Master Thesis

FPGA support for OSGi

by
Ramon Küpfer

Advisors:
Jan S. Rellermeyer
Prof. Dr. Gustavo Alonso

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland
Abstract

Mobile devices implement much of their functionality in ASICs, an integrated circuit with a fixed implementation. Once manufactured and implemented in a mobile device the ASIC cannot be reconfigured compared to an FPGA. An FPGA is known for its reconfiguration ability to alter the fabric and implement new hardware modules in it. With the technology advance in FPGAs, partially reconfigurable FPGAs have been developed which can alter a part of their fabric during runtime. With an appropriate management lifecycle and module layer, the OSGi framework, an approach is provided to load hardware-supported modules into a software environment. This thesis introduces a new OSGi bundle, the FPGA Extender, which allows a partially reconfigurable embedded design to handle hardware-supported OSGi services and benefit of the hardware-acceleration. To evaluate a possible performance increase for Java application, a hardware and software implementation accessed with JNI is compared to a hardware and software implementation accessed in C. The presented approach shows the benefit when managing a partially reconfigurable system with the OSGi framework and that JNI hardware-implemented methods can almost out-perform C software-implemented methods.
# Contents

1 Introduction ........................................ 1  
   1.1 Motivation ...................................... 2  
   1.2 Outline ......................................... 4  

2 Background ......................................... 5  
   2.1 Introduction to FGPA ............................ 5  
      2.1.1 FPGA Chip ................................. 5  
      2.1.2 Configuration .............................. 7  
   2.2 Reconfigurable Designs ......................... 12  
      2.2.1 Modular Design Flow ...................... 14  
      2.2.2 Early Access Design Flow ............... 15  
      2.2.3 Bitstream properties .................... 20  
   2.3 Accessing the Hardware with JNI .............. 20  
   2.4 OSGI ........................................... 21  

3 Related Work ...................................... 29  

4 PR Designs ......................................... 35  
   4.1 A reconfigurable design ........................ 35  
   4.2 Design Flow ................................... 38  

5 Architecture ....................................... 43  
   5.1 FPGA Extender .................................. 43  
      5.1.1 JNI to access the Hardware .............. 47  
      5.1.2 Java Accessing the ICAP ................. 47  
      5.1.3 Socket Bridge ............................. 48  

6 Evaluation ......................................... 51  
   6.1 Improvements with FPGA Bundles ............... 51  
   6.2 Performance improvement through hardware acceleration 54  

7 Conclusions and Future Work .................... 65  

Bibliography ........................................ 71  

A Linux - OS and drivers ............................ 73  

B Setup & Configuration ............................. 77  
   B.1 ELDK - Embedded Linux Development Kit ....... 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2 Debootstrap</td>
<td>77</td>
</tr>
<tr>
<td>B.3 Impact - libusb driver</td>
<td>77</td>
</tr>
<tr>
<td>B.4 Bitstream Compilation</td>
<td>78</td>
</tr>
<tr>
<td>B.5 Virtex II Pro Board Configuration</td>
<td>79</td>
</tr>
<tr>
<td>B.5.1 9.2 board design</td>
<td>79</td>
</tr>
<tr>
<td>B.5.2 10.1 board design</td>
<td>81</td>
</tr>
<tr>
<td>B.5.3 Patching a bitstream with software code</td>
<td>82</td>
</tr>
<tr>
<td>B.5.4 Device Tree Generation and Kernel Compilation</td>
<td>82</td>
</tr>
<tr>
<td>B.5.5 Creating the SystemACE</td>
<td>83</td>
</tr>
<tr>
<td>B.6 JNI Header Generation</td>
<td>83</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The current development of computer systems tend to shrink the hardware. This is a fact stated by Moore’s law. The hardware specifications in a small laptop ten years ago are comparable to today’s mobile devices like the iPhone 3GS. As a result, embedded computer systems are also becoming increasingly pervasive. For instance, today’s cars include embedded systems to monitor the engine performance and can change the injection pump to increase the efficiency. Another field of embedded systems are set-top boxes that receive an encrypted signal of the telecommunication providers and convert it. Their implementation language C/C++ and Assembler were meant to be replaced by Java. With a Java VM running on such boxes the functionality of the device could be extended by loading new Java bytecode in the VM. Why not provide only software bytecode but additional hardware code to reconfigure the hardware if possible. But those systems are mostly built of ASICs (application-specific integrated circuits) which, once installed, cannot be reprogrammed but fulfill their intended tasks. A malfunction in one ASIC often results in a complete replacement of the faulty component. Also has the complexity of an ASIC bolstered the development time and cost.

The ASICs disability to embrace changes is promoting their counterpart, the FPGA chips. These chips can be reconfigured in their logic with an appropriate interface once deployed in a design. ASICs were traditionally preferred over FGPA because of their speed, lower power consumption, higher functionality. However, the improvements on the FPGA in the last years have almost closed this gap. In addition, development time and time-to-market could be decreased through the usage of FGPA. Maintenance can be done when an error is found in the implemented design since the FPGA fabric can always be reconfigured.

Sony has already delved into the use of reconfigurable hardware in mobile devices. Their VME (Virtual Mobile Engine) included in several products like music players and the PSP (Playstation Portable) decreases power consumptions through dynamic reconfiguration, [24].

Silicon Blue announced their lower-power FPGA chip iCE65 for Smart Phones. Their product is a 65-nm "ASIC-like" chip that is suitable for handheld devices due to its efficiency. Here the term "ASIC-like" refers to the benefits of an ASIC, low material costs, low power consumption and no NRE (Non-recurring
Introduction

This chip cannot be reconfigured in runtime after deployment but the technology made progress to provide a small FPGA chip running in mobile phones.

Other chips feature a hybrid approach where a hardcore processor such as the PowerPC is embedded into the FPGA’s logic fabric. Those chips are not required to emulate a softcore processor in their designs but can access the hardcore processor which performs better, is smaller and often has more features included.

With constant development in the field of FPGA they have become increasingly adaptive. Some of them support reconfiguration of only a part of the FPGA while the rest of the system can be kept operational. This feature allows to load required function logic into the FPGA during runtime. Unused logic can also be unloaded. With this in mind, the application logic can be implemented in swappable hardware tasks and probably increasing the efficiency compared to a complete software approach, see the work of Wolf[30] or Chiodo et al.[4].

1.1 Motivation

The development of embedded system led from small, low-resource hardware towards capable systems able to run resource consuming applications. Considering the performance increase, these devices are also outfitted with features such as wireless capabilities ranging from Bluetooth to UMTS, music- and video-player and more. Those capabilities are programmed into ASICs included in the device and only due to the fact that ASICs are small in size a device can contain such a variety of functionality.

As a matter of fact, mobile devices are more restricted by their small user interface compared to desktop-sized devices. A multi-window user interface is prohibitive due to the considerably lower screen resolutions of typical mobile devices. The interaction with the mobile device is done by a tiny keyboard integrated in the device or a touch-screen. As a result, the user is restricted to interact with one application at once. E.g., a user cannot look at a video streamed from storage drive and surf the Internet at the same time. In the same way, he is not be able to listen to music and make a phone call.

An interesting option would be if the focused application could reprogram an FPGA to run some application functionality completely on hardware. Hardware implemented functions would then gain an increased efficiency. The application can use the hardware functions to gain a performance increase. This either manifests in shorter response time or a higher throughput compared to the same work implemented in software. Another motivating example is extensibility. A decoder module can include new codecs for various music and video files. This offers application to include specific hardware code which is used to reconfigure the FPGA. Those will then support new logic functions.

The provided hardware module must be loaded in a controlled manner as an FPGA can contain other hardware modules supporting critical system functionality. Those modules cannot be replaced or overwritten on-demand by an
1.1 Motivation

In addition, the FPGA is space-constrained. There is a limited number of hardware modules which can be loaded, compared to the numbers of kernel modules that can be loaded into the kernel in a Linux system. The introduction of a managing layer which takes care of proper hardware module loading is thus unavoidable.

Summarized, the issues arising are the lifecycle management of such dynamic hardware modules. For software modules, there are already platforms which solve these issues. For instance, OSGi is a specification for such a lifecycle management system for Java. An OSGi framework is a module system for Java implementing a complete and dynamic component model. Applications can be loaded as so called bundles into the system and then extend the functionality of the running system. Although the earlier focus of OSGi concentrated on service gateways the actual usage is much wider. The OSGi specifications are now used in applications including mobile phones, PDAs, automobiles, and more.

It is the ability to handle dynamism in OSGi that make this approach promising. The FPGA is restricted to manage only one or at least a few hardware modules in it. This requires to swap hardware modules in and out of a running environment and inform the depending applications of it. OSGi implementations already provide such a lifecycle- and notification mechanism. The benefits in a Java-based approach is the architecture independent bytecode, the simplified memory management and automatic garbage collection which prevents memory leaks and a built-in threads library. The language has boundary checking to prevent the programmer from writing and reading out of an array. In addition, Java is object-oriented which enforces to structure the data and functions into classes. Java was historically intended to run on embedded devices but has more evolved to a language running on desktop-systems. Probably also, because Java does not provide a direct interface to underlying hardware and depending on a JVM Java applications are less efficient than their direct C implementations.

To run Java bytecode an FPGA has to provide an embedded processor whose architecture is either supported by a Java VM or a Java processor such as JOP (Java Optimized Processor)\cite{20} can be used. JOP aims to run the JVM directly as a hardware component on an FPGA chip. The major difference between those two approaches is the standard OS layer missing in a Java processor design. The reason to design a system relying on an embedded processor and an OS is the stable environment. Here a Linux system already provides a wide range of tools to control the hardware and the communication with it.

With those restrictions, the overlying OSGi framework can be extended to handle OSGi bundles with additional hardware data to reconfigure the FPGA included in the system. A possible approach is shown in this thesis.

Contributions presented in this thesis is the designed system presented in chapter 4 which has been assembled with the help of examples provided by Xilinx and some minor modifications. Concierge, an OSGi framework has minor modifications to support the FPGA Extender. He has been developed to manage new FPGA Bundles, an extended variation of standard OSGi bundles including a hardware module.
1.2 Outline

To give a full understanding of FPGA reconfiguration and the implementation design the thesis is organized as followed: The chapter 2 introduces the internals of an FPGA chip, through which interfaces the chip can be reconfigured, and how its configuration data is structured. It is presented how used *Early Access PR design flow* looks like and how he has evolved to the previous *Modular design flow*. Further an introduction to JNI and the OSGi framework is given. In chapter 3, related work is discussed. Chapter 4 delves into the concrete design decisions. It presents the system layout running on the FPGA and the single components. Additional the design flow involving planAhead is explained. Chapter 5 connects the OSGi model to FPGA. Chapter 6 presents the evaluation of the implementations and methods. Chapter 7 concludes the thesis and discusses opportunities for future work.
Chapter 2

Background

2.1 Introduction to FPGA

The concept of a module system that can alter the underlying hardware design during active state can not be applied to every custom FPGA board. Similar are the effects that the running operating system on the board has on the overall design. The next chapter thus presents and explains the mechanisms about how its internal configuration is written and rewritten and also the process how a Linux system is operating on an FPGA board.

2.1.1 FPGA Chip

The requirements of the thesis demand the use of an FPGA that is capable of synthesizing a PowerPC or better a hard-wired processor component on the chip layout. The XUPV2P is Xilinx University Program Virtex-II Pro Development System provides a board with an FPGA chip including two hard-wired PowerPC in it, thus makes it suitable for the following work.

Field-programmable gate array  FPGAs are integrated circuits which can be reconfigured by the customer or designer after manufacturing. This programmable device consists of three main parts: a set of programmable logic cells (PLC) or configuration logic blocks (CLB), a programmable interconnection network and a set of input and output cells around the device.

The actual implementation of a configuration logic block can vary and depends on the FPGA implementation. The CLBs are the actual basic logic unit in an FPGA. The CLBs of a Virtex-II Pro groups four slices together which each contain two actual basic blocks. A basic block consists of a LUT with 4 inputs and one output, a set of multiplexers, arithmetic logic and a storage element.

The LUT (Look-Up Table) is a group of memory cells which contain all the possible results of a given function for a given set of input values. It is used to store the configuration.

The FPGA can be seen as a 2-dimensional array with the smallest element a CLB where the system design can be seen as a set of modules. The modules are implementation of logic functions which occupy a certain amount of space. Modules are organized as rectangular areas which have to be positioned on the
array so that all module areas do not overlap and inter-communication timing constraints between all modules are met. The layout of the FPGA structure such as the arrangement of the logic blocks and the interconnection paradigm of the logic blocks is vendor-dependent. FPGAs can be classified in four such categories: symmetrical array, row-based, hierarchy-based and sea of gates. The Xilinx Virtex II Pro, for instance, is a symmetrical array. The other categories and more details about it can be found in [3]. The general routing structure is a net of vertical and horizontal wires which allows the routing inside the FPGA fabric. A symmetric design of the inter-connection network ties each CLB to a switch matrix to access this general routing structure. This makes the routing on the FPGA possible.

If the FPGA chip needs to communicate with off-chip modules there are I/O components located around the periphery of the device for this purpose. Those I/O are programmable just like the CLBs and can allow the design inside the FPGA to use a single interface pin as input, output or bidirectional gate. The Xilinx I/O components are called I/O block (IOB). For the actual pin-
2.1 Introduction to FGPA

The data to configure FPGA is called a bitstream. Bitstreams can be downloaded into the device via several configuration ports (JTAG and ICAP are the preferred approaches).

2.1.2 Configuration

The configuration process involves two parts. One aspect covers the bitstream data which is written to the FPGA, the other describes the devices that writes the bitstream to the FPGA.

Bitstream

An introduction to the Xilinx bitstream format is presented in [36]. In [37] the bitstream and reconfiguration procedure is explained in more detail. Generally, the bitstream is built from commands which prepare the writing of the configuration data into the device and raw configuration data which are mapped to the internal configuration memory.

A frame is the smallest granularity in which the FPGA allows configuration data to be read and written. A configuration frame is a sequence of bits that is one bit wide and spans the full height (column) of an FPGA. The IOBs at the top and bottom are also part of the configuration data and can be configured. A single column of CLBs contains multiple configuration frames. Though Xilinx offered with their JBits [7] implementation a way of altering the LUT of a CLB the Virtex II Pro devices are organized frames or column-slices. This is the smallest unit of a reconfiguration. Other boards like the Virtex 4 or Virtex 5 support a reconfiguration with the size of a complete tile no longer affecting a entire column.

The Frames again are grouped into larger units called columns. Of those exists several different types of columns:

<table>
<thead>
<tr>
<th>Column Type</th>
<th># of Frames</th>
<th># per Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>CLB</td>
<td>22</td>
<td>46 # of CLB columns</td>
</tr>
<tr>
<td>IOB</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Block RAM Interconnect</td>
<td>22</td>
<td>8 # of Block RAM columns</td>
</tr>
<tr>
<td>Block RAM Content</td>
<td>64</td>
<td>8 # of Block RAM columns</td>
</tr>
</tbody>
</table>

Table 2.1: Configuration Column Type

The center column of the device includes configuration for the four global clock pins. The most left and right column represent configuration for all IOBs on the edges of the device. The majority of the columns are CLB columns consisting of CLBs and the two IOBs above and below those CLBs. The block RAM as the remaining columns involve two types: one for content and the other for interconnection.

Each frame has a unique 32-bit address. The addressing is composed of a block
address (BA), a major address (MJA), a minor address (MNA) and a byte number. A specific column within a block is identified by the major address, while the minor address identifies a specific frame within a column. Only the Virtex-II Pro configuration logic uses the byte number. Virtex-II Pro configuration memory is divided into three independently addressable blocks: All GCLK, IOB, IOI and CLB configuration columns are contained in the Block Address 0, the Block address 1 contains all BRAM columns while Block Address 2 addresses all BRAM Interconnect columns.

