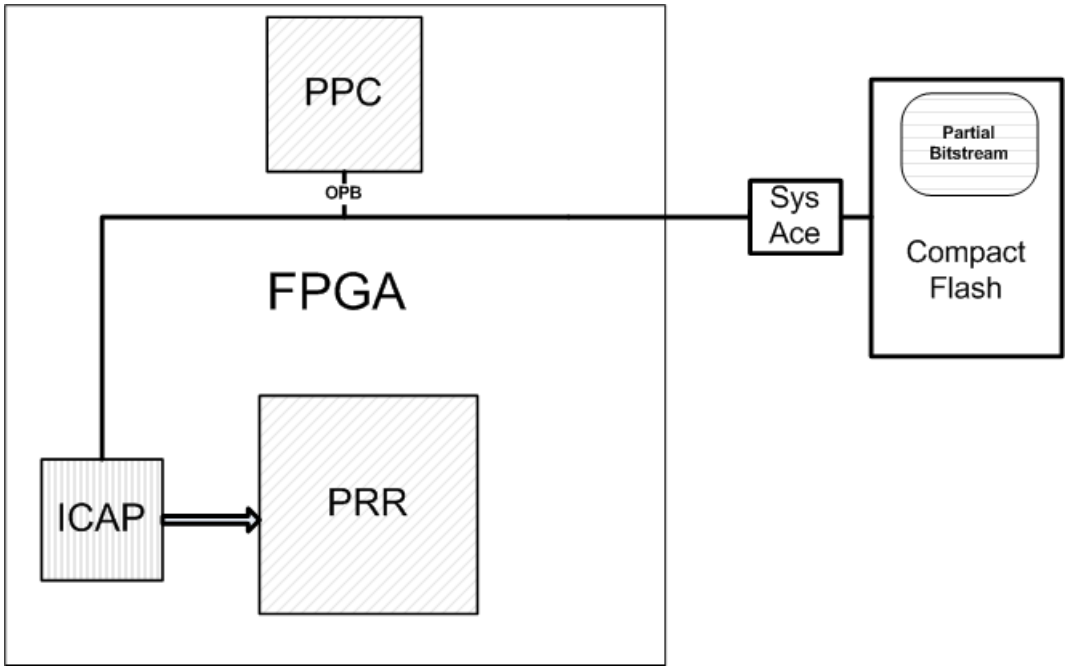

Figure 3.4: Endo-Reconfigurable

- **Hybrid:** This case is some combination of the previous two ways. The system can contain the partial bitstreams, but it has the capability to look for missing or updated bitstreams on a remote host.

For the purpose of this thesis, we used the Endo-Reconfigurable approach[1].It can be easily expanded to support a hybrid version, by integrating an ethernet interface allowing to download bitstreams from the Internet.

3.2.2 Required Steps

The process of dynamic reconfiguration, requires the transfer of the partial bitstreams from their external storage area, e.g. the Compact Flash, to the ICAP, so that the configuration memory can be modified. This process includes the following steps:

- Transferring the partial bitstreams from the Compact Flash to a temporary buffer in the DDR RAM.
- Transferring the contents of the buffer to the Block Ram of the ICAP core.
- Reading, modifying and writing the frames of the configuration memory of the FPGA.

[1]More details are contained in the following section

Chapter 4

Initial Attempts and Problems With Porting the OS on the FPGA

The selection and configuration of an appropriate linux kernel for our case, required many unsuccessful attempts with different distributions[1]. The basic concept was to get a Linux kernel and filesystem, set up on a Compact Flash card, that would enable us to boot the operating system by, simply, powering up the board. Our initial approach was the uClinux [25] distribution. uClinux is a special port of the Linux kernel targeting systems without a memory management unit (MMU). It supports a variety of target processors, including the Microblaze CPU, supported by Xilinx. The uClinux port for the Xilinx Microblaze CPU was done by Dr John Williams [15]. By doing some research, we came across PetaLogix [20], an embedded Linux solutions provider founded by Dr Williams, who provides an updated and more featured version of the uClinux distribution, called PetaLinux for the Microblaze. This distribution

[1]Technical details on the problems are located in Appendix A

provides an automated BSP generator to automatically target Linux systems to a particular custom hardware platform, self-contained GCC cross-compiler and cross-debugger toolchains, and application and kernel module generator. Thus, we decided to proceed using this distribution[2].

The process of successfully building a working kernel for our board, involved overcoming several problems. Since, the XUP Virtex-II Pro board is not included in the reference designs, we followed the Tutorial Guide on how to create a design from scratch. The default system configuration proposed by the guide required the presence of an Ethernet MAC Controller and a Flash 2Mx32 peripheral. We were unable to include either of these peripherals. The default installation of the EDK 9.1 tools does not include a license for the Ethernet MAC peripheral and the Base System Builder Wizard does not provide an option for including the Flash 2Mx32 peripheral, as described in the guide. Nevertheless, we decided to proceed.

However several issues emerged The default setup of PetaLinux, consists of a two-stage bootloader. The first stage, called FS-Boot, gets integrated, as an EDK Software Project, within the download bitstream and requires the presence of the Flash Memory component mentioned previously. Errors occured during the compilation of FS-Boot, within the EDK tool as well as during the compilation of the kernel, regarding the missing Flash peripheral. Thus, we decided not to build the FS-Boot, disable the Memory Technology Devices (that were enabled in accordance to the guide) and continue with the compilation of the kernel. Further issues emerged. We had to manually edit the file sumversion.c to include the missing library <linux/limits.h>, in order to eliminate a compilation error regarding the undeclared variable "PATH_MAX" This

[2]Version 0.30 was the latest at that time

resolved the error and the compilation procedure continued. However, new problems occured concerning the second-stage bootloader called U-Boot. By default, U-Boot searches for a kernel on a network mount, through an ethernet interface. However, this didn't suit our case due to the absence of the Ethernet component and was not in accordance to our initial idea of set-up [3]. This caused many errors during compilation. Modifying the U-Boot bootloader to load the kernel from a Compact Flash was more difficult and time-consuming than expected and therefore, was abandoned as an idea. So, we continued by disabling U-Boot, to see if we could get a working kernel, that we could download on the board by using the download option of the xmd software included in EDK. We managed to successfully build and download a working Petalinux kernel on the board using the Xilinx USB Platform cable. Table 4.1 presents the tools used for this first unsuccessful attempt.

Due to the problems mentioned above and the limitations imposed by them, an alternative had to be found. The lower performance of Microblaze as compared with the hard core PowerPC, induced us to use the PowerPC core of the Virtex-II Pro FPGA. An attempt was made to use a modified version of the uCLinux distribution with support for the PowerPC [10]. However, this attempt caused numerous problems and was abandoned. Then, we came across a guide from the department of Computer Science and Engineering of the University of Washington [17], demonstrating the way to port a Linux kernel, version 2.4.26, for the Power PC on the XUP board and build a filesystem with many tools ready to be used from the operating system. This distribution includes only the kernel sources and not the necessary tools to compile them. Since the host and target architectures are different, the use of a cross compiler

[3] A presence of a host system would be essential in this case in order to boot the kernel

was necessary. Crosstool was selected, according to the tutorial. The procedure was completed rather smoothly and provided an autonomous system with the ability to configure and boot without the need for a host system. However, two issues emerged. There was no driver support for the HWICAP IP core and a simple driver had to be written to allow communication with our OPB_DCR_SOCKET IP core, which is used for extending the OPB bus (allowing us to connect our reconfigurable modules to our system). An attempt was made to integrate the driver sources for the HWICAP, written by Dr. Williams for the Microblaze port of uCLinux. As a result, the HWICAP was recognized during the boot of the Linux kernel but correct functionally was not tested because a better alternative was found. Table 4.2 presents the tools used for this second attempt.

During research, we came across the open source Linux project from Xilinx. This Linux project features:

- A very modern Linux kernel (ver. 2.6.29 at the time)
- Integrated device drivers for a variety of Xilinx IP cores (including the HWICAP)
- Easy configuration of the kernel by using a Device Tree[4]

Thus, we decided to proceed with the open source Linux project from Xilinx. Information about the tools used for this attempt can be found in table 4.3.

[4]It includes vital information about the system assembly including bus addresses, inter-connection details etc. making it easier to set up the system and develop drivers.

Component	Name	Version
Operating System	Petalinux [20]	0.30rc1
Host Operating System	Ubuntu [24]	8.10
Cross Compiler	Built-in	3.4.1
Filesystem	RAM Filesystem	N/A
Xilinx Tools	ISE/EDK	9.1

Table 4.1: Initial Attempt

Component	Name	Version
Operating System	Linux 2.4 for PowerPC [12]	2.4.26
Host Operating System	Ubuntu [24]	8.10
Cross Compiler	Cross-Tool [3]	0.43 (gcc version 3.4.4)
Filesystem	Busybox [4]	1.14.3
Xilinx Tools	ISE/EDK	9.1

Table 4.2: Second Attempt

Component	Name	Version
Operating System	Xilinx OSL [18]	2.6.29
Host Operating System	Ubuntu [24]	8.10
Cross Compiler	ELDK [6]	gcc 4.2.2
Filesystem	Busybox [4]	1.14.3
Xilinx Tools	ISE/EDK	9.1

Table 4.3: Successful Attempt

Chapter 5

Running the OS on a Virtex-II Pro FPGA

In this chapter, we describe the system setup, and the procedure for compiling and properly setting up the Linux kernel, on a XUP Virtex-II Pro FPGA.

5.1 Setting up the host system

The need for compiling a Linux kernel necessitated that all the necessary tools for building the system, had to be setup in a Linux-based host system. These tools consist of the Xilinx ISE SP2 and EDK 9.1 SP2 software with PR support, Xilinx PlanAhead 10.1.8 and a cross-compiler, used for compiling the software for the PowerPC architecture. The Linux distribution chosen for the host system was Ubuntu 8.10 32-bit Desktop edition with kernel version 2.6.27, due to its ease of use and excellent community support. It was installed on an Intel Core 2 duo system, running at 2.5 GHz with 4 GBs of RAM.

5.1.1 Setting up the Xilinx Software

The installation of the Xilinx tools, consists of firstly installing the software and then installing the platform cable driver for downloading designs on the board.

The installation of the software was completed rather smoothly, by running the Linux installer scripts located in the setup folder. The selected destination folders in which the tools were installed are /opt/Xilinx/ISE and /opt/Xilinx/EDK. In order for the user to be able to update the software, the folder permissions had to be changed for full access with the following command:

```
chmod -R 0777 /folder.
```

The installation of the PR support was completed according to the instructions found on the Partial Reconfiguration Early Access software tools [19] web site. The installation of service pack 2 for the ISE tool was necessary because of the lack of support of the PR design flow in the newer versions, of the tools.

The installation of the cable driver proved to be quite problematic. The driver provided by Xilinx was designed to work in the enterprise editions of the RedHat and SUSE linux distributions. Therefore an alternative had to be found. After many unsuccessful attempts, we finally came across a library [35], based on the libusb project, which allows the tools to access the JTAG cable without the need for a proprietary kernel module. The library was compiled with the *make* command and was utilized using the *export* command to set the *LD_PRELOAD*[1] environment variable as follows:

```
export LD_PRELOAD=/path/to/libusb-driver.so.
```

[1]The environment variable used by the Xilinx tools to locate the JTAG driver

The installation of PlanAhead was successful, however it did not function properly. So the installation of PlanAhead on a windows platform was necessary.

5.1.2 Setting up the cross compiler

Since the architectures of the host machine and target board are different, the use of a cross-compiler for the target architecture is required. Researching the web, we came across the Embedded Linux Development Kit [6] (ELDK) maintained by DENX Software Engineering [7]. The ELDK includes the GNU cross development tools (gcc ver. 4.2.2, glibc ver. 2.8.90), such as the compilers, binutils, gdb, etc., and a number of pre-built target tools and libraries necessary to provide some functionality on the target system. It is provided for free use with full source code, including all patches, extensions, programs and scripts used to build the tools. We proceeded with downloading the appropriate version for our target CPU (PowerPC AMCC 4xx without FPU support) and installing it on our host system by using the following command

```
./install -d /opt/ELDK/ ppc_4xx
```

After we successfully installed ELDK, two more steps were necessary to complete the setup.

- The first is to set up the path to the cross compiler executables by using the command:

```
export PATH=/opt/ELDK/usr/bin:/opt/ELDK/bin:$PATH
```

- The second one is to set up the environment variable CROSS_COMPILE by using the command:

```
export CROSS_COMPILE=ppc_4xx-
```

After performing these steps[2], the host system is ready to compile the required linux kernel sources and test programs.

