I N A U G U R A L - D I S S E R T A T I O N

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht - Karls - Universität
Heidelberg

vorgelegt von
Diplominformatiker Norbert Abel
Tag der mündlichen Prüfung: 22.2.2011

T h e m a

Design and Implementation of an Object-Oriented Framework
for Dynamic Partial Reconfiguration

Gutachter: Prof. Dr. Udo Kebschull
Prof. Dr. Reinhard Männer

Abstract

Nowadays, two innovative future trends regarding embedded hardware development and hardware description can be found. The first trend concerns the hardware itself. Modern FPGAs (Field Programmable Gate Arrays) provide the possibility that parts of the hardware can be exchanged while the rest of the circuit is running untouched – which is called dynamic partial reconfiguration (DPR).

The second trend concerns the way hardware is described. Currently, the most important hardware description languages (HDLs) are VHDL and Verilog. Although they allow to describe hardware on a very high level, the developer still has to handle registers, clocks and clock domains. Using an HDL operating on the algorithmic level, this is not necessary any longer. Here, designs can be described exactly as they are in software languages like C, without the need to care about registers or clocks – which is called high level synthesis (HLS).

Although both, DPR and HLS are very important future trends regarding hardware design, they develop rather independently. Most of today's software-to-hardware compilers focus on conventional hardware and therefore have to remove dynamic aspects, such as the instantiation of calculating modules at runtime. On the other hand, DPR tools work on the lowest possible layer regarding FPGAs: the bitfile level. Currently, the use of DPR leads to a struggle with architectural details of the FPGAs and the corresponding synthesis and implementation tools. A hardware developer who makes use of DPR would focus most of the time on DPR and only a small part of the time on the implementation of the actual functionality — which is obviously the opposite of what hardware engineers want.

This thesis focuses on a combination of DPR and HLS, since this has the potential to kill two birds with one stone. Firstly, DPR can change the programming paradigm in future HDLs with regard to dynamic instantiations. Dynamic parts would not have to be removed any longer, but could be realized on the target FPGA using DPR. Secondly, a high level language support of DPR technologies could help to end its shadowy existence and to become a common used method. Hence, the aim of this study is to find a solution how HDLs on algorithmic level and DPR can be combined, solely using language constructs which are already well-known to software-developers. As a first step regarding the development of this framework, the typical structure and behavior of reconfigurable hardware has been analyzed. Thereby it turned out that the best way to describe such hardware is to make use of the object-oriented paradigm combined with multi-threading. In consequence, an enriched subset of Java, forcing the programmer to make use of multiple objects running in parallel, has been defined: POL (Parallel Object Language).

The specification of POL comes with a set of requirements. The most challenging part is the high degree of flexibility regarding object instantiation and inter-object communication. POL allows the user to instantiate and to destroy objects as well as to establish

and to dissolve their connection at *any* position in the code. Beyond that, POL allows an overmapping of the FPGA, which is realized via DPR.

In order to enable the evaluation of the possibilities and limitations of POL, a development framework has been implemented. This framework includes an emulator which allows the execution of POL in software, a compiler which is responsible for the translation from POL to VHDL, a so called Communication Matrix which serves as fast and flexible communication structure on the FPGA, and a scheduler that decides which hardware module is loaded when. Two example applications have been implemented: Pong and an audio filter.

The Pong example shows, that all parts of the framework are working correctly and thus that it is really possible to describe DPR in an HDL on algorithmic level. Furthermore it proves that it is possible to overmap the FPGA via DPR.

The audio example is used to analyze the behavior of the framework regarding data streams. It shows that overmapping can be used in environments with a data rate of $\sim$100 000 samples/s, while scenario-based scheduling algorithms can be used in streaming applications with data rates of $\sim$100 000 000 samples/s. These maximum data rates are solely possible due to the usage of object-orientation in POL and the corresponding optimizations of the reconfiguration times.

A further important result is that the framework helps to significantly increase the productivity of hardware developers who want to make use of DPR. Based on these promising results, first cooperations focusing the combination of DPR an HLS in a commercial product could already be initiated.

Kurzbeschreibung

Der aktuelle Stand der Mikrochipentwicklung offenbart zwei wichtige Zukunftstrends. Der erste betrifft die Mikrochips selbst. Moderne FPGAs (Field Programmable Gate Arrays) erlauben den dynamischen Austausch von Teilen ihrer Schaltung, während der Rest des Chips ungestört weiterläuft. Diese Technologie nennt sich dynamische partielle Rekonfiguration (DPR).

Der zweite Trend betrifft die Art wie Hardware beschrieben wird. Zur Zeit sind VHDL und Verilog die beiden wichtigsten Hardwarebeschreibungssprachen. Allerdings agieren beide auf dem sogenannten Register-Transfer-Level, was die Verwendung von Registern, Taktsignalen und Taktungszonen (Clock-Domains) nötig macht. In einer Sprache die auf dem algorithmischen Level agiert ist dies nicht mehr notwendig. Hier gleichen die Hardwarebeschreibungssprachen üblichen Programmiersprachen wie C oder Java. Eine Beschreibung von Takten oder Taktzyklen entfällt. Die Generierung von Hardware aus einer Sprache auf algorithmischer Ebene nennt sich High-Level-Synthese (HLS).

Obwohl DPR und HLS vielbeachtete Zukunftstrends darstellen, entwickeln sich beide nahezu unabhängig voneinander. Die meisten der heutigen HLS-Tools konzentrieren sich auf konventionelle statische Hardware und müssen daher dynamische Sprachelemente, wie das Instantiieren von Objekten zur Laufzeit, durch andere Verfahren ersetzen oder sie sogar ganz verbieten. Auf der anderen Seite arbeiten die meisten heutigen DPR-Tools auf der für FPGAs niedrigsten Ebene: dem Bit-Level. Dadurch müssen sich derzeit Hardwareentwickler, die DPR einsetzen wollen, mit Details der FPGA-Architektur und Details der entsprechenden Synthese-Tools auseinandersetzen und sind dadurch gezwungen den Hauptteil der Entwicklungszeit in DPR statts in ihre eigentliche Arbeit zu investieren — was natürlich das Gegenteil dessen ist, was Hardwareentwickler wollen.

Diese Doktorarbeit hat sich zum Ziel gesetzt, DPR und HLS zu kombinieren, da dies das Potential birgt, zwei Probleme auf einmal zu lösen. Zum einen kann DPR die Programmierparadigmen zukünftiger Hardwarebeschreibungssprachen bezüglich der dynamischen Instantiierung von Objekten wesentlich verändern: Dynamik müsste nicht mehr ersetzt oder entfernt werden, sondern könnte mittels DPR direkt auf dem FPGA umgesetzt werden. Zum anderen vereinfacht die Unterstützung von DPR in einer Hochsprache die Verwendung dynamische Elemente und könnte dazu beisteuern, dass DPR zu einer allgemein anerkannten und genutzten Technologie wird.

Dabei sollen ausschließlich Hochsprachenkonstrukte, welche bereits aus der Softwareentwicklung bekannt sind, zum Einsatz kommen. Zuerst wurden hierzu die typische Struktur und die typischen Einsatzgebiete rekonfigurierbarer Hardware untersucht. Ergebnis dieser Untersuchung war, dass sich vor allem die Objekt-Orientierung in Kombination mit Multithreading eignet um Hardware auf algorithmischer Ebene zu beschreiben. Aus diesem Grund wurde eine Sprache namens POL (Parallel Object Language) entwickelt, welche ein leicht erweitertes Subset von Java darstellt und die Benutzung von parallel laufenden Objekten forciert.

Die Spezifikation von POL stellt hohe Anforderungen an die zu generierende Hard-

ware. Der herausfordernste Part ist die große Flexibilität bezüglich der Instantiierung, Zerstörung und Verbindung von Objekten. POL erlaubt selbiges an *jeder beliebigen* Stelle des Codes. Darüber hinaus erlaubt POL ein sogenanntes Overmapping des FPGA, das sich nur durch den Einsatz von DPR realisieren lässt.

Um die Fähigkeiten und Grenzen von POL testen zu können, wurde eine komplette Entwicklungsumgebung implementiert. Diese enthält einen Emulator der es erlaubt POL in Software auszuführen, einen Compiler der POL nach VHDL übersetzt, die sogenannte Kommunikations-Matrix welche eine schnelle und flexible Kommunikationsstruktur auf dem FPGA zur Verfügung stellt und einen Scheduler der entscheidet, welches dynamische Modul zu welchem Zeitpunkt auf den FPGA geladen wird. Zwei Beispielanwendungen wurden mit Hilfe der Entwicklungsumgebung entwickelt und übersetzt: Pong und ein Audio-Filter.

Anhand des Pong-Beispiels kann gezeigt werden, dass alle zuvor genannten Teile der Entwicklungsumgebung korrekt funktionieren, es also tatsächlich möglich ist, DPR in einer Hochsprache zu beschreiben. Weiterhin zeigt es, dass ein Overmapping möglich ist.

Das Audio-Beispiel zeigt, dass Overmapping in einer Umgebung mit einer Datenrate von maximal $\sim$100 000 Samples/s verwendet werden kann. Ein szenario-basiertes Scheduling hingegen lässt sich sogar in Umgebungen mit einer Datenrate von maximal $\sim$100 000 000 Samples/s einsetzen. Diese Maximalwerte konnten nur aufgrund der Verwendung von Objekt-Orientierung in POL und den dadurch ermöglichten Optimierungen in der darunterliegenden Hardware erreicht werden.

Ein weiteres wichtiges Ergebnis der durchgeführten Tests ist, dass die Entwicklungsumgebung hilft, die Produktivität von Hardwareentwicklern die DPR einsetzen wollen, signifikant zu erhöhen. Aufgrund dieser vielversprechenden Ergebnisse konnten bereits erste Kooperationen, welche zum Ziel haben DPR und HLS in einem kommerziellen Produkt zu vereinigen, initiiert werden.

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die diese Arbeit möglich gemacht haben. Zu allererst bei meiner Familie die mich immer treu unterstützt hat und ohne die diese Arbeit nicht möglich gewesen wäre.

Mein besonderer Dank gilt Prof. Dr. Udo Kebschull für die Möglichkeit, diese Arbeit zu schreiben und seine intensive Unterstützung dabei. Für das Probelesen dieser Arbeit bedanke ich mich bei: Annemarie Hallier, Matthias Rosker, Frederik Grüll, Sebastian Manz, Heiko Engel, Jano Gebelein, Matthias Kretz, Hannah Meierhofer, Wade Johnson, Stefan Seidel und Danny Gehl.

Ein großer Dank gilt allen Studenten, die an der Implementierung des Frameworks beteiligt waren: Frederik Grüll, Johannes Nick Meier und Andreas Beyer. Ohne die vielen produktiven Diskussionsrunden sowie ihre selbständige und zuverlässige Arbeit wäre die Implementierung des Frameworks nicht möglich gewesen.

Für Jenny, Niels und Luise.
Bis zum Mond und zurück.

Contents

List of Important Acronyms

ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CPU	Central Processing Unit
CPLD	Complex Programmable Logic Device
DCB	Dynamic Control Bus
DDR	Double Data Rate
DPR	Dynamic Partial Reconfiguration
FF	Flip-Flop
FIFO	First In, First Out
FSM	Finite-State Machine
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
ISE	Integrated Software Environment
LUT	Lookup Table
NoC	Network-on-Chip
OOP	Object-Oriented Programming
PLA	Programmable Logic Array
POL	Parallel Object Language
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SRAM	Static RAM
TCL	Tool Command Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XST	Xilinx Synthesis Tool

1 Introduction and Motivation

1.1 Hardware Evolution

Nowadays, mankind is surrounded by a tremendous number of micro chips that make our daily life easier and more convenient. Integrated circuits (ICs) serve as integral part of advanced driver assistance systems increasing the car safety with technologies like the air bag, the anti-lock braking system, or the traction control system. They enable the development of smart but small mobile phones containing a camera, an organizer, a radio, and even a video player. Even a simple device such as a toaster is equipped with an IC. So, without exaggeration one can say that the lifestyle of a citizen of the 21st century highly depends on the availability of these small ICs.

In 1941, it was Konrad Zuse who built the first working programmable binary computing machine based on a large number of electromechanical relays. His Z3 was the first Turing complete computing machine [1]. However, solely relay based computers only had a short appearance in the history of computing since already in 1946 ENIAC (Electronic Numerical Integrator And Computer) was constructed. It was built of 17 468 vacuum tubes, 7 200 crystal diodes, 1 500 relays, 70 000 resistors and 10 000 capacitors. ENIAC heralded the period of vacuum tube based computers [2].

Already in 1947 the transistor was discovered at the Bell Laboratories; but it took 8 years until it became possible to use it for computer technology. In 1955 Bell Labs developed the TRansistorized Airborne DIgital Computer (TRADIC) the first computer, which was based exclusively on transistors replacing the vacuum tubes [3]. Transistor technology is the prerequisite to build small integrated circuits which are exclusively based on electronic components.

The first IC (a flip-flop) was build in 1958 by Jack Kilby. It was based on germanium. Only half a year later Robert Noyce built an IC that was made of silicon (like all ICs today). The first integrated circuits contained only few transistors (from 2 to 100). Thus they are called Small-Scale Integration (SSI) circuits. But the size of integrated transistors decreased continuously and hence the packing density increased. In the late 1960s integrated circuits could contain up to 1 000 transistors, called Medium-Scale Integration (MSI). Further development led to Large-Scale Integration (LSI) in the mid 1970s, with up to 100 000 transistors per chip. Based on this, it became possible to manufacture an entire processor as a single integrated chip. At the beginning of the 1980s one IC could already contain up to 1 000 000 transistors, which is called very-large-scale integration (VLSI). This enabled the manufacturing of Random Access Memory (RAM) with a size of 1 MB. In 1989 micro processor chips passed the million transistor mark. In 2005 ICs

could contain more than one billion transistors. In 2007 more than 10 billion transistors were integrated.

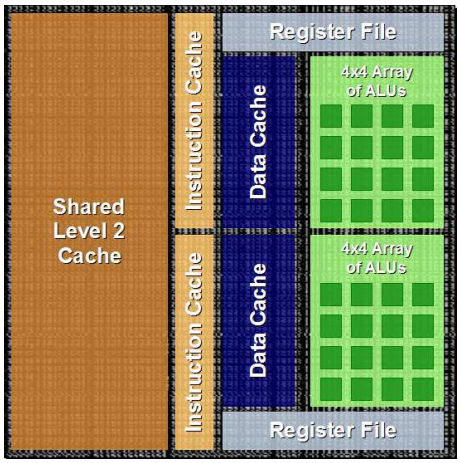


Figure 1.1: Dual Core Processor Floorplan

1.2 Productivity Gap

The increasing number of transistors per IC did not only come with advantages, but also led to a completely new problem, the so called productivity gap. Each year the complexity of embedded systems increases by about 60%, but the corresponding productivity[1] only increases by about 20%. As a consequence, hardware developers all over the world are searching for methods to describe hardware more efficiently. The aim is to decrease the designing effort and thus to decrease the time-to-market. [4]

In 1960 the optimal arrangement (that means the placement of the transistors and their interconnections) could be found very easily. Due to this, until 1960 it was common to do the placement manually. In contrast, it is no longer possible to determine the optimal placement for a chip containing millions of transistors by hand. Thus, computers are used to calculate the optimal placement with a process called Electronic Design Automation (EDA). However, searching for the optimal placement is a problem covered in graph theory and has been proven to be NP-hard. Consequently, even a supercomputer is unable to calculate the best transistor arrangement for complex ICs in an acceptable timeframe. A resulting common method to reduce the complexity is the hierarchical floorplanning. Instead of optimizing the complete chip, the IC is separated into a number of areas which are placeholders for several functional blocks (e.g. a dual core processor IC could be divided into shared level-2 cache, two register files, two instruction caches, two level-1 data caches and many ALUs (arithmetic logic units) — see figure 1.1). These functional

[1] Productivity is commonly understood as the ratio of produced outputs to consumed resources

blocks are then optimized individually (or even separated further into sub-blocks) and have well defined interfaces or buses to communicate with each other.

Another common design simplification method based on hierarchical floorplanning is the reuse of logic blocks. For example, a new quad core based on the dual core in figure 1.1 could make use of the same ALUs and register files as the dual core. They would not have to be redesigned at all. Of course, such a reuse strategy decreases the development time and the time-to-market significantly. Furthermore, all the verification and testing of the reused components already has been done. This considerably increases the reliability. Hence, today several vendors focus on the creation of so called IP-Cores[2]. These are small standard logic blocks which usually are intensely tested and intended to be used in various ICs. The usage of these standard blocks is a further method to significantly increase the productivity.

1.3 Programmable Hardware

The idea of using standard blocks has reached its peak with programmable hardware. There, standard logic cells are already placed but their interconnections are programmed subsequently to determine the functionality of the chip. First versions of programmable hardware were so called PLAs (Programmable Logic Arrays). PLAs consist of programmable AND gate planes, which link to a set of programmable OR gate planes, which in turn produce the output (see figure 1.2). The first PLAs could only be programmed once.

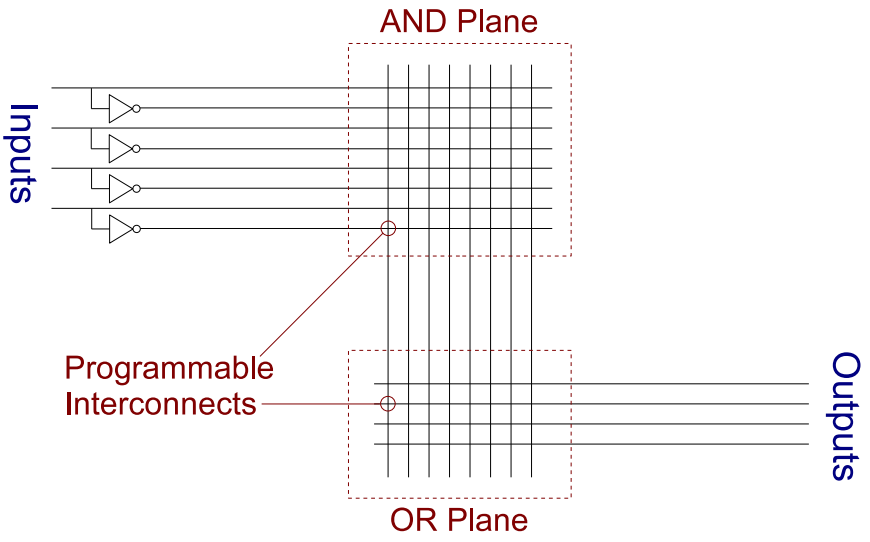


Figure 1.2: Programmable Logic Arrays

[2]Intellectual Property Core

PLAs were followed by so-called FPLAs (Field-Programmable Logic Arrays). FPLAs can, in contrast to simple PLAs, be reconfigured again and again (after being manufactured or "in the field"). Simple PLAs can be used to represent any logic function, but since they do not contain storage elements, they cannot be used to represent sequential logic. Therefore CPLDs (Complex Programmable Logic Devices) attracted attention. CPLDs are based on disjunctive normal forms too but they additionally contain flip-flops. Thus, CPLDs have the potential to be used to implement any conceivable functionality. However, CPLDs are relatively slow and do not use the underlying hardware in an efficient way (many connection wires stay unused). In 1985, the first commercially viable FPGA (Field Programmable Gate Array) was invented by Xilinx. The most important difference to standard CPLDs is that FPGAs make use of SRAM (Static Random Access Memory) based LUTs (lookup tables) instead of logic arrays. This enables a much more efficient usage of the available logic resources. Since SRAM is much faster than Flash memory, this increases the speed of the device significantly. Nowadays, the border between FPGAs and CPLDs became a gray area, since there are LUT based CPLDs [5] as well as Flash based FPGAs [6]. However, a further decisive differentiating factor is that FPGAs usually make use of additional components such as integrated memory, multipliers or even processors to enrich their capabilities.

1.4 Hardware Description Languages

The productivity gap led to completely new ways of describing hardware. At the beginning of the semiconductor technology, integrated circuit design was done manually with pen and paper. Later on, computer aided design emerged, but still the transistors and wires had to be instantiated manually. This changed in the early 1980s, when the use of Hardware Description Languages (HDL) became common. One of the first HDLs was ABEL (Advanced Boolean Expression Language), using boolean equations and truth tables to describe the hardware. Also in the 1980s more sophisticated HDLs, describing the hardware on register transfer level were developed. The two most important examples are VHDL[3] and Verilog. Both languages make it possible to describe hardware with constructs that are comparable to software programming constructs (such as the possibility to use sequential processes, if-clauses, and loops). The use of these HDLs was largely motivated by the possibility to do quick changes and by a high potential of reuse. If HDLs are used correctly, a change from a subcomponent using an 8 bit wide data bus to a subcomponent implementing the same functionality with a 16 bit wide data bus, can simply be done with a change of one single parameter. In contrast, if hand wired subcomponents were used, the whole design would have to be redrawn. HDL written designs are thus much easier to adapt and therefore much more reusable than hand wired circuits. Furthermore, the design of VHDL and Verilog easily allows simulations. This significantly simplifies the design verification and thus the reliability.

[3] **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

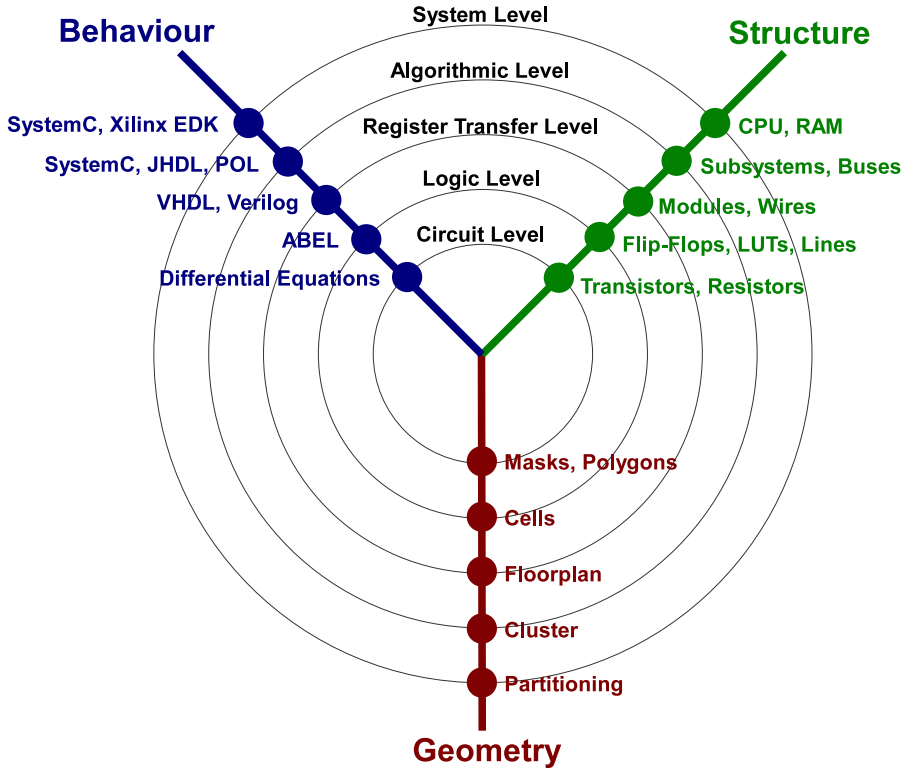


Figure 1.3: Gajski-Walker-Diagram [7]

1.5 Future Trends

The Gajski-Walker-Diagram (figure 1.3) illustrates the views and the layers on which hardware circuits can be described. The lowest layer is the electrical / analog level. Here the circuit's behavior is not described with logical states (one or zero) but using analog signal levels. The second layer is the logic level where digital components are used to generate a logic function. VHDL and Verilog reside on layer 3, which is the register transfer level (RTL). At this level, a circuit's behavior is defined as a flow of data between hardware registers, and the logical operations performed on this data. Although HDLs already allow to describe hardware on a very high level, the developer still has to handle clocks and clock domains. On the algorithmic level this is not necessary any longer. Designs can be described exactly as in software languages like C without the need to care about clocks. The corresponding synthesis process is called high level synthesis (HLS). Many hardware developer groups are looking forward to use HLS, since it has a great potential to increase the programming productivity and therefore to close the productivity

gap. Thus, there are a lot of research projects dealing with HLS — but the vital breakthrough is still pending[4]. Nevertheless it seems clear that HLS is the future of hardware development.

When looking at programmable hardware, a second future trend arises. Newer Xilinx FPGAs (like the Virtex series) provide the possibility to be reconfigured partially and dynamically [8]. Partial reconfiguration means that parts of the hardware can be exchanged while the rest of the circuit continues to run untouched. Dynamic reconfiguration means that the reconfiguration process does not cause glitches on the reconfigured circuits, as long as one does not change the configuration (identical overwrite). The combination of both, called DPR (dynamic partial reconfiguration), opens a large field of new functionalities with FPGAs.

Examples of applications which can be improved with DPR are video processing [9], automotive[10] or packet filtering [11]. Other technologies like dynamically reconfigurable processors [12], scrubbing [13] or dynamically loadable hardware modules [14] are not even possible without DPR. The basic idea behind all these applications is to use the reconfiguration not only for maintenance, but also to utilize it during normal operation. With DPR, the chip can customize itself to the current requirements. For example a dynamically reconfigurable processor is able to adapt its instruction set to the software running on it. This way hardware/software codesign becomes much more flexible and thus much more powerful.

1.6 Goals of this study

Although both, DPR and HLS are very important future trends regarding hardware design, they develop rather independently. Today's software-to-hardware compilers focus on conventional hardware and therefore have to remove dynamic aspects, such as the instantiation of calculating modules at runtime [15, 16]. On the other hand, DPR tools work on the lowest possible layer regarding FPGAs: the bitfile level. Currently, the use of DPR leads to a struggle with architectural details of the FPGAs and the corresponding synthesis and implementation tools. A hardware developer who makes use of DPR would focus most of the time on DPR and only a small part of the time on the implementation of the actual functionality — which is obviously the opposite of what hardware engineers want.

This study focuses on a possible combination of DPR and HLS, since this has the potential to kill two birds with one stone. Firstly, DPR can change the programming paradigm in future HDLs with regard to dynamic instantiations. Dynamic parts would not have to be removed any longer, but could be realized on the target FPGA using DPR. Secondly, a high level language support of DPR technologies could help to end its shadowy existence and to become a common used method. Hence, the aims of this study are to find a

[4]Several approaches are presented in chapter 3

solution how HDLs on algorithmic level and DPR can be combined — and to implement a Framework[5] which provides the necessary functionality. Resulting questions are:

1. Which high level language should be used to support DPR?
2. Which underlying hardware infrastructure is needed?
3. Can DPR be used to increase the capacity utilization of hardware created via HLS?
4. How does the combination of DPR and HLS influence the performance?

This thesis is organized as follows: Chapter 2 presents the basic principles of underlying technologies. Chapter 3 takes a look at former developments — especially regarding DPR and HLS. Chapter 4 refines the approach based on the current state of the art and denotes the concrete aims of this thesis. Furthermore, it will answer question 1. Chapter 5 analyzes the demands which come with the idea of merging DPR and HLS and will answer question 2. Chapter 6 presents the resulting design of the Framework. Chapter 7 illustrates how the Framework was implemented. Finally, chapter 8 presents measurements and analyses of the Framework and answers the questions 3 and 4. Chapter 9 concludes this study.

[5] In the following chapters, the term "the Framework" represents the tools and the chip design that have been developed and implemented during this thesis. It is used as shortform of "Object-Oriented Framework for Dynamic Partial Reconfiguration".

2 Basic Principles

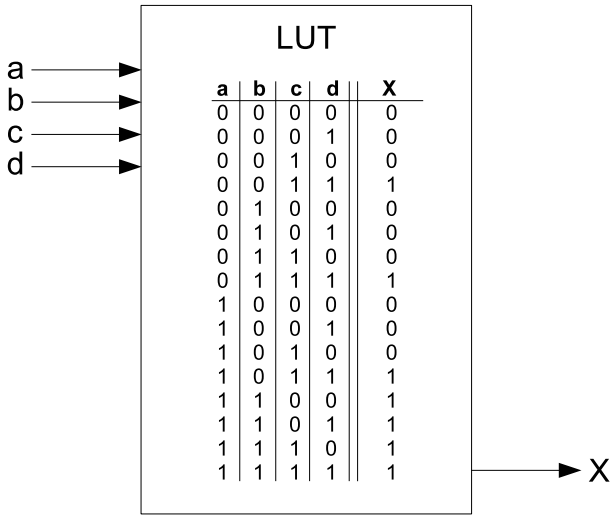
This chapter describes the basic principles which underlie the requirements analysis as well as the design and the implementation of the DPR[1] Framework. The first six sections focus on Xilinx FPGAs, how DPR is realized and which tools support DPR. After that, software development methodologies (namely object-oriented programming, multithreading and the Qt framework) are described. Finally, the translation from software to hardware is illustrated.

2.1 Field Programmable Gate Arrays

Since the implementation of the Framework was done using Xilinx FPGAs as target architecture, these chips and their layout shall be illuminated in more detail. Xilinx FPGAs are based on look up tables (LUTs). Until Virtex-5 and Spartan-3 these LUTs had 4 inputs and 1 output. For Virtex-6 and Spartan-6 they have 6 inputs and one output. The dependency between output and input is freely programmable. Figure 2.1 illustrates an example implementation of such a LUT. In contrast to CPLDs, Xilinx FPGAs are based on SRAM[2]. That means, every LUT is an array of SRAM cells. Each cell stores a value of zero or one. The input signals are used as address signals to this small memory. This way the configuration of the memory cells determines the logic function represented by the LUT. Furthermore, the SRAM cell arrays can be used directly as small memory cells. Due to this they do not only have the 4 or 6 address input wires, but also data input wires, which enable memory access during normal operation. Additionally the SRAM arrays can be configured to represent shift registers. The operation mode of an memory array as LUT, RAM or as shift register can be determined via additional configuration bits.

Other very important basic modules of FPGAs are the flip-flops (FFs). These little memory units enable the change from simple combinatorial circuits to sequential logic, which contains inner states. This is the basic prerequisite to implement finite state machines (FSMs). FFs can be configured to be synchronous (clocked) or asynchronous (in this case they are called Latches). In FPGAs, LUTs and FFs are packed together in so called Slices. One Slice on a Virtex-4 consists of two LUTs, two FFs and several routes between them. The routes are controlled by configurable multiplexers which also belong to the Slice. Figure 2.2 shows the layout of a Virtex-4 Slice. The upper LUT can only be connected to the upper FF and the lower LUT can only be connected to the lower FF. The multiplexers can be configured in a way, that in a data flow LUT, FF, or both are used.

[1] Dynamic Partial Reconfiguration
[2] Static Random Access Memory



a	b	c	d	X
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure 2.1: Example LUT with 4 inputs: X <= (a and b) or (c and d)

Auxiliary wires like “cin” or “cout” are particular short wires between the Slices, which for example enable the efficient implementation of the internal carry bits of a 16-bit adder which utilizes more than one Slice.

The third important basic component of FPGAs is the programmable switch matrix (PSM). The PSMs are used to connect several Slices and therefore represent the actual routing. Every Slice’s inputs and outputs are directly connected to a PSM. In fact, four Slices are packed together to a so called complex logic block (CLB) and connected to a single PSM. [17] Figure 2.3 shows the corresponding layout.