<table>
<thead>
<tr>
<th>Column Type</th>
<th>GCLK</th>
<th>IOB</th>
<th>CLB</th>
<th>IOI</th>
<th>CLB</th>
<th>BRAM</th>
<th>BRAM INT</th>
<th>BRAM INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MJA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td>m</td>
<td>m</td>
</tr>
</tbody>
</table>

Figure 2.2: Column-Level (MJA) Configuration Memory Map

A sequence of configuration bits forms a configuration bitstream. The configuration bitstream is a set of configuration commands and configuration data. The basic layout of the Xilinx bitstream format is depicted in Figure 2.3

<table>
<thead>
<tr>
<th>CMD 1</th>
<th>data</th>
<th>CMD 2</th>
<th>data</th>
<th>CMD 3</th>
<th>data</th>
<th>CMD 4</th>
</tr>
</thead>
</table>

Figure 2.3: Bitstream example

Writing configuration data is done by issuing configuration commands to the desired interface (ICAP or JTAG) followed by the particular configuration data. The configuration data is organized in 32-bit words. The Virtex-II Pro configuration logic consists of a packet processor, a set of registers, and a global signals that are controlled by the configuration registers.

Via 32-bit registers (configuration registers) the configuration logic is accessed and controlled. The packet processor controls the incoming flow of configuration data from the configuration interface (SelectMAP, JTAG, ICAP) to the appropriate register. The device is controlled with a read and write operation, a configuration command is executed when it is read or written to the appropriate command register. The other aspects of the configuration is controlled by the registers. A full explanation is given in [37] or [33]. Commands are organized as packets with a command header word and optional data words. For a read or write operation the header word is first written in the appropriate command register. The command header format is shown in
2.1 Introduction to FGPA

<table>
<thead>
<tr>
<th>OP</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>01</td>
</tr>
<tr>
<td>Write</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2.2: Read and Write Operation

The three most left bits describe the header type. 001 is a standard type on which can follow at most 2047 words. An extension header (type 010) header is used if the data exceeds this value. The next two bits, defining the operation mode, are set accordingly to a read or write operation, see 2.2. The Register Address, defined in the next fourteen bits, holds the configuration register address, see 2.3. Bit 12 and 11 are reserved, and the last eleven bits define the word count of following the configuration data.

Figure 2.4.

In a running system, the packet controller waits for a synchronization word appearing on the configuration interface. Upon receiving it the controller starts interpreting any data. After synchronization a valid packet header, see 2.4, has to be sent to drive incoming data into the targeted configuration register until the word count specified by the packet header reaches zero. The packet headers pass through a 64-bit buffer before they are processed by the controller who interprets the commands in 32-bit words. Thus the buffer has to be flushed with NOOP commands. Every sequence of configuration commands must end with at least two 32-bit NOOPs.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th># Binary Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic Redundancy Check</td>
<td>CRC</td>
<td>0000</td>
</tr>
<tr>
<td>Frame Address</td>
<td>FAR</td>
<td>0001</td>
</tr>
<tr>
<td>Frame Data Input</td>
<td>FDRI</td>
<td>0010</td>
</tr>
<tr>
<td>Command</td>
<td>CMD</td>
<td>0100</td>
</tr>
<tr>
<td>Status</td>
<td>STAT</td>
<td>0111</td>
</tr>
<tr>
<td>Legacy output register</td>
<td>LOUT</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 2.3: Example of some Configuration Registers

The layout and addressing of frames in the Virtex-II Pro architecture 2.2 and the registers listed in table 2.3 are of particular interest when bitstream relocation is discussed. Thus the need to mention it shortly.

CRC Register (CRC) is usually checked after a write to the CRC register by comparing with a calculated cyclic redundancy check value using the
register data and address bits whenever configuration data is written to any register other than LOUT. This assures the integrity of configuration data by the 16-bit Cyclic Redundancy Check and protects from bitstream corruption.

Frame Address Register (FAR) addresses the configuration frame where an incoming configuration data packet should be located in the FPGA. Writes to the FDRI and reads from the FDRO act on the address specified in the FAR.

Frame Data Input Register (FDRI) and Frame Data Output Register (FDRO) is filled with the configuration data which has to be written to memory or read from. The addressed frame is specified in the FAR register.

Command Register (CMD) serves to instruct the configuration control logic to check for global signals and perform other configuration functions.

Status Register (STAT) indicates the value of numerous global signal, such as the status of the ICAP’s reconfiguration status or CRC error status.

Mode The XUPV2P board supports various ways to upload a configuration bitstream. On the VirtexII Pro devices the bitstream is first loaded into internal memory using one of the following modes:

- Slave-/Master-serial mode
- Slave-/Master-SelectMAP mode
- Boundary-Scan mode (IEEE 1532)
- Internal Configuration Access Port (ICAP)

The most usual case is to program the Virtex Board through an attached JTAG-Cable in boundary scan mode (the XUPV2P board can also be programed over an USB cable), but this is an externally driven configuration. For the goals of this thesis, the internal configuration access port is more interesting, because it enables the device to reconfigure its own chip layout. This makes the device reconfiguration independent of external programming devices.

The serial and selectMap mode are other supported modes in Virtex-II Pro families. The selectMap mode is able to read and write the device byte wise while the serial mode write one bit per clock cycle. The Master-/Slave- modes are only distinct in kind how the CCLK (Configuration Clock) pin is driven. In Slave-serial mode the FPGAs CCLK is driven by an external source while the master-serial mode drives the clock itself.

JTAG Widely used for IC debugging. While IC are not available to probe another way of debugging those had to be found. Besides debugging, another application of JTAG is allowing to program the device through this port (transferring data into non-volatile device memory)
ICAP  The Internal Configuration Access Port is an IP core that enables an embedded microprocessor to read and write the state of the FPGA configuration memory at run time, see [31] and [33]. This allows the system to dynamically load a bitstream into the FPGA to alter the functionality of implemented modules (the module region). The reconfiguration can be done by a read-modify-write mechanism in which the peripheral determines the configuration frames that must be modified, and then loads it into Block RAM. The configuration data in the Block RAM can now be modified, before the content of each frame gets written back. One frame is the atomic unit that can be written in one run by the ICAP.

Although a single CLB or flip-flop can be modified, the underlying mechanism of a Virtex-II Pro chip requires that the full column is read into Block RAM. This also means that other logic blocks can be modified that were not the previous intended. Unmodified configuration data written back to the FPGA does not affect the running system due the fact that the FPGA memory cells have glitchless transitions. The resource controlled by a bit does not change its output for a short duration (no short pulse occurs) if a configuration bit holds the same value before and after configuration. The states for CLB with exception of LUT RAM and SRL16 primitives [31].

This implies that a reconfigurable region does not have to span the full height of the chip area.

The ICAP device can be accessed by mapping the hardware based memory address into userspace. Once this is done the following registers are accessible, see 2.6.

The BRAM is a second memory location where the configuration data is stored before it gets written onto the device. It also holds a read frame from a Readback process. In the Size register the length of the bitstreams in words has to be stored before it is written. The BRAM Offset defines the location where the data is stored on the BRAM or has to be written to. RNC (short for Readback Not Configure) orders the ICAP either to write a frame or read configuration data from the device from or into the BRAM. The result of the process is written in the Status register.
<table>
<thead>
<tr>
<th>BRAM</th>
<th>block RAM holding 1 frame of configuration data at a time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Number of 8-bit bytes to transfer between ICAP and BRAM</td>
</tr>
<tr>
<td>BRAM Offset</td>
<td>Start address for data transfer into and out of BRAM</td>
</tr>
<tr>
<td>RNC</td>
<td>Readback not Configure register</td>
</tr>
<tr>
<td>Status</td>
<td>Status of ICAP and configuration/readbacks</td>
</tr>
</tbody>
</table>

Xilinx is constantly driving their developed FPGA technology so that their newest FPGAs are currently Virtex-6 chips. The Virtex-II Pro chip, soldered onto the XUPV2P board, has reference documents dating back to 2005 but is already viewed as a mature product by Xilinx no longer (fully) supported in EDK version 10.1. Also the IP cores have been constantly developed. E.g. there are three different versions of the PLB and IP cores compatible to one PLB are not with another. The ICAP core in EDK 10.1 does not seem to work at all with the XUPV2P board. In addition, the newer ICAP cores are only working for Virtex4 chips and later. EDK 10.1 supports the XUPV2P board with the help of peripheral repository files. The final design has been created and synthesized with the help of EDK/ISE version 9.2. This is also the latest version supporting the partial reconfiguration tools.

Naturally Xilinx provides in its EDK libraries to access the particular cores in an embedded system. Plain C programs running on such an embedded system often lacks a memory controller and hardware is directly addressed. When using the provided library functions in cooperation with Linux they have to be adapted to remap the core address into user memory. Then the mapped user address can be used to operate on the hardware.

### 2.2 Reconfigurable Designs

A board’s design is always designed to fulfill a specific purpose. To achieve this, it will consist of some I/O interfaces and internal modules which do system management and calculations. E.g. Figure 2.7 shows a board which has a network link for receiving and sending data, three internal module cores for processing the data and some I/O modules like monitor and keyboard.

The basic design can be improved in several ways. For instance, all modules cause energy consumption even when some of them are only used infrequently. However, energy consumption is often a driving factor in the design of mobile devices that are battery-powered. Loading such modules on-demand could save precious energy and thus increase the operating time of mobile devices. Modules fulfilling disjoint functionality could share the same chip space. Additional space for both the modules could be omitted and the used chip area could be shrunk so that the chip again becomes smaller and less power consuming. Maintenance of embedded system could be improved. Faulty implementations could be exchanged with a corrected version and even new functionality could be
2.2 Reconfigurable Designs

added. E.g. if the embedded system uses relies on a crypto algorithm that has become obsolete an new and better algorithm could be loaded into the system.

**Reconfiguration** The Virtex-II Pro board can always be reconfigured over an extern cable (such as JTAG or USB). After uploading the bitstream the I/O devices and other static cores may still be the same while actually only one module has been change. Reconfiguration means to entirely rewrite the design of the FPGA chip. The side-effects are that the reconfiguration-time is linear to the size of the bitstream to write and the board is shutdown thus interrupting the running processes.

![Sample PR design with three PRRs and three PRMs](image)

Partial Reconfiguration A first step to improve this process is to allow to rewrite only a part of the chip. Specific regions on the chip are marked as reconfigurable in which only a partial bitstream with new functionality is loaded. In order to allow the FPGA to reconfigure only a part of its chip, the FPGA fabric is partitioned into a static region, holding runtime essential modules like bus systems, and one or more partially reconfigurable regions (PRRs). This allows to minimize the reconfiguration-time as the bitstream is only a subset of the entire FPGA logic located in the PR region. Functional modules/tasks (reconfigurable modules or PRMs) can then be mapped to a single PRR and mutually independent PRMs can share a PR region. This can reduce the required FPGA area for a given application which changes its task over time. Several bitstreams data can made available either on an internal PROM or Flash Card because partial bitstream’s size is now smaller compared to the entire bitstream. However, partial reconfiguration does not mean that the board continues operating during reconfiguration. Depending on the hardware it can be the case that reconfiguration requires the board to be in an inactive state.

Dynamic Partial Reconfiguration If the chips basic logic can be reconfigured during runtime without interruption of the system flow then the chip supports dynamic partial reconfiguration. This is one of the interesting features of the Xilinx VirtexII Pro chip. While the base microprocessor is operational
the module part is being reconfigured to support a new function. Partial reconfiguration is therefore useful to applications requiring the loading of different designs into the same area of the device. This allows to flexible exchange a defined parts of a design without having to either reset or completely reconfigure the entire device. This capability offers new application areas become possible:

- In-the-field hardware upgrades and updates to remote sites
- Runtime reconfiguration
- Adaptive hardware algorithms
- continuous service applications
- reduced device count
- reduced power consumption
- more efficient use of available board space

Xilinx offers some application notes about their specific design flow for partial reconfiguration. Although their application notes gets rewritten from time to time and early design notes appear much of the basics are explained in one of the first versions of [38].

XAPP290 note [38] explains the two design flows to reconfigure the FPGA structure of which the later Early Access Design Flow bases on Modular Design Flow.

### 2.2.1 Modular Design Flow

Partial reconfiguration has been evolved from the modular design flow. Initially this was intended to allow several engineers to cooperatively work on the same project and not to support partial reconfiguration. At first the project leader identifies all components in the whole system design, estimates the amount of resources that will be consumed, layouts the top-level module defining each region for every component and lets the single parts be developed by the different engineers. Once the single components are developed they are integrated into the whole design to build a functioning circuit. The modular design flow consists of 4 steps: design entry and synthesis, initial budgeting, active implementation and assembly.

**Design Entry and Synthesis**  This step requires to develop the modules using a hardware description language (HDL) which are then synthesized into a device netlist format (.NGC file). The synthesized netlists are used to estimate the amount of needed resources.

Further on the the implementation of the netlist consists a sequence of three steps

**translate**: combine all the input netlists and the constraints to a logic design file. This is done with ngdbuild and its result is stored in a .NGD (Native Generic Database) file.
2.2 Reconfigurable Designs

**map:** the modules are divided in sub blocks to be placed in logic blocks on the chip. It maps the logic elements into the targeted FPGA elements. A .NCD (Native Circuit Description) is then generated which is a physical representation of the design.

**par:** place and route process places the sub blocks from the map process into logic blocks according to the given constraints and connects the logic blocks.

**Initial Budgeting** Every module on the top-level is assigned with constraints (pin locations, area constraints for each module, timing constraints, etc.) which results in topX.ucf (user constraint file). The synthesized top-level netlist, e.g. top.ngc (or system.ngc in cooperation with EDK) is used together with the constraint file (topX.ucf) for further steps.

For reconfigurable modules special constraints are to be met:

- the height should always be the full height of the device. Due to the glitchless reconfiguration feature of the Virtex-II Pro board, this property has been stated as obsolete in [35].
- the module’s width must always be a multiple of 4 slices (e.g. 4, 8, 12)
- once the design is fixed the module boundaries cannot be moved or resized
- communication between modules or static region must be routed through bus macros
- The implementation has to be designed in a way such that the static portion do not rely on the state of the module under reconfiguration. During the reconfiguration process the design has to be in an always well defined state. An explicit handshaking (module ready/not ready) logic may be required.

The outcome of the initial budgeting phase is a constraint file .ucf which contains all placement and timing constraints. Each module is using this file to met the required constraints, additionally to any module-specific module constraints.

**Implementing Active Modules** Up to this point the partial reconfiguration design has been design, synthesized, floorplaned and constrained. In this phase all the separate modules will be implemented but always in the context of the top-level logic and constraints. For every reconfigurable module a bitstream is be generated.

**Module Assembling** The final stage is the process of fusing each of the implemented modules into a complete FPGA design. The achieved placements and routings during the active implementation phase is preserved. The resulting full design can be used to generate the bitstream.

2.2.2 Early Access Design Flow

With the upcoming interest in partially reconfigurable designs, Xilinx updated and enhanced the Modular Design Flow resulting in the EA Design Flow, thus
the general implementation work stays the same as with the Modular Design Flow. Required in this design flow are the provided implementation tools of Xilinx. Those are modified map, par and library files for partial reconfiguration and work in combination with the ISE release, the implementation tools must be installed on top of it.

Design Entry and Synthesis  There are not much difference to the Modular Design. A first step includes the description of the design and then synthesizing it. The EA Design Flow can be coupled with the EDK which will supply system.vhd, the top-level base design. Partially reconfiguration requires a hierarchical design approach and this as to be followed during the HDL coding process.

Contained in the top-level modules are

1. all global logic, such as I/Os, global clocks, and DCMs
2. all multiple base-design modules and partially reconfigurable modules instantiated as black-boxes with only ports and port direction specified,
3. signals connecting modules to each other and to I/O pins.

After the HDL design is synthesized, the next step is to design the constraints and write the .ucf constraint file. In special, PR design must be constrained with AREA GROUP, AREA GROUP RANGE, MODE and LOC constraints.

