FACULTY OF PHYSICS AND ASTRONOMY

UNIVERSITY OF HEIDELBERG

DIPLOMA THESIS
IN PHYSICS

SUBMITTED BY
JOHANNES NICK MEIER
BORN IN
HAGEN, GERMANY

OCTOBER 2008

Development of a Framework for Dynamic Partial Reconfiguration serving the Object Oriented Hardware Programming Language POL

This diploma thesis has been carried out by **Johannes Nick Meier** at the

Kirchhoff Institute for Physics

under the supervision of

Prof. Dr. Udo Kebschull

Entwicklung eines Frameworks für dynamische partielle Rekonfiguration zur Unterstützung der objektorientierten Hardware-Entwicklungssprache POL

Obwohl es seit den neunziger Jahren möglich ist, feld-programmierbare Hardware zur Laufzeit neu zu konfigurieren, wird diese Technik in kommerziellen Anwendungen überhaupt nicht und im akademischen Bereich nur sehr selten angewendet. Dies liegt zum einen daran, dass es bisher keine Anwendungsbeispiele gibt, die eine Nutzung dieser dynamischen Rekonfigurationstechniken zwingend notwendig machen. Zum anderen gibt es kein allgemein zu nutzendes Framework, das die Funktionalität der dynamischen partiellen Rekonfiguration kapselt. Damit ist die Entwicklung eines Hardwaredesignes, das DPR verwendet, immer eine Insellösung. Eine Wiederverwendung der DPR-spezifischen Designteile ist nur selten möglich. In dieser Arbeit wird eine erste Version eines solchen Frameworks auf der Hardwareseite vorgestellt. Mit den hier entwickelten Hardwarestrukturen soll eine allgemeingültige Umgebung für die Entwicklung von DPR-fähigen Anwendungen bereitgestellt werden. Damit soll die Wiederverwendbarkeit der DPR-Funktionalität für verschiedene Projekte und Designes garantiert werden und dem Entwickler die sehr speziellen und aufwendigen Entwicklungsschritte, die nur für ein DPR-fähiges Design nötig sind, abgenommen werden. Als Beispielimplementierung wurde eine Audioanwendung auf einem Xilinx Evaluationsboard (ML505) realisiert.

Development of a Framework for Dynamic Partial Reconfiguration serving the Object Oriented Hardware Programming Language POL

Since the nineties it is possible to reconfigure the field programmable hardware during runtime. Nevertheless these technologies are not recently used in commercial applications and are seldom used in academic research. This is caused by the fact, that most common application do not need DPR technologies mandatory. Another problem is, that no general hardware framework exists, which encapsulates the functionality of DPR. Therefore, the development of a DPR using application is always a stand-alone solution and in most cases a reuse of the DPR specific part of the design is not possible. In this thesis a first attempt to create such a framework, serving the hardware components, is presented. The developed hardware structures will use DPR and provide a general framework for the development of applications. This will improve the re-usability of the DPR functionality of different projects and designs and the developer is liberated from the DPR specific development steps. As an example implementation, a simple audio application is realised on a Xilinx evaluation board (ML405).

Contents

List of Figures

List of Tables

1 Introduction

This diploma thesis is carried out at the Reconfigurable Hardware workgroup at the Kirchhoff Institute for Physics. The workgroup focuses different themes like fast boot, low power consumption, radiation tolerance and software-to-hardware compilation. The common denominator of all these topics is the usage of the dynamic and/or partial reconfigurability of Xilinx FPGAs[1]. For more information about the topics of this workgroup, see [Con].

This thesis is part of the software-to-hardware compiler group. This group focuses the translation of dynamic software to also dynamic hardware. Most of the current software-to-hardware compiler projects try to find data flow parallelism in a sequential program description and use this parallelism to generate parallel running hardware components. Although this approach is sensible and suggestive, the usage of the sequential description is an additional overhead, if the application was formally described in parallel.

1.1 Motivation

In the development of a software design, object-oriented programming (OOP) and the thread concept, are used very frequently. In a design using OOP, the variables and the functionality are encapsulated into objects, which are individual and active parts of the application. Where as, in a design based on procedural programming, the variables are absolutely passive and the programme flow is sequential. The included thread concept enables the objects to act as independent application parts and to work in parallel. With the OOP and the thread concept, it is also possible to explicitly use the parallel architectures provided by modern CPUs.

The focus of the software-to-hardware translator, developed in this workgroup, is to describe an application in a high level programming language, using OOP and threads, and to translate the software objects into parallel hardware objects. The software application will explicitly be developed with a parallel description, and will be directly translated into a equivalent and also parallel hardware realisation, without using the sequential execution data generated by a software compiler. The translator will use the intrinsic parallelism of the software objects and will also be capable to handle the dynamic instantiation of these objects. The dynamic instantiation of hardware objects will explicitly

[1]Field Programmable Gate Array

require the use of DPR[2] technologies. For the description of these applications a new programming paradigm for hardware is needed.

In the software world, the objects are instantiated during runtime. This kind of dynamic runtime development of the application cannot be mapped into hardware using a static hardware environment (e.g. using ASICs[3] for the hardware realisation, the design is fixed, when the ASICs are produced). With the progress of the DPR technology of Xilinx FPGAs the dynamic manipulation of logic on the FPGA after the development process is now possible.

The new programming paradigm for a software-to-hardware language has two main requirements. The application is described with different objects, that are truly running in parallel and can be instantiated during runtime. Many high level programming languages include the idea of OOP, dynamic instantiation of objects and a thread concept, but the first attempt in this workgroup is done with the programming language POL, which consists of a subset of JAVA, also see [Gru09]. That means, it's syntax and semantic is based on JAVA, but not every statement that can be written in JAVA and is sensible for software, can be used in POL. This is owed to the fact, that POL is used to describe hardware. On the other hand, POL is extended by special scheduling functionality, needed for the dynamic instantiation of hardware. What POL will do for the developer in detail is described in section 2.3 and more general in [Nor08]. Other languages like C++ for a software attempt, or Verilog and VHDL[4] for a hardware attempt, can also be used, but for the first evaluation of the principles of the programming paradigm, JAVA as the basis was chosen. JAVA does not include pointers in its programming concept, and that is it's most important advantage concerning hardware. The hardware framework used for the execution of the translated hardware will use a pipelining structure, to encapsulate any reconfiguration times during run-time. Therefore, it is in principle not possible to map the pointer concept into a high-performance hardware realisation, if the pipelining structure of the hardware framework should persist. Using JAVA, it is not necessary to prevent the pointer. The programming language JAVA also provides a very neat concept for the dynamic instantiation of objects, the *new()* command. For more information about JAVA see [JAV].

Most of the projects, that also focus on OOP, usually use the FPGA as a co-processor. Application parts, that are predestined for a hardware execution, are sourced out to the FPGA. That can improve the performance of the application, but the communication between the processor and the FPGA is still sequential. The instantiation of application parts in hardware is done for a mere speedup, but not to run several objects truly in parallel.

[2]Dynamically Partial Reconfiguration
[3]Application Specific Integrated Circuit
[4]Very High Speed Integrated Circuit Hardware Description Language

The software-to-hardware compiler developed in this workgroup introduces a new programming paradigm for the description of hardware applications. The hardware is described with parallel running objects, that can be instantiated during runtime. The POL-to-hardware compiler is equipped, with an explicit DPR extension, to serve the dynamic of the thread concept. The DPR extension includes a hardware framework, which encapsulates the DPR functionality and the parallel communication structure. The compiler can translate the POL application into concrete hardware, which can be executed on a FPGA.

In this work the hardware environment necessary for the hardware execution of POL is developed. The framework of the hardware description language POL should provide every low level functionality to the application, that is needed in a hardware environment, concerning the DPR technologies. This framework is called communication matrix, because the main problem for a hardware realisation of common streaming problems, is the capability of a parallel communication structure, established in a dynamic environment. The main challenge of such a dynamic hardware environment is the communication between the hardware objects. In the software environment the threads are only running quasi in parallel. Therefore it is not interfering the functionality, that only one thread can work on the processor and especially on the bus at the same time. But this implies a serialised communication flow. The communication matrix uses a completely message based communication structure, which also realises a pipeline structure. This pipeline structure is used to improve the performance of the communication matrix. The resources of the FPGA are finite, but for dynamic instantiation more resources can be requested, than are available. The communication matrix can handle this over-allocation with DPR. The DPR technique enables the communication matrix to exchange parts of the application logic. Therefore it can handle more logic, than is fitting into the FPGA. On the other hand any reconfiguration will take some time. These reconfiguration times have to be masked by the communication matrix. Furthermore the location of the functional logic is no longer fixed. The scheduler is capable to freely configure the hardware objects in any reconfigurable region of the FPGA. Therefore the communication between the hardware objects must be updated, due to the possible new locations of the hardware objects. The communication matrix includes also a scheduler for the reconfiguration and can store all the streaming data and also the content data of any scheduled hardware object. The main improvement of the communication matrix is the fact, that all the functionality described above is hidden from the developer. The developer has in principle not to think about the scheduling and the realisation of the dynamic instantiation of the hardware objects; and most important, it is insignificant for the application that the data messages are exchanged in a pipelining structure.

1.2 Outline

This thesis is divided into nine main chapter. In chapter 2 the basic principles are described, which are necessary to understand the technical background about what is done in this work. Chapter 3 gives an outline to previews attempts and different approaches for a solution of the problem of a software-to-hardware compiler. In chapter 4 the requirements and the principles of the new hardware framework are described and different approaches for the realisation are discussed and evaluated against each other. This evaluation results in a precise definition of the communication matrix, which is noted in chapter 5, as a summarisation of the made decisions. This definition of the communication matrix is evaluated against the needed performance and resource allocation, with an explicit implementation of the communication matrix, which is described in chapter 6. The results of this evaluation are presented in the following chapter 7. As the DPR framework including the communication matrix is not completely developed and therefore cannot serve any requirement of POL, some future work is left. The developments, needed for a full working POL hardware framework, are listed in chapter 8. The last chapter (9) contains a review of the presented work.

2 Basic Principles

This chapter gives a short overview to the main hardware, software and tools used in this work. These descriptions are in principle not necessary to understand the main idea developed in this thesis (chapter 5), but are recommended to gain deeper insight to the concepts presented and to the realisation and the exemplary implementation of this work (chapter 6).

2.1 Field Programmable Gate Array

There are many possibilities to realise reconfigurable hardware. A good introduction to reconfigurable architectures is given in [Bob07]

In general every binary logic function can be represented by **and**, **or** and **not** gates. So reconfigurable hardware only needs to connect the inputs and the negated inputs via an **and** matrix and a **or** matrix to the outputs. The intersection of these matrices can be programmed to be conductive or insulating. With enough wires and big enough matrices any logic function can be realised. A better realisation is done with Look-Up-Tables (LUT). A LUT stores the output pin configuration depending on the input pin configuration.

Xilinx FPGAs are composed of three parts. The main logic resources, a general routing matrix and inputs and outputs. The logic resources are called Configurable Logic Blocks (CLB) and are connected to the routing matrix via a Programmable Switch Matrix (PSM). The input and output blocks (I/O) are also connected to the routing matrix and can also be configured. The configuration of the CLBs, the I/O blocks and PSMs are stored in SRAM (Static Random Access Memory) blocks. These memory blocks depend on the electric power supply. If the power supply is switched off the content of the SRAM blocks is lost. So the FPGA has to be completely configured after every power up.

In Xilinx Virtex-4 devices the CLBs are built up by four interconnected slices, see figure 2.1. In principle a slice contains flip-flops and LUTs to realise the logic functions. One CLB of a Virtex-4 device contains four slices with each eight LUTs, eight flip-flops, eight multiplier or *and* components and two arithmetic or carry-chains. So in general, the number of LUTs and flip-flops define the size of a FPGA and the magnitude of the design suitable for the device. A short list of available resources of Virtex-4 devices is given in table 2.1.

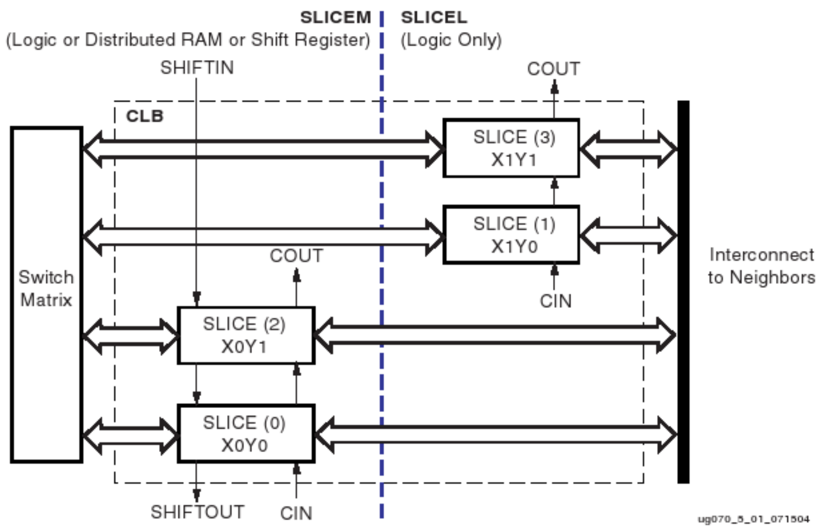


Figure 2.1: Content of a Virtex-4 CLB. [Xil08c]

Additionally, the Virtex-4 FPGA also includes specialised components, so called primitives. These are for example, clocking components like Digital Clock Manager (DCM), fast random access memory (BRAM), multiplier (MULT18x18). Some devices also include a complete embedded CPU (PPC). For more information about the Virtex-4 device see [Xil08c].

2.1.1 Configuration Interface: SelectMAP

The Virtex-4 devices have 3 different interfaces to perform configuration. Only the SelectMAP interface will be used for dynamic partial reconfiguration, as the internal access to the configuration pins is realised with this interface. Admittedly the initial configuration occurs by the JTAG interface.

This chapter will give an brief overview to the SelectMAP interface and the general configuration process. For more details of the SelectMAP interface, for the JTAG interface or any other FPGA configuration interface see [Xil08b].

The configuration memory of a FPGA is accessed by frames. So any operation is always performed on a complete configuration frame. The amount of configuration frames of a FPGA scales with the size of the device. Table 2.2 shows the configuration frame counts of Virtex-4 devices.

Device	Number of Slices	Number of LUTs	Number of Flip-Flops
XC4VLX15	6,144	12,288	12,288
XC4VLX25	10,752	21,504	21,504
XC4VLX40	18,432	36,864	36,864
XC4VLX60	26,624	53,248	53,248
XC4VLX80	35,840	71,680	71,680
XC4VLX100	49,152	98,304	98,304
XC4VLX160	67,584	135,168	135,168
XC4VLX200	89,088	178,176	178,176
XC4VSX25	10,240	20,480	20,480
XC4VSX35	15,360	30,720	30,720
XC4VSX55	24,576	49,152	49,152
XC4VFX12	5,472	10,944	10,944
XC4VFX20	8,544	17,088	17,088
XC4VFX40	18,624	37,248	37,248
XC4VFX60	25,280	50,560	50,560
XC4VFX100	42,176	84,352	84,352
XC4VFX140	63,168	126,336	126,336

Table 2.1: Virtex-4 Resources

The SelectMap interface has a 8 bit bi-directional data bus for configuration and read back. Where reconfiguration means, to write a special file to the FPGA which triggers the FPGA to realise the expected hardware. Read back is the reversed process and enables the developer to read out the hardware configuration instantiated on the FPGA. The interface is displayed in figure 2.2

Also important for reconfiguration is the general configuration process of the FPGA illustrated in figure 2.3.

The first step is to supply the FPGA with power. The Virtex-4 needs a 1.2 volt source.

After that the configuration memory is cleared sequentially in step two, and in step three the FPGA begins to sample the configuration pins. Now the configuration process becomes more interesting because for any dynamic partial reconfiguration the next steps are also crucial.

With the synchronisation, the configuration device is disposed to perform a reconfiguration. After that, the target device is checked against the destination device of the bitstream. This prevents the configuration of the FPGA with a bit file generated with the wrong format.

Now the configuration data is loaded to the configuration memory. The data can be relevant for the whole chip or only a certain part of it. The uploaded configuration data can

Device	Configuration Frames	Device Frames	Configuration Array Size
XC4VLX15	3,600	3,740	147,600
XC4VLX25	5,928	6,162	243,048
XC4VLX40	9,312	9,688	381,792
XC4VLX60	13,472	14,008	552,352
XC4VLX80	17,720	18,430	726,520
XC4VLX100	23,376	24,324	958,416
XC4VLX160	30,720	29,460	1,259,520
XC4VLX200	39,120	37,500	1,603,920
XC4VSX25	6,940	7,380	284,540
XC4VSX35	10,410	11,070	426,810
XC4VSX55	17,304	18,408	709,464
XC4VFX12	3,600	3,848	147,600
XC4VFX20	5,488	5,848	225,008
XC4VFX40	10,296	10,956	422,136
XC4VFX60	15,976	17,016	665,016
XC4VFX100	25,170	26,830	1,031,970
XC4VFX140	36,444	38,868	1,494,204

Table 2.2: Virtex-4 Frame Count, Bitstream Size in 32 bit words

be verified by the FPGA to prevent any transfer errors during the upload of the configuration data. As any application running with the communication matrix will depend on reconfiguration to work, this verification is very important to detect any errors compromise the correct function of the application.

At last the start up sequence is performed. For the dynamically partial reconfiguration, the start up sequence is not performed, because it is not sensible to reset the static design parts of the FPGA. This is also important for the dynamic reconfiguration of the dedicated parts of the FPGA. The static part of the design is unaffected by the partial reconfiguration but for the dynamic part a special start up sequence is needed to transfer the dynamic area into a reasonable state. This start up sequence is not included in the Xilinx tool flow and must be provided by the framework of the DPR system.

For a more detailed description of the configuration sequence see also [Xil08b].

2.1.2 Bitstream Composition

The FPGA is configured by a configuration interface, e.g. the SelectMAP interface, with a special data format, called bitfile. This configuration file contains the instructions for the configuration control logic and data for the configuration memory. Although the bitfile is mostly independent from the used configuration interface, for dynamic partial

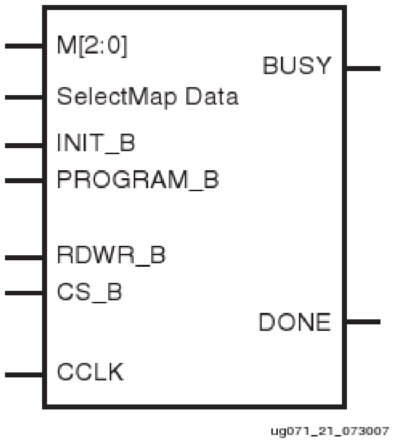


Figure 2.2: SelectMAP Configuration Interface. [Xil08b]

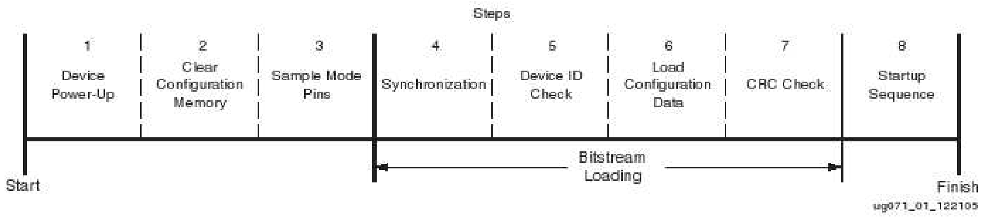


Figure 2.3: Configuration process of a Virtex-4 device. [Xil08b]

reconfiguration a special kind of bitfile is used, the partial bitfile. As the name suggests this bitfile includes only some part of the configuration data, only a dedicated part of the FPGA is arranged to be configured.

Both bitfiles have a header, which provides information about the used FPGA, the file name, the creation date and similar things. This is not important for the partial reconfiguration. The rest of the bitfile consists of two types of 32 bit packets for the instructions and of course 32 bit data words. The first one is the normal instruction word for the configuration interface, the second one is used when the configuration data word count is too long for a type one packet. The type two packet always follows a type one package, which contains the address for the data.

The type one package consists of the type information, the operation code, the register address and the word count, see table 2.3.

The type two package only needs the register address and the operation code, as the register address is already included in the preceded type one package. The format of the type two package is displayed in table 2.4.

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRxxxxx	RR	xxxxxxxxxxx

Table 2.3: The type one packet header is used for the normal data words, if the word count is less than 2048 words.

Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxx

Table 2.4: The type two packet header is used, if more then 2048 words have to be written or read.

To understand the partial bitfile and the dynamic partial reconfiguration flow, four of the configuration registers are needed. They are listed in table 2.5 and described below.

Reg. Name	Read/Write	Address	Description
CRC	Read/Write	00000	CRC register
FAR	Read/Write	00001	Frame Address Register
FDRI	Write	00010	Frame Data Register, Input
CMD	Read/Write	00100	Command Register

Table 2.5: The Configuration Register needed to understand the partial bitfiles used in this work.

The CMD register is used for the commands to the ICAP needed to proceed a reconfiguration. The FAR register and the FDRI register are used to address the frames and overwrite the frame data with the new values. With the CRC register the uploaded configuration data is verified.

To perform a dynamic partial reconfiguration the following four commands are used, listed in table 2.6.

The first command (WCFG) is used to write the configuration data of the dynamic area into the configuration memory. The RCRC command is most important as we only want to change part of the design currently running on the FPGA. The CRC register has to be reset because the partial bitfile is not equivalent to the full bitfile. It does not contain any resources associated to the static part and the containing dynamic part is different to the one currently running. So a CRC check with the old value would fail.

The desynchronise commend (DESYNC) is used to tell the device that the configuration is performed and the configuration pin should be ignored till the next reconfiguration. The partial reconfiguration should leave the non dynamic part of the design unaffected,

Comand	Code	Description
WCFG	0001	Write Configuration Data: used prior to writing configuration data to the FDRI.
RCRC	0111	Reset CRC: resets the CRC register
DESYNC	1101	Reset DALIGN Signal: used at the end of configuration to resynchronise the device. After de-synchronization, all values on the configuration data pins are ignored.
LFRM	0011	Last Frame: Deasserts the GHIGH_B signal, activating all interconnect. The GHIGH_B signal is asserted with the AGHIGH command.

Table 2.6: The command codes for the CMD register used for reconfiguration.

so the I/O pins of the FPGA should not go to high impedance. Therefore the LFRM command is used at the end of the partial bit file.

Finally there are the frame data included into the partial bitfile. Although PlanAhead takes care of these data, it is helpful to understand the frame addressing of the Virtex-4 device, because the content data of the BRAMs will be written without the help of PlanAhead.

The configuration data of the FPGA is separated into different frames, which can be addressed separately. The chip is divided into an upper and a lower part. Both halves are divided into different rows, the number of rows increases with the chips size. Again, the rows are divided into major frames. The major frames are the smallest entity of configuration data that can be changed, although they are also divided into minor frames. The minor frames are needed to keep control of the configuration data stream, therefore the minor frames have a fixed length of 41 words.

All the described separations of the configuration data can be seen in the structure of the Frame Address Register displayed in table 2.7. The half of the chip is selected, the block type, the row and the minor and major frames.

Reserved	Half	Block Type	Row Address	Major Address	Minor Address
[31:23]	[22]	[21:19]	[18:14]	[13:6]	[5:0]
RRRRRRRRR	x	xxx	xxxxx	xxxxxxxx	xxxxxx

Table 2.7: The FAR register is used to address the configuration frames of the BRAMs.

POL will serve the dynamic instantiation of hardware objects. For this purpose, it is recommended, that the BRAM content is saved, and can be written back to the FPGA later. A detailed description of the read back usage of the BRAM content is given in the POL subsection or in chapter 8.2.

For the readout of the BRAM content the FAR register of the SelectMAP interface is also used. The block type selects the type of the configuration data. For example, the normal logic, the BRAM content and the Bram interconnect frames are independently addressed in the frame register of the configuration register. So for the read back or write of the BRAM content the block type *010* is selected.

The read back can be performed in a single minor frame, but to write the BRAM content the whole major frame must be written. Unfortunately, a major frame contains four BRAMs in a Virtex-4 FX20 FPGA. That means, that the BRAM memory of the chip can be allocated by different framework functionality, only in blocks of four BRAM. The configuration of the BRAM content of one BRAM in a major frame disables all the other BRAMs associated to the major frame. The explicit addressing of the BRAM content frames of the Virtex-4 FX20 is shown in picture 2.4.

The aggregation of four BRAMs in one content frame is another placement constraint for the dynamic areas providing the hardware objects. The dynamic area has always to enclose a full BRAM frame, and so all four BRAM. Otherwise the remaining BRAM can not be used by the static part or any other dynamic area, because it will be disabled during the reconfiguration of the actual dynamic area.

There is another problem with the read back of the BRAM content. In every minor frame additional bits are included, which corresponds to the amount of BRAMs included into the major frame. These additional bits are present in the data frame and also in the PAD frame. Before the content data of the BRAMs can be written to the FPGA, the additional bits have to be removed from the data. If these bits occur in the PAD frame or in the date the write back operation will fail. Fortunately, the position of these additional bits is equal for every minor frame, as long as the BRAM content frames are located in the same chip half. At the chip's half border the bit position will shift by 48 bits.

For the read back of the objects content the additional bits have to be extracted. Not only the bit position of the additional bit changes at the half border, but also the bit encoded data of the BRAM content. That can be a problem if the different dynamic areas are placed on both halves of the chip. If the hardware object has to cross the half border, the whole content data has to be shifted by the 48 positions, before it can be written to the FPGA. The main advantage of POL and the operating hardware framework is, that this complex read back operations are encapsulated from the developer.

The next section describes the real primitive, which provides the ability of the self-configuration of the Xilinx FPGAs.

2.1.3 Internal Configuration Access Port

Any FPGA out of the Virtex-II Pro, Virtex-4 and Virtex-5 series provide the capability to access the configuration interface from the inside of the FPGA. This is done by the

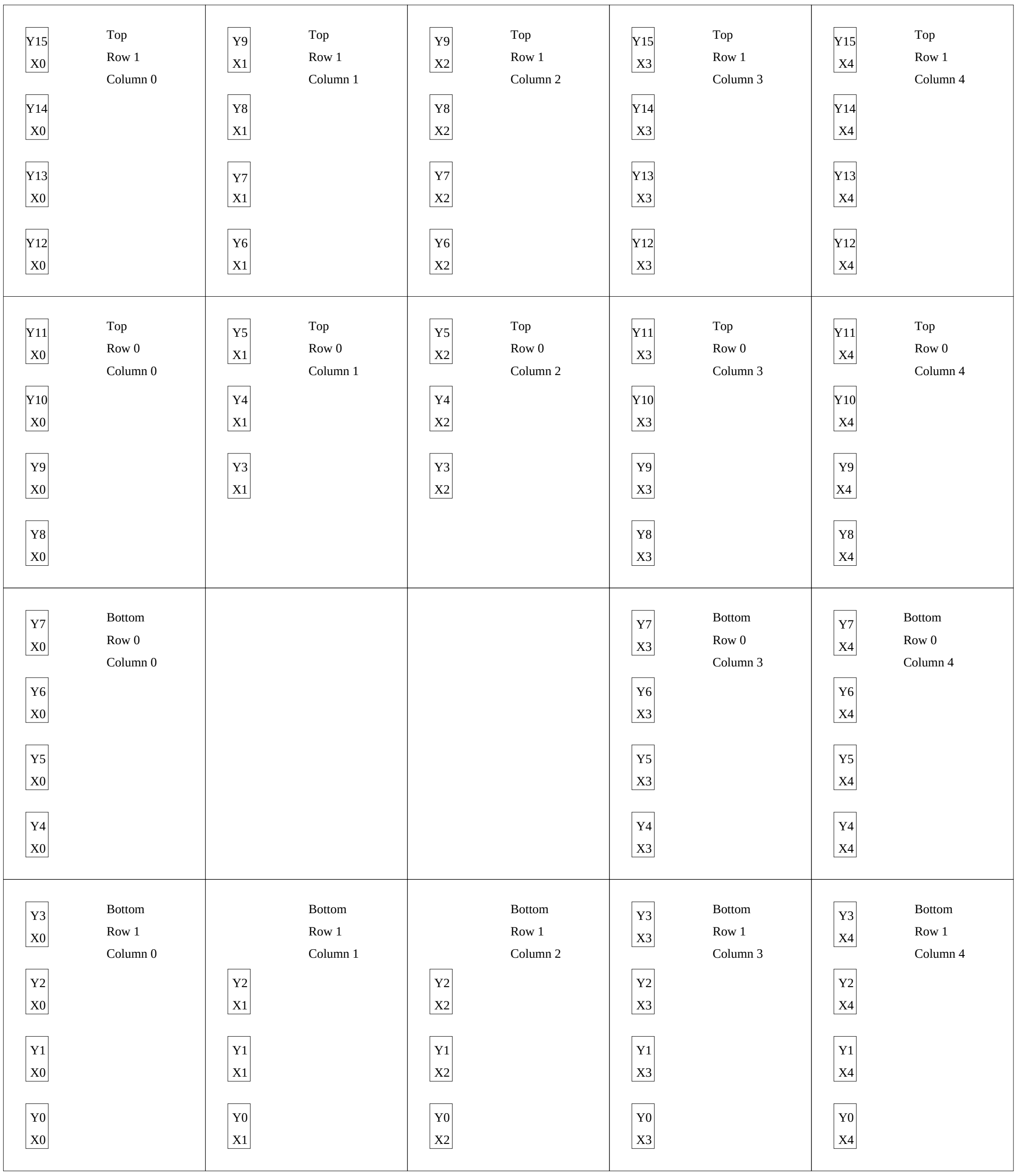


Figure 2.4: This is the BRAM arrangement of the Virtex-4 FX20 FPGA. The position of the BRAMs on the chip and the addresses of the BRAM content frames in the configuration data are displayed.