5.2 Configuring and Compiling the Linux Kernel

5.2.1 Process of building the Operating System

Building the entire Operating System requires the partitioning and formatting of the Compact flash, the compilation of a Linux kernel and the creation of the filesystem.

5.2.1.1 Configuring the Compact Flash

The Compact Flash must be partitioned into three parts. One FAT partition, which holds the system.ace for configure the FPGA, one swap partition required by the Linux system and, finally, an ext2 partition which hosts the filesystem of the Operating System, as well as the partial bitstreams. A Compact Flash of 1GB in size was used.

For partitioning the CF, the tool fdisk was used, which is a menu driven program for the creation and manipulation of partition tables. After successfully creating the partition table, the partitions were formatted using the following commands[3]:

- `mkdosfs -s 64 -F 16 -R 1 /dev/sdc1` for the FAT partition

[2]Setting these variables is necessary each time a new linux console is launched

[3]The device files used in the commands may vary depending on the setup of the host machine

- `mkswap /dev/sdc2` for the swap partition
- `mke2fs /dev/sdc3` for the filesystem partition

5.2.1.2 Building the Linux kernel

The process of configuring and compiling the Linux kernel requires several steps. Here is a simple outline for a base system setup, supporting a minimum set of the requirements without reconfiguration support:

1. Download the Linux kernel and the device tree generator from the Xilinx git tree repository [26]
2. Create the system setup with Xilinx EDK.
3. Setup the Xilinx EDK to use the device-tree generator.
4. Generate a device tree for the specific system setup.
5. Copy the .dts file to the /arch/powerpc/boot/dts folder of Linux kernel source tree.
6. Configure the kernel.
7. Build the kernel.
8. Load and run the kernel on the board.

In more detail:

- **System setup:** The base system setup consists of the following configuration
  - PowerPC running at 100MHz

- JTAG debug
- No Data and Instruction On Chip Memory
- Cache enabled for instructions and data with burst mode enabled
- OPB UARTLITE with 38400 baudrate and interrupt enabled
- OPB SYSACE Compact Flash controller with interrupt enabled
- 128MB PLB DDR RAM to be used as main memory for Linux
- PLB BRAM IF CNTRL 16 KB for the Linux bootloader to be stored in

The system is then synthesized and the bitstream is created.

- **Setup the Device Tree Generator in Xilinx EDK:** Setting up the device-tree consists of simply copying the bsp in parent directory of the edk project,selecting it in the Software Platform Settings dialog box, and setting the following values in OS and Libraries sub-menu:
  - console device : RS232_uart
  - bootargs : console=ttyUL0 root=/dev/xsa3 rw[4] . These can be easily changed to fit different needs

- **Configuring the kernel:** The easiest way to get an initial configuration for our board was to use a provided default configuration for a different board(virtex4) which, however uses the same PowerPC. This can be easily done with the following command:

[4]The first argument indicates that we are using the uartlite core, the second tells the kernel to search for a file system on the third partition of the Compact Flash and the third mounts the filesystem with read-write capability

```
make ARCH=powerpc 40x/virtex4_defconfig
```

Some modifications had to be made to fully support our base system, consisting of enabling some Device Drivers. This can be easily done through a configuration menu, accessed with the following command:

```
make ARCH=powerpc menuconfig
```

The following device drivers were enabled, by navigating to the appropriate menus:

- Xilinx SystemACE support (located in Device Drivers→ Block Devices)
- Xilinx uartlite serial port support (located in Device Drivers→ Character Devices→ Serial drivers)

- **Building the kernel:** Building the kernel is accomplished by simply running the following command:

```
make ARCH=powerpc simpleImage.virtex405-xup
```

where virtex405-xup is the name we gave to the .dts file generated previously. The device-tree is built into the kernel and is read during boot by the enabled device drivers, for configuration. The output file is named simpleImage.virtex405-xup.elf and is located in the /arch/powerpc/boot directory. We copy this file in the EDK project directory.

- **Loading and Running the kernel:** Loading the kernel to the board requires the creation of .ace file that is loaded during power up of the board from the

Compact Flash. It programs the FPGA and loads the Linux bootloader on the PowerPC that ultimately boots the system. This file is created by running the following command:

```
xmd -tcl genace.tcl -opt genace.opt
```

The file genace.opt includes several arguments passed to the xmd tool for successfully creating the .ace file for our board. These are:

- jprog
- target ppc_hw
- hw implementation/download.bit
- elf simpleImage.virtex405-xup.elf
- board user
- configdevice devicenr 1 idcode 0x127e093 irlength 14 partname xc2vp30
- debugdevice devicenr 1 cpunr 1
- ace system.ace

The resulting file is then copied on a FAT partition on the CF and finally loaded during power-up.

5.2.1.3 Creating the filesystem

For creating the root filesystem, Busybox [4] 1.14.3 was used. BusyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities usually found in GNU fileutils, shellutils, etc. These utilities generally have fewer options than their full-featured GNU

cousins; however, the options that are included provide the expected functionality and behave similar with their GNU counterparts. Busybox provides a fairly complete environment for any small or embedded system, fully customizable by providing the capability to include or exclude tools during compilation. The process of building the filesystem consists of the following steps, according to the guide from the University of Washington:

1. **Configure Busybox:** This is done by issuing the following command:

   ```
   make menuconfig
   ```

   The default configuration was used, except for 2 changes:

   - Setting the cross compiler prefix (Busybox Settings→Build Options→Cross Compiler Prefix)
   - Setting the installation path (Busybox Settings→Installation Options→Busybox installation prefix)

2. **Modify and run installation script:** An installation script "mkrootfs.sh" from Klingauf[5] [13] was used for easier creation of the filesystem. This script deletes any existing files in the destination folder, creates all the necessary folders, copies the /etc folder provided by Klingauf and builds Busybox. Certain paths had to be altered as follows:

   - LFS=/installation/path/xup_rootfs
   - CC=ppc_4xx- (Cross Compiler prefix)

[5]The files downloaded from Klingauf have DOS newlines. They had to be replaced with Unix newlines for proper functioning

- TARGET_PREFIX=/opt/ELDK/ppc_4xx
- BUILD_TOOLS=/opt/ELDK/
- PPC_KERNEL_VERSION=2.6.29

The script is executed with the following command

```
./mkrootfs.sh as root.
```

3. **Last modifications** We also had to overwrite the file /etc/inittab with the following commands, in order to correctly configure the serial console output and define the commands to be executed during the initialization, shutdown and restart of the system:

```
::sysinit:/etc/init.d/rcS
#::askfirst:-/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
::restart:/sbin/init
#::respawn:/sbin/getty 38400 tts/0
::respawn:-/bin/sh
```

It was necessary to create a special device file for console in the /dev directory. This can be easily done with by using the mknod command:

```
mknod /dev/console c 5 1
```

Afterwards, the filesystem is ready to be copied on the third partition of the Compact Flash which hosts the filesystem of the OS.

Chapter 6

Development of the OS-based Infrastructure

Due to missing elements (like the HWICAP,OPB/DCR Socket IP cores and inability to access the custom peripheral OPB/DCR socket) that restricted us from having a fully functional system, further steps were required in order to fully support the DPR operation, and access the registers of the OPB/DCR socket IP for writing and reading data, to and from the PRMs. In this chapter, we describe the process of implementing the system to support all the necessary functionality.

In order to fully develop the functional infrastructure, we proceeded with implementing a small reconfigurable system, with an adder and multiplier as the partially reconfigurable modules, placed in the same PRR. They perform addition and multiplication[1] and, both, have two 32-bit wide inputs and produce a 32 bit wide result. The procedure of creating a Linux-based, dynamically reconfigurable system, based

[1]The multiplier performed the operation on the 16 lower bits to ensure that the outcome would be 32 bits wide

on the design flow proposed by Xilinx in the EAPR Lounge [19], can be outlined by the following steps:

- Create a processor system with EDK
- Create the custom logic IPs, to be used as the PRMs, and synthesize them.
- Connect the processor system and the IP under a top-level design. The dynamically reconfigurable logic have been instantiated as "black boxes". The DCMs are removed from the EDK project and instantiated in the top-level design, so that the design tools can recognize the clock signals.
- Use Planahead to place the PRRs and bus macros, run DRC rules check and run the necessary commands to execute the PR flow. This results in the full and partial bitstreams.
- Use the device tree generated file from the first step (as described in chapter 5) to compile the kernel.
- Integrate the Linux kernel with the full bitstream.

Below follows a description of certain key points in the process.

6.1 Implementation of dynamically reconfigurable system

The base system, of our design (as described in chapter 4) is depicted in Figure 6.1:

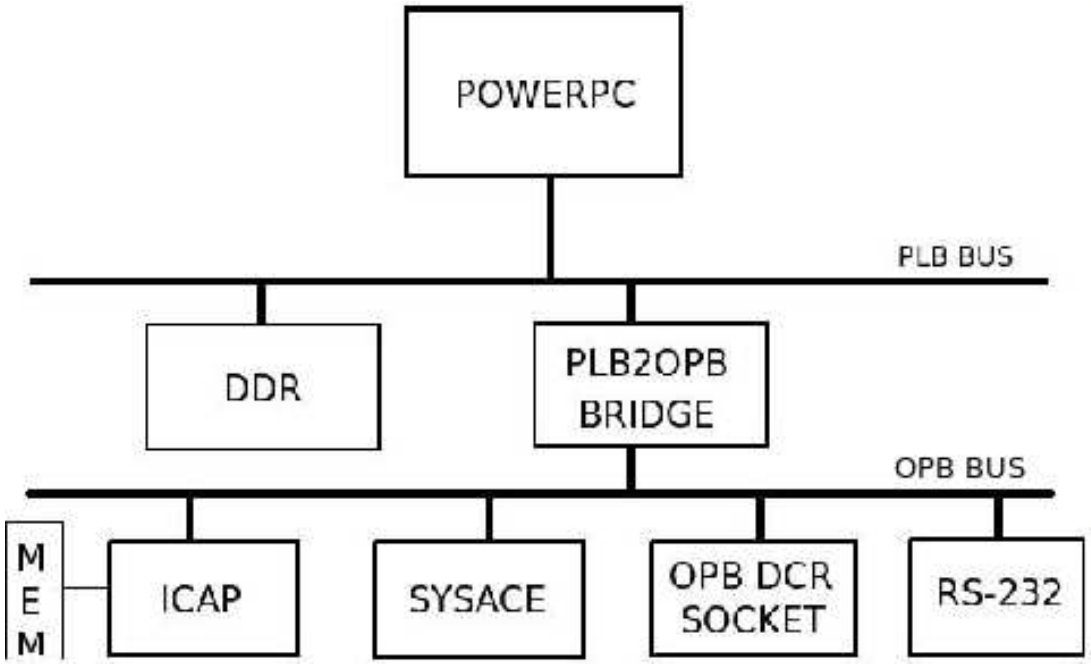

Figure 6.1: Base System Setup

It consists of:

- A PowerPC, running at 100 MHz
- A DDR controller, supporting a DDR RAM of 128MB in size. It serves as the system memory. The Linux kernel gets expanded here allowing it to boot. It is also used by our test program as storage for a temporary buffer, where the partial bitstreams are loaded from the Compact Flash
- RS-232 for communicating with the host PC. It serves as input and output for the Linux console, allowing the user to issue commands to the operating system. The baud rate was set at 38400Kbps.
- An OPB-DCR Socket. This peripheral extends the OPB Bus interface by making it external and allows access to the bus macro enable pin, by using the DCR. This allows for the easy connection of the peripheral where our partial reconfiguration modules are swapped in.
- A SYSACE Controller. This peripheral allows the use of Compact Flash as storage space for the initial system bitstream as well as the partial bitstreams. It is also used as storage for the root filesystem of our Linux operating system.
- The HWICAP module. As described in Chapter 3, the HWICAP module makes it possible for the processor to access the ICAP. The Linux Device Driver for the HWICAP core, which is integrated in the kernel sources, had to be enabled from the Configure menu of the Linux kernel (located in Device Drivers→Character Devices).

6.1.1 Assigning the PRR and bus macros

Placing a PRR in the FPGA has to follow some restrictions:

1. Outside the PRR, a space of two slice column must be left, to allow the placement of bus macros.

2. The sides of the PRR must not be placed on the FPGA border to allow signal accessing I/O buffers to be routed successfully.

3. The PRR must not cross slice-columns in the area of the ICAP (located in the bottom right corner of the FPGA) as this causes random freezes of the system.