AREA GROUP Constraints (AG) prevents logic in the base design from being merged with the logic in the PR modules. AG constraints have to be set for every PR module, by default modules not belonging to any AG are assigned to the static design.

```
INST "prm_1" AREA_GROUP = "AG_PRegionA";
```

AREA GROUP RANGE Constraints (AG range) must be defined for each reconfigurable region to set the placement and shape. Two ranges are defined for one region, one for the slice range and one for the block RAM. There are not many requirements when defining PR regions except for the slice range cannot fall between slices in a CLB. Thus the (minX, minY)-coordinates are always EVEN and the (maxX, maxY)-coordinates are always ODD of an AG range.

```
AREA_GROUP "AG_PRegionA" RANGE =
SLICE_((minX)(minY)):SLICE_((maxX)(maxY));
AREA_GROUP "AG_PRegionA" RANGE =
RAMB16_((minX)(minY)):RAMB16_((maxX)(maxY));
```

MODE Constraints prevents ngdbuild from failing with unexpanded block errors during base and PR module implementation and must be defined for all reconfigurable regions.

```
AREA_GROUP "AG_PRegionA" MODE = RECONFIG;
```

LOC (location) Constraints are used in constrained pins, clocking primitives (DCMs, BUFGs, etc.) and bus macros.
2.2 Reconfigurable Designs

As identifying the particular slices for a reconfigurable region is cumbersome, a more save and facilitated way is to use planAhead, a visual editor to design and layout system and module constraints.

**Bus Macro** If the router does not have to meet any communication constraints between two modules the signal will much likely cross the module boundaries where the best timing can be afforded. The routing can be different for every module implementation. Hence, bus macros are used as fixed data paths for signals going between modules. They provide the means of locking the routing between reconfigurable modules and the static part, making the modules' pin compatible with the base design. (GNR, VNC and global signals are handled automatically by the tools in a way transparent to the user). Beneath the feature to lock the routing path through a bus macro the signal transmitting itself can be disabled or enabled on purpose. Signal transmitting has to be disabled while reconfiguration, and enabled after, to avoid bus congestion.

A bus macro designed for a Virtex-II Pro architecture has a width of 8 bits, in addition the direction of a bus macro has to be considered as they are directed. Bus macros can be categorized by their direction, their clock behavior (synchronously with the data or unlocked as the old bus macros) and their spanning width (how many CLB they can bridge), see [35].

device

- xc2vp - for a Virtex-II Pro,
- xc2v - for a Virtex-II,
- xe4v - for a Virtex-4 FPGA,

direction

- r2l - right-to-left
- l2r - left-to-right
- b2t - bottom-to-top (Virtex-4 only)
- t2b - top-to-bottom (Virtex-4 only)

synchronicity

- sync - synchronous or
- async - asynchronous

width

- wide - spanning four CLBs or
- narrow - reaching over two CLBs
To have a module communicating through a 32 bit wide fixed bus, 4 bus macros of 8 bit width have to be placed on the module boundaries. Each of the bus macros taking care of 8 bits of the bus. This only serves one direction, another 4 bus macros have to be used to lock the other 32 bits of the other direction.

The bus macros are built out of basic logic blocks, fixed routing paths between CLBs. This bases on conventional logic an should not introduce any additional latency that has to be especially considered. Though the final design can and should be checked with e.g. DRC (Design Rule Checks) provided with Xilinx design tools.

**Initial Budgeting**  Initial budgeting as well as the activation phase is not much different to the *Modular Design*. Area constraints can be set more freely and the bus macros have to be deployed concerning the type and direction.

- the bus macro has to be locked on a CLB’s X- and Y-coordinate divisible by two.
- each endpoints of a bus macro has to lie in a different area.

After the the system has been synthesized the system.ngc file is being translated corresponding to the constraint file, mapped and placed and routed.

```
ngdbuild -uc top.ucf -modular initial system.ngc

map system.ngd

par -w system.ncd system_base_routed.ncd
```

The output of this process is a fully routed netlist of the static design and another file, *static.used*, containing the list of routes of the static design crossing the modules boundaries. This occupied routes cannot be taken into account of the modules netlist and is taken as input for the placement and routing later on.

**Activation**  The partially reconfigurable modules are implemented in this phase. Although signals passing in and out of the module’s region are routed through a bus macros other signals not interfering with internal logic of the
2.2 Reconfigurable Designs

module region can still be routed through the module’s region. Bitstreams designed for this particular PR regions have to consider the static routes crossing the module logic. Each module uses the previously generated static.used renamed as arcs.exclude to take care of wires routed through the PR region. The modules are then run through the known sequence of ngdbuild, map and par.

```
ngdbuild -uc top.ucf -modular modular module -active
prm_al system.ngc

map system.ngd

par -w system.ncd prm_al_routed.ncd
```

Final Assembly This stage aims at assembling the static and the reconfigurable modules into the top-level design. Xilinx provides two scripts pr_verifydesign and pr_assemble to facilitate the work. These are two Bash scripts wrapping and controlling the process flow of their Perl counterparts.

```
pr_verifydesign system_base_routed.ncd prm_al_routed.ncd
pr_assemble system_base_routed.ncd prm_al_routed.ncd
```

PR_verifydesign.pl The purpose of this script is to generate all of the required bitstream files defined by the user input and to validate the bitstreams of the reconfiguration modules are writing all of the necessary frames for a successful reconfiguration. The process includes:

Step1: a bitstream is created for the static design

```
bitgen system_base_routed.ncd system_base_routed.bit
```

Step2: every reconfigurable module is then merged with the static design to generate the full design netlist and the partial module netlist. The partial module netlist is then compiled into a bitstream with

```
PR_mergedesign system_base_routed.ncd prm_al_routed.ncd

bitgen prm_al_routed_partial.ncd prm_al_routed_partial.bit
   -g ActiveReconfig:Yes
```

Step3: Finally, the partial bitstream is checked against the static bitstream to verify the frame boundaries of the reconfigurable module are correct. The delta is gained with

```
bitgen -r system_base_routed.bit prm_al_routed_partial.ncd
```

PR_assemble.pl At last a design and initial bitstream has to be generated with this script. This involves only the static part of the system and the initial modules which have to be stored in the PR regions. The static design is merged in a row with all the PRMs with help of PR_mergedesign and and a final bitstream is generated with bitgen as well as the blanking bitstreams. The output of those two scripts are a final full bitstream, the partial bitstream for the particular module and a blanking bitstream which is able to wipe the PR region of its logic. Note that, accessing a blanked PR region with Linux system calls will effectively freeze the system.
2.2.3 Bitstream properties

A system design intended to offer a dynamic runtime PRM placement requires multiple PR regions where the PRMs can be loaded into. For a PRM, the EA PR design flow generates a single bitstream for every PR region available in the design. For the sample design depicted in Figure 2.7 the EA PR design flow would thus generate 9 different bitstreams identical in its function logic. But generated bitstreams cannot simply be placed in a different PR region than intentionally created though identical in its function.

The redundant bitstreams differ in their routing. Already mentioned the static_used file that restricts the PRM not to use certain wires routed through a PR region, those are the unconstrained clock signal routing. The EA PR design flow specifies not to route clock signals through bus macros and is automatically handled by the EA PR tools transparent to the user. Besides the clock routing other signal will cause different routing for different PR regions due to the so called I/O-Pin-Problematic. The XUPV2P board in particular has peripheral components accessible to the FPGA fabric through the I/O pins. The I/O pins connects peripherals such as Video, RAM, Audio, SystemACE, Reset or Ground. Bitstream relocation thus cannot be done in a feasible manner for an XUPV2P board. Though other design take a 1-dimensional design layout into account where two opposed edges are not used for I/O pins. This reduces complex routing and offers a unique scheme for every single PR region, see chapter 3.

2.3 Accessing the Hardware with JNI

Normally, an embedded system is be programmed as a standalone system without the controlling processes of an OS which enforces memory access restriction to ensure system stability. The hardware is accessed directly with the given address in the Address Map.

If Linux manages the hardware some memory protections are mostly implemented and virtual memory is available. This means we are no longer able to directly address the hardware by its memory address but have to remap the hardware address to some user space memory. This is done with mmap:

```c
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

The communication with the implemented IP core does not differ from writing or reading to and from memory. Further more the hardware can communicate over direct memory access (DMA), generate interrupts, read and write memory register. More sophisticated techniques like DMA and interrupts are require device driver support.

JNI Java as a safe and platform independent language avoids access to low-level I/O devices or direct memory access. This is no drawback in standard Java application where low-level I/O is rarely a concern. But in an embedded system where the hardware is tightly coupled to the software application it is essential to access and control the underlying hardware. Especially in an application that
should bridge the gap between OSGi and an FPGA. Here comes Java Native Interfaces in handy. JNI are used to bridge the gap between a Java virtual machine and applications and libraries written in C, C++ or Assembler. With JNI, code specific to a given hardware or operating system platform can be called and this native code can call back the JVM. In this work, JNI is used to access C libraries that help accessing the underlying hardware. Java is not suitable to this task as its code has to be as much machine-independent as possible. The use of JNI to access underlying C libraries and hardware is the fastest way for a proof of concept. It introduces a few safety and design issues because of the handling of allocated memory, the pointer management and the access of a single hardware instance. Safety can not be guaranteed when more than one hardware instances are created mapping always the same hardware memory region. In addition, the object oriented concept of Java gets lost when invocation of C functions are needed. It also introduces a certain overhead when calling native functions and converting Java structures to C readable types. JNI can map simple primitive types like byte, int, double etc. to C types. Arrays and Classes have to be converted through provided functions into C arrays and structures, too.

If the C code has to deal with Java arrays of any sort there are several ways to convert those. The provided functions copy the array data in a C buffer. If an array is expected as return value by the Java code a new array has to be created in the C code for the data to be stored into. A modified C buffer does not affect the original Java array as it could be copied by the VM (it is not determine when an array gets copied).

A more C like approach is a ByteBuffer that is backed by a memory region which can directly be accessed within the C code. Altering this memory region also alters the ByteBuffer.

To overcome these implications Hardware Objects for Java are suggested in [21]. How to integrate and react on hardware interrupts is described in [11]. Both of these presented solution imply changes on the JVM.

2.4 OSGi

OSGi [15][25] is a Java framework which provides a managed, secure and general-purpose service platform supporting the deployment of extensible applications known as bundles. The framework can load bundles at runtime when they are needed and unload them when they are no longer required. Bundles seen as applications can publish own services available to other bundles in the framework. This is achieved by a detailed management between bundles and the provided services. The Framework itself can be layered down to a

- Module layer
- Life Cycle layer
- Services layer
- Security layer

The module layer is responsible for bundle management providing a generic and standardized solution for Java modularization. The life cycle layer manages the
relations between bundle and framework, and provides a notification system of bundle and framework events. And as well important is the services layer defining a dynamic collaborative model highly integrated with the Life Cycle layer. A hardware implemented service will be inserted into the system as a service offered by a bundle. The security layer is not be touched by any modification of the framework or new bundles because it is a non-crucial part of the system. The OSGi system is an implementation of at least the specification release version 4.

There are free available implementation of OSGi frameworks like Equinox, Knopplerfish or Apache Felix. The work presented in this thesis bases on the R4 implementation of Concierge [17] developed at the ETH.

Bundles and BundleListeners According to the OSGi specification an OSGi bundle is a common Jar file on the filesystem which contains resources and additional meta-data. The resources may be of any file type i.e classes for the Java programming language, as well as other data such as HTML files, icons, etc. The manifest file MANIFEST.MF, included in the bundle, contains additional descriptions of the bundle content. This file uses specific headers to specify information which the framework needs to install and activate a bundle correctly, e.g., dependencies to other bundles or resources. As the framework must ignore unrecognized manifest headers the manifest can be extended with own manifest-headers providing additional information. Hence, the manifest can be extended with non-OSGi conform header information pointing to extra resources contained in the bundle.

For further explanation some of the more important headers are described

**Bundle-Activator** specifies the bundle’s activator class used to start and stop the bundle. The framework creates an instance of the given activator class and register it in the framework.

**Bundle-NativeCode** contains all the names of the native libraries in the bundle. Those libraries are to be loaded by the Bundle Classloader. The namespace of the native library is coupled with the Bundle Classloader.
**Bundle-SymbolicName** specifies a non-localizable name for this bundle. The bundle symbolic name together with a version must identify a unique bundle.

**Export-Package** lists all the Java packages in the bundle that are exported to the framework and can be accessed from other bundles. This offers the possibility of making the bundle implementation dependent exported classes of other bundles.

**Import-Package** defines packages exported from other bundles and imported into the bundles own namespace. If the bundle depends on an exported class of another bundle it must be imported into its own namespace.

Bundles can implement the activator interface and with it the methods `start(BundleContext context)` and `stop(BundleContext context)`. Those are called whenever the framework loads such a particular bundle or stops it. Upon its activation a bundle receives the bundle context. The bundle context interface methods can roughly be divided in the following categories:

- **Information** - access to framework specific information
- **Life Cycle** - the bundle can install other bundles into the system
- **Service Registry** - a bundle can register a service

The framework keeps an overview over the known bundle in the system and its state. A bundle’s state can be:

- **installed** - the bundle has been successfully brought into the system. No header errors.
- **resolved** - the bundles dependencies have been resolved, all Java classes are available, and it is ready to be started or has been stopped.
- **starting** - the bundle is currently starting meaning the Bundle Activator start method has been called but no yet returned.
- **active** - the bundle is running, start method has returned
- **stopping** - the bundle is being stopped and the Bundle Activator stop method has not yet returned.
- **uninstalled** - the bundle has been uninstalled.

The lifecycle of each bundle in the framework can be observed by a bundle listener. With the help of the framework and an event system, appropriate notifications are made to each listener registered in the framework. A bundle listener registers itself by implementing the BundleListener interface with its method `bundleChanged(BundleEvent event)`. The bundle event reports the current state.
Loading Native Code Libraries  Whenever a class loaded by its bundle’s classloader attempts to load a native library, by calling System.loadLibrary, the bundle classloader must invoke the findLibrary method to return the file path in which the framework has made the requested native library available. The name of the native library must be provided in OS independent form, e.g., FOO instead of libFOO.so or FOO.dll. To map the given library name to the OS specific name the mapLibraryName method from the Java VM is used, e.g., mapping FOO to libFOO.so for a Linux system.

There exist restrictions on loading native libraries due to the nature of class loaders. A fact to consider is that native libraries are unloaded only when the class loader which loaded the class has been garbage collected. This means any uninstalled or updated native libraries of bundles reside in memory until the corresponding classloader has been gc’ed. Furthermore, the garbage collection does not happen until all references to objects in the bundle have been gc’ed, and all bundles importing packages from the updated or uninstalled bundle are refreshed.

Service and ServiceListeners  The OSGi framework defines a dynamic collaborative model which is highly integrated with the Life Cycle Layer: the Service Layer. Bundles can register a conventional Java object as a service under one or more Java interface with the service registry. Additionally, a set of properties (keywords) can be passed along for the registered service to give other bundles the possibility to look for suitable services. A standard approach is to publish, find and bind the service using the whiteboard pattern. After a service has been registered other bundles can search for them or they receive a notification from the framework when the service registration state changes, e.g., from resolved to installed.
A brief overview of the service system and their parts follows:

service: the Java object which has been registered within service registry under one or more interfaces. The service can be retrieved and used by bundles.

service reference: a reference to a service. This provides access to the service properties but not to the actual object itself. The service object must be acquired through a bundle’s Bundle Context with the help of a service reference. The service reference actually contains the service properties.

service registration: when a service has been registered the service registration is returned as a receipt. It is be used to update the service properties and to unregister the service explicitly.

service factory: with this facility the registering bundle can customize the returned service object requested by a bundle. The framework routes every invocation of the `getService` and `ungetService` method to the corresponding service factory first. Especially, the framework is caching factored services per bundle. If requested, every bundle receives the same instance of the service.

The registered service object is owned by, and runs within, a bundle. Normally a service is registered by the bundle itself but since OSGi Release 4 it is possible to access a foreign bundle context and register services under their context. This requires additional caution as the foreign bundle may not handle the service un-registration if the bundle itself is uninstalled. Thus the registering bundle has to observe the foreign bundle’s lifecycle and should unregister the related service accordingly.

Services are not directly obtained but requested with the help of a service reference. A bundle does not need to get a service object itself but rather its properties. Thus a service reference avoids creating unnecessary dynamic service dependencies between bundles. The ServiceReference object will store the metadata to describe properties of the service object. This information can be used to select the service best adapted to the task. The properties can be applied to a service at its registration time when calling `BundleContext.registerService(String, Object, Dictionary)` or later by using the method `ServiceRegistration.setProperties(Dictionary)`. A ServiceReference object can be passed on to other bundles and stored without the implication of such dependencies. Once a bundle needs the service object itself it obtains the service object by passing the Service Reference to `BundleContext.getService(ServiceReference)`. The returned object is either the service instance implementing the interface as defined by the objectClass property or null if the underlying service object has been unregistered. Null can be returned if the ServiceReference of the requested service has been unregistered (e.g., in another bundle).

If the service object does implement the ServiceFactory and the bundle context usage count of the service is one, the object is cast to a ServiceFactory object. Then the `getService` method of the ServiceFactory is invoked to create a customized service object for the calling bundle which is returned. Otherwise, a cached copy of this customized object is returned.
The Service Factory is an interface defining the two methods `getService(Bundle bundle, ServiceRegistration registration)` and `ungetService(Bundle bundle, ServiceRegistration registration, Object service)`. If, instead of a plain service object, a service implementing the ServiceFactory is registered, the framework must reroute the `BundleContext.getService` call to the Service Factory method. The call to `BundleContext.ungetService(ServiceReference)` must also be routed to this method by the Framework. This way the ServiceFactory object can release the service object created previously. ServiceFactory object may help to manage bundle dependencies that are not explicitly handled by the framework itself. Invoking the ServiceFactory the service can be notified when a bundle gets or releases a service. Resources associated with the service can be allocated or released.

To release a service `ungetService(ServiceReference)` is invoked on the BundleContext. This means that the usage count of the used service objects in the service reference is decreased. Once the providing bundle is uninstalled the service has to be unregistered. The ServiceReference interface defines with the method `unregister()` to remove the service from the framework's service registry. Only the bundle holding the service reference is able to unregister said service. Thus passing of the reference has to be done with great care not to inflict the service management.

**Service Listener** Whenever a service gets registered, modified or unregistered in the system a notification is sent to every service listener implementing the ServiceListener interface.

Possible events caused by services can be:

- **registered** this is delivered synchronously right after the service has been registered with the framework.
- **modified** this is delivered synchronously whenever the service properties have been modified.
- **modified_endmatch** this is delivered synchronously when the property has been modified that the listener is filtering and no long longer matches.
- **unregistering** this is delivered synchronously before the service has completed unregistering. The service object is thus still valid and a bundle that receives this event has to release every reference it is holding to this service before this method returns.

**Extender Pattern** The extender pattern is an alternate approach to the whiteboard pattern to handle incoming bundles. Service have often been registered by the own bundle when they have been resolved and activated. Other bundles used a service tracker to listened for newly added or changed services. The extender pattern reverse the whiteboard pattern and is used to handle newly or already registered bundles.

An extender bundle implements a synchronous BundleListener. This includes to overwrite the `bundleChanged(BundleEvent event)` method and register the
bundle listener to the BundleContext. The bundle itself can implement the BundleListener interface and add the own reference to the bundle listeners.

![Figure 2.11: Extender Pattern](image)

The extender then gets notified about bundle state changes and can act on the activation of a newly installed bundle. Now the extender checks for specific criteria. With the BundleEvent the extender gets access to the BundleContext and can check for specific method handlers or headers in the manifest file. The extender pattern allows bundles to extend the functionality in a specific domain.
Chapter 3

Related Work

Considering the loading of a module once in hardware and once in software, in software additional modules can be loaded almost indefinitely due to effective memory management and will only fail if there is not enough memory space to load the code into it. If an additional component gets loaded onto the FPGA, in contrast, there is only limited place to put the component into. Additionally the FPGA layout is not as dynamic as a software module structure. Chip constraints have to be met such that at runtime configuration the chip is not destroyed.

The tutorial of Hübner et al.\cite{8} lay out the overall benefit the partial and dynamic reconfiguration can provide. The chip size can be reduced due the fact that not used component are stored in external memory and brought into the chip if required. This again will lessen the power consumption as the not integrated components do not allocate memory. Their system design is divided in a static area and three PR regions (slots) which hold the PR module. However, mentioned is that separate bitstreams have to be generated. Furthermore the designer of such a system must be sure that there is no signal lines crossing the border of a module during reconfiguration, since such a signal may cause a malfunction or a short-circuit, which destroys the FPGA. They propose the design and use of a bus macro in one of their other works in \cite{9}.

Ullmann et al.\cite{26} present a complete approach to a module based architecture is presented. Their implementation aims at the field of automotive control devices. Todays automobile classes contains up to 100 control devices which becomes sooner obsolete and the product life cycle decreases from 5 to 2 years. The flexibility adaptivity of reconfigurable devices could increase the product life cycle and reduce the cost and risk for development and later maintenance. On the memory structure their partially dynamically reconfigurable structure consists of separated slots for the function module which are connected to a bus structure. A bus macro \cite{1} is used to ensure the physical separation between the slots and connect them the signal lines between the functional blocks and the arbitration/ runtime system module. Further on they describe all the necessities that handles the communication between the function blocks and how loading and unloading is been done. A important role plays the reconfiguration unit that controls the reload process of requested function into the FPGA. It
processes a request to load a function module in four steps. First the recon-
figuration unit look for a free slot where to place the block. If no free slots
are available idle blocks have to be identified and can be unloaded to use their
place. Second upon selecting the free slot a state backup is initialized. Third
the save state information is saved and reconfiguration is initialized. At least
the state information is sent.

A high-level approach to use the Xilinx FPGA reconfiguration ability is ex-
plained in the work of Williams et al.[29]. Their field of research bases on a
modular platform for RSoC called Egret. The philosophy behind Egret is that
complex systems can and should be designed by stacking needed parts together.
The specification of this assembled hardware module stack is given to a software
tool that constructs the appropriate FPGA configuration, as well as software
infrastructure such as device drivers.

An interesting twist in their approach is the usage of the built-in ICAP de-
vice. The system builds atop a Microblaze system architecture and uClinux
as the OS. They present a first ICAP driver implementation beneath the now
included driver in the Xilinx Kernel repository. Their simple driver implements
the functions read(), write() and ioctl() providing a simple handling of the ICAP
resource and allow to load a bitstream as follows:

```
$ cat partial.bit > /dev/icap
```

Apparently they left the crucial parts of system design, managing reconfigurable
module regions and show some possibilities such an ICAP device offers. Apart
from that it is mentioned that the bitstreams remaining different from device
to device will still be a challenge to overcome. A first glimpse on the non-
relocatability.

[8][26] and [29] focused on the use of high-level approaches of reconfiguration.
They offer an overview of possible applications, design considerations and re-
strictions, and as well an introduction of the ICAP in an OS based environment.
However, to approach bitstream relocation in managed hardware systems the
followed work offers some detailed informations.

J. Seixas et al.[23] point out the complexity to cope with when the dynamic and
partial reconfigurable platform is implemented together with all the control,
such as a soft core microprocessor, on the same FPGA device. Their approach
is to divide the design and build a hybrid platform, based on two different
platforms: control and reconfiguration. The master platform is a Stratix-II ar-
chitecture responsible for controlling and sending of the configuration data to
the slave platform which is the Virtex-II device. The master platform is con-
trolled by a Nios II soft core microprocessor while the Virtex-II FPGA device is
divided into two reconfigurable areas. On the master platform runs the uClinux
embedded operating system due to its device drivers to perform complex dy-
namic reconfiguration on Virtex-II and to manage the board resources, such as
file access, host communication, etc. The host platform send its configuration
data (bitstreams) to the Virtex-II platform via the 32-bit Avalon bus, there a
IPSelectMap device driver allows the correct access to the slave platform.
Although the dedicated Virtex-II device could possibly support bitstream re-
location the paper does not state any method to do so. Only the fact that
modules have implemented with the help of the Xilinx modular design flow.

Walder et al.[28][27] present a similar approach but re-factored a prototype board assembling the controlling chip and the reconfigurable chip on the same board. The XF-Board Prototype has two Xilinx Virtex-II for operation. A C-FPGA (CPU FPGA type XC2V1000) for a Microblaze architecture connected with a second R-FPGA (reconfigurable FPGA type XC2V3000) on which the hardware tasks can be loaded. The R-FPGA layout is a 1-dimensional resource model dividing the space in several columns making one of them module slots.

![Figure 3.1: R-FPGA 1D resource model](image)

Memory and I/O connected to the pins at the top and bottom edges of the FPGA cannot be efficiently made available, as the corresponding FPGA I/O blocks undergo a reconfiguration whenever a new task is loaded. Thus Memory and I/O pins are only allocated at the left and right edge of the FPGA chip. Static regions at the side are reserved for Operating system critical processes. Communication with the user tasks are routed through this logic. The actual user tasks are swapped in and out of the reconfigurable frames and communicate through the user task communication bus. This design actually supports bitstream relocation due to the 1-dimensional design layout of the R-FPGA.

A first introduction how bitstream relocation can actually be accomplished is presented in Kalte et al.[10]. A component, the REPLICA2PRO filter, takes care of relocating modules. This is done by manipulating the FAR (Frame addresses) in the corresponding partial bitstream. Before the configuration manager, responsible for loading bitstreams, sends the bitstream to the configuration memory it passed the filter where the FAR is rewritten. The manipulations by the filter are done during the standard configuration thus avoiding extra time overhead.

The reconfiguration again relies on a horizontal communication infrastructure on which the task can be freely relocated.

The work of Flynn et al.[6] exposes the problems in bitstream relocation when using Xilinx EA PR design flow. The transparently handled and unconstrained clock signal causes this inoperability. By modifying the EA PR design flow to route the clock signal through bus macros produced arbitrarily relocatable bitstreams. But routing the clock signal through bus macros implies the signal to be routed through CLBs which are not intended for clock signals. Undesirable
side effects such as higher clock skew, timing inconsistencies and lower PRM operating frequency can occur. To circumvent this effects dedicated clock resources are used rather than bus macros. To relocate the produced bitstreams the FAR (Frame Address) is modified and the CRC has to be dealt with. Because bitstream corruption is currently no concern it has been disabled.

The work of Sedcole et al.\cite{22} focuses again on modular dynamic reconfiguration in Virtex-II and Virtex-4 FPGA devices. \textit{direct dynamic reconfiguration} is their term to describe the Xilinx EA PR design flow. Modules are directly written in the PR region and a separate PRM has to be generated for every PRR. This is mentioned as short introduction to reconfiguration design and a new method is presented, the \textit{merge dynamic reconfiguration}. A key idea is to reserve static routes through the PR region. As an example, it is stated that the Virtex routing architecture consisting of horizontal and vertical channels has 24 long lines and 120 hex lines as well as other more local routing resources. Within modules the routing is done via short lines whereas the signals passing the PRR are best routed on long lines. Thus by allocating 100\% of the long lines and 20\% of the short lines for every PRR the layout would look the same when generating a bitstream PRM.

In their current design flow the ISE par tools could not be provided with the necessary constraints of this wire reservations. Therefore a post-par re-routing step is performed on both the static and module designs. A custom tool has been implemented and used to generate routing constraints on a tile-by-tile basis. The way the reconfiguration is done is by reading back the current configuration, modifying it with information from the partial bitstream and the writing back. The new bitstream information is applied using a XOR function, thus not altering bits that are not in the module boundaries.

To show that implemented bitstream were relocatable, a design is implemented with two PR regions, one on the upper half the other on the lower half, aligned on the same frame boundaries. By shifting the all the frame bits of the module partial bitstream by a given number of CLB rows it is possible to move one bitstream between these two PR regions. This seems only to be one of a special case in bitstream relocation since no horizontal relocation is described in this work.

XPART (Xilinx Partial Reconfiguration Toolkit)\cite{2} derived from the JBits API \cite{7} would provide bitstream relocation for partial bitstreams. The XPART API is just another interface sitting on top of the ICAP. It abstracts the bitstream details providing seemingly random access to select FPGA resources. On the fly resource modification could be done through getCLBBits() and setCLBBits() methods. To deal with relocatable modules the two methods \textit{setCLBModule()} and \textit{copyCLBModule()} can be used. However, XPART was never released.

There exist a lot of approaches to achieve dynamic reconfiguration at the simplest level. To generate an explicit bitstream for every PRR seems the simplest solution. If no second FPGA chip can be used as a dedicated module storage \cite{28} \cite{23} one has to deal with static routes crossing module boundaries. And those routes give every module a unique layout which affect the corresponding bitstream.
Bitstream relocation can be achieved as presented in [10] [22][6] but no particular framework is available leaving the need to implement one on its own.

A last review on the possibility to omit the OS layer and let Java run directly on the FPGA platform is presented in [20], a Java optimized processor. The author mention that the usage of C and Assembler on top of a SoC is rather archaic compared to software design for desktop systems, and the following features Object-Oriented, Memory management with a garbage controller, Implicit memory protection and threads are not found in C. Furthermore, the main component of an OS in an embedded system includes (besides drivers) the memory management and threads. If they could already be found in the language the need for a OS can be omitted. The work then presents a JVM implemented as processor on an FPGA. The original Java bytecode is translated by the processor into an address in the own microcode format that implements the JVM. To handle more complex Java bytecode like new or invokevirtual a sequence of JOP instruction is to be executed.
Xilinx provides several documents and examples on their website explaining how a reconfigurable system architecture can be designed. Those documents give a solid base to design custom reconfigurable design. However, additional system components such as an interrupt controller had to be included to get an OS running on it.

4.1 A reconfigurable design

There are two possible approaches shown by Xilinx EA PR design for a reconfigurable design. The bus macros can either be included and instantiated in custom top-level module or integrated in a peripheral core. The actual PR region has to be constrained in the top-level module, see Initial Budgeting in 2. Instantiating the bus macros in the top-level module is limiting the flexibility in the design. A top-level module has to be designed where the underlying system, created by EDK, and the PR region is included. The bus macros are instantiated in the top-level module to create the locked connection between PR region and system. And, as stated in the Xilinx documents, the DCM (Digital Clock Manager) has to be instantiated in the top-level too, see 4.1.

But the top-level approach did not create a feasible design flow because instantiating the bus macros and DCM in the top-level module creates additional complex signal routing, an inflexible design and it was not possible to compile a running bitstream out of it.

This led to instantiate the bus macros in an own socket bridge core 4.3 which connects the actual system with the PR region. This design does not need an extra top-level module, it can be synthesized fully by EDK. The constraint file system.ucf of the EDK design can then be modified to constrain the PR region and place the bus macros.

Figure 4.2 depicts the final system design including one PR region where PR modules are written to. As the EA PR design flow works with ISE and EDK version 9.2 the system has a PLB bus 3.4 for the processor and memory communication but an OPB bus for the attached modules like RS232 Uartlite, SystemACE and HwIcap.
The system consists of two bus systems. This is because the PowerPC is designed to be attached to the PLB bus.

- **plb.v34** - the Processor Local Bus (PLB) is part of the International Business Machines (IBM) CoreConnect Bus Architecture specification and is the high-speed data interface to the PowerPC core. Later on in Xilinx core releases the PLB v34 is replaced by the PLB v46. Cores interacting with the v34 bus system are not able to interact with the never v46 system.

- **opb.v20** - the On-Chip Peripheral Bus (OPB) has been deprecated for use by Xilinx. It is another piece of the IBM CoreConnect Bus Architecture specification. The reason for using this bus is the dated state of the XUPV2P board and the fact that the ICAP core communicates only over the OPB bus system.

The Block RAM (bram_block) and the PLB BRAM Interface Controller (plb_bram_if_cntrl) are connected to the PLB although they could be attached to the OPB.

- **bram_block** - is being used as the processor’s main memory in this design. It can also be used as a cache for the PowerPC to increase the performance. Block RAM tend to be rather small like 64KB, when applications get bigger the use of DDR RAM is necessary

- **plb_bram_if_cntrl** - this module is the interface to access the BRAM.

The plb2obp core bridges the communication between the PLB and the OPB. Attached to the OPB are

- **opb_uartlite** - is an RS232 Uart core to transfer characters to a standard VT100 terminal and communicating with the outside. The opb_uartlite core does not need any licensing like the Xilinx 16550 compliant UART.
4.1 A reconfigurable design

- **SysACE_CompactFlash** - the System ACE core is the memory solution to store non-volatile data. In the current design it is the place where the larger bitstreams are stored and loaded from if the board has to do a dynamic partial reconfiguration.

- **opb_hwicap** - the Internal Configuration Access Port taking care of the reconfiguration process.

- **opb2dcr_bridge** - this core bridges the gap between OPB bus and the Device Control Register (DCR) Bus. Generally the DCR is used to bypass the standard memory and bus controllers. Here it is used to switch the bus macro on and off. The DCR bus system is a daisy-chain architecture propagating the signals through all DCR attached cores. This will be useful later on as can be seen.

- **opb_dcr_socket** - a rather simple but important piece in the design. The OPB DCR socket establishes the bridge between the OPB bus and the connected core. As a core can be cause of a contention when the ICAP reconfigures the PRR, the bus macros are disabled no longer passing any signals from bus to attached core. Thus every time there is another RM loaded the bus macro has to be turned off before and turned on after the reconfiguration.

The layout has to be extended by a interrupt controller such as the **opb_intc** taking care of the **SysACE** and **Uart**. Additionally more system memory is needed involving the DDR Memory that is on the board. The now included DRR Memory controller demands a second clock as driver thus the final design includes two DCM instances (**dcm_0**, **dcm_1**).

Furthermore planAhead facilitates the placement of the bus macros which lock the routing between the static system and the PRRs. Generally a peripheral is
connected to the particular bus system through an IPIF [32] (IP Interface). This is a set of VHDL libraries which help to highly adapt a IP Core to changing needs. These wrappers encapsule the OPB bus and offer an interface which helps accessing the underlying component over software registers, interrupt or even DMA. However, Xilinx’ tutorial do not rely on the usage of an IPIF in its PRR and does an own address bus translation. Thus the most part of the example code has been dropped and replaced by an IPIF implementation. Now, whenever the user code is change and e.g. more software registers are needed this can be provided by a change in the parameters or by creating a new IP core with the help of EDK and copy the essential part in the PRR code.

Another piece in tying a RM to the static system is the OPB Socket Bridge.

![Figure 4.3: Internals of the OPB Socket Bridge](image)

The OPB socket bridge instantiates the bus macros. If those are defined in the top-level design, every time the design is altered the top-level design had to be modified. It also facilitates the PR region creation process. As the OPB socket is connected to the OPB bus there are 14 bus macros that take care of the OPB communication, see [32] and the source code. This is only a minimal set of signals that are routed so far. Depending on the methods the PR module has to implement the OPB socket has to include more signals and additional bus macros need to be instantiated. Some of the bus macros have to be disabled during reconfiguration, to prevent bus congestion, and this has to be signaled to the OPB socket. Therefore, the OPB socket connects to the DCR bus which is intended to control the OPB socket to enable and disable the bus macros within. The DCR bus is coupled to the OPB bus which allows to control the OPB socket offer the OPB bus. This is simply a design choice.

### 4.2 Design Flow

Once a particular design has been created it has to be synthesized. Again the partial reconfiguration tools from Xilinx only support ISE and EDK up to the version 9.2, additionally Xilinx cut their support for the XUPV2P board making the later versions infeasible to be used.

After evaluating Xilinx’s documents about partial reconfiguration and how to
4.2 Design Flow

synthesize the created design two possible approaches have been manifested. The more and complex way of the command line PR implementation and the other way of using Xilinx’s planAhead which almost takes care of everything. Here only the command line PR implementation could be fully evaluated as the planAhead tool has not been obtained. Although a brief glimpse shall be given.

**Step 1: Create Processor System** The initial step is to create a PR design. This is done in EDK 9.2, the reference design is related to 4.2 with only one PR region. Another PR region is not feasible at the time as relocation of bitstreams is not possible. EDK 9.2 does not support the XUP board in its BSB, thus the IP cores can be added but have to be connected by hand. To operate the boards EDK does not yet support partial reconfiguration and some modifications have to be done in the hardware specification file (system.mhs). This includes the pin configuration of the OPB socket bridge and PRR.

**Step 2: Create Peripheral IPs** EDK provides a wizard to create new IP cores and import them. A template of a IP core is generated with all the specified options (number of software registers, interrupt control, DMA). The template will then implement the desired logic functions which will become the PR module. 

**Step 3: Synthesize initial design** Once the system has been designed it will be synthesized. This includes the static design as also the peripherals. To synthesize and to obtain the corresponding netlist, the .ngc files, for the static part use Hardware -> Generate Netlist. The RM can be synthesized with the help of ISE. When synthesizing an RM the insertion of I/O buffer has to be disabled as this is a lower-level module.

**Step 4: Create a planAhead Project** The command-line involves a accurate process flow that is error-prone when incorrectly executed. PlanAhead is a graphical editor to manage modular system design and constrained layout and can also be used for partial reconfiguration design. PlanAhead is given the top-layer netlist (system.ngc), the location of the bus macro file and directories where the custom peripherals are stored. To enable PR management, click File -> Set PR Project.

**Step 5: Create Area Groups/Place Components** Once the project has been created planAhead shows an overview of primitives and nets as the floor-plan of the FPGA chip. Pblock are placed to mark the PR region on the layout and set as reconfigurable reconfigurable. Although it is not compulsory to use the full height of the chip any longer its advisable to still take the full height. Here the PR region is at the right edge (X58,Y2 to X64,Y93). Several PR module (netlists) can then be assigned to the PR region as implementation and are included into planAhead’s design flow. After the PR region is placed, the bus macros and clock primitives have to be constrained on the layout. The DCM_INST instance is placed at DCM_X2Y0, the CLK0_BUFG_INST at BUFGMUX0P. All the available bus macros are placed at the left side of the PR region. The narrow-width bus macros cover two CLBs of which the left rests in the static region and the right in the PR region. After the primitives
have been places planAhead takes full care of the synthesis. Additional design rule checks can reveal timing issues or placement errors. At this stage, the ExploreAhead Runs tab should show a static design and at least one RM module entry.

**Step 6: Run PR Flow** Before the process flow is finally activated planAhead needs to know the BRAM composition (logical) where program code will be stored when the FPGA will be configured. This is described in the system_stub.bmm file and the later system_stub_bd.bmm file describes actual BRAMs used in the implementation.

In the ExploreAhead Runs tab select static and the options tab in Run Properties view. From there select the -bm option under ngdbuild and set it to the system_stub.bmm. Now the bitstreams can be generated by right-click and Launch Run. For the static design, planAhead will now invoke the ngdbuild, map and par as followed:

```bash
ngdbuild -intstyle ise -modular initial
-uc "static.ucf"
-bm <project dir>/implementation/system.bmm
"./top.edn"
map -intstyle ise -pr b "top.ngd"
par -intstyle ise "top.ncd" -w "top_routed.ncd"
```

And for each reconfigurable module:

```bash
ngdbuild -intstyle ise -modular module -active opb_prr_0_wrapper
-uc "opb_prr_0_wrapper.ucf"
"././././top.edn"
map -intstyle ise -pr b "top.ngd"
par -intstyle ise "top.ncd" -w "top_routed.ncd"
```

After generating the bitstreams for the static design and each RM bitstream of the modules we have to generate the particular bitstreams to reconfigure only the PRR instead of the entire FPGA. To build those the command `PR_verifydesign` is used.

Execute:

```bash
PR_verifydesign static.ncd adder.ncd multiplier.ncd
```

This will create the partial bitstreams adder_partial.bit, multiplier_partial.bit which contains only logic for individual RMs. Once the partial bitstreams are generated the different bitstreams of the design have to be assembled into a single one. The final design will have a PR module already loaded into the PR region. To assemble the different bitstreams `PR_assemble` is used.

Execute:

```bash
PR_assemble static.ncd adder.ncd
```

This creates static_full.bit which contains RM adder in the PRR. It will also create U2_math_blank.bit which can replace the math PRR with blank logic (no functionality in the corresponding region).

**Step 7: Create Image and Test** At last planAhead creates a static_full.bit, the static system, and the partial bitstreams for the RM as additional bitstreams to blank the RM meaning that the PRR gets erased and a blank PRR is left. The static_full.bit contains up to this point only the hardware description. What is needed is the software written in the BRAM such as the Linux kernel.
4.2 Design Flow

Execute:

```
data2mem --bm implementation/system/_stub/_bd
   --bt Merges/static/_full.bit
  --bd simpleImage.virtex405--mine.elf tag ppc405\_0
   --o Merges/download.bit
```

This will generate the download.bit in the Merges directory.

**Necessary Tools**  
Once an initial design flow has been done and a system is compiled the question rises if this same process has do be done over and over again only to compile one additional module into the existing system and what are the minimal tools to be used when a new reconfigurable modules wants to be generated. There are some some ways to shorten the path. But first have a look at the way EDK is synthesizing and compiling the system.

At first EDK generates the necessary .vhdl files from the system.mhs platgen -p xc2vp30ff896-7 -lang vhdl system.mhs
This will create the top-level wrapper files of all the system’s core, its BMM file and synthesis project files.

The synthesis project file can then be processed by xst (short for Xilinx Synthesis Technology). And at least the final .ngc file of every core xst project is composed from the xst generated .ngc netlist files.

The initial platgen call can be dropped, too much effort and time consuming. This left two possible points open for a short-cut. Either the xst synthesizing of a project or the subsequent ngcbuild where final netlist is composed.
Chapter 5

Architecture

5.1 FPGA Extender

As mentioned, standard OSGi bundles are not much different from Jar files. So the extended version of the OSGi bundle, an FPGA Bundle, will not be much different but only contains more information. A bitstream has to be included in the OSGi bundle and the classes that provide the interface to the PR region on the FPGA. The interfacing with the PR region is done with JNI. A native library, here written in C, is used to memory map the hardware into user space and through Java native methods the library functions are accessed. The Java class providing the interface to the native library and the library itself can be seen as a driver file. The driver has the task to operate the hardware correctly, react on status changes, read out memory register and feed in data.

The FPGA Bundles have to provide additional information in the header manifest. As these header information can be read out yet before the actual bundle has to be loaded and resolved it provides the place to store information about information of the included bitstream. A design decision is to let the manifest file not contain all the information about the FPGA Bundle as it is actually meant to provide OSGi specific details. Thus a separate XML file is the solution to describe the overall structure of the FPGA Bundle and it can be parsed by already available Java classes. The XML parsing is done with the Java SAX packages.

OSGi services are normally registered by the providing bundles itself. The common approach is to register the service object when the start method of the bundle activator is called. The framework then notifies all Service Listener that a service has been installed. Every listening bundle requiring this service can then request it.

With the PR region, a single resource is introduced to the system which cannot be reconfigured arbitrarily. If a bundle service already depends on the loaded RM another bundle cannot reconfigure the used PRR and load another bitstream. The framework has first to check if the RM is still used and inform the requesting bundle of the current state. This introduces too much complexity and overhead if the reconfiguration of the PR region would be left to bundles. Thus, a bundle would have to make sure
that the PR region can be reconfigured and is no longer utilized by other services but the framework in its current state does not support such a feature. Second, the bundle programming would be more error prone as more use cases had to be considered. The extender pattern is used. It assigns the registration of special marked bundles to one bundle – the FPGA Extender. Upon activation of the FPGA Extender it registers itself as a bundle listener. From now on he gets informed about every bundle’s state change. Referring back to the various bundle states the FPGA Extender can handle an FPGA Bundle when it gets installed, resolved or activated.

A possible entry point to manage the additional information like bitstream and driver class would be when the bundle has been resolved. But Concierge does seem to have loaded the bundle’s context only after the bundle has been fully activated (started). This makes it a bit difficult for a bundle to provide its own hardware relied service. Since the bundle context is not ready in the resolved state, a forced activation by the FPGA Extender (using Bundle.start()) will start the bundle which assumes its service has already been installed. But this is not the case because the FPGA Extender can only have registered a service for the bundle when its bundle context would have been available, which is not in a resolved state. This implies that bundles can not rely on their own services which includes the hardware module. Thus a separation into an application bundle and a service providing bundle, the FPGA Bundle.

To register a service its class object has to be instantiated. The class name is normally known to the registering bundle but does not appear to the FPGA Extender. To be able for the FPGA Extender to register the service of the FPGA Bundle it must know the class name to instantiate.

```java
Class<? extends Class<?>> clazz = bundle.loadClass(className);
if (clazz != null) {
    Object service = clazz.newInstance();
    FPGAService fpgaService = new FPGAService(service, this);
    ServiceRegistration serviceRegistration =
        bundle.getBundleContext().
            .registerService(className, fpgaService, props);
}
```

Listing 5.1: FPGA Extender

The bundle variable is a reference to the registered bundle implementing the service class. The method `loadClass(String)` creates a class which can be instantiated. Once the service class has been created it could technically be registered if the framework would ensure that the PR region is correctly registered every time the service is called. But this remains part of the FPGA Extender’s task. Thus the FPGA Extender has to manage the FPGA Bundles, its bitstreams and reprogram the FPGA to the systems requirements. Mostly this will be one application which profits the most by accessing the hardware. The application can be the currently running and focused foreground process, or a calculation-intensive application profiting from the hardware support. The applications can use different service implementing a hardware module to profit of the hardware. A method invocation of a service object requires the FPGA Extender to reconfigure the system appropriately.
This requires to notify the FPGA Extender somehow of the method invocation. What can be done by re-routing the `BundleContext.getService(ServiceReference)` and `BundleContext.ungetService(ServiceReference)` with a ServiceFactory. Instead of the service instance, an instance implementing the ServiceFactory interface can be registered and provides the FPGA Extender with the ability to react on `getService` and `ungetService` methods.

The FPGAService constructor takes the service object and the FPGA Extender itself. Every time the `getService` or `ungetService` method is called the FPGAService is requesting the FPGA Extender to reprogram the PR region for its need. Upon success the FPGAService returns the previously stored service object. If the reconfiguration cannot be done, because another service occupies the PR region, null is returned. This inflicts the framework and the service request. Concierge has to cope with the fact that a service object can possibly be null. The current implementation had to be adapted to this case. As now a service reference can return null for a service object the requesting bundle has to check on null too.

Previous implementation of the OSGi framework did not provide to register a service under another bundles reference. Thus every FPGA Bundle would have its services registered under the FPGA Extender bundle. But this would be a wrong view of bundles providing services. To force the framework to register the service under the original bundle reference its bundle context can be requested. The FPGA Extender can register the service on the context of the installed bundle. Now that the service registration is done by another bundle than the one that defines the service implementation the class name must also be defined in the XML file for the FPGA Extender to create an instance out of it.

This brings down the design of the XML file to specify the place of the bitstream in the bundle and the interfacing class name to the hardware under which the service will be installed.

The information which have still to be in the bundle manifest are the native libraries that have to be loaded into the system and the exported package that offers the service. An FPGA Bundle manifest header looks as followed:

```manifest
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: BndAdd
Bundle-SymbolicName: BndAdd;singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-NativeCode: lib/libJNIMath.so; osname=Linux; processor=ppc
Import-Package: org.osgi.framework;version="1.3.0"
FPGA-Contribution: config.xml
Export-Package: bndadd
```

Listing 5.2: MANIFEST.MF

This defines a bundle which uses the library libJNIMath.so to interface with the hardware and exports the package bndadd in which some of its offered service is stored.

---

1This should be coherent with OSGi specification 4.2. Described in [15], section 5.2.9 "Getting Service Object", is stated that the `getService()` method returns `null` if the underlying service object has been unregistered. Which is kind of true if the hardware module has currently been unloaded.
FPGA-Contribution marks the bundle as an FPGA Bundle and specifies the location of the XML config-file.

```xml
<?xml version="1.0"?>

<fpga xmlns="http://www.iks.ethz.ch/fpga">
  <package>
    <bitstream>opb_prr_0_adder_partial.bit</bitstream>
    <serviceclass>bndadd.MathAddImpl</serviceclass>
    <properties>
      description=an FPGA bundle,
      version=1.0.0,
      author=rkuepfer@student.ethz.ch
    </properties>
  </package>
</fpga>
```

Listing 5.3: XML config file

The `<package>` clause enable the bundle to define more than just one bitstream in it. A `<package>` has to define the bitstream name, the service class name which is registered as a service and which the FPGA Extender will instantiate. The `<properties>` clause define the properties for this one particular service. They are passed along in the service registration and can later be used to find and identify an appropriate service.

Once the FPGA Bundle has been loaded into the system the offered service can be accessed by another bundle, see 5.4.