The PSMs are connected to each other by hard wired local lines and long lines. The configuration of the PSMs determines, which wires are connected to each other. In conclusion one can say that FPGAs mainly consist of PSMs and CLBs, while the CLBs contain the actual logic cells, and the PSMs connect them to each other. However, almost every FPGA contains some additional components to enrich its capabilities. For example the implementation of bigger memory arrays using Slices is inefficient and resource-wasting. Therefore most Xilinx FPGAs contain so called Block RAMs (or BRAMs), which are hard wired memory arrays implemented as subcomponents on the FPGA. The integrated BRAMs have a much higher packing density as it would be possible with programmable hardware (like Slices). In a way, subcomponents like BRAMs can be seen as ICs inside an IC. Other hard wired subcomponents that can be found on FPGAs are:

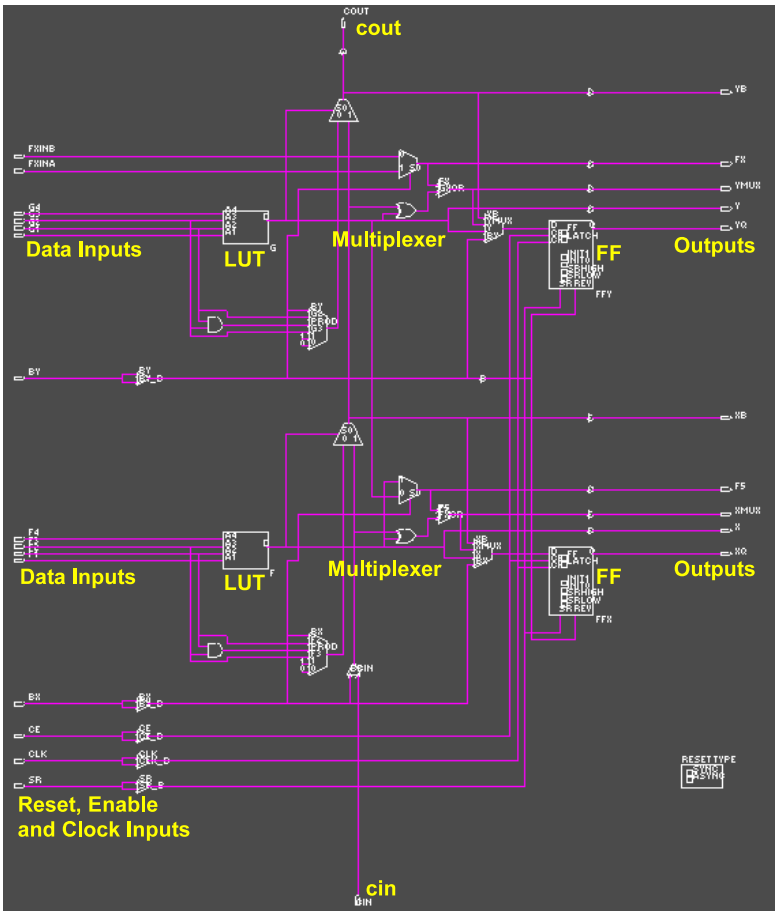


Figure 2.2: Slice of a Xilinx Virtex-4 FPGA

- Digital Signal Processor (DSP)
- Multiplier
- Ethernet MAC
- Multi-Gigabit Transceiver (MGT)
- PowerPC
- ...

Although these subcomponents are hardwired, they can be configured. For example a BRAM can either act as a ROM, as a single port RAM, as a dual port RAM (even with separated clocks), or as a FIFO (First In, First Out) memory.

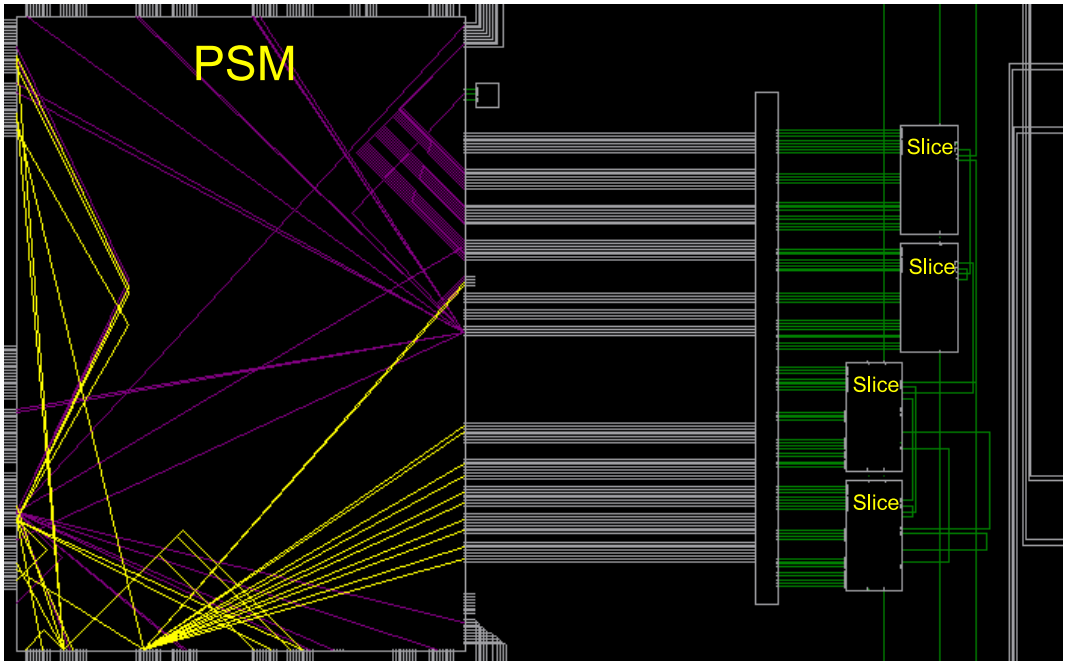


Figure 2.3: PSM and CLB of a Xilinx Virtex-4 FPGA

In conclusion, every component (LUTs, FFs, PSMs or even BRAMs) is controlled by a number of configuration bits. The entirety of these configuration bits is the payload of the so called bitstream. Therefore, it is the bitstream, which determines the current functionality of the FPGA. Thus, to configure or reconfigure an FPGA a new bitstream has to be transmitted to the FPGA's configuration unit. The next section will take a closer look at this process.

2.2 Bitstream Composition

The bitstream send to the FPGA contains both, payload data (the actual configuration bits) and meta data (32-bit commands optionally followed by n 32-bit data words) called packets. Table 2.1 shows the most important packets:

Value	Packet Name	Explanation
0xFFFFFFFF	Dummy-Word	
0xAA995566	Sync word	
0x20000000	NO-OP	
0x30008001	WCMD	Write 1 word to CMD register
0x30002001	WFAR	Write 1 word to FAR register
0x3000A001	WCTL	Write 1 word to CTL register
0x30004000	WFDRI	Write some words to FDRI register — this way the actual configuration data is written
0x30004000	RFDRO	Read some words from FDRO register — this way the actual configuration data is read back

Table 2.1: Bitstream commands [18]

Xilinx uses two kinds of packets. The first type (called Type 1 Packet) is used for register reads and writes. It contains a packet type field, two read/write bits, the address of the register that has to be read or written, and how many words shall be read from or written to the register. For the configuration registers normally only 1 word is written. In contrast, hundreds or even thousands of words are written to the FDRI (Frame Data Register, Input). Due to this it can happen that the number of bits, which are used to determine the number of following words, is not sufficient. In this case one can set this part of the word to zero — and the Type 1 Packet has to be followed by a Type 2 Packet to determine the number of following data words. Since a Type 2 Packet only consists of its header and the number of following data words, its size is sufficient in every case.

2.2.1 Configuration Registers (Excerpt)

Address	Read/Write	Name	Description
00001	Read/Write	FAR	Frame Address Register
00010	Write	FDRI	Frame Data Register, Input (write configuration data)
00011	Read	FDRO	Frame Data Register, Output register (read configuration data)
00100	Read/Write	CMD	Command Register

Table 2.2: Command Register [18]

The Frame Address Register (FAR) determines which part of the FPGA shall be read or written. It is very important for the partial reconfiguration and will be described in section 2.5. The FDRO and FDRI registers are used to read and write the actual configuration data. The Command Register enables the excution of global commands (e.g. a global reset) and the adjustment of global configurations. **Please note, that the command last loaded to the CMD register is additionally executed each time the FAR is written.** Table 2.3 lists an excerpt of the available commands.

Data	Command	Description
0000	NOP	do nothing
0001	WCFG	Write Configuration Data: used before writing configuration data to the FDRI
0100	RCFG	Read Configuration Data: used before reading configuration data from the FDRO
0101	START	Begin Startup Sequence: initiates the startup sequence
1000	AGHIGH	Assert GHIGH_B Signal: places all interconnect in a high-Z state
1010	GRESTORE	Pulse the GRESTORE Signal: sets/resets (depending on user configuration) IOB and CLB flip-flops
1101	DESYNC	Reset DALIGN Signal: used at the end of configuration to desynchronize the device

Table 2.3: CMD Register [18]

2.2.2 Example Stream

To round out this section, a small example configuration stream is presented.

Value	Description
0xFFFFFFFF	Dummy word
0xAA995566	Sync word
0x30002001	Write 1 word to FAR Register:
0x00000000	Frame Address = 0
0x30008001	Write 1 word to CMD Register:
0x00000001	WCFG
0x30004000	Write n words to FDRI:
0x5003B568	Type 2 Packet — n = 243048
0x????????	Configuration payload word 0
0x????????	Configuration payload word 1
0x????????	Configuration payload word 2
...	...
0x????????	Configuration payload word 243047
0x30000001	Write 1 word to CRC Register:
0x????????	The CRC
0x30008001	Write 1 word to CMD Register:
0x0000000A	GRESTORE
0x20000000	NO-OP
...	about 100 NO-OPs
0x30008001	Write 1 word to CMD Register:
0x00000005	START
0x20000000	NO-OP
0x30008001	Write 1 word to CMD Register:
0x0000000D	DESYNC

Table 2.4: Example Stream [18]

More detailed information about the bitstream can be found in [19], [18], [20], [21] for the Virtex series and in [22], [23], [24] for the Spartan series. Please note that the different chips of Xilinx have mostly similar bitstream commands but nevertheless differ at some crucial points. The rest of this document will focus on Virtex-4 FPGAs, since the Framework presented here has been implemented on this model.

2.3 SelectMAP Interface

The bitstream is loaded into the FPGA through special configuration pins, which serve as the interface for three different configuration modes:

- Serial configuration mode
- SelectMAP (parallel) configuration mode
- JTAG/Boundary-Scan configuration mode

In the following section the focus is on the SelectMAP configuration mode, since it has a special meaning for DPR. Figure 2.4 illustrates the pinout of the SelectMAP interface, while table 2.5 describes the pins.

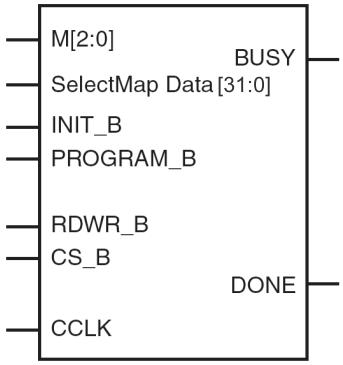


Figure 2.4: Virtex-4 SelectMAP Configuration Interface [18]

Using the SelectMAP interface the configuration can be done in two ways: continuous and non-continuous. Continuous data loading is used if the configuration controller is able to ensure an uninterrupted stream of configuration data. After power-up, the configuration controller sets the RDWR_B signal to zero and asserts the CS_B signal. This causes the device to drive BUSY low. (RDWR_B must be driven low before CS_B is asserted, otherwise an ABORT occurs.) On the next rising CCLK edge, the device begins to sample the SelectMAP data pins. Figure 2.5 illustrates how a typical continuous configuration looks like.

Non-continuous data loading is used if the configuration controller is not able to ensure an uninterrupted data stream. There are two ways to pause the configuration. Firstly, one can deassert the CS_B signal (which is called “free-running CCLK method”, see figure 2.6). Secondly, one can pause the clock (which is called “controlled CCLK method”, see figure 2.7).

Pin Name	Description
M[2:0]	Determines configuration mode
CCLK	Configuration clock source for all configuration modes except JTAG
SelectMAP Data	32 bit wide configuration and readback data bus, clocked on rising edge of CCLK. For backward compatibility it can be used in a byte-wide (8 bit) mode. In this case D0 is the most-significant bit (MSB), D7 the least-significant bit (LSB). In 32-bit mode, D0 is the LSB and D31 is the MSB.
BUSY	Indicates that the device is not ready to send readback data. For Virtex-4 devices, the BUSY signal is only needed for readback.
DONE	Active-high signal indicating configuration is complete
INIT_B	Before MODE pins are sampled, INIT_B is an input that can be held low to delay configuration. After MODE pins are sampled, INIT_B is an open drain active low output indicating whether a CRC error occurred during configuration.
PROGRAM_B	Active-low asynchronous full-chip reset
CS_B	Active-low chip select to enable the SelectMAP data bus
RDWR_B	Determines the direction of the SelectMAP data bus: 0 = inputs 1 = outputs

Table 2.5: SelectMAP Pins [18]

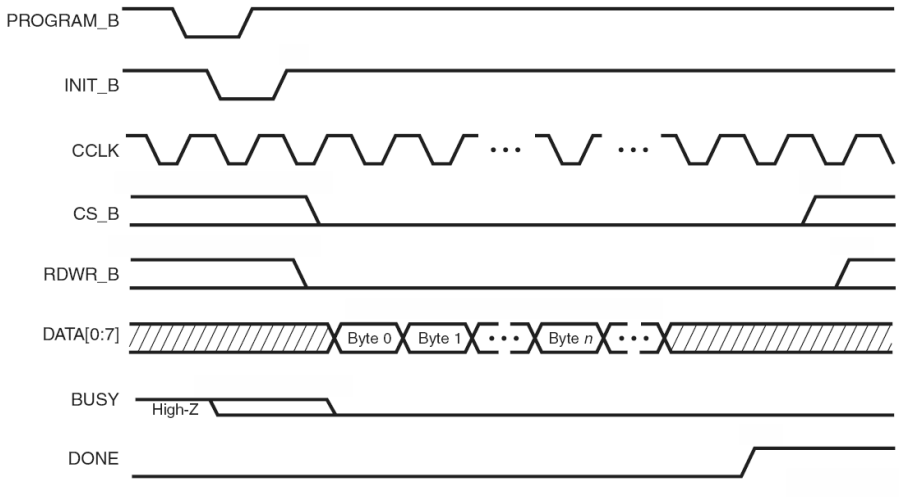


Figure 2.5: Continuous Configuration [18]

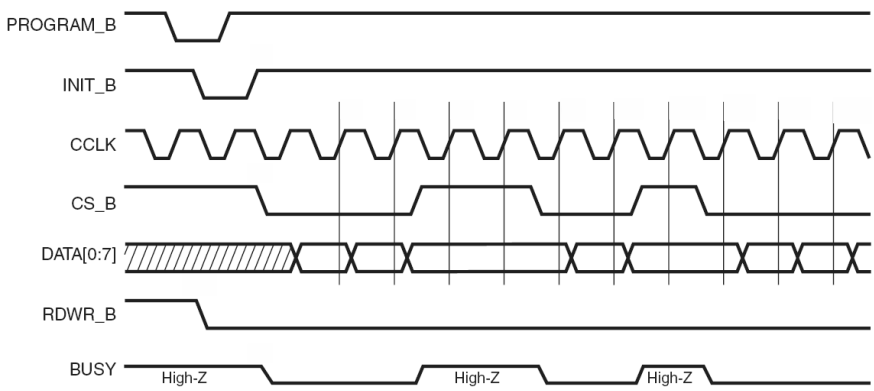


Figure 2.6: Non-Continuous Configuration using a free-running clock [18]

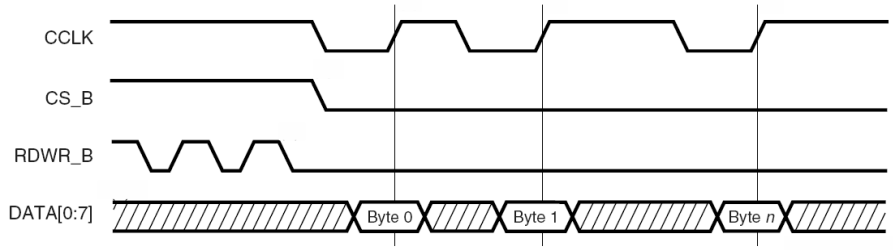


Figure 2.7: Non-Continuous Configuration using a controlled clock [18]

2.3.1 Internal Configuration Access Port

The Virtex FPGAs contain an additional configuration interface, the Internal Configuration Access Port (ICAP). This interface is realized as a subcomponent of the FPGA (like a BRAM) and is designed to provide access to the configuration controller from inside the FPGA. Thus, utilizing ICAP, the logic residing on the FPGA is able to alter itself. Of course, this has a strong correlation to DPR. The ICAP's protocol is almost similar to the SelectMAP protocol except the following differences:

- ICAP has no configuration mode pins
- ICAP has no INIT_B pin
- ICAP has no PROGRAM_B pin
- SelectMAP uses bidirectional data signals, ICAP uses 32 unidirectional input wires and additionally 32 unidirectional output wires instead
- SelectMAP uses bidirectional CCLK, ICAP uses CCLK solely as input signal

On Virtex-4 and Virtex-5 chips the ICAP can be used in an 8 bit mode and in a 32 bit mode. Both FPGA series contain two ICAP devices: TOP and BOTTOM. The two interfaces share the same underlying logic. Therefore, they can never be active at the same time. Which ICAP is active after startup and which bit mode is used, is determined by the ICAP's configuration (and hence by the initial bitstream).

2.3.2 Throughput

For Virtex-2 the official maximum frequency regarding CCLK is 33 MHz. Nevertheless, the Virtex-2's ICAP can be overclocked up to 100 MHz, but in this case the BUSY output has to be monitored. If BUSY is active, the configuration has to be delayed until BUSY is inactive again. Due to this a clock rate higher than 33 MHz does not allow continuous configuration on Virtex-2. However, Virtex-4 and Virtex-5 can be configured with 100 MHz without the need for monitoring BUSY. For readback operation BUSY is still important.

The theoretical maximum throughput of the ICAP can be calculated as the result of the bit width divided through the maximum frequency. For Virtex-2 this is 100 MB/s. The real value is lower, since at 100 MHz the configuration controller sometimes asserts BUSY. In [25] a maximum throughput of approximately 90 MB/s has been measured. For Virtex-4 and Virtex-5 the theoretical maximum throughput is 400 MB/s. In [26] this value has been measured on a Virtex-4 and even exceeded on a Virtex-5. Here, a maximum of 1 200 MB/s has been achieved, using a CCLK of 300 MHz.

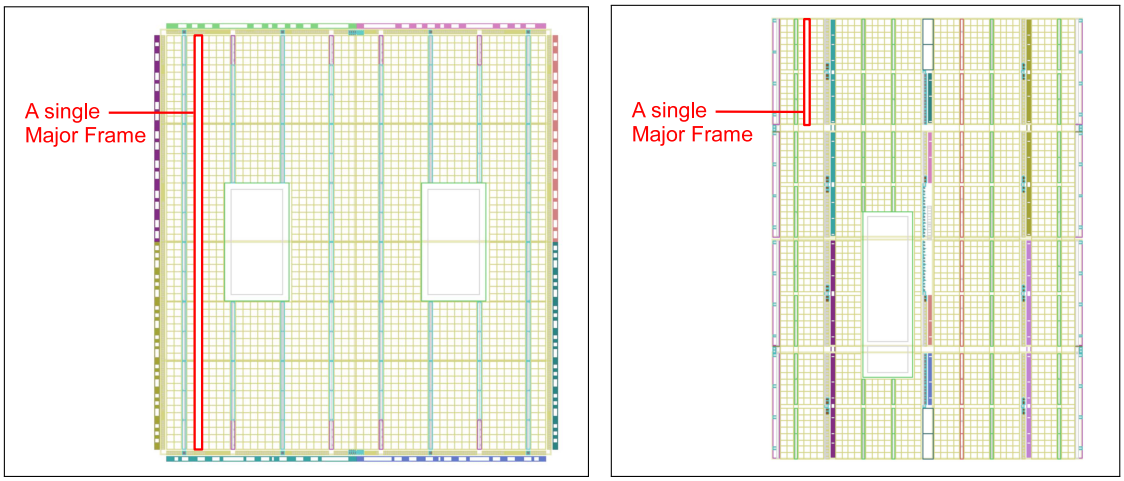
2.4 Dynamic Reconfiguration

Dynamic Reconfiguration enables the reconfiguration of an FPGA while it is running, without causing any glitches on the circuits. This feature is used in combination with partial reconfiguration (which is then called DPR and will be described later) and in radiation environments, where the radiation tolerance of FPGAs needs to be increased. The two most popular civil examples for this are space applications [27] and detector experiments at particle accelerators (like ALICE at CERN in Geneva [28] or CBM at FAIR in Darmstadt [29]). Radiation in the form of electrons, protons or even more massive particles can hit the SRAM cells of an FPGA (in flip-flops or LUTs) or a wire between them. In all these cases it is possible that the inner state of the circuit or even the configuration of the FPGA is corrupted. Thus, some FPGA vendors provide radiation hardened FPGAs [30, 31]. The logic of these chips is hit much less frequently. However, corrupting hits still can happen. To mitigate their effects, a technology called scrubbing is used [32]. Scrubbing means, the configuration of the chip is written continuously. Thus, errors in the configuration, caused by particle hits, are corrected immediately. Obviously, scrubbing depends on the dynamical reconfiguration feature. In combination with double module redundancy (DMR) [33] or triple module redundancy (TMR) [34] this can lead to a significant extension of the chip's uptime [35].

To be able to reconfigure an FPGA dynamically, the bitstream has to be adapted to become a dynamic bitstream. The difference to a normal bitstream is small and is limited to the removal of some global startup and reset commands (the commands START, AGHIGH and GRESTORE have to be removed). This skips the FPGA's startup sequence which of course would disturb the running logic. Furthermore the design must not contain LUTs which are configured to operate in Shift Register Mode or as a RAM[36].

2.5 Partial Reconfiguration

Partial reconfiguration — in contrast to full reconfiguration — means that only parts of the FPGA instead of the whole chip are reconfigured, while the rest of the chip stays untouched and can continue its calculation completely undisturbed. This enables the substitution of parts of the chip at run-time and therefore the placement of new (dynamic) modules on demand. In principle, partial reconfiguration is enabled by the configuration unit of the FPGA. Every Xilinx FPGA is divided into thin columns, called Minor Frames. These frames are the smallest reconfigurable units. They are addressed via the FAR register. Multiple Minor Frames are combined to a Major Frame[3] which either describes the full functionality of a CLB + PSM column or addresses the content of a BRAM column. (Furthermore, there are Major Frames used to describe the configuration of the IO Buffers and the configuration of the global clock net, but usually these Frames stay completely untouched by partial reconfiguration.) For Virtex and Virtex-2, every MinorFrame reaches from the bottom of the chip to its top. Thus, the Virtex and Virtex-2 series could only be parted on the x-axis. For Virtex-4 and Virtex-5, a single FPGA's column is divided into a few (e.g. 4) Minor Frames (see figure 2.8).



(a) Virtex-II Pro 20 (b) Virtex-4 FX 20

Figure 2.8: Comparison of Virtex-II and Virtex-4 regarding their frame granularity

[3]The precise number of Minor Frames per Major Frame depends on the used FPGA — for Virtex-4 it is 64

From the low-level point of view (circuit level), partial reconfiguration is already fully supported. The problem is that more complex fields of application, like reconfigurable processors or dynamically loadable hardware modules, influence the place and route tools, since here the chip has to be separated into a static area (which is not reconfigured during runtime at all) and one or more dynamic areas (which are reconfigured on demand). The tools have to ensure that the static components are only placed in the static area and the dynamic components are only placed in the dynamic areas. Furthermore, the routing of the dynamic components is not allowed to cross the boundaries between both. The solution for the placement problem is quite trivial. It can be done via so-called placement constraints, which work very well. Unfortunately, the routing problem is much more complex, since until May 2010 the original Xilinx routing tools did not support the idea of parting the routing resources. Therefore Xilinx provided so-called Partial Reconfiguration Early Access (PREA) tools [37, 38], which supported partial routing. The name of these tools is based on the fact, that Xilinx did not provide any official support for partial reconfiguration, but provided an early access for hardware developers who were keen on experimenting. The PREA tools were delivered in form of a patch, which had to be installed on top of an exact version (including the correct service pack) of the Xilinx tools. Table 2.6 lists the ISE[4] versions, which were supported by the PREA patch. The ISE Versions 10 and 11 were not supported at all, since Xilinx planed to implement the PREA patch into the normal tool flow. For ISE 10 and ISE 11 this was not successful. The first official support of DPR came with ISE 12 in May 2010. However, since the implementation of the Framework was almost completed before May 2010, ISE 12 is not considered in the following sections.

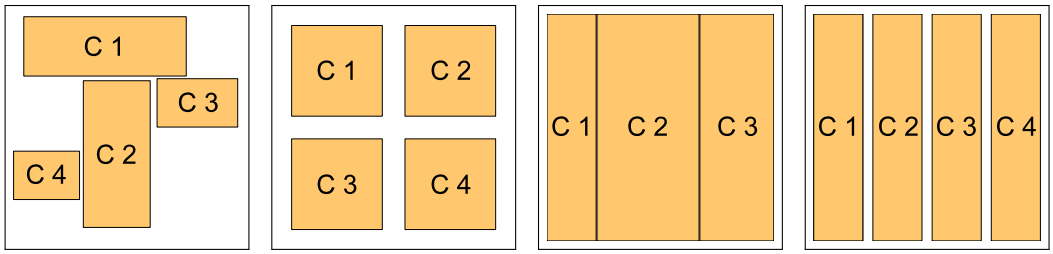
ISE 6.3i SP3
ISE 8.2i SP1
ISE 9.1i SP2
ISE 9.2i SP4

Table 2.6: Supported ISE Versions [37, 38]

[4]Integrated Software Environment

2.5.1 Partitioning Options

A very important question regarding partial reconfiguration is: what should the dynamic and the static area look like and where should they be placed? In the following the four most popular layouts [39, 40] are described. All layouts are based on rectangles and represent basic approaches. More complex structures, further improvements and hybrid forms are conceivable (see chapter 3).



(a) Free 2D Partitioning (b) 2D Block Partitioning (c) Free Horizontal Partitioning (d) Horizontal Block Partitioning

Figure 2.9: Partitioning options [39, 40]

Free 2D Partitioning

In this variant the dynamic components can be placed at any position of the chip. This enables a very efficient usage of the FPGA's resources. However, a longer reconfiguration process, including the loading and the removal of dynamic components, leads to a cumulative fragmentation of the chip. This potentially prevents the placement of a dynamic module, which actually (without fragmentation effects) would fit. An even more serious problem is, that this variant does not specify, how the dynamic components shall communicate with the static part or the outer world. Therefore the routing has to be done dynamically, which is very time-consuming.

2D Block Partitioning

Using 2D Block Partitioning, every dynamic area is a rectangle with a fixed size. Each has well defined fixed interfaces to the surrounding system. All dynamic components are using the same standard interface. Therefore every module can be loaded into any area. The disadvantage of this variant is the fixed size of the dynamic areas. If a dynamic component is smaller than the dynamic area, the rest of the area stays unused. If a dynamic component is bigger than the dynamic areas, it cannot be placed at all.

This partitioning method is typical when working with Virtex-4 chips, which natively support two-dimensional partitioning. Of course, the size of the blocks is geared to the size of the Minor Frames. Nevertheless, it is possible to do a more fine-grained partitioning based on dynamic partial reconfiguration. For this a method called read-modify-

write is used: First, the complete Minor Frame is read. Second, only a part of the bitstream is changed. Third, the modified bitstream is written back to the FPGA. Due to the dynamic reconfigurability, the unchanged parts are not disturbed by the reconfiguration process at all and can therefore be part of the static area. This method has a special meaning regarding Virtex-2 chips since here one Minor Frame spans over the complete height of the FPGA.

Free Horizontal Partitioning

In this variant the dynamic area has the same height as the FPGA, but the width is variable. Due to this, every dynamic component allocates exactly the amount of dynamic area it needs. Compared to the Free 2D Partitioning the placement algorithms are much simpler, but this layout also leads to a cumulative fragmentation.

Horizontal Block Partitioning

Using Horizontal Block Partitioning, every dynamic area has the same height as the FPGA and a fixed width. Each has well defined fixed interfaces to the surrounding system. The disadvantage of this variant is the potential waste of space regarding small dynamic components. This partitioning method is typical when working with Virtex-2 chips.

2.5.2 Inter-Module Communication

One of the biggest problems regarding partial reconfiguration is the intermodule communication. Components placed at run-time on the device may need to exchange data with each other. The most flexible, but also most complex technique is to do run-time routing. At this, the routing of the wires between dynamic and static components is calculated on demand (as part of the partial reconfiguration process) and instantiated using partial and dynamical reconfiguration. However, this has been proven to be very time-consuming [41]. Furthermore, this method does not provide any standardization, which makes it hard to add dynamic components to an existing system. Therefore, standard interfaces, which are less flexible but lead to a much faster reconfiguration process seem better suited[42].

Busmacros

Nearly all standard interfaces between dynamic and static areas make use of so called busmacros [43]. A busmacro is a hard macro built on ordinary slices, usually containing an 8-bit wide bus. It can be customized with Xilinx tools such as the FPGA Editor and is also available from the busmacro library provided by Xilinx. Properties like direction, synchronization, etc. have to be considered when creating or choosing a busmacro. Regarding the timing there are two versions of busmacros: synchronous and asynchronous.

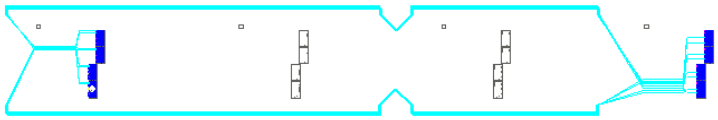
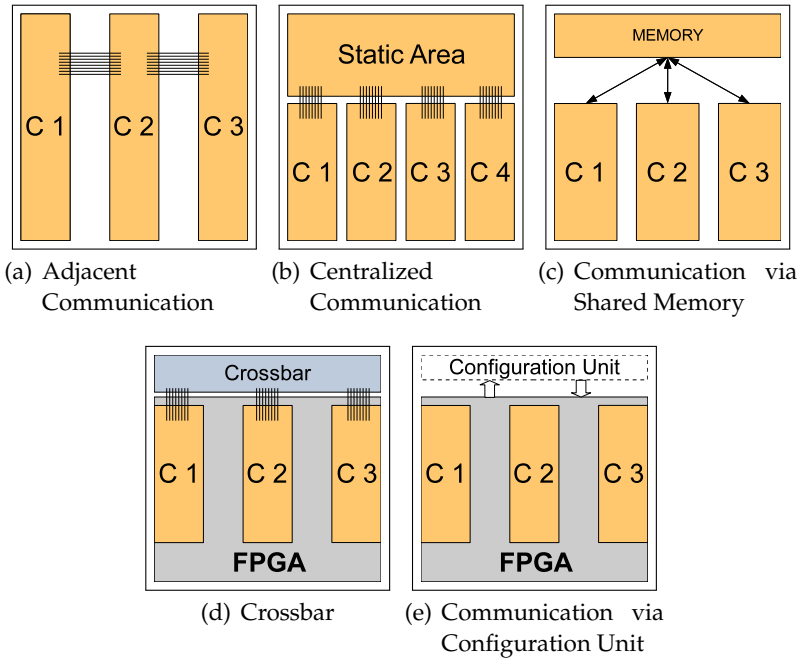


Figure 2.10: Virtex-4 asynchronous wide busmacro heading from left to right [43]

Synchronous busmacros have a better timing behavior, since they sample the data only at the rising edge of a connected clock and thus they separate the timing of the dynamic part from the timing of the static part. The disadvantage is the resulting additional pipeline step. Using asynchronous busmacros, there is no additional pipeline step, but the timing is much harder to meet [44]. Figure 2.10 illustrates an asynchronous Virtex-4 busmacro heading from left to right. The left 4 Slices each provide two hard macro inputs. The right 4 Slices each provide two hard macro outputs. The flip-flops of the Slices are by-passed, since it is an asynchronous busmacro. Synchronous busmacros make use of the flip-flops of the output Slices and have therefore an additional clock input on each output Slice.