The peripherals used in this initial attempt are very simple. The resource requirements are rather small and, thus, a small PRR was required. Bus macros were used to connect the OPB bus extension, provided by the OPB DCR socket, and the PRR. The bus macros were placed on the left side of the PRR and, therefore, bus macros with direction left-to-right were chosen for the input signals and right-to-left for the output signals. The PRR and bus macros can be seen in Figure 6.2

6.1.2 Placement of the DCMs

The system makes use of two DCMs to produce the necessary clock signals for the DDR and the system clock. These DCMs must be instantiated manually in order for the design tools to recognize the clock signals correctly and route them through the global clock nets. The DCMs were assigned to positions X2Y0 and X2Y1.

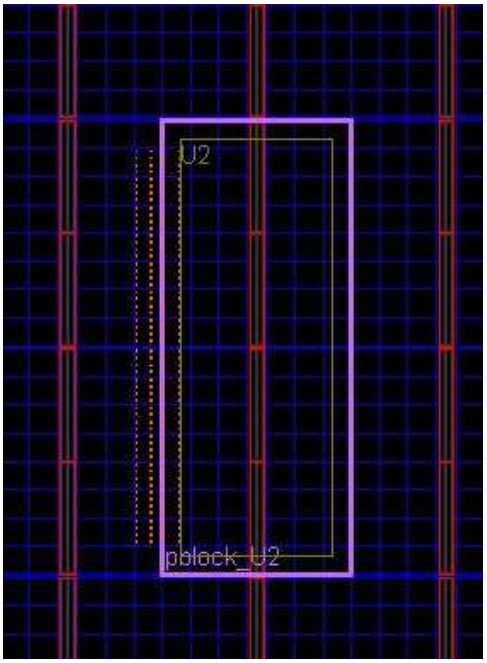

Figure 6.2: Placement of the PRR and bus macros for the adder/multiplier

6.2 Booting the System and checking functionality

After successfully compiling the Linux kernel and integrating with the system design, we proceeded with booting-up the board. The process was completed successfully except for a few issues.

The HWICAP was recognized by the kernel and the appropriate linux driver was loaded to support it, remapping it to a virtual address. However, a special device file binded to the HWICAP major device number[2] had to be created, to allow users to instantiate the ICAP in their programs. This can be easily done by using the following command:

```
mknod /dev/console c 258 1
```

[2]The major device number is used to differentiate the system devices. The minor number is used to differentiate different instantiations of the same device type

The major number 258 is pre-selected in the HWICAP driver. The HWICAP driver allows access to the ICAP with the use of four simple operations depicted in Table 6.1:

Method	Functionality
open	Open the port and initialize for access
release	Release the port
write	Write a bitstream to the configuration processor
read	Read a data stream from the configuration processor

Table 6.1: ICAP access operations

After being opened, the port is initialized and accessed to avoid a corrupted first read which may occur with some hardware. The port is left in a desynched state, requiring that a synch sequence will be transmitted before any valid configuration data. A user will have exclusive access to the device while it remains open, and the state of the ICAP cannot be guaranteed after the device is closed.

The second issue that emerged, involved the detection and remapping of the OPB DCR socket peripheral, which allowed us to access to the PRR. During boot there was no detection and it was not contained in the list with the loaded devices[3]. In order to make the device detectable by the operating system, a simple driver had to be written[4]. This driver was based on a Xilinx application note, describing the procedure of integrating a specific EDK peripheral into Linux [11]. During boot, it

[3]The probed devices can be easily viewed by using the cat /proc/devices command
[4]Source code for this driver is included in Appendix C

parses the device tree to find all the necessary attributes, like physical addresses. It provides the operations shown in Table 6.2.

Method	Functionality
open	Open the socket and initialize for access
release	Release the socket
write	Write a value to a register of the custom peripheral
read	Read a value from a register of the custom peripheral
ioctl	Defines the read or data offset of the read and write addresses for the peripheral registers. It also disables and enables the Bus Macros

Table 6.2: OPB_DCR Socket access methods

Currently, the driver supports 12 registers for read and write. It can be easily updated to support any number of registers. It was integrated in the kernel sources by copying the source code in the kernel drivers directory and modifying the Makefile to include the sources. We created a special character device in the filesystem with the command

```
mknod /dev/console c 177 1
```

Then, compilation and booting of the system was successful and the peripheral was correctly probed and remapped. A simple program was written verify the functionality, that writes two operands and reads the result.

We then proceeded to test the reconfiguration process. The procedure followed in the test program is as follows:

1. Open the socket and the ICAP devices
2. Open the partial bitstreams and store them in temporary buffers in the DDR. We manipulate the bitstream files as any file by using the functions “fopen”, “fclose”, “fread” etc.
3. Write two operands in the two inputs of the mathematical function.
4. Read the result.
5. Disable the bus macros, in order to disconnect the PRR from the rest of the system.
6. Write the partial bitstream of the other math function by using the command

   ```
   fwrite(bit_buffer, sizeof(char), fileLen, icap);
   ```

   where bit_buffer is a void* buffer containing the previously read bitstream, sizeof(char) defines the size of a single element to be written, fileLen is the size of the buffer in bytes and icap is the file pointer to the ICAP device.
7. Re-enable the bus macros.
8. Test the newly reconfigured core by checking correct functionality
9. Repeat from step 5

Reading and storing the partial bitstreams immediately with the beginning of the program, and then using the temporary buffers to reprogram the FPGA, acts as a form of prefetching. The bitstreams remain loaded in the DDR RAM, ready to be used, which is much faster compared to the Compact Flash.

Chapter 7

Running a real application on the OS-based System

After successfully setting up the infrastructure to support DPR, we proceeded with the implementation of the final test system. For the purpose of this thesis, we used the encryption cores, used in the thesis of G. Nikoloudakis [32].

7.1 Background on Cryptographic Algorithms

Today's digital age makes the protection of data against malicious users extremely important. Safely storing the data and transmitting it over insecure networks demands the encryption of data. As a test application for this thesis, we have used two of the most popular encryption algorithms; Triple-DES and AES. The algorithms were selected due to the following reasons:

- Widely used in data encryption

- Implementation of a unified encryption platform will enable us to study the application of dynamic reconfiguration in:
  - Implementation of algorithms with significant computational load and complexity
  - Dynamic swapping of algorithms with varying resource requirements
  - Implementation of algorithms whose co-existence in the system is impossible due to a lack in availabe resources.

These algorithms belong in the class of Symmetric-Key encryption algorithms. Basic functionality of this type of encryption is depicted in Figure 7.1:

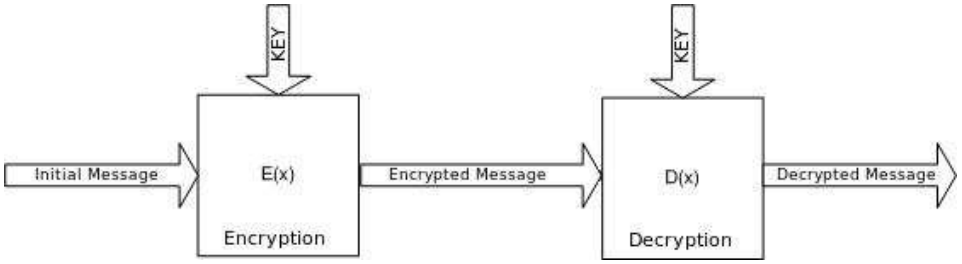

Figure 7.1: Symmetric Key Encryption Algorithms

These algorithms are also known as private-key algorithms because their security depends on the encryption/decryption key which is known only by the sender and the receiver of the message.

7.2 Integrating the Encryption Cores in the system

During the process of integrating the encryption cores in our system, an issue emerged. The maximum frequencies supported by all the different components of the system are shown in table 7.1.

Component	Maximum Frequency (MHz)
Static System	100
Triple-DES	170
AES	79.3

Table 7.1: Maximum operating frequencies

We proceeded with implementing the optimal configuration for these cores. We added an additional DCM to the two already present in the system, and configured them to output two additional clock signals (besides the system clock set at 100MHz, the clk90 and ddr90 clock signals were required for the DDR controller). Thus, there were 5 clock signals present in the system. Although the procedure of creating the system went rather smoothly, the final step of creating the static and partial bitstreams resulted in error concerning short-circuits in the system. Several attempts were made to resolve this issue, including different placement of the PRR, the Bus macros and the DCMs, as well as selecting a high placer and router effort level. Finally, we decided to use a common clock for the two cores, set at 50 MHz. This resulted in a successful generation of the static and partial bitstreams. Thus we

decided to proceed with this clock configuration.We followed the process already described in Chapter 6. A few details are shown below.

7.2.1 Resource Requirements

The slice resource requirements for the every system component (table 7.2). The FPGA provides 13696 slices and our system occupies 13669 or a 99.8% of the available resources. It is obvious that a static version of our system, supporting the two algorithms, would make it impossible to support any additional logic or encryption algorithms. The dynamically reconfigurable version, however, supports a great number of algorithms (theoretically, any algorithm with size equal to the AES). This allows for great system flexibility.

Component	Required Slices	% of FPGA slices
Static System	2152 out of 13696	15.71%
AES	9671 out of 13696	70.61%
Triple-DES	1846 out of 13696	13.48%
Total	13669	99.8

Table 7.2: Percentage of FPGA slice usage

7.2.2 Placement of the Encryption PRR

The size of the PRR is determined by the size of the largest design; in this case the AES algorithm. Table 7.3 illustrates the slice requirements of these algorithms[1].

Algorithm	Slices	% of PRR slices
AES	9671 of 9920	97.48%
Triple-DES	1846 of 9920	18.61%

Table 7.3: Percentage of PRR usage

This table shows two disadvantages of the Module-based design flow:

1. Acquision of resources that may go unused by any of the PRMs but are, also, unavailable to the rest of the system since they are a part of the PRR

2. Each reconfiguration traverses the entire PRR even though some parts do not contain any logic.

[1]The PRR could not be set at exactly the size of the AES algorithm. This is due to the different limitations concerning PRR placement

Resource allocation of the two algorithms within the PRR are shown in Tables 7.4 and 7.5.

Site Type	Available	Required	% Utilization
LUT	19840	15853	79.90
FF	19840	1678	8.46
SLICE	9920	9671	97.49
MULT18X18	92	0	0
RAMB16	92	0	0
TBUF	4960	0	0

Table 7.4: PRR usage for the AES

Site Type	Available	Required	% Utilization
LUT	19840	3026	15.25
FF	19840	1718	8.66
SLICE	9920	1846	18.61
MULT18X18	92	0	0
RAMB16	92	0	0
TBUF	4960	0	0

Table 7.5: PRR usage for the DES

Placement of the PRR, must conform to the restrictions described in Chapter 5. For purposes of direct comparison of our experimental results with the results of [32], we used the same exact PRR placement. The coordinates of the PRR are X10 Y154 to X80 Y5. We chose AES as the active reconfigurable module.