```
String serviceName = "example.Service"
context.getServiceReference(serviceName);

if(sref != null) {
  IService service = (IService)context.getService(sref);
  if (service != null)
    System.out.println("Service_object_received");
}

context.ungetService(sref);
```

Listing 5.4: Service Access

Once the service object has been received by calling the `getService(ServiceReference)` method the FPGA Extender is notified that the service is in use. Every request of a bundle on the same hardware module increases the reference counter. As long as the service object is used by any bundle any other FPGA Service cannot be loaded into the system and accesses by a bundle. This is because the `getService(ServiceReference)` method is rerouted to the FPGA Extender, checked if another service is already in use and returns null if so. This also shows the importance for bundles to release a no longer used service, otherwise it is not possible for the FPGA Extender to load another FPGA Service on demand. So once a bundle does not need an FPGA Service anymore it is strongly advised to release the FPGA Service with `ungetService(ServiceReference)`.
5.1 FPGA Extender

5.1.1 JNI to access the Hardware

A short introduction to JNI is in section 2.3 but the use of JNI in this thesis is best explained at actual code. Native methods are handled by the JVM as functions provided by a shared library which has been previously loaded by the class loader. Calling a native method without a proper loaded library providing the implementation will result in an UnsatisfiedLinkError.

```java
class hardware {
    private native int init();
    private native int calculate(int a, int b);
    public hardware() {
        System.loadLibrary("JNIMath");
        if (init() != 0)
            System.out.println("hardware initialization failed!");
    }
    /∗ public methods to expose private native methods ∗/
    public synchronized int calc(int a, int b) {
        ...}
}
```

Listing 5.5: JNI native methods

Listing 5.5 shows a minimal class to make the underlying hardware available to the system. The native init function is be used to map the hardware memory into user space memory. An important point to consider is accessing the hardware in a controlled manner. A service cannot just be opened to the public and let any method invoked on it. A method operating on the hardware cannot be invoked several times without waiting that previous invocations have finished. Synchronized methods have to be introduced. In the calculate functions the mapped memory will be accessed. The loadLibrary functions loads the native library libJNIMath.so in the classloaders namespace to make the init and calculation functions available to the Java VM. With Java tools it is possible to generate the corresponding C header files which will implement the underlying C library.

```c
JNICALL jint JNICALL Java_jni_Hardware_calculate (JNIEnv *, jobject, jint, jint);
JNICALL jint JNICALL Java_jni_Hardware_init (JNIEnv *, jobject);
```

Listing 5.6: JNI native methods

Those are the prototype of the functions to be implemented. A compiler then generates the shared library which is later be loaded into the JVM.

5.1.2 Java Accessing the ICAP

Two different methods for accessing the ICAP have been implemented and evaluated. The first one is to map the hardware memory region of the ICAP into
user-space by using mmap. Xilinx further provides libraries and functions to operate on the ICAP in a controlled manner. This involves initializing, reading and writing configurations as checking on status information can be done by the provided functions. Error checking and ensuring to execute the commands in correct order has to be taken care by the programmer itself. Those functions are intended to be used in a SoC probably without a complete OS layer installed, but can also be used in C programs compiled for the OS running on top of the FPGA. This means they can be wrapped by JNI functions and imported to a Java framework.

Once there is running a Linux OS on the FPGA there is also the option to install a kernel extension (usable also as module) that provides an interface to the ICAP hardware as character device (normally under /dev/ICAP). This kernel component is delivered in the Xilinx Kernel. As the ICAP device is implemented as a character device complete bitstreams can directly processed into it, by e.g.

\[ \text{cat bitstream} > /dev/icap \]

Another fact is that with the plain C functions a bitstream has to be stripped from its header information and only configuration data itself has to be written to the ICAP. If this has not been met a second reconfiguration will fail again with a freeze. Strangely the Linux device does not have such a boundary and can be fed with the whole bitstream at once. There are reasons for both of the two approaches to access the ICAP and reprogram the PRR. However, the ICAP device presents a single point to access. Controlling happens by writing an appropriate bitstream sequence and reading out from the device. C programs happen to open a file descriptor on the ICAP device file. Unfortunately, Java is more complicated to handle when it comes to deal with device files. While the device can be opened with a FileOutputStream and the bitstream can be written to the Stream it requires to open the device file again for every reconfiguration. This is because the ICAP device initializes the underlying hardware whenever it is accessed. If reconfiguration happens infrequently, this overhead is affordable. An alternative approach is to open the ICAP device with a RandomAccessFile. This allows the control of the device by sending forged bitstreams containing commands, see section 2.1.2. However, a wrong sequence of commands can bring the device in an undefined state and freeze the system. Accessing the ICAP with a FileStream is thus more secure, bringing the ICAP device to a valid state for every reconfiguration and only use complete bitstreams.

5.1.3 Socket Bridge

As the ICAP device driver does not close the bridge between the PRR and the static System this has to be done by another piece of code. The bridge has to be closed before reconfiguration so the reconfiguration process does not inflict bus flaws. Two approaches to control the bridge are possible. The first, like for the ICAP case, to memory map the hardware memory with the C library mmap and lets JNI wrappers provide the interface to control the bridge. The second is to write a character device, e.g., /dev/socketbridge on a Linux OS. A kernel module [5] implementing the device’s open, write and read functions provides the required functionality to control the bridge in a convenient way. To open or
close the bridge between static system and PRR a 1 or 0 is written to the device. If the state of the bridge is required it can simply be read from the device file. Accessing the bridge out of Java is done in a similar way to accessing the ICAP hardware. A RandomAccessFile instance is used to operate on the device file. The driver is written to change the state depending on the first 4 bytes written to it. Thus Java will have to send a byte array of 4 bytes with the last defining the desired state (0, 0, 0, 0 for close and 0, 0, 0, 1 for open).
Chapter 6

Evaluation

The first dimension of evaluation is the static overhead of the system over a non-reconfigurable system. Running an OSGi framework on a system gives a software developer the possibility to extend the software system with additional functionality by installing new software bundles into the system. With the proposed FPGA Extender, the OSGi framework is now additionally able to reconfigure the underlying hardware if an FPGA is present in the system. FPGA bundles allow other software bundles to access a hardware-accelerated implemented function through their service interfaces. This offers the possibility to increase the application performance, or exchange a complete algorithm with another one.

With this in mind the following use cases are described for a small embedded device such as a mobile phone.

Setup A: The user likes to watch and hear videos. With its colleague he casually exchange music and video file. And because there are so much standards of music- and video-formats, it happens that his mobile device does not implement the required decoder.

Setup B: The user is a bit wary and likes to encrypt his mail. The mobile device already supports a corresponding encryption module implemented like DES. But this does not longer achieve the requirements for a secure email encryption.

Setup C: This kind of scenario will include an application which can benefit of two hardware-accelerated services. The FPGA Extender has to reconfigure the PR region alternating with the two bitstreams of the hardware module components.

An evaluation of the new FPGA Bundles is now presented and applied to the three use cases.

6.1 Improvements with FPGA Bundles

In the three setups, if the mobile device has the required functions (codec, algorithm) implemented in an ASIC, an installation of additional functions seems
Evaluation

not quite possible. Perhaps the running OS can install some software implementation but these do not get hardware support.

In Setup A the music player application will already be installed in the system and interacting with OSGi. The file format of the music file is determined. Then the music player searches the corresponding decoder service for the file format in the OSGi services and requests it. The decoder services implement all the same interface. He requests the service by calling BundleContext.getService(). The returned service object, implementing the given interface, can be used by the music player to decode the file format. That way, a new file format can easily be inserted into the OSGi framework. The provided service does not even need to be an FPGA Bundle. Additionally, the bundle can also be purely implemented in software, as long as the service interface is correct. But there could be data restrictions, e.g. a high enough sampling rate, where the software implementation cannot cope with the data volume. This is, where the OSGi framework can rely on a hardware supported implementation of the codec and gain the required calculation power to decode the file.

For Setup B the same approach fits. The application uses the provided services to encrypt and decrypt any data. Also here, the required service does not have to be based on a hardware component but will perform better if the underlying FPGA can be reconfigured.

The third scenario Setup C is rather a seldom case where the OSGi framework needs to reconfigure the PR region in short time to adapt to e.g., two different bundles and their service requirements.

Further for the above examples, it has been exposed that a stand-alone application is not feasible with the current FPGA Extender model and the implementation of Concierge. The FPGA Extender cannot access the bundle context of a non-activated bundle, thus it prevents to register bundles services under its context in the framework. Force-activating the bundle would first activate the bundle, assuming its own hardware services are available, but the FPGA Extender is installing the services not until he received an activation event of the bundle.

With a stand-alone application not entirely working the FPGA Bundle has been split into two separate bundles. One common OSGi bundle which implements the application and requires the resources of the FPGA Bundle which includes the bitstream and implements the driver functionality. The FPGA Bundle thus has to be first loaded into the system where the bundle's bitstream is stored and the service object is created. This is done by the FPGA Extender upon receiving the bundle's activated event from the framework.

With the above scenarios in mind, four FPGA Bundles have been created. Two bundles includes the bitstreams with simple functions to validate and check the reconfiguration properties. And the other two bitstreams (DES, Triple DES) to evaluate a hardware over a software implementation.

BndAdd, BndMul: Two simple modules which are handled by the FPGA Extender. The bitstreams are stored and service objects are created and registered. The service object can then be requested by another bundle.
6.1 Improvements with FPGA Bundles

Those bundles have been created to check the correct handling of reconfiguration.

**BndDES, BndTDES:** The complexer modules include a DES/Triple DES encryption. Those bundles contain some more method to control the underlying hardware.

As the FPGA Bundle only provides the bitstream and interface another two bundles are created to test these FPGA Bundles.

**Bundle Properties** The PR region of the system has been placed at the right edge of the chip are. It covers almost the full height of the device except the four rows of IOB and IOI at the top and bottom. The right edge has been chosen as there are as less I/O pins used at the right side in the current design while the entire memory bank is connected at the left side which would increase the static routes to consider crossing the module boundaries. The width of the PR region spans 8 CLBs, this allows to use the following resources:

<table>
<thead>
<tr>
<th>System</th>
<th>Slice</th>
<th>Mult</th>
<th>Ram16</th>
<th>TBUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRR</td>
<td>13696</td>
<td>136</td>
<td>136</td>
<td>6848</td>
</tr>
</tbody>
</table>

Table 6.1: Overall resources

<table>
<thead>
<tr>
<th></th>
<th>PRR</th>
<th>add</th>
<th>mul</th>
<th>des</th>
<th>tdes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>4480</td>
<td>90 (2.01%)</td>
<td>58 (1.29%)</td>
<td>1081 (24.13%)</td>
<td>3008 (67.14%)</td>
</tr>
<tr>
<td>FF</td>
<td>4480</td>
<td>176 (3.93%)</td>
<td>144 (3.21%)</td>
<td>513 (11.45%)</td>
<td>1527 (34.08%)</td>
</tr>
<tr>
<td>Slice</td>
<td>2240</td>
<td>108 (4.82%)</td>
<td>88 (3.93%)</td>
<td>660 (29.46%)</td>
<td>1835 (81.92%)</td>
</tr>
<tr>
<td>Mult</td>
<td>20</td>
<td>0</td>
<td>1 (5.00%)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2: Physical Resources used

Table 6.1 shows the overall resources available on the FPGA chip. The PRR takes approximately 16% of the entire FPGA fabric. While table 6.2 shows the amount of resources used by the different bitstreams. A slice contains two basic blocks which are made up of a LUT and a flip-flop. Thus, number of LUTs and FFs is two times the number of slices. Multiplier are not really used in those design except in the Multiplier module itself.

It can clearly be seen that the simpler modules (Adder, Multiplier) are only using very little of the available resources in the PR region. The DES implementation uses about a third of the PR region and the Triple DES implementation has 3 times the size of DES, this because there are 3 instances of the DES component in the Triple DES design.

The entire design is about 1 MB in size while the bitstream size are all equally sized, around 10% of the initial bitstream (static_full.bit). Each partial bitstream has to cover the entire PR region, only due that planAhead generates compressed bitstreams let the bitstream size vary.

The reconfiguration of the system is done in approximately 10-20 millisecond (single cases up to 35 milliseconds) which corresponds to measurements done
bitstream | bytes | time to write
--- | --- | ---
static_full.bit | 144887 (100%) | —
add | 123249 (8.50%) | 0.011159 secs
mul | 128222 (8.85%) | 0.024896 secs
des | 149087 (10.29%) | 0.013441 secs
tdes | 149088 (10.29%) | 0.013374 secs

Table 6.3: Bitstream size

by Ming Liu et al. The measured time to load an entire FPGA Bundle into the system, reconfiguring the PR region and loading the JNI libraries, takes about 30-100 milliseconds. The time seems in a usable range for an embedded platform if the tasks are not switched in the same time scale. Consider that allocating as much space as possible for a PR region could indeed resolve space issues but wastes a big amount of resources if only a small module is loaded. See numbers for Adder and Multiplier module in table 6.2. A tradeoff has to be made between the size of an allocated PR region size and the used size of implemented modules, which probably cannot estimated.

Given the position of the PR region at the right edge and covering almost one-sixth of the entire system, it is hard to position a second PR region. There are two PowerPC cores embedded and the I/O pin used to communicate with board components which creates unique static routes and different bitstreams for each PR region.

A possible approach is to split the one available PR region and make two. The modules FAR will lie in the same frames and the bitstream can be relocated by shifting its bits by a given offset, see [22]. It would be likely that the PR region, covering half of the height, will not be able to contain a Triple DES implementation as not enough space could be available.

### 6.2 Performance improvement through hardware acceleration

Apart from the feature to reconfigure the hardware, the benefit of a hardware implementation over its software implementation is of interest. To measure the overhead of accessing the hardware through JNI several methods have been evaluated to get a first impression.

As a use case Triple DES was chosen due to the complex operations made, and the manner how data is processed. Triple DES is a block cipher for data encryption processing 64-bit blocks. From an implementation point of view, this is easier to handle than a stream cipher processing a continuous stream of data. Additionally an implementation of Triple DES is provided as an open core project [14]. As a second point of comparison, the well-established software implementation of a Triple DES algorithm to compare the hardware implementation against, the library libdes 4.04b of Eric Young has been taken.
**Triple DES - Recapitulation**  A Triple DES algorithm is based on a multiple execution of a simple DES algorithm. The DES algorithm is a block cipher thus operating on a 64-bit wide data block. The initial key length used to encrypt the data block is also 64 bit wide of which 56 bits are selected by Permuted Choice. The 8 bits remaining are either discarded or used as parity check.

A single DES encryption consists of an initial and final permutation (IP and FP) \(^1\) also of 16 rounds applying the Feistel function (Box F). Before the main rounds, the 64-bit block is divided into 32 wide blocks which are processed alternatively. The exact procedure the Feistel function is doing can be looked up in [19]. The function generally does some permutations and substitutions with S-boxes.

Applying Triple DES involves every single data block being encrypted with key1, decrypted with key2, and finally encrypted again with key3. It is advisable to set the single keys independently from each other and not null. While in ECB mode (electronic codebook), the blocks are encrypted individually and form the resulting output.

The CBC mode (cipher block chaining) adds additional security as every next block gets XOR-ed with its predecessor and the first block is xor'ed with an initialization vector. The used IP core is a simple implementation of a Triple Des algorithm based on serial execution of the DES algorithm. This simple implementation thus only implements an ECB mode. CBC and other operation modes could be built either in the hardware layer by extending the implementation or it is possible to add this functions on a C

\(^1\)IP and FP have almost no cryptographic significance, but were apparently included to facilitate loading blocks into mid-1970s hardware, as well as to make DES run slower on software.
or JNI layer. Presumably a hardware based extension is faster than a software emulating implementation that has to manage the data blocks.

![Cipher Modes Diagram](image)

**Figure 6.2: ECB and CBC cypher mode**

The actual Triple DES module takes three 64-bit key register divided each on two 32-bit register as the PLB bus only transfers 32-bit at maximum in non-burst mode. There are two 32-bit register each for the 64-bit input and output and last but not least a 32-bit register to control the logic such as signaling the key/data is ready to load or the data has been processed. All in all are 11 32-bit register needed to operate the Triple DES module. One such software register can directly written with a 32-bit integer.

How the corresponding Java class has to be specified for JNI has been described in chapter 5.1.1. Listing 6.1 depicts the methods of the Java class to interact and control the Triple DES module. \(c\_load\_key\), \(c\_decrypt\) and \(c\_encrypt\) are invoked with a Java byte array which has to be extracted later on the JNI level. \(init()\) calls the JNI method to initialize the hardware such as memory map.