Internal Configuration Access Port primitive (ICAP). The ICAP uses a similar interface as SelectMAP and the same interface signals. Only the data bus is separated into read and write buses.

The ICAP needs a clock (CLK) and provides a chip-select signal (CS), a read/write control signal (RD) and two data buses, one for ”read” operations and one for ”write” operations. The data bus width can be 8 bit or 32 bit.

The ICAP primitive must be placed in a static part of the design. The logic controlling the ICAP may not be changed during the reconfiguration, if the functionality should remain.

Virtex-4 devices and also Virtex-5 devices have two ICAPs, one in the top half of the FPGA and one in the bottom half. Only one ICAP can be active at the same time. The ICAP on the top half is activated by default after the reconfiguration of the FPGA.

For more information about the ICAP see [Xil08b].

2.2 Dynamic Partial Reconfiguration

As mentioned in chapter 2.1 it is possible to change the logic executed on a FPGAs. This case is called reconfiguration.

Furthermore it is possible to reconfigure only a part of the logic provided on the FPGA during runtime, this is called partial reconfiguration. Finally Xilinx FPGAs can perform a reconfiguration during runtime without any glitches. Witch means for partial reconfiguration a fractional change of the functionality, while the unchanged part of the logic remains unaffected. What we get is Dynamic Partial Reconfiguration (DPR), witch enables us to change given parts of the logic, while allowing the rest of the application to work independently.

DPR:
With dynamic partial reconfiguration, it is possible to change given parts of a running hardware design without interfering with the remaining hardware. The system remains unaffected, apart from the changed logic.

It is also possible to access the configuration pins from within the FPGA. This enables a system to change its own logic during runtime without any interference from outside the FPGA. Therefore the hardware design must be arranged differently. The next section describes the different design rules and the tools needed for the use of DPR.

2.2.1 DPR Tool Flow

There are many different approaches to create a hardware design including DPR. The easiest one, at the moment, is described in this section.

There are mainly two major steps to create a design using DPR technologies. The first one concerns the synthesis of the hardware design. This is done by the Project Navigator and XST from Xilinx. The rest is encapsulated by PlanAhead, namely the translating process, the mapping, the placement, the routing and for DPR, the merge process. PlanAhead itself only provide the needed floor planning of the dynamic and static areas. For the remaining processes the ISE tools are used. The results of the tool flow are two full bit files, one including the last tasks and one with empty task areas and partial bit files for every task.

The ISE tools need a special patch, which enables the tool to generate DPR capable hardware design. For the partial reconfiguration special routing constraints have to be obeyed. For example, only routing logic of the static parts can be present in the dynamic areas, nothing else. This patch is currently not available for the ISE version 10.1 and the Virtex-5 devices, but this will change with the ISE version 11.

There are some design rules for the developer that must be obeyed, when using PlanAhead and DPR. The design has to be divided into a static and a dynamic part in the top level of the VHDL code, as shown in picture 2.5. The complete dynamic part must be reconfigured and the static part must remain. Of course it is possible to define more than one dynamic part on the FPGA.

Both parts are connected with special predefined hardware components called busmarcos. PlanAhead can detect the transition between the static part and the reconfigurable part with these macros. The busmacros also ensure that the interface of the dynamic part remains steady during reconfiguration.

Due to the width and direction of the busmacro, there are two different types of busmacros which can be used in a DPR design. The busmacros have the ability to be enabled and to be disabled. They can also be synchronised or asynchronous. The synchronised busmarcos forward the data by every clock cycle whereas the asynchronous busmacros forward immediately.

The enable/disable functionality is required because of mixed states between the old logic and the new logic of the dynamic part during reconfiguration. The FPGA is reconfigured serially from left to right in steps of minor frames. The behaviour of the dynamic part containing this mixed state is not predictable, manageable or even deterministic. Therefore it is necessary to separate the dynamic part from the remaining system completely. With the busmacros, it is possible to prevent any interference between the whole System and the dynamic part during reconfiguration. The decision between synchronous and asynchronous busmacros is only important when the timing is crucial.

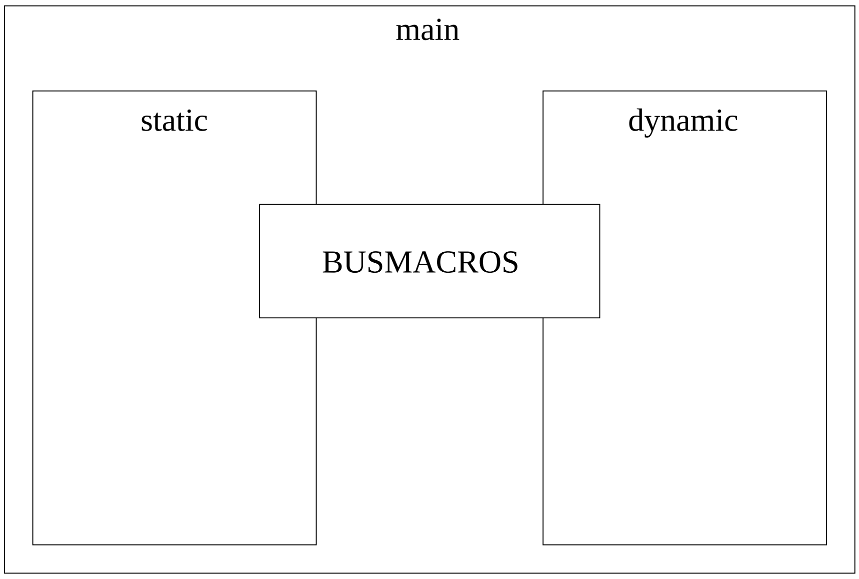
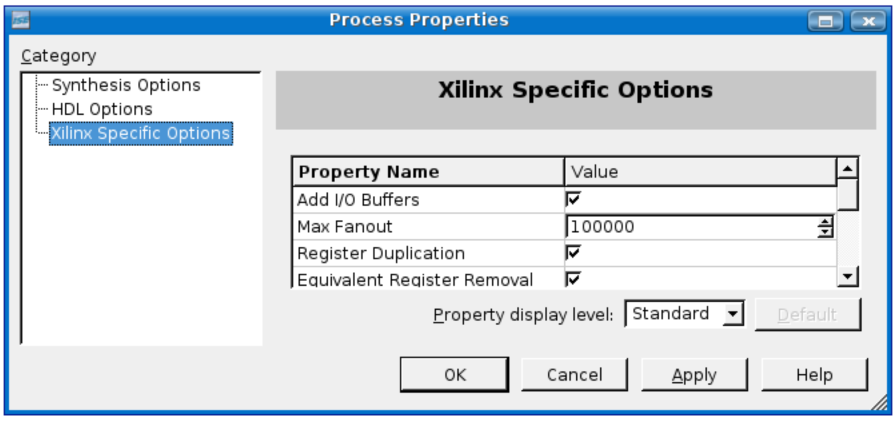


Figure 2.5: The ISE design is divided into a static part and a dynamic part. Both are connected via busmacros inside of the main VHDL component.

The next step, after generating a suitable VHDL design, is the synthesis with the Project Navigator. At first the static subcomponent and the dynamic subcomponents of the VHDL design have to be synthesized. The components are marked as top module and synthesised one after another. For PlanAhead every dynamic module must have the name given in the top level component, so the dynamic modules cannot be numbered and must be stored in different folders. It is recommended that the XST does not add any input buffer or output buffer as these components are internal. This is done by the Synthesis Properties. With the Xilinx Specific Option slide one can choose whether the XST adds any I/O buffer see figure 2.6.

After every synthesis activity the resulting NCD file has to be backed up from the project directory, because the project must be cleaned afterwards.

Finally the top module is selected. Any other VHDL component is removed from the project. The top module component only sees black box modules for the dynamic and static parts and of course the busmacros. This enables PlanAhead to exchange the dynamic modules more easily. At this point one has to act with caution, concerning the I/O buffer. The inclusion of the I/O buffer, into the hardware design, is not done within the top module anymore and we prevent the XST tool from adding any I/O buffer, when processing the low level modules. We can now let the XST tool add the I/O buffer. It will guess which buffer is needed by the *IN* and *OUT* statements in the port list of the top module. But, if any bidirectional pin of the FPGA or any bidirectional signal between the black box modules is used, The XST tool cannot handle the signal properly and will simply add an output buffer. For example the DDR ram needs such pins. The XST will not



Property Name	Value
Add I/O Buffers	☑
Max Fanout	100000
Register Duplication	☑
Equivalent Register Removal	☑

Figure 2.6: With the Xilinx Specific Option slide one can choose, whether to allow the XST to add any I/O buffer.

add the right buffer to the bidirectional signals, if it sees only black boxes and does not know anything about the use of these signals, but this can be performed by PlanAhead. Unfortunately it is necessary to add the remaining buffer by hand into the top module. If no bidirectional signal within the top module is used, the XST can add the I/O buffer on its own.

For the further steps with PlanAhead, the following files are recommended. The NGC files of the top module, the static part and every realisation of the dynamic part. That means of course every NGC file used in the project must be related to the static part or one of the dynamic parts of the design. An UCF file for the used board and last, but most important, the NMC files of the used busmacros.

Sorting the files generated with XST into a folder structure like this, makes the use of PlanAhead much easier. The following order is recommended, when selecting for PlanAhead.

PlanAhead folder structure:

- top
- static
- busmacros
- dynamic
- UCF

PlanAhead is used to place the DPR regions and the connections with the static part of the FPGA. But the most important part is to generate the bit files of the static and dynamic parts of the design. After that PlanAhead uses the tool for the mapping, the place and route and the bitfile generation of ISE, and also merges the different bit files of the static and dynamic areas. Which means it can detect the differences between the designs by including any of the tasks. To enable the ISE tools to generate any partial bitfile, a special patch for DPR is needed. This patch modifies the ISE tool, especially the routing tools, to achieve the special DPR requirements. The partial areas of the design, dedicated to include the dynamically instantiated logic, are not allowed to contain any other logic, for example of the static design. Only routing logic is allowed, which enables the partial areas to border special primitives, like I/O pins, of the FPGA.

After generating the new project, the first step is to tell PlanAhead that this project is meant to use DPR. This can only be done with the command line.

hdi::pr setProject -name "<project name>"

Now the dynamic module must be set as reconfigurable with a right click, and the associated *p_plock* has to be placed. It is recommended that the resources covered by the *p_plock* satisfy the requested resources of the dynamic module.

Running the DRC check, reveals any primitive left to be placed by hand and where to place it. If the position of the busmacros and other primitives, like a DCM, included into the design, are not already constraint in the UCF file of the ISE project, they must be placed in the PlanAhead GUI by hand. Another problem can be the difference between the chip half and the clock regions. If the BUFG for the clock domain crossing between the static and dynamic parts of the design, is not placed in the correct half of the chip, the DRC check will show a warning or even an error. It is not recommended to place the BUFG in the ISE project, because PlanAhead is used for the floorplanning.

If the DRC does not report any further errors, and everything is suitable, the design can be mapped, placed and routed with the PlanAhead runs, which also only call the patched XST tools. At first the static run must be performed. After that, every dynamic module must be activated with a right click and processed. It may be necessary to repeat the run of the static module with every activated dynamic module beforehand. The last step is to start the merging process with a right click on the last activated (default) dynamic module.

As mentioned above the results are some partial and full bit files; One static bit file with empty dynamic areas, one full bit file containing the static part and a default dynamic module for every dynamic area and partial bit files representing all the dynamic modules are generated.

2.3 Parallel Object Language

The Parallel Object Language (POL) is a new programming paradigm for hardware designs and uses explicit, object oriented techniques and the thread concept. POL is also a Java pre-compiler, as the POL code can be translated into an executable hardware design or into a JAVA software version, that can be used to simulate the behaviour of the application within the real communication matrix. It uses a subset of the Java syntax, but includes more hardware specific constructs, which are translated for software simulation. The software simulation uses the Java thread concept explicitly to simulate the hardware behaviour.

The application is described with different classes. Each of them inherits from a special class, to obtain all the functionality required to be a parallel hardware object. This class is called *ParObj* and inherits from the JAVA *Thread*-class. This enables all the instances to run in parallel, whether in the software simulation or the hardware execution. They also can be instantiated or destroyed during runtime by the application. The functionality of the *ParObj* class, which is performed continuously, is described in one single method, the *calc()*-method. Therefore, this method is empty and must be overwritten by the subclass, representing the hardware object. To get a more detailed overview of POL, a very simple data flow is considered in picture 2.7.

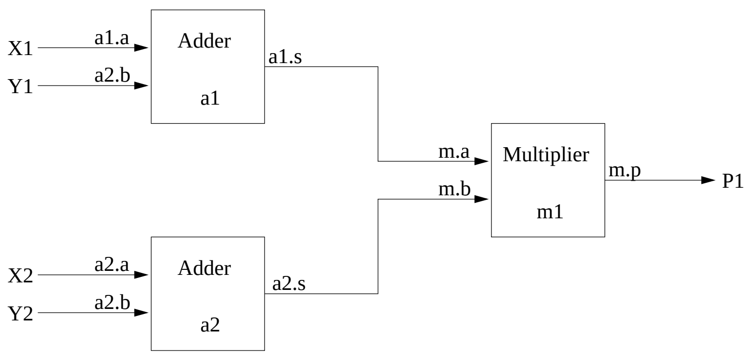
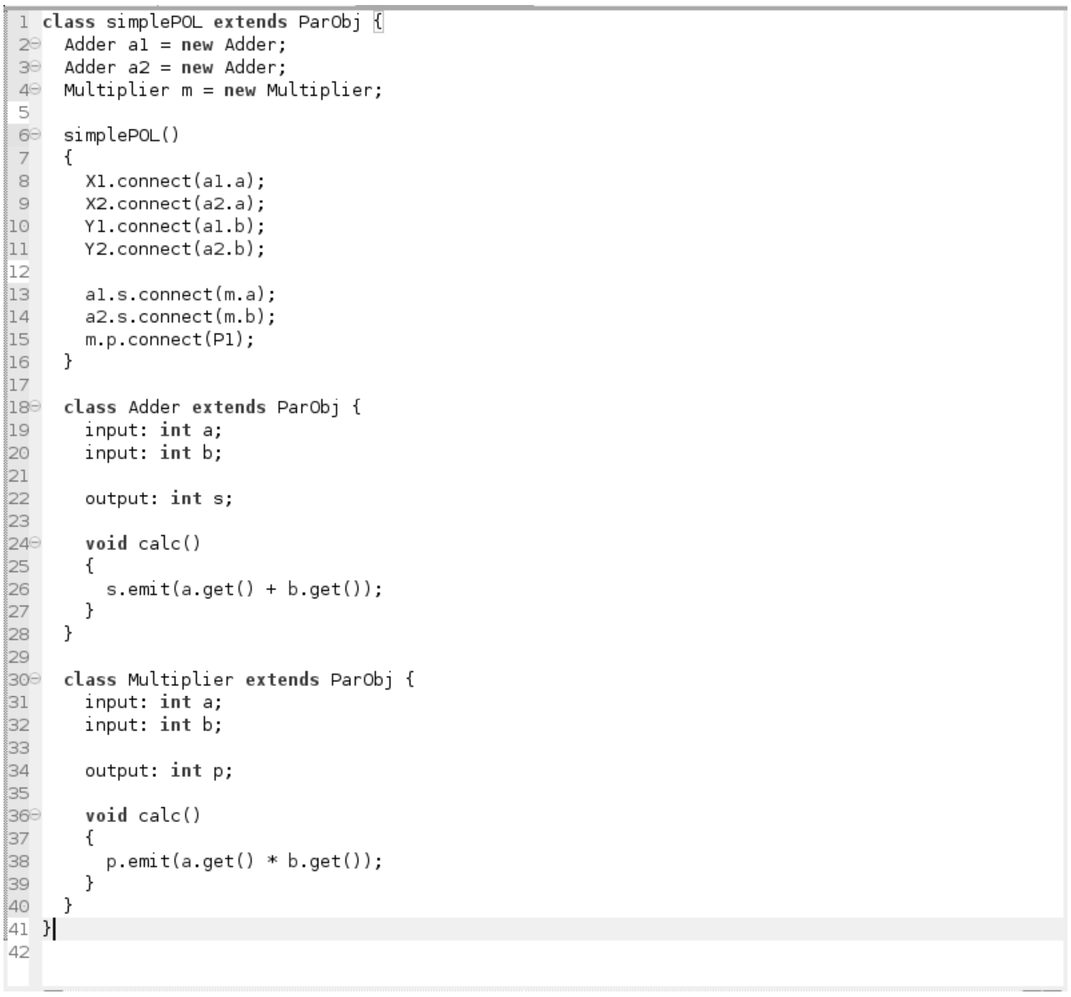


Figure 2.7: In this simple data flow two adder are processing data for a multiplier. According to this example, the input, the output, the connection concept and the *calc()*-method of POL are explained.

The application consists of two classes, an adder and a multiplier. Both classes are translated into individual hardware objects. For the displayed data flow two adder and one multiplier are instantiated and connected. The functionality, adding and multiplying, is realised in the *calc()*-method of both *ParObj*-subclasses, the corresponding POL code is displayed in picture 2.8.

Both classes, the adder and the multiplier, have two inputs and one output and a similar process flow. The data, available at the inputs, is read and after an elementary operation



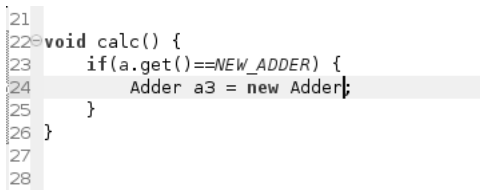
```
class simplePOL extends ParObj {
  Adder a1 = new Adder;
  Adder a2 = new Adder;
  Multiplier m = new Multiplier;

  simplePOL()
  {
    X1.connect(a1.a);
    X2.connect(a2.a);
    Y1.connect(a1.b);
    Y2.connect(a2.b);

    a1.s.connect(m.a);
    a2.s.connect(m.b);
    m.p.connect(P1);
  }

  class Adder extends ParObj {
    input: int a;
    input: int b;

    output: int s;

    void calc()
    {
      s.emit(a.get() + b.get());
    }
  }

  class Multiplier extends ParObj {
    input: int a;
    input: int b;

    output: int p;

    void calc()
    {
      p.emit(a.get() * b.get());
    }
  }
}
```

Figure 2.8: The POL code realisation of a simple data flow, consisting of two adders and one multiplier.

passed to the output. The addition and the multiplication are performed in the *calc()*-method of the classes, which always contains the functionality of the hardware object. The execution of the *calc()*-method is permanently, due to the thread concept of POL.

The encapsulating top class *simplePOL* provides the connections of the subclasses. The constructor is used to connect the input of the multiplier and the outputs of the adder, that will generate the data flow displayed in picture 2.7. In this example, the dynamic instantiation of hardware objects is not used. The adder and the multiplexer are directly instantiated in the declaration section. Nevertheless it is possible to instantiate the classes within the constructor or the *calc()*-method.

For the instantiation of the hardware objects in POL, it is not recommended, that every possible instance is present in the static hardware design, including every possible use case. POL is using the dynamic partial reconfiguration technology to instantiate any required hardware object. The next code fragment in picture 2.9 shows such a dynamic instantiation of an adder during runtime, inside of the *calc()*-method.



```
void calc() {
    if(a.get()==NEW_ADDER) {
        Adder a3 = new Adder;
    }
}
```

Figure 2.9: It is possible to instantiate a POL class, which inherits from the *ParObj*-class, inside the *calc()*-method.

The instantiations of the parallel objects can be performed in the declaration part of a POL class, this would be a static instantiation, most common for hardware, it can be performed in the constructor of a *ParObj*-class, being also a static instantiation, as the compiler can know the amount of generated hardware objects, and it can be performed inside of the *calc()*-method, and that is the required dynamic instantiation. Using the functionality of dynamic instantiation of hardware objects, the compiler can not estimate the amount of objects, needed for the application. Therefore POL includes the functionality of DPR in its framework, and can provide the same flexibility of software in the generated hardware.

The functionality described in the *calc()*-method is always processed sequentially, but the individual objects (*ParObj*) are all processed in parallel. As mentioned above, the common approaches for a software-to-hardware compiler are working with a sequential program and try to find parallelism in the data flow. For the content of the *calc()*-method these projects can be used to generate most effective hardware from the JAVA program code. It is not the attention of POL to solve the problem of software-to-hardware translating at this low design level, but on the system level. Therefore the object-oriented programming concept is used in POL.

POL:
The **Parallel Object Language** is a new programming paradigm, and offers a developer, to use object-oriented programming concepts in a hardware design. Especially the dynamic instantiation of hardware objects during runtime is possible. The needed reconfiguration techniques are generated by the POL compiler, usind DPR.

As mentioned above the programming language POL not only provides a software-to-hardware compiler, but also includes the hardware framework for the execution of the application. To provide the dynamic instantiation of hardware objects, POL needs a scheduler included into the hardware framework. From the point of view of the application, the generated instance are working in parallel, but for POL and the hardware design only the instances that are currently needed to process data, are configured to the FPGA and therefore working. If the hardware environment only supports a single hardware object to ran at the same time, the application will behave similar to any software project, as the scheduling of the hardware objects is equal to the thread concept. But, if more then one hardware object can be executed at the same time, the POL application can improve the performance of any software project, as it provides a full parallel environment, against any other solution, including multi-core processors.

The new hardware programming paradigm POL is developed along with the communication matrix developed in this work. POL is constructed to describe a given application on the system design level and to generate hardware components (that's mainly VHDL files) which would work within the communication matrix, as the supporting hardware framework. On the other hand the communication matrix is developed to support any component generated with POL.

3 State of the Art

Although the POL project is about the development of a software-to-hardware compiler, the main focus of this work is the hardware framework, as an important subcomponent, supporting POL. Therefore this work is focused on the communication structure and especially on the reconfiguration techniques. Also the project descriptions in this chapter refer mainly to the communication structure and the realisation of the hardware reconfiguration, provided by Xilinx FPGAs. Nevertheless some additional approaches to realise a software-to-hardware compiler, one concerning the communication structure (network-on-chip) and one concerning the programming environment (JHDL), are mentioned.

Most of the projects, which are also using hardware and especially the reconfiguration ability, are using the FPGAs in a different way, than the software-to-hardware compiler projects. These hardware designs are mainly used as co-processors, to improve the performance of software based applications or for similar purposes. With POL it is possible to develop such applications, but POL intents to approach the problem on a higher level, as it is done in software. Therefore, this chapter is focused on more general approaches.

Up to now, a DPR design is developed individual for every application. The developer needs a complete knowledge about how the FPGA is reconfigured. The development time to enable an application for reconfiguration technologies is very long and a reuse of a system for other applications is not possible. Therefore it is recommended that the reconfiguration techniques are encapsulated and hidden from the developer. Every software-to-hardware compile will have to provide a framework, which can exactly do this encapsulating. This framework must provide also a special communication structure to serve the streaming character of any hardware application.

In the next two sections the preceding work of former diploma students is shortly described, because this thesis is fundamentally based on their work. Also another attempt for a communication structure in a dynamic environment, using the DPR technology, the network-on-chip concept, which is based on a bus, is described in this chapter. As a complete attempt on a software-to-hardware compiler, the JHDL project is shortly described, which also uses a high level programming language (JAVA) to develop hardware, but without the dynamic aspects of the approach described in this thesis. The last section is about a project, that tries to construct flexible reconfigurable areas using a bus structure, which is called ReCoBus. This bus structure is an alternative for the task areas of the communication matrix, which are generated with PlanAhead and have a rather static structure and other drawbacks, that will be described later, e.g. in section 5.3.

3.1 Fast DPR and Inter-Task-Communication

The starting point of the development of POL and the communication matrix is the hardware scheduling framework developed in [Abe05]. This good working framework enables the scheduling of hardware components. The hardware design is divided into a static part and a dynamic part. The application is also divided into different parts, which are alternately executed in the dynamic part of the system. Therefore the application can use more resources, than available in the dynamic area of the system. The scheduling of the application parts, which are called tasks, is done with the DPR functionality of the Xilinx FPGAs.

In the diploma thesis [Abe05] a task-manager was developed. The task-manager enables a task-to-scheduler communication, inter-task communication, the storage of the task context and a faster reconfiguration of the hardware tasks. The inter-task communication is realised with a system-to-task communication. Because of this standardised communication link between the different tasks, which is fixed, the developer can develop new tasks very easily.

With the framework developed in this diploma thesis, it is possible to store the calculated data and the inner state of the hardware tasks. This functionality is the first step for an environment, that can support the execution of hardware objects developed with DPR. Additionally the reconfiguration speed was improved, to reduce the dead time, of the reconfigurable design parts, for the calculation. Although this is only done for the Virtex II FPGA, it cannot take too much time to adept this feature to the Virtex-4 FPGAs currently used for POL.

In picture 3.1 the main structure of the framework is presented. The content memory is realised with external memory and divided into a global memory part and a local memory part. The communication data is exchanged, using the global memory part and the inner state of the tasks are stored to separated local memory parts. It is directly apparent, that the communication is serial. This is not fatal for this framework, as it only provides one task area and therefore only one task area is working at the same time.

Nevertheless this framework provides the ability of hardware scheduling and dynamic partial reconfiguration the basic approach has some drawbacks. It is not possible to generate any new task during the execution of the application, because the partial bitfile is generated by a read back of the static system including the new task. Therefore, this framework does not provide an easy to handle update mechanism for the application, although an existing task can be instantiated dynamically. The second problem is the serial communication structure for the exchanged data, and the storage process of the inner task state. Both data types are stored in the external memory and are exchanged by a bus. This will work as long as the environment provides only one reconfigurable area, to execute the hardware tasks. If the environment provides more than one reconfigurable

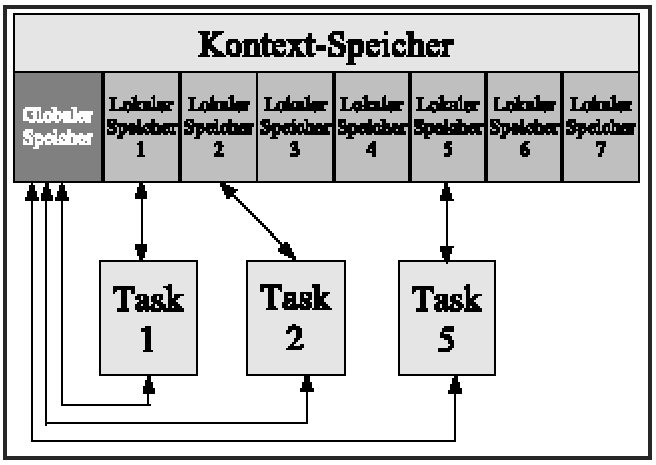


Figure 3.1: The whole communication and the inner state of the tasks is stored in the context memory. The communication is performed with the global memory section, and the task state is stored into the local memory sections. [Abe05]

area, the communication via the data bus must be serialised. In this case the task may be able to work in parallel, but the communication will be serial.

The main basics of this framework are used for the current work, but the new framework is specified to use an explicit parallel communication structure. This will provide more flexibility for the application, the environment and the communication. The communication is explicitly parallel, and will have the same parallelism, whether the environment has one, two, or even more reconfigurable areas. The data connections are not bound to any bus system, but point-to-point, and also do not depend on the access times of external memory. And even the content data, the task state, can be directly exchanged between the ICAP buffer and the external memory via DMA.

3.2 Implementation of JAVA-Threads in Reconfigurable Hardware

A further enhancement of the hardware environment for the scheduling of hardware tasks, developed in [Abe05], is done in [End08]. The diploma thesis introduces a framework for the high-level programming language JAVA, which extends the tread concept, to allow the development of hardware/software tasks, that are executable in hardware and also in software. The task are capable to communicate with each other, whether they are executed in hardware, in software and even in both execution types. It is also possible to change the execution type during runtime. The tasks include the same functionality, as the JAVA threads. Therefore it is very easy to convert any application build up with treads into equivalent tasks, which can be executed in hardware and software.

The communication of the tasks with each other is realised, like in [Abe05], with shared memory. This shared memory is realised in the periphery of the FPGA, in the SDRAM, included on the XUP development board. The work was developed on the XUP[1] development board with a XC2VP30 FPGA. This was done, because the internal memory capacity of the FPGA was to small, for the different hardware tasks.

As mentioned above, it is possible to execute the task in hardware and also in software, and both execution types are explicitly equivalent. In picture 3.2 the data flow of both execution types is displayed. In the hardware execution, the task is configured into the task slot on the FPGA. The task can directly access the DDR ram of the FPGA's periphery, using the OPB (on-chip peripheral bus), where the shared memory is located. This is displayed in part *b)* of the picture. For the software realisation, the task is executed in the JAVA environment of an additional PC. The PC is connected to the processor of the FPGA (PPC) with the serial connection port, and the PPC is also connected to the DDR ram with the OPB. This is necessary to provide the DDR ram access to the PC. The connection between the PC and the PPC of the FPGA is displayed green, and the connection between the PPC and the peripheral DDR ram is displayed in red, both in part *a)* of the picture 3.2.