The placement of the PRR is depicted in Figure 7.2

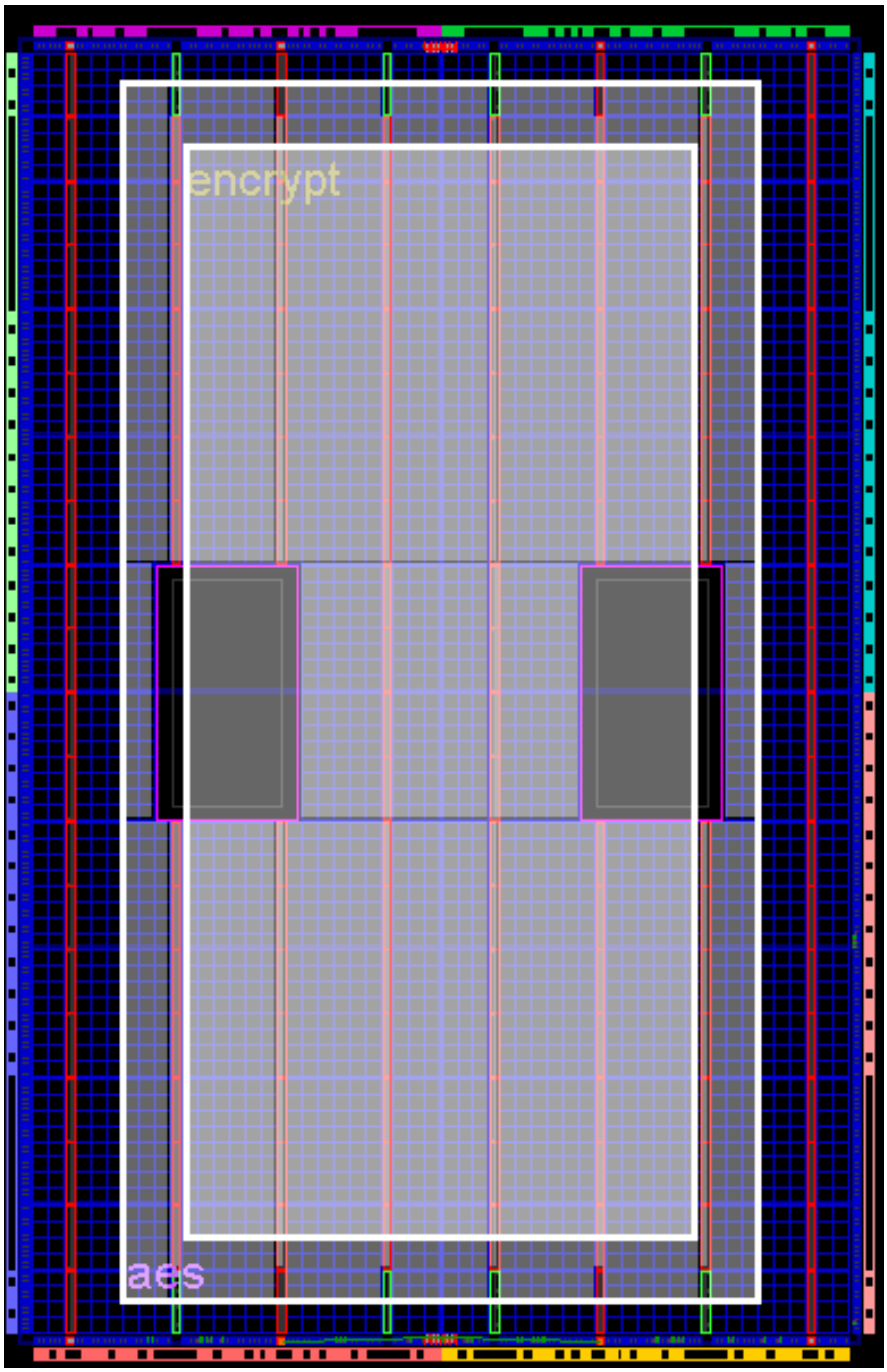

Figure 7.2: Placement of the Encryption PRR

7.2.3 Placement of the Bus Macros

The Bus Macros used are the same with those used in the initial attempt described in Chapter 5, since we are using OPB Bus compatible peripherals in both cases. We made a small modification in the encryption cores, adding a 4-bit output signal, used as input for the LEDs found on the XUP board. We used this signal as a mean of ensuring that the reconfiguration process is completed successfully. Since any signal (besides the clock signals) that comes in or out of a PRR has to pass through a bus macro, an extra Bus Macro was added. The number and types of Bus Macros used are shown in Table 7.6

Type	Size	Number of BMs	Type of BMs
Input Data Bus	32 bit	4	Left-to-Right
Output Data Bus	32 bit	4	Right-to-Left with Enable
Address Bus	32 bit	4	Left-to-Right
Input Control Signals	8 bit	2	Left-to-Right
LED Signals	4 bit	1	Right-to-Left with Enable
Total	108 bit	15	

Table 7.6: Bus Macros

The placement of the Bus Macros is shown in Figures 7.3 and 7.4. The Bus Macro for the LED signals, had to be placed in the lower left corner of the PRR. This was necessary as the LED output pins are located in the right side of the FPGA and thus, placing it there resulted in the least possible obstruction for the system routing.

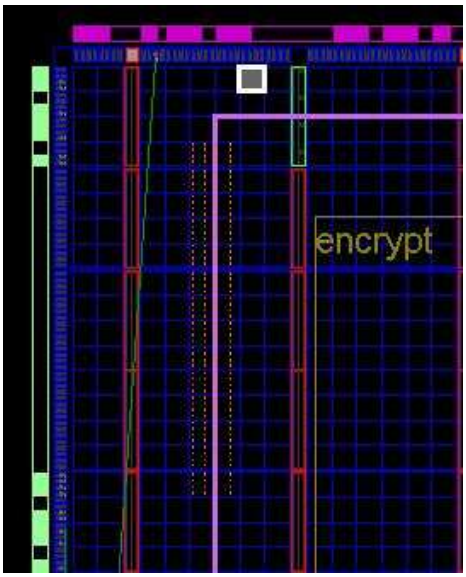

Figure 7.3: Placement of the Encryption Bus Macros

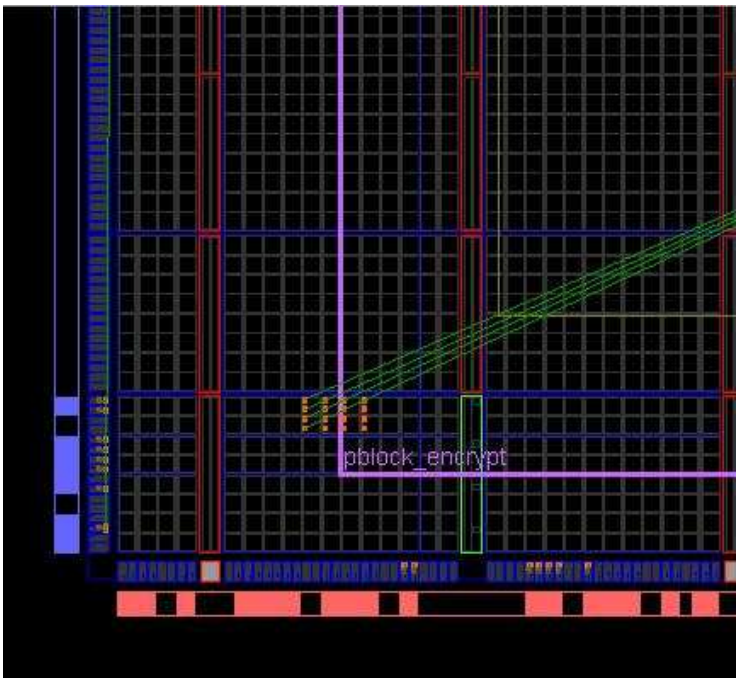

Figure 7.4: Placement of the led Bus Macro

7.3 Testing the system

After successfully integrating the encryption cores to the existing Linux-based infrastructure, we proceeded with system testing. System boot was completed successfully. Since correct functionality of the cores was thoroughly tested by G. Nikoloudakis [32], a simple program was written that performs an encryption with the default PRM loaded (AES), displays the result, swaps to the Triple-DES, performs an encryption, displays the result and then swaps back to the AES module. This process is repeated in an infinite loop.

During initial attempts, the encrypted data produced by these two algorithms were incorrect. Trying to resolve this problem, we realized that it was necessary to write the reconfiguration bitstreams two times in a row, before accessing the peripheral. This was the only way to perform a successful reconfiguration. The causes for this were not found. A guess about it is that due to the size of the PRR and, therefore, the size of the bitstreams, there is a limitation in the HWICAP linux driver, since the Planahead design tool (used for bitstream generation) and the HWICAP module version are exactly the same with the ones used by G. Nikoloudakis.

After modifying the test application to perform the reconfiguration call twice, we were able to get correct encrypted data for both the AES and DES cores.

Chapter 8

Experimental Results

In this chapter, we describe the experimental results that were extracted during system testing.

8.1 Methodology

For the purpose of measuring the time needed to perform the reconfigurations, we used the software function int gettimeofday(struct timeval *tv, struct timezone *tz); which is defined in the libraries <time.h> and <sys/time.h>. This function gives the number of seconds and microseconds since the Epoch[1] and thus, through the use of two timeval arguments, we can measure elapsed time in microsecond precision.

[1] 00:00:00 UTC, January 1, 1970

8.2 Analysis of Results

The process of dynamic reconfiguration includes two steps, as described previously:

- Transferring the bitstream in a buffer stored in the DDR RAM.
- Writing the bitstream to the ICAP.

During experimental testing and online research, we realised that Linux uses a dynamic amount of the available free RAM, as a cache, in order to store transferred pages from the CF and have them available for possible future usage. Thus, we made two measurements for the time needed to transfer the bitstreams, an initial read and a cached read. Also, the PowerPC processor offers the possibility of enabling and disabling the onboard cache (16KB Instuction cache and 16KB Data cache). We made measurements with the CPU cache both enabled and disabled, in order to examine its impact to the general system performance. After creating the system and partial bitstreams, we noticed that, even though the netlists used for the reconfigurable modules, the position of the PRR and the placement of the Bus Macros were exactly the same, the resulting bitstreams of the reconfigurable modules for the two systems varied slightly in size. The sizes of the bitstreams are depicted in Table 8.1:

Reconfigurable Modules	Bitstream Size (bytes) (CPU Cache Disabled)	Bitstream Size (bytes) (CPU Cache Enabled)
AES	752245	755393
DES	743289	752177
Blank Encrypt	741862	744790
Adder	124478	125250
Multiplier	130991	130955
Blank Math	124513	124449

Table 8.1: Bitstream Sizes

Tables 8.2 and 8.3 contains the times measured with the CPU cache disabled and Tables 8.4 and 8.5 contains the times measured with the CPU cache enabled. These values are depicted in Figures 8.1 and 8.2 as well. In Figure 8.3 we present a direct comparison of the performance increase we achieved by enabling the CPU cache, in combination with the DDR-RAM cache provided by the Linux kernel.

Partial Bitstreams	$t_{CFtoDDR}$ (msec)	$t_{DDRtoCM}$ (msec)	t_{TOTAL} (msec)
AES	795.6181	144.5136	940.1317
DES	789.0413	141.2400	930.2813
Blank Encrypt	786.3364	142.84	929.1764
Adder	132.0619	26.8564	158.9183
Multiplier	136.1577	28.2064	164.3641
Blank Math	132.3275	27.0257	159.3532

Table 8.2: Results with CPU cache disabled. Direct transfer of bitstream from the CF

Partial Bitstreams	$t_{CFtoDDR}$ (msec)	$t_{DDRtoCM}$ (msec)	t_{TOTAL} (msec)
AES	32.1553	144.5136	176.6689
DES	31.5876	141.2400	172.8276
Blank Encrypt	31.5234	142.84	174.3634
Adder	5.7270	26.8564	32.5834
Multiplier	6.0622	28.2064	34.2685
Blank Math	5.9139	27.0257	32.9396

Table 8.3: Results with CPU cache disabled. Cached read of bitstream from the DDR-RAM cache

Partial Bitstreams	$t_{CFtoDDR}$ (msec)	$t_{DDRtoCM}$ (msec)	t_{TOTAL} (msec)
AES	709.4078	124.0119	833.4197
DES	704.7288	123.6254	828.3542
Blank Encrypt	698.2548	122.9256	821.1804
Adder	116.6073	21.9517	138.5590
Multiplier	120.5624	22.6724	143.2348
Blank Math	116.6983	21.6132	138.3115

Table 8.4: Results with CPU cache enabled. Direct transfer of bitstream from the CF

Partial Bitstreams	$t_{CFtoDDR}$ (msec)	$t_{DDRtoCM}$ (msec)	t_{TOTAL} (msec)
AES	15.5578	124.0119	139.5698
DES	15.4021	123.6254	139.0275
Blank Encrypt	15.2073	122.9256	138.1329
Adder	2.5864	21.9517	24.5381
Multiplier	2.7138	22.6724	25.3863
Blank Math	2.6666	21.6132	24.2798

Table 8.5: Results with CPU cache enabled. Cached read of bitstream from the DDR-RAM cache