```java
class Core3DESnio {
    private native void c_load_key(byte[] key);
    private native byte c_decrypt(byte[] cipher);
    private native byte c_encrypt(byte[] plaintext);
    private native void init() throws IOException;
```

Listing 6.1: JNI method

The previous byte array is on the JNI level accessible as a `jbyteArray` array reference which has to be handled to receive a C byte array. `jbyteArray` is an opaque handle to a VM-internal data structure and thus cannot be accessed from C directly. Instead JNI provides functions to convert them to C array primitives. A first implementation to read the `jbyteArray` data out of the JVM uses the function `GetByteArrayRegion()` which stores the retrieved array bytes into a given, previously allocated, native memory region. To receive the `jbyteArray` length `GetArrayLength()` is used. Listing 6.2 shows an example code how the array is accessed.
6.2 Performance improvement through hardware acceleration

```c
JNIEXPORT jbyteArray JNICALL Java_tdes_jni_Core3DES_c_lencrypt
(JNIEnv ∗env, jclass class, jbyteArray plaintext) {
    size_t size;
    jbyte ∗_c_plaintext, ∗_c_cipher;
    jbyteArray result;
    size = (∗env)−>GetArrayLength(env, plaintext);
    _c_plaintext = (∗env)−>GetByteArrayElements(env, plaintext, NULL));

    /* call the actual crypt function, omitted */
    _c_cipher = encrypt(...)

    result = (∗env)−>NewByteArray(env, size);
    (∗env)−>SetByteArrayRegion(env, result, 0, size, _c_cipher);
    (∗env)−>ReleaseByteArrayElements(env, plaintext, _c_plaintext, 0);

    free(cipher);

    return result;
}
```

Listing 6.2: JNI call

After the bytes have been processed a new array has to be instantiated to return the result to the JVM. The function call `NewByteArray()` creates a new Java byte array in the JVM and `SetByteArrayRegion()` writes copies the bytes of the C native buffer into the Java array region. To avoid memory leaks, the retrieved array `c_plaintext`, has to be explicitly freed with `ReleaseByteArray()` before the native functions can return. An important point when receiving the pointer to the elements of primitive arrays is to note that it could return a copy of the original primitive array, because the underlying garbage collector may not support pinning, see [12]. This entire procedure of retrieving the array data out of the JVM, create a new return array and set its value creates a certain overhead comparing the JNI interface to a plain C implementation. Including the possibility of receiving an array copy means the original data cannot directly altered.

**Results** Testing of the above approach has been done on the XUPV2P board including one PowerPC in the system which runs at 300MHz. 256MB of external DDR Ram memory are available to the Linux OS. The system design is similar to the one presented in section 4.1, however, all the cores are attached to the PLB which is clocked at 100MHz. The benchmarks have been done with this system configuration first before the actual PR design has been implemented. A test run includes to process packet size of a power of two, ranging from 8 to 32768. There have been 100 such test runs executed of the gained values the mean and standard deviation is calculated. The gained values are the average values of 100 test runs, the error bars show the standard deviation.
On figure 6.3, there seems little to no difference comparing the hardware versus the software implementation. Running with a plain C application standard deviation is very small in comparison to accessing the crypto functions through JNI. The JVM cannot execute JNI methods within a constant time compared to the C program. This can have its cause in memory allocating structures which are passed down to the native methods. If OS primitives would be involved then C would suffer of the same effects. Another explanations the Java thread model. The C program accesses the hardware within an own thread while Java manages also other threads.

6.4 continues the row up to 32768 showing that the plain C methods almost have no deviation. The JNI void call measure the time the actual JNI call requires to get the array data out of the JVM without encrypting it. For smaller packets the void calls are slower than the actual C implementation, reaching a packet size of more than a 10000 bytes the JNI void calls are faster returned than the plain C calls. But the difference between them is also small so that it is not likely that the encryption in Java beats the performance of the plain C implementation at any point.

The fact that the JNI software implementation is not competitive against the C software implementation has also its cause in how the Triple DES function is applied on the data. In C a array data can directly altered when not defined as constant, in Java the received buffer cannot directly processed due the chance of been copied and a return array is created.

Since the result is not satisfying in terms of performance JNI offers support for the java.nio package which improves performance in the areas of buffer manage-
6.2 Performance improvement through hardware acceleration

Figure 6.4: Benchmark for 1024 up to 32768 bytes

The NIO ByteBuffer, direct buffers, are a construct which allows a JNI C function to directly access the Java buffer without the need to invoke the proper JNI function. The contents of a direct buffer can, potentially, reside in a native memory outside of the ordinary garbage-collected heap. The API introduce of only 3 new essential functions to support the proper use of ByteBuffer constructs.

**NewDirectByteBuffer** Allocates and returns a direct java.nio.ByteBuffer referring to the block of memory starting at the memory address address and extending capacity bytes.

**GetDirectByteBufferAddress** Fetches and returns the starting address of the memory region referenced by the given direct java.nio.Buffer.

**GetDirectByteBufferCapacity** Fetches and returns the capacity in bytes of the memory region referenced by the given direct java.nio.Buffer.

Now the encrypt and decrypt method of the NIO version have the input parameter changed from a byte array type to a ByteBuffer type. Thus there is also no need to return a byte array as the NIO implementation will do the encryption and decryption directly on the passed buffer, shown in listing 6.3
public class Core3DESnio {
    ...  
    private native boolean c_decrypt(ByteBuffer cipher);
    private native boolean c_decrypt(ByteBuffer plaintext);
    ...  
}

Listing 6.3: JNI call

A ByteBuffer is is either direct or non-direct. If the ByteBuffer is instantiated as direct buffer, the Java Virtual Machine will make a best effort to perform I/O operations directly upon it, thus it will attempt to avoid copying of the buffer’s content from and to an intermediate buffer before or after each invocation of one of the underlying operating system’s native I/O operations. The direct ByteBuffer has thus a somewhat higher allocation and deallocation cost than non-direct ones. Which becomes redundant with large enough amount of data processed. The C code can now be adapted and most JNI upcalls are replaced by the ByteBuffer functions.

```java
JNIEXPORT jboolean JNICALL Java_tdes_nio_Core3DESnio_c_encrypt(JNIEnv *env, jobject class, jobject buffer) {
    jlong size;
    /
    /* get the address of the data buffer */
    char * _jni_plaintext = (char *)(*env)->GetDirectBufferAddress(env, buffer);
    size = (*env)->GetDirectBufferCapacity(env, buffer);
    /
    /* call the actual function to encrypt */
    /* encryption done on the plaintext buffer */
    return crypt((char *) _jni_plaintext, size, ENCRYPT);
}
```

Listing 6.4: JNI NIO

In listing 6.4 is shown that now only 2 JNI upcalls are used to access data of the buffer. Once the memory address of the buffer itself and once to get the size of the buffer.

The C implementation is again the lowest possible time to process the data while the JNI methods show a high deviation. This shows that the invocation time of JNI methods could not be decreased significantly through java.nio. Compared to figure 6.3, it seems that invocation time for small packets is even worse. Tough, the hardware implementation of Triple DES will require less time than the software implementation to process the data, seen in figure 6.5.

In figure 6.6 the progress for bigger packets is pictured. The plain C software implementation could not be beaten with a JNI hardware implementation but came closer to the time the C software implementation requires. Still with a big deviation. It can be seen that with the use of NIO primitives, the overall processing time of the JNI approaches has dropped by estimated 3 to 4 milliseconds.
In order to do more fine granular bookkeeping on the time consumed in a single JNI call, a higher precision probe mechanism is required. The PowerPC provides capability to read out the PowerPC time counter which is incremented either equal the processor core clock rate or as driven by a separate timer clock input. The PowerPC time count is a 64-bit vector field guaranteeing an accurate time scale. The current time count is read out with:

```c
static __inline__ unsigned long long rdtsc(void)
{
    unsigned long long int result=0;
    unsigned long int upper, lower, tmp;
    __asm__ volatile(
        " 0:
        \tmftbu %0\n
        \tmftbu %1\n
        \tmftbu %2\n
        tcmpw %2,%0\n
        "tbne 0b\n
    : "=r"(upper),"=r"(lower),"=r"(tmp)
    ) ;

    result = upper;
    result = result<<32;
    result = result|lower;

    return(result);
}
```

Listing 6.5: Read-out time counter

This has been used to measure the time for single operations like the transferring of array data out of the JVM into a plain C buffer and the actual encryption/de-
The results are shown in 6.7 and 6.8. The result for the packet size from 8 to 256 bytes is shown in Figure 6.7, measured are the overall ticks, and the time to process the data in the encryption function. The time to pass the data between the Java/C boundary is the difference of the overall time and the encryption time. The behavior of JNI for the lower sized packet is not neat and constant, the measured clock time has no linear increase, the same behavior is seen in Figure 6.3. The behavior of the C program shows a linear increase in the overall time and the difference to the processing time is decreasing with bigger packets.
Figure 6.7: JNI Clock Benchmark packet size 8 to 256

Figure 6.8 now shows the bigger picture of the range from 512 to 32768 bytes size packets. The development of the measured time is partly explainable. First, the difference of the overall time between the C and JNI implementation is approximately a factor of two. And it can be seen that with bigger packets the passing of data between the Java/C boundary does not account much to the overall time. Strangely, the measured time the data is processed in the actual hardware is in the JNI approach also a factor two bigger than in the plain C program. Theoretic, the encryption process should perform in approximately the same time, as it is done in hardware. This is only explainable if the native C function are processed by the JVM and thus sustain a performance loss.
Figure 6.8: JNI Clock Benchmark packet size 512 to 32768
Chapter 7

Conclusions and Future Work

In this thesis a possible approach is presented to combine the dynamic model of OSGi with reconfigurable FPGAs. Partially reconfigurable designs have been analyzed to understand the features and drawbacks. Then an own embedded system has been implemented with one PR region which can be reconfigured to implement different and new hardware components. With the help of the FPGA-intern ICAP the particular FPGA can be reconfigured during runtime while the OS is operating. A device driver, the OPB socket bridge, has been written to control bus signals between the OPB and PR region. The socket bridge prevents bus corruption during reconfiguration. OSGi, an application lifecycle management, has been set on top to provide a module system which implements a complete and dynamic component model. The used OSGi framework is Concierge which had to be slightly modified to handle developed FPGA Extender bundle and the FPGA Bundle. The FPGA Extender controls the ICAP and the socket bridge driver and allows the use of a new OSGi bundle, the FPGA Bundle.

The FPGA Extender is the only instance that manages the reconfiguration and the FPGA Bundles. As only one PR region is available, the FPGA Extender makes sure that no two hardware services can interfere with the current hardware component in the PR region. This has been achieved with a ServiceFactory. The FPGA Extender is notified by a particular ServiceFactory instance to do a reconfiguration. This offers some security as no other services can request a reconfiguration. But a change to a proxy pattern, Java Proxy, could allow reconfiguration on a fine-grained method-level and not on a service request/release-level.

The FPGA Bundles are hardware supported services including a bitstream to reconfigure the PR region with new hardware components. They implement a given interface and can requested by other bundles to use its hardware accelerated functions. As a result, the new introduced FPGA Bundles offers a flexible and transparent way to include hardware-accelerated services into the software system. The new FPGA Bundles are not any different request and invoke than any other standard OSGi bundle. This preserves any paradigm shift of current programming models which allows to support even current application bundle
relying on standard OSGi services and provide a hardware-acceleration.

Also, various methods to access the underlying hardware have been evaluated. These results are not as promising as hoped. The chosen way to access the hardware has been JNI which suffers of some overhead compared to a C approach. However, it can be seen that a hardware implementation of Triple DES accessed through JNI can cope with a C software implementation of Triple DES. To beat the C hardware implementation, this is probably out of reach for the next few years. Building the system on top of a Java optimized processor could omit the OS Layer and eliminate the need for JNI. This would probably also lower the time used to pass data to hardware components.

It has been seen that bitstream relocation is almost not to achieve, especially on a XUPV2P development board where the static system lies on the same FPGA as the hardware module components. Also it is not approachable in a feasible design flow. [10], [6] and [22] achieved bitstream relocation in special cases. Some approaches could be the topic of future work. Bitstream relocation could increase flexibility of the FPGA Extender as more than just one PR region would be usable.

**Future Work** There exist some ideas how to extend the current design to cope better with demanded services and the reconfiguration of the underlying PR region. Also, considering a Java optimized processor, the OS layer could probably be left away and the OSGi framework directly applied to the embedded system.

- The current design relies on a PowerPC architecture and the above OS. Both could maybe replaced by JOP [20]. A Java processor would omit the OS layer and the OSGi could be set on top of the embedded system.

- Also the current system has one PR region designed in it. This provides only one service with hardware support. Although the sources are split about the resources to relocate a bitstream to another PR region it would be worth investigating the current possibilities.

- Current FPGA Bundles only provide a bitstream of a hardware implemented service. A possible approach will be to pack an additional software implementation of the service in the bundle. This lets the FPGA Extender the choice which implementation to take. If the PR region is currently used the FPGA Extender can instantiate the software implementation. This could help to keep the bundles, requiring a hardware service, at work, though not with hardware support.

- The solution to use the ServiceFactory of OSGi to intercept the BundleContext.getService() and BundleContext.ungetService() can probably be optimized. As already mentioned a Java Proxy class could possibly be taken instead the ServiceFactory. This will allow a fine-grained usage model but as well increase the management complexity as the FPGA Extender has to manage hardware module loading and unloading on a invoked method base.
• In the current implementation most of the JNI drivers and device drivers have hardcoded hardware addresses. This could be made more sophisticated that they e.g. load their hardware address dynamically out of the kernel device tree.
Conclusions and Future Work
Bibliography


BIBLIOGRAPHY


Appendix A

Linux - OS and drivers

Linux is an operating system which runs on almost any architecture. Due to the fact that its source code is open, drivers for rather exotic hardware (such as the ICAP) are available or can easily be written, and many communities also provide a certain amount of support. Still there are some restriction on the choice of the linux distribution depending on the underlying hardware.

Microblaze and uClinux  Xilinx offers in EDK a softcore instance of a processor called Microblaze. This, like every other FPGA module, is written in a hardware description language and can be synthesized in a netlist as well as run on the chip. The Microblaze itself is a 32 bit RISC processor. The lack of a MMU requires the OS to take care of Memory Management itself, like uClinux or MontaVista Linux which are developed for embedded systems with low resources such as FPGA. But using a Microblaze would make the two hardwired PowerPC405s on the Virtex-II Pro obsolete.

PowerPC405 and Xilinx Kernel  The PowerPC405 is a 32 bit RISC processor containing 16KB of Instruction, 16KB of Data Cache and a MMU. Usually the caches are pre-loaded with a software application which is executed on startup and runs completely in the cache without requiring external memory. A Linux kernel configuration typically requires between 2 and 8 Mb of memory. The kernel has to be loaded from external memory. While the bootloop.elf file is stored in the cache, a self-branching executable guaranteeing that the processor is operating in a known state. As the boot loop is stored in the FPGA BRAM the describing BRAM file <system>.bmm is used to utilize PLB BRAM.

Bootup procedure  Apart from loading the Linux kernel via JTAG and xmd, it can also be stored and loaded from a separate partition of the flash card. The bootup sequence involves the System ACE CF companion chip which can be seen as part of the boot-chain, a bootloader in an abstract way, see [18]. The bitstream, including the Linux kernel, is processed by GenAce, a tcl script executed in xmd, and a system.ace file created. This file can then be read by the System ACE chip.
On power-on, the System ACE chip will read the system.ace from the flash card,
configures the FPGA and PPC, and then loads the kernel image into processor memory, see A.1. The PowerPC405 program counter is set to the starting address of the Linux kernel and after the system is up and running, the processor can use the flash card as a secondary storage device for its root file system.

Figure A.1: boot process with System ACE

The included MMU allows a normal Linux distribution to run on the system, instead of a modified version such as uClinux or MontaVista Linux which are handling the absence of the MMU.

Although it is assumed that the processor architecture would follow a big-endian architecture, it has to be mentioned that the PowerPC405 is behaving "oddly" and mentioned as a bi-endian processor. This expresses itself in the behaviour of a little-endian on the application level (user level) and as big-endian on the kernel level. Xilinx provides under [39] an own kernel adapted for embedded system supporting various of their reference designs. The Xilinx kernel can be compiled for the PowerPC405 architecture with a cross compiler. Setting up a complete cross compile toolchain can be done with various tools, a comfortable way is to use ELDK (Embedded Linux Development Kit). ELDK provides a complete image of a cross compile toolchain such as compilers, binutils, gdb, etc. Some versions come with two libraries Glibc or uClibc. Installation instructions are described under B.1. The flash card of the Virtex-II Pro board will provide the place for the Linux root filesystem beneath the partition that holds the initial bitstream which is loaded at startup. Debootstrap, explained under B.2, offers a simple way of creating a full root filesystem of a complete GNU/Linux distribution on the flash card.

When EDK synthesizes a bitstream for the FPGA every core gets a memory address assigned and as seen in the memory map of the implemented system, listing A.1. This memory map is reflected in the device tree (.dts file) which describes the entire system design.
Listing A.1: Address Map for Processor ppc405

The device tree file (.dts), generated by tools in EDK, describes the entire FPGA core structure. Corresponding to the memory map the address and size of the core is listed. All the buses (such as PLB, OBP and DCR) are listed and the file also shows how their are connected with each other, up to the processor system and memory bank layout. For how to generate the device tree, see [39] or B.5.4. The device tree is an essential part of the kernel compilation and has to be copied into the `<kernel sources>/arch/powerpc/boot/dts`.

The synthesis process generates the `system.bit` representing the placed and fully routed modules on the chip, though this would not be runnable due the lack of a simple bootloader. The last step is to code a bootloader in the bitstream with `data2mem` which helps to process software code images into the memory options of an FPGA without the need to (re)run place & route (par). The final bitstream gained is `download.bit` which can be processed further into a `system.ace` file to load from a flash card.
Appendix B

Setup & Configuration

B.1 ELDK - Embedded Linux Development Kit

- First, download the ELDK 4.2 ISO from http://www.denx.de/. The downloaded ISO is either to be burnt on DVD or directly mounted as a loop device.
  
  root> mkdir /media/cdrom
  root> mount -o loop -t iso9660 ppc-2008-01-04.iso /media/cdrom

- Next step is to install ELDK. Create a directory to install the ELDK into, and run the install script out of the ELDK directory. To import the ELDK's environment variables into the shell environment execute source eldk_init 4xx in the ELDK directory. This has to be done for every new bash. Alternatively include this in the .bashrc file.

  root> mkdir /opt/ELDK
  root> cd /media/cdrom
  root> ./install -d /opt/ELDK/4.2
  root> cd /opt/ELDK/4.2
  root> source eldk_init 4xx
  root> /media/cdrom/ELDK_FIXOWNER -a ppc_4xx
  root> /media/cdrom/ELDK_MAKEDEV -a ppc_4xx

  root> source /opt/ELDK/4.2/eldk_init 4xx

B.2 Debootstrap

- Debootstrap is a simple solution that allows you to copy a fresh copy of GNU/Linux into a directory. The installed system will have all the basic packages to run a simple system.

  root> debootstrap sarge /mnt/

B.3 Impact - libusb driver

A problem is that Xilinx's Platform Studio utilizes the Jungo Winddriver which does not compile with recent Linux kernel. The solution is to switch to the libusb
drivers provided by the community to use the download cable without Xilinx’s cable driver. Although there is plenty of information on the INTERNET some short notes about installation and configuration of the driver should be given. Upon installing the USB-drivers the environment variable LD_PRELOAD has to be set to the installation directory of libusb-driver.so.