In the following, five types of inter-module communication are presented. These five types only represent a basic set of possibilities. Improved structures and hybrid communication forms are described in chapter 3.



(a) Adjacent Communication

(b) Centralized Communication

(c) Communication via Shared Memory

(d) Crossbar

(e) Communication via Configuration Unit

Figure 2.11: Intermodule Communication [42]

Adjacent Communication

Adjacent communication is done between two dynamic areas, which are directly connected to each other using busmacros (see figure 2.11(a)). This communication method provides the fastest way for adjacent components to communicate with each other, but a communication between areas over a longer distance (e.g. between *C1* and *C3*) requires the cooperation of the components in the middle (*C2*). Furthermore the placement of the components is quite complex, since the proximity of the components influences their communication options and therefore their functionality.

Centralized Communication

Using centralized communication, each dynamic area is connected to the static area via busmacros. Communication between dynamic modules is established by the static area. In the simplest case this is done via shared memory. However, much more complex communication structures (like communication networks on chip (NoC)) are conceivable.

Communication via Shared Memory

Using a shared memory, all components have access to a memory which is part of the static area and use this memory to exchange data with each other and with the outer world. This method is very flexible, but comes with a performance problem, since not all components can access the memory at the same time. Therefore an arbiter, serializing the memory access, is needed. Obviously this reduces the throughput proportional to the number of components.

Crossbar

Crossbar communication means, that interconnections between the dynamic areas are not realized on the FPGA itself, but on the underlying board. Therefore every dynamic area contains I/O pins connected to the external crossbar. This crossbar can be simple wires, a CPLD or even an additional FPGA. The Erlangen Slot Machine (ESM) makes use of this method [45].

Communication via Configuration Unit

A special communication form coming with DPR is the usage of the configuration unit to establish a communication between the components. In this variant, the state of flip-flops or the content of a BRAM of component *C1* is read back and either stored in an external memory or written into a dedicated flip-flop or BRAM of component *C2*. This method does not need any additional wires or busmacros, but depends on a determined placement of the corresponding flip-flops or BRAMs. It is the slowest form of communication between dynamic components.

2.6 Dynamic Partial Reconfiguration

The closer look on partial reconfiguration in section 2.5 showed that in many cases partial reconfiguration is used in combination with dynamical reconfiguration. The reason is that a pure partial reconfiguration requires an absolutely strict partitioning guided by the layout of the Minor Frames — which is almost impossible. This is especially true for Virtex and Virtex-2 FPGAs, where a Minor Frame has the same height as the whole FPGA. However, there are also crucial reasons on Virtex-4, which prevent such a strict partitioning (see figure 2.12). First of all, the I/O pins of an FPGA are spread all over the chip. Therefore, declaring a part of the FPGA as a dynamic area either makes the corresponding pins unusable for the static area, or requires a route from the static area to these pins, which is contained in every dynamic module. Since the number of I/O pins is very limited, usually the second solution is preferred. This is especially true, if partial reconfiguration shall be used on an already manufactured board and the pin assignment is already fix (e.g. on Xilinx Evaluation Boards). Secondly, using more than one dynamic area can make it necessary to have routes from the static area to a dynamic area, which cross another dynamic area. Thirdly, if the chip resources utilized by the dynamic areas exceed a certain percentage it becomes impossible to route the static area without crossing the dynamic areas.

In all three cases the dynamic areas contain routes, which are actually part of the static design. These routes are called feed-through routes. Feed-through routing depends on the ability to reconfigure dynamically, since the dynamic character of the reconfiguration process keeps the functionality of the routes untouched as long as they are overwritten identically. Without dynamic reconfigurability every reconfiguration process would at least cause glitches on these routes.

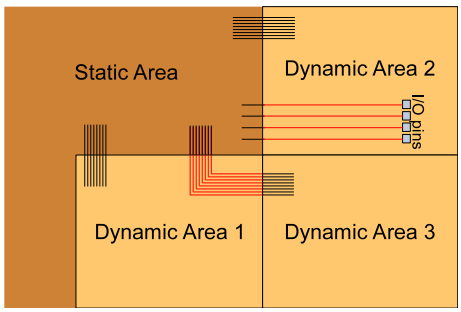


Figure 2.12: In red: feed-through routes

The usage of feed-through routes has two important consequences: The DPR tools have to ensure, that every dynamic component contains the needed static routes. Due to this the components are not instantly relocatable, since every dynamic area comes with its own static routes, which have to be added to the dynamic component, before it can be loaded.

Please note, that the routes of the dynamic components are absolutely not allowed to use parts of the static resources or parts of other dynamic areas. This would lead to an unmanageable number of static route fragments needed by several dynamic components, which in the worst case could block each other.

2.6.1 Tools

iMPACT

iMPACT is the programming tool used to transfer the bitstream from a PC to the FPGA. It supports all types of bitstreams: full bitstreams, dynamic bitstreams and partial bitstreams. The reason is that iMPACT simply transfers the bitstream to the FPGA and does not care about its content. Dynamic bitstreams just do not contain the reset and startup commands, partial bitstreams just do not address all Minor Frames of the FPGA.

iMPACT also allows to read back the current configuration. However, iMPACT is designed to be executed on a PC and therefore makes use of interfaces like the parallel port or USB. These interfaces depend on a special Xilinx adapter chip which has a very limited throughput and thus is not able to achieve the maximum throughput shown in chapter 2.3.2.

PlanAhead and ISE

PlanAhead is a floorplanning tool, originally developed by Hier Design, which in 2004 has been acquired by Xilinx. Regarding DPR it is a very important tool, since it provides the functionality to create dynamic areas and to assign dynamic modules to them. Furthermore it offers a simple way to generate all the needed bitstreams, which are the full bitstream (containing the static area and the initial dynamic modules) and the partial bitstreams representing the dynamic modules. PlanAhead creates placement and routing constraints to assure a correct feed-through routing. As a consequence of this feed-through routing, every module has to be synthesized once for every dynamic area it shall be loaded to. Hence, a design containing two dynamic areas and three dynamic modules (which shall be loadable to both areas) leads to six partial bitstreams.

During all these steps PlanAhead only acts as an orchestrator. The actual tools, creating and merging the bitfiles are part of the Xilinx ISE enhanced by the PREA patch. Therefore, everything could be done without PlanAhead, setting the routing and placement constraints manually and calling the corresponding ISE tools directly. However, PlanAhead makes all this much more intuitive and easier manageable, which finally leads to higher productivity. Furthermore, PlanAhead does not only offer a GUI [5], but also a CLI[6] which enables the programmer to orchestrate the creation of the bitstreams via the console or even automated via scripts.

[5]Graphical User Interface
[6]Command Line Interface

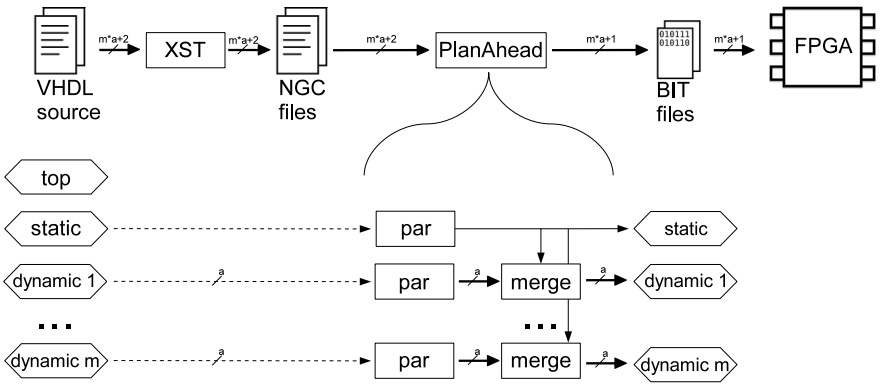


Figure 2.13: DPR tool flow using PlanAhead

Figure 2.13 illustrates the PlanAhead tool flow. Everything begins with the source files. They can be written in VHDL, Verilog, HandleC, SystemC or any other HDL[7]. First of all a top file is needed, which solely describes the basic structure of the design. It contains the static area as a black box subcomponent, all dynamic areas as black box subcomponents and the busmacros connecting them to each other. A synthesize tool (e.g. XST[8]) is used to translate this top file to a netlist (ngc-file). Secondly, the static design is described in its own file (which can contain subcomponents in further files). It is also synthesized. Last but not least the dynamic modules have to be described. Let's assume that m modules and a areas are used. As a consequence, the synthesize tool has to be started $m \cdot a$ times to generate m dynamic ngc-files per area. These ngc-files serve as input files for PlanAhead. The top.ngc is used to get the basic structure, static.ngc defines the static components and the dynamic ngc-files are used to define the dynamic modules.

PlanAhead is used to do the floorplanning. At this step the size of the dynamic and static areas is determined and the busmacros are being placed (see figure 2.14). Finally, PlanAhead calls *par*[9] to generate the ncd[10] files. This has to be done once for the static part and a times for each module (provided that each module shall be loadable to each area). Thus, par is called $a \cdot m + 1$ times. Next, the originated ncd files are being merged. This enables feed-through routing. Finally the merged files are translated to bitfiles using bitgen. The results are one full initial bitfile, containing the static area and the initial dynamic modules, and m partial bitfiles for each of the a dynamic areas. Beyond that PlanAhead provides an "empty" bitfile for each dynamic area, which represents the dynamic area without any dynamic module loaded to it. These "empty" bitfiles solely contain the feed-through routes.

[7] Hardware Description Language
[8] Xilinx Synthesize Tool
[9] Xilinx place and route tool — part of the ISE and enhanced by PREA
[10] native circuit description — the placed and routed design mapped to the components in the FPGA

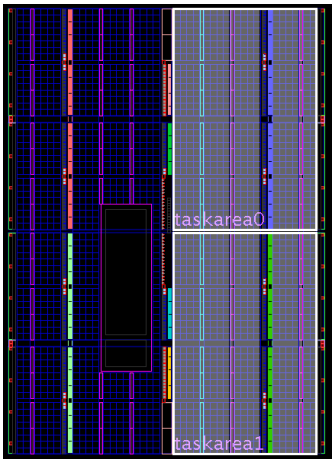


Figure 2.14: Floorplanning via PlanAhead — example with 2 dynamic areas

Missing Language Support

It has been shown, that Xilinx tools are essential for DPR. Having said that, it is important to underline that these tools represent the necessary basic equipment, but they do not completely fulfill the needs of hardware developers. This is primarily caused by the lack of language support for DPR. Of course all HDLs can be used to generate partial designs, but in this case an external work flow (as shown above) is needed. What is missing is an intrinsic language construct which can be used to denote the dynamic character of submodules, like *new* is used to denote the dynamic instantiation of objects in C++.

2.7 Object-Oriented Programming and Multithreading

2.7.1 Object-Oriented Programming

Since the 1960s object-oriented programming (OOP) changed the way software design is done. Using OOP the central point of software development are interacting objects with assigned attributes and methods. For example describing the motion of a car in procedural programming results in a program that changes some variables representing the position of the car. In OOP the car is represented by a *car* object with assigned attributes for its position and an assigned method *move*. Other objects can call *move* to move the car. The difference between these two descriptions is fundamental, since in procedural programming the handled variables are passive whereas in the OOP every object acts as an active part of the program[46, 47]. Today's object-oriented languages include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance which help programmers and programming teams to write well-arranged and re-usable code.

Data Abstraction

Data Abstraction means to reduce and factor out details so that one can focus on a few concepts at a time. For example *cars* and *bikes* can be abstracted to *vehicles*, which at least all have a *position* and a *velocity*. This way it is possible to summarize them under one label and to handle common attributes and methods with one piece of code.

Encapsulation

Encapsulation means that the access to an object's internal states is restricted. This can be used to hide implementation details from other objects. This way the developer has the possibility to change the internal implementation of an object without affecting the rest of the program. Furthermore, an erroneous access to internal states can be prevented.

Modularity

Modules represent a separation of the program into logical parts and improve maintainability by enforcing logical boundaries between modules. These boundaries are incorporated into the program through interfaces, which define the elements that are provided and required by the module. This enables members of a programming team to develop their modules independently.

Inheritance

Inheritance is a way to form new, more specialized classes using base classes that have already been defined. This helps to reuse existing code with little or no modification, since the new sub-classes inherit attributes and behavior of the base classes. The functionality of the new class can be changed by re-writing methods or creating new methods in the

derived class and leaving the rest untouched. For example, *cars* and *bikes* could be represented by classes which inherit from a *vehicle* class and differ in their implementation of *move* (e.g. regarding maximum speed).

Polymorphism

Polymorphism is the ability of an object of one type to appear as and be used like an object of another type. This allows objects belonging to different types to respond to method or property calls of the same name, each one according to an appropriate type-specific behavior. The programmer (and the program) does not have to know the exact type of the object. The exact behavior is determined at run time, which is called late binding. For example, an object that handles *vehicles*, but does not know whether the *vehicle* is a *car* or a *bike*, just calls *move*. Based on late binding the correct *move*-method is called, depending on the real object type (*car* or *bike*).

Instantiation

In OOP the programmer does not directly describe the objects, but (abstract) classes, which summarize the attributes and methods of objects with a specific type. These classes have to be instantiated to become a real object. All objects of the same class have the same set of attributes and the same methods, but the content of the attributes can differ. For example *Herbie* is an instance of the class *car* whose *color* is *white*. Other *cars* also have a *color*, but it does not have to be *white*. Normally a class can have an unlimited number of instances. Sometimes this behavior is not wanted. Therefore design patterns such as singletons exist which restrict the number of instantiations of a class.

2.7.2 Multithreading

Multithreading means that multiple threads are executed concurrently. Thereby "concurrently" either means really simultaneously (like two hardware components) or quasi-simultaneously (which means that a scheduler executes the threads alternately). The usage of multithreading became standard in all current operating systems. Even small embedded systems like in mobile phones make use of it, since multithreading allows the operating system to provide various functionality to the user at the same time. Furthermore, threads waiting for external events or external data are not blocking the whole system. The corresponding thread is just being postponed.

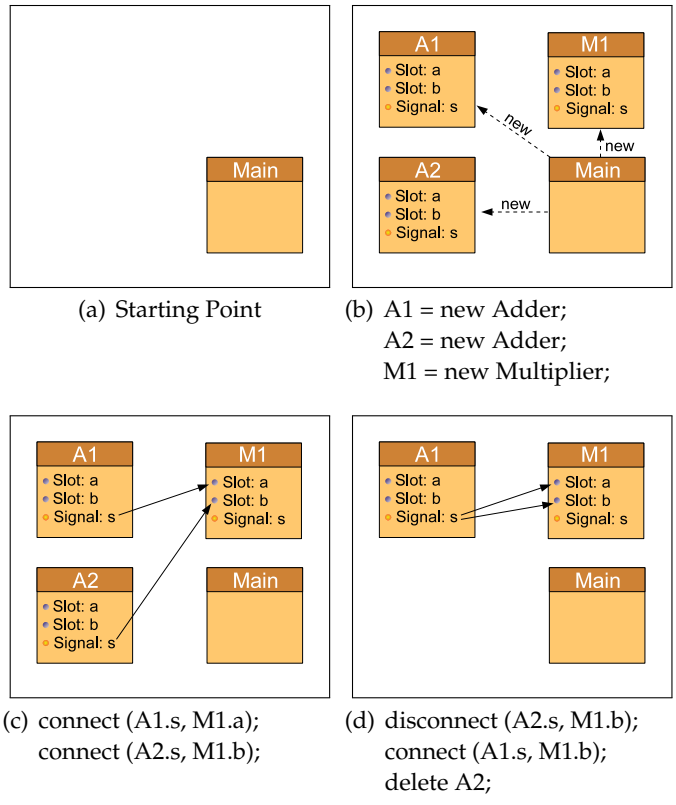
However, multithreading does not only come with advantages. It also brings synchronization problems which in the worst case could block the whole system. These synchronization problems can be solved with semaphores and mutexes provided by the operating system.

In modern languages (like C++ or Java) OOP and multithreading come together. Thereby it turned out, that OOP is a very good way to realize parallel programming. The concept of encapsulation and modularization makes it possible to handle multithreading in a

well-defined way. To enable multithreading in an OOP language, selected objects are defined as independent instances, which are running in parallel to the rest of the program.

2.7.3 Qt

One of the major problems regarding multithreaded OOP is the inter object communication. If it is done the wrong way it can lead to erroneous results or even to a blocking program. In 1992 a team of programmers started to develop Qt (pronounced as the English word "cute") which is a cross-platform application development framework. It offers the programmer an easy way to create a GUI and makes use of well-defined interfaces (called Signals and Slots). Signals send a message out of an object, while Slots are the corresponding receivers. An object can make use of several Signals and Slots. The connection between objects is established via the connection of a Signal to a Slot. Figure 2.15 illustrates the establishing and the removal of connections between Signals and Slots.



(a) Starting Point

(b) A1 = new Adder;
A2 = new Adder;
M1 = new Multiplier;

(c) connect (A1.s, M1.a);
connect (A2.s, M1.b);

(d) disconnect (A2.s, M1.b);
connect (A1.s, M1.b);
delete A2;

Figure 2.15: Qt: Signals, Slots and their usage

Qt became very popular, since it was used as the programming base of KDE[11], which 1998 was going to become one of the leading desktop environments for Linux. Con-

[11] K Desktop Environment

troversial debates on Qt's licensing finally led to a dual licensing model. Today, Qt is available under the QPL[12] and under the GPL[13]. Due to this many open source projects made use of Qt. Today Qt can be found in KDE, Google Earth, Opera, Skype, VLC media player and many other applications. Furthermore it has been ported from C++ to other languages such as Python, Ruby, Java, PHP, Haskell and Perl [48]. The announced goal of the Qt developers is to have "Qt everywhere" [49].

[12] Q Public License
[13] GNU Public License

2.8 Software to Hardware compiling

Operating on register transfer level, hardware developers have to handle clocks and clock domains. On the algorithmic level this is not necessary any longer. Here, designs can be described without the need to care about clocks. Hence, many hardware developing groups are looking forward to be able to describe hardware on the algorithmic level or even above, which is called ESL (Electronic System Level) Design. Since software languages such as C or Java already operate on this level, the idea of an automated translation from software to hardware came up. The basic concept of this approach is to take an ordinary software program (e.g. a C program) and to translate it to an ordinary HDL such as VHDL. This process is called HLS (High Level Synthesis). In principle, this translation can be done in a very self-evident simple way using a single FSM[14] (see figure 2.16). However, this naive approach leads to an absolutely sequential execution of the program and therefore to some kind of FSM based emulation of the processor which is executing the software. Since reprogrammable hardware is much slower than integrated processors, this finally leads to a much worse performance.

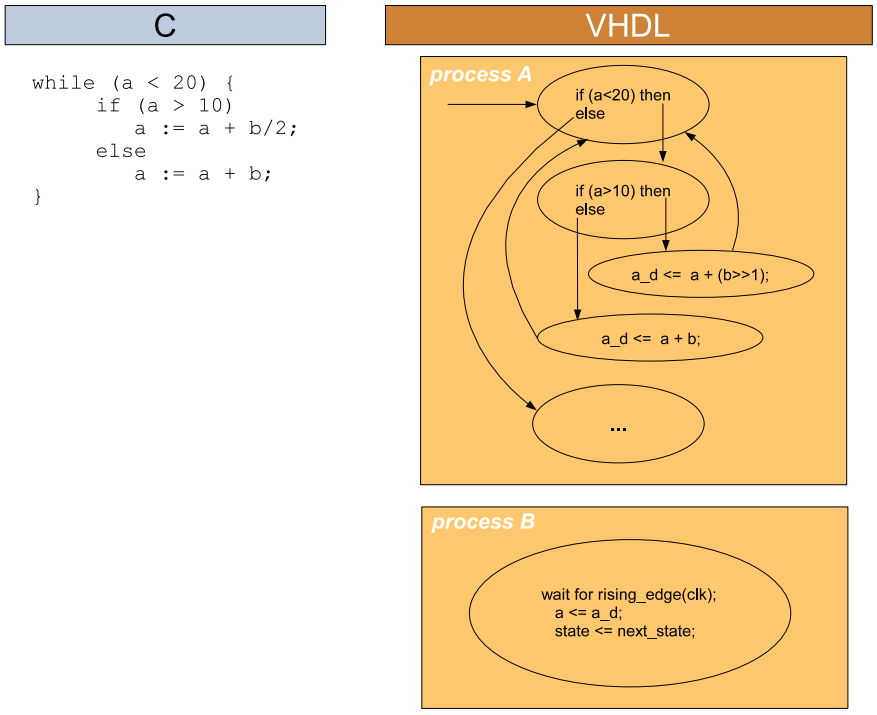


Figure 2.16: Naive software to hardware translation

[14]Finite State Machine

What is really needed is a translation method, which does not naively translate software to hardware, but uses the intrinsic parallelism of the hardware as much as possible. This intrinsic parallelism is the actual excellence of hardware. Unfortunately, parallelizing a given piece of software is a very hard task residing in the gray area between syntax and semantics. Thus, today many research groups are looking for the best way to automate this process. Some typical representatives are shown in chapter 3.3.

All improved hardware to software compilers make use of the so called data flow analysis. At this, the compiler analyzes how the variables depend on each other and generates a data flow graph. All nodes which are not connected to each other can be computed in parallel. Furthermore pipelining can help to parallelize processes, which depend on each other. The main problems are programming constructs like branches or loops, which potentially invalidate a pipeline or even hinder the usage of pipelines. Unfortunately these constructs cannot be forbidden since they are necessary to achieve Turing completeness.

3 State of the Art

This chapter focuses on the current state of the art regarding DPR[1] and HLS[2]. In the first section, several important research activities regarding DPR are illuminated. The second section focuses on reconfigurable processors, since this is one of the most important utilizations of DPR. Furthermore, reconfigurable processing can often be found in combination with HLS. The third section takes a look at the conversion from software to hardware, and therefore at the synthesis from algorithmic level to register transfer level. It presents fundamental principles and current research as well as popular languages used for HLS. Finally, the fourth section focuses on frameworks which combine DPR and HLS.

Although this state-of-the-art chapter has more than 30 pages, it can only serve as a short survey of DPR and HLS. There are many more research groups, languages and industrial projects dealing with DPR or HLS than can be presented here. Thus, every sub-topic focuses on a few selected, characteristical examples.

3.1 Dynamic Partial Reconfiguration

On conferences and in journals one can find an almost countless number of publications dealing with DPR. Unfortunately there is a growing trend to incoherent publications which focus on a fancy side issue (like an optimized defragmentation algorithm for free 2D partitioning), but lack a working implementation or even an idea how their approach could be used in practice (e.g. they focus on free 2D partitioning but have no idea how to establish the inter-module communication). Thus, in the following section the focus is on DPR projects that include a running implementation. For further investigations the book "Dynamically Reconfigurable Systems — Architectures, Design Methods and Applications" [50] can be recommended. It presents a good overview to current DPR projects.

[1] Dynamic Partial Reconfiguration
[2] High Level Synthesis

3.1.1 Erlangen Slot Machine

The Erlangen Slot Machine (ESM) has been developed at the Institute for Hardware-Software-Co-Design (HSCD [51]) which is part of the technical faculty of the University of Erlangen-Nuremberg. The ESM is a setup consisting of 2 FPGA boards [52]. The first board (called Babyboard) contains the main FPGA which performs the actual calculations. For this, it is separated into slots, which can be reconfigured individually (based on partial reconfiguration). The communication between the reconfigurable modules is realized via the second board (called Motherboard). It contains a Crossbar FPGA (see chapter 2.5.2) and peripheral devices. Thus, different I/O requirements (e.g. a PCI Motherboard in a PC versus a standalone Motherboard in an embedded system) lead to different Motherboards but the Babyboard implementation always stays the same.

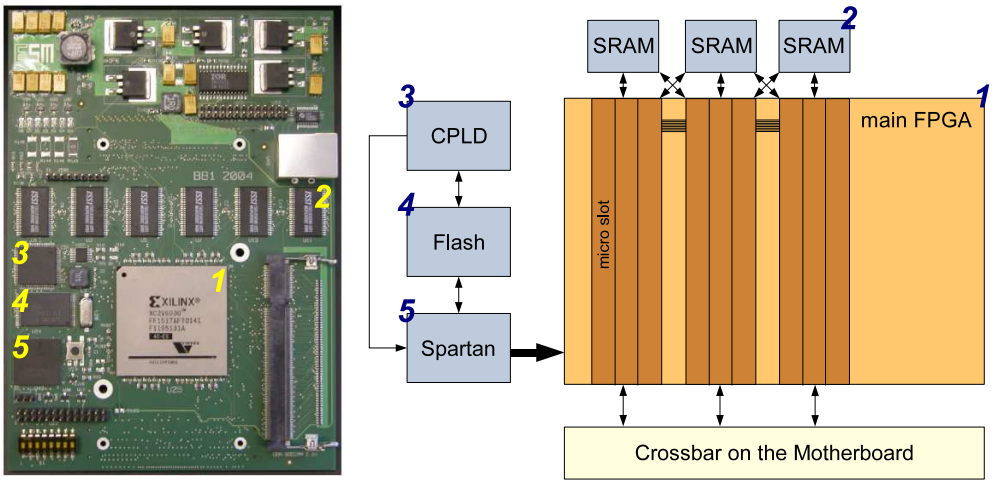


Figure 3.1: Photo[53] and architecture[54] of the ESM Babyboard

The Babyboard consists of a Xilinx Virtex-2 6000 (the main FPGA), several SRAMs, a small CPLD, a Spartan-2E 100 FPGA, Flash memory and the interconnects to the Motherboard [55]. The CPLD configures the Spartan-2E at startup. Thereafter the Spartan is responsible for the (partial) reconfiguration of the main FPGA. The ESM makes use of a mixed form of Horizontal Block Partitioning and Free Horizontal Partitioning: the main FPGA is logically divided into columns of 2 CLBs called micro slots. These micro slots are the smallest reconfigurable units. Each reconfigurable module consists of a given number of micro slots and can be placed freely, using the micro slot grid as minimal granularity. The Spartan is responsible for the adaptation of the bitfiles so that the reconfigurable modules can indeed be placed freely — although the micro slots are slightly different from each other [56]. However, the free placement leads to fragmentation effects. Due to this, the HSCD also focused on defragmentation algorithms [57, 58]. Those algorithms depend on the possibility to replace reconfigurable modules. The corresponding functionality is also implemented in the Spartan-2E.

Since the inter-module communication is realized via an external Crossbar FPGA, there are no feed-through routes crossing the reconfigurable modules. Thus, the ESM implements a strict partial reconfiguration which does not depend on dynamical reconfigurability [59]. Every micro slot is connected to a set of I/O pins which are connected to the Crossbar FPGA. Furthermore every micro slot is connected to SRAM pins. The SRAMs serve as additional local memory of the reconfigurable modules and can additionally be used for shared memory communication between neighbor modules (e.g. for streaming applications). Due to the number of pins needed to access one external SRAM, each reconfigurable module willing to use an SRAM must consist of at least 3 micro slots and has to follow more restrictive placement rules [60, 42].

The Motherboard consists of the Crossbar FPGA and additional application-specific controllers and interfaces. Currently, the Crossbar FPGA is a Spartan-2E 600. It realizes the fully flexible and run-time adaptable inter-module communication, so that modules which are running on the main FPGA and need access to each other or to external peripherals are not restricted by the physical location of the main FPGA's peripheral pins [61, 62].

The ESM was first published in 2005. Since then, it has been enhanced and improved. One very important improvement came with the introduction of the Virtex-4, since this FPGA series is reconfigurable two-dimensionally. The original design of the ESM was strongly influenced by the architectural limitations of the Virtex-2 FPGAs and their solely column-wise reconfiguration. This limitation was the primary motivation for the implementation of the Crossbar on a separate FPGA. The reconfiguration options coming with the Virtex-4 enable a union of the Babyboard and the Motherboard. Instead of using a main FPGA and a separate Crossbar FPGA, both designs could be united in one Virtex-4 FPGA — placed in different (separately reconfigurable) rows. Thus, the strict partitioning would be kept, but the additional effort of managing the reconfiguration of two FPGAs could be omitted. Furthermore, the micro slots could not only be separated on the x-axis, but also on the y-axis. This lead to the usage of a strict 2D Partitioning. However, such a two-dimensional partitioning leads to completely new requirements regarding the communication interfaces between the reconfigurable modules. One solution to handle these extended requirements is the usage of the ReCoBus, presented in the following part.

Nowadays, the HSCD Erlangen also focuses on the implementation of applications running on the ESM and using its reconfiguration features. Examples are video streaming [63], parallel sorting [64] and real-time image recognition [65].

3.1.2 ReCoBus

One of the central limiting factors for the wide use of DPR is the problem of inter-module communication. To solve this problem in a standardized but also flexible way, the HCSD developed the ReCoBus. The phrase "ReCoBus" is used in two ways. Firstly, denotes a hard macro containing all logic and routing of a backplane bus. Secondly, it describes the technology of integrating modules that have been synthesized, placed and routed in advance to a reconfigurable backplane (as known from the ESM) [66]. The ReCoBus is built to connect micro slots with a size of 2 CLB columns to each other or to the outer world. As trade-off between flexibility and maximum throughput, a multi chain architecture has been chosen. The number of chains is user-settable. A single chain contains 8 signals [67].