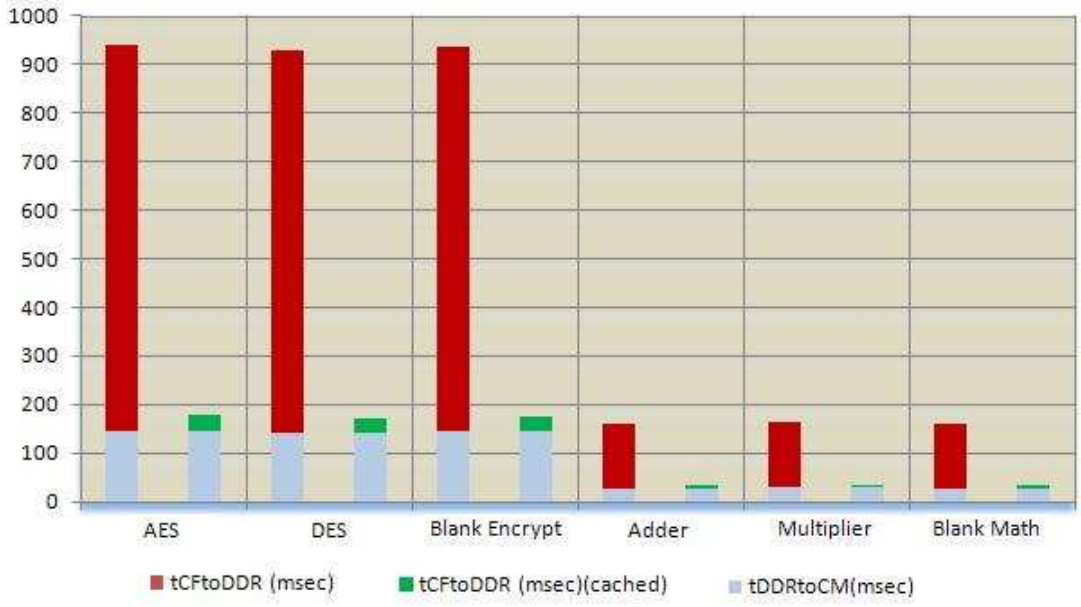

Figure 8.1: Measured Times with CPU cache disabled

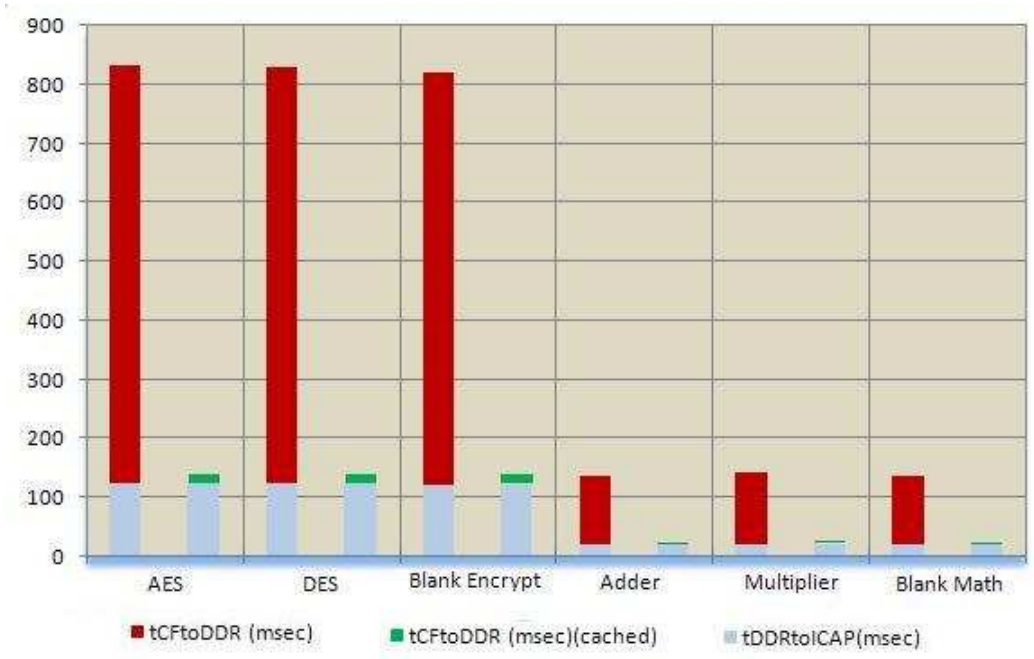

Figure 8.2: Measured Times with CPU cache enabled

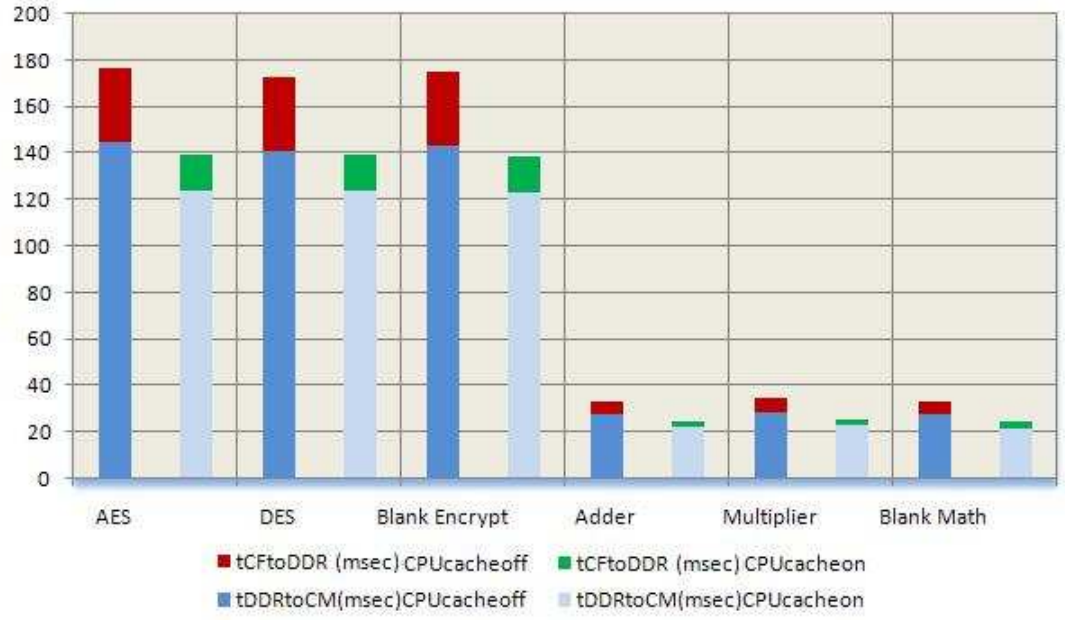

Figure 8.3: Comparison of measured times with CPU cache disabled and enabled.

Tables 8.6 and 8.7 makes a comparison between these times, illustrating the performance increase for each bitstream, as well as on average, that was achieved by enabling the CPU cache.

Partial Bitstreams	CFtoDDR	DDRtoCM	Total Time
AES	10.84%	14.19%	11.35%
DES	10.6%	12.47%	10.96%
Blank Encrypt	11.2%	13.94%	11.62%
Adder	11.7%	18.26%	12.81%
Multiplier	11.45%	19.62%	12.86%
Blank Math	11.81%	20.03%	13.2%
Average:	11.27%	16.48%	12.22%

Table 8.6: Improvement by enabling CPU Cache. Direct transfer of bitstream from the CF

Partial Bitstreams	CFtoDDR	DDRtoCM	Total Time
AES	51.62%	14.19%	21%
DES	51.24%	12.47%	19.56%
Blank Encrypt	51.76%	13.94%	20.78%
Adder	54.84%	18.26%	24.69%
Multiplier	55.23%	19.62%	25.92%
Blank Math	54.91%	20.03%	26.29%
Average:	53.27%	16.48%	23.09%

Table 8.7: Improvement by enabling CPU Cache. Cached read of bitstream from the DDR-RAM cache

Tables 8.8 and 8.9 illustrate information extracted from the experimental results that can be used for estimating reconfiguration times for future designs. Reconfiguration time per ColumnSlice is included due to the fact that reconfiguration on a Virtex-II Pro FPGA, re-programs the entire Column of the FPGA. The encryption cores occupy 72 Columns of the FPGA and 9920 slices, while the mathematical cores occupy only 12 Columns and 384 slices.

Partial Bitstreams	Time(msec) per KB of Bitstream	Time (msec) per Slices	Time (msec) per ColumnSlice
AES	1.2798	0.0948	13.0574
DES	1.2816	0.0938	12.9206
Blank Encrypt	1.2826	0.0937	12.9052
Adder	1.3073	0.4138	13.2432
Multiplier	1.2849	0.4280	13.6970
Blank Math	1.3105	0.4150	13.2794
Average:	1.2911	–	13.1838

Table 8.8: Elementary Reconfiguration Times with CPU cache disabled

Partial Bitstreams	Time(msec) per KB of Bitstream	Time (msec) per Slices	Time (msec) per ColumnSlice
AES	1.1298	0.0840	11.5753
DES	1.1277	0.0835	11.5049
Blank Encrypt	1.1290	0.0828	11.4053
Adder	1.1328	0.3608	11.5466
Multiplier	1.1200	0.3730	11.9362
Blank Math	1.1381	0.3602	11.5260
Average:	1.1296	–	11.5824

Table 8.9: Elementary Reconfiguration Times with CPU cache enabled

One observation that can be made is that the Reconfiguration time per number of slices increases as the designs get smaller. This can be explained by the fact that during reconfiguration, the entire column of the FPGA gets reprogrammed. The time required to re-program an FPGA Column can be seen in the third column of tables 8.8 and 8.9 and is about 13.2 msec (with enabled CPU cache) and 11.6 msec (with disabled CPU Cache). If the PRR extends to a large portion of a Column, then the required time to reconfigure a single slice of the design is reduced (as it reconfigures more design slices per column). Thus, the time required to reconfigure a single slice of the Encryption cores is much smaller than the time required for the mathematical cores (which are quite smaller in size).

In table 8.10, we present the Throughput of the Compact Flash and the HWICAP peripheral as extracted from the previous measurements.

	Without Cache	With Cache
CFtoDDR	0.9023 MB/sec	1.0213 MB/sec
DDRtoICAP	4.6965 MB/sec	5.64 MB/sec

Table 8.10: Transfer Rates

8.2.1 Comparison with non-OS Implementation

In the following tables, we make a comparison between our implementation and the implementation of G. Nikoloudakis. The cores used in both implementations were exactly the same. We only show the results taken with the CPU cache disabled as this was the case with the other implementation.

Present Implementation	Nikoloudakis [32]
CF to DDR	CF to PLB_BRAM
4.6965 MB/sec	1.15 MB/sec
DDR to ICAP	PLB_BRAM to ICAP
0.9023 MB/sec	0.260 MB/sec

Table 8.11: Comparison of transfer rates

	Present Implementation	Nikoloudakis [32]
Time/slice of PRR	0.0941 ms	0.360924215 ms
Time/KB of bitstream	1.2911 ms	5.074729498 ms
Time/ColumnSlice	13.1838 ms	49.72733628 ms

Table 8.12: Comparison of elementary Reconfiguration Times

	Present Implementation	Nikoloudakis [32]	Improvement
Bitstream Transfer(sec)	0.795618	2.782043	3.4967x
Write Bitstream to CM(sec)	0.144514	0.620120	4.2910x
Total(sec)	0.940132	3.402163	3.6188x

Table 8.13: Comparison for the AES Encryption core

	Present Implementation	Nikoloudakis [32]	Improvement
Bitstream Transfer(sec)	0.789041	2.720075	3.4473x
Write Bitstream to CM(sec)	0.141240	0.605509	4.2871x
Total(sec)	0.930281	3.325583	3.5748x

Table 8.14: Comparison for the DES Encryption core

	Present Implementation	Nikoloudakis [32]	Improvement
Bitstream Transfer(sec)	0.786336	2.500617	3.18x
Write Bitstream to CM(sec)	0.143457	0.557390	3.8854x
Total(sec)	0.929176	3.058007	3.2911x

Table 8.15: Comparison for the Blank bitstream

Chapter 9

Conclusions and Future Work

For the purpose of this thesis, we implemented an autonomous Linux-based Dynamic Reconfiguration Task Manager. The system supports virtually any type of reconfigurable peripheral that can fit in the FPGA with no changes in system set-up. We described the process of implementing such a system as well as overcoming different problems and issues that may emerge.

The experimental results were quite promising. They showed a great improvement in all areas regarding dynamic reconfiguration, compared to a non-os implementation, thus reducing the impact of DPR in system performance. The use of the much faster DDR (compared to the CF) as a temporary storage buffer greatly reduces the overhead of reconfiguration as it eliminates the need of constanly retrieving the bitstreams from the Compact Flash.

As part of future work, the following modifications can be implemented, reducing the cost of DPR :

- Taking advantage of the memory management performed by Linux, by retriev-

ing the bitstreams during system boot. The bitstreams are then stored in the RAM cache. User applications will retrieve the bitstreams from the cache and not from the CF, thus eliminating the need to re-transfer them from the CF

- Implementing a DMA controller for the transfer of the bitstreams from the CF to the DDR and from the DDR to the ICAP without the need of the processor.
- Dividing the PRR into smaller PRRs, thus resulting in smaller bitstreams.
- Improving the OPB/DCR socket driver and extending it to support multiple instantiations of the peripheral, thus allowing for more PRRs to be attached to the system. This provides the capability to add different types of reconfigurable peripherals for various operations.
- Extending the system by adding Ethernet support (making it possible to retrieve bitstreams from a network and/or updating local bitstreams)
- Applying encryption and/or compression to bitstreams, thus increasing security and size of bitstreams.