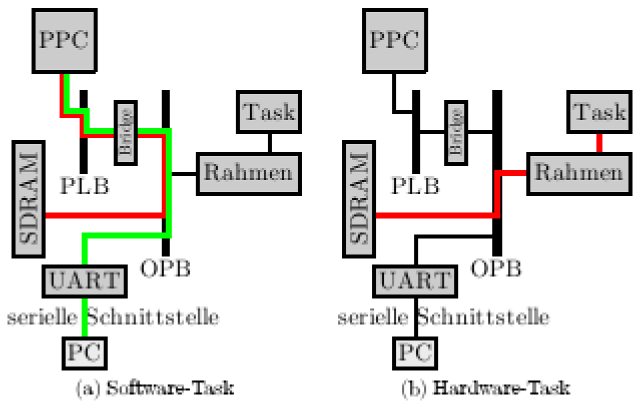


Figure 3.2: The tasks can be executed in software on the PC, or in hardware on the FPGA. In both cases the shared memory is located in the DDR ram of the FPGA board. The communication is allwas performed with the OPB. [End08]

This approach includes the additional functionality of choosing the execution type of the task, therefore it uses the thread concept. But it does not solve any problem concerning the serial data flow and the disadvantage of the shared memory, which is used to establish the communication between the application parts.

[1]Xilinx University Programme

3.3 Network-on-Chip

As mentioned in the two sections above, one main problem of a reconfigurable architecture is the communication. The work of [Abe05] and [End08] suffers from the usage of a bus system. This is not only a problem of hardware designs focussing on hardware scheduling, but for the hardware development in general. Another approach for the general communication structure inside of hardware designs and FPGAs is the network-on-chip (NoC) concept.

In general system-on-chip (SoCs) designs, the components are mainly connected via a bus like system. The idea is, not to create individual components for every application, but to design re-usable components. Therefore, these components also have to use a global and general interconnect concept, e.g. busses. But it has already been suggested, to replace the global bus like interconnect with a on-chip network. This approach is called network-on-chip.

In [Rad06] the general role of network designs, used as an on-chip interconnect concept and the usability of a realisation of an on-chip communication structure based on NoC, is described. This paper is an attempt to generalise the approaches for an on-chip network solution of the components interconnect. It tries to figure out in what kind of application NoC can be useful and what challenges have to be solved.

A more concrete realisation of the NoC concept is given in [Thi06], and it is also about dynamical reconfiguration. The NoC concept for a SoCs design can definitely be very dynamically, concerning the number, size and location of the components, especially if the DPR techniques of Xilinx FPGAs are used.

In this paper such a NoC, using dynamic routing tables, is developed, which can be partially reconfigured, without interrupting the communication. In picture 3.3 the modification of the SoCs and the NoC is shown.

In part *a)* the components are exchanged. After that, the routing of the messages is updated and the new switch is inserted (*b)*). The last part of the picture (*c)*) shows the reconfigured system.

In this approach two reconfigurations are necessary, which will take more time. The most important problem, concerning our work, is a common network problem. The order of the communication messages can in general not be preserved, if the switches can change their connections during the communication. Another problem is, that such a reconfigurable system can not use the DPR tool flow preferred in this work. Too much bitfiles have to be generated.

The last section, which is about the NoC concept, deals with the resource consumption and the performance of this concept. A network-on-chip communication structure requires more resources, than the today's standard bus architectures (e.g. OPB and PLB).

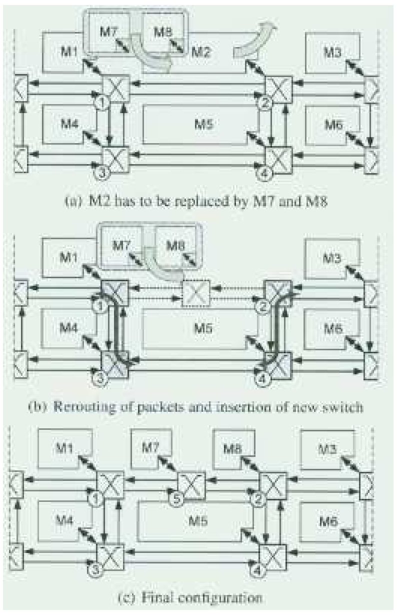

M7
M8
M1
M2
M3
M4
M5
M6
1
2
3
4
5
(a) M2 has to be replaced by M7 and M8
(b) Rerouting of packets and insertion of new switch
(c) Final configuration

Figure 3.3: Dynamic reconfiguration of the system and the NoC. [Thi06]

In [Gra08] the performance of real applications in different NoC communication structures are evaluated. The main conclusion was, that the configuration of the NoC structure directly determines the additional resource consumption and the performance of the application. Such a dependency between the application and the supporting framework is not recommended in this work.

3.4 The JHDL Framework

JHDL is also an approach to describe hardware designs with a general-purpose programming language, as the name *Just Another Hardware Description Language* suggests. In the case of JHDL, this language is also JAVA, but where as POL is a high-level hardware description language, JHDL is a low-level approach. POL focuses on the description at the system level, and JHDL describes the hardware at the register-transfer level.

With JHDL it is possible to describe the hardware circuit structures, as wires, counters, registers, and platform-independent APIs, of the hardware design with JAVA classes. The general approach is done to a software simulation and the actual hardware generation. For the partial reconfiguration, a special PRSocket is used. The different JHDL generated logic can be configured into this special sockets, as long as they provided the specified interface, see picture 3.4. For more information about JHDL see [Pet98] or [JHD].

Although JHDL also wants to describe the run-time scheduling of reconfigurable modules, it does not solve the main problem of DPR. The main problem of DPR is not the

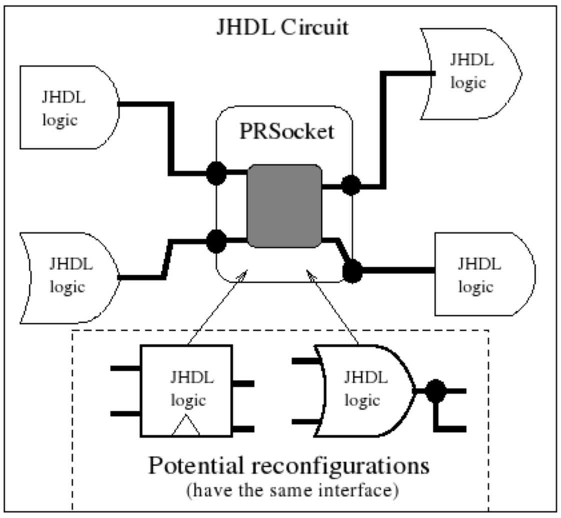


Figure 3.4: Description of Partial Reconfiguration in JHDL. Every JHDL logic, that fulfils the interface can be placed into the PR Socket. [Pet98]

description of the reconfigurable modules themselves, but to provide a communication structure, that enables the modules to communicate with each other. This communication structure must be able to handle the communication, although the modules are scheduled all the time. Such a framework, needed for the description of a DPR capable hardware system is not implemented for JHDL. It may be possible to realise the hardware framework presented in this work with JHDL, but JHDL itself does not provide any structures, that allows the instantiation of hardware modules at run-time.

The main problem of JHDL is, that it should provide an alternative tool flow, contrary to the Xilinx tools, for the generation of hardware designs. But especially the floor planning, as it is provided by PlanAhead and ISE, is not implemented in the tool flow of JHDL. Although it may be possible to describe a DPR using design with JHDL, it can never be synthesised.

3.5 The ReCoBus

Most of the hardware designs, that currently use the DPR technology, are using an island-style solution for the reconfigurable areas of the FPGA. The FPGA is divided into a static part and a dynamic part. The dynamic part is dedicated for the reconfigurable part of the design, and also divided into different areas. In every of these different dynamic areas only one module can be configured at the same time.

The ReCoBus technology tries to use another approach. It will allow different modules to use the same reconfigurable area. As the ReCoBus architecture is currently developed only for Virtex-2pro devices, it provides a slot-style solution for the reconfigurable areas. For the Virtex-2pro FPGA the configuration frames are only divided into columns, which means that there are no rows that can be addressed separately. In general even a grid-style solution is possible. For the Virtex-4 devices, the configuration frames are also divided into rows. This enables a more precise separation of the reconfigurable areas.

The ReCoBus architecture and the corresponding tool, the ReCoBus-Builder, are introduced in [Dir08] and provides a bus-based communication structure, that also allows additional point-to-point connections. The modules can be placed freely inside of the reconfigurable area. The general architecture is displayed in picture 3.5, for more information about the technology, see [ReC]. Part *a)* shows the common realisation of a multiplexer based bus. The architecture for a partially reconfigurable system must assure, that every signal is placed at the same position for every slot of the reconfigurable area, as shown in part *b)*. This will allow, to move the partial module across the reconfigurable area. For the general routing it is necessary that the bus specific routing of every slot is completely identical.

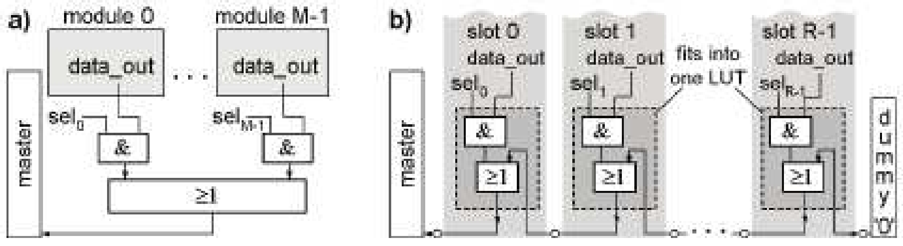


Figure 3.5: Communication structure for a read data line of a partially reconfiguration system. [ReC]

The communication structure provided by the ReCoBus approach is able to implement many bus protocols, see [Dir08]. But in correlation with the system level approach, made in this work, this is not completely the desired functionality. The ReCoBus architecture allows to instantiate more than one module into every reconfigurable area, but this will again serialise the data flow. Nevertheless, the architecture can be used for the island-style solution of a DPR system, as introduced in this work.

The DPR system using the ReCoBus technology can have one or more reconfigurable areas, and will be able to use each of them for different reconfigurable modules. The main advantage is the low level communication structure, which the ReCoBus-builder provides for the system. The reconfigurable modules communicate with a fixed bus, which is dedicated for the reconfigurable area. That means, every reconfigurable area will have its own bus structure. It is also possible to have a direct point-to-point connection with other parts of the FPGA, like I/O pins.

Although the communication inside of a reconfigurable area is serial, this approach is a good alternative to the PlanAhead design flow, if only one module is placed into every reconfigurable area. The ReCoBus approach enables an exchange of tasks over different reconfigurable areas, without re-synthesising of the module for every dynamic area, which must be done for PlanAhead.

Using the ReCoBus-builder, instead of PlanAhead, may be an improvement for the communication matrix. As the basic bus structure of the ReCoBus is located on a lower level than the communication matrix, it can be used as the reconfiguration technique provided by the communication matrix. Especially, if only one module is configured into every reconfigurable area, no serialisation of the bus structure will occur. The ReCoBus approach can offer a more flexible environment for the communication matrix, than PlanAhead, as it provides more than one interface between the reconfigurable area and the system.

4 Specification Analysis

The actual communication matrix, as the result of this work, was not present from the beginning, as described in chapter 5. Before the development of the communication matrix can be started the basic requirement and the restrictions of the communication matrix have to be specified, these are documented in the first two sections. The main part of this chapter is used to specify and evaluate the different approaches to realise the functionality, which the framework must provide for the communication. In chapter 5 the concrete specification of the communication matrix, regarding the analysis made in this chapter, is described.

In the end the communication matrix is meant to serve any application developed with POL. But the development went from one requirement and constraint to the other, generating new principles each time. Additionally, this in principle incremental process, contains many loops as the programming language POL is developed at the same time. So the described principles and requirement concerning the communication matrix were changing during the development process. Nevertheless, this section does not correspond to the chronological development order of the communication matrix.

4.1 Requirements for the Communication Matrix

In chapter 1 and 3 the basic ideas for the requirements of the communication matrix are suggested. This section will describe the requirements in detail.

Requirements:	**Description:**
Maximal Parallelism	Due to the slower clock frequency of FPGAs, in comparison to CPUs, the application parts have to run completely in parallel.
Minimal Memory Consumption	The less memory for the tasks are needed the more bandwidth and reconfiguration time can be realised.
Minimal Slice/Resource Consumption	The less resources the communication matrix will need, the more resources are left for the application.
Preserve of the Data Order	The data order must be preserved against the scheduling of the application parts.

Table 4.1: The communication matrix must fulfil the here listed requirements.

An application developed in software will use a CPU with a clock frequency in the range of GHz, but the application parts will only run quasi-parallel. The same application developed for a FPGA based system has to use a clock frequency of 100 MHz. To achieve a better performance the application parts must run really in parallel. This requirement of maximal parallelism is important for the number of task areas and the scheduling algorithm. If the number of task areas is less than the number of tasks, a scheduler is required. The scheduling of tasks during runtime will reduce the parallelism of the application parts execution.

Another problem of the design is the data storage capacity needed during the scheduling. The memory resources of the chip are limited and the amount of data relates to the sample frequency, the data width and the number of tasks of the application. The communication structure of the communication matrix should obey the requirement of minimal memory consumption to increase the capability of the application.

The same problem occurs for the general resource consumption of the application. To realise more and bigger task areas on the chip, the communication matrix itself should use as few hardware resources as possible.

The developer using POL and the communication matrix does not see the constraints in the scheduling and the communication matrix layer. Therefore the communication matrix must preserve the data order towards the scheduling. This is the most important requirement for the communication matrix. Any other requirement affects only the performance of the communication matrix or rather the application running within the communication matrix. Otherwise, the assurance of the data order is directly required for the correct functionality of the communication matrix.

4.2 Principles of the Communication Matrix

Apart from the requirements for the communication matrix, also the principles, which the communication matrix will fulfil, have to be described, before the specification of the communication matrix is described. The principles are divided into two different groups. The first one describes the general development direction and functionality of the communication matrix, where as the second group is aimed to motivate the criteria, which evaluates the design and the decisions made during the development process.

Principle:	Description:
Utilisation of DPR	The dynamic implicated in the high level description POL, requires DPR technologies.
Utilisation of Object Orientation	A high level language always uses object oriented concepts.

Table 4.2: The communication matrix must be conform to the listed requirement.

The communication matrix is developed for the high level programming language POL, and POL is dedicated to describe hardware. The parallelism of the application is explicitly described using object-orientated programming, and in this case the programming language JAVA is used. This recommends the communication matrix also to realise object orientated aspects. Additionally the dynamic instantiation of objects, included in JAVA will be used. So the application will need a corresponding dynamic mechanism for the instantiation of tasks as application parts. As the available hardware resources are fixed and limited, the communication matrix will use DPR technologies to realise the POL functionality for the application.

Similar to the requirement of maximal parallelism of the application part, the communication matrix must be parallel itself. This means, that not only the application parts have to run in parallel, but also the communication or the data message exchange have to be parallel, too.

These parallel running application parts, the tasks, have to be suspend able. If there are more tasks than task areas, the scheduler needs to exchange the tasks. To do this without any data loss, the tasks must be brought into a precisely defined state for reconfiguration. This is described by the suspend ability of the task. If there is more than one task area, which is recommended for a good performance, the resources provided in the areas should not differ. This is the only case which enables the tasks to exchange freely. From the communication matrix point of view the only difference between the different tasks is the individual unique ID and the content data. The number of tasks, respectively object instances, is not given in the compile time. Therefore the amount of tasks must be dynamic during the runtime. This is considered with the scalability of the communication structure.

The last and most important point is the serialise ability of the application data flow, predefined not only, but mainly by the communication matrix. The functionality and a predictable behaviour of the application depend on the preservation of the data flow. The validity of the data exchanged within the application is constituted by the assumption, that every data flow is equivalent to even one serial data flow.

Now that all the considered principles and requirements of the communication matrix are described, the next step is the evaluation of the current, final communication matrix itself. This will give a more detailed overview, why the communication matrix is presented in the actual shape.

4.3 Evaluation of the Communication Matrix

As mentioned above, the main idea during the development of the communication structure for the POL application, especially including dynamic partial reconfiguration, was

Criteria:	Description:
Parallelism	It is not only necessary to let the tasks work in parallel but also to communicate in parallel
Suspendability	During scheduling the serial data stream is interrupted somehow. This should not be recognised by the application and the user.
Flexibility	The playcement of a task should not correspond to the functionality of the task or a given application.
Scalability	To instantiate a class during runtime the quantity of instances must be dynamic.
Serialisability	There must be even one data flow provided by the communication matrix, which is equivalent to any serial data flow.

Table 4.3: The communication matrix is basically evaluated against these criteria.

the static attribute of this structure. Former attempts always uses a dynamic bus structure or shared memory to perform inter task communication (see chapter 3). This will always lead to a sequential data flow, which will affect the performance during any reconfiguration process.

These resulting performance constraints and any other resulting constraint should not concern the communication structure developed in this work. Regarding the intention of this work an explicitly parallel communication structure is recommended.

The main approach of this work is to use a communication structure, which is static concerning the parallelism, but highly dynamic for the connections between the application parts. The design provides several task areas, to serve the application parts. These task areas must be connected among each other. This is done with a static communication structure responsible for the dynamic establishment of communication connection between the tasks located in the task areas. The communication links between the task areas will change during runtime due to the scheduling. The allocation of free task areas for the tasks, to be instantiated, is not determined. The communication structure must react to the decisions made by the scheduler.

This is described in picture 4.1. Tasks, which have established a communication link, can be relocated in the framework by the scheduler. In such a case, the communication structure must update the communication link between the tasks. This will allow the tasks to communicate independently from their relative position on the chip.

In a bus like environment the reconnecting of the task's virtual communication links is done with the real address space of the application. But in a real parallel environment, the connections of the tasks are real and the address space is virtual. The physical realisation of a communication link must change.

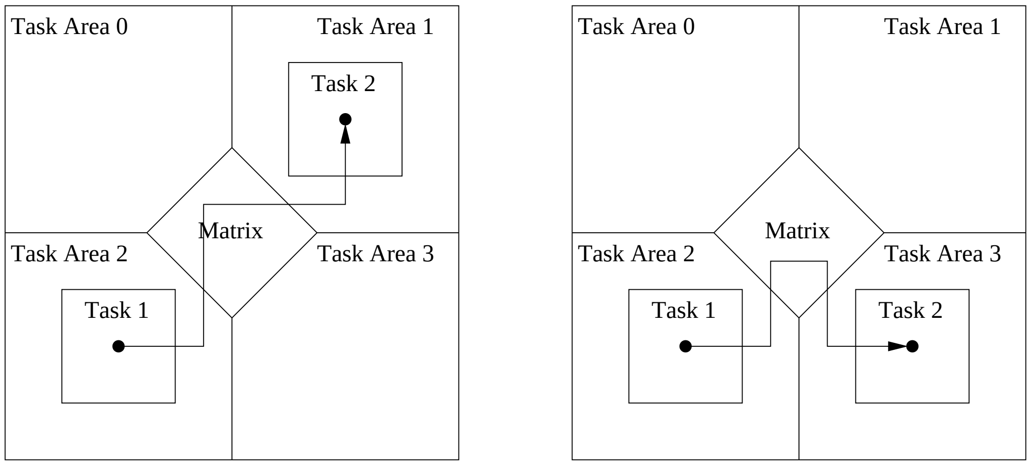


Figure 4.1: The Communication Matrix connects the task areas with a structure independent of reconfiguration. That allows the tasks to communicate independently from the position on the chip and the state of third task areas. This is provided by the static character of the matrix.

Nevertheless, this is still not how the communication matrix is working. As already mentioned, one main aspect of the communication matrix is the dynamic instantiation of hardware objects during runtime. How dynamic instantiation is handled in software and what it means for a hardware design is considered in the next subsection. The communication matrix will also serve the main advantage of hardware, the streaming applications. This is also considered in a subsection. After that the general structure of the communication, the buffering with FIFOs, the updating of the communication links and the scheduling of the tasks are described and evaluated. There are also some very special decisions described, concerning direct task-to-task communication, meta data and content memory and even the re-configurability of the communication structure itself. At the end the general idea of the functionality of the communication structure, realised with the communication matrix, is summarised.

4.3.1 Dynamic Instantiation: *for-each* vs. hardware

The first application to be realised in software using POL was the ancient computer game Pong. The game was previously programmed in hardware, running on a Virtex-2pro XUP board. The main advantage of this hardware realisation is the modularisation of the hardware components. In principle the bars and balls run independently from each other, so they are predestined to be realised as DPR modules. On the other hand, the balls need the position of the bars. Therefore the modules have to be able to exchange data.

The first software POL version of the pong game includes two bars and a variable amount of balls. The balls, the bars and the viewer are realised as Java threads running in parallel.

With such a structure it is very easy to instantiate new objects during runtime. The idea for the hardware development was to describe an application with a dynamic amount of objects which are communicating with each other.

There are several ways to describe this dynamic amount of balls in software, but in hardware every possible ball would have to be represented in the hardware design. This problem should be solved by partial reconfiguration. It was not clear at this early state of work, as to what kind of structures dynamic instantiation would lead to in a hardware design.

In the software POL design, the position of the balls and bars are retrieved with a *for-each* statement. This statement should be translated to a corresponding hardware structure. The first idea was to use the internal block ram in a dual port mode, to realise shared memory within the FPGA. The balls can compare the position of the bars with their own position and send the result to the viewer. The ball and bars in this environment can run independently. The viewer will generate the required functionality to the user, which are visible balls and bars. There is still no concept for an exact dynamical instantiation of modules and how to handle the connection between different modules. Picture 4.2 shows the theoretical realisation of this concept. Here the missing instantiation dynamic becomes very clear.

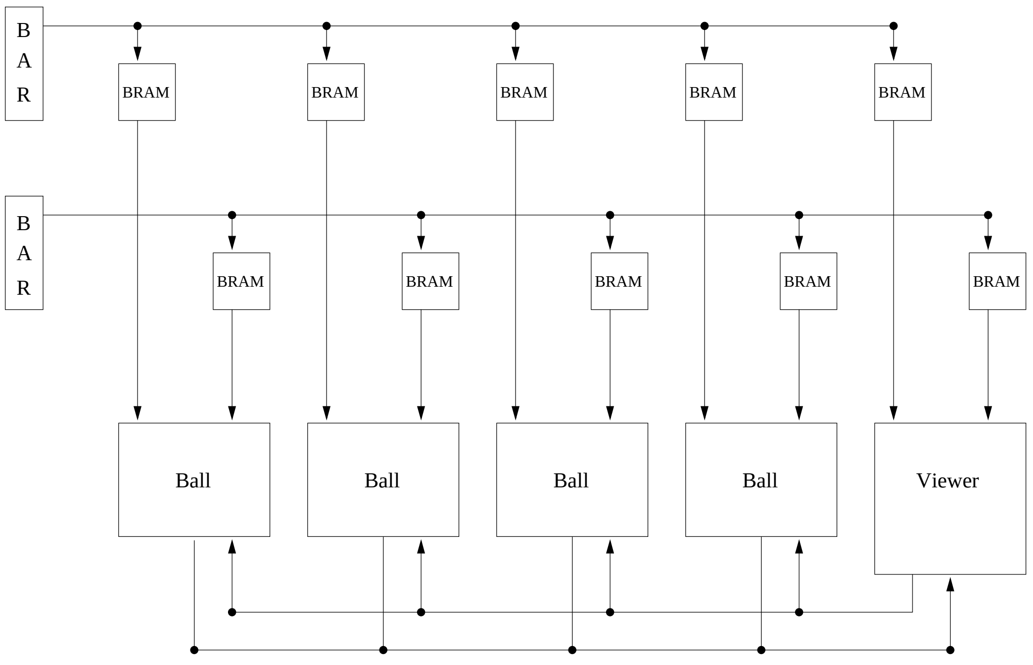


Figure 4.2: The communication relation between the components of the video game Pong (balls and bars) is described with a *for-each* statement in software. This statement must be translated into a dynamical hardware construct.

The computer game Pong is a good application example for dynamic instantiation of hardware objects, if the amount of balls or even bars is not fixed. But the main advantage of hardware is fast data processing and FPGAs are nearly always used in such applications. For this utilisation of hardware, Pong is a very weak example, because the exchanged data are few, slow and not crucial. More suitable examples are streaming applications.

The dynamic instantiation of hardware objects is a new feature of hardware designs, introduced by the new programming paradigm of the hardware description language POL. Although POL can support a broad spectrum of different application types, the main issue of every hardware design is still the streaming aspect and the processing of huge data streams.

4.3.2 Streaming in a Dynamic Environment

Streaming examples are an easy to handle approach to consider the different criteria for the development of a communication structure, which offers an application in the use of dynamic partial reconfiguration. The considered streaming applications should contain different filters to work in parallel on the data stream. As a precise example an audio data stream is used, but for further progress an VGA video stream is designated.

The audio application should consist of a static input and output framework. The data is shifted from the input to the output, while the different audio filters are working on the data. As the data stream is sequential, this is regarded by the application and the various filters can work successively on the data. The filters are placed in DPR areas of the chip and can be exchanged. This will allow the application to use generally more filter (but not simultaneously) than logic resources that are available for the filter. The number of filters that can work simultaneously on the data stream is determined by the DPR area count and the reconfiguration time. What effect these aspects will have on the scheduling, is described later.

The work flow of the example application is considered as follows. The static input component writes the data into a dual port block ram on the FPGA and the also static output component reads the data from a second block ram. Between these block rams the application tasks are instantiated. The task areas are linked between the input buffer and the output buffer. Every area including the static input and output areas must be separated by two buffers. From the task areas point of view, every area has a previous input buffer and a following output buffer. The task can read the data from the previous buffer and sends it to the following buffer. During the configuration time the input buffer slowly fills up and the output buffer becomes empty. If the buffer depth is sufficient and the task is processing the data fast enough, the data stream seems to be interrupted for the user. The task has to shift every data from the previous buffer to the following buffer in the time that it is available on the FPGA between two performed configurations.

In principle it is completely sufficient to buffer the data during the reconfiguration time for the streaming examples. This would not be most efficient, but scheduling and buffer constraints are discussed in a following section. A possible realisation of these streaming examples is shown in picture 4.3. The general idea to interconnect the tasks into the general data stream is shown, as well as the concrete realisation using a single task area.

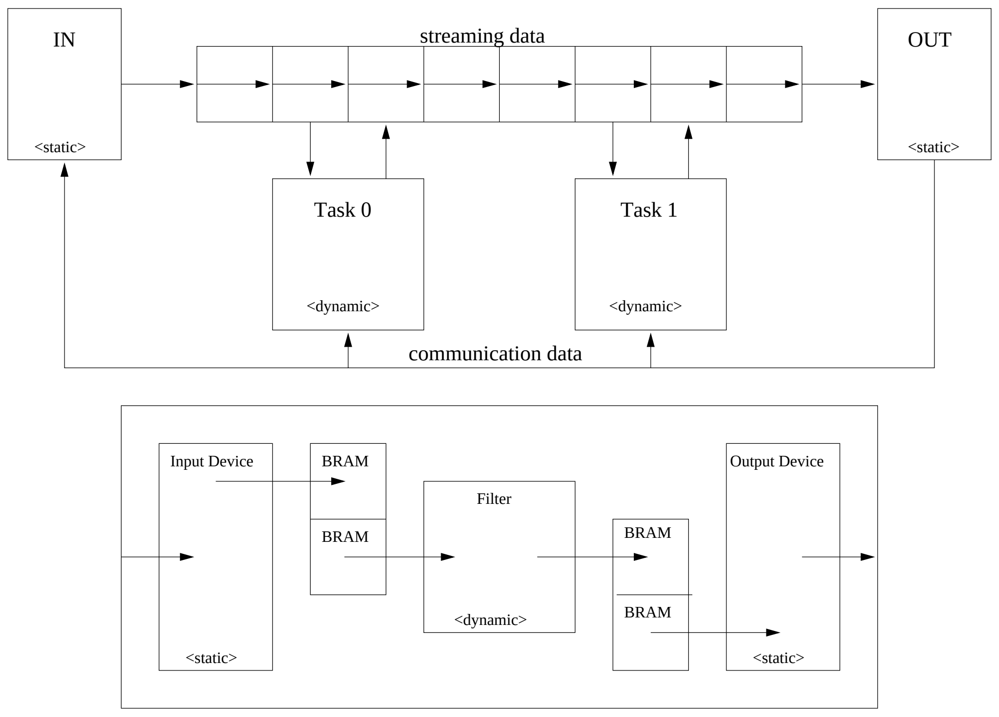


Figure 4.3: A more suitable example than Pong for the use of dynamic partial reconfiguration is a streaming application. A video data stream is orchestrated by different filters. The upper part shows the idea of interconnecting the tasks to the data flow and establishing a separated data controll path. The lower part is a concrete realisation using one task area, the controll data flow is not shown.

Additionally the different tasks, representing the various possible filter, must be able to communicate with each other. A particular filter, for example, can reduce the sound volume in the audio application, and a additional amplifier should compensate for it. This will require additive communication between the tasks.