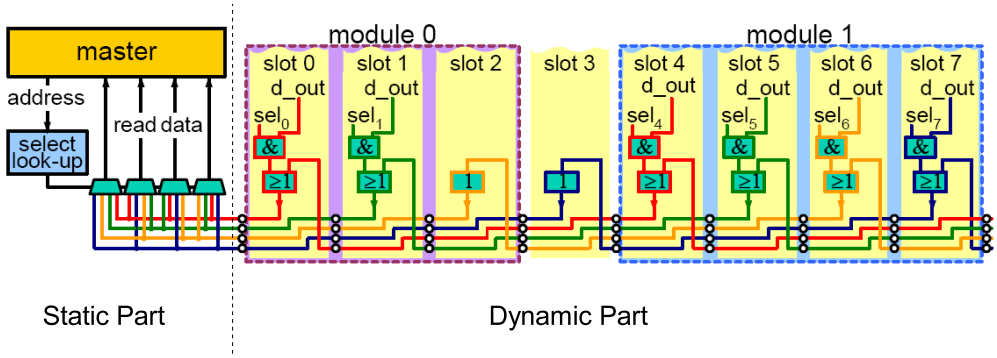


Figure 3.2: Simple ReCoBus example utilizing 4 chains [66]

Figure 3.2 illustrates a simple ReCoBus example with 8 slots, 4 chains and 2 reconfigurable modules. As one can see, based on the 4 chains, only every 4th slot is connected to a single chain. The other chains inside a slot are simple feed-through routes. Hence, the number of chains determines on the one hand the granularity of the connections and on the other hand the routing delay. A finer granularity leads to a higher delay and vice versa. Furthermore, the size of a reconfigurable module (more precisely the number of utilized slots) determines the number of chains which can be accessed by the module. This correlation is intended, since higher bandwidth is usually correlated with higher complexity and therefore with bigger modules. For example, a UART (Universal Asynchronous Receiver Transmitter) is a quite small module and requires only 1 chain (8 bit), while an Ethernet controller is much more complex, hence much bigger, and requires at least 4 chains (32 bit). The corresponding chain-module interface occupies only two LUTs in all (one for the select logic and one for the OR logic). If a module does not want to make use of the provided connection, the OR logic can be exchanged by a simple feed-through logic.

Making usage of the ReCoBus comes with a changed synthesis flow and requires a late insertion of the ReCoBus interconnections (implemented as hard macros). Furthermore, the slot grained placement leads to the need for the same bitfile adaptations as necessary at the ESM [68]. Thus, the HSCD provides an altered toolflow and a GUI called the **ReCoBus-Builder** [69], designed to simplify the design of a reconfigurable system and to encapsulate the implementation of the ReCoBus. A very detailed description of the ReCoBus-Builder and its functionality can be found in the ReCoBus User Guide, available at [66]. However, one of the most outstanding characteristics of the ReCoBus-Builder is its toolflow, since it is not using the special DPR toolflow offered by Xilinx (see chapter 2.6.1). In fact, the ReCoBus-Builder is designed to be used as a DPR floorplanning tool instead of PlanAhead, since PlanAhead relies on simple bus macros and does not support the ReCoBus at all. Furthermore, the HSCD found a workaround to allow strict area limited routing, without depending on the PREA patch. This workaround is based on a method called Blocking. In this method, all outgoing signals within a CLB or a Block RAM are allocated and hence marked as not usable for the Xilinx routing tools. This allows the definition of prohibited regions, which cannot be used by the router. If a dynamic area is surrounded by such a prohibited region, the router is forced not to instantiate any routes between the static and the dynamic area, although it is not a DPR aware router [70, 71].

In conclusion, the ReCoBus can be seen as a further development of the ESM. The strict partitioning is lost (due to the feed-through chains which require dynamic reconfigurability), but it can be used to avoid the need for an external Crossbar FPGA, even on Virtex-2. Currently, the ReCoBus-Builder supports Virtex-2 and Spartan-3. Virtex-4 support is announced and will become a very interesting option since the ReCoBus could be used to solve the communication problems coming with the 2D Partitioning of the ESM's Virtex-4 extension.

3.1.3 Two-Dimensional Partitioning including Online Routing

In chapter 2.5, the options regarding chip partitioning have been presented. The most flexible, but also most complex approach is the Free 2D Partitioning. In this variant the dynamic components can be placed at any position of the chip, which enables a very efficient usage of the FPGA's resources. However, a huge drawback is the lack of standardized communication interfaces between the dynamic modules and the static part of the chip. Furthermore, fragmentation effects can occur.

The Institute for Information Processing Technology (Institut für Technik der Informationsverarbeitung - ITIV[72]) of the University of Karlsruhe has been focusing on two-dimensional partitioning since 2006 [73, 74, 75]. They make use of a mix of Free 2D Partitioning and 2D Block Partitioning. The chip is divided into a static area (which is not reconfigured at all) and one or more dynamic blocks. Each dynamic block serves as a placeholder for one or even more dynamic modules. In contrast to the conventional Block Partitioning, the dynamic modules can be placed freely inside a dynamic block. This enables a much more efficient usage of the chip resources inside a block but does not

provide any standard communication interfaces between the dynamic modules and the static part. To solve that problem, the ITIV developed an on-chip run-time router, which is running on an embedded processor inside the FPGA and calculates the routes from the dynamic component's outputs to the static design's inputs and vice versa [76, 77]. The separation of the FPGA into several dynamic blocks can be seen as coarse-grained reconfiguration method. In contrast, the instantiation of dynamic routes affects only a few slices and routing points. The ITIV calls this mixture of coarse-grained and fine-grained reconfiguration methods "Multi-grained Reconfiguration" [78, 79].

A further problem regarding reconfiguration (especially fine-grained reconfiguration as well as run-time routing) is that one cannot see with the naked eye what happens inside the chip. For normal static designs, tools like the FPGA editor, which display the design based on its source files, are used. These sources are used for bitfile generation and therefore the FPGA editor's view is identical to the design on the chip. Unfortunately, this is only true until a reconfiguration occurs. From that moment on, the design on the FPGA differs from the original sources and therefore from the FPGA editor's view. Tools like PlanAhead provide the possibility to generate all possible permutations of components regarding coarse-grained reconfiguration in advance (and therefore the chance to take a look at every possible combination), but this method cannot be used for fine-grained reconfigurations, especially not for run-time routing. Thus, the ITIV developed a tool which operates in the opposite direction: It reads back the configuration of the chip and generates a netlist from the resulting bitfile. Finally, this netlist is viewable. Therefore, this reversed tool flow enables hardware developers to take a look at the current design, even after fine-grained reconfiguration [80].

A very impressive demonstration of the multi-grained two-dimensional reconfiguration methods including dynamical routing and on-line visualization has been given at the DATE[3] conference in 2007 [41].

3.1.4 Busmacros

Using block partitioning (either 2D or horizontal) enables the utilization of standard communication interfaces between the dynamic areas and the static area. These interfaces are realized as hard macros. Until 2005, the standard method on the Virtex-2 was to use TBUFs[4] as a basis for these busmacros [81]. In 2004, the ITIV presented an alternative busmacro implementation based on ordinary Slices [82]. Since TBUFs turned out to be problematic and the Virtex-4 and all later FPGAs do not contain TBUFs any longer, this new method attracted Xilinx's notice. Thus, in 2006 the slice-based busmacros became the standard Xilinx busmacro implementation [43, 83]. Furthermore, the ITIV developed a method to simulate the behavior of a dynamic design based on "virtual multiplexors" which for simulation act as mutliplexors and for synthesis are replaced by busmacros

[3] Design, Automation and Test in Europe
[4] Tristate-Buffer

[84]. Finally, together with the LIS Munich they developed an XDL (Xilinx Design Language) based tool to automatically generate and route customizable busmacros [85].

3.1.5 Autovision

In contrast to other DPR research teams, the Institute for Integrated Systems (Lehrstuhl für integrierte Systeme - LIS[86]) of the Technical University of Munich first focused on an application (video processing) and enriched this application with DPR later on. First publications regarding MPEG-4 and MPEG-7 can be found in [87, 88, 89]. These works primarily focus on the porting of video decoders from software to hardware in a very efficient way. Thereby an SoC architecture consisting of a standard embedded RISC core as well as coprocessor modules for macroblock algorithms and motion estimation is used. In 2004, the LIS presented a first approach to implement the whole system on a single FPGA and to exchange the coprocessor modules on demand, based on DPR [90, 91]. Since 2005, the LIS has been publishing papers which focus on automotive [92], more precisely on video-based driver assistance like lane departure warning. Here, DPR is used to offer the necessary flexibility coming with the different driving conditions. For example, driving on a highway at daylight needs a completely different coprocessor algorithm than driving in a tunnel with low luminance level [93, 94, 95]. At DATE 2008, the LIS showed a very impressive demonstration of the object detection, using a video stream originally coming from a moving vehicle's camera (see figure 3.3) [96]. The resulting processed video streams can be found at [97] and at [98].

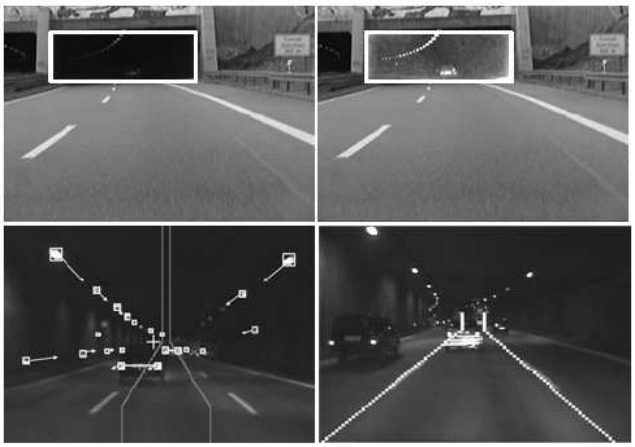


Figure 3.3: Detection of the tunnel (upper left) and contrast enhancement (upper right) Taillight tracking (lower left) — Edge detection and lane tracking (lower right) [99]

3.1.6 JCAP

Talking about real-world applications always leads to talking about pricing. Due to this, it is very helpful to be able to use DPR not only on Xilinx's high-end Virtex series, but also on the much cheaper Spartan series. Thus, the ITIV investigated the reconfiguration possibilities regarding Spartan-3 [100]. One major drawback of Spartan-3 is the missing ICAP. Hence, the ITIV developed the so called JCAP, which is an interlink between selected FPGA pins and the FPGA's JTAG[5] interface, finally enabling the chip to reconfigure itself [102, 103].

3.1.7 Reconfiguration Speed

The time a reconfiguration process consumes determines in which fields of applications DPR can be used and in which it cannot. The less time is needed for the reconfiguration itself, the more reconfigurations can be performed and the more flexible a design can become. Thus, it is an aim of all DPR research groups to increase the reconfiguration speed as much as possible. In 2008, the ITIV Karlsruhe and the LIS Munich presented an IP core that enables fast on-chip DPR close to the theoretical maximum speed [25]. This fast ICAP controller is based on DMA (direct memory access) on the PLB (processor local bus) and bitfile compression[104], which significantly lowers the needed throughput. Compared to other realizations, an increase in speed by a factor of 20 could be obtained [105]. In [26] a throughput of 400 MB/s has been measured on a Virtex-4 and even exceeded on a Virtex-5. Here, a maximum of 1 200 MB/s has been achieved, using a clock rate of 300 MHz.

[5] Joint Test Action Group [101]

3.2 Reconfigurable Processors

The idea of reconfigurable processors (RPs) goes back to the early 1990s, where so called ASIPs (application-specific instruction processors) became popular. These ASIPs are equipped with an instruction set, tailored to benefit a specific application such as fast Fourier transformations, sorting algorithms or video stream compression. Traditional ASIPs are implemented using fixed logic and therefore not adaptable to new or changing requirements. This changed with the upcoming utilization of FPGAs. An ASIP implemented on an FPGA is reconfigurable and therefore adaptable to completely different algorithms and their requirements. Even a change of the instruction set *during operation* is conceivable [106]. In principle, RPs can be divided into 3 groups [107].

Group 1 — Monolithic

The first group of RPs makes use of exactly one reconfigurable area. This reconfigurable area is a placeholder for one RFU (reconfigurable functional unit) at a time, which either replaces the traditional fixed ALU (arithmetic logic unit) completely or serves as a co-processor to a GPP (general purpose processor). The area has a fixed size which is determined while designing the system. If it is too small, then not all potential RFUs fit within. If it is too big, then this has a negative effect on the hardware utilization and the reconfiguration time.

Group 2 — Modular

The second group of RPs makes use of multiple reconfigurable areas. Therefore multiple RFUs can be loaded at the same time. This approach can help to decrease the delay caused by a reconfiguration since it allows a predictive configuration of a RFU which is needed quite soon. For a given logic capacity of the reconfigurable fabric, the question arises whether to partition it into few rather big areas or into more rather small areas.

Group 3 — Overlapping

The third group uses multiple *overlapping* RFUs. This method is based on the observation that particular RFUs implement a quite similar functionality. Refining the granularity by splitting the RFUs into sub units, enables the usage of a sub unit by multiple RFUs. This can help decrease the reconfiguration time and to utilize the given logic capacity more efficiently, but it also leads to a much more complex communication structure.

In the following, five exemplary reconfigurable processor architectures are presented.

3.2.1 CoMPARE

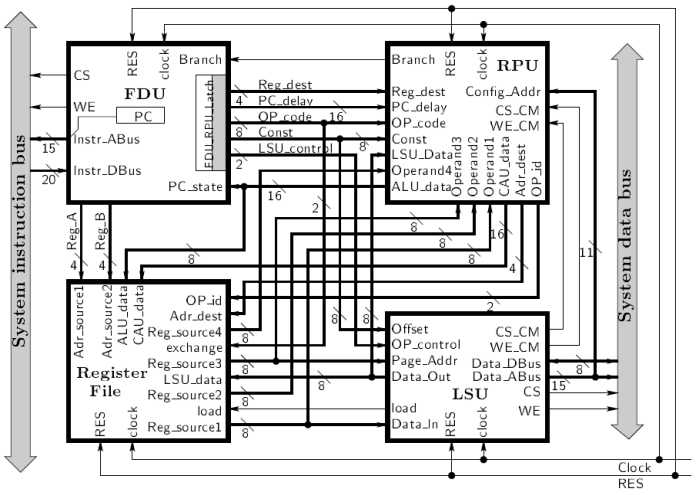


Figure 3.4: Block structure of CoMPARE [108]

CoMPARE (Common Minimal Processor Architecture) makes use of exactly one reconfigurable area (Group 1 — Monolithic). It uses an FPGA to replace a conventional ALU by a RFU (see figure 3.4). Its design process was mainly guided by the idea of simplicity and scalability. Therefore only 16 simple instructions have been implemented on a small RISC processor which consists of four basic units: the FDU (fetch and decode unit) which accesses the instruction memory to fetch the next instruction, the register file, the RPU which consists of a standard ALU and an application specific part, and the LSU (load and store unit) which combines the results from the hard-wired logic and from the reconfigurable logic and stores them into the register file. The whole system has been implemented on a Xilinx Virtex FPGA and achieved a maximum clock rate of 11 MHz. However, an average speedup of 2 compared to a solely static design could be achieved [108].

3.2.2 Chimaera

Chimaera is a high-performance co-processor based on a RFU. In contrast to CoMPARE, the RFU is not monolithic but is divided into sub units called RFUOPs (reconfigurable functional unit operations). Therefore Chimaera is a representative of Group 2 (Modular). Figure 3.5 illustrates its architecture. The reconfigurable array (RA) is a placeholder for the RFU respectively the RFUOPs. The execution control unit (ECU) decodes the incoming instruction stream and directs execution. It detects RFUOPs and controls their execution on the RA. If necessary, it notifies the configuration control and caching unit (CCCU) of currently unloaded configurations. The CCCU is responsible for loading and caching configuration data. Although up to 9 RFUOPs can be loaded into the RA at a

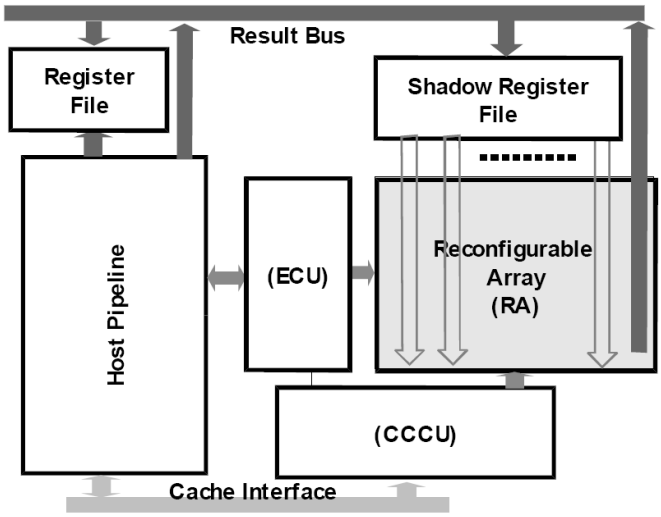


Figure 3.5: Overview over the Chimaera architecture [109]

time, only one RFUOP can produce output data at a time (the RFU has 9 inputs, but only 1 output). Nevertheless, the RFUOPs can be executed in parallel and exchange data among each other. Furthermore, based on the set of already prepared RFUOPs, delays caused by on-demand reconfigurations can be reduced. This way, an average speedup of 1.3 could be achieved. Thereby the speedup strongly depends on the executed application. While the performance of some applications became worse, the measured peak speedup was 2.4 [109].

3.2.3 MOLEN

The two main components of the MOLEN polymorphic processor are a general purpose processor (GPP) called Core Processor and the Reconfigurable Co-Processor (RP) — see figure 3.6. The ARBITER performs a partial decoding on the instructions in order to determine where they should be executed. General instructions are handed to the GPP. Application-specific instructions are redirected to the RP. Data transfers from and to the main memory are handled by the Data Load/Store unit. The Data Memory MUX/DEMUX establishes a communication between the Load/Store unit and either the GPP or the RP. The Exchange Registers are used for direct communication between RP and GPP. The reconfigurable processor consists of the reconfigurable microcode unit (RMU) and the custom computing unit (CCU). An operation, executed by the RP, is divided into two phases: *set* and *execute*. The *set* phase is responsible for reconfiguring the CCU for the operation. This is realized by the RMU. In the *execute* phase, the actual execution of the operations is performed in the CCU. To reach a significant speedup, special *set* commands are used. These *set* commands can be included in the source code with a sufficient distance to the corresponding *execute* commands. This way, the reconfiguration process

is concluded before the CCU is called for execution. For benchmarking, a MPEG-2 encoder and a MPEG-2 decoder have been implemented. The encoder speedup was about 2.9 while the speedup of the decoder was about 1.6 [110].

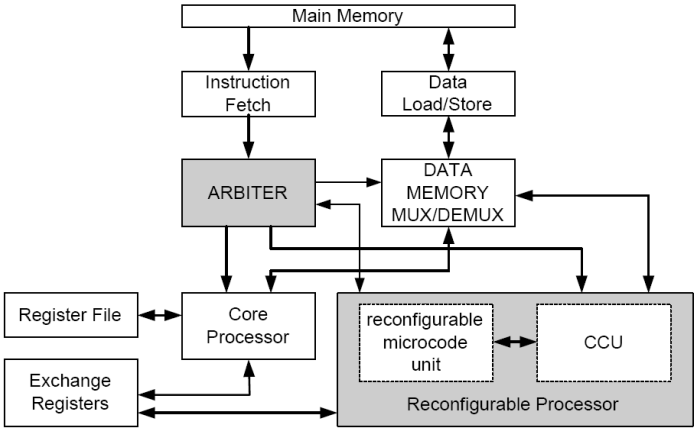


Figure 3.6: Overview over the MOLEN architecture [110]

3.2.4 RISPP

RISPP stands for Rotating Instruction Set Processing Platform. The term "Rotation" denotes the enhancement of traditional ASIPs by DPR. Thereby the RISPP implements the sub unit concept (Group 3 — Overlapping). The sub units are called Atoms. A special focus is put on the trade-off between area consumption and execution time. A given algorithm can be accelerated using pipelining and parallelization. However, these methods usually require additional resources. For example, the two commands "$a = 19 \cdot b$" and "$c = 7 \cdot d$" could either be executed sequentially (using only one multiplier) or in parallel (using two multipliers). The RISPP platform calculates an optimal trade-off based on Amdahl's law [111]. If a special instruction (SI) is called very rarely, its implementation can be done in an area-saving way (using few or even only one Atom). In contrast, if a SI is called quite frequently, its implementation should be as time-efficient as possible (using many Atoms). In [112] this concept has been demonstrated with the help of an H.264 video encoder whose major functional blocks are Motion Estimation (ME), Motion Compensation (MC), Transform and Quantization (TQ), and Loop Filtering (LF). The size and the execution time of these components differ significantly. While the MC implementation requires 199,812 Gate Equivalents (GE), it is only executed 17% of the time. In contrast, the ME implementation only requires 27,438 GE, but it is executed 70% of the time. To determine the best trade-off, the RISPP provides a theshold called Multiplication factor (α) which determines the size of the reconfigurable area. If $\alpha = 1$, the biggest functional block exactly fits the reconfigurable area. If $\alpha < 1$, the biggest functional block can

be supported only partially. This means that it is devided into two (or even more) sub modules which are loaded sequentially to the FPGA. Using this sub-partitioning or an $\alpha > 1$, the remaining space of the reconfigurable area can be used for prefetching Atoms required fairly soon. Based on these technologies, a speedup of 26.6 could be achieved compared to a General Purpose Processor. Futhermore, the speedup was 1.24 compared to other Reconfigurable Processors specialized for an H.264 video encoder [113].

3.2.5 WARP

The WARP processor architecture focuses less on the design of the FPGA and more on an automated run-time generation of RFUs. Unlike other approaches, no special compiler is used. Instead, a WARP processor operates on a standard binary. It dynamically detects the binary's critical regions, reimplements those regions as RFUs, and replaces each critical software region by a call of the corresponding RFU. As shown in figure 3.7, a WARP

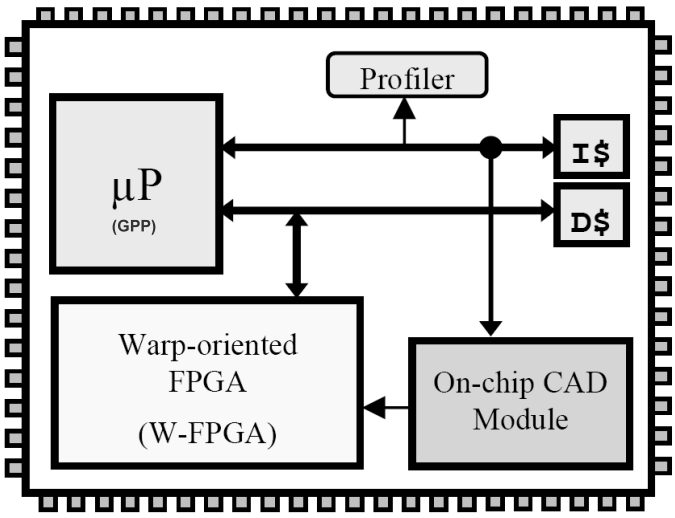


Figure 3.7: Overview over the WARP architecture [114]

processor consists of a General Purpose Processor (GPP), an on-chip profiler (responsible for the detection of the critical regions), an on-chip computer-aided design (CAD) module and an FPGA. The CAD is the most outstanding part of the WARP architecture since it realizes the translation from software binaries to hardware including synthesis, placement and routing at run-time [115]. The efficient realization of this approach is a very ambitious aim since even the automated generation of well designed hardware from software at compile-time is a problem still not sufficiently solved (see chapter 3.3). However, in [116] an average speedup of 6.3 compared to a simple GPP implementation has been measured. Furthermore, in [114] an average power reduction of 74% has been achieved, based on the WARP processor architecture.

3.3 From Software to Hardware

Nowadays, high performance computing (HPC) changes significantly. In former times HPC programmers could rely on new processors to get performance improvements without changing the existing code. This is no longer true since today's processor generations are characterized more by the number of their cores than by the speed of an individual core. Thus, formerly single-threaded code has to be transformed into multithreaded code which is using the given number of processor cores as efficiently as possible. Furthermore, today the costs of a computing center performing HPC are mainly determined by its energy requirements. Therefore FPGA based co-processors attracted notice. The FPGA's intrinsic parallelism makes it possible to achieve high performance with a much lower energy consumption compared to CPUs. Thereby, a very important aspect is the reconfigurability of FPGAs which makes it possible to adapt the functionality of the co-processor to the current needs (as shown in chapter 3.2).

A consequence of this new trend is that many research groups and IT companies are porting formerly single-threaded software to concurrent hardware. Thereby a huge hindrance is the fact that programming FPGAs has to be done on register transfer level (RTL), which requires skills and techniques outside the expertise of most HPC developers, who typically use languages such as C++ or Java. Therefore, several research groups are looking forward to create a compiler which automatically translates ordinary C, C++ or Java code to VHDL or Verilog [117].

The topic of translating software to hardware can be seen in two different ways: *top-down* and *bottom-up*. The *top-down* view follows the reasoning above, coming from a top-level description of the algorithm (e.g. in C++) and trying to generate efficient hardware with as few changes as possible to the original code. In contrast, the *bottom-up* approach results from the point of view of hardware developers who want to improve their productivity. Here, every language construct that is able to improve the performance of the generated hardware is welcome, even if it turns a high level language like C into something common software developers are not familiar with. Typical examples for such additional language constructs are the *par* and the *delay* statement in Handel-C [118], which are used to explicitly describe parallelism and clock cycle delays.

Although the two approaches are quite different, both have to face the same principal problems. The biggest challenges are concurrency and timing [119]. In typical HDLs like VHDL and Verilog, *processes*, which run in parallel and whose timing is precisely determined (depending on a given clock), are used. In typical software languages like C or Java, such constructs have no real counterpart. Even though modern software languages make use of parallelism (in the form of multithreading), they do not provide any construct to define an exact chronological correlation between the threads. It is the scheduler provided by the underlying operating system that decides which thread is executed at which time. So, to translate software to hardware, the compiler has to analyze the software code and to find sequential-written software constructs which can be executed in parallel. For this, methods like data path analysis, pipelining, Loop Unrolling and Loop

Shifting are used. Following the *top-down* approach, the compiler has to do this by itself without any help from the programmer. In contrast, *bottom-up* oriented languages provide additional constructs which shall be used to help the compiler make the right decisions.

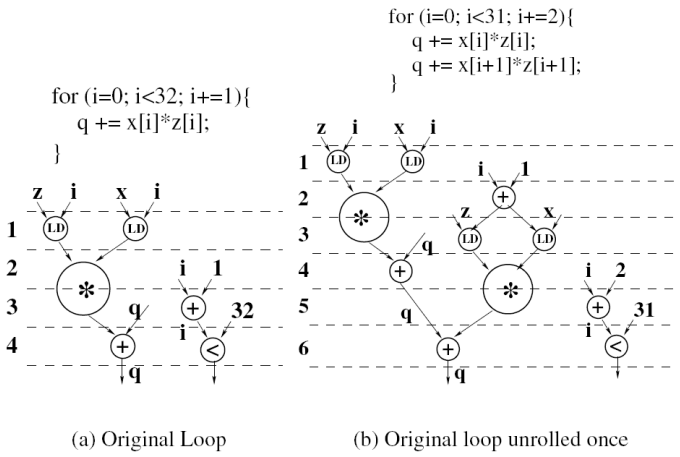


Figure 3.8: Loop Unrolling Example [120]

3.3.1 Code Conversion

The most common method to enhance the performance of automatically generated hardware is Loop Unrolling. At this, the body of the loop is duplicated multiple times to enable additional parallelizations between consecutive loop iterations. Figure 3.8 illustrates this method. The original loop requires approximately $32 \cdot 4 = 128$ clock cycles (ignoring the initialization part). In contrast, the loop unrolled once requires just $16 \cdot 6 = 96$ clock cycles. This is caused by the fact that the increment of the iterator *i* and the loading of *x[i+1]* and *z[i+1]* belonging to the second command (*q+=x[i+1]*z[i+1]*) can be done in parallel to the multiplication belonging to the first command (*q+=x[i]*z[i]*). Without Loop Unrolling, the compiler would not be able to detect this chance of parallelization.

However, Loop Unrolling also comes with two drawbacks. First, the number of performed iterations must be known at compile time. Otherwise Loop Unrolling is not possible. Secondly, a maximal unrolling does not always lead to the best performance since Loop Unrolling increases the delay of the critical path (and thus may decrease the maximum possible clock rate). Furthermore, it increases the size of the data flow controller, which can also lead to a lower clock rate. Thus, in [120] the authors present an algorithm to determine the optimal unrolling factor without the need for an exhaustive synthesis. Their estimated delays for several unroll factors differ from the actual delay by just about 7%. The estimation process needed less than 5 minutes — while the synthesis times exceeded 6 hours for some code examples.

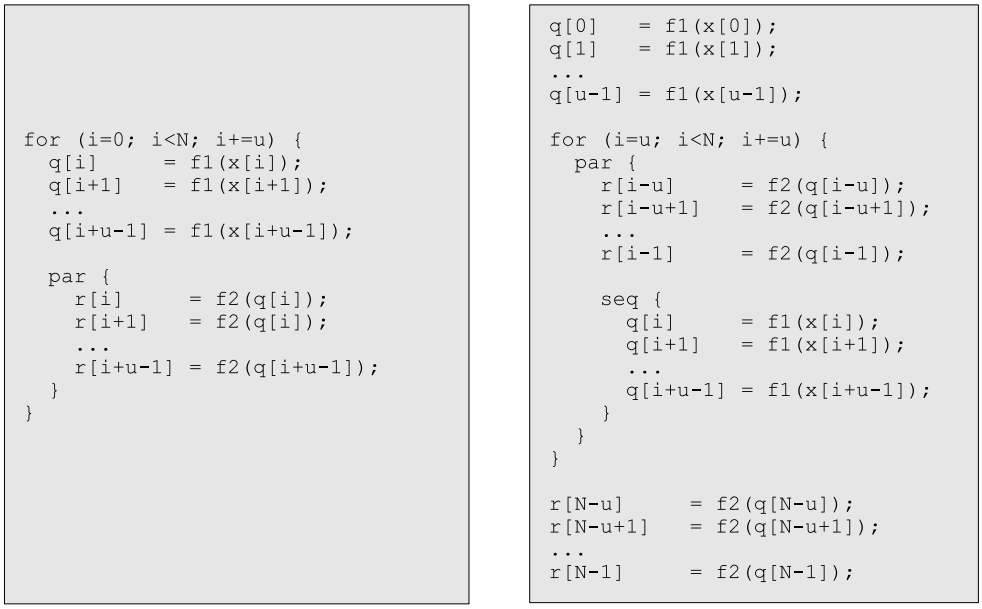


Figure 3.9: Loop Shifting Example [121]

In [121] a further optimization method regarding loops is presented: Loop Shifting — which means moving a function from the beginning of the loop body to its end for eliminating data dependencies. Figure 3.9 illustrates such a Loop Shifting based on the MOLEN machine organization (see chapter 3.2.3). Function *f1* is executed in hardware by the CCU while function *f2* is executed in software on the GPP. Without Loop Shifting, the *u* instances of *f1* can be executed in parallel, but due to the data dependencies between *f1[i]* and *f2[i]*, the execution of *f2* cannot start before *f1* has finished. Using Loop Shifting, *f1[i]* is always executed one iteration before *f2[i]*. Hence, the GPP and the CCU can calculate in parallel (namely *f1[i+1]* and *f2[i]*). To get experimental results, the DCT[6] of the MPEG-2 algorithm has been implemented. Based on Loop Shifting, a speedup of 19.65 using an unrolling factor of 8 could be achieved, while without Loop Shifting even an unrolling factor of 48 only led to a speedup of 17.71.

Loop Unrolling does not only come with advantages but also leads to a significantly higher resource consumption. Therefore the synthesis tool often has to find a trade-off between performance and area. At this, a good control value is the already presented unrolling factor. Beyond that, for very big components (such as multipliers) it can make sense to additionally insert Data Path Merging, which means that a particular component is used in two (or even more) independent data paths at different points in time [123] (see figure 3.10). To keep the independence, multiplexers are added. In [124] the authors could produce circuits which are 85.33% smaller than those synthesized by integer linear programming approaches which do not use Datapath Merging.