Bibliography

[1] Bee2OperatingSystem - BEE2Wiki. http://bee2.eecs.berkeley.edu/wiki/Bee2OperatingSystem.html.

[2] BitKeeper: The scalable Distributed Software Configuration Management System. http://www.bitkeeper.com/.

[3] Building and Testing gcc/glibc cross toolchains. http://kegel.com/crosstool/.

[4] BusyBox. http://www.busybox.net/.

[5] ChipScope Pro and the Serial I/O Toolkit. http://www.xilinx.com/tools/cspro.htm.

[6] DENX Software Engineering. http://www.denx.de/wiki/DULG/ELDK.

[7] DENX Software Engineering. http://www.denx.de/en/News/WebHome.

[8] Digilent. http://www.digilentinc.com.

[9] Embedded Linux - Wikipedia, The free encyclopedia. http://en.wikipedia.org/wiki/Embedded_Linux.

[10] Integrating an EDK Custom Peripheral with a LocalLink Interface Into Linux. http://www.itee.uq.edu.au/∼jwilliams/mblaze-uclinux/powerpc_linux/.

[11] Integrating an EDK Custom Peripheral with a LocalLink Interface Into Linux. http://www.xilinx.com/support/documentation/application_notes/xapp1129.pdf.

[12] Linux 2.4 for PowerPC development tree. http://ppc.bkbits.net:8080/linuxppc_2_4_devel/.

[13] Linux on the Xilinx ML300. http://www.klingauf.de/v2p/index.shtml.

[14] List Of Operating Systems - Wikipedia, The free encyclopedia. http://en.wikipedia.org/wiki/List_of_operating_systems#Embedded.

[15] Microblaze uCLinux Project. http://www.itee.uq.edu.au/∼jwilliams/mblaze-uclinux.

[16] ML310 Linux configuration. http://www.xilinx.com/products/boards/ml310/current/reference_designs/base/linux/ramdisk.image.gz.

[17] Mosaic: Embedded Linux on the XUPV2P Development Board. http://www.cs.washington.edu/research/lis/mosaic/xup_ppc_linux.shtml.

[18] Open Source Linux Project Xilinx. http://xilinx.wikidot.com/open-source-linux.

[19] Partial Reconfiguration Early Access software tools for ISE 9.1i SP2. http://www.xilinx.com/support/prealounge/protected/archive_91.htm.

[20] Petalogix - Linux Solutions for a Reconfigurable world. http://www.petalogix.com/.

[21] Software-defined radio. http://en.wikipedia.org/wiki/Software-defined_radio.

[22] The GNU General Public License. http://www.gnu.org/copyleft/gpl.html.

[23] The GNU Lesser General Public License. http://www.gnu.org/copyleft/lesser.html.

[24] Ubuntu Linux. www.ubuntu.com.

[25] uCLinux - Embedded Linux/Microcontroller Project. http://www.uclinux.org/index.html.

[26] Xilinx git tree repository. http://git.xilinx.com/cgi-bin/gitweb.cgi.

[27] A. Anifantis. Performance Evaluation of Dynamic Reconfiguration in Modern Field Programmable Gate Arrays, Diploma Thesis, ECE Dpt., Technical University of Crete, 2007.

[28] E. S. Arnaud Lagger, Andres Upegui and I. Gonzalez. Self-reconfigurable pervasive platform for cryptographic application. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–4, Aug 2006.

[29] G. Brebner. The swappable logic unit a paradigm for virtual hardware. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 77 – 86, Apr 1997.

[30] C. Effraimidis. A Self Reconfigurable Architecture To Support Multiple Fitness Functions In Genetic Algorithm, Diploma Thesis, ECE Dpt., Technical University of Crete, 2009.

[31] H. W. Christoph Steiger and M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, 2004.

[32] G. Nikoloudakis. Design and Implementation, Using Dynamic Partial Reconfiguration, Of A System Implementing Multiple Cryptographic Algorithms, Diploma Thesis, ECE Dpt., Technical University of Crete, 2009.

[33] K. K., M. F., and K. K. Run-time management of reconfigurable hardware tasks using embedded linux. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 209 – 215, Dec 2007.

[34] D. S. Marco D. Santambrogio, Vincenzo Rana. Operating system support for online partial dynamic reconfiguration management. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 455–458, Sept 2008.

[35] Michael Gernoth. XILINX JTAG tools on Linux without proprietary kernel modules. http://www.rmdir.de/∼michael/xilinx/.

[36] Milan Saini, Technical Marketing Manager, Xilinx, Inc. Unleash Your Creativity with Embedded Linux on Virtex-II Pro FPGAs. *Xcell Journal*, 2003.

[37] J. Williams and N. Bergmann. Embedded linux as a platform for dynamically self-reconfiguring systems-on-chip. *IEEE Transactions on Computers*, 53(11):1393–1407, 2004.

Appendices

Appendix A

In Appendix A, we describe the process of compiling the two unsuccesful attempts, the problems we faced and the solutions found. The first attempt was with the Petalinux provided by Petalogix [20].The most current version at the time was 0.30rc1.

The Petalinux distribution provides all the necessary tools and compilers for compiling the kernel sources as well as versions 2.4 and 2.6 of the Linux kernel. We downloaded the sources from the official size and extracted them. Since, the XUP Virtex-II Pro is not included in the reference designs, we followed the Tutorial Guide on how to create a design from scratch.

A.1 Compiling the Petalinux 2.6 kernel version

1. The first step is to set up the appropriate environment variables for Petalinux. This can be easily done with the following command:

```
source ./settings.sh
```

2. Next, we created a new vendor and platform combination for our target system. The first attempt was to use the 2.6 version of the kernel as it is newer and more

optimized. Thus we issued the following command:

```
petalinux-new-platform -v Xilinx -p XUPV2P -k 2.6
```

which sets up a new platform "XUPV2P" under vendor name "Xilinx" for the 2.6 kernel. This platform can be removed by using the "petalinux-remove-platform" command

3. Following, the next step is to set up our newly created platform as the current build target, as this allows the user to edit and subsequently save the settings configured for this platform. This can be done by using the menuconfig tool to select the platform from the Vendor/Product Selection menu.

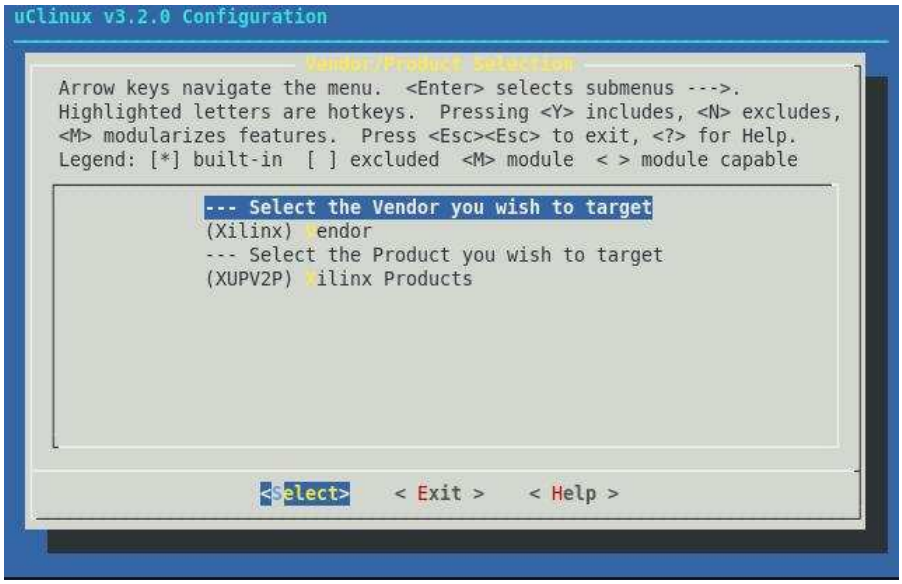

Figure A.1: Setting the Platform as the target system

After saving and exiting from the menuconfig tool, a default configuration for our system is created. It provides a baseline to start configuring the platform. Before creating the hardware project, we need to set up a Hardware Design Project Directory. This can be done by changing directory to the $PETALINUX/hardware/user-platforms/ and creating a new directory to host our project. We then have to copy the Petalinux bsp files (located in $Petalinux/hardware/edk_user_repository/) into our project directory, or create a symbolic link using the following command:

```
ln -s ../../edk_user_repository edk_user_repository
```

4. Next, we proceed with creating our hardware platform through the use of the EDK design tool. The system configuration we used is as follows:

   - A Microblaze processor, running at 100MHz with 8KB of local memory and enabled cache.
   - OPB UARTLITE peripheral with 115200 baudrate and interrupt enabled.
   - 128MB MCH OPB DDR RAM to be used as main memory for Linux
   - One OPB Timer with 32 counter bit width and interrupt enabled
   - 8KB Data and 8KB Instruction cache for the DDR RAM.

   The Tutorial guide, also listed an Ethernet MAC peripheral but it was excluded from our design due to a lack of license.

   Certain modifications were required before proceeding with the generation of libraries and netlists:

- Edit the debug_module peripheral to disable the UART Interface on OPB.
- Enable interrupt for the debug_module by setting its interrupt Net to "debug_module_interrupt" and adding it to the opb_intc's list of connected interrupts.
- Include the Petalinux BSP. This is done by setting the Repository peripheral (located in the Project Options dialog box) to /path/to/Petalinux/hardware/project/dir/edk_user_repository
- Add FS-Boot bootloader. This is done by creating a new Software Application in the EDK,adding the existing files located in $Petalinux/hardware/fs-boot and editing the Compiler Options for this application to match the ones depicted in the following table: The FS-Boot Application was marked

Attribute	Value
Environment	
Application Mode	executable
Output ELF file	default value
Linker Script	Use default Linker Script
Debug and Optimisation	
Optimization Level	Size Optimized (-Os)
Advance	
Other Compiler Options to Append	-Wall

Table A.1: FS-Boot Compiler Options

to Initialize the processor's BRAM.

- Configure the system software settings with following attributes:

Attribute	Value
Processor Parameters, Driver and Interrupt Handlers	
xmdstub_peripheral	none
OS and Library Settings	petalinux
PetaLinux version	1.00b
OS and Library	
lmb_memory	dlmb_cntlr
main_memory	DDR_128MB_16MX64_rank1_row13_col9_cl2_5
main_memory_bank	0
stdout	RS232_Uart_1
stdin	RS232_Uart_1

Table A.2: Software Platform Settings

5. We, then, proceeded to configure the Petalinux. The Petalinux distribution includes a special script which allows for automated synchronisation of hardware and software system configurations. From within the edk project directory, we issued the following command:

```
petalinux-copy-autoconfig
```

We then proceeded by invoking the menuconfig tool by issuing the following command from within the $Petalinux/software/petalinux-dist directory:

```
make menuconfig
```

We then enabled the Customize Kernel Settings and Customize Vendor/User Settings optionsA.2.

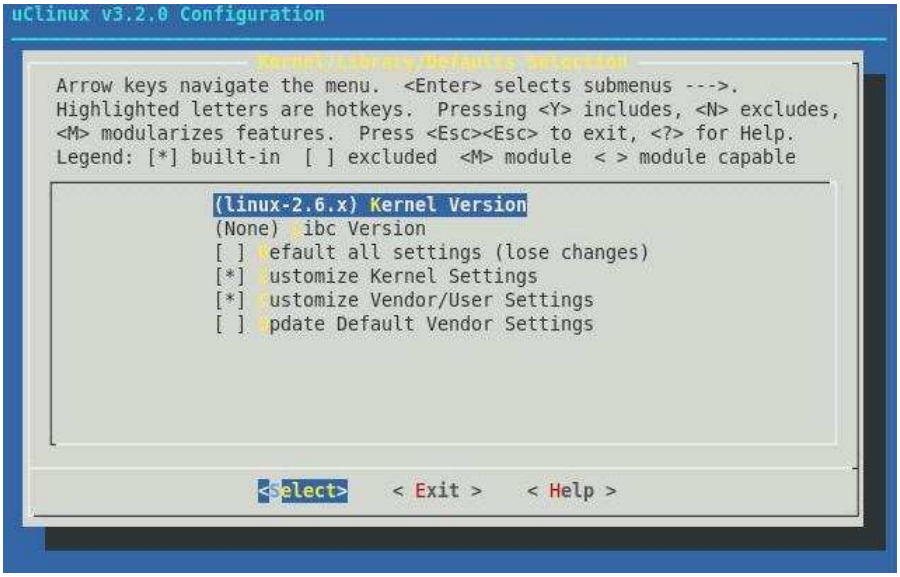

Figure A.2: Settings Options

These options allow for configuration of the Linux kernel and Vendor/User Settings. We made the following changes, according to the guide, by navigating to the appropriate menus

- In Kernel Settings we made the following modifications:
  - Disabled the GPIO driver (located in Device Drivers→Character Devices→Xilinx OPB GPIO Support)

  - Enabled the UARTLITE console support (located in Device Drivers → Character Devices → Xilinx uartlite serial port support and Support for console on Xilinx uartlite serial port)
  - Disabled Networking support as there were no network devices present in our system (located in Networking → Networking Support)
  - Enabled support for Intel/Sharp flash chips (located in Device Drivers → Memory Technology Devices(MTD) → RAM/ROM/FLASH chip drivers → Detect non-CFI AMD/JEDEC-compatible flash chips)

- In Vendor/User Settings we enabled the Build U-Boot option (located in System Settings)

We, then, re-run the menuconfig tool to update the settings and proceeded with the compilation of the kernel

```
yes '' '' | make oldconfig dep
make all
```

After running the "make all" command an error occured about an undeclared variable called "PATH_MAX". By doing some research, we modified the sumversion.c file located in $PETALINUX/software/petalinux-dist/scripts/mod/ by adding the following line in the beginning of the file:

```
#include <linux/limits.h>
```

This eliminated the error and compilation proceeded. A new error occured concerning jedec_probe.c file. Disabling support for the Intel/Sharp flash chips

eliminated the error (There are no flash components in our system, so disabling this driver has no effect on the functionality of the system).