It turned out that these are two different kinds of communication implemented in such a streaming examples. The individual filter should exchange control data, this is inter-task-communication. They should work on the serial data stream, which they must do independently and in parallel and using pipelining. These two communication types have already been described in chapter 2.

But for POL, the communication matrix should be able to do more than storing the streaming data and exchange the tasks. Considering the use of reconfiguration as realisation of pipelining, it is necessary to buffer the serial data stream and the exchanged control data. One aspect is to assure the data stream to remain uninterrupted, not only for the user, but also for the application, another aspect to be considered is that the order of task execution is only dynamic using the reconfiguration. Even if the streaming example is only one aspect of POL the unrestricted combination of communication links is most important.

It is not the requested role of POL to serve specialised kinds of applications. The main issue of POL is to enable the developer to describe his hardware design encapsulating the functionality of DPR. This includes streaming application and also the dynamic instantiation of hardware objects. The main problem for these application types is for the streaming part a communication structure serving the huge communication traffic and for the dynamic instantiation of hardware objects, to provide the DPR functionality to the application.

4.3.3 The Communication Structure

It is directly apparent, that the common approach for the communication structure is related to networks. The first aspect is, how to connect multiple task areas and how to establish the communication links. A circular shaped communication matrix or a radiating communication matrix have been directly apparent. As the data stream is buffered during reconfiguration, also a circular buffer had been accounted. And last but not least we did not disregard a bus like structure. In this section, this four basic structures are described and evaluated against each other. But for the final communication matrix, functional parts of all these communication structures are used.

The task areas will have a point-to-point connection to the communication matrix. This is recommended for the usage of the reconfiguration buffer. The tasks, on the other hand, can be connected freely. Especially connections to, due to scheduling, currently not available tasks are required. So it is contemplated to use the communication matrix for data buffering and to reconnect the tasks after scheduling.

Centralised Communication Structure

A centralised communication structure is the most simple realisation of the point-to-point connection between the communication matrix and the different task areas. The principle of a centralised communication structure is displayed in figure 4.4. The communication matrix provides the functionality of an arbiter, but from the point of view of a single task area and not of the communication matrix itself or the connections. The communication matrix establishes the connections for a task area. A single task area can have only one connection at certain time, but every task area connected to the communication matrix can have a connection at the same time. This enables the tasks, configured into the task areas, to communicate with each other in parallel and independent from any other communication link, established by the communication matrix.

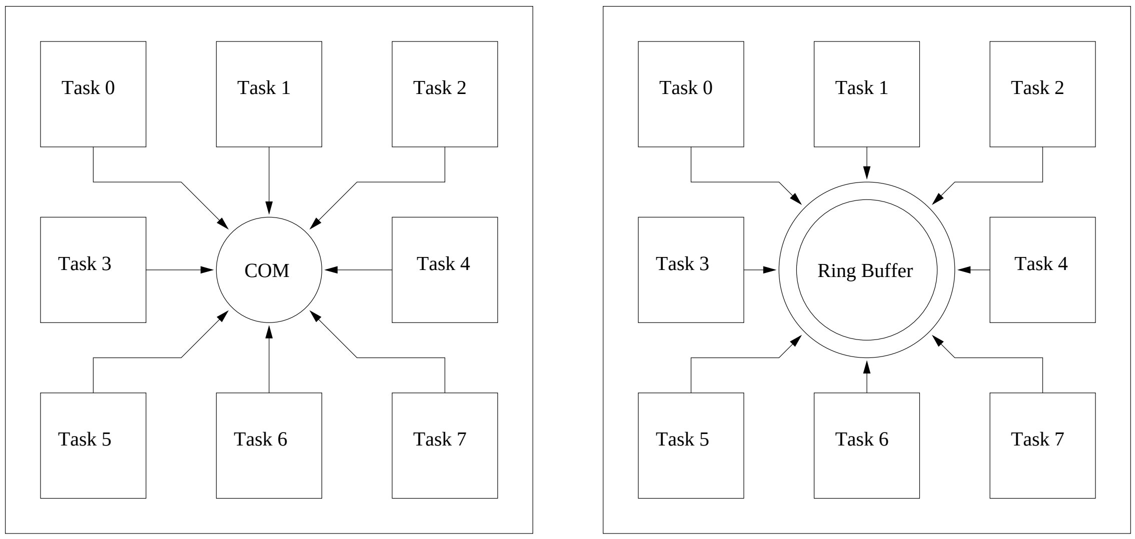


Figure 4.4: A radiating communication structure was one of the first ideas, the communication matrix as the central component for communication connecting the task areas to each other. The inner structure of the communication matrix is similar to the functionality of a ring buffer.

Circular Communication Structure

A circular communication structure will use the direct task area-to-task area connections, see picture 4.5. As the connections between none touching task areas over more than one task area will increase the resource consumption and the communication control complexity above average, a circular connection of the task areas is the best realisation of a communication matrix using direct task area-to-task area connections. Nevertheless an additional task area bypass is needed for reconfiguration times. During the reconfiguration of a task area, the communication interface is also not available. The reconfiguration bypass can assure the communication links between the task areas, even in the dynamic environment of DPR.

Bus Like Communication Structure

A bus like communication structure is easy to realise, as the needed work to develop a communication matrix based on a bus is low. To instantiate a bus on the chip and connect the task areas too the bus with a busmacro based interface, will serialise the communication between the tasks. In figure 4.6 a simple bus-based communication structure is displayed. Only one task area can work on the bus at a given time. It is not possible for

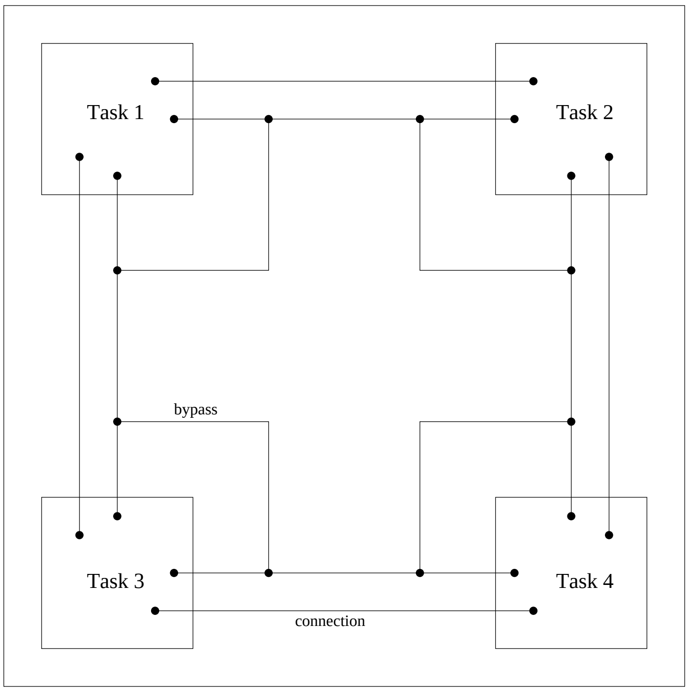


Figure 4.5: As the position of a task is not predictable and a task area can be locked, a circular communication structure would need a configuration bypass.

more than one task to communicate simultaneously on the bus. To improve the parallelism of the communication, the bus can be divided into subcomponents. One approach can be the instantiation of a separate bus for every task used for the application. This can help to improve the parallelism of the communication. Another approach is the separation of application parts. Tasks, which will communicate directly with each other, are placed on the same bus. In this case direct communication also means serialised communication, so the bus like structure do not reduce the parallelism of the communication. But both approaches will reduce the flexibility of the framework. A modification of the communication links during runtime will result in a complete replacement of the application parts.

Validation of the Communication Structure

What kind of communication structure will serve the framework for POL best? Any attempt regarding bus-like structures is rather pointless, because of the immense drawbacks generating a sequential dataflow. In any software based environment the communication performed with a bus is sufficient, as only one thread can be active at the same time. In hardware this is not desired. The ability of processing the data in parallel, is the

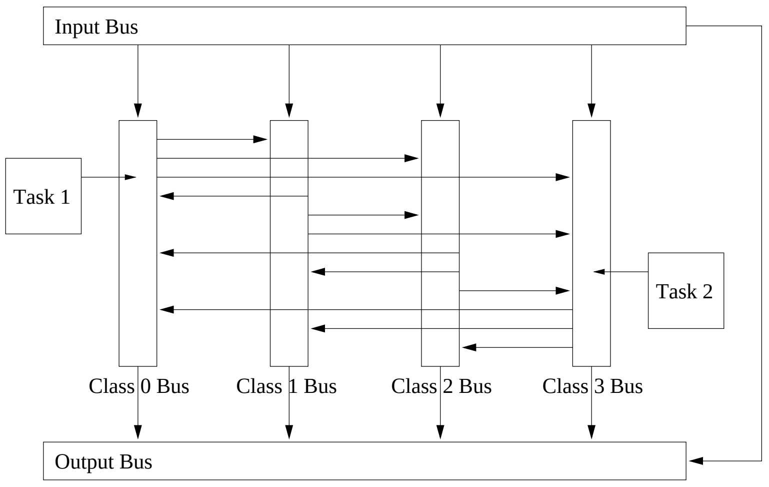


Figure 4.6: A bus like structure will always serialise the communication. Separating the classes can help to avoid this problem.

main advantage and must be used in a POL based system. Therefore a bus like structure is excluded. This is also verified by former attempts to create a DPR framework, see chapter 3.

A circular connection structure is also not the desired result. The reconfiguration bypass generates more problems than it is solving. The scheduling of tasks would be bounded to the streaming example, because only this data flow will get any advantage from such a communication structure. The order of tasks included into the application data flow is not determined. But this is required for a good scheduling decision during runtime. The scheduler will have to analyse the dynamic data flow and try to find the correct instantiation order fitted to the point-to-point communication realised in such a circular communication structure.

The ring buffer structure separates the connections from the task area state using external memory. The configuration bypass is not needed and the scheduler does not need to ponder about the actual data flow. Nevertheless, the scheduler does not have any chance to adept the instantiation of tasks to the amount of data available for this task. Which means, that the performance also depends on the current data flow, which can be dynamic in a POL application.

The only thing left is the radiating communication structure. This is actually the most sufficient basic concept for the development of the communication matrix concerning POL, but only using several task areas. If a huge number of task areas can be used, this structure is the best one. Unfortunately the expected major cases assumes, that the number of task areas is very small, and that the number of tasks is bigger than the number of task areas. In the example application of this work, four tasks and only one task area are used. So it is a good basic design but must be adapted to the required purpose of POL.

Therefore, as mentioned above, none of the here described communication structures are used for the communication matrix in its purest design. The communication matrix is realised as a central object with a point-to-point connection to every task area. But it also has the functionality of a ring buffer, but will allow the scheduler to react on the amount of data available for a given task, rather than reacting due to the data flow. The components of the communication matrix use different addresses to communicate with each other, which is similar to busses. What kind of buffered connection structure is used for the communication matrix is described in one of the next section. But before some more fundamental decisions made are documented, which are made contrary to most of the other common approaches of a DPR framework.

4.3.4 Direct Task-to-Task Communication

It is also possible to allow direct task-to-task communication. The connection can use additional FIFOs or not. The advantage of the additional FIFO is, that the reconfiguration time can be used to fill the FIFO. This will reduce the costs of every reconfiguration. To use the direct task-to-task connection the task has to know who its neighbours are. This can be done by the scheduler or with a direct connection. The scheduler would need to know the exact data flow to place the appropriate tasks in adjacent task areas. This is not possible in a design using dynamic instantiation. An additional connection between task areas to verify the adjoining task will solve this problem. But in this case the scheduler has no influence on the placement concerning the direct task area connection, and it is possible that the additional connection is never used in the application.

The direct task-to-task communication is realised by a direct task area connection avoiding the communication matrix. This can increase the performance of the task connection. Another point to keep in mind, is the fact that with such a direct task area connections the task areas are not equal anymore. The task areas placed at the edge of the reconfigurable area of the FPGA will have less direct connection to other task areas, than task areas placed in the centre of the structure. This will give an additional placement constraint to the scheduler. Tasks with many connections to other different tasks will have to be placed into the inner task areas and tasks with fewer connections will be best placed in the outer regions.

This is exemplary shown in picture 4.7. The middle task areas have two possible direct connections to the upper and lower task area, whereas the border task areas have only one direct connection. In the given example, the direct connection between the task areas is realised with additional FIFOs.

This design including a direct task area connection using additional FIFOs does not solve any problems of the communication matrix and makes the scheduling much worse. The

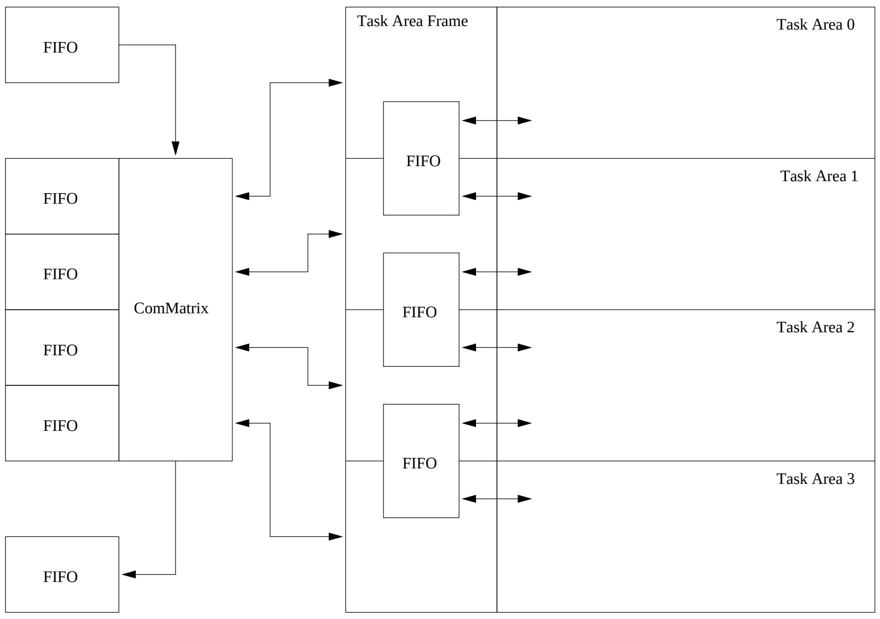


Figure 4.7: A direct task-to-task communication is not desired because the data flow would dominate the scheduling. That would implicate that the scheduler is aware of the future data flow in order to make a suitable resource allocation for the scheduled tasks.

data flow must be present to the scheduler, if the direct connection between task areas is to be used successfully. So it was decided not to use the possibility of a direct task to task communication establishing a direct connection between individual task areas.

4.3.5 The Meta Data and Content Memory of the Hardware Objects

One major problem of the communication matrix are loop backs from the data flow within the application. This can be demonstrated with an echo filter. The filter stores the audio data in a buffer using a BRAM and adds the stored data to the current audio data after a certain time. The BRAM buffer is designed as a FIFO. The depth of the buffer realises the delay of the echo.

The audio data are normally stored, from the tasks point of view, into an internal BRAM. An alternative task structure can use a loop back, to store the audio data external, which means inside of the communication matrix. The data is send via the communication matrix back to the same task. The delay produced with this structure will also depend on the buffer depth. But in this case the depth is not chosen for the application but for the performance of the communication matrix.

This stored data represents meta data of a task. The stored data are only valid for the exactly that filter. This meta data has to be stored elsewhere during reconfiguration. In a design using the loop back, this problem would not occur.

The available memory on the FPGA is quite limited. Using the loop back design, every instance of a task containing meta data will need additional memory. This will limit the number of instances an application can instantiate on the FPGA simultaneously.

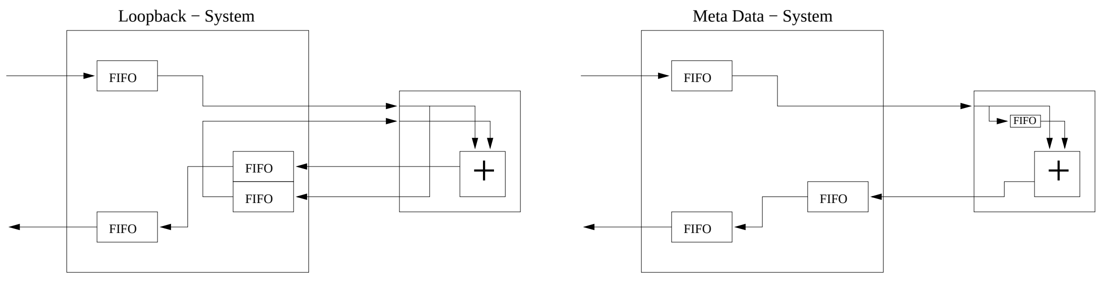


Figure 4.8: A filter realising an echo has to store the audio data in order to add it, after a certain time, to the new audio data. To do this the task has to have an internal buffer. This data will occur as meta data for the reconfiguration process and must be stored additionally. Another approach is to use a loop back.

The read back of the meta data stored in a BRAM is performed by a short reconfiguration. As the name of the reconfiguration suggests, the costs of a short reconfiguration are less than that of a reconfiguration that changes the hardware itself. With an internal content memory, the task is independent from the communication matrix and can manipulate its own data. The loop back will separate the content data from the task thus changing the functionality of the task.

The communication matrix will provide the ability of a full content cover, if the task can realise its state with meta data stored in a BRAM. This will reduce the communication overhead and will save valuable internal memory space needed to cover the reconfiguration time during scheduling. If a task wants to use a loop back, it can do so but has to assure the required functionality by its own.

4.3.6 The Multiplexer-FIFO Concept

As already mentioned, a direct task-to-task communication is not desired because it is not possible to detach the scheduling from the application internal flow of events. The communication would depend on scheduling or vice versa. That's why any communication, the inter task communication and also the serial data stream, should be exchanged via the static communication matrix.

The multiplexer-FIFO concept, used for the communication matrix, is somehow a mixture of the contemplated structures, described above in section 4.3.3. The individual tasks should be connected with multiplexer with each other and the data stream is buffered with FIFOs.

It is possible to keep the serial data stream outside of the primary communication structure. A pure realisation of this concept will not work properly, although the data stream can be coupled very dynamically to the individual tasks, a free scheduling of the resources is not possible. For example, the execution order is defined by the sequence of task areas. To insert a task in front of the application would cause every task area to reconfigured, see picture 4.3.

Nevertheless it is necessary to decouple the serial data stream from the main functionality of the task, in order to get a full working communication matrix. The reconfigurable parts of the FPGA, the task areas are decoupled from the system via FIFOs and the serial data stream is directed to the correct task with a static construct of different multiplexer.

There are many possible approaches for such a multiplexer-FIFO construct, but the crucial point is the resource overhead generate with the communication matrix. The overhead should not negate any advantage of the reconfiguration technologies, regarding the resource consumption of an application. In principle the needed amount of FIFOs, is the problem to be solved. Now we have to consider, whether the FIFOs are related to the task areas, or rather to the tasks themselves. Both approaches have some advantages, but in the end, the FIFOs of the communication matrix are related to the tasks.

A task area related FIFO concept means, that a task stays configured in a task area, as long as data is present in the related FIFO buffer of the task area. The buffer is bound to the task, as long as the buffer contains related data. Although the buffer belongs physically to the task area, it is allocated by the task, concerning the content data of the buffer itself.

To consider the optimal FIFO count, different fictional application only represented by black box tasks are invented. These tasks have a given data flow and should be scheduled on a varying count of task areas. The question was, how many task areas and FIFOs are needed for the application.

One thing, which is not accounted at this point is the flexibility and dynamic of the data flow. The estimated FIFO count depends directly on the suggested application. This is working as long as the data flow do not include any branching. If the data is separated into different paths and recombines later, this FIFO structure does not work. Additional FIFOs for every path are needed. But using dynamic instantiation of hardware objects for any POL application, neither the data flow, nor the branches are known at the compile time of the system. So, POL do not exactly know how many FIFOs are recommended, and how to connect them. This is not the main problem of the FIFO concept and is neglected in the following considerations.

Another interesting assumption is the minimal amount of task areas and buffer needed for an application with a known data path. The number of task areas is not so crucial as the number of buffer of each task area. If the number of buffer per task area in respect to the amount of data paths is sufficient the number of task areas is not important for the matrix to work. Only if the buffer number falls below a critical value additional buffer for every data path are needed. Only, the scheduling is still not expecting a dynamical data path, which will occur for any application using dynamic instantiation of hardware objects.

In picture 4.9 a fictional application is displayed in part *a)*. The application is scheduled on one task area in part *b)* and is scheduled on two task areas in part *c)*. The application needs a minimal amount of three FIFOs, as it has three branches in the data flow. In this case, it is not important, whether the FIFOs are assigned to the task area or the tasks.

To use only one task area for the application, causes the environment to provide an additional buffer for every branch in the data path. This is shown in part *b)* of picture 4.9. The tasks five and seven realises such a branching data path. These tasks generates the data for the additional buffer. If the application is scheduled on two task areas, as shown in part *c)*, both branches can be separated by different task areas. The data in the buffer assigned to the task areas can now be dedicated for a task not present on the right task area. Which means, that although the buffer are assigned to a concrete task area, the destination task of the data can be scheduled in any task area. Another point is that the buffer overhead will increase with the count of the task areas. In part *b)* only three buffer are needed, where as part *c)* needs four buffer, letting one buffer unused.

Till now the explicit connection between the task areas and the FIFO buffer was not accounted. The most sensible way to establish the connections between the task areas, is to use logical connections from any task area to the system and to every other task area. A connection is composed of an input and an output. And these connections should be buffered against the reconfiguration. With an example of n task areas, $2 * n^2$ FIFOs are needed.

$$(2 * n) + (n * 2 * (n - 1)) = 2 * n^2 \tag{4.1}$$

But, as the tasks are only scheduled into the task areas when they are needed, it is equal to use an input buffer or an output buffer. If a task, present on the chip, can produce data and is able to write it to a buffer at any time, one can choose, whether the input buffer or the output buffer is used. The communication buffer do not need both. That's why the output buffer can be leaved out. The output buffer is placed into the task area, to enable the task to write data at any time, but is not needed to ensure the connection during the reconfiguration, this is explicitly explained in chapter 5. This results into $n^2 + n$ FIFOs.

$$(2 * n) + (n * (n - 1)) = n^2 + n \tag{4.2}$$

It is also not recommended to buffer the system separately for every task area, so the total amount of FIFOs can be reduced to $2 + n^2 - n$.

$$2 + (n * (n - 1)) = 2 + n^2 - n \tag{4.3}$$

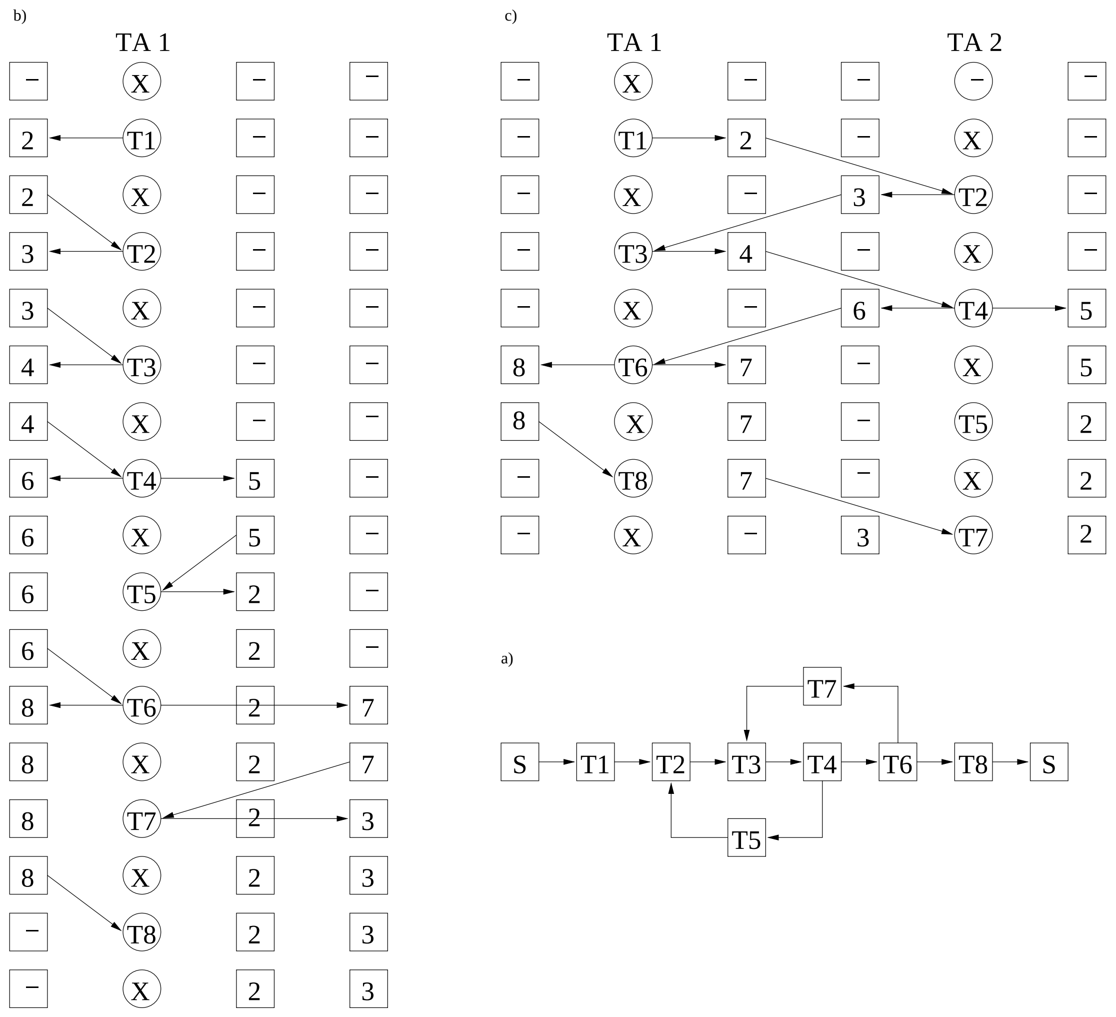


Figure 4.9: A fictional application data flow is displayed in part *a)*. The scheduling of the application is described in part *b)* for one task area and in *c)* for two task areas. The scheduling of tasks will create an additional buffer offset for every data path branch, if the buffer are assigned to the task areas. The affiliation of the buffer to the different tasks must also be accounted by the scheduler.

This concept is illustrated for an example of four task areas, using $12 + 2$ FIFO buffer, in picture 4.10. The task area are connected with each other using two FIFOs, one for every direction. The system inputs and also the system outputs are not separately buffered for every task area, but only once for the serial data stream. The system inputs and system outputs must be buffered both, because the data stream is seen uninterrupted during any reconfiguration concerning the applications point of view.

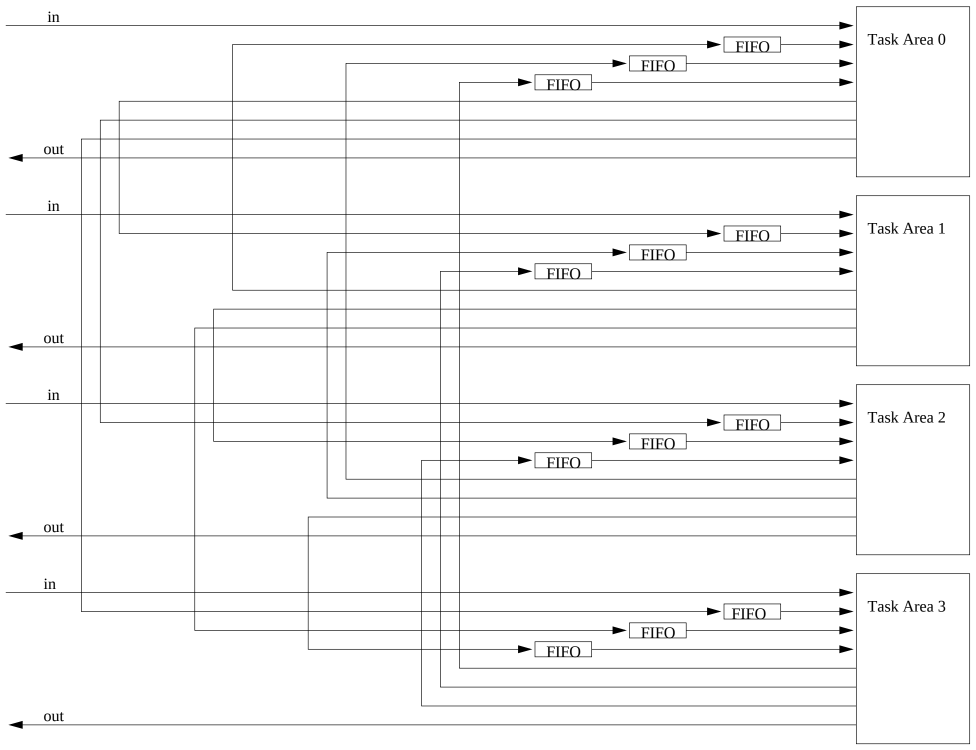


Figure 4.10: The inputs of every task area must be buffered to cover the reconfiguration time. The outputs do not need a buffer, when the inter-task connections are realised as **1:n**.

The task area related FIFO concept is limited for the branching complexity of the application, because every data path will need an additional FIFO. The amount of task areas and additional buffer determines the branch count of the application, but not the POL class count. This FIFO concept can support the dynamic instantiation of hardware objects, but is overstrained, if the data flow evolves branches.