[6]Discrete Cosine Transformation

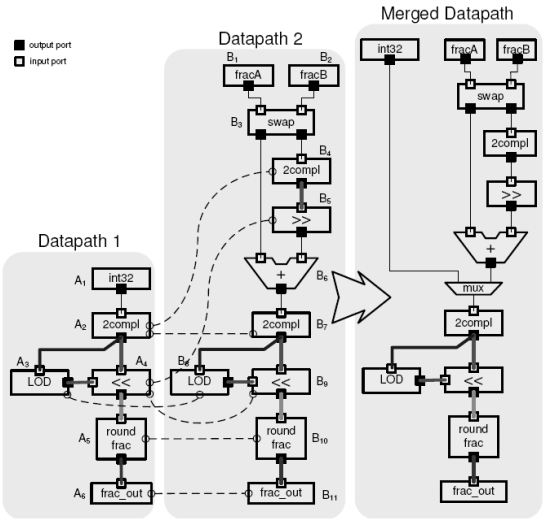


Figure 3.10: Datapath Merging example [122]

Further problems regarding the translation from C-like languages to hardware are recursions (which usually require a stack) and pointer-arithmetic. A stack and free pointer-arithmetic both rely on a memory implementation, which normally represents a bottle neck. Therefore SystemC [125], Handel-C [118], HardwareC [126], SpecC [127] and other nameable C-like languages do not support recursions and pointer-arithmetic at all. Regarding the *top-down* approach, this is a problem since it makes C-code which uses pointer-arithmetic or recursions unsynthesizable. Due to this, in [128] the authors present a method for mapping recursive functions to reconfigurable hardware without the use of a stack — instead the recursion is unrolled using DPR. The compiler C2Verilog [129] supports pointers, recursion and even dynamic memory allocation, which makes it very powerful. Unfortunately, all these constructs have a negative impact on the performance of the generated hardware.

3.3.2 ROCCC

In [130] the authors present ROCCC (Riverside Optimizing Configurable Computing Compiler), a compiler designed to generate VHDL from C source code. In principle it follows the *top-down* approach but it does not support pointers which cannot be statically unaliased and recursion. The ROCCC compiler makes use of full Loop Unrolling, loop-mining and loop fusion. Furthermore, function calls are either inlined or whenever possible made into a lookup table. To translate C to VHDL, the data flow is analyzed and parted into soft nodes and hard nodes of a control flow graph (CFG). Each soft node contains commands that can be executed in parallel. To enable pipelining, branches are synchronized via additional delay elements which are placed in so called hard nodes (see figure 3.11). Afterwards, ROCCC generates one VHDL component for each node.

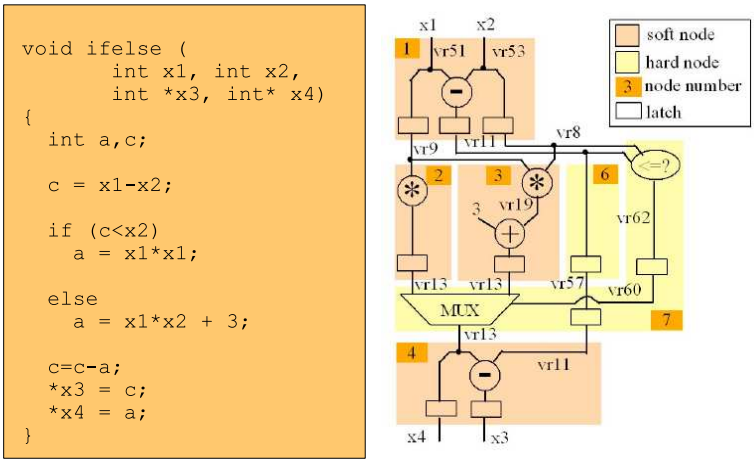


Figure 3.11: ROCCC Example [130] (the pointers *x3 and *x4 are needed to support the usage of two return values — they can be statically unaliased and thus are supported by ROCCC)

To evaluate ROCCC, Xilinx IP cores (coming with Xilinx ISE 5.1) have been compared to the generated hardware. The results show that the speed of the generated hardware is within 10% while the consumed area is larger by a factor of 2 to 3. For some examples (e.g. an 8-bit unsigned divider or a 24-bit square root calculator), the generated hardware is even faster than the corresponding Xilinx IP core.

3.3.3 CHiMPS

CHiMPS (Compiling High-Level Languages into Massively Pipelined Systems) is a compiler that inputs generic ANSI-C code and targets a hybrid CPU-FPGA architecture (comparable to MOLEN). For this, it automatically generates VHDL code which represents a customized, parallel FPGA accelerator. It has been developed to provide HPC developers an easy and familiar way to accelerating their applications and is therefore one of the strictest representatives of the *top-down* approach. Each ANSI-C instruction is first translated into an assembly-like language called CTL (CHiMPS Target Language) which consists of 42 instruction blocks. Most of these blocks are comparable to usual assembler instructions. Additional instruction blocks have been introduced to handle dataflow-specific commands like *if/else*, *for*-loops and *break*. Figure 3.12 illustrates the conversion from C code to CTL and the translation from CTL to VHDL blocks. Please note the insertion of a FIFO to synchronize the data paths and thus to enable pipelining.

A unique feature of CHiMPS is its memory structure. Instead of one monolithic cache many small individual caches are created. Thereby CHiMPS does not waste resources trying to keep these caches coherent, but only instantiates a local cache if the corresponding memory area is used by just one single function. Thus, the data inside these local caches is solely used by one hardware component and therefore does not need to be kept coherent. CHiMPS creates a separate cache for any unique range of memory. To help

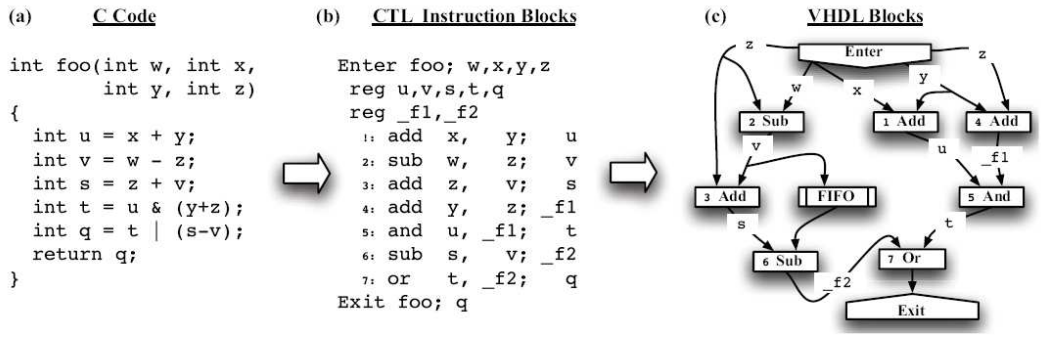


Figure 3.12: CHiMPS Example [117]

the compiler recognize such unique ranges, the keyword *restrict* can be used. It has the same meaning as in ANSI-C and is therefore no real amendment to the ANSI-C standard. Nevertheless, it has a much greater influence on the performance regarding synthesis than regarding a usual GPP compiler. In [117] CHiMPS has been used to implement five benchmarks which are typical for HPC: *Black-Scholes*, *Smith-Waterman*, *immul*, *sobol* and *swm*. The code of these benchmarks was modified in two ways. First, the keyword *restrict* has been added, wherever possible. Second, the functions which should be executed on the FPGA have been selected manually (using a *#pragma* statement). With only these minor source code modifications a speedup from 2.8 to 36.9 has been achieved.

Although CHiMPS is a very strict representative of the *top-down* approach and in [117] the authors attach great importance to the fact that the compiler inputs usual ANSI-C, CHiMPS provides several *#pragma* statements to convey additional information to the compiler:
Cache parameters can be used to specify configuration parameters regarding the caches to optimize the resource consumption and the performance. Developers can specify cache size, line length, associativity, number of banks and cache type (read-only, write-only, read-write).
Separate memory spaces are used to direct CHiMPS to use embedded SRAM memories (e.g. BRAMs) instead of external system memory.
Implementation style forces CHiMPS to use Datapath Merging. Furthermore it can be used to specify sections of code which should be implemented as software on a small embedded processor (e.g. Xilinx MicroBlaze).
Loop unrolling is used to engage and control Loop Unrolling.
Manual bit-width specification trims the size of the resulting bit-vectors.
Using these additional *#pragma* statements a speedup of 119.7 instead of 17.6 without the pragmas has been achieved on *Black-Scholes*.

3.3.4 Handel-C

Handel-C is one of the best-known representatives of the *bottom-up* approach. It uses much of the syntax of conventional ANSI-C, but introduces a powerful set of commands to manually control concurrency and timing. The Handel-C compiler does not search for or introduce concurrency by itself — instead the wished parallelism has to be expressed explicitly by the programmer. Handel-C clearly has not been designed to enable the translation of conventional ANSI-C to hardware, but to provide hardware developers a standardized and productive way to describe hardware. Thus, Handel-C is subject to some strict restrictions: floating-point types are not supported at all, recursion is not allowed, pointers which cannot be statically unaliased are not supported, functions can only be called in expression statements, parameter lists of variable length are not allowed, the *main()* function takes no arguments and has no return value, empty loops are not allowed, there are no unions, and dynamic memory allocation (via *malloc* or *free*) is not supported. However, the biggest difference to ANSI-C is the explicit expression of concurrency, based on the keyword *par*. Figure 3.13 illustrates a simple Handel-C program.

```
void main(void) {
  int 6 a;  int 6 b;
  int 7 s;  int 12 p;

  a = a + b;
  b = 2 * b;

  par {
     s = a + b;
     p = a * b;
  }
}
```

Figure 3.13: Handel-C Example

Please note that integers are not limited to a specific width. If a variable is defined, its size should be specified as well. In figure 3.13 *a* and *b* have a size of 6 bit, *s* has a size of 7 bit, and *p* has a size of 12 bit. This way developers can keep the resource consumption minimal. The command *a=a+b* is executed first. Afterwards the command *b=2*b* is executed. Third, the commands *s=a+b* and *p=a*b* are executed in parallel. Sequential and parallel blocks can be nested. A sequential block is tagged with *seq*.

Functions are not allowed to be called in parallel since they correspond to a shared piece of hardware. Thus, Datapath Merging can be forced by using functions. To avoid such a resource sharing, functions can be declared as *inline*.

For communication between branches of code executing in parallel, Handel-C provides a special form of interface: channels. They are declared using the keyword *chan*. A channel can be a simple register or even a fifo. Its depth is defined by the parameter *fifolength*. Reading from a channel is done with the *?*-operator. Writing to a channel can be done with the *!*-operator (see figure 3.14).

```
void main(void) {
  unsigned 8 a;
  unsigned 8 b;
  chan ch with { fifolength=2 };

  par {
     seq {
       a = a * 7;
       ch ! a;
     }
     seq {
       ch ? b;
       b = b + 3;
     }
  }
}
```

Figure 3.14: Channels in Handel-C

The commands inside the two *seq* blocks are executed sequentially, but the two *seq* blocks run in parallel to each other. First *a=a*7* and *ch?b* are executed. *ch?b* blocks since the channel *ch* is empty. Next, *a* is written to *ch*. After the channel has been filled, the command *ch?b* can take the data out of the channel and continue. At last, *b=b+3* is executed. Please note that if the channel is not empty and not full, it does not block and therefore does provide the possibility to be written and read in parallel. Please further note that this way channels can be used to synchronize independent branches (even if they belong to independent clock domains).

Handel-C does not only express concurrency explicitly but also has a very straightforward way to express timing. Each assignment takes exactly 1 clock cycle. Everything else takes place in the same clock cycle. Thus, the command *x=((a+b)*(c+d))* is executed in 1 clock cycle, while *y=a+b; z=c+d; x=y*z* takes 3 clock cycles. Although the Handel-C Language Reference Manual states that this way "even the most complex expression can be evaluated in a single clock cycle" [118], one should keep in mind that a too complex expression will not get timing closure during synthesis and therefore will lead to invalid hardware. Thus, it is the developer who has to decide which expressions lead to efficient hardware and which expressions exceed the capabilities of the target architecture. The compiler performs no timing optimizations at all. Beyond that, it is the developer who is in charge of avoiding combinational loops. Empty loops and empty branches are not allowed since the loop header and the branch control do not take any clock cycles (because they are no assignments). Due to this, Handel-C provides the *delay* statement, which does nothing else but take one clock cycle. It can be used to avoid empty loops or branches (see figure 3.15).

```
while (x!=3)
{
   if (y>z) {
     a++;
   } else {
     delay;   //Avoids a possible combinational loop
   }
}
```

Figure 3.15: Timing control via *delay* in Handel-C

3.3.5 SystemC

Over the last years, SystemC became the leading approach to system-level modeling. The fundamental motivation of SystemC is to provide a modeling framework in which high level functional models and detailed register-transfer level implementations can be described in one single language [125]. SystemC supports several models of computation like RTL modeling, behavioral modeling, timed functional modeling and untimed functional modeling. The idea is to give developers the opportunity to start the system design at a very high level, where even the separation between hardware and software is not defined, and to refine the design until it becomes synthesizable. In the early stages of a hardware design, this allows quick simulations and therefore quick decisions. Furthermore it allows the re-use of components described on a very high level and therefore enhances the design space exploration productivity. SystemC does not focus on the translation of conventional C-code to hardware but introduces special macros and objects which shall be used to describe the target design. Hence, SystemC is a representative of the *bottom-up* approach.

SystemC is an extension of C++ and uses macros and inheritance to provide the developer programming constructs to describe the design. To be able to use these constructs, every file should start with *#include "systemc"*. All functionality takes place in SystemC modules named *sc_module* (see figure 3.16). These modules are objects with a constructor and several functions which contain the actual functionality. To comply with the SystemC standard, the *SC_CTOR* macro has to be used when declaring or defining a constructor of a module. The name of the module class being constructed has to be passed to the macro as the argument. The most important task of the constructor is to specify the type of the functions. SystemC knows 3 different types: *SC_METHOD*, *SC_THREAD*, and *SC_CTHREAD* [131].

```
#include "systemc"

SC_MODULE(M)
{
  sc_in<bool> clk;
  sc_in<unsigned> a;
  sc_in<unsigned> b;

  SC_CTOR(M)
  {
    SC_METHOD(a_method);
    sensitive << a << b;

    SC_THREAD(a_thread);
    sensitive << a << clk.neg();

    SC_CTHREAD(a_cthread, clk.pos());
  }

  void a_method();
  void a_thread();
  void a_cthread();
}
```

Figure 3.16: Modules in SystemC

Functions of the type ***SC_METHOD*** are called method processes. They are comparable to processes in VHDL and therefore require a sensitivity list which is represented by the subsequent expression *sensitive « SIG1 « SIG2 « ... « SIGn*. A method process is called every time a variable of the sensitivity list changes its content. It is always executed from beginning to end.

Functions of the type ***SC_THREAD*** are called thread processes. Like method processes, they require a subsequent sensitivity list. In contrast to method processes, they are called immediately (independent from the sensitivity list) and they are called only once. Inside a thread process, *wait()* can be used to make the process wait until a signal of the sensitivity list changes. Developers can include an infinite loop containing *wait()* within such a method to prevent the process from terminating.

Functions of the type ***SC_CTHREAD*** are called clocked thread processes. In principle, they work exactly like thread processes, but they are only sensitive to a specific edge (rising or falling) of a single (clock) signal which is not specified in a subsequent sensitivity list but is delivered as the second argument: *SC_THREAD(methodname, SIG.pos())*.

In the following the focus is on the different models of computation (MOC).

RTL Modeling

SystemC's register-transfer level MOC is in many ways comparable to conventional HDLs like VHDL or Verilog. It only makes use of method processes which are a close match to VHDL's processes. All communication between processes occurs through signals. The processes themselves either represent sequential logic (in which case they are only sensitive to a clock edge) or combinatorial logic (in which case they are sensitive to all their inputs). Figure 3.17 opposes the implementation of a simple FSM in SystemC to the same implementation in VHDL.

SystemC

```
SC_MODULE (FSM) {
  sc_in<bool> CLOCK;
  sc_in<bool> RESET;
  sc_in<bool> NEXT;

  sc_out<bool> red;
  sc_out<bool> yellow;
  sc_out<bool> green;



  enum state {STOP, GO, YEL};
  sc_signal<state> curr_state, next_state;

  SC_CTOR(FSM) {
    SC_METHOD(ctrl_fsm_state);
    sensitive << CLOCK.pos();
    SC_METHOD(ctrl_fsm);
    sensitive << RESET << NEXT << curr_state;
  }

  void ctrl_fsm_state() {
    curr_state.write(next_state.read());
  }




  void ctrl_fsm() {
    control_state ns = curr_state;
    red.write(0);
    yellow.write(0);
    green.write(0);

    if (RESET.read() == 1) {
      ns = STOP;
    } else {
      switch (curr_state.read()) {
      case STOP:
        red.write(1);
        if (NEXT.read()==1) ns=GO;
        break;
      case GO:
        green.write(1);
        if (NEXT.read()==1) ns=YEL;
        break;
      case YEL:
        yellow.write(1);
        if (NEXT.read()==1) ns=STOP;
        break;
      }
    }

    next_state.write(ns);
  }
}
```

VHDL

```
entity FSM is
port (CLOCK    : in STD_LOGIC;
      RESET    : in STD_LOGIC;
      NEXT     : in STD_LOGIC;
      red      : out STD_LOGIC;
      yellow   : out STD_LOGIC;
      green    : out STD_LOGIC);
end FSM;

architecture imp of FSM is

  type state is (STOP, GO, YEL);
  signal curr_state, next_state: STATE_TYPE;

begin



 ctrl_fsm_state: process (CLOCK)
  begin
    if rising_edge(CLOCK) then
      curr_state <= next_state;
    end if;
  end process;



  ctrl_fsm: process (RESET, NEXT, curr_state)
  variable ns : state;
  begin
    ns := curr_state;
    red<='0'; yellow<='0'; green<='0';

    if (RESET=1) then
      ns := STOP;
    else
      case curr_state is
        when STOP =>
          red<='1';
          if (NEXT=1) then ns := GO;
          end if;
        when GO =>
          green<='1';
          if (NEXT=1) then ns := YEL;
          end if;
        when YEL =>
          yellow<='1';
          if (NEXT=1) then ns := STOP;
          end if;
      end case;
    end if;

    next_state <= ns;
  end process;
end imp;
```

Figure 3.17: Implementation of a FSM in SystemC's RTL MOC and in VHDL

Behavioral Modeling

Behavioral modeling is the level at which the primary concern is the order of input and output signals. In contrast, on RTL the designer must decide which states shall be used, how one state transitions to another, and which operations take place in which state. In a behavioral model the design is seen as a program flow. External behavior is defined by the sequencing of input and output events, not by clock cycles. Nevertheless, a clock can be used to keep the design synchronous. This behavioral clock should not be taken too literally, since it is more of a synchronizing strobe signal than the real clock used in the final hardware design. In particular, it is not possible to determine the exact timing (in clock cycles) of a design solely based on the behavioral description.
Figure 3.18 illustrates the implementation of Euclid's Algorithm as Behavioral SystemC.

```
SC_MODULE(euclid_gcd)
{
  sc_in_clk CLOCK;
  sc_in<unsigned> A, B;
  sc_out<unsigned> C;
  sc_out<bool> READY;


  SC_CTOR(euclid_gcd)
  {
    SC_CTHREAD(compute, CLOCK.pos());
  }

  void compute() {
    unsigned tmp_a = 0, tmp_b;

    while (true){

    C.write(tmp_a);
    READY.write(true);
    wait();

    tmp_a = A.read();
    tmp_b = B.read();
    READY.write(false);
    wait();

    while (tmp_b != 0){
       unsigned r = tmp_a;
       tmp_a = tmp_b;
       r = r % tmp_b;
       tmp_b = r;
    }
  }
}
```

Figure 3.18: Behavioral description of Euclid's Algorithm in SystemC [125]

The body of *compute()* is written just as it would be in a conventional software programming language. For hardware synthesis the model has to be refined. The 3 command blocks have to be assigned to states of a FSM. Furthermore the while-loop has to be represented by several FSM states (since it cannot be fully unrolled). At last, the line *r=r%tmp_b;* could be changed to *while(r>=tmp_b) r=r-tmp_b;* to save resources.

Functional Modeling

In the early stages of a design process, hardware developers are only interested in describing the functionality of a system's components. Details like timing, communication protocols or even the partitioning into hardware and software shall be encapsulated at this point. The aim is to create an executable specification which allows to verify the correctness of the system design as early as possible. Avoiding implementation details (like clocks) increases the re-usability of the specification and speeds up its simulation. Delays can be emulated using *wait(time, unit)*. Figure 3.19 illustrates the timed functional description of an adder with a generic type *T*. If the line *wait (200, SC_NS)* is omitted, the timed functional description becomes an untimed functional description which does not make any assumptions regarding execution time.

```
template <class T> SC_MODULE(Adder)
{
  sc_fifo_in<T> A, B;
  sc_fifo_out<T> C;

  SC_CTOR(Adder)
  {
    SC_THREAD(compute);
  }

  void compute() {
    while (true) {
      T data = A.read() + B. read();
      wait (200, SC_NS);
      C.write(data);
    }
  }
}
```

Figure 3.19: Timed functional description of an adder in SystemC [125]

Refinement and Synthesizability

SystemC provides different MOCs to give hardware developers a chance to choose the implementation granularity they need. Normally, creating a design in SystemC starts with an untimed functional model of the system's components. Here, the focus lies on the principle functionality of these components and not on their concrete implementation. However, after verifying the system design on such a high level, implementation details

attract more and more notice. Finally, a synthesizable description of the system is needed. This process of transitioning from an abstract model to a more detailed specification is called refinement.

Refinement is a very challenging task demanding a deep understanding of the targeted architectures, the implemented algorithms and the requirements of the application. Which Loop Unrolling factor shall be chosen? What is the ideal trade-off between performance, resource consumption and energy consumption? How does this influence further optimization methods such as Datapath Merging? The answers to these questions are as different as the possible target architectures (e.g. a wrist watch or a supercomputer cluster). Due to this, the refinement process is a task still a not fully automated. In [16] the Open SystemC Initiative defines an official synthesizable subset, which can be synthesized fully automatically. On page 50, it is stated that "SC_THREAD is non-synthesizable". SC_CTHREADs are synthesizable with reservations. SC_METHODs are synthesizable as long as they do not contain any wait statement. Furthermore, they must not contain any loop which is not unrollable. These restrictions limit SystemC's official synthesizable subset to a MOC on register-transfer level. As shown before (see figure 3.17), this MOC is quite similar to a hardware description in VHDL or Verilog. Due to this, a common final refinement step is to rewrite the components in VHDL or Verilog, where they can be implemented as efficiently as possible.

However, several industrial compilers focus on the extension of SystemC's synthesizable subset. Examples are Celoxica who provide the Agility Compiler which inputs behavioral SystemC models [132], and Mentor Graphics who developed Catapult-C, a compiler that produces RTL implementations from abstract specifications written in C++ or SystemC [133].

3.3.6 Conclusions

The high aim of high level synthesis (HLS) is to take ordinary C, C++ or Java code as input and to generate resource efficient hardware with high performance. In search of the best approach to do so, it became clear that the change from RTL to the algorithmic level is a more smooth transition than a sharp break. Handel-C provides hardware developers with a good way to improve their productivity. Nevertheless, due to the explicit expression of timing (in particular the *delay* statement) it is very close to the RTL. Other languages like Transmogrifer-C [134], Streams-C [135] or Dime-C [136] are much closer to the algorithmic level but do not support recursion, pointer-arithmetic and other software constructs which hinder the generation of efficient hardware. In contrast, C2Verilog focuses on the full translatability of ANSI-C code and does therefore support pointers, recursion, dynamic memory allocation and other unruly software constructs — which finally leads to a very bad performance. The same conclusion applies to Liquid Metal [137], which is a project to translate usual Java code (including all its features) to FPGA designs. Today's best compromise seems to be SystemC which unites several modules of computation in one language.

A very important but difficult to answer question is how far the refinement process is automated and how good the resulting generated hardware is. Looking at the promotion of several industrial products or at the achieved speed-up in several academic publications, one might consider the problem solved. Unfortunately, the outstanding results of the presented compilers often depend on the parallelization-aware implementation of the original software program [138]. For example, in the introduction of [117] the authors state that CHiMPS is a compiler which requires neither abandoning conventional programming abstractions nor changing the original code. Although this is fully true, the presented speed-up could only be reached by "minor source code modifications". These minor modifications include the addition of the keyword *restrict* and the insertion of *pragmas* to manually select the functions to be executed on the FPGA. While the first change really can be called minor (and maybe could even be automated), the second one requires an extensive design space exploration, which significantly stretches the meaning of the word "minor".

The conclusion is that many important break-throughs regarding HLS already took place but the problem of translating software to hardware turned out to be much more complex than initially expected. Nevertheless, today's HLS tools, languages and frameworks already help to significantly increase the productivity of hardware developers and to improve the performance of HPC applications, using FPGA based accelerators. So, HLS remains a future trend — which already has been proven to be very useful.

3.4 Frameworks combining DPR and HLS

In the following three frameworks are presented which are close to the approach of this PhD thesis since they address both DPR and HLS.

3.4.1 JHDL

JHDL (Just another Hardware Description Language) is an HDL that allows programmers to describe reconfigurable systems in a Java-like style. It has been developed at the Brigham Young University and comes with a simulator (which executes the sources in a JVM[7]) and a compiler which translates JHDL to EDIF[8]. According to [139] the goals of the JHDL project are:

1. Usage of Java without any language extensions
2. Independence from the target architecture
3. Language support of DPR
4. Usage of the same source code for both simulation and compilation

JHDL uses the Java features encapsulation and inheritance to provide the developer programming constructs to describe the design. Thus, the JHDL sources are executable with any standard Java 1.1 distribution. Nevertheless one should not mix up JHDL and ordinary Java, since in JHDL one has to use special classes which derive from the given class *Logic* and which contain special methods to write synthesizable code. Due to this, JHDL is a representative of the *bottom-up* approach presented in chapter 3.3. Figure 3.20 shows a full-adder implementation taken from the JHDL online tutorial [140].

First of all, to be able to use JHDL classes, every file should start with *import byucc.jhdl.base.*;* and *import byucc.jhdl.Logic.*;*. All classes describing hardware have to derive from the class *Logic*. Input ports and output ports of the class have to be declared in the static variable *cell_interface*. The line *in ("a", 1)* describes an input port named "a" which is 1 bit wide. Furthermore, the input and output ports have to be connected to *Wires* that were passed to the constructor via *connect*. Now, one can use methods like *or*, *and* or *xor* to represent the logical function of the class. Of course, more complex structures like for-loops can also be used.

[7] Java Virtual Machine

[8] Electronic Design Interchange Format — a vendor neutral format representing netlists and schematics (the result of synthesis)

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

public class FullAdder extends Logic {
   public static CellInterface[] cell_interface = {
       in ("a", 1), in ("b", 1), in ("cin", 1), out ("sum", 1), out ("cout", 1)
   };

   public FullAdder{Node parent, Wire a, Wire b, Wire cin, Wire sum, Wire cout} {
      super(parent);
      connect("a", a); connect("b", b); connect("cin", cin);
      connect("sum", sum); connect("cout", cout);

      or_o (and(a,b), and(a,cin), and(b,cin), cout);
      xor_o (a,b,cin,sum);
   }
}
```

Figure 3.20: JHDL implementation of a full-adder [140]

DPR in JHDL

In [139] the authors clearly state that JHDL shall "support run-time and partial reconfiguration". To do so, the class *PRSocket* has been introduced, which makes it possible to describe the runtime exchange of one class by another. The list of configurations is encapsulated in an object named *ConfigGroup*. Figure 3.21 illustrates the usage of *ConfigGroup* to describe the alternate loading of 3 different classes.

```
class myConfigGroup extends ConfigGroup {
  Node getNewCircuit(int id, PRSocket sock){
    switch (id){
     case 1:
       return new Ciruit1(...);
     case 2:
       return new Ciruit2(...);
     case 3:
       return new Ciruit3(...);
    }
  }
}
```

Figure 3.21: JHDL description of DPR [139]

Problems

There are several serious problems regarding JHDL. First of all it is not really understandable why quite awkward constructs like the static array of type *CellInterface[]* and the corresponding *connect* calls have to be used. Encapsulation and inheritance should make it possible to hide such linking details from the developer.

Secondly, the extensive usage of constructs like *or* and *xor* guides the developer to design on the logic level instead of designing on the register transfer level — which is a step backwards. According to the current work of the Brigham Young University [141] the implementation focus seems to lie more on the optimization of EDIF netlists than on the creation of a high level synthesis language.

Finally and most importantly, there is no DPR support at all, regarding synthesis. The JHDL compiler is open source and can therefore be downloaded and analyzed. A closer look at the compiler sources reveals that the synthesis toolflow is not able to describe runtime reconfiguration. The partitioning of the chip and the generation of a reconfiguration controller are not implemented at all. Furthermore, the compiler lacks a DPR aware toolflow such as presented in figure 2.13. In conclusion, the claim that JHDL "has been designed to directly support run-time reconfiguration, both partial and global" [139] is not comprehensible. Of course, JHDL can be used to create EDIFs which serve as dynamic modules in a DPR system, but this is also true for components described in ordinary VHDL or Verilog.

One has thus to conclude that JHDL comes with many good aims and ideas to combine DPR and HLS but fails to present a compiler that keeps up with the high goals.

3.4.2 OSSS+R

OSSS+R is a SystemC based software library. It has been designed to support both simulation and synthesis of runtime reconfigurable hardware. OSSS (Oldenburg System Synthesis Subset) is a design flow for simulation and synthesis of a given SystemC subset [142]. OSSS+R augments this design flow with the support for DPR ("+R" stands for "plus runtime-reconfiguration"). It has been developed within the ANDRES[9] project [143].

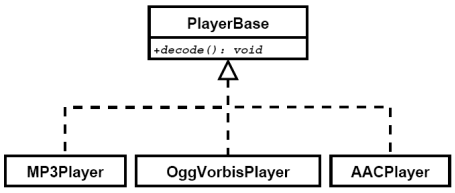


Figure 3.22: OSSS+R polymorphic class hierarchy [144]

The essential idea of OSSS is to enhance the usage of object-oriented C++ features such as classes, objects and inheritance more than it is typical for native SystemC. OSSS+R extends this approach by adding polymorphism to describe partial reconfigurability (see figure 3.22). The basic idea is to assign class A to a generic pointer — and later to assign class B to the same pointer. The change from A to B denotes the change from partial configuration a to partial configuration b, which is realized using DPR. Thereby class A serves as the source of configuration a and class B serves as the source of configuration b.

[9] Analysis and Design of run-time Reconfigurable, heterogeneous Systems

The corresponding reconfigurable area which is used by *a* and *b* is called Recon-Object. Figure 3.23 illustrates the description of the Recon-Object *decoder*.