Compilation carried on but another error occured concerning a syntax error in the petalinux-uboot-config script

```
u-boot/petalinux-uboot-config:  line 177:  + :  syntax error:
 operand expected (error token is " ")
```

By examining the source code of the petalinux-uboot-config script, we found out that this error concerned a FLASH Memory which was unavailable on our board. Thus, we proceeded by disabling the build U-boot option which allowed for correct compilation of the Linux kernel. However, due to the problems described above and the limitations they impose (the need to manually download the kernel on the board by using the xmd software), we deciced to try the 2.4 version of the kernel.

A.2 Compiling the Petalinux 2.4 kernel version

Compiling the 2.4 version requires the same steps as described above except for a few differences.

- The creation of the new platform is accomplished by using the following command

```
petalinux-new-platform -v Xilinx -p XUPV2P -k 2.4
```

- Setting the Platform as the target system now requires setting the Kernel Version to linux-2.4.x (figure A.3)

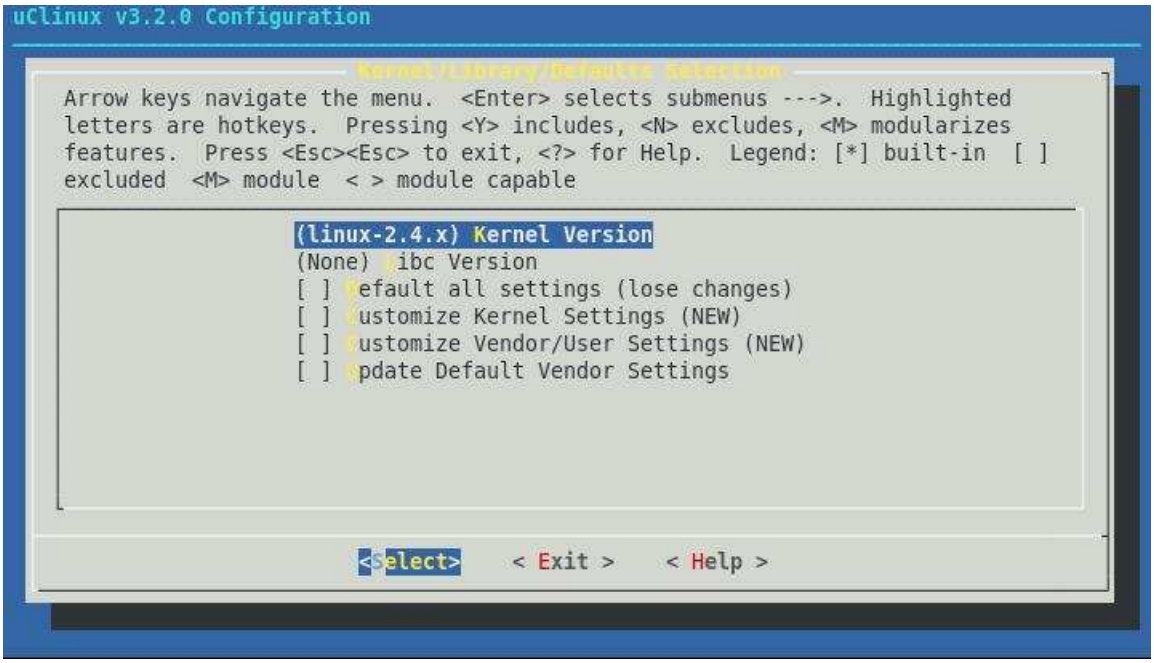

Figure A.3: Setting the Platform as the target system

The other settings were configured as described previously with a few changes in the kernel paths. During compilation an error occured, concerning an undeclared variable "CONFIG_XILINX_FLASH_START". During research for a solution, we came along a workaround solution to this problem by defining two random values for CONFIG_XILINX_FLASH_START and CONFIG_XILINX_FLASH_SIZE in the auto-config.in file located in $PETALINUX/software/petalinux-dist/linux-2.4.x/arch/microblaze/platform/Xilinx-XUPV2P/

for example :

```
define_hex CONFIG_XILINX_FLASH_START 0x82000000
define_hex CONFIG_XILINX_FLASH_SIZE 0x00800000
```

This resolved the issue, however correct functionality of the kernel cannot be guaranteed. However, a new error occured, concerning undeclared functions regarding the ethernet connectivity during U-Boot bootloader compilation. Enabling Networking support and enabling network devices resulted in further error regarding the absence of declared peripheral addresses. Thus, we disabled the compilation of the U-Boot bootloader to see if we could compile the kernel. This resulted in a successful compilation of the kernel, which we downloaded on the board by using the xmd command.

The Petalinux distribution was therefore abandoned as an idea, as it didn't meet the specifications set for this thesis.

Appendix B

B.1 Compiling PowerPC Linux 2.4

The second attempt on compiling a Linux kernel was based on a guide by the department of Computer Science and Engineering at the University of Washington [17].

Since the architectures of the host machine and target board are different, the use of a cross-compiler for the target architecture is required. The Crosstool [3] toolchain was selected. After downloading and unzipping the tarball of the latest version (0.43) at the time of this document, we proceeded by editing the appropriate download/build script for our architecture. More specifically, the changes made are:

- The line RESULT_TOP=/opt/crosstool was replaced by
  RESULT_TOP=$PREFIX/crosstool
- The line

  ```
  eval 'cat powerpc-405.dat gcc-3.4.1-glibc-2.3.3.dat' sh all.sh --notest
  ```

  was replaced by

```
eval ‘cat powerpc-405.dat gcc-3.4.4-glibc-2.3.3.dat‘ sh all.sh --notest
```

We then set the following environment variables in a linux console and executed the script:

```
export TARGET=powerpc-405-linux
export PREFIX=<target_directory>/$TARGET
unset LD_LIBRARY_PATH
sh demo-ppc405.sh
```

In order for the script to function correctly, we had to downgrade the gcc compiler to version 3.4.

The final step was to add the cross-compiler's bin directory to the compiler search path by running export PATH=$PREFIX/bin:$PATH in a console. This has to be done each time a new console is open.

The process of configuring and compiling the kernel, requires several different steps, as shown below:

1. Firstly, a baseline system must be created in EDK so that the bitstream and base support package (BSP) can be created. The configuration we used for our system is shown below:

    - PowerPC 405 at 300MHz
    - JTAG debug
    - No Data OCM
    - No Instruction OCM

- Cache Enabled for instructions and data with burst enabled.
- OPB UART 16500 with interrupt
- OPB Sysace with interrupt
- PLB 128MB DDR without interrupt
- PLB BRAM IF CNTRL (128 KB)

The ppc405_0_bootloop was selected as the default application to initialize the BRAM and then, Generate Bitstream was selected to create the netlist and the download bitstream.

2. We, then proceeded with the generation of the Base Support System or BSP. It is a set of files required so that Linux will compile and contain the required drivers and .h files for our system. We made the following changes in the software platform and settings menu:

   - Under Software Platform, we set ppc405_0 to use linux: linux_mvl31 version 1.01a
   - Under Library/OS Parameters we set MEM_SIZE = 0x08000000 for our 128MB RAM, PLB_CLOCK_FREQ_HZ = 100000000 and TARGET_DIR = /path/to/store/generated/bsp/files.
   - We added all devices that Linux should know about(under connected_periphs): RS232, SysACE, OPB interrupt controller.

   We, then, continued with generation of the BSP files by clicking Generate Libraries and BSPs in the Tools menu. The resulting BSP files were tarred up.

3. The next step involves retrieving the Linux sources using bitkeeper [2], installing the BSP, and patching the kernel.

   - Retrieving the kernel sources was performed by using bitkeeper:

     ```
     ./bkf clone bk://ppc.bkbits.net/linuxppc_2_4_devel
     linuxppc-2.4.26
     ```

   - Installing the BSP files was performed by untarring the archive created in step 2 in the root directory of the kernel sources
   - In order for the kernel to compile successfully for our target system, it was necessary to apply a patch located on the website of the tutorial. The archive was untarred in the kernel sources and the touchup_linuxppc_tree.sh shell script was executed.

4. The final step involves configuring and compiling the Linux Kernel. As the default configuration to base our system, we used the ML310 Linux configuration file on the Xilinx website [16]. We downloaded and untarred the configuration file in <linux_kernel_src_dir>/arch/ppc/boot/images/. We made the following modifications to the Makefile (located in the root directory of the kernel sources):

   ```
   ARCH := ppc
   CROSS_COMPILE = powerpc-405-linux-gnu-
   ```

   Running "make oldconfig" imports the default configurations from the downloaded config file. Based on this configuration, we changed the following settings:

- "console=ttyS0,38400 root=/dev/xsysace/disc0/part3 rw" as the Default bootloader kernel arguments (located in General Setup). This tells the kernel to look for a console on the ttyS0 device at 38400 baudrate, to look for a filesystem on the third partition of the compact flash and mount the filesystem with read and write privileges.
- Disabled Xilinx on-chip GPIO (located in Character Devices)

The Linux kernel was successfully compiled by issuing the following commands:

```
make dep
make zImage.initrd
```

B.2 Integrating the Icap Driver

In order for the operating system to correctly probe and set up the HWICAP module, it is necessary to provide drivers for it and integrate them in the kernel sources. We decided to integrate the HWICAP linux driver found in the kernel sources of the uCLinux [15]. We copied the files to the <linux_kernel_src_dir>/drivers/misc directory. In order to integrate it in the kernel, the following steps were necessary:

- Correct the path of the EXTRA_CFLAGS in the Makefile of the driver to point to the correct xilinx_ocp (-I$(TOPDIR)/arch/ppc/xilinx_ocp).

- Add the following lines in the Config.in file located in the misc directory:

```
if [ ‘‘$CONFIG_40x’’ = ‘‘y’’ ]; then
tristate ’Xilinx ICAP driver’ CONFIG_XILINX_HWICAP
fi
```

- Add the following lines to the Makefile located in the misc directory:

```
mod-subdirs += xilinx_hwicap
subdir-$(CONFIG_XILINX_HWICAP) += xilinx_hwicap
obj-$(CONFIG_XILINX_HWICAP) += xilinx_hwicap/xilinx_hwicap.o
```

- Add the following line to the Config.in file located in < *linux_kernel_src_dir*/arch/ppc/

```
source drivers/misc/Config.in
```

- Add the following lines to the adapter.c file located in xilinx_hwicap folder

```
source drivers/misc/Config.in
```

- Add the missing files xpacket_fifo_l_v2_00_a.c, xpacket_fifo_l_v2_00_a.h, xpacket_fifo_v2_00_a.c and xpacket_fifo_v2_00_a.h to the arch/ppc/xilinx_ocp folder. These files can be easily found by searching the internet.