So far about the task area related FIFO concept. On the other hand, the task related FIFO concept means, that a task stays configured on the chip as long as the buffer related to the task contains any data. It is equal for the application and the communication matrix in which task area the task is configured. It is also possible, but not recommended due to the reconfiguration time costs, that a task changes the task area during the execution. The data is no longer buffered for the task area but for the tasks itself, whether for the class or for every instance will become an important decision.

If the buffer are directly related to the POL class or even to the instance, the hardware objects, the absolute number of classes, that can be instantiated on the FPGA is predefined. But the number of instances of every class is not associated to the number of buffer. Well, if the buffer are related to the instances themselves, the trick of short reconfiguration is needed, see chapter 5.1, to achieve this constraint. What is left for the multiplexer of the communication matrix is the connection between the task buffer and the task area containing the dedicated class.

Although it is not possible to include hardware objects of new POL classes in the task related FIFO concept as easily as in the task area related FIFO concept, the task related FIFO concept was chosen, because the flexible connectivity between the hardware objects and the possibility to receive branches, is the main advantage of the POL programming paradigm. Now it is recommended to discuss the task related FIFO structure, due to scheduling, pipelining and the DPR concept. There are in principle two different aspects for the FIFOs leading to eight criteria for a decision, see table 4.4.

The most important criteria is the provided parallelism of the communication matrix. Only, if the multiplexer-FIFO concept can establish a full parallel communication connection between the hardware objects, the hardware realisation of the application will benefit from the fact, that it is not using a bus like structure. As the hardware environment is more conditioned by performance aspects, the communication matrix should use as few resource as possible. Any saved resources can be used for the application. Also the numbers of reconfiguration should be kept low, because the task areas are dead space during the reconfiguration, not available for the application. The needed resources of the multiplexer-FIFO structure depends on the amount of connection between the hardware objects and therefore also on the partitioning of the functionality between the different hardware objects. The last thing, which has to be considered, is that the communication matrix will establish a pipelining structure for the application, that is not visible within the application. That implicates, that the communication matrix is responsible to preserve the order of the exchanged messages across the pipeline structure. This is important for the different connection and also for different objects.

In general, the FIFO buffer are related to the tasks, which represents the hardware objects. But for the detailed evaluation of the multiplexer-FIFO structure, six different way of relating the buffer and the tasks are possible. It is possible to differ between the POL classes and the tasks, which are instances of these classes. If the buffer are related to the classes, every data for all the instances is written into the same buffer, but if the buffer are related to the instances, the tasks, for every task an additional buffer is needed. The stored data of both relating types can also be differed into connection related data and input related data. A connection summarises every input that is operated by the same task, as a single task can write to different inputs of the addressed task. In this circumstances, the data for every input concerning a task or even the class, is stored in one buffer. These six different relation types are listed in table 4.5.

Criteria:	Description:
Parallelism	The structure of the FIFO buffer should serve the application tasks to work in parallel.
Synchronisation on Connections	Exchanged data between different connections should not get mixed.
Synchronisation on Inputs	Exchanged data from one task over different inputs should not get mixed.
Efficiency	The wasted configuration time when scheduling must be kept small.
Changeability	It must be possible to instantiate any task during runtime or change a whole task.
Cohesion	To scatter the functionality, will increase the needed FIFO count.
Coupling	The amount of connections between different tasks will increase dependencies.
Resources	The recommended resources should be less than for a static system.

Table 4.4: This eight criteria are used to evaluate the FIFO structure of the communication matrix.

These different relation types will affect the given criteria for the communication matrix. To give a more comprehending overview of these relation types, some of them are illustrated in figure 4.11 and 4.12.

The first thing to consider are the buffer related to the instance and the inputs. This will cause the messages of one instance to be completely not synchronous, and also the messages between different instances are not synchronised. But this FIFO concept will provide the full parallelism for the instance, this can be call parallelism of inputs. The buffer overhead can be reduced by concerning buffer, related to the instances and the connections. That will also cause the messages of a single instance to be synchronous, but the messages of different instances are sill not synchrony. In contrary, the parallelism is lowered, but still high, it can be called parallelism of instances.

Now the buffer, only related to the classes, have to be reckoned. To use these buffer, related to the inputs, is not useful, because this will make the data flow confusing. But it is useful to make the buffer related to the connections. That means every input of a class, not an instance, will have a separated buffer. This will separate the application parts into different groups, that have to work arbitrarily. Only if these groups are established correctly the messages of the instances and the groups are synchronised. The FIFO structure will save many memory resources with this buffer concept, but on the other hand this will reduce the parallelism of the application, as only the groups are working in parallel and the group parts are executed alternately. Inside of the groups it is not possible to establish

FIFO Structure:	Description:
per instance	All the messages are kept synchon, but the parallelism is restricted to the instance groups.
per instance and task input	The different messages of one or more instances, writing to different inputs of a single instance, synchronised. The instances can work completely in parallel.
per instance and connection	The messages of a single instance are synchronised, but the messages of different instances are not synchronised. The parallelism is reduced by the FIFOs related to every connection.
per class	All the messages are kept synchon, but there is no parallelism left for the execution of the application parts.
per class and task input	The message order will be completely confused, a proper execution of the application is not possible.
per class and connection	The messages of a single instance and of every instance group are synchronised, but the parallelism is restricted to the instance groups.

Table 4.5: This relation types of the FIFO buffer can be realised in the communication matrix.

a pipelining structure, without destroying the data order. This problem can be topped if only one buffer for each class is used, because this will cause the whole application to be in a single group. Although this FIFO structure will need the lowest amount of memory resources, it is not a sufficient structure for the communication matrix, as the application loses every possibility of an parallel execution, and this negates every advantage of the hardware realisation of the application.

For the communication matrix the instance and connection related buffer are chosen, because this concept will ensure the data order and also provide a high degree of parallelism. It is important that the messages can not overtake each other due to the established pipelining structure. The POL developer does not see this pipelining structure of the communication matrix, and therefore can not react to miss leaded messages, caused by the communication matrix. And the communication matrix can provide a high degree of parallelism for the hardware objects, which is most important to use the advantages of hardware against any processor based software realisation.

To provide FIFOs that are generally related to the instances, these FIFO must be placed in the dynamic part of the task area and therefore, they must be reconfigurable. As long as these FIFO types are currently not available, see chapter 8, the only way to grant the data order across the scheduling is to use class related FIFO buffer, under the constraint that a POL class will never send data to two different inputs of a single task and that no POL class will receive data from more than one other task. Although these constraints negates everything said about POL so far, it does not matter, because this does only concern the implementation done in this work and not the general specification and the implemented

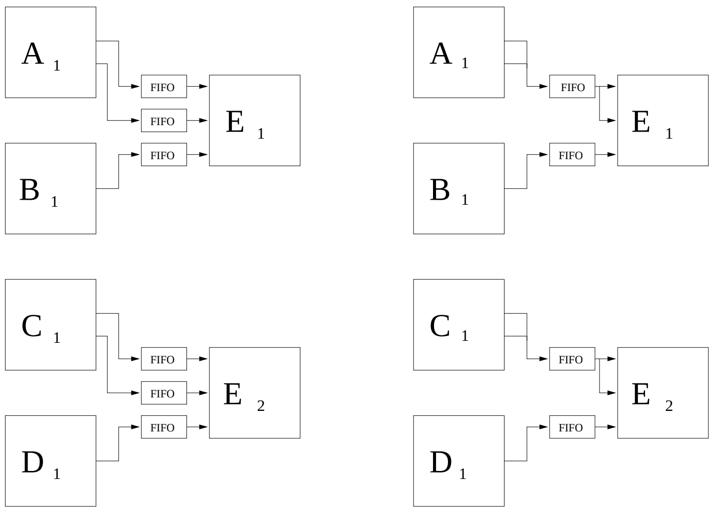


Figure 4.11: If the buffer are related to the class instances, the inputs of the objects and the connections between the tasks have to be considered, in order to evaluate the structure of the communication matrix.

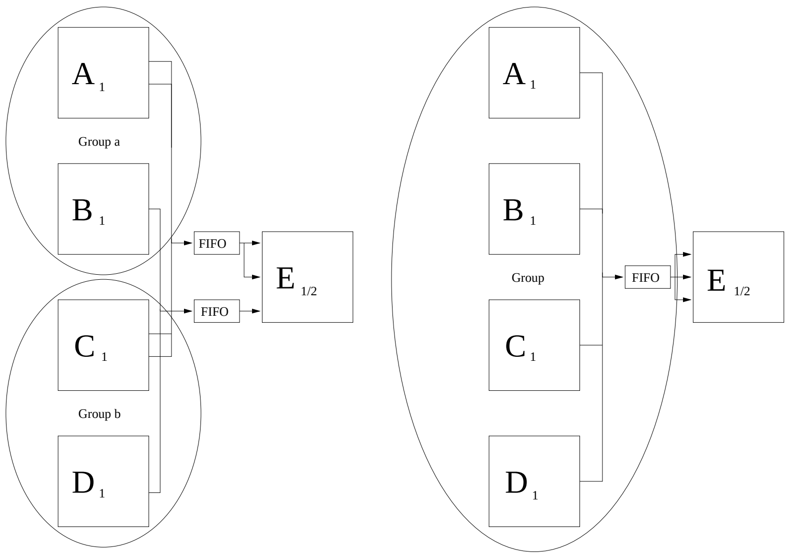


Figure 4.12: If the buffer are related to the classes, the connection have to be considered. A relation to the inputs is not reasonable, because the instances are combined in this buffer.

application also fulfil these constraints from the beginning.

Before the last point, concerning the conditions for the addressing of the tasks, the task areas and the buffer, is described, one additional alternative for the improvement of the memory resource consumption of the communication matrix, has to be mentioned. The needed depth for the FIFOs depends on the data frequency, the reconfiguration time and the amount of needed reconfigurations. So it would be an improve for the required resources, if the FIFO depth can be adjusted dynamically, as it is needed. This would be realised by a BRAM pool from which the FIFO can allocate memory, see picture 4.13.

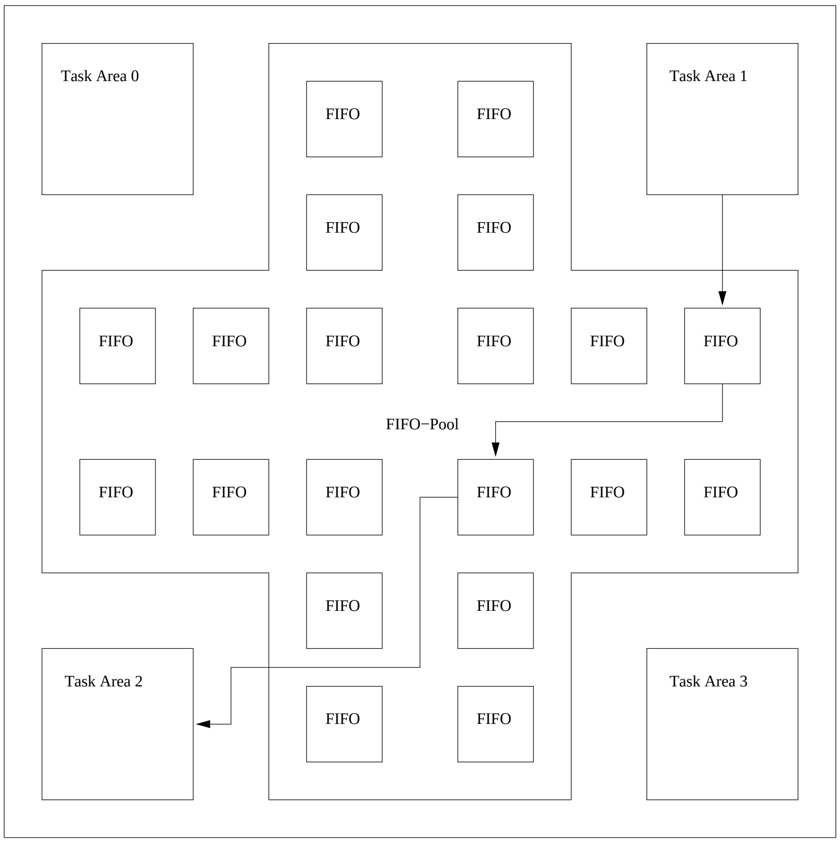


Figure 4.13: The data emergence for any connection is not balanced. To adept the data rate of a connection dynamically would increase the performance of the communication matrix and would help to cover the long reconfiguration times.

In a real environment including an application it turns out, that the contrary case is more common. To save resources, it can be helpful to combine different buffer to one BRAM, this is described in chapter 8.2. If the communication link is only seldom used, the fixed minimal FIFO depth, which is one BRAM, can be far to big. In such a case it can improve the resource usage, if the FIFO depth can be reduced by using for example only a half BRAM. In principle it is not much work to develop FIFOs that can allocate fractions of the

BRAM, but is not so easily to perform short reconfigurations on this specialised FIFOs, as the short reconfiguration always concerns a whole content major frame, and that are four BRAMs.

The last point to discuss is the structure of the addressing inside of the communication matrix. Every task area and every buffer are connected. But only one buffer is important for a task area at a given time, this is determined by the task configured into the task area, where as more than one task area can be important for a buffer, defined by the number of tasks that are currently producing data for tasks related to this buffer. How can the data find a way through the communication matrix to its destination task.

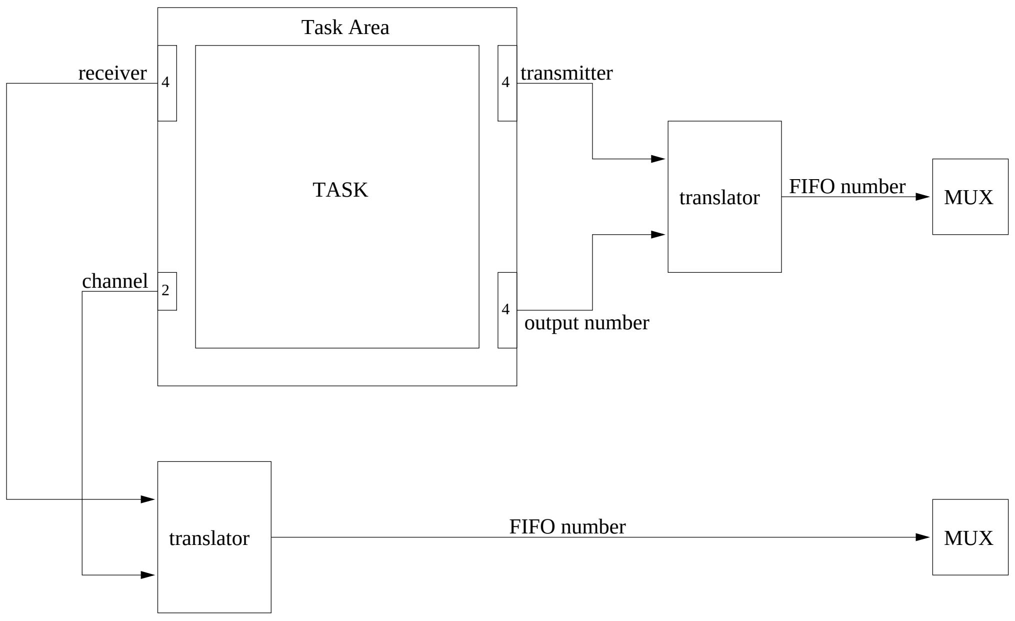


Figure 4.14: An easy attempt to handle the variability of the address space is to encode the task ID within the communication matrix. The translator will use the task ID, the channel number and the output number to generate the correct FIFO number. The data connections between the task area and the communication matrix are not displayed.

The most simplest way is to use the communication matrix to encode the destination of a message. The task presents its ID and an input number to get data or an output number to send data. A translating part of the communication matrix will then decode the desired FIFO number and program the multiplexer. These translation units are designated to be realised in hardware, as a software solution for the PPC would not be fast enough.

The translator functionality of the communication matrix is displayed in picture 4.14. The task announces its ID first as a receiver to get data from the communication matrix and then also as a transmitter to write data to the correct buffer. As an additional information the output number to address the input of the destination object, and the channel number to notice the input that is ready to read data from a buffer, have to be implicated by the translator of the communication matrix, to encode the correct FIFO numbers. If the multiplexer are correctly programmed, the task can exchange data with the communication matrix.

To store the destination with the proper data into the FIFO, is the more valuable approach. This will increase the data width because of the address, or the FIFO needs a additional flag for the address of the containing data. Another possibility is to select the FIFO by its VHDL component name, but this is contrary to any decision made to ensure the dynamic instantiation of hardware objects.

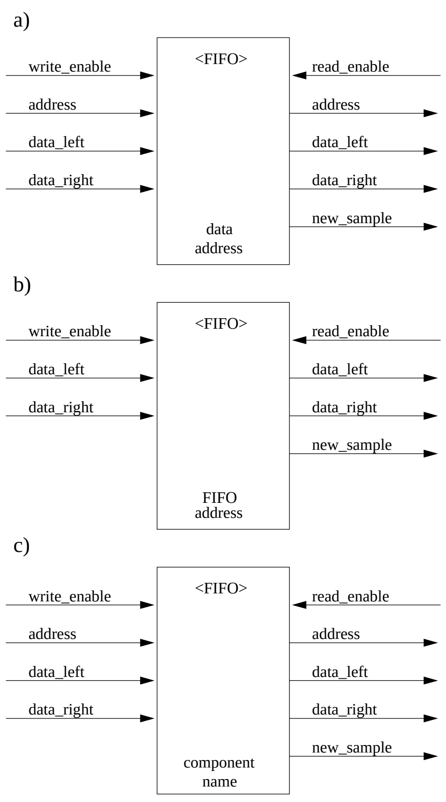


Figure 4.15: The correct FIFO is selected by the data address (*a)*), the FIFO address (*b)*) or the FIFO component name (*c)*).

To store the destination of the data only once for every FIFO will reduce the resource consumption, especially the memory consumption, because of the higher data width, but it will not be possible to merge different FIFOs. But the main disadvantage is the loss of any data verification ability. The storage of the destination data for every data word will enable the buffer to be allocated flexibly by more than one class or, most important, by more than one instance.

As the FIFOs are used for different buffer and the flexibility of the communication matrix is a major constraint, it is sensible to store the whole destination data into the FIFO. In a real application it is not possible to avoid to store major parts of the destination address and the data into the buffer, because we do have to differ between the classes and the real instances, only one FIFO flag for the data destination is not sufficient.

4.3.7 The Scheduling Concept

The whole functionality of the application and the data order is independent from the scheduling. To prove the data flow on these criteria, the transaction concept of data base systems can be used. A transaction T_i is a composition of elementary operations. For POL an elementary operation is the eradication of the *calc()* routine once. On these transaction an order is defined. The application operations have a strict sequential sequence.

With a transaction, a couple of output data is generated with exactly one couple of input data. The amount of steps to generate the output is optional. Within the application we have to consider different histories. A history is the sequence of an interlocked execution of several transaction. The order of the instantiation of task is not determined. Therefore we get many equivalent histories with the same set of transactions. But the histories are serialise able, if one execution order is serial. A more detailed description is given in the most books about data bases, like [AK06].

A data flow is independent from scheduling, if the communication is serialise able. What does this mean for an application using POL and the framework including this communication matrix? As a task can have more than one input and connections to more than one other task, we get multiple data words from different tasks written to different inputs. But the order of data words must be preserved. The scheduler can also let the task calculate at any time and in principle as long as he wants.

Serialisability:
The data order of the communication during the execution of an application is preserved, if there is one execution order, that provides a serial data flow. This is guaranteed by the special buffer structure of the communication matrix for the scheduling.

This gives the several constraints on the buffer structure for the communication matrix, described in the previous section. The communication matrix is specified in a way, that guarantees the data order within the scheduling.

4.3.8 The Reconfigurable Communication Matrix

Contrary to everything described above, it is possible to use a communication matrix based on partial reconfiguration. With the FIFO concept, the produced data is stored during configuration and the multiplexer can be replaced with a dynamic task area. The multiplexer will become a special task, which will provide the connection between the buffer and the task area. As only one connection has to be realised at a certain time, the needed hardware resources to realise the connection are reduced.

But this would increase the amount of reconfigurations, which always requires a lot of time. Additionally the number of partial bitfiles would increase, as well. With the assumed four task areas 12! different bitfiles are needed only for the multiplexer functionality of the communication matrix. In general $(n * (n-1))!$ bitfiles for the required functionality of the communication matrix are needed. But it still gets worse, to create a partial bitfile with PlanAhead we need every possible combination of tasks in task areas and communication matrix states. For $n = 4$ task areas and $m = 16$ tasks we have to compile the whole system much too frequently.

$$(n * (n-1))! + m! = 12! + 16! = 20922789888000 \tag{4.4}$$

This is not only a problem for the realisation of the multiplexer with reconfigurable connections but also for every design containing many task areas and applications consisting of several tasks.

For a real communication matrix realised with dynamic reconfiguration, the possible connections have to be restricted. This will reduce the number of possible task areas and the number of used POL classes in the application. The communication matrix will depend on the compiled time costs and not on the available resources of the used FPGA. This is not an acceptable dependency of the communication matrix, and therefore the communication matrix must be static and use a multiplexer structure.

The main idea of this thesis was not to allow an application, which needs too many resources, to run on the FPGA, but to develop the hardware framework for the programming language POL, which enable the developer to describe any hardware designs with a high level programming language, including object oriented principles and dynamic instantiation. As a result of the dynamic instantiation aspect, it is necessary to provide the DPR functionality for POL. Therefore it is not the intention of the hardware framework to overexcite the DPR functionality of the Xilinx FPGAs.

4.3.9 The Principle Functionality of the Communication Matrix

So how does the communication matrix work in the end for a streaming example? The main functionality of the communication matrix will be the shifting of data between the FIFOs and the task areas across the multiplexer. This is done in parallel for every task area, FIFO and multiplexer.

A typical data flow is presented in picture 4.16. The system contains an input buffer, an output buffer, a single task area and buffer for an application with two tasks. Therefore the application in this example is processing the data with these two tasks. The

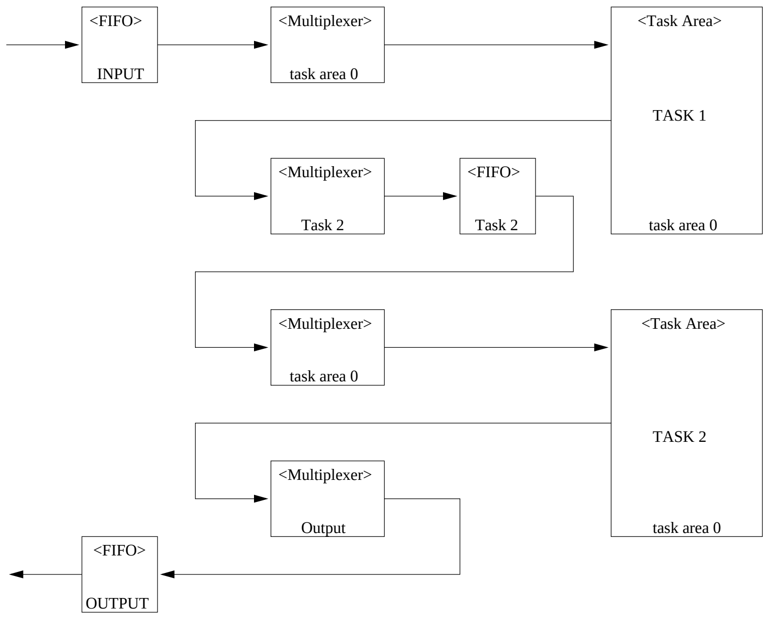


Figure 4.16: The main functionality of the communication matrix is the data shifting between the task areas and the FIFOs across the multiplexer.

data enters the application and is stored into an input buffer, that's the first FIFO in this multiplexer-FIFO design. The multiplexer of the task area knows that task one is present in the task area and shifts the data into the task area. Task one is processing the data and sends the data to task two. This is recognised by the multiplexer serving the task 2 buffer and it shifts the data into the task two buffer. After that, task two is instantiated into the task area, and the multiplexer of the task area switches to the task two buffer and shifts it to the task area now containing task two. Task two also processes the data, but sends the data to the output. This is recognised by the multiplexer serving the output buffer and the data is shifted to the output, where it leaves the system.

The concrete realisation of these concept is described in the following chapter. In this description the explicit interaction between the multiplexer, the FIFOs and the task areas is not clear. Also, the consequences of the different communication types and the scheduling are described in the next chapter.

5 The Specification of the Communication Matrix

In this chapter the concrete specification of the communication matrix is described. The chapter is meant to be a general summarisation of the previous chapter (4.3), and includes the explicit specification of the communication matrix. What is implemented for the evaluation and the prove of concept is described in chapter 6.

The main issue of the communication matrix is to distribute the data between the task areas, which offers the functionality of the application. It is recommended that the communication matrix can handle huge and fast serial data streams and also the control data signals.

In order to improve the resource usage of the FPGA the functional units of an application should only exist, as real hardware, if they are currently needed. Therefore, dynamic partial reconfiguration and scheduling algorithms are used. This explains, why it is not trivial to realise a structure providing the functionality described in the previous chapter.

Although the communication among the functional units of an application is quite serial, the application itself should work in parallel. As a result it is essential to buffer the exchanged data during reconfiguration, which means the individual units are not able to work independently from the communication structure. The scheduler will freely place any task into an available task area. Leading to the fact, that the buffered data has to be tracked to the changing location of the tasks.

But the real reason, why the communication matrix is not trivial, results on the demand for the developer. The developer of an application should not be aware of the data buffering. And also the scheduling of the tasks should not be visible for the developer. The application is not aware of the dynamical environment, either. Nevertheless it must be designed by object oriented aspects including modularisation and the dynamic instantiation of objects. The programming constraints, when using the communication matrix, should not contain any detail based on reconfiguration or scheduling. To be honest, as far as possible.

The DPR environment should satisfy two main requirements for inter task communication. The hardware module, which will perform this functionality, is named communication matrix. The communication matrix is unaffected by any reconfiguration process performed on the FPGA. It provides the connectivity between different parts of an application, the tasks. The connections between the task areas are static. For a given application the connection between the tasks are also static, but highly dynamic for the task areas attached to the communication matrix. There should be no other constraints for the scheduler, when placing an instance on the FPGA, than an available task area.

The development of the communication matrix started with one given constraint. The matrix should remain static in the system to ensure permanent connectivity for application purpose. The final communication matrix is composed of task areas and class buffers. The inputs and outputs of the application (not of the FPGA) are organised in static task areas which cannot be changed during runtime. The components (tasks) of the application are placed in the dynamic task areas.

To assure the application data while some tasks are scheduled by dynamical partial reconfiguration, every class needs a buffer. This implicates that an application is limited to run with a given communication matrix only by inputs and outputs and the number of required classes. Assuming that the resource of the FPGA and especially the resources allocated by the task areas are sufficient.

5.1 Reconfiguration

For the scheduling of the tasks the hardware on the chip must be changed. The application is realised with different instances of classes. In principle, there is no limit for the number of instances of a given task. The tasks are composed of two different configuration aspects. First the hard logic of a task must be configured into a dynamic task area. Also a task has its own content data and inner state, which is different from every other instance independent from the class.

5.1.1 Short Reconfiguration

If the class remains on the chip and only the instance is changed, a short reconfiguration is performed. The content buffer of the tasks is exchanged and the instance is transferred into its old inner state with a special reset procedure.

Short Reconfiguration:
Performing a short reconfiguration, leaves the task logic unaffected in the dynamic area. Only the content and inner state of the instance is changed to the new one. The old content and state must be stored into an external memory.

An additional improve of the short reconfiguration is the fact that it is not bound to the areas dedicated for reconfiguration. The short reconfiguration only affects the content of the BRAMs of the FPGA. This content can be changed independently from the logic configured for the BRAMs.

5.1.2 Long Reconfiguration

If the scheduler wants to exchange the whole class, a long reconfiguration must be performed. The complete logic of the dynamic task area is replaced by the new logic. To do this, more frames of the FPGA configuration have to be changed than for a short reconfiguration, so it will take more time.

Long Reconfiguration:
The long reconfiguration changes the whole logic present on the dynamic task area. But it does not change the content data of the old instance. This must be done with a short reconfiguration afterwards. Any reconfiguration process is finished with a read back of the old instance state from the content buffer.

After a long reconfiguration a short reconfiguration must be performed. This is recommended and must always be done, because the content and the inner state will change as well. This also increases the configuration time, as the short reconfiguration time will add to the configuration time of the long reconfiguration.

5.2 The Communication Types

For any application using DPR we have to consider different types of communication. First of all we have the data exchanged between the different application parts, which are represented by the tasks. Most application performed in hardware have to handle huge data streams. The applications handle these data serial but the DPR environment will work in parallel on the exchanged data. Nevertheless it is possible and also necessary for the tasks of the application, to communicate between each other additionally to this serial data stream. And at last the DPR environment itself needs specific communication. It has to realise the dynamic instantiation of a class and of course the scheduling of the tasks needed for the application.

The particular communication between the tasks of an application has to be grouped into two different communication types, which are briefly described in chapter 4, when the synchronisation of the messages and the general FIFO concept is described. The additional control communication, used for the scheduling of the communication matrix, is also described in this section, after the description of the communication types for the inter-task-communication. These communication types are the serial data stream and the inter-task communication. In this context, the serial data stream is not necessarily a real serial data stream for the application. As the communication matrix and the whole DPR framework will transfer the application tasks into different pipelining steps, the normal communication data of the task can be regarded as a serial data stream. It turns out, that it is possible and reasonable to handle both communication types, the serial data stream and the inter-task-communication, as equal.