```
SC_MODULE(DUT)
 {
 ...
 osss::osss_recon<PlayerBase<Memory>> decoder;
 ...
 SC_CTOR(DUT):decoder("decoder_module")
 {
   decoder.clock_port(clock);
   decoder.reset_port(reset);

   SC_CTHREAD(core, clock);
   reset_signal_is(reset, true);
   osss_uses(decoder);
 }

 void core () {
   while (true) {
     decoder = MP3Player<Memory>();
     osss_call(decoder)->decode(input_buffer, output_buffer);
     decoder = OggVorbisPlayer<Memory>();
     osss_call(decoder)->decode(input_buffer, output_buffer);
     decoder = AACPlayer<Memory>();
     osss_call(decoder)->decode(input_buffer, output_buffer);
   }
 }
};
```

Figure 3.23: OSSS+R description of runtime-reconfiguration [144]

The line *osss::osss_recon<PlayerBase<Memory» decoder* declares *decoder* to be a Recon-Object. Since every Recon-Object needs a clock and a reset signal, the corresponding ports of the module *DUT* are bound to *decoder*. The method *core()* is a SC_CTHREAD. Every process that is accessing a Recon-Object must be registered to it via *osss_uses*. The process *core* consecutively assigns an *MP3Player*, an *OggVorbisPlayer* and an *AACPlayer* object to *decoder*. Every *decoder=...* denotes a dynamic partial reconfiguration.

In contrast to JHDL, OSSS+R comes with a completely working synthesis flow which fully supports DPR. The OSSS+R models are synthesized to hardware using the synthesis tool Fossy [145]. The partial reconfiguration has been realized based on Xilinx' PREA tools (see chapter 2.5). In [146] an evaluation of OSSS+R is presented, based on the cryptographic algorithms *Triple DES*, *Blowfish* and *AES*. Using DPR, these benchmarks have been loaded alternatively to a Virtex-4 LX25 residing on a Xilinx ML401 development board. The reconfiguration controller has been designed manually, using the FPGA's ICAP with a maximum bandwidth of roughly 600 Mb/sec. The partial bitstreams had a size of 159 kB to 203 kB. The dynamic partial reconfiguration took 199 to 254 microseconds. The authors of [146] conclude that "the overall implementation cost is acceptable given the potential save of FPGA area through the use of DPR".

However, this most up-to-date evaluation of OSSS+R comes with a huge drawback: the hardware modules used for DPR are not synthesized using the OSSS+R toolflow, but "have been injected manually as HDL IP blocks" [146]. Strictly speaking, this reduces the results to an evaluation of the manually implemented reconfiguration controller — thus, unfortunately they say nothing about the efficiency of the OSSS+R compiler.

3.4.3 MORPHEUS

MORPHEUS stands for Multi-purpOse dynamically Reconfigurable Platform for intensive HEterogeneoUS processing. It has been developed by a consortium consisting of 18 partners throughout Europe [147]. MORPHEUS is an integrated design toolset addressing both HLS and DPR. Thereby, MORPHEUS focuses on three different target architectures:

- XPP-III [148] — a coarse grain reconfigurable array used for algorithms with mostly deterministic control and dataflow
- PiCoGA (Pipelined, Configurable Gate-Array) [149] — a medium-grained reconfigurable array consisting of 4-bit ALUs
- FlexEOS [150] — a dynamically reconfigurable FPGA

All three processing units are combined on the MORPHEUS chip, which furthermore contains an NoC to let the three units communicate with each other, and an ARM 926EJ-S embedded RISC processor which handles control, synchronization, and reconfiguration. Figure 3.24 illustrates the MORPHEUS architecture.

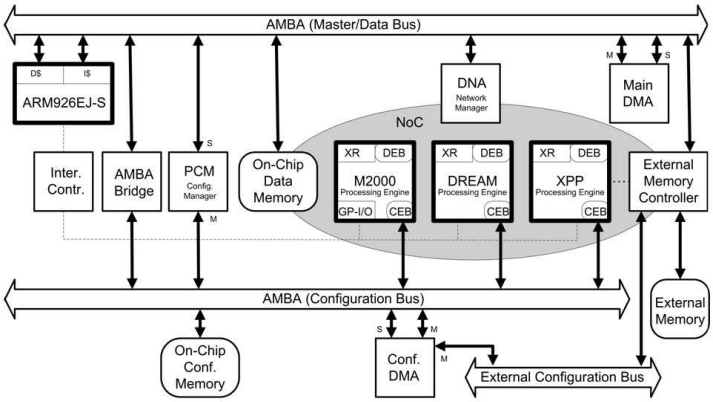


Figure 3.24: MORPHEUS architecture[147]

Originally, each of the three target architectures is described in its own language. For example, programs for the PiCoGA are written in C or GriffyC, while the design for the FPGA is described in VHDL or Verilog. Furthermore, each target architecture comes with its own compiler. The aim of the MORPHEUS project is to hide these differences from

the developer and to make it possible to describe the MORPHEUS chip with one single language: C code that is enriched with pragmas which allow the developer to determine on which architecture the functions shall be executed [151]. For the realization of the software/hardware interaction, the MOLEN compiler is used (see chapter 3.2.3). Functions (that shall be accelerated) are loaded to one of the reconfigurable target architectures on demand. Figure 3.25 shows a C source code example.

```
#pragma MOLEN_FUNCTION 140
void the_acc_func (unsigned int *in, unsigned int *out){
    return;
}

unsigned int test_picoga(){
  the_acc_func(data_in, data_out);
  data_reorg();
  data_compare();
  return 1;
}
```

Figure 3.25: MORPHEUS: C code with pragmas [152]

In [153] the MORPHEUS project has been evaluated. The MOPHEUS chip has successfully been designed and manufactured. Furthermore, the compiler has been proven to work correctly. For evaluation, a video streaming algorithm has been implemented. MORPHEUS could reach 90 GOPS (Giga Operations Per Second) with an energy consumption of 20 GOPS/W. For comparison: an ARM9 processor reaches 0.35 GOPS with 1 GOPS/W, the Phillips Xetal II reaches 107 GOPS with 170 GOPS/W.

3.5 Summary

Already in the 1980s, partial reconfiguration was supported by Xilinx's FPGAs [154]. However, it had the reputation of being a cumbersome technology and of lacking real world applications. Therefore, industry always hesitated to use it in production systems. Nevertheless, academic research groups focused on dynamic partial reconfiguration and the development of better tools and/or the improvement of the performance of DPR. Setups like the Erlangen Slot Machine prove that it is possible to utilize such a profoundly dynamic element in combination with a well standardized system architecture. Technologies like bitstream compression and overclocking led to a significant decrease of the reconfiguration times. Finally, cooperations with the automotive industry proved that DPR is indeed able to improve real world applications.

Due to this, today the reputation of partial reconfiguration is undergoing a radical change. This is underlined by the official support of DPR by Xilinx's ISE 12 (no additional patch is necessary any longer) [155], the support of DPR on Xilinx's low-cost Spartan-6 series [24] and the announcement of Altera to support DPR[156] (while in the last years Altera was one of the most prominent doubters regarding DPR). Furthermore Xilinx announced the production of an "Extensible Processing Platform" which pairs an ARM processor with programmable elements (FPGAs) [157]. If and how partial reconfiguration will be used in this setup is not clear today, but the design presented by Xilinx is very similar to the MOLEN polymorphic processor and it would be astonishing if DPR would not be utilized at all. In conclusion, nowadays dynamic partial reconfiguration is changing from a purely academic topic to a standard solution used in industry.

Having said that, it is important to underline that there is still much work to be done. Especially the combination of DPR and HLS is still in its infancy. This is caused by two facts:

First, HLS itself is still an unsolved problem. Although many milestones have been reached, the final break-through is still pending. The reason is that HLS is located in the gray area between syntax and semantics: it highly depends on the surrounding conditions which implementation is the best. However, the high acceptance of high level languages such as SystemC and the continuously improving performance of HLS tools like Catapult-C suggest that HLS is the programming paradigm of the future. Thus, trying to combine DPR and HLS seems to be worth the trouble.

Second, all language extensions which focus on the combination of DPR and HLS treat runtime-reconfiguration as a foreign element which has to be introduced using quite unusual constructs. In JHDL, it is a class deriving from *ConfigGroup* which has to make use of the class *PRSocket* and a fixed *switch* construct. In OSSS+R, it is the class *osss_recon* and the method *osss_uses* which have to be used to denote DPR. All these approaches lack an internal language support of DPR which only makes use of well known language constructs. In other words: DPR is introduced subsequently instead of being a natural part of the language.

4 The Approach

In chapter 3 several approaches regarding DPR[1], HLS[2], and the combination of both have been presented. It has been shown that in the recent years, many notable research groups focused on DPR. As a consequence, partial reconfiguration turned from a fancy academic topic to a serious technology ready to be used by industry. This also applies to HLS. For example, SystemC evolved from a small open-source initiative to the standard language used for hardware design on system level and for the corresponding refinement process. However, the combination of both is only at the beginning yet. A few projects (like JHDL and OSSS+R) already focused on it but all of them introduced DPR subsequently into an existing language, using quite unusual constructs.

The aim of this thesis is to describe DPR, solely using language constructs which are already well-known to software-developers. In other words: DPR shall be described, following the *top-down* approach presented in chapter 3.3. The resulting questions are:

- Which software language constructs already exist that can be used to express dynamic partial reconfiguration?

- Which language should be used by the Framework?

4.1 Object-Oriented Hardware Description

Before the above-mentioned questions shall be answered, the focus is on a second aspect regarding HLS, which has a strong influence on the choice of the language: programming paradigms. The choice of a paradigm determines the concepts and abstractions used to represent the elements of a program. The three most important programming paradigms are: functional (e.g. Haskell, Scala, Makefile), procedural (e.g. COBOL, C), and object-oriented (e.g. C++, Java). As a first step regarding the development of the Framework, the typical structure and behavior of reconfigurable hardware has been analyzed. Thereby it turned out that the best way to describe such hardware is to make use of the object-oriented paradigm combined with multi-threading. In the following part this conclusion shall be clarified.

[1] Dynamic Partial Reconfiguration
[2] High Level Synthesis

Conventional HDLs

Conventional hardware description languages like VHDL or Verilog already describe hardware in a way which is close to the object-oriented paradigm. Although data abstraction, inheritance, and polymorphism cannot be found in these HDLs, the basic principles encapsulation, modularity and instantiation are fully supported. Due to this, VHDL components can be compared to classes. The corresponding *in*- and *out*-ports are comparable to *set*- and *get*-methods in object-oriented languages. *Signals* defined for a component are encapsulated and only visible inside this entity. Well-designed VHDL code is not realized using one single entity but via a structural separation into multiple subcomponents — in short: using modularization. Finally, components have to be instantiated (once or even multiple times) to be able to be used.

The Target Architecture

The main elements of FPGAs are look-up tables (LUTs), multiplexers and flip-flops (see chapter 2.1). These elements can be seen as small objects with a simple state (if any) and provide the surrounding system with simple methods. During synthesis these small objects are aggregated to bigger objects, like adders or multipliers, which have a more complex state and provide more complex methods. These objects in turn are aggregated to even bigger objects, representing the system components (such as an I2C controller or a bus). Finally, one can see the whole FPGA as one big object, providing its functionality to the outer world. The distinctive feature of FPGAs (in contrast to common software) is that **every** object (even the smallest) is running concurrently. In software projects based on C++ or Java, such a design method would be quite unusual since here the processing units are just a few (or even only one) processor cores executing the program. Thus, such a high parallelism would not really be realizable by these processing units and therefore would only come with disadvantages like a higher synchronization effort. Nevertheless, the multi-threading constructs already existing in object-oriented languages can be used to describe a system where all objects are running concurrently, without overstretching the language specification. In other words: the description of multiparallel hardware lies within the boundaries and abilities of today's object-oriented languages — for hardware description they only have to be used the right (highly concurrent) way.

To demonstrate the similarity between FPGAs and object-oriented programming, figure 4.1 presents a multi-threaded object-oriented representation of LUTs and flip-flops. First of all, an object called *ParObj* is created. It is deriving from *Thread* and calls the method *calc()* continuously. *calc()* contains the actual functionality. To represent the characteristics of hardware (continuous execution), the calc() method runs in a loop. It must therefore not be mistaken for the run() method in Java threads that ends after one execution. All objects representing a piece of hardware derive from *ParObj*. The object *LUT* contains an internal array (*SRAM*) storing 16 different values. The value of the output *o* depends on the applied address *a*. The object *FF* represents a flip-flop, which is trig-

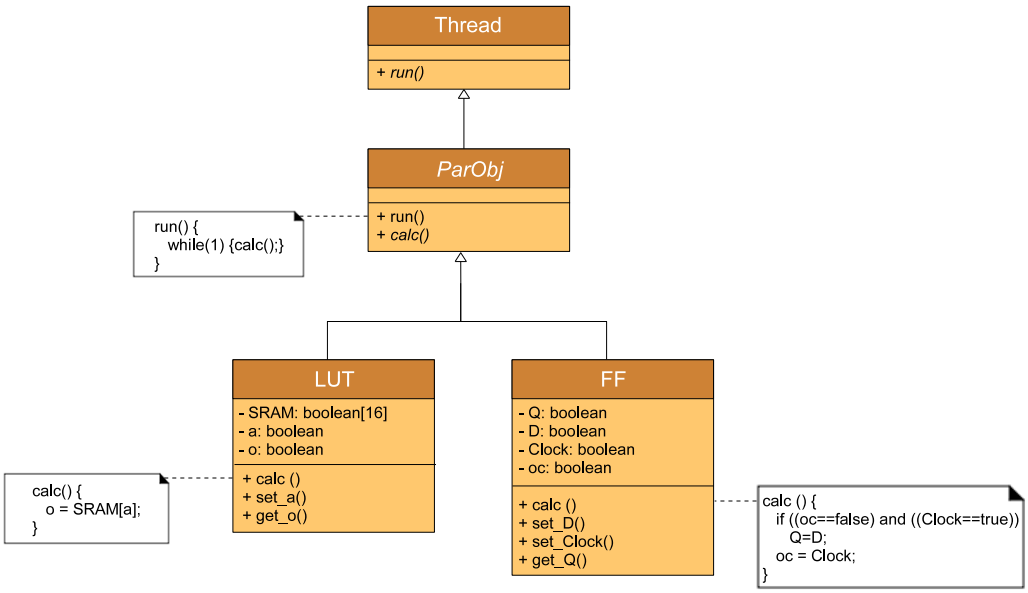


Figure 4.1: UML representation of an FPGA's basic elements

gered by a rising edge on the variable *Clock* (for edge-detection, the auxiliary variable *oc* is used). If a rising edge is detected, Q gets the value of D.

After defining these basic elements, *LUT* and *FF* can be aggregated to logic functions like *AND* and *XOR*, which finally can be used to build a more complex object such as a halfadder (see figure 4.2) or a finite state machine. This object-oriented way of object description and aggregation exactly matches the way LUTs and flip-flops are interconnected inside an FPGA by common synthesis tools.

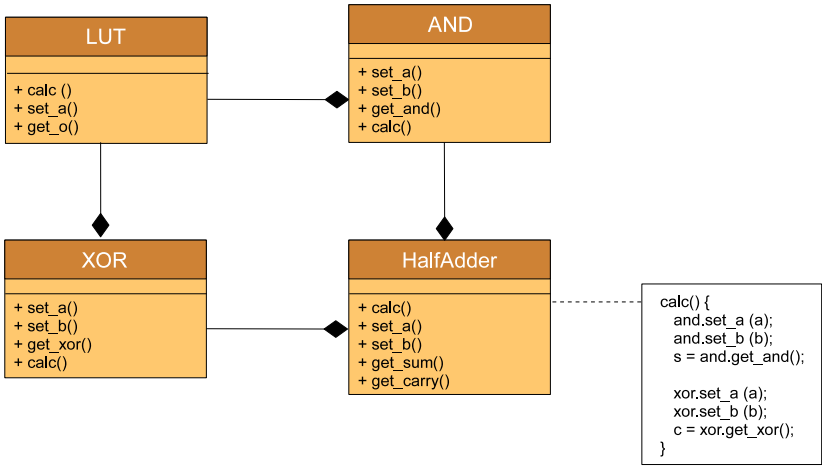


Figure 4.2: UML representation of a Halfadder

Software's Paradigm Shift

Nowadays, hardware developers have to face both hardware and software development since modern FPGAs are big enough to contain embedded processors or even whole processor systems (which is called SoC — System on Chip). The software running on these embedded processors is underlying particular constraints coming with the surrounding architecture. It has to be as simple as possible, as small as possible, as fast as possible and as close as possible to the underlying hardware. Scalability, modularity, abstraction, compatibility and convenience often are no criteria at all. Due to this, in many cases embedded processors still are programmed in simple C or even in Assembly Code.

In contrast, in the last 20 years software development regarding desktop computers as well as high-performance computing has been undergoing a radical paradigm shift. Famous applications like Open-Office or Eclipse are not the work of one single zealous programmer but are developed and improved by hundreds of programmers over years. These programmers are spread all over the world and do not necessarily know each other. Such huge projects are demanding and require well-defined programming methods and paradigms, such as encapsulation and modularization, which help the programming team to keep the code well-organized, readable, changeable and manageable. All these demands led to the common usage of object-orientation (using languages like C++ or Java). Furthermore, programmers are no longer allowed to write single-threaded, sequential code and to rely on new processors to get exponential performance improvements. The reason is that today's processor generations are more characterized by the number of their cores as by the speed of an individual core. Thus, formerly single-threaded code was or is transformed into multi-threaded code which utilizes the available processor cores as efficiently as possible. The upcoming change from multi-core to many-core architectures even intensifies this trend.

This situation results in two completely different perspectives. In hardware development, software is usually running on an embedded system and operating on a very low level. Due to this, software is often seen as something solely sequential — which is underlined by statements like "*C code (and software designed for microprocessors in general) is a sequential, instruction based language. One instruction is executed after another in a sequence*" [158] or "*C and C++ are optimized for expressing sequential algorithms and contain no language-level support for concurrency, in part because there is no agreed-upon model for parallel programming*" [119]. Although the conclusion that C has originally been designed as a sequential language is true, these statements do not take heed of the above-mentioned paradigm shift coming with multi-threading. This leads to the curious situation that many HLS languages are trying to subsequently introduce the concept of concurrency into software (e.g. using *par* statements in Handle-C), although these concepts already *have been* introduced by the software developers themselves and today are a natural part of modern programming languages.

Since the aim of this thesis is to describe hardware without using language constructs software-developers are not familiar with, the resulting questions are:

- Which constructs are used by today's software developers to express concurrency?
- Which methods are used to manage communication between concurrently running threads?

The first question can easily be answered: object-orientation in combination with multi-threading. Without any doubt, this is today's standard approach which can be found in nearly every modern software project.

The answer to the second question is not that obvious since there are various approaches to realize inter-thread communication. Nevertheless, one particular framework became very popular: Qt. The actual aim of Qt is to provide a cross-platform development framework that offers the programmer an easy way to create a GUI. Such graphical interfaces are a typical example for programs using concurrently running objects. Every graphical window is represented by an object which is running in parallel to the other objects. Thus, the Qt developers had to find an efficient as well as convenient way to let these objects communicate with each other. To reach this goal, Qt makes use of *Signals* and *Slots*. *Signals* send a message out of an object, while *Slots* are the corresponding receivers. In contrast to simple *set* and *get* methods, *Signals* and *Slots* act as buffers, which are able to store a sent information until it is retrieved. Furthermore, every object solely accesses its **own** *Signals* and *Slots* (in contrast, in the conventional way, it would have had to call the *set* and *get* methods of **another** object). This makes the synchronization much easier. Due to the efficiency and elegance of Qt, many software projects make use of it (e.g. KDE, Google Earth, Opera, Skype, VLC media player). Furthermore, Qt is available for C++, Java, Python, Ruby, PHP, Haskell and Perl.

Dynamic

The original question was which already existing software language constructs can be used to express dynamic partial reconfiguration. Using an object-oriented representation of hardware, every hardware component is represented by an object. From this point of view, the subsequential loading of a hardware component using DPR is comparable to the dynamic instantiation of an object. So the question regarding object-oriented programming is:

- Which object-oriented constructs exist to dynamically create or destroy objects?

The answer to this question is quite obvious since the dynamic creation and destruction of objects is a natural part of object-oriented languages. The creation is realized via the keyword *new*. Regarding the destruction of objects, the two most popular language representatives (C++ and Java) differ a little, but both fully support it. C++ denotes the destruction of an object explicitly, using a destructor and the keyword *delete*. In Java the removal of unused (that means unreferenced) objects is realized automatically by the garbage collector. Deleting an object is therefore realized via setting the corresponding

object reference to *null*. Figure 4.3 illustrates the elegant straightforwardness of the creation and destruction of objects in today's object-oriented languages using the example of Java.

```
public class MainClass {
   public void static main(String[] args) {
     ...
     MyClass dynObject = new MyClass();  //dynamically create an instance of MyClass
     ...
     dynObject = null;                   //dynamically destroy this instance
  }
}
```

Figure 4.3: Creation and destruction of an object in Java

A very important aspect regarding the dynamic instantiation of (concurrently running) objects is the corresponding inter-object communication. The underlying communication structure has to be very flexible, so that not only the objects themselves can be created and destroyed dynamically but also the communication channels between the objects can be changed at runtime. The already mentioned Qt framework provides a very elegant way to establish and to dissolve communication channels between several objects dynamically. For this, the methods *connect* and *disconnect* are used (for more details about Qt's communication concept, see chapter 2.7.3).

At this point, it is important to underline that object-oriented languages are uniquely qualified to support concurrency and dynamic in such a well-defined way. Of course it is possible to use C or other procedural languages in combination with multi-threading. However, without a well-defined encapsulation, inheritance and modularization as is usual in object-oriented languages, such a high parallelism coming with programmable hardware would quickly become confusing and unmanageable. For example, the communication between several threads could not be realized via well-defined *connect* and *get* methods but would have to be realized via shared (global) variables.

Functional languages are a very good candidate to describe concurrency but they do not provide an elegant possibility to denote the dynamic instantiation of modules coming with the usage of DPR [159].

4.2 Parallel Object Language

In the last section it became clear that object-oriented programming in combination with multi-threading is a very good method to describe dynamic hardware. The reasons are:

- Conventional HDLs already describe hardware in a way which is close to the object-oriented paradigm
- The basic elements of FPGAs can be represented quite well using objects
- The description of multiparallel hardware lies within the specification of object-oriented languages
- Today's software is highly concurrent — and uses multithreaded object-oriented programming languages like C++ or Java
- Frameworks like Qt provide a very elegant solution for communication and synchronization problems
- The dynamic instantiation of objects is a natural part of object-oriented languages — and can be used to express DPR

The two most popular representatives of object-oriented programming languages are C++ and Java. Thus, the decision had to be made, which one of them should be used to describe hardware. Since C++ relies on the usage of pointers which cause many problems regarding synthesis (see chapter 3.3), the decision was made to use Java. However, it is important to underline that the design and implementation of the Framework depends more on the chosen paradigm (object-orientation) than on the concrete language. The gained insights are easily transferable from Java to C++.

Having listed all the advantages and elegant possibilities coming with object-orientation, it is important to say that Java (and C++ as well) do not forbid the programmer to create ugly code ignoring all the advantages coming with object-orientation. For example, in Java it is possible to put all functionality into *public* methods of one single class that also contains the *main* method — and therefore to fall back to the procedural programming paradigm. Since Java is only a good choice for describing hardware as long as all the features listed above are utilized, the decision was made to support only a well-defined subset of Java, forcing the programmer to make use of multiple objects running in parallel. Thus this synthesizable subset of Java was named POL (Parallel Object Language).

4.2.1 Basic Concept

The basis of POL is a class called *ParObj* (see figure 4.1). Every class that shall be synthesizable has to derive from it. Figure 4.4 shows the implementation of *ParObj*.

```
public abstract class ParObj extends Thread {
    private boolean isRunning = true;
    public ParObj() {
       this.start();
    }
    public abstract void calc();
    public void run() {
      while(isRunning)  {
        calc();
      }
    }
    public void finish() {isRunning=false;}
}
```

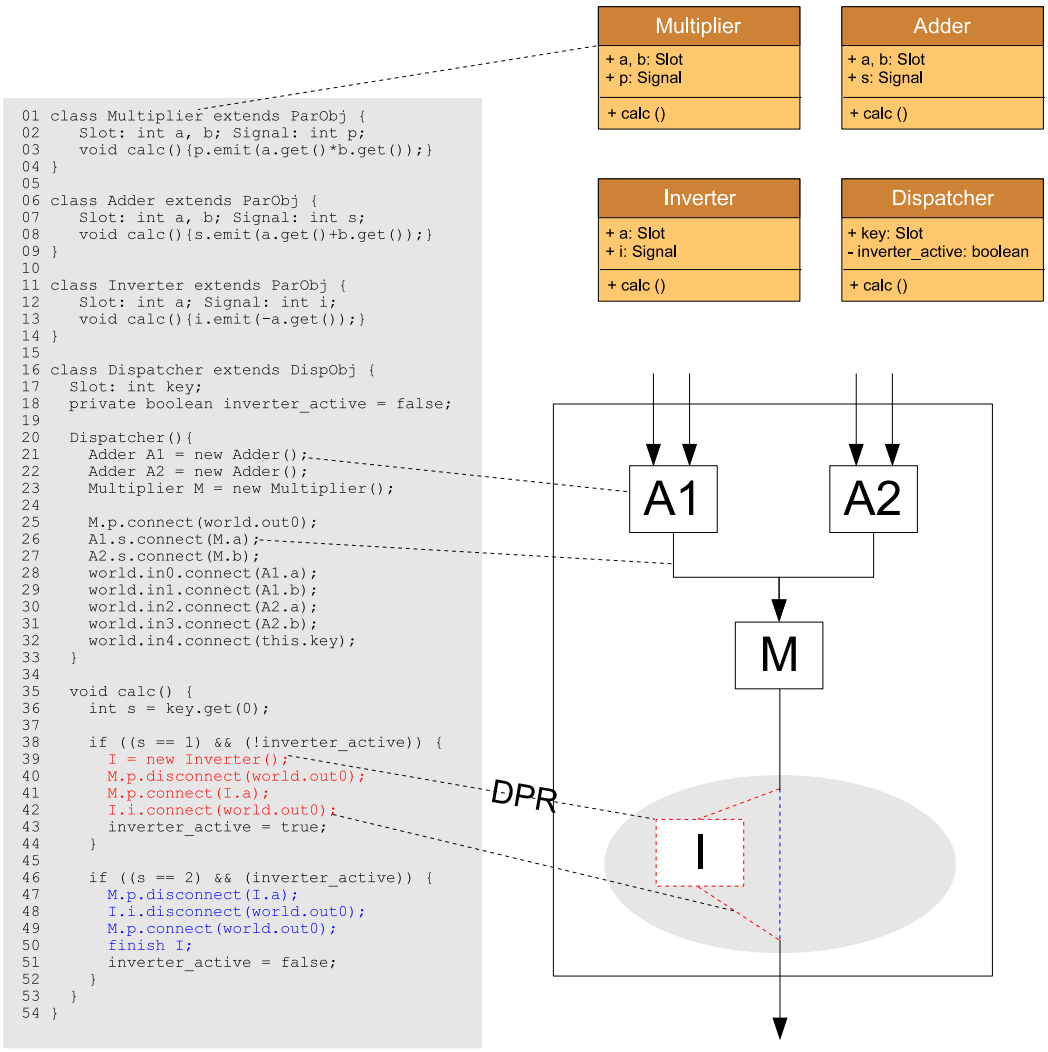
Figure 4.4: Implementation of ParObj

Since *ParObj* derives from *Thread* it is executed as an independent thread. The constructor of *ParObj* calls the method *start()*, which starts the thread (and therefore starts *run()*). The method *run()* just calls the method *calc()* again and again. The actual functionality has to be placed inside *calc()* by the class deriving from *ParObj*. This way, the concurrent and ongoing character of hardware components is represented.

Figure 4.5 illustrates an extensive example of two *Adders*, a *Multiplier*, and an *Inverter* implemented in POL. The communication between several objects is realized via *Signals* and *Slots*. The method *get()* receives a message from a *Slot*. The method *emit()* sends a message to a *Signal* (line 03). If object *A* shall send messages to object *B*, *Signals* from object *A* have to be connected with *Slots* from object *B*. This is realized via the method *connect()* (line 41). Established connections can be dissolved using the method *disconnect()* (line 47). The creation of a new object is realized via a simple *new* (line 39). The destruction of an object differs a little from the Java standard since the destruction delay coming with the usage of a garbage collector is a very unwanted effect regarding hardware. Thus, POL provides the method *finish()*, which is used to end the execution of a thread (line 50). Both *new* and *finish* are highly correlated to DPR since POL allows their usage at every position in the code. This is the biggest difference between POL and existing HLS languages.

The class *Dispatcher* derives from *DispObj* (line 16) which is very similar to *ParObj*. The difference is that the object deriving from *DispObj* is a singleton and is automatically instantiated by the Framework at start-up. Therefore it is comparable to the *main* method in Java or C++.

As one can see in figure 4.5, the calc routine is allowed to contain usual Java.