Appendix C

C.1 OPB/DCR Socket Device Driver

Below follows the device driver we developed in order to support the OPB/DCR socket peripheral which was needed for accessing the reconfigurable peripherals. It is implemented by two files, socket.c and socket.h

socket.c:

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/ioport.h>
#include <linux/interrupt.h>
#include <linux/xilinx_devices.h>
#include <asm/dcr.h>
#include <linux/of_device.h>
#include <linux/of_platform.h>

#include "socket.h"

/*
 * The major device number used by this driver.
 */
#define SOCKET_DEV_MAJ       177

#define REG_CTRL_ENABLE  0x00000001
```

```
#define REG_CTRL_DISABLE 0x00000000
#define NUM_REGS 1
#define REG_SOCKET_ADDR 0
#define REG_CTRL 1

#define DRIVER_NAME         "socket"
#define DRIVER_DESCRIPTION  "Xilinx OPB/DCR Socket driver"
#define DRIVER_VERSION      "1.00a"

/*
 * The OPB/DCR Socket has DCR interface to its bus macros enable input. This
 * macro allows for direct access to the dcr interface
 */
#define xilinx_opbdcrsocket_out_be32(driverdata, offset, val) \
dcr_write(driverdata->dcr_host, offset, val)

/*
 * The OPB/DCR Socket device structure.  Only a single instance is
 * provided for.
 */
struct socket_dev *socket_dev;
int socket_debug=0;
/*
 * socket_open:
 * The user has opened /dev/socket
 */
int socket_open(struct inode *inode, struct file *filp)
{
    struct socket_dev *dev; /* device information */
    int retval;

    retval = 0;
    dev = container_of(inode->i_cdev, struct socket_dev, cdev);
    filp->private_data = dev; /* for other methods */

    mutex_lock(&dev->mutex);

    /*
     * Only one writer and only one reader of the device.
     */
    switch (filp->f_flags & O_ACCMODE) {
    case O_RDONLY:
        if (dev->readers) {
            retval = -EBUSY;
            goto out;
        } else {
            dev->readers++;
        }
        break;
    case O_WRONLY:
        if (dev->writers) {
            retval = -EBUSY;
            goto out;
        } else {
            dev->writers++;
```

```
        }
        break;
    case O_RDWR:
    default:
        if (dev->writers || dev->readers) {
            retval = -EBUSY;
            goto out;
        } else {
            dev->writers++;
            dev->readers++;
        }
    }
dev->write_to=0;
dev->read_from=0;
out:
    mutex_unlock(&dev->mutex);
    return retval;
}

/*
 * socket_release:
 * The user has closed the device file.
 */
static int socket_release(struct inode *inode, struct file *filp)
{
    struct socket_dev *dev = filp->private_data;

    mutex_lock(&dev->mutex);

    /*
     * Only one writer and only one reader of the device.
     */
    switch (filp->f_flags & O_ACCMODE) {
    case O_RDONLY:
        dev->readers--;
        break;
    case O_WRONLY:
        dev->writers--;
        break;
    case O_RDWR:
    default:
        dev->writers--;
        dev->readers--;
    }

    mutex_unlock(&dev->mutex);
    return 0;
}

/*
 * socket_read:
 * A userspace read of the device file.  Copy one of the receive
 * buffers to userspace
 */
ssize_t socket_read(struct file *filp, char __user *buf, size_t count,
```

```
                  loff_t *f_pos)
{
    struct socket_dev *dev = filp->private_data;
    ssize_t retval = 0;
    //struct llex_rx_buff *rx_buff;
    void* buff;
int val;
    int err;
buff = kmalloc(count + 1, GFP_KERNEL);
    mutex_lock(&dev->mutex);

    if (socket_debug) {
        printk(KERN_INFO "\n%s: START: f_pos %lld  count %d\n",
               __FILE__, *f_pos, count);
    }
    /*
     * Copy this buffer to user space.
     */
val=in_be32(socket_dev->mapaddr+socket_dev->read_from);
memcpy(buff, &val, 4);
    err = copy_to_user(buf, buff, 4);
    if (err < 0) {
        printk(KERN_INFO
               "There was a mistake in moving data to userspace");
        goto out;
    }
    count -= 4;
    buf += 4;
    *f_pos += 4;
    retval += 4;

  out:

    mutex_unlock(&dev->mutex);
    return retval;
}

/*
 * socket_write:
 * A userspace write to the device file.
 */
ssize_t socket_write(struct file *filp, const char __user *buf,
                     size_t count, loff_t *f_pos)
{
    struct socket_dev *dev = filp->private_data;
    ssize_t retval = 0;
    void * buffer;
int val;
    mutex_lock(&dev->mutex);

    buffer = kmalloc(count + 1, GFP_KERNEL);
    if (buffer == NULL) {
        retval = -ENOMEM;
        goto out;
```

```
    }
    memset(buffer, 0, 4);
    if (copy_from_user(buffer, buf, count)) {
        retval = -EFAULT;
        goto out;
    }
memcpy(&val,buf,4);

    out_be32(socket_dev->mapaddr+socket_dev->write_to,val);
        retval = 4;
    *f_pos += count;

  out:
    mutex_unlock(&dev->mutex);
    return retval;
}

/*
 * socket_ioctl:
 * This function is used to set the write and read offset to
 * the base address of the device, in order to read from any
 * different peripheral registers. It is also used to disable
 * and enable the bus macros.
 */
int socket_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
 unsigned long arg)
{
switch(cmd){
case 0:
socket_dev->write_to=0;
break;
case 1:
socket_dev->write_to=4;
break;
case 2:
socket_dev->write_to=8;
break;
case 3:
socket_dev->write_to=12;
break;
case 4:
socket_dev->write_to=16;
break;
case 5:
socket_dev->write_to=20;
break;
case 6:
socket_dev->write_to=24;
break;
case 7:
socket_dev->write_to=28;
break;
case 8:
socket_dev->write_to=32;
break;
case 9:
```

```
socket_dev->write_to=36;
break;
case 10:
socket_dev->write_to=40;
break;
case 11:
socket_dev->write_to=44;
break;
case 12:
socket_dev->write_to=48;
break;

case 13:
socket_dev->read_from=0;
break;
case 14:
socket_dev->read_from=4;
break;
case 15:
socket_dev->read_from=8;
break;
case 16:
socket_dev->read_from=12;
break;
case 17:
socket_dev->read_from=16;
break;
case 18:
socket_dev->read_from=20;
break;
case 19:
socket_dev->read_from=24;
break;
case 20:
socket_dev->read_from=28;
break;
case 21:
socket_dev->read_from=32;
break;
case 22:
socket_dev->read_from=36;
break;
case 23:
socket_dev->read_from=40;
break;
case 24:
socket_dev->read_from=44;
break;
case 25:
socket_dev->read_from=48;
break;

case 26:
dcr_write(socket_dev->dcr_host,0,REG_CTRL_DISABLE);
break;
case 27:
```

```
dcr_write(socket_dev->dcr_host,0,REG_CTRL_ENABLE);
}
return cmd;
}

struct file_operations socket_fops = {
        .owner =    THIS_MODULE,
        .read =     socket_read,
        .write =    socket_write,
        .open =     socket_open,
        .release =  socket_release,
.ioctl =    socket_ioctl,
};

/*
 * Remove the driver hooks from the system.
 */
static void socket_remove(void)
{
    dev_t devno;
        if (socket_dev->mapaddr) {
            iounmap(socket_dev->mapaddr);
        }

        kfree(socket_dev);

    devno = MKDEV(SOCKET_DEV_MAJ, 0);
    unregister_chrdev_region(devno, 1);
}

/*
 * socket_setup:
 * Hook the driver into the filesystem and perform hardware setup.
 */
static int socket_setup (struct socket_dev *socket_dev,dcr_host_t dcr_host,
   unsigned int dcr_start, unsigned int dcr_len)
{
    dev_t devno;
    int   err;

    devno = MKDEV(SOCKET_DEV_MAJ, 0);
    socket_dev->devno = devno;
    err = register_chrdev_region(devno, 1, "socket");
    if (err < 0) {
        printk(KERN_ERR "%s: Unable to register chrdev %d.\n",
               DRIVER_NAME, SOCKET_DEV_MAJ);
        goto cleanup;
    }
    /*
     * Memory Map SOCKET core to a virtual address
     */
    socket_dev->mapaddr = ioremap(socket_dev->physaddr, socket_dev->addrsize);
    printk("%s: phys addr : 0x%08x mapped to 0x%08x.\n", DRIVER_NAME,
 (u32)socket_dev->physaddr, (u32)socket_dev->mapaddr);
```

```
socket_dev->dcr_start = dcr_start;
socket_dev->dcr_len = dcr_len;
socket_dev->dcr_host = dcr_host;

/* Turn on the bus macros */
socket_dev->reg_ctrl_default = REG_CTRL_ENABLE;
     printk("%s: OPB/DCR DCR address: 0x%0x\n", DRIVER_NAME,
       socket_dev->dcr_start);
xilinx_opbdcrsocket_out_be32(socket_dev, 0,
socket_dev->reg_ctrl_default);

    /*
     * Plug the character device into the filesystem
     */
    cdev_init(&socket_dev->cdev, &socket_fops);
    socket_dev->cdev.owner = THIS_MODULE;
    err = cdev_add(&socket_dev->cdev, devno, 1);
    if (err) {
        printk(KERN_NOTICE "Error %d adding %s", err, DRIVER_NAME);
        goto cleanup;
    }
cleanup:
    if (err) {
        socket_remove();
    }
    return err;
}

/*
 * Initialize the driver
 * This function is called as the result of an Open Firmware (device tree)
 * match of an entry in socket_of_match[] against
 * linux/arch/powerpc/boot/dts/virtex405-xup.dts which contains an
 * "xlnx,opb-dcr-socket" compatible entry.
 */
static int __devinit socket_of_probe (struct of_device *ofdev,
                                      const struct of_device_id *match)
{
    struct resource     r_mem;
    int err;
int start, len;
dcr_host_t dcr_host;

    printk(KERN_INFO "Device Tree Probing \'%s\'\n",
                        ofdev->node->name);

    /*
     * Allocate a private structure to manage this device.
     */
    socket_dev = kmalloc(sizeof(struct socket_dev), GFP_KERNEL);
    if (socket_dev == NULL) {
        return -ENOMEM;
    }
    memset(socket_dev, 0, sizeof(struct socket_dev));
```

```
    mutex_init(&socket_dev->mutex);

    /*
     * What is the physical address of the peripheral?
     */
    err = of_address_to_resource(ofdev->node, 0, &r_mem);
    if (err) {
        dev_warn(&ofdev->dev, "invalid address\n");
        return err;
    }
    socket_dev->physaddr = r_mem.start;
    socket_dev->addrsize = r_mem.end - r_mem.start + 1;

  //Setting up dcr interface
start = dcr_resource_start(ofdev->node, 0);
len = dcr_resource_len(ofdev->node, 0);
dcr_host = dcr_map(ofdev->node, start, len);
if (!DCR_MAP_OK(dcr_host)) {
dev_err(&ofdev->dev, "invalid address\n");
return -ENODEV;
}
    err = socket_setup(socket_dev, dcr_host, start, len);

    return err;
}

static int __devexit socket_of_remove(struct of_device *dev)
{
    socket_remove();
    return 0;
}

/*
 * List of items which might be found in the device tree which could
 * be serviced by this driver.
 */
static struct of_device_id socket_of_match[] = {
    { .compatible = "xlnx,opb-dcr-socket", },
    {}
};

MODULE_DEVICE_TABLE(of, socket_of_match);

static struct of_platform_driver socket_of_driver = {
    .name        = DRIVER_NAME,
    .match_table = socket_of_match,
    .probe       = socket_of_probe,
    .remove      = __devexit_p(socket_of_remove)
};

/*
 * Absolute initialization entry point for the driver.
 * Everything begins here.
 */
static int __init socket_init(void)
```

```
{
    int status;
    status = of_register_platform_driver(&socket_of_driver);
    return status;
}

static void __exit socket_cleanup(void)
{
    of_unregister_platform_driver(&socket_of_driver);
}

module_init(socket_init);
module_exit(socket_cleanup);

MODULE_AUTHOR("Spanakis Manolis");
MODULE_DESCRIPTION(DRIVER_DESCRIPTION);
MODULE_LICENSE("GPL");
```

socket.h:

```
#ifndef __SOCKET_H__
#define __SOCKET_H__


/*
 * The socket device structure.
 */
struct socket_dev
{
int readers;
int writers;

dcr_host_t      dcr_host;
unsigned int    dcr_start;
unsigned int    dcr_len;
    /*
     * Device information
     */
    u32 reg_ctrl_default;
    int write_to;
    int read_from;
    void            *mapaddr;          /* virtual address of socket core */
    unsigned         physaddr;         /* bus address of socket core */
    unsigned         addrsize;         /* Size of periph. address space */
    dev_t            devno;
    struct mutex     mutex;
    struct cdev      cdev;
};

#endif
```