The third communication type is used by the communication matrix itself, to address the tasks directly. An important feature of the communication matrix is the scheduling of the hardware objects. If a task should be removed from the FPGA the content and the inner state of the task must be stored. And later, if the instance is again needed for the application the scheduler must be able to recover the preserved state. The content data and the inner state of the hardware object are stored in a special BRAM and are exchanged with a short reconfiguration. To perform this short reconfiguration properly, the scheduler has to prepare the task before the reconfiguration, and must release a special reset after the task is scheduled back to the FPGA. This communication is called task-control-communication and is performed with a special data connection of the communication matrix.

While the scheduler realises the application components in a pipeline structure to cover reconfiguration time, this is not visible or controllable for the application itself. Nevertheless the order of communication messages must be preserved. This is achieved by sorting the inter-task-communication into the serial data stream, when handling the data stream with the communication matrix.

The communication matrix will establish a pipeline structure for the application to cover any reconfiguration dead times. In general, a task will have to produce certain amount of data before the addressed task is scheduled into an vacant task area. The data is stored in the different buffer, and this can cause the message order to get lost, if the task uses more than one input, which will destroy the functionality of the application. The buffer structure of the communication can prevent this.

In picture 5.1 such a situation for different buffer structures is displayed. The first problem occurs, if for example two task writes to also two inputs of a third task. The third tasks needs the data alternately, but the pipelining character of the communication matrix will let the first task produce an amount of data, and after that the second task will produce data. The third task will get the data word of the first task, but the data of the second task will never reach the input of the third task. The second problem concerns the difference between the streaming data and the inter-task-messages. The last ones are seldom used, but if stored in different buffer, the inter-task-messages can overtake the streaming data, due to the FIFOs. Those problems can only be prevented, if a special construct in POL is used, which enables the developer to combine these connections. Therefore the buffer structure must also support this functionality.

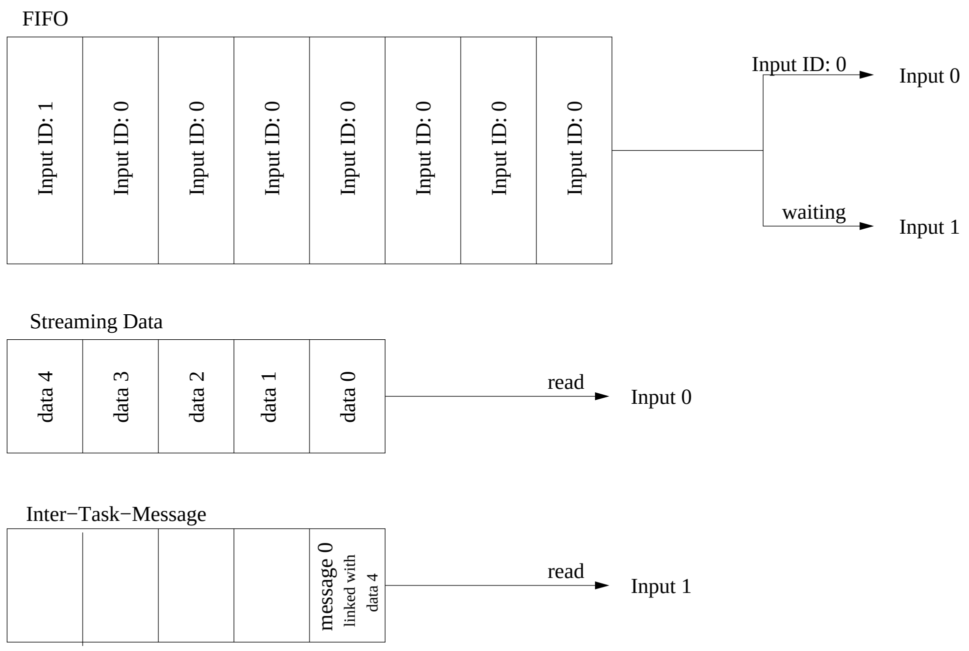


Figure 5.1: Sorting problems of the communication, concerning the pipelining structure of the communication matrix.

In the end both data types, the streaming data and the inter-task-communication data, are equal, due to the special buffer structure of the communication matrix, and the developer must not consider the unique environment of a DPR capable hardware design.

Although it can be possible for the application parts to use a direct task-to-task communication interface, which connects the task areas with each other, it is not feasible for the kind of application, we want to described with POL. The communication between different parts of the application would depend on the scheduling. Even if the direct task-to-task connection can be buffered, to allow the task to communicate with other tasks, currently not present on the FPGA, the participating tasks must be scheduled into dedicated task areas. Therefore this kind of communication is not designated for POL applications, and the communication matrix will not provide a direct task-to-task connection.

Communication Types:

There are three types of communication. Every inter-task-communication is converted into a serial data stream by the communication matrix. The data order must be preserved, when the inter-task communication is sorted into the actual serial data stream. This allows the communication matrix to hide the pipelining character of the communication from the application. And of course the scheduler have to communicate with the tasks, in order to perform a neat exchange procedure for the tasks, that will be the task-control-communication.

If the tasks communicate among each other, using the static part of the system, especially the communication matrix, they can exchange data with tasks, which are currently not available. No one has to know the current where about of a task. It is equal, whether the task is present on the chip or stored into the memory of the scheduler. And even if the task is present on the chip, it does not matter in which task area the task is configured. For a static application the scheduler can know the data flow among the tasks, but if it is possible to create new tasks during run time and connect them freely, the data flow is far too complex for the scheduler to keep track. So it is mandatory for the scheduler, that the tasks are not dedicated to run in a single task area.

As mentioned above, there is no difference for the application between those communication types. Nevertheless there is a logical difference and this was most important for the development of the communication matrix and POL. The communication matrix can sort the inter-task-communication data, because the buffer structure is generated by POL to serve the application correctly. It is equal whether the real streaming data or the data stream generated by the scheduling are sorted by the buffer structure, both is specified with POL. Therefore both communication types are differed and described.

5.2.1 The Serial Data Stream

Most applications realised in hardware have to handle huge data streams. Thus, it is easily anticipated, that most of the applications dedicated for the use of the communication matrix are streaming examples. That means, the application has to work on a serial data stream with a given width and frequency. The data stream enters the FPGA, is processed by the application, and leaves the chip afterwards. Additionally the communication matrix will also transform any kind of communication between the tasks into a serial data stream, by transferring the tasks into different pipelining steps.

The different application parts, which are processing the data, must be able to work with a higher frequency than the sample rate of the serial data stream. The processing tasks are scheduled during runtime, due to the functionality of the task. Every reconfiguration disables the concerned task area and the data stream must be buffered, to prevent any data losses, and only the higher working frequency of the tasks can prevent an interruption of the serial data stream.

5.2.2 The Inter-Task Communication

In contrary to the serial data stream mainly coming from the FPGA periphery the tasks themselves should be able to exchange control messages. This communication type is used seldom compared to the streaming data, but for the tasks, it is always synchronised to the data stream in spite of the pipelining character of the communication matrix. The communication matrix must keep the relation between the data stream and the control messages. The difference between both communication types is a simply logical one, and the communication, as seen from the application, is not different for both of the communication type.

Although the inter-task communication is also transformed into a serial data stream, the processing time is not so crucial for the inter-task message itself. Most important, the communication matrix will keep the relation of the inter-task communication and the serial data stream.

5.2.3 The Task-Control Communication

The task-control communication is completely unaffected by the inter-task communication and the serial data stream. This communication type is used by the scheduler to communicate with the tasks. The inner state and the content data of a task must be preserved during the scheduling. Therefore the task must be forced into a special state, that allows the removal of the task and a clean resuming, if the task is scheduled back to the chip. For this purpose the communication matrix provides this special communication type.

The scheduler tells the task, that it will be removed from the FPGA. Now the scheduler must wait, until the task has saved its inner state and the content data. If the task is ready, the scheduler can exchange the tasks. After the task is written back to the FPGA, it must be transferred into its last state with a special reset for the dynamic area.

5.3 The Task Area

Most important for the developer using POL and the framework developed in this work, is the sufficient granularity of the application. POL is constructed to use object orientation. Therefore everything is centred on objects. But using the communication matrix, as the main functional component of the framework, the granularity is somewhat different.

The application can be divided into functional groups which run in parallel and are able to be scheduled. These groups are called tasks. A task is an instance of a POL class and would be better named dynamic hardware instance. But, as it is scheduled between several areas on the chip reserved for dynamic hardware and called task areas, the name task has established itself.

Task Area:
The dynamic partial reconfiguration is only performed in designated areas of the FPGA. These task areas are completed from the system by special structures, the busmacros.

A POL class is in principle a JAVA class extending thread, enhanced with special hardware specific structures. And this is also the analogy between POL and object oriented programming languages. In principle, a thread is equivalent to a task.

Task:
The tasks are the main application parts that are scheduled on the FPGA. They are communicate with each other and with the System via a communication matrix. All the generated or instantiated tasks build up the whole application.

That implies that the application consists of components, which are working in parallel with one another, and communicates with each other using a fixed interface.

Now it is mandatory to have a closer look at the task area. The static kind of the task areas is used for the input and the output of the application, representing the system interface. And the dynamic kind of task areas provides the functionality of the instantiation of tasks during runtime with dynamic partial reconfiguration.

The interface of the dynamic task areas are standardised, but the interface of the static task areas depends on the input or output functionality, provided to the application. Just imagine a *led* as an output and a *pushbutton* as an input, or even an *uart*, using both, an input and an output. Both task areas will write or read the defined data words, but the input data or the output data is different. A static task area will always provide an output or an input, but never both of them. So for the *uart* two static task areas are needed.

The task area consists of three components. One Multiplexer, a FIFO to store data and of course the task slot. For the dynamic task area the task slot is part of the reconfigurable regions of the FPGA, and for the static task areas, the task slot is also static. As mentioned in the previous chapter 4.3, the FIFO buffer of the task area is used to separate the writing functionality of the task from the state of the communication matrix and the application.

A task area can have several class buffer as inputs. That's why it needs a special kind of multiplexer at the task area input. This multiplexer monitors the class buffer of the communication matrix and, if the multiplexer finds suitable data, it starts to read from the class buffer FIFO. Therefore the FIFOs must provide a first word fall through (FWFT) functionality. This enables every multiplexer of all task areas to read the data of the class buffer, but only the corresponding multiplexer will give an acknowledge to the data. The data words are passed to the task inputs. The accordant task input has a control signal, to announce that it is ready for the next data word to the multiplexer. This also needs the FWFT functionality of the buffer.

After the task has processed the input data, the output data is written into the FIFO of the task area. That permits the task to produce data independently from the supporting framework. It is already mentioned that an application should be unaware of the communication matrix state and functionality.

5.3.1 The Static Task Area

The periphery of the FPGA, that are the input pins and output pins, have to be encapsulated in order to ensure the use of the data encoding of the communication matrix. This must be done by using static task areas. The interface of the FPGA, like the structure of the communication matrix can not change during runtime. That implicates that the static task area do not need the multiplexer. The static task area, containing always the same task, is only connected to one specific class buffer.

Providing the interface to the FPGA, we have to differentiate between the inputs and the outputs. The inputs will generate data and massages consistent with the communication matrix protocol and the outputs will receive those data and massages. So the static input task areas must contain the task area output FIFO, the outputs do not. Otherwise the static output task areas have to define their own output interface and the static input task areas their own input interface.

In picture 5.2 the functionality of a static input task area and a static output task area are displayed. In the POL world only the message type of the communication matrix exists, and the tasks of these static task areas provide the translation of the I/0 communication into the data messages of the communication matrix.

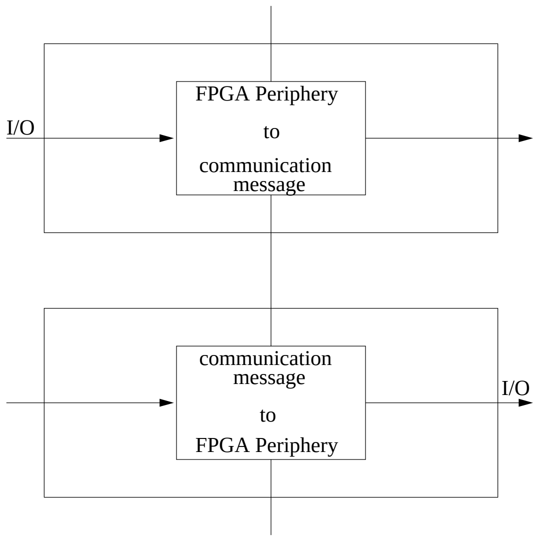


Figure 5.2: The static task area contains the I/O logic. The system input data is translated into the communication matrix data format and the system output is generated from the data words within the application.

This separation of the I/O communication is not necessary for the functionality of the communication matrix, but a main purpose of POL is the re-usability of the DPR designs. If the developer have to construct the application periphery for every new project repeatedly, the usage of POL is abated. Another advantage of this periphery structure, is the explicit de-serialisation of the I/O communication. The developer has a complete knowledge, which I/O communication is parallel or sequential.

5.3.2 The Dynamic Task Area

The dynamic task area is far more complex than any static task area or class buffer. It contains not only the output buffer, the input multiplexer and the task slot, but has to handle the reconfiguration domain crossing (RDC). For optimal use of reconfiguration technologies the components of the dynamic task area are separated and placed in different layers of the VHDL structure.

The multiplexer and the FIFO are placed in the static part of the design, where as the task area is placed in the dynamic part. The distinction between static and dynamic, according the design, refers to the design hierarchies described in the DPR tool flow (section 2.2.1) and not to the functional distinction of the task areas. The crossover between both is occurs through the busmacros, which unfortunately have to be placed in the top module. This causes the data messages, to pass the full design hierarchy two times, when shifted from the input multiplexer to the output FIFO.

In picture 5.3 this crossover is displayed. The data path begins in the static part of the dynamic task area, where the data for the configured task is present in one of the buffer connected to the multiplexer. The multiplexer passes the data over the busmacros into the dynamic part of the dynamic task area. The supporting frame of the task slot sorts the data into the correct input register and acknowledges the data transfer over the busmacros and the multiplexer to the buffer. Now the data is available for the task, which is configured into the task slot. After the processing of the data, the task can directly write into the output buffer of the task, again over the busmacros. If the output buffer is full, the scheduler must suspend the task.

The design hierarchy needed for the PlanAhead tool flow, makes it hard to understand the functionality of the dynamic task area. But in principle, the dynamic task area only takes data from the remaining communication matrix, waits until it is processed by the task, and writes it back to the communication matrix. The really complicated part of the dynamic task area are the inputs of the task slot. These are realised with the reconfigurable instance buffers, which are described in the next section.

5.4 The Buffer

The reconfiguration takes quite a long time to proceed, compared to the clock frequency of the design, which defines the calculation time for the application. Therefore it is necessary to buffer the data produced for a given task during reconfiguration. Additionally

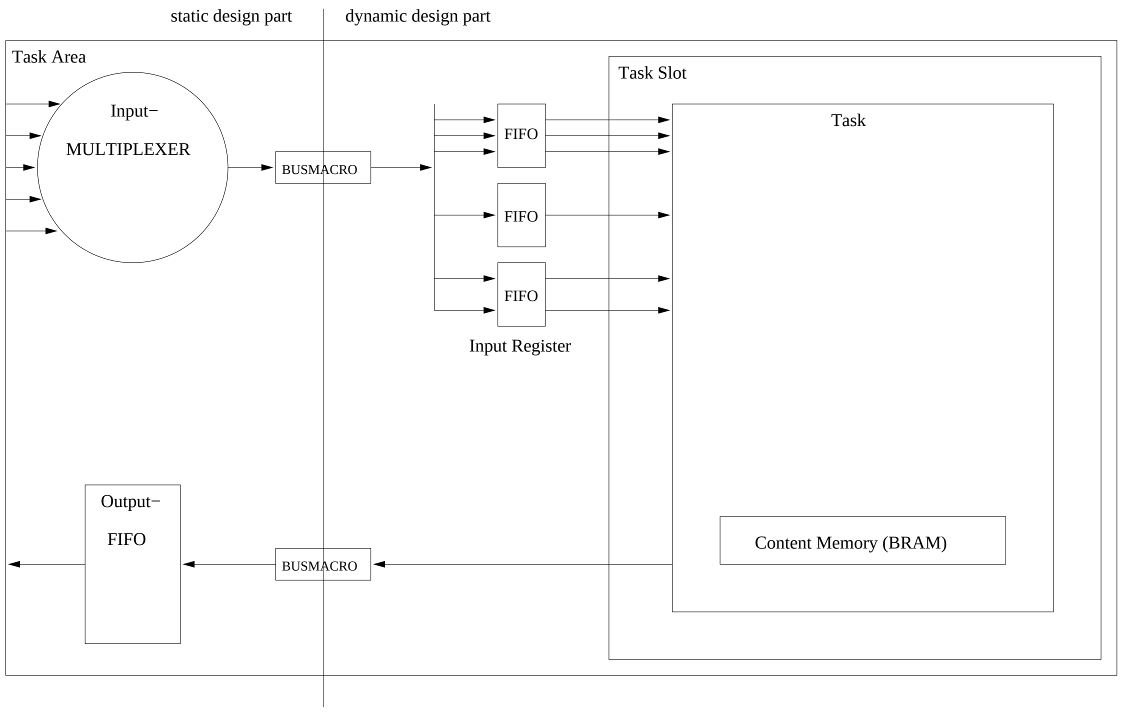


Figure 5.3: The dynamic task areas defines the periphery of the application tasks. They provide any functionality needed by the tasks. It sorts the data into the inputs of the task and provides the buffer for the output of the task.

the resources of the FPGA should be shared between different tasks. So it is estimated that a task may be not available when data, for this task, is available.

Nevertheless, from the point of view of the application, these latencies must be covered. This is performed by different buffers which are included into the structure of the communication matrix for this purpose. It is recommended that the buffer in co-operation with the matrix are not able to distort the data flow and especially the synchronisation of the data. The order of the messages, send and received between the tasks, has to persist. This is most important speaking about pipelining the different tasks.

Configuration Buffer:
The possible serial data steam and the application in general must be as visible as uninterrupted, although the individual tasks of the application are scheduled. Therefore the data of the application parts has to be buffered during reconfiguration. This makes it possible to present the application uninterrupted.

In the end, three different buffer are used. The first buffer is used at the output of a task area, which allows the tasks to write data independent from the state of the matrix or the application, and is called task area buffer. The second one pre-selects the data from the task areas and sorts them by classes, and therefore is called class buffer. This makes the first buffer more independent from the scheduling. The third one, most important for the message order and the event order, is specific for every instance and therefore, included into the dynamic part of the dynamic task area, is also reconfigured with a short reconfiguration, along with the task. This buffer is called instance buffer.

5.4.1 The Task Area Buffer

The individual tasks should not know anything about the environment and even be independent from the state of the environment and the scheduling state of the application parts. Therefore the task must be able to produce any amount of data at any time.

The communication matrix itself must ensure this interdependency. The individual multiplexer are working self-sufficient and in parallel. So they can not be forced to fetch data from a given task area or task. Thus it is recommended to buffer the data, produced by a task, inside of the task area, before presenting the data to the communication matrix.

This role is taken by the task area buffer as a part of the task area. The width of the buffer is defined by the objects word width, but the depth is in principle not determined. It depends on the amount of produced data and the ration between the tasks and the task areas. The multiplexer is not controllable and the depth of the buffer can only be estimated. This question is left for optimisation considerations, which are not discussed in this work.

The FIFO of the task area buffer must present the next data word to any multiplexer, in order to allow the right communication matrix part a data readout. Therefore the task area buffer must include a first word fall through functionality.

The task area buffer is simply a FIFO, which realises the FWFT functionality and nothing more. The principle design of the task area buffer is displayed in picture 5.4. The task area buffer separates the output of the task and the input of the communication matrix with a FIFO.

The functionality of the task area buffer will allow the task to produce a data word with every clock cycle, what ever the communication matrix or the other tasks are currently doing.

5.4.2 The Class Buffer

The tasks are placed into the task area or sourced out arbitrarily. This implicates, that the output FIFO of a task area can contain data words of many different tasks, and the data words can occur in any order. The scheduling of the task should be done according to

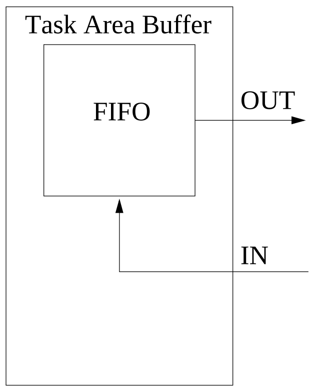


Figure 5.4: The task area buffer are needed to simulate an un-interrupted connection between the tasks. In this case the point of view of the output is covered. The task output is independent from the state of the communication matrix and other tasks.

the amount of data available for the dedicated task. The data word count is estimated, to decide which task has to be instantiated on the FPGA. Unfortunately, this is not possible within the task area buffer, because of the possible mixture of the data words, produced by different tasks. The data is pre-sorted into the class buffer to improve the scheduling performance, and to keep the task area buffer empty.

The class buffer has a multiplexer and also a FIFO. The FIFO provides the same functionality as the task area buffer and the multiplexer assures that the FIFO only contains data from the associating class. This is done by checking every task area output for data with a suitable class ID and transferring the data into the FIFO.

In picture 5.5, the design of the class buffer is displayed. Every task area is connected with the multiplexer, which writes the data concerning the dedicated class into the FIFO. After that, every task area can read the data from the FIFO.

In contrast to the task area buffer the class buffer is a complete and independent entity of the communication matrix. It permits the scheduler to decide which class is to be instantiated on the FPGA by means of data quantity. Therefore it provides the *fifo_full*, the *fifo_empty*, the *read_count* and the address of the next data word to the scheduler. The last point, the address information of the next data word, is used by the scheduler to detect deadlocks and miss -leaded data words.

In principle the class buffer is not necessary for the functionality of the communication matrix, if the instance buffer, which are described in the next section, are completely implemented. But the class buffer can improve the performance of the communication matrix. The data is pre-sorted by the classes and therefore it cannot happen, that long reconfiguration are performed alternately. Nevertheless, the class buffer can not prevent an alternating execution of short reconfigurations.

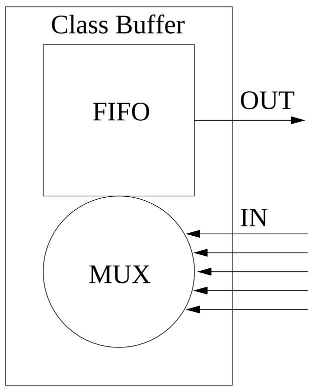


Figure 5.5: The class buffer is needed to presort the data of the instances by classes. This is necessary because of the short reconfiguration performed with every instance swap. Although the short reconfiguration is faster than the long reconfiguration it takes some time.

5.4.3 The Instance Buffer

The instance buffer is a compromise between the maximal pipelining performance and the resource consumption of the communication matrix. Certainly it is most efficient for pipelining to use a buffer for every instance instead of using a class buffer. This implies two major disadvantages which are influenced by each other. In the case of dynamic instantiation of objects, the instance buffer results in a resource consumption which is not able to satisfy. On the other hand the limitation of the used resources makes it impossible to use dynamic instantiation of objects.

The class buffer prevents the excessive performance of long reconfigurations. The class buffer only contains data for a single class. Therefore the scheduler can realise the executing data flow into a pipelining structure. But the class buffer can not provide a serial execution flow of the data words within this pipelining structure.

The needed compromise is the instance buffer. The instance buffer is part of the frame of the task area, but individual to the task, which is located within the reconfigurable region of the FPGA and also part of the dynamic task area. This will leave the resource consumption independent of the quantity of instances. The dynamic instantiation of instances does not influence the needed resources for the matrix any longer. Remains the reconfiguration costs of frequently performed short reconfigurations in order to sort the data in the correct data flow, which can be disturbed by the pipelining process.

The instance buffer are nearly equal to the task area buffer comparing the data storing functionality. They contain a FIFO with first word fall through functionality. But there is an important difference. The FIFO of the instance buffer must be reconfigurable, as it is part of the dynamic part of the task area, although belonging to a task. That means the content of the included BRAM, the actual read address and the write address must be stored into the DDR ram with a short reconfiguration.

The instance buffer must be a customised buffer. The FIFO of the instance buffer must be based on a BRAM, to be available for a short reconfiguration. The buffer must also store the read address and the write address of the FIFO into the BRAM. After a short reconfiguration is performed, the FIFO of the instance buffer can recover the read address, the write address, the read count and the FWFT functionality with the dynamic reset procedure, which must also be implemented. The general design of the instance buffer is displayed in picture 5.6.

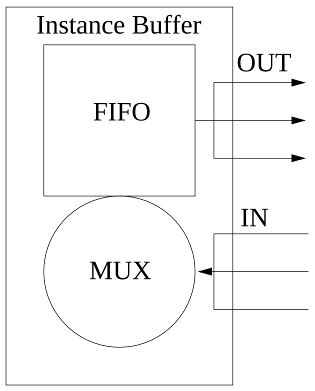


Figure 5.6: The instance buffer are needed to keep the data ordering of the application and to improve the performace at the same time. This buffer realises the FIFO structure described in section 4.3.

Another feature of the instance buffer is the preservation of the data order. To keep the data order, it is possible in POL to combine different inputs to one connection. The data for these combined inputs must be written in only one buffer. The instance buffer must therefore have a kind of arbiter, which serialises the access of the FIFO. Behind the FIFO, the data must be sorted back to the different inputs.

This reconfigurable instance buffer can improve the reconfiguration time cost and the resource consumption of the communication matrix. Nevertheless, the most important point is, that the instance buffer enables the scheduler to establish a pipelining structure for the application, but the communication matrix is still capable to preserve the data order of the application.

5.5 The Task Interface

The task itself, developed with POL, must serve a special interface. This interface consists of two different connections, the actual data connection with the communication matrix and the control connection with the scheduler, realised with the DCB.

The data interface defines a fixed number of inputs and one output for the task. The width of every input and the single output is defined by the POL developer. It is very important, that the task will acknowledge every data word it reads to the task area, otherwise the task area can not refill the input register. On the other hand, the task area will indicate the new data to the task, it do not have to detect the new data on its own, because of the pipelining structure of the communication matrix. Therefore every input has a *new_data* flag, which is set by the task area and a *acknowledge* strobe signal, which is handled by the task. The inputs are filled parallel by the dynamic task area of the communication matrix and the *new_data* flag goes high. The task can decide when to read the input data and acknowledge the read with the *acknowledge* strobe signal.

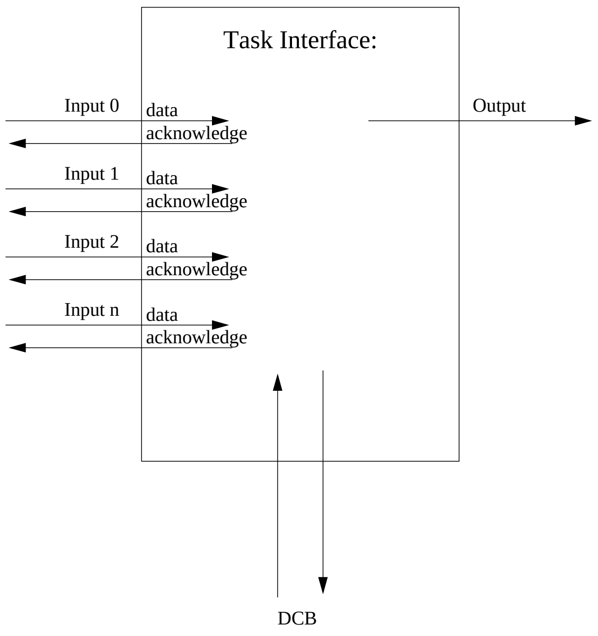


Figure 5.7: The task interface must be obtained by every task generated with POL. The exact number of inputs and the data width is configurable. And also the width of the bidirectional DCB can be defined be the POL developer.

The control interface connects the task with the scheduler over the DCB. With this bus the task is prepared for any reconfiguration by the scheduler, and the task can instantiate any hardware object available by the system. Therefore, the DCB is a bidirectional bus. The concrete realisation of the DCB depends on the implementation of the concrete design. The width of the class ID and the instance ID, which are needed for the instantiation of hardware objects can differ from design too design, and therefore the DCB will also change, because the ID's should fit into the DCB width.

The principle design of the tasks interface is displayed in picture 5.7. A more detailed description with a concrete example realisation of the task interface is given in picture [?] of chapter 6.

The output data is directly written into the output FIFO of the task area, which is, as mentioned above, always possible. The concrete width of the data words of the tasks inputs and of the DCB can be chosen freely during the development of the application with POL. The most important part is the acknowledge concept of the inputs. A task must serve this concept, because the communication matrix will establish a pipeline structure for the hardware objects and this will help to prevent deadlocks.

5.6 The Scheduler

The scheduler should decide which task has to be instantiated concerning the filling level of the different FIFOs. A task should be present on the chip, if enough data is available for the appointed task. This is necessary because every reconfiguration is associated with performance costs. In this case the short reconfiguration and the long reconfiguration will behave different in one point. The long reconfiguration is executed, concerning the filling level of the class buffer of the communication matrix, but the short reconfiguration is constrained by the instance ID of the data inside of the class buffer. This will cause the short reconfiguration to be performed very frequently.