```
01 class Multiplier extends ParObj {
02    Slot: int a, b; Signal: int p;
03    void calc(){p.emit(a.get()*b.get());}
04 }
05
06 class Adder extends ParObj {
07    Slot: int a, b; Signal: int s;
08    void calc(){s.emit(a.get()+b.get());}
09 }
10
11 class Inverter extends ParObj {
12    Slot: int a; Signal: int i;
13    void calc(){i.emit(-a.get());}
14 }
15
16 class Dispatcher extends DispObj {
17   Slot: int key;
18   private boolean inverter_active = false;
19
20   Dispatcher(){
21     Adder A1 = new Adder();
22     Adder A2 = new Adder();
23     Multiplier M = new Multiplier();
24
25     M.p.connect(world.out0);
26     A1.s.connect(M.a);
27     A2.s.connect(M.b);
28     world.in0.connect(A1.a);
29     world.in1.connect(A1.b);
30     world.in2.connect(A2.a);
31     world.in3.connect(A2.b);
32     world.in4.connect(this.key);
33   }
34
35   void calc() {
36     int s = key.get(0);
37
38     if ((s == 1) && (!inverter_active)) {
39       I = new Inverter();
40       M.p.disconnect(world.out0);
41       M.p.connect(I.a);
42       I.i.connect(world.out0);
43       inverter_active = true;
44     }
45
46     if ((s == 2) && (inverter_active)) {
47       M.p.disconnect(I.a);
48       I.i.disconnect(world.out0);
49       M.p.connect(world.out0);
50       finish I;
51       inverter_active = false;
52     }
53   }
54 }
```

Figure 4.5: POL implementation utilizing DPR

4.2.2 Toolflow

To be able to use POL to develop and to describe FPGA designs, the corresponding toolflow has to provide a high-level simulation as well as a synthesis flow. Since POL is defined to be a subset of Java, the usage of the JVM[3] to emulate the behavior of the POL code suggests itself. The corresponding high-level emulator can make use of inheritance and encapsulation to provide the needed classes and methods (e.g. *ParObj*, *Signal*, *Slot*, *connect*, and *disconnect*). An important requirement regarding this emulator is that its behavior has to be identical to the behavior of the generated hardware. Thus, in the following chapters the focus is first on the realization of the FPGA design before the refinement of the emulator is discussed.

The synthesis flow starts with POL and translates it directly to VHDL (without using the intermediate step of Java Bytecode). From this point on, standard synthesis tools can be used to generate the needed bitfiles. Since this thesis focuses on the implementation of DPR and the synthesis of *new*, every *ParObj* is realized as a separate VHDL file which can be used to create a partial bitfile (e.g. using Xilinx's partial reconfiguration tools — see chapter 2.6.1). Chapter 5 will focus on the requirements coming with this specification of POL and the toolflow. Finally, chapter 6 will refine this approach to a detailed toolflow.

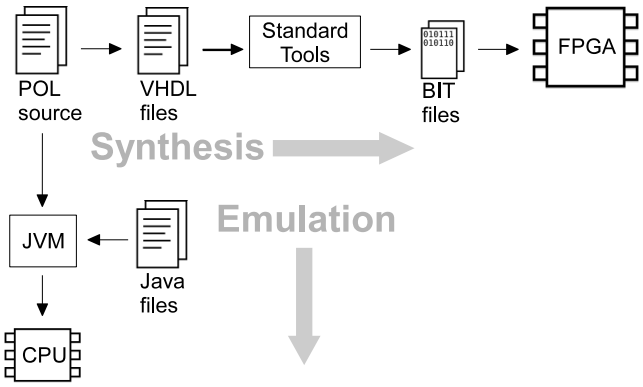


Figure 4.6: First sketch of the toolflow

[3]Java Virtual Machine

5 Requirements Analysis

In chapter 3 the current state of the art regarding DPR, HLS and the combination of both was analyzed. Chapter 4 concluded this analysis and showed that a very good way to describe dynamic hardware on the algorithmic level is to use an object-oriented multi-threaded language such as C++ or Java. A subset of Java, forcing the developer to make use of multi-threading, encapsulation, and well-defined inter-object communication channels (namely *Signals* and *Slots*) was introduced: POL (Parallel Object Language). This chapter focuses on the design space exploration of the corresponding hardware.

- How can POL be translated to valid VHDL code?
- Which kind of infrastructure is needed to support inter-object communication?
- Is it necessary to introduce additional restrictions to POL, to be able to synthesize it?

To answer these questions, requirements coming with the specification of POL, typical applications implemented on FPGAs, and the constraints coming with the usage of DPR are analyzed. The resulting design of the Framework is described in chapter 6.

5.1 Parallel Object Language

POL has been designed to describe concurrent hardware, using objects. All synthesizable objects derived from *ParObj* and make use of *Signals* and *Slots* for inter-object communication. The distinctive feature of POL is that all objects are allowed to be instantiated at *any* position in the code. If class *B* is instantiated in the *calc()* function of class *A*, the corresponding synthesis process has to make use of DPR to be able to instantiate hardware object *B* dynamically. Thus, the FPGA design has to support the dynamic instantiation and destruction of instances. Hereby POL defines no limitations in the number of classes or instances.

If a class is instantiated at start-up and never destroyed, it can be seen as a static class (which exists all the time and therefore does not depend on DPR). However, using DPR it is possible to overmap the FPGA (details on overmapping are shown later). In this case it makes sense to implement these classes as (reconfigurable) dynamic classes even though they could be realized as purely (conventional) static classes. The question of whether a potentially static class should be implemented as a static or as a dynamic class is part of the design space exploration process and therefore hard to answer automatically (see

chapter 3.3). Therefore it makes sense to introduce a third synthesizable object type to POL (namely *StatObj*) which enables the denotation of static classes explicitly.

The code inside *calc()* is standard Java and provides no additional constructs to explicitly express concurrency or timing. Therefore the synthesis of POL depends on a Java-to-VHDL compiler which is able to automatically translate the ordinary Java inside *calc()* to VHDL. This leads to a multi-grained description of concurrency. The fine-grained concurrency inside *calc()* (mostly depending on data flows — leading to the usage of Loop Unrolling, Pipelining and other well-known parallelization methods) is determined automatically, while the coarse-grained concurrency (between several instances) is expressed explicitly.

Since POL makes use of *connect* and *disconnect* to denote the dynamic instantiation and destruction of communication channels between several objects, the corresponding hardware design has to provide a very flexible communication infrastructure which potentially connects all hardware instances to each other. Furthermore, POL classes are not embedded in a surrounding static infrastructure class (like in JHDL or OSSS+R) which allows usage of the same VHDL signals for all alternately loaded hardware classes. Instead of this every POL class has the flexibility of defining a highly customized interface. In other words: each class is free to decide how many *Signals* and *Slots* it uses. Therefore an additional abstraction layer between the *Signals/Slots* and the VHDL inputs and outputs of a VHDL component (e.g. an NoC[1]) is needed.

Since HLS is a very complex topic, there will always be cases in which the automatically generated VHDL code does not fulfill requirements. Thus, POL has to provide a possibility of implementing hand-written VHDL subcomponents. This approach is comparable to the possibility of embedding hand-written assembly code in C++.

[1] Network on Chip

5.2 FPGA Applications

Since DPR is solely usable on FPGAs, it is important to analyze the typical application areas of FPGAs and to deduce the consequences regarding POL and the Framework. Nowadays, four different types of architectures are used to perform calculations: CPUs (Central Processing Units), GPUs (Graphics Processing Units), FPGAs (Field Programmable Gate Arrays), and ASICs (Application Specific Integrated Circuits). Each of these processing architectures has its own advantages and drawbacks — hence, each has its own niche where it is the dominating solution.

CPUs are the most flexible solution, since the actual functionality is fully determined by the software running on it. Thus, a change in functionality is a simple change of some bits in a RAM. Their major drawback is that they are comparatively slow regarding streaming applications. GPUs are also fully controlled by software, but they are much more specialized than CPUs. Originally they were designed to quickly calculate image transformations. Today they are used to improve vector and matrix calculations whenever possible. ASICs can in some ways be seen as the opposite of CPUs. They are highly specialized, usually do not use any software, and provide a maximum possible throughput with minimum power consumption — but the functionality of a manufactured ASIC cannot be changed at all. FPGAs are the compromise between flexibility and throughput. Since all components of an FPGA run concurrently, its were developed to support the prototyping of hardware developers: after all bugs have been removed, the design is implemented in an ASIC. Today, FPGAs are used in a much wider spectrum. This is primarily due to the fact that the reconfigurability of FPGAs provides the possibility to change and to extend their functionality even after delivery. In the following two typical examples of the usage of FPGAs are shown: video streaming and data-acquisition (DAQ) in high energy physics.

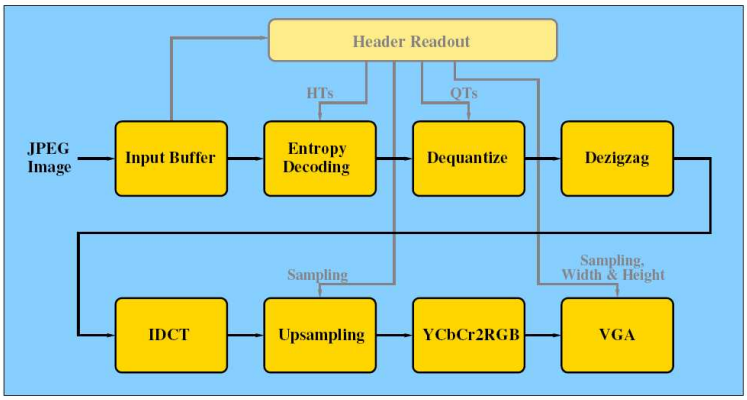


Figure 5.1: Essential calculation steps of a JPEG decoder [160]

5.2.1 Video Streaming

Video streaming is a typical example application which can be highly accelerated, if FPGAs instead of CPUs are utilized. The reason is the internal structure of video compression and decompression algorithms. They consist of several independent calculation steps which have to be executed consecutively for each datum of the video stream — but all data of the stream has to take the same calculation steps. Thus, video streaming can be massively accelerated using pipelining.

Figure 5.1 illustrates the essential steps of a JPEG decoder. With the exception of the header readout all components can be arranged in a pipeline chain and therefore be executed concurrently. Thus, even a CPU which is able to perform each calculation step as fast as an FPGA, would be 8 times slower than the FPGA. However, video decoding is not only very demanding regarding performance, but also regarding flexibility. Nowadays, there is a huge number of different video codecs and sub-codecs, each coming with its own calculation chain. Furthermore, every year new video codecs come into the market. Therefore it makes sense to utilize FPGAs instead of ASICs for video decoding.

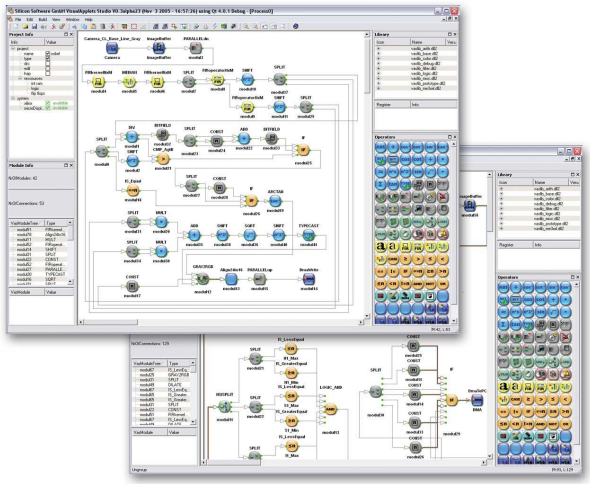


Figure 5.2: Graphical User Interface of VisualApplets [161]

A real-world application example of the realization of video streaming on FPGAs is the company Silicon Software (SiSo) [161], which distributes *microEnable* (an FPGA board particularly designed to support live video streaming) and *VisualApplets* (the corresponding software). Using SiSo's framework, it is possible to implement a video decoder and several filter functions (such as a FIR filter or a Sobel filter) just by drawing the corresponding data flow (see figure 5.2). A change in the data flow causes a new synthesis and implementation process, which ultimately leads to a new bitfile that is loaded onto the FPGA. Thus, the impact of a filter change on the video signal can be observed almost immediately. In a nutshell, one can say that SiSo's portfolio is based on the combination of high performance (using pipelining) with high flexibility (using reconfiguration).

5.2.2 Data Acquisition in High Energy Physics

Over the past century, scientists have built up a deep understanding of the subatomic constituents of matter in the universe and the fundamental forces binding them. More recently, they have developed compelling theories of how those building blocks came into being. Nevertheless, there are still significant gaps in the knowledge of the nature and evolution of matter on both a cosmic and a microscopic scale — and there are many questions to explore. To answer these questions, high-energy particle accelerators such as the LHC[2] in Geneva or FAIR[3] in Darmstadt are constructed. These accelerators provide high-energy, precisely-tailored beams of many kinds of particles at unprecedented quality and intensities. These charged particle beams are accelerated and employed to create new, often highly exotic particles. The particle accelerators are connected to a set of experiments such as the CBM (Complex Barionic Matter) experiment. Each experiment consists of a set of detectors that are used to determine which particle was located at which position at which time. Figure 5.3 illustrates the design of the CBM experiment.

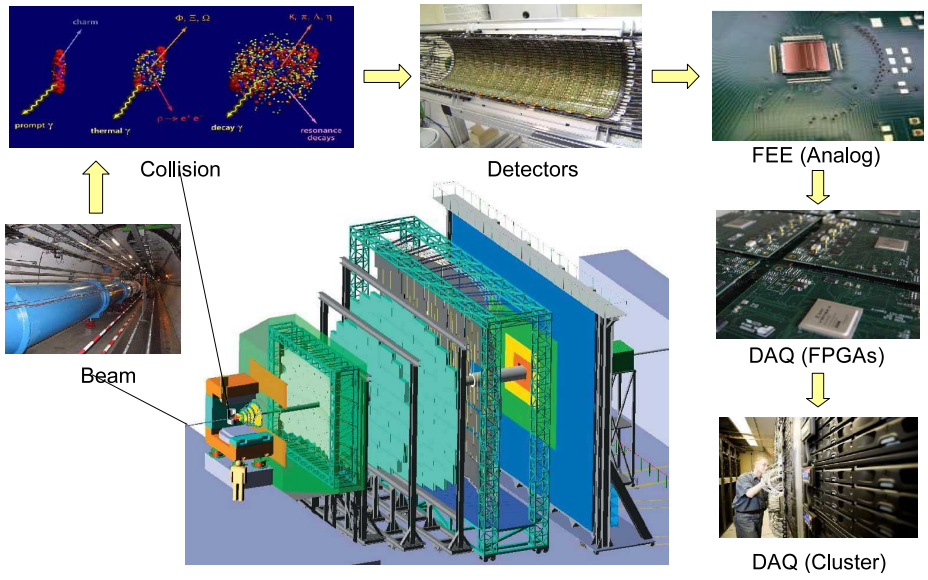


Figure 5.3: General DAQ setup [14]

A detector usually is a combination of many identical subdetectors, with each subdetector producing data on a dedicated channel. The proper design and development of the readout chain is crucial for the experiment. The data of the subdetectors is transferred as electrical amplitudes to the so called front end electronic (FEE) and represents the particle interactions in the detector. The FEE converts these analog electrical pulses to digital signals [162].

[2]Large Hadron Collider
[3]Facility for Antiproton and Ion Research

After the conversion from analog to digital signals, the FEE transmits the digitalized data to the data acquisition (DAQ). The DAQ is arranged in several layers, of which the first layer (often called Read Out Controller — ROC) receives and combines the data of several FEE boards. The next layer combines the data of several ROCs and so on. Since higher layers have access to the data of detectors which are localized at quite different places, the filter complexity increases with each layer number. While the FEE can only interpret the data from one detector channel, the first layer of DAQ can already use information from several channels to perform a first event selection, i.e. identifying particles and filtering out unnecessary data. The higher the layer, the more detector channels can be used and the less data from each channel needs to be processed. This allows to perform more complex algorithms (like ring finding, track finding and even vertex finding) in order to distinguish between interesting and uninteresting events. Since modern high-energy physics experiments aim for the detection of rather exotic particles which occur in events with a very low probability, very high event rates are needed. This results in a lot of uninteresting events that are already understood and can be filtered out. Hence very high compression rates can be achieved. Usually, the highest layer of DAQ is a computer cluster, which performs the most complex algorithms in software and stores the measured data and the calculated results in huge storage systems [163, 164]. The crucial question regarding all the data filtering in the DAQ chain is: which data is needed and which can be filtered out and irrevocably be thrown away?

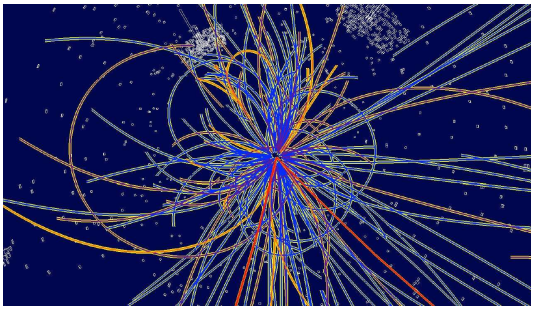


Figure 5.4: Simulation of a particle shower including the Higgs boson — all particles but the Higgs boson should be filtered out [165]

The answer to this question can change for two reasons: first of all, it is possible that important data is filtered out, due to a wrong decision in the past or due to a design error. Secondly, new measurements and the discovery of new particles can lead to completely new challenges and physical aims. In both cases a change in the filter logic is highly necessary. At this point FPGAs attracted notice, since their functionality can be changed (based on reconfiguration) although their throughput and performance is comparable to ASICs (based on concurrency and pipelining). Hence, big farms of FPGAs are used for DAQ in all modern high-energy physics experiments.

5.2.3 Resulting Requirements

Both video streaming and data-acquisition make use of FPGAs, since FPGAs provide a elegant combination of flexibility (based on reconfigurability) and high performance (based on concurrency and pipelining). This also applies to nearly all other applications implemented on FPGAs (like networking, automotive, and high performance computing). Since CPUs and GPUs do not support data-flow pipelining, and the functionality of ASICs is unchangeable, flexible pipelined algorithms clearly are the application niche of FPGAs. Thus, POL and the corresponding Framework have to be designed to support flexibility and pipelining.

It is pretty obvious that the usage of DPR is in line with the support of flexibility. In fact it even increases the flexibility of a given FPGA, since using DPR the chip can adopt its functionality *during runtime*. The more challenging part is the pipelining which highly influences the way the classes communicate with each other. In chapter 5.1 it has been shown that the usage of *connect* and *disconnect* leads to the need for a very flexible communication structure. The simplest way to achieve such flexible communication is to make use of a central memory which is accessed by all objects. However, this solution leads to a very strange situation: all objects can calculate concurrently but still have to wait for each other to be able to store the calculated data. In applications which highly depend on pipelining this destroys the whole parallelism — and therefore makes the utilization of an FPGA useless. Thus, the usage of a central RAM is not possible. The same conclusion applies to a simple bus which interconnects the objects with each other. Such a bus would sequentialize the inter-object communication and therefore prevent pipelining.

In conclusion, the Framework has to provide a communication structure which enables the objects to publish their results simultaneously. Moreover the Framework has to support pipelining as a natural feature. The utilization of a central memory or a simple bus for inter-object communication is off-limits.

5.3 Scheduling

Depending on the application, the reconfiguration can be a very rare up to a very frequent event. In the following, three basic types of scheduling scenarios are illustrated using the already presented example of DAQ algorithms [14].

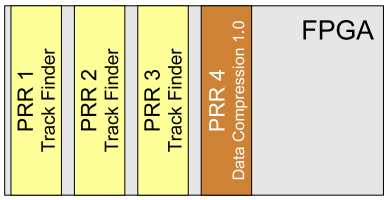


Figure 5.5: Reconfigurable DAQ system running 4 hardware modules

5.3.1 Update Scheduling

The most obvious replacement strategy is Update Scheduling. Figure 5.5 illustrates a system containing four reconfigurable modules placed in four PRRs (partial reconfigurable regions). Three modules are track finders and one is responsible for data compression. If it turns out that the data compression can be enhanced using a newer version, the chip can be updated using DPR. In this case, only the reconfigurable module for data compression would be exchanged (see figure 5.6). The rest of the FPGA can continue to calculate uninterrupted. The updating reconfiguration process is quite seldom (once per week or even less).

Due to the infrequency of the reconfiguration process, Update Scheduling often does not really depend on a *partial* reconfiguration. For most systems a global downtime once per week is acceptable.

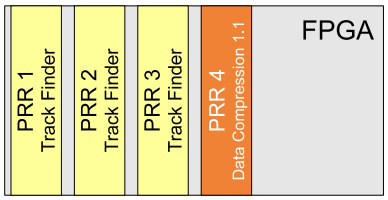


Figure 5.6: Reconfigurable DAQ system after Update Scheduling

5.3.2 Scenario-Based Scheduling

A more complex replacement strategy is Scenario-Based Scheduling. If the scenario of the experiment changes, the resulting data rates can change as well. Using Scenario-Based Scheduling, one is now able to react to the new conditions and to alter the DAQ system. For example, two of the three track finders could be replaced each by a vertex finder, as illustrated in figure 5.7. This way the FPGA would still operate at full capacity although the data rate has decreased. Thus, the FPGA calculates as much as possible and no resources are wasted performing active waits. If the conditions change again (back to maximum data rates), the system can react to it and replace the two vertex finders with two track finders again.

In contrast to Update Scheduling, in most cases Scenario Based Scheduling depends on *partial* reconfiguration. This is caused by the continuously increasing number of LUTs and FFS per FPGA, which causes the implementation of complete (processor) systems on one single FPGA. Part of these systems are the computing units as well as the communication units. In many cases the communication units have to be synchronized with the surrounding system. The loss of this synchrony can cause very unwanted effects. Real-world examples are the timestamp synchronization of the nXYTER[4] ROC (the re-synchronization depends on a global reset [167] of the detector) and the PCI Express interface of the ABB[5] (a full resynchronization of the board causes a deactivation of the PC's corresponding PCIe Slot — the reactivation of the PCIe Slot requires a full reboot of the PC [168]). Using *partial* reconfiguration, the process of Scenario-Based Scheduling only affects the modules that shall be exchanged and leaves other sensitive modules (like synchronized communication units) untouched.

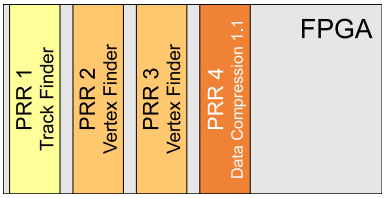


Figure 5.7: Reconfigurable DAQ system after Scenario-Based Scheduling

[4]Read-out ASIC for high resolution time and amplitude measurements — see [166]
[5]Active Buffer Board — the interface board between CBM DAQ and the corresponding computer cluster

5.3.3 Runtime Scheduling

Running an operating system like Linux on a PC leads to the seemingly simultaneous execution of hundreds of threads, although the underlying processor cores are not able to run more than a few (e.g. 2) threads at once. This is accomplished via Runtime Scheduling, which means the threads are executed alternately over time. Using DPR, it becomes possible to take advantage of this method with hardware. For example, a DAQ algorithm which needs 5 calculation modules can be executed on an FPGA containing only 4 PRRs. For this, the reconfigurable modules are loaded alternately into the PRRs (see figure 5.8). Using Runtime Scheduling this way, the reconfiguration process would be performed permanently. This is the most challenging but also the most powerful approach, since this way it is possible to overmap the FPGA. Furthermore, a scheduler which is able to perform *Runtime Scheduling* is automatically able to carry out Update Scheduling and Scenario-Based Scheduling as well. Especially Scenario-based Scheduling can be automated as far as possible, if the scheduler is able to swap in or swap out reconfigurable modules autonomously, based on the incoming data.

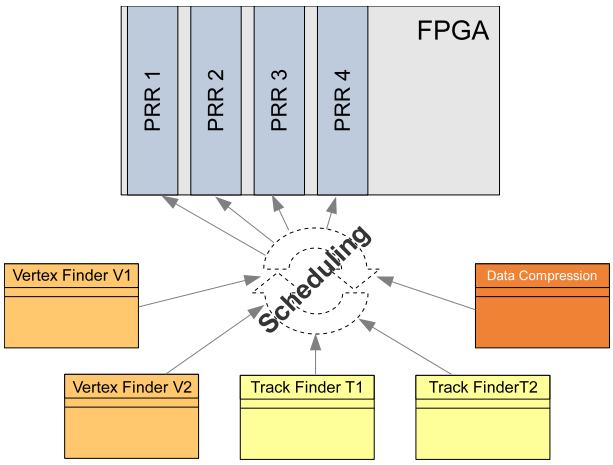


Figure 5.8: Reconfigurable DAQ system using Runtime Scheduling

The concept of overmapping the FPGA using Runtime Scheduling does not only come with advantages, but also has drawbacks and limitations. First of all it has to be emphasized that the principle of overmapping cannot be overdrawn. Every reconfiguration process takes time (at least several microseconds). During this reconfiguration process the corresponding PRR cannot be used at all. Furthermore, modules which are swapped out, obviously are not able to calculate anything. Thus, the exaggerated usage of overmapping leads to a significant reduction of the throughput of the design. Furthermore, all incoming data destined for temporary absent modules has to be buffered. Thus, an excessive usage of overmapping leads to the need for unrealistically huge buffers.

Nevertheless, if overmapping is used correctly it can lead to better resource utilization without reducing the throughput. In principle, overmapping is comparable to Datapath Merging presented in chapter 3.3.1. The big difference is that the multiplexed PRRs are able to completely change their functionality.

In chapter 3.2.4 the RISPP platform has been presented. RISPP attaches great importance to the trade-off between area consumption and execution time. If an instruction is rarely called, it can be implemented an area saving way. In contrast, if an instruction is called quite frequently, its implementation should be as time efficient as possible. This principle can be applied to the scheduling strategies: if a module is required very often, it should stay configured at all times and only be changed via Update Scheduling or Scenario Based Scheduling. If two or more modules are required rather seldom (e.g. in specific situations), they can be loaded alternately to the FPGA. In [112] this concept has been demonstrated using the example of an H.264 video encoder whose major functional blocks are Motion Estimation (ME), Motion Compensation (MC), Transform and Quantization (TQ), and Loop Filtering (LF). The size and the execution time of these components differ significantly (ME: 70%, MC: 17%, TQ: 8%, LF: 5%). Using the RISPP platform these components could be implemented achieving an area saving close to 50% compared to a static implementation of all functional blocks. Nevertheless a speedup of 26.6 could be achieved compared to a General Purpose Processor.

5.3.4 Resulting Requirements

The hardware scheduling algorithms described above result in additional demands on the Framework. Using Runtime Scheduling, data heading for temporary absent modules has to be buffered. This has to be done in a way that preserves the parallelism of the system. The scheduler that controls the scheduling process has to be responsive to changing conditions. It has to recognize autonomously which module has to be loaded, based on the incoming data and the data produced by the modules. In other words: the scheduler is responsible for the correct realization of a possible overmapping process. The reconfiguration overhead has to be kept as low as possible. Otherwise the system wastes most of the time, performing reconfigurations — or, in a worst case scenario, might not even be able to process the incoming data. The interconnections have to be very flexible, since it is possible, that e.g. Track Finder *T1* was at first loaded into *PRR 1*, but later is loaded into *PRR 2* (due to scheduling operations). The underlying NoC (Network on Chip) has to be able to handle this correctly. Nevertheless the throughput should stay as high as possible.

Although Runtime Scheduling is very challenging, the decision was made not to limit the number of instances in POL to the number of dynamic areas and thus to allow overmapping. Therefore it is the task of the POL programmer to use this technology wisely and not to overstretch it. If overmapping is used correctly, it can lead to better FPGA utilization without reducing the overall throughput (as the RISPP group showed).

5.4 Dynamic Partial Reconfiguration

In the following, the insights into DPR which were presented in detail in chapter 2 and in chapter 3 are summarized — and the resulting guidelines for the Framework are deduced.

5.4.1 Partitioning Options

In principle there are four different partitioning strategies: free 2D partitioning, 2D block partitioning, free horizontal partitioning and horizontal partitioning[6]. The two free partitioning options come with the need for online-routing, since they do not provide a well-defined steady interface. Although the ITIV Karlsruhe has proven that such online-routing is possible[7], it dramatically decreases the reconfiguration speed. This is in opposition to the requirement that the reconfiguration overhead should be as low as possible. Therefore the two free partitioning options cannot be used in the Framework.

A purely horizontal partitioning is very demanding regarding the pins of the FPGA (used for DDR-RAM, Ethernet, etc.), since the location of the pins highly depends on the available number of banks and the corresponding I/O standards and less on the internal partitioning of the FPGA. Thus, using purely horizontal partitioning depends on the usage of a DPR aware substructure (e.g. a specialized board) such as the Erlangen Slot Machine[8] — or requires a very extensive feed-through routing depending on a huge number of feed-through routes and the corresponding routing and placement constraints [40]. In addition to this, the Virtex-4, Virtex-5, Virtex-6, and Spartan-6 FPGAs provide a very convenient and easy access to 2D partitioning. Thus, the decision was made to make use of 2D block partitioning, which significantly increases the flexibility of the floorplanning process.