```bash
export LD_PRELOAD=<install_dir>/libusbdriver.so
```

Afterwards it is possible to load the ppdev module into the kernel to bring up the /dev/parport0 device.

The `impact` tool can now be started and interact with the Xilinx board through the Xilinx USB platform cable.

### B.4 Bitstream Compilation

The bitstream contains the boards layout. If the board is rewritten and components added or removed to or from the board the new bitstream only contains the difference between the old and the new layout. Thus, rewriting should be faster than programming the whole board again.

**create bitstream**  Xilinx application note [38] offers a detailed view of the possible reconfiguration strategies and how to setup the bitstream. Here a short summary explains which essential options have to be set to prepare a bitstream for partial runtime reconfiguration.

The initial bitstream requires the `-g Persists:Yes` switch and the `-g security:None`. For active partial reconfiguration the bitstream has to be generated with the `-g ActiveReconfig:Yes`, meaning that the device remains fully operational while the new partial bitstream is being downloaded. The `-g Persist:Yes` is also required when utilizing partial reconfiguration through the SelectMAP Mode.

The `-r` switch used in the BigGen utility produces a difference-based partial reconfiguration bitstream, only containing the differences between the input `.ncd` file and the original bit file.

**Generic Example:**

```bash
bitgen -g ActiveReconfig:Yes -g Persist:yes -r <original.bit> <new.ncd> <new.bit>
```

**Test Example:**

```bash
bitgen -g ActiveReconfig:Yes -g Persist:Yes -r and <test.bit and test2.ncd and test2\_partial.bit
```

Create a Partial Bitstream to Restore the Original Design:

```bash
bitgen -g ActiveReconfig:Yes -g Persist:yes -r and <test2.bit and test.ncd and test\_partial.bit
```

The Xilinx Platform Studio does not offer a comfortable way of configuring the generated bitstream. But the bitstream configuration for every single project can be manually configured in the file `bitgen.ut` in the project directory etc/.

In `bitgen.ut` the option `Persist` has to be changed appropriately and the option `ActiveReconfig:Yes` is added, `Security:None` should already be set accordingly.

Note: if the Persist switch is set the ICAP device is disabled!
B.5 Virtex II Pro Board Configuration

The software implementation of this master thesis has been written in connection with a Virtex II Pro FPGA board. Xilinx already stated on their website that the virtex II pro board is a mature product and is no longer fully supported with the Xilinx Platform Studio.

It has to be mentioned that the switches on the Virtex-II Pro Development System board have to be set correctly so as not to turn off the intern ICAP device. The two essential switches are the CONFIG SOURCE (SW9) and CONFIG SELECT (SW8).

![Figure B.1: CONFIG SELECT and CONFIG SOURCE](image)

The CONFIG SELECT switch is used to choose the partition of the flash card from which to load the bitstream. With three handles there are eight possible partitions to choose from.

The CONFIG SOURCE switch indicates the configuration device from which to load the bitstream.

Configuration:
- PROM/GOLDEN PROM VERSION: loads the built in testbench program
- PROM/USER: loads the bitstream from the flash card on the selected partition from CONFIG SELECT
- JTAG/USER: waits to receive the bitstream from the JTAG device

The available EDK Version 9.2 does not support the Virtex II Pro in the BSB. Whenever a system design for this board has to be created it has to be done from scratch. Xilinx still provides the EDK board definition files for EDK 10.1 SP3 for ongoing board support. When creating a new layout for the board those files have to referenced in the BSB which will let the BSB recognize the board as a XUP Virtex-II Pro Development System board revision C.

If legacy or deprecated cores have to be taken into the system design they will probably not be listed in the IP catalog. These cores can be listed again under Edit→Preferences→IP Catalog and IP Config Dialog

There is a checkbox "Display 'Available' IP Cores" (including legacy PLB/OPB cores) in IP Catalog. From now on legacy and deprecated cores are listed and can be included into the design.

B.5.1 9.2 board design

As the EDK does not support the Virtex Pro II board in its BSB the system design has to be made by hand. Most of the IP cores can be picked from the
catalogue and are instantly inserted in the design. Needed parts are:

- 2 PowerPC 405 instances 2.00.c opb hwicap 1.00.b
- jtagppc controller 2.00.c opb socket bridge
- proc system reset 1.00.a opb prr
- plb bus 1.02.a plb ddr (memory) 2.00.a
- opb bus 1.10.c plb bram interface controller 1.00.b
- plb2opb bridge 1.01.a bram block 1.00.a
- opb uartlite 1.00.a 2 dcm module 1.00.c
- opb sysace 1.00.c dcr bus 1.00.a
- opb interrupt controller 1.00.c opb2dcr bridge 1.00.b

- Add all previously mentioned cores to the design
- Interrupt and Signals: the System ACE and Uart interface have to be added to managed interrupts in the interrupt controller. The System ACE should have the highest priority otherwise the system will not boot.
- After the socket bridge instances have been added to the design open up the system.mhs and add the following lines right after BUS_INTERFACE and before END of opb_dcr_socket_0 peripheral:

  PORT ROPB_Clk = ROPB_Clk_0
  PORT ROPB_Rst = ROPB_Rst_0
  PORT ROPB_ABus = ROPB_ABus_0
  PORT ROPB_BE = ROPB_BE_0
  PORT ROPB_RNW = ROPB_RNW_0
  PORT ROPB_select = ROPB_select_0
  PORT ROPB_seqAddr = ROPB_seqAddr_0
  PORT ROPB_DBus = ROPB_DBus_0
  PORT RSln_DBus = RSln_DBus_0
  PORT RSln_errAck = RSln_errAck_0
  PORT RSln_retry = RSln_retry_0
  PORT RSln_toutSup = RSln_toutSup_0
  PORT RSln_xferAck = RSln_xferAck_0

Listing B.1: Socket Bridge External Ports

- Add the following lines between PARAMETER_INSTANCE and END of opb_PRR instance

  PORT OPB_Clk = ROPB_Clk_0
  PORT OPB_Rst = ROPB_Rst_0
  PORT OPB_ABus = ROPB_ABus_0
  PORT OPB_BE = ROPB_BE_0
  PORT OPB_RNW = ROPB_RNW_0
  PORT OPB_select = ROPB_select_0
  PORT OPB_seqAddr = ROPB_seqAddr_0
  PORT OPB_DBus = ROPB_DBus_0
  PORT Sln_DBus = RSln_DBus_0
  PORT Sln_errAck = RSln_errAck_0
  PORT Sln_retry = RSln_retry_0
  PORT Sln_toutSup = RSln_toutSup_0
  PORT Sln_xferAck = RSln_xferAck_0

Listing B.2: PRR External Ports
B.5 Virtex II Pro Board Configuration

B.5.2 10.1 board design

The design can be made with help of the BSB (Base System Builder) which facilitates the correct inclusion of interrupts and the correct configuration of the memory controller. The peripheral repository files are required to allow the BSB to configure the XUP board, those files are under:
http://www.xilinx.com/univ/XUPV2P/lib/lib_xupv2p_edk_10_1_sp3.zip

Create a new System and choose "Base System Builder wizard".

Choose a location to store the files.

The Project Peripheral Repositories are needed to bring in the design of the Virtex-II Pro development board. Enable it and look for the corresponding directory, something like: ./lib_xupv2p_edk_10_1_sp3/lib

Select to create a new design and select the XUP Virtex-II Pro Development System.

Select the PowerPC as processor, leave the rest on default.

The CPU frequency should be selected for either 300 Mhz or 100 Mhz. The bus frequency should be set to 100 Mhz. Additionally, enable the cache setup.

For the IO devices, check (enable) the following devices

RS232_Uart_1
  Peripheral: XPS UARTLITE
  Baudrate: 9600 baud
  Data bits: 8, Parity: None
  Use interrupt

Ethernet_MAX
  Peripheral: XPS ETHERNETLITE
  Use interrupt

SysACE_CompactFlash
  Peripheral: XPS SYSACE
  Use interrupt

DDR_SDRAM
  Peripheral: MPMC

every other device can be disabled as it is not required.

Choose at least 16 KB for the XPS BRAM IF CNTRL, all the 64 KB can be taken, however.

Choose to enable ICache and DCache for DDR_SDRAM.

The Memory test and Peripherals selftest can be disabled. All other settings can be left on default.

Click through until the design can be generated.

EDK now generates the instances of the system and all that remains to do is to turn on double buffering for the Ethernet_MAC: double click on "Ethernet_MAC" from the "System Assembly View" and enable Duplex Mode, Include Second Receiver Buffer, Include Second Transmitter Buffer
B.5.3 Patching a bitstream with software code

The final bitstream, static_full.bit, generated by planAhead cannot yet be loaded on the board and be run.

```
data2mem -bm implementation/system_bd
          -bt implementation/static_full.bit
          -bd bootloops/ppc405_0.elf tag ppc405_0
          -o b implementation/download.bit
```

Remember that the bmm file was specified in NGDBuild with the -bm switch. That file location is embedded in the NCD, and the BMM file must reside in the directory that the -bm switch pointed to when NGDBuild was run.

B.5.4 Device Tree Generation and Kernel Compilation

Before generating the Linux device tree, open the menu Software -> Software Platform Settings. Choose the device-tree as OS. In “OS and Libraries” set

```
console device: RS232_Uart_1
bootargs: console=ttys0 rw root=/dev/xsa3
```

Generate the device tree by clicking the menu Software -> Generate Libraries and BSPs. The device tree file will be created under the EDK project ./ppc40 5_0/libsrc/device-tree/xilinx.dts.

Rename the generated .dts (e.g. virtex405-mine.dts) and copy the file into <kernel sources>/arch/powerpc/boot/dts. The prefix “virtex405-” is required to signal the reference design to the Linux build process. The build process will do a few extras to make things come out right for Xilinx parts.

Kernel config:

Processor support:
  Processor type: AMCC 40x

Platform support
  Generic Xilinx Virtex board

Kernel options:
  Timer frequency (250 Hz)

Bus options:
  PCI support: No
  PCCard (PCMCIA/CardBus) support: no

Networking support:
  Networking options:
    IP: kernel level autoconfiguration: Yes (to enable NFS root)

Device Drivers:
  Block devices:
    Xilinx SystemACE support: Yes

Network device support:
  Ethernet(10 or 100Mbit)
    Xilinx 10/100 OPB EMACLITE support: Yes
  Ethernet(1000Mbit)
    Xilinx LLTEMAC 10/100/1000 Ethernet MAX driver: Yes
    Xilinx LLTEMAC PHY Support: GMII
Character devices:
  Serial drivers:
    Xilinx uartlite serial port support: Yes
    Support for console on Xilinx uartlite serial port: Yes

Filesystems:
  Second extended fs support: Yes
  Ext2 execute in place support: Yes

Network File Systems:
  NFS client support: Yes
  Root file system on NFS: Yes

B.5.5 Creating the SystemACE

The system.ace file in which the bitstream is encoded and which is required to boot up the board is generated with the xmd-tool. This is available with the Xilinx environment variables set. Create an option file for the xmd debugger, called xupGenace.opt, and fit the -hw and -elf parameter to the corresponding directory structure:

```
-jprog
-board user
-target ppc_hw
-bw ./implementation/download.bit
-elf ./linux-2.6-xlnx/arch/powerpc/\boot/simpleImage.virtex405-mine.elf
-configdevice devicenr 1 icode 0x1127e093 \
  ilength 14 partname xc2vp30
-debugdevice devicenr 1 cpunr 1
-ace system.ace
```

Listing B.3: xupGenace.opt file context

The xmd command generates the system.ace file with the help of the tcl script genace.tcl:

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

B.6 JNI Header Generation

The javah executable is used to generate the corresponding header file for native methods implemented by a Java class. Javah is invoked with the -jni parameter on a class with its fully qualified name. In Eclipse the External Tools Configurations can be used to facilitate the header file generation.

Open Run -> External Tools -> External Tools Configurations

Create a new Program and name it <ProgramName>

```
Location: /usr/bin/javah
Working Directory: ${workspace_loc:<Java_Project>/bin}
Arguments: -jni -d ${workspace_loc:<C_Project>/src} <class name>
```

Click Run -> External Tools -> <ProgramName>

The header file to implement the JNI native methods have been created under <C_Project>/src.