To illustrate the data based scheduling for the short reconfiguration and the long reconfiguration, a simple example is displayed in picture 5.8. A short reconfiguration must be performed, every time the instance ID of the data contained in the class buffer changes. In the worst case, this happens after every read operation of the task multiplexer. The long reconfiguration is performed by the scheduler concerning the filling level of the class buffer. It must be performed, if the buffer of the currently active task is empty, or the overflow of a different class buffer is imminent.

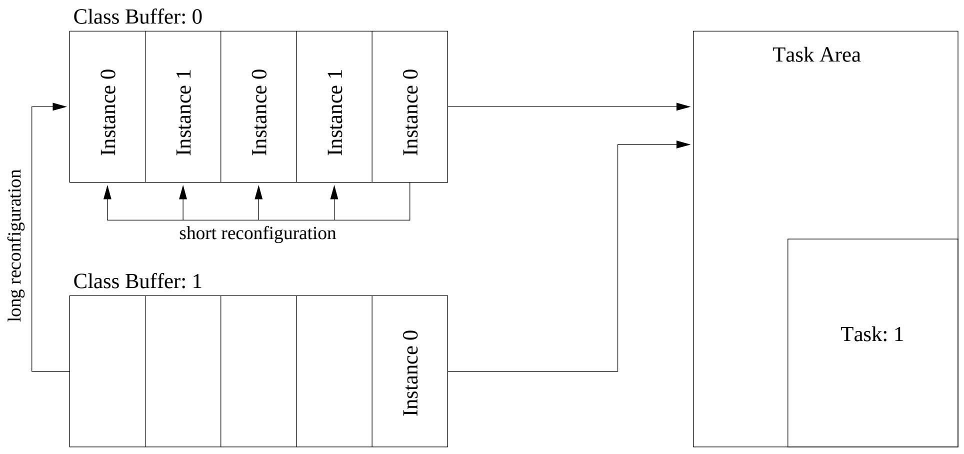


Figure 5.8: A simple scheduling example to demonstrate the difference of short and long reconfigurations.

Using the instance buffer, it is possible, that the proper task is kept idle, but the instance buffer are scheduled alternately for the task, to fill in the instance data. This will prevent the task to be capable for any procession during the short reconfigurations, until the scheduler decides, that the data filled into the instance buffer is adequate.

This kind of scheduling will generate the pipelining structure for the application parts communicating over the communication matrix. The generated data messages are collected and stored into the buffer. This will allow a task configured on the FPGA, due to this data collection, to process the data in some kind of a burst. This is recommended to cover the reconfiguration time needed to place the task on the FPGA.

The second role of the scheduler is the dynamic instantiation of tasks, which represents the instances of the hardware objects specified with POL. Therefore the scheduler must manage the different instance IDs of the different POL classes. The tasks can instantiate any POL class during run-time and will tell the scheduler which class they want to instantiate. After that, the tasks expect from the scheduler the instance ID of the new instantiated class. For the class it is not important, whether the new tasks are configured on the FPGA instantly, due to the Buffer, but the scheduler must take care, that the task is configured, when data is available.

Scheduler:
The scheduler is a component of the system, deciding which task is currently running on different task areas. It makes it possible to allocate more hardware resources than available on the FPGA.

The scheduler is responsible for the functionality of dynamic instantiation, required by the POL classes, and the covering of the reconfiguration times, establishing a pipelining structure for the communication. Nevertheless the communication matrix is explicit developed to serve the scheduler, concerning those two responsibilities.

6 Implementation

The specification of the communication matrix is now adequate documented. But to verify the resource and performance improvement, a concrete realisation of the communication matrix is needed. This chapter will describe how the communication matrix specified in chapter 5 is implemented. This implementation is used afterwards for the verification of the communication matrix, as specified for POL, in chapter 7. The concrete implementation of the communication matrix will not only be capable for the verification of the concept of the communication matrix, but can also be directly used for general POL project, except from the static task areas for the input and the output of the application. The used periphery of the chip and any other input or output of the application are always individual for the application.

6.1 The Demonstration Application

The demonstration of the communication matrix concept for POL is done with an example audio application on a *ML405* evaluation board from Xilinx with a Virtex-4 XC4VFX20 device. Most Xilinx evaluation platform include an *ac97* analogue to digital converter for audio data. For more information about the ML405 evaluation board see [Xil08a]. The evaluation board can be placed between an audio player and a speaker, and the digitalised audio data is manipulated on the FPGA, see picture 6.1.

The intention of this implementation of the communication matrix for this audio application, is to practically evaluate the functionality of the communication matrix. Although the reasons for the development, described in chapter 4.3 and the specification in chapter 2 legitimates the presented hardware framework, it is recommended to prove this concept with a real example implementation.

In a former practical project, four different filter for the digital audio data stream have been developed. The filter represents a distortion, a high pass, a low pass and an echo. This filter are realised in different VHDL modules, which makes it very easy to adapt them for a DPR application. Only the interface has to be changed, to enable the filter to communicate with the communication matrix as tasks.

The most important thing for a developer implementing with POL, and using the communication matrix, is the interface provided for the task and the format of the data words. In principle the width of a data word of the communication matrix can be chosen by the

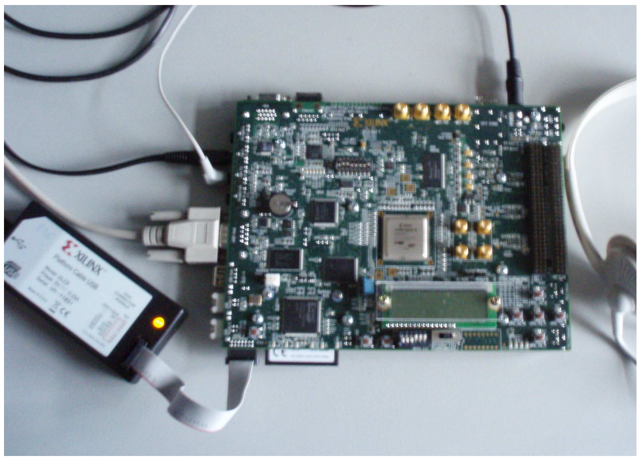


Figure 6.1: The used test setup for the audio application

developer. While using the BRAM blocks on the FPGA it is reasonable to use a 32 bit data word, because this is the maximal data width of a single BRAM. This will make the most efficient use of the resources on the FPGA.

The *ac97* generates a 32 bit audio data stream, which contains 16 bit for the left audio data and 16 bit for the right audio data. For the evaluation of the communication matrix, the left and right audio data are separated and also processed by different tasks. These task represents different instances of the same filter. This is used to prove the usage of different instances of the same filter, and therefore the functionality of the class buffer. While the development of the instance buffer are currently in progress, the different audio data is pipelined by the input tasks, to prevent a permanently scheduling of the instances, and therefore the execution of short reconfigurations.

The application provides the user to choose which filter and how many of the filters will process the audio data. It is also possible to change the execution order of the filter. This is done, for the example application, with only one task area. Nevertheless the audio stream should sound like uninterrupted, due to the buffering during the reconfiguration of the filters. This is displayed in picture 6.2.

The selection of the different filters is done with a simple GUI running on an external PC, displayed in picture 6.5. The number of filters and the filter order is transferred to the scheduler running on the PPC of the FPGA. This can be done during run-time, and also without a hear able interruption of the audio data stream.

It is intended, that all four filters are configured alternating for the same audio stream, but the audio stream will occur uninterrupted for the application and the user. In this application the dynamic instantiation is actually not explicitly used. But the main issue of the communication matrix is the establishing of the required communication structure, which is also crucial for the dynamic instantiation of POL classes. This functionality is

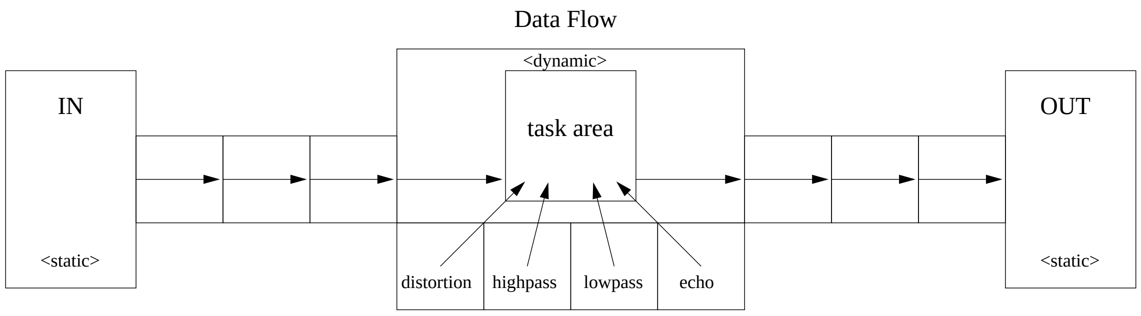


Figure 6.2: The example application can use up to four filter, to process a two channel audio stream.

proved with the changeable filter order of the audio application. The communication matrix and also the POL programming language can react to the changing connections between the application parts.

6.2 Implementation of the Communication Matrix

As long as POL is not completely developed, the communication matrix must be implemented by hand. This is not intended to be necessary for the use of the final POL development environment.

The communication matrix used for the example application includes only a single dynamic task area. For the audio input two static input tasks are used, one for the left audio stream and one for the right audio stream. The execution order of the different filter is assigned by the user and therefore an additional input for the application is used. This additional input forwards the connection and reconnection orders, given by the user. The audio output is also realised with two static output task areas. The complete audio stream is merged outside of the communication matrix. This is done, to explicitly use the full parallelism of the application and the hardware framework, preventing a serialisation inside of the communication matrix. For every task of the four filters a separated class buffer is included. And also the static input task area need a class buffer for the connect and reconnect orders. The output task areas will also get a class buffer, as they do contain a task as well. But this is in principle not necessary using only one dynamic task area, because the multiplexer is not needed. The scheduling is running on the PPC of the FPGA and is connected to every communication matrix component, to control the data flow and the reconfigurations. Additionally the DCB is implemented, which is described in section 6.4. The composition of the communication matrix for the audio application is displayed in picture 6.3. The additional reconnection task area, which will occur in the source code of the communication matrix, is not displayed, as it is a application specific realisation of a direct user access, which is only necessary to demonstrate the functionality of the communication matrix, but is not recommended for a real POL application.

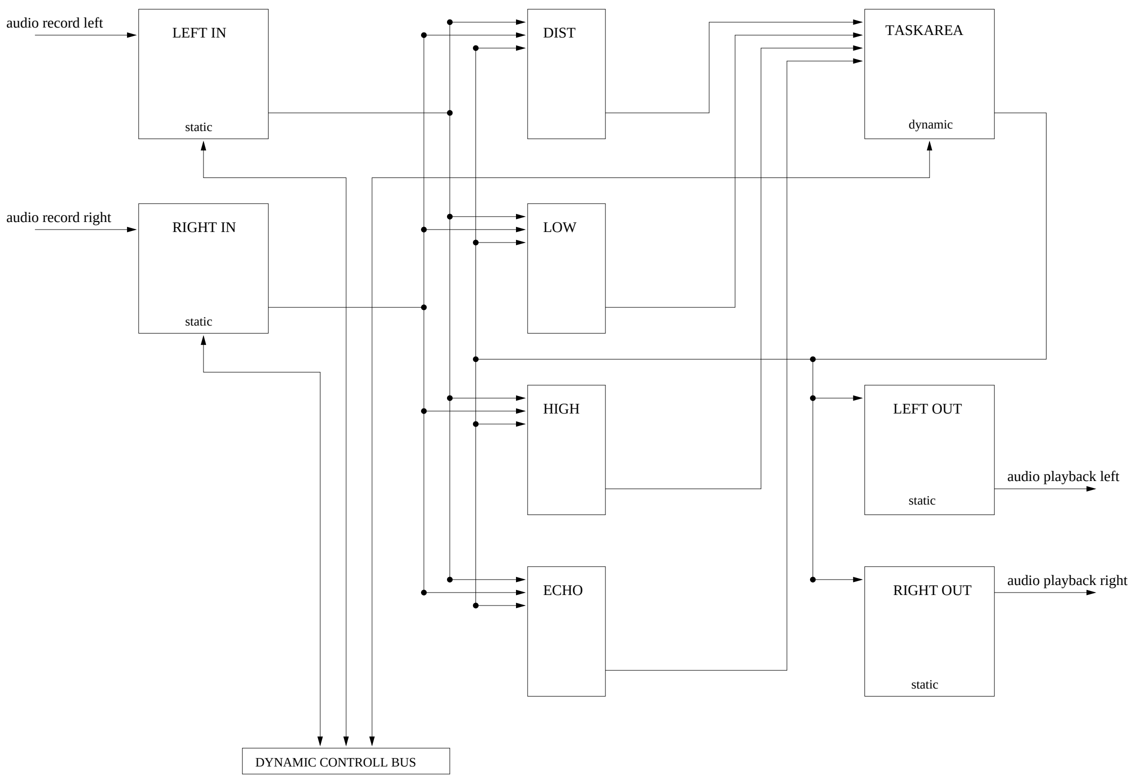


Figure 6.3: The example implementation includes one dynamic task area, two static input task areas, two static output task areas and eight class buffer. Therefore, the system can provide one task slot and four classes can be instantiated.

The actual source code of the communication matrix and its components is completely available on the included CD, see appendix B. The main functionality of the implemented communication matrix is the parallel interconnection of the task areas and the class buffer. The remaining functionality is provided by the sensible scheduling of the hardware tasks.

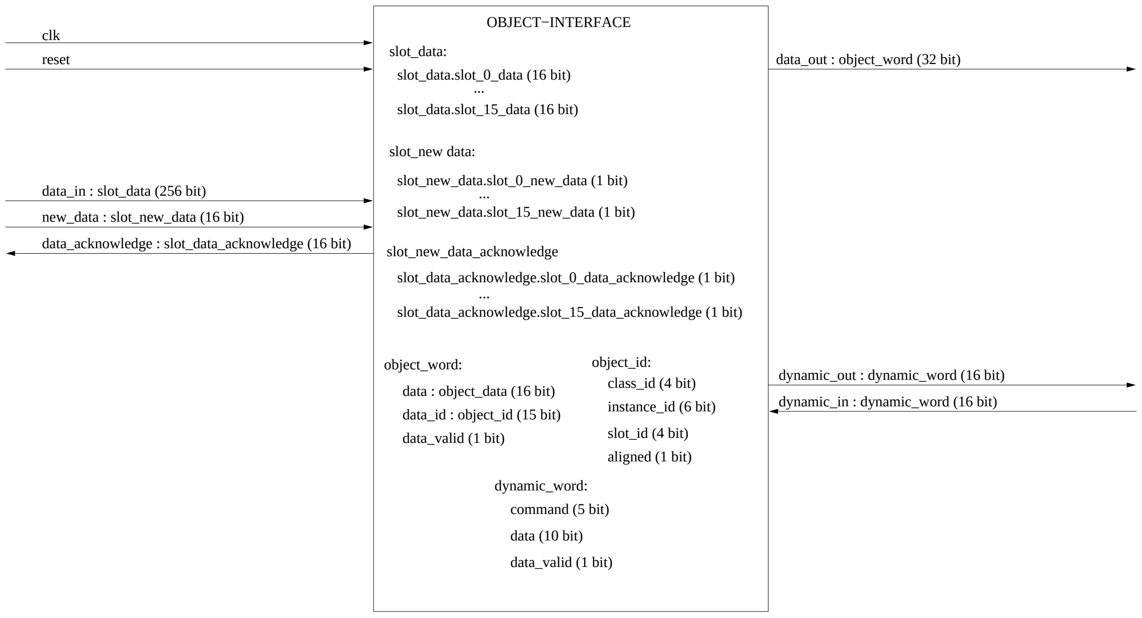
How are the different class buffer and task areas connected? The class buffer of the filters get the input data from both input task areas and the dynamic task area. The remaining class buffer are related to the static task areas. The class buffer of the input task areas only get the data from the periphery of the system, the reconnects, and the class buffer of the output task areas get the data from the dynamic task area. The dynamic task area is the only task area, that has four related class buffer, the four filters.

In principle one can say, the task areas are connected to the class buffer of the classes, that can be placed into the task area. But the class buffer are connected to every task area, that can produce data, because the tasks are freely connectable.

6.3 Implementation of the Task Interface

The developer, using POL, will not have to work on the communication matrix itself. And also the interface for the user defined tasks is generated by POL. Nevertheless it is possible to use the hardware framework and the communication matrix without POL. In this circumstances, the interface of the task area must be at last fulfilled.

In this application 16 inputs are used. Every input contains a data connection, suitable for the used data words, the *new_data* flag and the *acknowledge* flag. The output also provides a data connection suitable for the data words. Finally, the task must implement the DCB connection. The complete interface is displayed in figure 6.4.



OBJECT−INTERFACE
clk
reset
slot_data:
slot_data.slot_0_data (16 bit)
...
slot_data.slot_15_data (16 bit)
data_out : object_word (32 bit)
slot_new data:
slot_new_data.slot_0_new_data (1 bit)
...
slot_new_data.slot_15_new_data (1 bit)
data_in : slot_data (256 bit)
new_data : slot_new_data (16 bit)
data_acknowledge : slot_data_acknowledge (16 bit)
slot_new_data_acknowledge
slot_data_acknowledge.slot_0_data_acknowledge (1 bit)
...
slot_data_acknowledge.slot_15_data_acknowledge (1 bit)
object_word:
data : object_data (16 bit)
data_id : object_id (15 bit)
data_valid (1 bit)
object_id:
class_id (4 bit)
instance_id (6 bit)
slot_id (4 bit)
aligned (1 bit)
dynamic_out : dynamic_word (16 bit)
dynamic_in : dynamic_word (16 bit)
dynamic_word:
command (5 bit)
data (10 bit)
data_valid (1 bit)

Figure 6.4: The task realises the interface specified for the communication matrix. Beside the common clock and reset signals, it provides the (16) inputs, with the additional controll flags and strobes, the (single) output, and the bidirectional DCB interface.

The data word contains an ID, which represents the destination address, and the intrinsic data. For this implementation a 32 bit data word is used, 16 bit are reserved for the ID and 16 bit for the data. The ID of a task is composed of the class ID, the instance ID and the register ID, see table 6.1. The *aligned* bit is used to tell the multiplexer, that the following data must be kept together. This is most important for the re-connect messages. The class ID and the instance ID represents the instantiated application part, the task. The register ID is needed to address the inputs of the task. For this application two inputs are needed. One for the audio data and one to change the connections of the task.

The class ID and the register ID get four bits, so we can handle 16 task with a possible amount of 16 inputs. The instance ID gets six bits, and the application can instantiate up to 64 objects of each task. That is far more, then is used for the example application. The implementation should in principle be able to serve also more common POL applications.

Data	ID				data_valid
	class_id	instance_id	register_id	aligned	
[31:16]	[15:12]	[11:6]	[5:2]	[1]	[0]
xxxxxxxxxxxxxxxx	xxxx	xxxxxx	xxxx	x	1

Table 6.1: The data word used for the communication matrix contains a 16 bit ID for addressing and 16 bit data.

The order of the four task is not set, so the tasks have to change their writing IDs during runtime. In an average POL design we need two messages for a reconnect. One to select the output to be reconnected and one for the new destination ID. In this audio example application we have only one output, so the first message can be neglected and the new destination ID can be send directly.

If you do not want to use POL, to develop an application using the communication matrix, the task must serve the following interfaces. There are two interfaces to the communication matrix. The data interface is used for communication within the application and the dynamic control bus (DCB) interface is used as a direct connection to the communication matrix, especially to the scheduler.

6.4 Implementation of the DCB

For the audio application only one task area is used. To manipulate the right audio data and the left audio data, the different instances have to be scheduled into the task area with short reconfiguration. If even more than one filter should be used at the same time, also long reconfigurations are needed to exchange the different filter tasks.

The tasks have to be prepared, before they can be properly exchanged. This is done with the dynamic control bus (DCB). The DCB is 16 bit wide and is composed of a 5 bit command word, 10 bit data word and the valid bit, see table 6.2.

Command	Data	data_valid
[15:11]	[10:1]	[0]
xxxxx	xxxxxxxxxx	1

Table 6.2: Communication format for the DCB. The DCB is used to prepare the task for scheduling and the dynamic instantiation of new tasks.

The width of the data word of the DCB control bus must be feasible to contain the class ID or the instance ID of a given task. The scheduler and the tasks are using the following protocol to communicate with each other, which is displayed in table 6.3.

If the scheduler wants to exchange a running task, it have to announce the imminent reconfiguration to the task with the *prepare for reconfiguration* command. The task will start to save its inner state and the content data into the BRAM, which is dedicated for the short reconfiguration, if the task reaches a breakpoint in its calculation routine. After the task has saved its content, the task will tell the scheduler, that it is ready for the reconfiguration with the *ready for reconfiguration* command.

Command	Description	Data
	Task to Scheduler	
00000	reserved	
00001	new instance	class_id
00010	ready for reconfiguration	
00011	self delete	
	Scheduler to Task	
00000	reserved	
00001	new instance	instance_id
00010	prepare for reconfiguration	
00011	reserved	
	Debug	
00100	change class ID	class ID
10000	disable task	

Table 6.3: Commands for the communication between the scheduler and the tasks over the DCB in both directions.

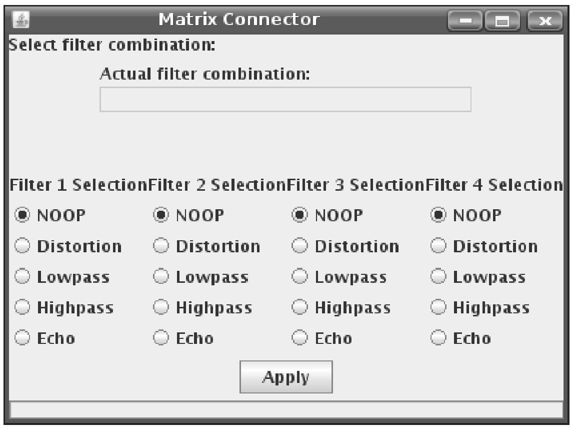
If the task wants to instantiate a new instance of a POL class, it has to write the *new instance* command to the scheduler, with the class ID included into the data section. The scheduler will generate a new instance and will send back the instance ID. The command is also *new instance*, but the data section will contain the new instance ID.

The extinction of any task must be implemented in POL by the developer, because the messages must be send through the communication matrix along with the normal data. This will prevent, that the instance is deleted before all the data inside of the communication matrix buffer are processed by the task. The scheduler provides the self-delete of the tasks with the DCB. Therefore the developer must implement the extinction of other instances in his application. If the tasks sends the *self delete* command to the scheduler, the task area multiplexer is told that no task is present in the task area and the task itself is removed with the next designated reconfiguration. The scheduler will remove the instance ID of the task from his scheduling list.

The DCB is used for the dynamic instantiation of POL classes, the generation of new tasks. It is also used for the self delete of the tasks, where as the delete messages themselves are exchanged along with the common data. But the most important functionality of the DCB is the preparation of the task for the reconfiguration.

6.5 Implementation of a Simple Scheduler

For the example application, the scheduler is realised in software on the PPC, concerning the reconfiguration, and on a external PC for the instantiation of the filters. The user can chose the number of filter and the execution order in a small program running on an external PC. This program transfers the instance IDs to the PPC and the scheduler over the *uart* interface.



Select filter combination:

Actual filter combination:

Filter 1 Selection	Filter 2 Selection	Filter 3 Selection	Filter 4 Selection
[✓ NOOP]	[✓ NOOP]	[✓ NOOP]	[✓ NOOP]
Distortion	Distortion	Distortion	Distortion
Lowpass	Lowpass	Lowpass	Lowpass
Highpass	Highpass	Highpass	Highpass
Echo	Echo	Echo	Echo

Figure 6.5: Screenshot of the control program for the application on the PC.

The actual scheduler is running on the PPC and monitors the status register of the communication matrix. These status register concerns the filling level of the different buffer and the next data address. This enables the scheduler to decide which instance of which object must be configured into the dynamic task area. It also have different control registers, which have a direct access to the communication matrix and especially the static part of the dynamic task area. These control register are used for the different reset procedures needed for the short and long reconfiguration and are also needed to tell the multiplexer of the dynamic task area, which task is configured into the task slot.

7 Results

In this chapter the concrete results for the example implementation of the communication matrix, concerning the resource consumption and the reconfiguration times, are documented. In the first section the reconfiguration times for the short and long reconfigurations are listed and the relevance for the POL application is discussed. Additionally, the theoretically reachable times are also considered. The performance of the reconfigurations directly affects the maximal data frequencies for the application, which the scheduler and the communication matrix can handle without an interruption of the data stream. This is also influenced by the number of tasks, which are processing the data. The second section concerns the additionally needed hardware resources for the communication matrix. The POL hardware framework is only useful, if the communication matrix does not need more resources, than can be saved with the scheduling of the hardware objects. It is reasonable, that the advantage of the POL framework will become apparent, if the used tasks need enough resources. A DPR realisation of a blinking LED design will never be performant in the resource consumption.

7.1 Reconfiguration Performance

The main problem for the reconfiguration performance is the in-optimised way, the ICAP is connected with the DDR ram. To transfer the data from the external memory to the configuration memory of the ICAP two OPB operations are needed. Every data word must be read from the DDR ram to the PPC and after that the data word can be written to the ICAP memory. Using the PPC like this, is the most inefficient way to perform the data transfer, but the needed components are prefabricated and are working instantly. As the optimisation of the reconfiguration process was not the focus of this work, but the communication structure provided by the communication matrix, the high configuration costs are accepted.

In table 7.1 the measured performance times of the different reconfigurations, used with the communication matrix and the POL framework, are displayed. The basic reconfiguration times for the real hardware, which is stored in the CLB frames, and the memory, which is stored in the BRAM content frames, are separated. It is also considered, that the external memory access to the DDR ram is slow and has to be kept in mind.

Reconfiguration Times:		
Reconfiguration Type:	avg. Time [ms]	audio sample
Basic Reconfigurations: Read		
CLB Minor Frame	0.026	2
BRAM Content Minor Frame	0.021	1
BRAM Content Major Frame	1.3	63
Basic Reconfigurations: Write		
CLB Minor Frame	0.031	2
BRAM Content Minor Frame	0.028	2
BRAM Content Major Frame	1.8	87
Reconfiguration Throughput:		
	Throughput [Kbyte/s]	
Reconfiguration of the Logic	932.7	
Reconfiguration of the BRAM content	2018.5	

Table 7.1: The performance times of different reconfiguration types.

The ICAP is connected to the PPC over the OPB. In [Chr08] the calculated and measured reconfiguration performance of the Xilinx ICAP component with an OPB interface is listed. The values concerning this work are listed in table 7.2.

Frequency:	100 MHz
Input Width:	4 byte
Calculated Throughput:	400 Mbyte/s
Measured Throughput:	5.07 Mbyte/s

Table 7.2: Calculated and measured values on the Virtex-4 from [Chr08].

As one can see, the reconfiguration can be five times faster, concerning the measured times of [Chr08] and even 400 times faster, concerning the calculated maximal throughput of the configuration interface. The ICAP is clocked with 100 MHz and the data bus has a width of four bytes. The audio tasks needed 42 frames and have a bitfile size of 177 Kbyte, the BRAM memory content needs 2624 byte.

What does these reconfiguration times, especially the needed times for the short and the long reconfiguration, mean for the usage of the dynamic design and the communication matrix, including the scheduling of the hardware tasks?

The tasks are processing the data words with a clock frequency of 100 MHz and they need only one clock cycle to process a data word. That means, that the tasks can process the data words of a full buffer, with a depth of 512 data words, in 5.12 μs, which is insignificant compared to the sample frequency of the audio data. The tasks can process up to 2083 data words between two samples of the audio data, which has a sample rate of 48 kHz, so we can completely neglect the process time, which the tasks needs for the complete filled buffer, if we consider the maximal reconfiguration times.

Therefore, the maximal performance would be achieved, if the buffer depth is 2048 data words, for this implementation 8192 byte. In this case four BRAMs are completely used, but the task can still process the complete buffer content before the next sample occurs. On the other hand, the reconfigurations can use the fourfold amount of time.

The currently used buffer (2048 byte) fills up in 10.7 ms. And this is also the time the maximal amount of four tasks have to process the data. So, regarding only one audio channel, the reconfiguration of a task can have a maximal duration of 2.7 ms, after that some audio data will get lost. The current system needs 190 ms with the actual ICAP throughput. So every task exchange will be hear able in the audio data stream. Even the measured throughput of [Chr08] will not be sufficient, because it reduces the reconfiguration time only to 35 ms, and that is still not fast enough.

The same arithmetic must be done for the short reconfiguration, which only exchanges the instances, which is in principle equal to the reconfiguration of a single BRAM content, if the echo task is ignored. Using only one filter the short reconfiguration must be performed in 5.35 ms. The content data is a lot smaller, than the configuration data of the tasks, but it also need 1.3 ms. Although the short reconfiguration is awfully slow, it is fast enough to proceed, before audio samples get lost.

In a real application both reconfiguration types occur at the same time. The short reconfiguration is used a lot more often, but it does not take as much time, as the long reconfiguration. Considering all this, it is very clear that the echo filter will definitely not work in this realisation of the reconfiguration and the scheduling. But this is not a principle problem of the introduced framework but to the inefficient implementation of the memory access and also the ICAP access.

But what kind of performance can be achieved using the hardware framework of the communication matrix? For this evaluation the calculated throughput of the ICAP and the optimal buffer size of 8192 byte is more significant.

Using the maximal buffer size (8192 byte), that is still small enough to process the data between two audio samples, the buffer fills up in 42.67 ms. For the task exchange of this audio application we need $177\,\text{Kbyte}/400\,\text{Mbyte}/s = 442.5\,\mu s$, if the maximal throughput can be performed. So using all four filter, the reconfiguration of the filter will have a duration of $4 * 442.5\,\mu s = 1.76\,ms$. That leaves 40.91 ms for the short reconfiguration, representing different audio channels (e.g. the left and right audio data stream). One has to keep in mind, that the short reconfiguration of the echo filter will take more time, due to the additional content data. Altogether the alternating use of the configured tasks will take $3 * 6.56\,\mu s + (3 * 6.56\,\mu s) = 39.36\,\mu s$ for all four tasks. But this would mean, that we can use up to $42.67\,ms/39.36\,\mu s = 1084$ channels for the audio application. This calculation of course neglects the additionally needed input buffer for the different channel.

In general, the communication matrix can perform 42.67 $ms/6.56\,\mu s = 6504$ short reconfigurations before the input buffer is completely filled. That represents also 6504 channels, without any long reconfiguration. The long reconfiguration, on the other hand, can be performed up 42.67 $ms/442.5\,\mu s = 96$ times before the input buffer is full, representing 96 different filters, that are processing the data stream channel.

The needed memory and the required clock frequency are available and realisable with the communication matrix and the used FPGA. These optimisations, regarding the reconfiguration, the scheduling and the memory access, do not affect the principle functionality of the communication matrix and the POL programming approach. The next section will be concerned with the resource overhead of the communication matrix, but it is already clear that the flexibility of the POL designs is reasonable and, most important, can be reached for streaming applications.

7.2 FPGA Resource Allocation

For a developer, using POL to describe an application, the additional overhead of the logic resources needed by the communication matrix and the rest of the parts of the framework are very important. To evaluate the concrete overhead the example application is compared with an equivalent static implementation. The static reference implementation also uses the PPC and an external PC to control the execution order of the different filters, but for the static application, all four filters, for both channels (left and right) are included into the design. In table 7.3 the used resources for the static design are listed.

Static Reference Application		
Resource	Number	percentage
Slices	7532	88
FlipFlops	8976	52
LUTs	11943	69
	208	route-thru
	316	Dual Port RAMs
	327	Shift Register
BRAMs	16	23
Gate Count	1292134	

Table 7.3: The needed resources of a static reference implementation of the example application.

For the static reference design the number of used audio channels (in the implementation two channels are used, the left and the right audio data stream) will directly correspond to the needed resources. Where as the dynamic implementation only have to instantiate a higher number of filter instances for the scheduler. In principle the number of possible channels is not limited, but in a real application the reconfiguration times limit the number of instances, that can process the data, before an overflow of the input buffer occurs.

In table 7.4 the basic resources needed for the different filters are listed. There are strong differences between the resources the individual filter have to allocate. The high pass filter and the low pass filter need a lot of flip flops and LUTs in comparison to the distortion filter, or the echo filter. The echo filter on the other hand needs a lot of BRAMs.

Basic Filter Resources				
Filter:	# Slices	# FlipFlops	# LUTs	# BRAMs
Distortion	53(0%)	74(0%)	69(0%)	1
Highpass	1888(22%)	2321(13)	3195(18%)	1
Lowpass	988(11%)	1222(7%)	1522(8%)	1
Echo	111(1%)	196(1%)	54(0%)	9

Table 7.4: The resources needed by the different filters of the example application.

The distortion filter is the smallest filter and the high pass filter needs the most resources. Therefore, the size of the task area must be adapted to the required resources of the biggest task dedicated for the task area. This will waste a lot of resources, if the smallest task is configured on the FPGA. Nevertheless, it is not necessary to reserve the resources of all tasks needed for the application in the task area.

In picture 7.1 the needed resources for a different number of channels are displayed. Even without the rest of the design, the used board cannot serve more than three channels, if only the needed filter resources are considered. But only the biggest filter must fit into the reconfigurable area and not the complete number of the used filters, if the communication matrix is used.

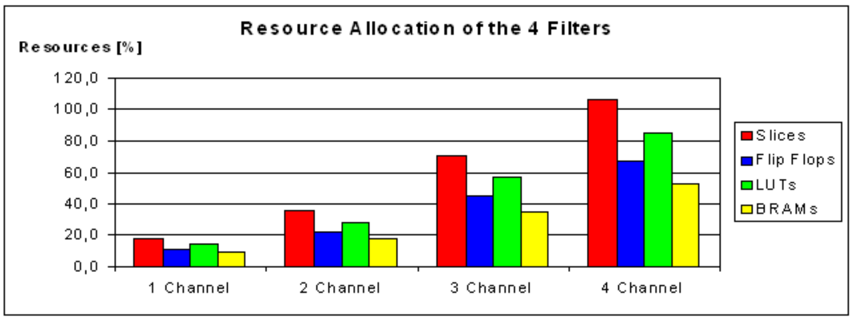


Figure 7.1: The resources needed by the four filters are illustrated for different numbers of channels. In the static realisation, the four filter, using four channels, will allocate more resources, than are available.

But how many resources does the communication matrix need? The needed resources for the dynamic design are listed in table 7.5. Also the dynamic area only contains enough resources for the high pass filter, as the biggest filter, it does need more resources, than the static reference design.

Dynamic Example Application			
Resource:	static part	dynamic part	total
# Slices	6225	1888	8113(95%)
# FlipFlops	7265	2321	9586(56%)
# LUTs	5068	3195	8263(48%)
# BRAMs	27	9	36(53%)

Table 7.5: The resources needed by the example application using the communication matrix. The static and dynamic areas must be accurately separated, which can lead to unused resources. Therefore they are also listed separately.

The scheduler for the example application realised in software using the PPC. And also the different bitfiles and the content memory are stored in the external DDR ram. This is necessary, because these files are far too big for the internal ram. The DDR ram is accessed with the PPC. But the usage of the PPC needs a lot of additional resources.

Plain PPC System		
Resource	Number	percentage
Slices	1739	20
FlipFlops	1586	9
LUTs	1580	9
	69	route-thru
	308	Dual Port RAMs
	190	Shift Register
BRAMs	8	11
Gate Count	582000	

Table 7.6: The resources needed for the PPC. The PPC design only includes the UART and the DDR ram controller and nothing of the reconfiguration stuff.

For the actual application the usage of the PPC is not necessary, if the scheduler and the DDR ram and the ICAP access are implemented in hardware. In table 7.6 the resource allocation of the pure PPC is shown. It is directly apparent, that a lot of resources can be saved, if the scheduler is realised directly in the hardware.

To have a better illustration of the proportion between the resource consumption of the PPC subcomponent, the allocated resources of the communication matrix are shown in

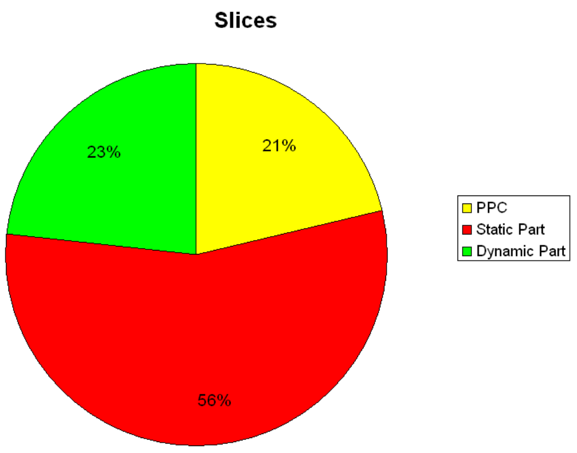


Figure 7.2: The PPC, the static part of the dynamic design and the reconfigurable part use different fractions of the total used slice number.

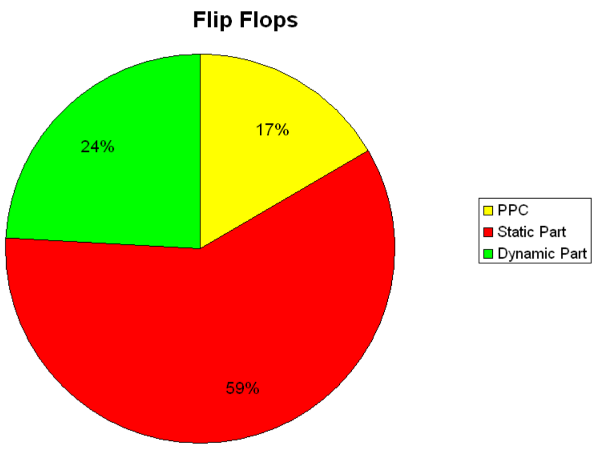


Figure 7.3: The PPC, the static part of the dynamic design and the reconfigurable part use different fractions of the total used flip-flop number.

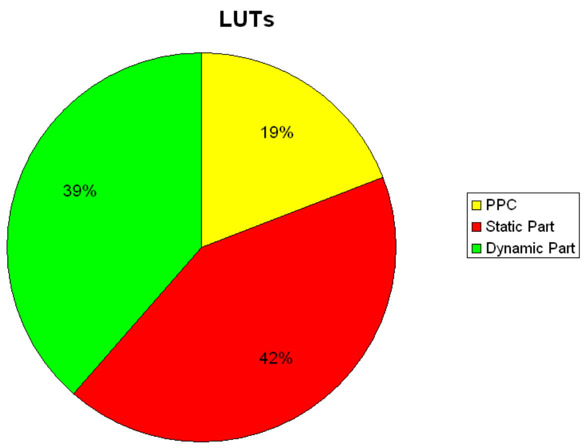


Figure 7.4: The PPC, the static part of the dynamic design and the reconfigurable part use different fractions of the total used LUT number.

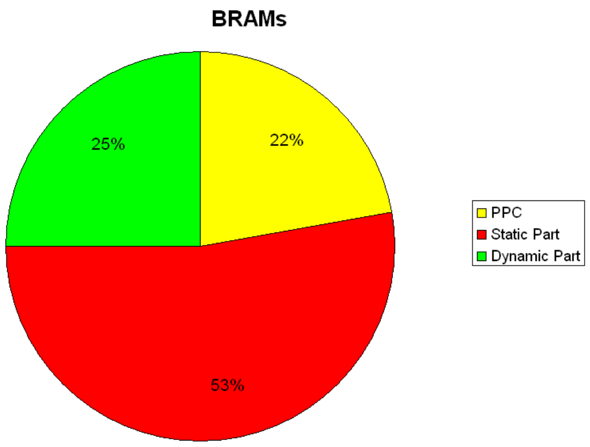


Figure 7.5: The PPC, the static part of the dynamic design and the reconfigurable part use different fractions of the total used BRAM number.

picture 7.2 for the needed slices, in picture 7.3 for the needed flip flops, in picture 7.4 for the needed LUTs and in picture 7.5 for the needed BRAMs.

Although the PPC is realised as a hardware primitive inside of the FPGA, it does need 21% of the slices, occupied by the whole application design, see picture 7.2. The general resource consumption of the PPC is in the region of the dynamic area, which needs 23% of the slices. So, if the PPC can be replaced, we get in principle a second task area for free.

With the needed slices, we only have a rough overview, but do not know which resources the design parts will need in detail. The next three pictures will itemise the required flip-flops, the required LUTs and the required BRAMs.

The PPC does not need much flip-flops in comparison to the remaining part of the static design and the dynamic part, see picture 7.3. The reason is the huge amount of registers needed to provide the communication inside of the communication matrix.

The high pass filter does need a lot of LUTs in contrast to the remaining static design and the PPC, see picture 7.4. This is application specific and is not a general matter of fact.

The PPC in general does not need much BRAM resources, if the program is stored in the DDR ram and not in the internal BRAM memory, see picture 7.5. The used PPC design does not contain the minimal amount of BRAMs, so to save the PPC will not improve the memory consumption of the system. The main memory consumption is caused by the different buffers. The needed number of classes directly scales with the memory consumption. Only the echo with its internal buffer does need a lot of BRAMs, but this is again an application specific matter of fact.

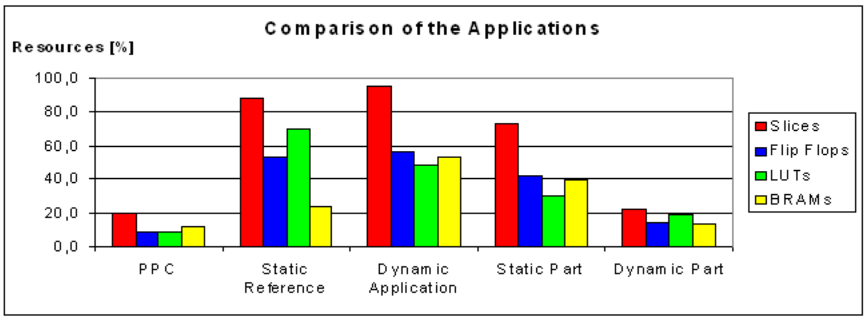


Figure 7.6: The resource consumption of the plain PPC design, the static referenc implementation and the dynamic example design is displayed. The dynamic design is also separately displayed for the static design part and the dynamic design part.

Now it is recommended to compare the static reference design and the dynamic example application directly. To get a concrete idea of the different resource consumption of the static design and the dynamic design, two different aspects have to be considered. In figure 7.6 the resource consumption is separated by the different design parts and in figure 7.7 the design parts are separated by the needed resources.

Although the filter resources and the communication resources cannot be separated for the static reference design, it is directly apparent, that although the dynamic design will need more resources, than the static design, this overhead is not crucial. The main overhead of the used slices is caused to the strict separation of the communication part and the filters. So, not used flip-flops and LUTs of the slices in the static part cannot be used for the dynamic part. If the relation between the resources, needed for the reconfigurable area, can be levelled with the general resource overhead, the dynamic design, with the communication matrix, is equal to the static design, but will contain a lot of more flexibility.

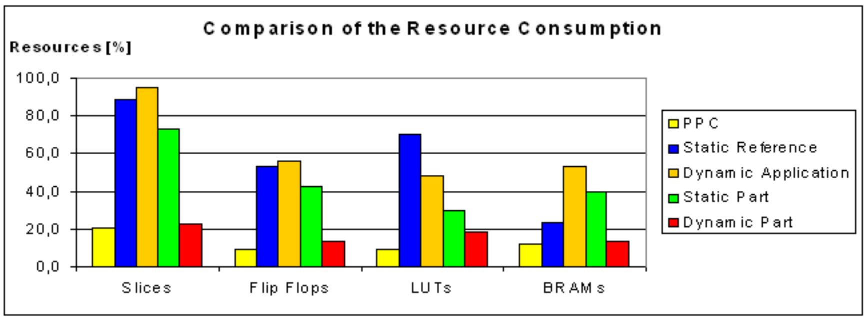


Figure 7.7: The three designs, differed by the resource types, are displayed. The most significant difference between the static and the dynamic design is recognisable in the LUT and BRAM consumption.

The communication matrix buffers the streaming data of the implemented audio application and the communication messages in general several times. Therefore it will have always a high BRAM consumption compared to any static realisation. This is owed to the required flexibility of the DPR application. On the other hand it does not need so much LUTs and the general slice consumption is not much bigger than the specialised static implementation of the application.

The static design is directly limited by the number of used audio channels. The design cannot contain more than two channels, if it shall fit into the used FPGA. Of course it is always possible to use a bigger chip, but this will increase the costs and does not improve the re-usability of the used hardware. The dynamic design can easily be expanded to more audio channels. There is also a limit with the available internal memory, but in principle the number of instances processing the audio data can be extended very freely. The general behaviour of the resource consumption, using different numbers of audio channels, is displayed in picture 7.8.

Although the BRAM inside of the reconfigurable area can not be used for the static part of the design, and this will limit the BRAM occupation to 75%, the dynamic design does

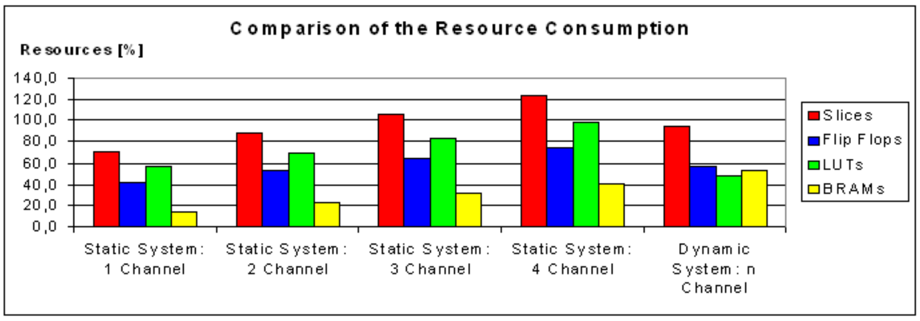


Figure 7.8: Resource consumption of the static and the dynamic design, using different channel numbers. Concerning 7.1, n represents 1084.

not lack memory space for additional audio channels.

The advantage of the communication matrix, despite the high flexibility of the application, can only be significant, if the needed resources of the tasks corresponds to the resource overhead of the communication matrix. That means, the task should not be too big for the general size of the used FPGA, and it should also not be too small, that the scheduling cannot compensate the resource overhead of the communication matrix.

Performance of the Communication Matrix		
Chip	Virtex-4 FX20	Virtex-4 FX20
Performance		
Filters	4	up to 96
Channels	2	up to 6504
Resource Consumption		
Order	O(n)	O(1)

Table 7.7: The presented design, using the developed communication matrix, can beat the resource allocation and the performance of the static reference design.

But in the end the superiority of the dynamic audio design, using the communication matrix and the DPR technology to exchange logic, beats any static audio design, running on the same chip. Concerning the reconfiguration times and the available audio channels, the dynamic design can outperform the static design with a factor of 3252 (table 7.7). It is also possible to use more than 95 different filters in the dynamic design, while the static design fills the whole FPGA with only four filters.

8 Perspective

The communication matrix presented and developed in this work has not been finalised. There are many parts of the concept and the implementation of the communication matrix that can be optimised. This optimisation can improve the performance of an application running with this framework. Also the resource consumption of the communication matrix can be reduced.

The next section will give an overview of the contemplated optimisations, which could not be realised and are left for future work.

8.1 Reconfigurable Buffer

The data bandwidth and the sample frequency of an application is confined to the available memory on the chip. The application data has to be buffered during the reconfiguration time. These buffers are realised as FIFOs with the on-chip block ram memory. A FIFO generated with the IP core generator of the ISE is not reconfigurable because the address pointer and the content data is lost during reconfiguration To increase the performance of the communication matrix and the simultaneous reduction of the needed memory resources, a customised FIFO has to be developed. This FIFO must implement the first word fall through functionality and must keep the address pointer in the same block ram that is used for the data. This would enable the FIFO to be reconfigured with a short reconfiguration, which stores the BRAM content into the DDR ram. The reconfiguration would restore the BRAM content and the FIFO can recover the address pointer with the reset procedure.

With such an customised FIFO the data buffer capacity and the pipelining ability of the communication matrix can increase without using additional resources and with minimal additional reconfiguration overhead.

8.2 Scalable Buffer

The possibility of the communication matrix to buffer the reconfiguration time is limited by the available memory on the chip. Especially the instance buffers are crucial for the memory consumption of a task area because not every input of a task needs the same

FIFO depth. This depth is dictated by the sample frequency of the dedicated input of the task. This gives another reasonable requirement for the customised FIFO developed for the communication matrix. Inputs of a task that are not often sampled can be merged or can be added to the memory space of a highly sampled input. This will require the customised FIFOs to share a single BRAM with a segmented address space.

With such a segmented address space, a multiple scalable buffer for different inputs of a task can be realised. This will reduce the required amount of BRAMs needed for the task. If the segmentation can be done dynamically without reconfiguration the buffer resources can be separated from the task area, as the short reconfiguration is not bound to the task areas representing the reconfigurable areas.

8.3 Multiple Task Areas

The performance of the communication matrix depends apparently on the ratio between the available task areas and the active task of an application. Currently, only a single task area is implemented. For the hardware design of the communication matrix it is not difficult to connect further task areas, but this will increase the compilation time of the design. The tool flow including PlanAhead will take more time to proceed and the short reconfiguration will require a more complex software support to shift the data between different BRAMs.

The advantage of more parallel running tasks in the communication matrix will compensate for the additional complexity. For this work it was decided to level the resource consumption of the DPR application due to the original example application. To prove the buffer concept of the communication matrix the reconfiguration overhead was increased.

For a proper use of the communication matrix including POL this is not the required state. The resource requirements are less than for a classic static design and the reconfiguration overhead is kept to a minimum.

8.4 Faster ICAP Access

In the implementation of the communication matrix done for this work, the ICAP is connected by the PPC's[1] OPB bus with the DDR ram containing the reconfiguration data, This is not a very high-performance design. As the reconfiguration in principle is not very fast, the additional time loss of the data transfer must be minimised. Therefore a customised ICAP access, which is independent from a processor and can use DMA, is

[1]included in the used chip

recommended. This ICAP access is already developed for the Virtex-2pro FPGAs but must be adapted for Virtex-4.

With a faster ICAP access the reconfiguration time is reduced, which will abate the costs of a reconfiguration, and the reconfiguration buffer size can be reduced. This will allow the communication matrix in particular and any DPR design in general to serve an application with higher data frequencies or data widths.

8.5 Intelligent Scheduler

Any consideration for the scheduler done in this work is based on the minimal or maximal amount of buffer size and task areas needed to serve a given application. The question of interest was how many task areas and buffer are needed for different application and data flows? A specific interest was given to the data order. The scheduling of the tasks should not change the data order.

In general the scheduling is based on the filling degree of the different buffer. But the scheduling algorithm itself is simply implemented as round robin. This is not the most efficient algorithm for every application or the communication matrix itself.

As the preservation of the data order during scheduling was high priority, a none efficient scheduling algorithm was implemented. For the performance of the communication matrix such an efficient scheduling algorithm is necessary and recommended and may also depend on the application.

8.6 Automated Communication Matrix Build

The POL developer should not be involved in any low level design step. So it is necessary to generate the hardware design of the communication matrix and the application tasks automatically. The developer will only generate the application using POL and specifies the hardware which will be used, and will not only get a hardware design of the application, but will also get a framework serving the application as well as the specified hardware.

It is not expected that this automated generation of the communication matrix will be hard to implement, but since POL and the communication matrix are still not completely developed, this step will not be very sensible. The used tool flow and the used generation steps of the communication matrix are not stable yet and will change profoundly with only small changes in the functionality. As the exact requirements of a POL application for the communication matrix are not currently well known, an automated generation of the communication matrix and the corresponding DPR design will not work properly.

9 Conclusion

Many different projects and approaches are working on the software-to-hardware compiler problem. The idea of the programming language POL is to transfer a major part of the software development principles to the hardware development. The paradigm of object-oriented programming and the parallel execution, using the thread concept, of the application objects, are adapted to the hardware development. It is not recommended to try to build the hardware out of a completely developed and working software application, but to describe the application in a way that allows the developer to design the hardware application very easily and to use most of the programming paradigm of the software development.

The example application was chosen to demonstrate, that POL hardware applications, using the communication matrix and DPR, cannot only match the performance of any static hardware realisation of the application, but can increase the functionality of the application significant. The static reference system uses the complete FPGA to realise two channels and four filters for the audio data stream. It is not possible to use more audio channels or filters with that design, or the used chip. But the dynamic demonstration implementation of the application can increase the number of audio channels very extremely. Theoretically, up to 6500 channels or 96 filters can be used, without a significant increase of the used resources, due to the hardware scheduling.

So the main advantage of POL projects, compared to any static design, will be located in applications also dedicated for hardware, that are applications using broad data width at high sample frequencies. It is possible for a POL project to design on much smaller FPGAs, than every static design can do, because of the resource sharing of DPR, and the OOP concepts included in POL. If the ICAP access is optimised, it is possible to develop real applications with the framework, introduced in this thesis, that will improve the resource consumption of the design.

Using the programming paradigm POL, it is possible, to describe an application, using all the features and advantages of software development, but execute the application in hardware. Of course, this is only sensible for applications dedicated to run in hardware, and are not software specific, although even these application can be translated into a hardware design, which will be pretty ineffective in comparison with the software execution. This new language POL will improve the development time, as nearly the same environment for the hardware and the software is used and, most important, the re-usability of the hardware design, as the tasks have a predefined interface and can be exchanged between different designs, is improved.

With the programming paradigm, developed in this FPGA workgroup and the according framework, which includes the communication matrix and the DPR techniques described in this thesis, as a part of the high-level programming language POL, it is now possible to describe hardware projects, that are not stand-alone approaches any more, but provide the same re-usability and flexibility, as any software project.

The programming language POL can improve the re-usability and flexibility compared to every static hardware design, as FPGAs improve the re-usability and flexibility of hardware designs compared to ASICs.

Appendix A

Picture of the Used Evaluation Board

Figure A.1: The *Xilinx Virtex-4 Evaluation Platform* board.

Appendix B

The CD

The CD-Rom provided with this thesis contains the following folders with the corresponding contents:

- `/`
  This thesis, as PDF-file and PostScript-file.
- `/latex`
  The Latex sources of this thesis.
- `/ISE_projects/audio_DPR_stat`
  The ISE project of the static version of the dynamic audio application.
- `/ISE_projects/audio_DPR_dyn`
  The ISE project of the dynamic audio application with separated design parts.
- `/ISE_projects/audio_reference`
  The ISE project of the static reference design.
- `/ISE_projects/plain_PPC`
  The ISE project of the plain PPC design.
- `/PlanAhead/PlanAhead_Files`
  The NGC files generated with the ISE for the PlanAhead project.
- `/PlanAhead/audio_DPR_pa`
  The PlanAhead project of the audio application.
- `/workspace`
  The Eclipse project for the Communication Matrix controll programm on the external PC.
- `/download`
  The files needed to run the application on the evaluation board.

Bibliography

[Abe05] ABEL, N., **2005**. *Schnelle dynamische partielle Rekonfiguration in Hardware mit Inter-Task-Kommunikation*. Diploma thesis, Universität Leibzig.

[AK06] ALFONS KEMPER, A. E., **2006**. *Datenbanksysteme*. Oldenbourg Wissenschaftsverlag.

[Bob07] BOBDA, C., **2007**. *Introduction to Reconfigurable Computing*. Springer Verlag.

[Chr08] CHRISTOPHER CLAUS, B. ZHANG, W. STECHELE, L. BRAUN, M. HÜBNER, J. BECKER, **2008**. *A Multi-Platform Controller Allowing for Maximum Dynamic Partial Reconfiguration Throughput*. *FPL 2008*.

[Con] *Configurable Hardware*. URL `http://www.kip.uni-heidelberg.de/ti/FPGA/cms/website.php`. The *Configurable Hardware* group at Kirchhoff Institute for Physics, Heidelberg.

[Dir08] DIRK KOCH, CHRISTIAN BECKHOFF, JÜRGEN TEICH, **2008**. *ReCoBus-Builder - A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs*. *FPL 2008*.

[End08] ENDRULLIS, S., **2008**. *Implementierung von Java-Threads in Software und rekonfigurierbarer Hardware*. Diploma thesis, Universität Leibzig.

[Gra08] GRAHAM SCHELLE AND DIRK GRUNWALD, **2008**. *Exploring FPGA Network on Chip Implementations Across Various Application and Network Loads*. *FPL 2008*.

[Gru09] GRUELL, F., **2009**. *Development of a Software-to-Hardware Compiler based on JAVA and VHDL (preliminary)*. Diploma thesis, Kirchhoff Institute for Physics. Publication planned.

[JAV] *Developers Resources for Java Technology*. URL `http://java.sun.com/`.

[JHD] *Just another Hardware Description Language*. URL `http://www.JHDL.org`.

[Nor08] NORBERT ABEL, FREDERIK GRÜLL, NICK MEIER, ANDREAS BEYER, UDO KEBSCHULL, **2008**. *Parallel Hardware Objects for Dynamically Partial Reconfiguration*. *FPL 2008*.

[Pet98] PETER BELLOWS AND BRAD HUTCHINGS, **1998**. *JHDL - An HDL for Reconfigurable Systems*. *Field-Programmable Custom Computing Machines*.

[Rad06] Radu Marculescu, J. Rabaey and A. Sangiovanni-Vincentelli, **2006**. *Is "Network" the Next "Big Idea" in Design? DATE 2006.*

[ReC] *The ReCoBus Project*. URL `http://www.recobus.de`.

[Thi06] Thilo Pionteck, Carsten Albrecht, Roman Koch, **2006**. *A Dynamically Reconfigurable Packet-Switched Network-on-Chip. DATE 2006.*

[Xil08a] Xilinx, **2008**. *ML405 Evaluation Platform User Guide*. Version 1.5.1.

[Xil08b] Xilinx, **2008**. *Virtex-4 Configuration User Guide*. Version 1.9.

[Xil08c] Xilinx, **2008**. *Virtex-4 User Guide*. Version 2.3.

Erklärung:

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 16. Oktober 2008

Johannes Nick Meier