5.4.2 Inter-Module Communication

The connection between the static part and the dynamic parts of a design can be realized either in a static way (e.g. using Busmacros) or in a dynamic way (using online-routing). As said before, online-routing significantly decreases the reconfiguration speed and therefore is not an option. Regarding a static communication, two solutions can be found: the Busmacros provided by Xilinx, and the ReCoBus provided by the HSCD Erlangen[9]. Unfortunately the ReCoBus-Builder only supports Virtex-2 and Spartan-3, while Busmacros are available for all FPGA series (since they can be created manually using the Busmacro tools developed by ITIV Karlsruhe and LIS Munich[10]). Thus, the decision was made to utilize Busmacros. The remaining question is: which type of inter-module communication should be used?

[6] see chapter 2.5.1
[7] see chapter 3.1.3
[8] see chapter 3.1.1
[9] see chapter 3.1.2
[10] see chapter 3.1.4

In chapter 2.5.2 five types of communication have been presented: adjacent communication, centralized communication, communication via shared memory, communication via crossbar, and communication via configuration unit. As shown before, typical FPGA applications highly depend on pipelining. Therefore, communication via shared memory and communication via configuration unit cannot be utilized. The usage of a crossbar depends on a specialized infrastructure (and a customized board) such as the Erlangen Slot Machine[8]. Since the Framework should not depend on a particular board design, this communication type also cannot be used. Adjacent communication provides the fastest way for adjacent components to communicate with each other — but wider communication requires the cooperation of the components in between. This makes the placement of the components very complex, since the proximity of the components influences their communication options and therefore their functionality. POL does not provide any constructs to determine which instance should be placed in which area. Thus, using adjacent communication would either lead to

1. significant overhead in each reconfigurable module (caused by communication units which are able to establish a wider communication)
2. a very complex scheduling strategy
3. additional POL constructs to denote the wished instance position

Option 1 increases the size of the reconfigurable modules considerably and thus significantly increases the reconfiguration times. Option 2 makes the job of the scheduler much harder and has a potential of conflict with other scheduling demands, which could lead to additional reconfigurations (which would not be necessary without adjacent communication). Both options would considerably decrease the throughput of the Framework and thus should not be implemented. The third option introduces low-level constructs to POL, which are only understandable and meaningful to a developer who has a deep understanding of the underlying Framework. This massively conflicts with the initial aim to describe DPR, solely using high-level language constructs that are already well-known in software-development. Due to this, the decision was made not to make use of adjacent communication at all.

The last communication type is centralized communication. Here, every dynamic module is connected to the static area. The interconnections between the modules are realized inside the static area. This can be done in a very simple way (e.g. using a simple bus or a shared memory) or in a very powerful and flexible way (e.g. using an NoC). The decision was made to use centralized communication and to implement a highly concurrent communication structure as part of the static area to connect the dynamic modules with each other.

5.4.3 Tools

Today, there are three ways of generating partial bitfiles:

1. manually, using the method of full write and partial readback [40]
2. using the Xilinx partial reconfiguration tools, utilizing PlanAhead for floorplanning[11]
3. using the ReCoBus-Builder, utilizing the blocking workaround [9]

At the time this thesis was started, only the second option provided both, an elegant floorplanning tool and support of all Xilinx FPGA series. Thus, the decision was made to utilize the Xilinx PREA tools.

5.4.4 Reconfiguration Unit

Finally, it has to be decided, which system component shall be responsible for the control and the execution of the reconfiguration. There are three options:

1. the scheduler is realized in software on a PC, which makes use of the FPGA's external reconfiguration interface (e.g. JTAG) to perform the reconfigurations
2. the scheduler is realized on an extra chip (CPLD or FPGA), which makes use of the FPGA's external reconfiguration interface
   (an example can be found in chapter 3.1.1)
3. scheduler and reconfigurable areas take place in the same FPGA — the scheduler utilizes the ICAP[12]

Although option 1 is the most convenient, powerful and flexible solution it is also the slowest one. The very high reconfiguration throughputs (of 400 MB/s or even 1200 MB/s) are in no way achievable using this reconfiguration method. Option 2 relies on a customized underlying board (such as the Erlangen Slot Machine[8]), which would limit the usage of POL to very specific hardware. Due to this the decision was made to place the scheduler and the reconfigurable areas in one single FPGA, since this option provides the best combination of reconfiguration speed and flexibility.

[11] see chapter 2.6.1
[12] Internal Configuration Access Port — see chapter 2.3.1

5.5 Summary

The specification of POL from chapter 4 comes with a set of requirements. The most challenging part is the high degree of flexibility regarding object instantiation and inter-object communication. POL allows the user to instantiate and to destroy objects as well as to establish and to dissolve their connection at *any* position in the code. Beyond that, POL allows an overmapping of the FPGA, which leads to the need for data buffering and to the usage of Runtime Scheduling. Nevertheless, the targeted field of application has a very high concurrency and throughput demand. Due to this, the Framework has to fulfill the following requirements:

- The modules have to be able to publish their results concurrently. Furthermore, pipelining has to be supported. Therefore, the utilization of a simple bus is not allowed.
- Dynamic modules can be interrupted during execution, swapped out and swapped in again later. The context of the affected modules has to be saved.
- Each module can be loaded into each dynamic area. The corresponding communication channels have to support this.
- Runtime Scheduling depends on a very fast reconfiguration process. Thus, the scheduler has to be placed on the FPGA and has to make use of the ICAP.

To enable a better utilization of the target FPGA's resources, POL has to be enhanced in order to support static objects (*StatObj*) as well as dynamic objects (*ParObj*). Finally, a type of high-level simulation is needed, which allows the developer to execute his or her POL code in software and thus to do a quick verification of the code. The resulting concrete design of the Framework, which is geared towards fulfilling all these requirements is described in the following chapter 6.

6 Design of the Framework

This chapter presents the design of the Framework. It illustrates which components and tools are needed for a toolset starting with POL and providing both hardware and software execution of the given code. First of all, the focus is on the general toolflow. After this, each component is described in detail.

6.1 The Toolflow

Figure 6.1 illustrates the toolflow of the Framework. From the users point of view, the process starts with the creation of the POL files.

After creating the POL sources, the Framework provides two ways of execution. The left part of the toolflow leads to an execution of a Java program on a general CPU. This enables a developer to verify the correctness of the program at an early stage and on a high level. For this, the POL sources are precompiled to usual Java files and combined with with the Java files of the Emulator. The Emulator contains a Java emulation of the surrounding network, input and output components (like a serial interface) and other system components as well as a emulation of *ParObj*, *Signals*, *Slots*, *connect()*, *disconnect()*, *emit()*, *get()* and other POL specific classes and methods.

The right part of the toolflow results in several bitfiles. These bitfiles are loaded onto an FPGA and represent the final hardware. For this, the POL-Compiler compiles the POL sources to VHDL. The focus of this compilation is on the dynamic object management and on the dynamic inter-object communication. There are many research groups which did or still do focus on the optimal translation of sequential algorithm descriptions to efficient hardware (see chapter 3.3). Therefore, the decision was made to keep this part simple. The idea is to prove the elegance and power of POL regarding dynamic hardware instantiation and communication, and at a later point in time (e.g. as a follow-up thesis) to combine this approach with an efficient C-to-VHDL or Java-to-VHDL compiler (handling sequential and purely static code). Thus, the POL-Compiler simply translates *calc()* to one single FSM (Finite State Machine). Each calculation is represented by one state of this FSM. Some optimizations are done (e.g. state merging based on data dependency analysis), but the focus clearly concentrates on the dynamic instantiation, swap-in, swap-out and the dynamic inter-object communication.

The static part of the design is represented by a so called System Template which contains the processor subsystem needed by the Scheduler as well as the Communication Matrix which is the communication structure that realizes the inter-module communication and the communication with the outside world. The Merger combines the generated

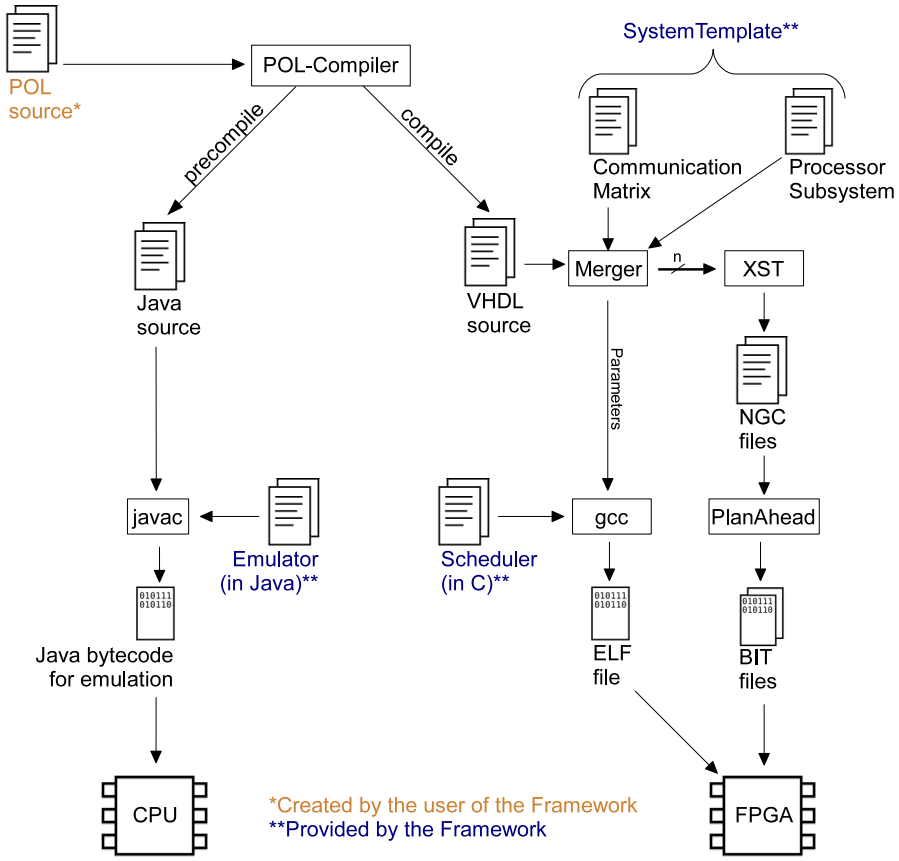


Figure 6.1: The general toolflow of the Framework

VHDL files and the System Template and starts multiple XST (Xilinx Synthesis Tool) synthesis runs, until all required netlist files (NGC files) are created. These files represent the functionality of the surrounding system and the reconfigurable modules. They are used as input for Xilinx PlanAhead which performs the floorplanning and generates the (partial) bitfiles. The Scheduler running in software on an embedded processor is controlling the Communication Matrix. It determines which reconfigurable module is loaded into which reconfigurable area at which time. Since the Scheduler also runs on the target FPGA, the reconfiguration has to be done via the ICAP[1]. Therefore, the generated design is a self-reconfiguring system.

The five central elements of the Framework are Communication Matrix, Scheduler, POL-Compiler, Merger, and Emulator. In the following sections, these elements are described in detail.

[1] see chapter 2

6.2 Communication Matrix

As shown in chapter 5, the benefit of dynamic hardware highly depends on a communication structure which is able to interconnect the dynamic modules in a flexible and concurrent way. In the Framework, this interconnection is realized by the Communication Matrix.

6.2.1 Data Format

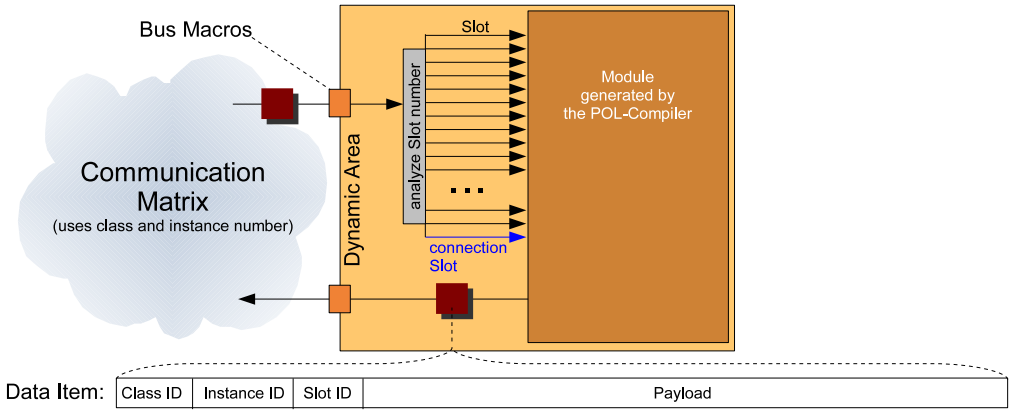


Figure 6.2: Interface between dynamic modules and Communication Matrix

Due to the requirements for the flexibility of the connections, it is not possible to translate POL *Signals* and *Slots* directly to VHDL input and output signals. On the one hand, each class can implement as many *Slots* and *Signals* as needed. On the other hand, each class shall be loadable to each dynamic area — therefore the interface between dynamic area and Communication Matrix has to be standardized. To resolve this contradiction, the Communication Matrix implements an additional abstraction layer. Each data item which is transported from one module to another contains the actual data and a destination address. This destination address contains the class ID, the instance ID and the Slot ID of the destination. For this, the POL-Compiler has to assign each class and each *Slot* with an ID (these IDs are fix at runtime). Furthermore, the Scheduler has to assign each instance with an instance ID (which is determined dynamically as soon as *new* is called — and therefore is part of the instance's context). The Communication Matrix uses the destination address to deliver the data item to the correct instance of the correct class. For this, the Scheduler has to tell the Communication Matrix, which instance is loaded to which dynamic area. Inside the dynamic area, the destination Slot ID is used to deliver the data item to the correct *Slot* (see figure 6.2). Using this kind of interface protocol, the VHDL interface between dynamic area and Communication Matrix is standardized while the number of *Signals* and *Slots* is as flexible as possible.

To establish and to dissolve connections (via *connect* or *disconnect*) a special data item type is used: Connection Messages. These messages are structured like normal data items, but the payload contains connection information (which *Signal* shall be connected to or disconnected from which *Slot*). Connection Messages make use of a special Slot ID, the so called connection *Slot*. More information about Connection Messages are shown in section 6.4.

6.2.2 Class Buffer

The use of Runtime Scheduling leads to the usage of swapping, which means that active instances are paused, removed from the FPGA and reloaded onto the FPGA at a later time to continue operation. Thus, it is possible that incoming data or produced data is addressed to an instance which is temporarily swapped out. Due to this, the Communication Matrix has to provide buffers which store the data until the corresponding instance is reloaded onto the FPGA.

The simplest approach would be to generate one buffer per instance. Unfortunately this is not possible, since the buffers are part of the static design and have to be instantiated at compile-time, while the number of instances is determined at run-time and unknown at compile-time. To solve this problem, the Communication Matrix instantiates one buffer per class (called Class Buffers). Multiple instances of one class share a single Class Buffer. This approach allows to handle the data streams of several classes in parallel, but avoids the instantiation of too many buffers. Since the number of classes is known at compile-time, the Communication Matrix only has to instantiate as many Class Buffers as classes that are used [169].

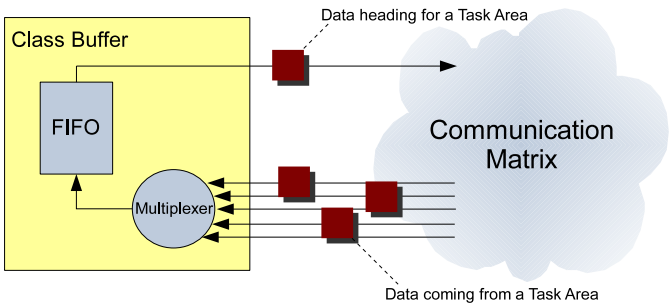


Figure 6.3: Design of a Class Buffer

As shown in figure 6.3 a Class Buffer consists of a FIFO and an active multiplexer. The FIFO is storing the data items. The active multiplexer is connected to the output of each Task Area. If a data item is heading for the class the Class Buffer belongs to, the active multiplexer takes this data item and sends a “data acknowledge” signal to the corresponding Task Area.

6.2.3 Task Areas

To enable the dynamic instantiation of modules, so called Task Areas are used. They consist of three parts (see figure 6.4). Part *I* is the inner module which the POL-Compiler has generated from the POL sources. Part *II* is an interface component which analyzes the incoming messages and transfers the data to the correct port. Part *I* and *II* form the dynamic area which is configured at runtime. Nevertheless, Part *II* actually belongs to the Communication Matrix and is represented by fixed VHDL code, while the inner module differs from object to object. Each inner module has n inputs (while n is the maximum possible number of *Slots*) and one single output. Its interface is therefore comparable to that of Chimaera (see chapter 3.2.2). For simpler scheduling, all dynamic areas have the same size. Thus, each dynamic module can be loaded to each dynamic area.

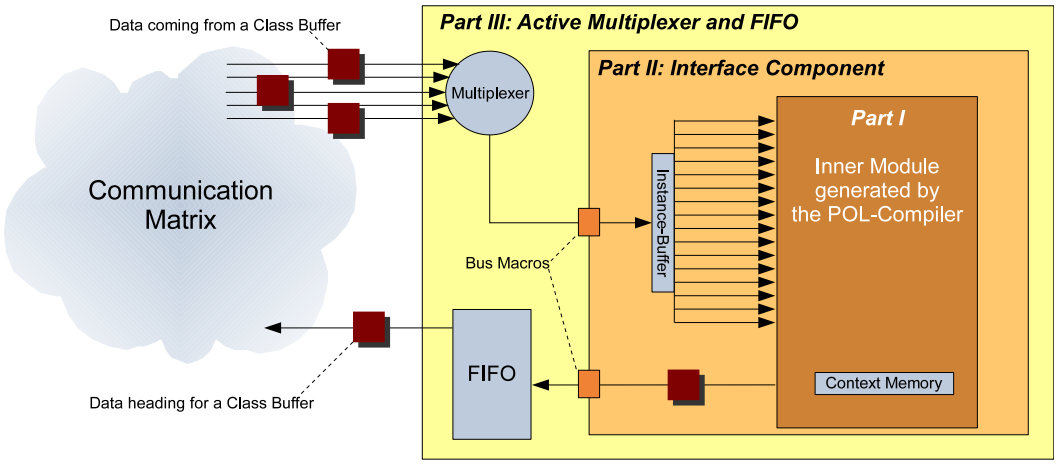


Figure 6.4: Design of a Task Area

Part *III* of the Task Area also belongs to the Communication Matrix and is part of the static design. It is dedicated to one particular dynamic area and contains an active multiplexer and a FIFO. The multiplexer is connected to each Class Buffer. When a data item is heading for the class that is currently loaded to the dynamic area, the active multiplexer takes this data item and sends a "data acknowledge" signal to the corresponding Class Buffer. Next, it transfers the data item to the interface component (Part *II*).

The FIFO stores data coming from the dynamic area. This way processed data can be stored, even if the Class Buffers are busy (e.g. because another instance of the same class is producing data concurrently). It is important that the FIFO is part of the static design, since this enables the reconfiguration of the dynamic area while data items are still stored in the FIFO and have not been delivered to the Class Buffers yet.

Context Memory

Since the Framework supports Runtime Scheduling, active instances can be paused, removed from the FPGA and continued at a later time. To enable this, the context of the corresponding instance has to be saved. It consists of intermediate data, connection information (which *Slot* is connected to which *Signal*) and general instance data (such as the instance ID). There are three possibilities, how this context could be saved. First, it could simply be stored in the flip-flops and read back via the reconfiguration unit. This way, the whole dynamic area would have to be read back for swap out. Second, it could be stored in extra-buffers in the Communication Matrix. This is very fast, but would significantly increase the size of the static area. Third, it could be stored in a BRAM inside the dynamic area, which is read back by the reconfiguration unit. This is slower than the second method, but provides the best trade-off between area and speed. Thus, the decision was made to use the third method in the Framework. Following this decision, each inner module contains a BRAM called Context Memory which is used to store the context of the object. More details about the Context Memory and its usage will be shown in the sections illuminating the Scheduler and the POL-Compiler.

Instance Buffer

POL provides two versions of the method *get* to receive a data item from a *Slot*. The first one is blocking (*get()*), the second one is not (*get(default value)*). The blocking *get()* is very useful to describe strict data dependencies (e.g. an Adder waiting for the two *Slots a* and *b* to contain data, before it can calculate $a + b$). However, using more than one blocking *get()* method the wrong way can easily cause deadlocks. For example, an Adder could first wait blocking for data on *Slot a*, but the incoming data stream first contains data for *Slot b* and afterwards for *Slot a*. In many cases,this dependency on the order inside the data stream is undesired. Therefore Part *II* contains an Instance Buffer which can be configured to provide an independent FIFO for each *Slot*. This way the dependency on the order is resolved. However, sometimes such a dependency is intended. In this case POL provides a statement to request a shared FIFO for two or more *Slots*. The Instance Buffer is realized using one or more BRAMs, whose content is saved via the configuration unit in case of a swap out (since the Instance Buffer belongs to Part *II*, it is part of the dynamic area).

Static Task Areas

If all instances of a class are only instantiated in the constructor of the *Dispatcher* and never destroyed at runtime, this class can be implemented as a *StatObj*, which means that the instances are never swapped out — and therefore can be implemented as part of the static design. These instances are placed in so called Static Task Areas. Static Task Areas have the same internal structure as Dynamic Task Areas, except that Part *I* and Part *II* are not placed in a dynamic area. This can be used to reach a better utilization of

the FPGA's resources - for example, if very small but permanently required modules are realized as *StatObj* instead of permanently using only a small part of a dynamic area.

System InOut

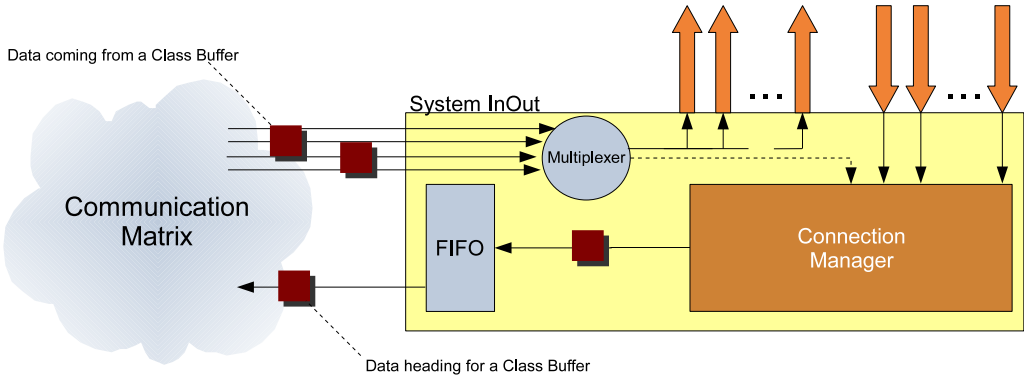


Figure 6.5: System InOut

Since the Communication Matrix makes use of a special addressing scheme, an interface between the outside world (only transmitting the actual payload) and the Communication Matrix is needed. This interface is called *System InOut* and is implemented as a smaller version of a Static Task Area. The Communication Matrix treats it as a normal object and therefore it has a class ID, an instance ID and requires the instantiation of one additional Class Buffer.

The active multiplexer of *System InOut* is used in two ways. First, it takes all data items which are sent to *System InOut* from the corresponding Class Buffer, removes the addressing information (class ID etc.) and transmits only the payload to the correct output port (depending on the received Slot ID). Second, it is used to receive Connection Messages that tell the Connection Manager, which input port should be connected to which destination (class ID, instance ID, Slot ID). This way, the Connection Manager is able to tag incoming data with the correct addressing information. The input part and the output part of *System InOut* share one single class ID. This is possible, since the input part does only make use of the Slot ID that belongs to Connection Messages, while the output port makes use of all *Slots* but the connection *Slot*.

6.2.4 The Big Picture

Figure 6.6 shows an example of a complete Communication Matrix, containing two task areas (one static, one dynamic), three Class Buffers and a connection to the outside world (*System InOut*). Furthermore the required interconnections are illustrated. Due to the three Class Buffers this version of the Communication Matrix supports up to two POL classes (and one *System InOut*).

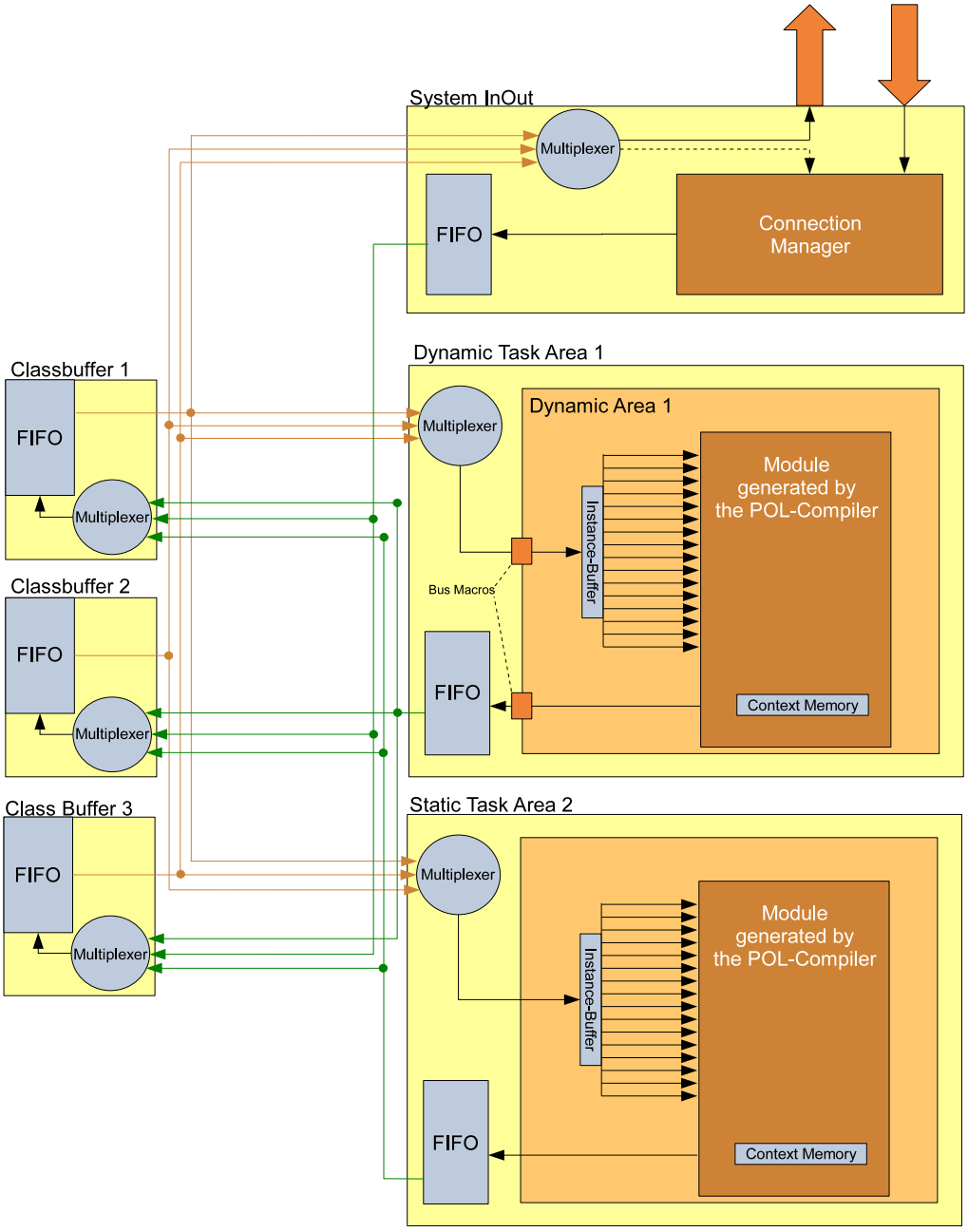


Figure 6.6: The Big Picture

6.3 Scheduler

The Scheduler is the part of the Framework that is responsible for the object management. That means, it conducts the creation of instances (via *new*), the destruction of instances (via *finish()*), and the swap-out process as well as the swap-in process needed for Runtime Scheduling. The Scheduler is a software program running on an embedded processor which belongs to the static part of the FPGA. It makes use of the ICAP to access the FPGA's reconfiguration unit.

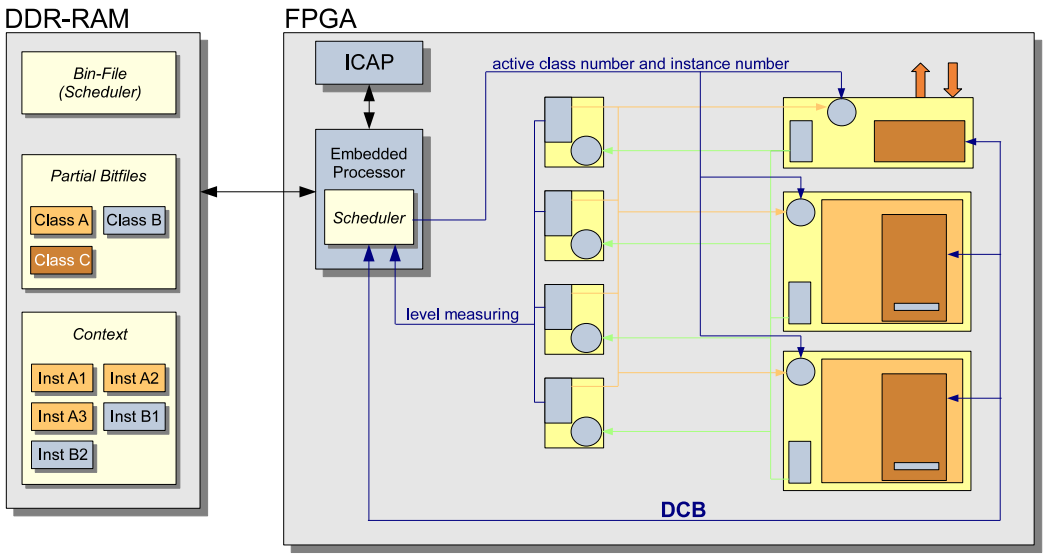


Figure 6.7: The Communication Matrix including the Scheduler's control flow (in blue) and memory usage

6.3.1 Dynamic Control Bus

POL allows the user of the Framework to create new instances at runtime (using *new* at any position in the code). Therefore running objects have to be able to signal the Scheduler, that they want to create such a new instance. Furthermore, the Scheduler has to notify a running object that it is going to be swapped out — so that the affected object can store its context in the Context Memory. Due to these requirements it is necessary that the running objects and the Scheduler are able to communicate with each other. For this purpose, the Framework provides an additional bus called DCB (Dynamic Control Bus). The DCB is a small and simple bus connected to all Task Areas and the Scheduler. It is only used to coordinate the creation, destruction and swapping of objects. Beyond that the Framework connects the filling level counters of each Class Buffer FIFO to the Scheduler. This way, the Scheduler always can consider the filling level of all Class Buffers. Finally, the Scheduler has write access to every Task Area multiplexer, so that it can notify each

active multiplexer which class and instance is loaded to the corresponding dynamic area. Figure 6.7 illustrates the resulting design of the FPGA.

6.3.2 Instance Management

To support the dynamic instantiation and destruction of objects, the Scheduler keeps a list of all classes and the associated active instances[2]. At this point it is important to point out that the creation of a new instance at first only leads to a new entry in this instance list. The actual configuration of the corresponding bitfile into a dynamic area is realized as a swap-in process which is explained in detail in the next section.

Every class is related to a partial bitfile which is stored in an attached DDR-RAM (see figure 6.7). Please keep in mind, that several instances of the same class share one single bitfile. This bitfile describes the functionality of the object and contains all major frames of the dynamic area except those describing the Context Memory and the Instance Buffer. These two components are placed in BRAMs and are handled separately as context of the instance. Each instance has its own context which is not shared at all. If the instance is swapped out, the context is stored in the DDR-RAM in a designated memory area.

[2] An instance is called active, when it has been created and not been destroyed, yet

Dynamic Creation

If a running instance (e.g. $A1$ — see figure 6.8) wants to create a new object, it uses the DCB to send the corresponding request and the class ID (e.g. B) of the object that shall be created to the Scheduler. As response, the Scheduler checks for a free instance ID and adds a new entry to the instance list (containing the new instance ID and a flag marking this instance as new). Next, the Scheduler uses the DCB to send $A1$ the ID of the new instance (e.g. 3). $A1$ uses this ID to create an object handler (in the example: $B3$) which can be used to establish and dissolve connections to the new instance (via *connect* and *disconnect*).

So, all instances of dynamic classes (derived from *ParObj*) are created by other instances. This approach requires an initial instance, which is automatically created at start-up by the Scheduler (comparable to the *main()* method used in Java). For this, POL provides the class *DispObj* which is a Singleton and is automatically instantiated by the Scheduler at start-up. Usually *DispObj* is used to create all objects that are needed right from the start, and to establish the communication between them.

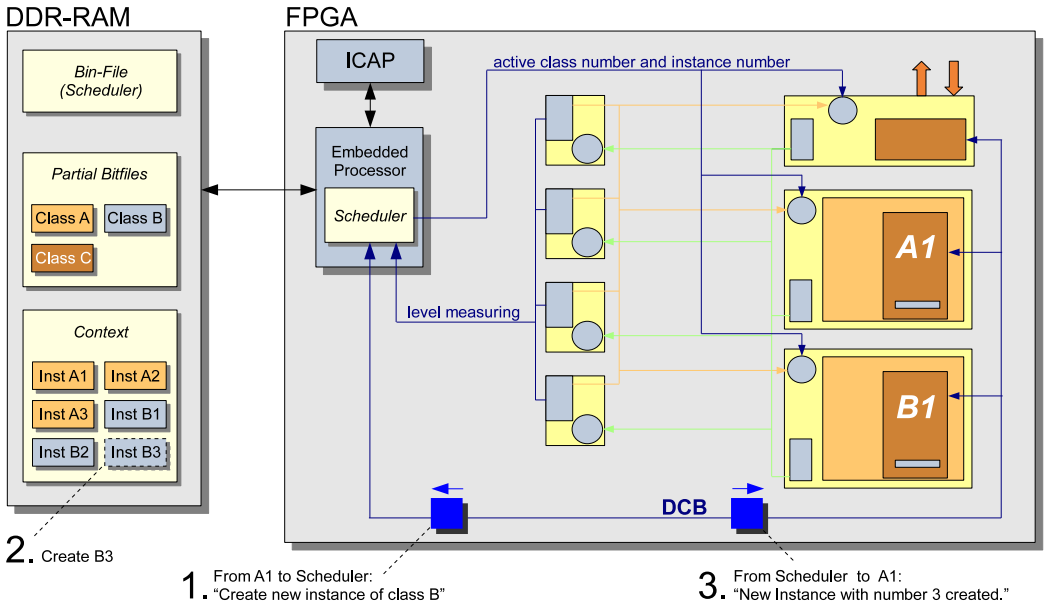


Figure 6.8: Dynamic creation of instance $B3$ by instance $A1$

Dynamic Destruction

The dynamic destruction of objects is a little more complex than the dynamic creation. This is caused by the fact that the destruction of an object shall happen only after processing all buffered data addressed to the object. Let's assume that instance $A1$ has sent some data to $B3$, and now wants to destroy $B3$ (because it is not needed any longer). Of course, this shall only happen *after* $B3$ has processed the data which has been send to it. Therefore $A1$ does not use the DCB directly to destroy $B3$, but sends a special message to $B3$ to request its destruction. This message is delivered by the Communication Matrix like all other data items and therefore reaches $B3$ after all data items. Once $B3$ receives the Destruction Message it stops calculating and uses the DCB to send the Scheduler the request for self-destruction. The Scheduler removes the instance from the instance list and marks the corresponding dynamic area as unused. The context of $B3$ is no longer needed. Thus, a read-back of the context is not necessary at all.

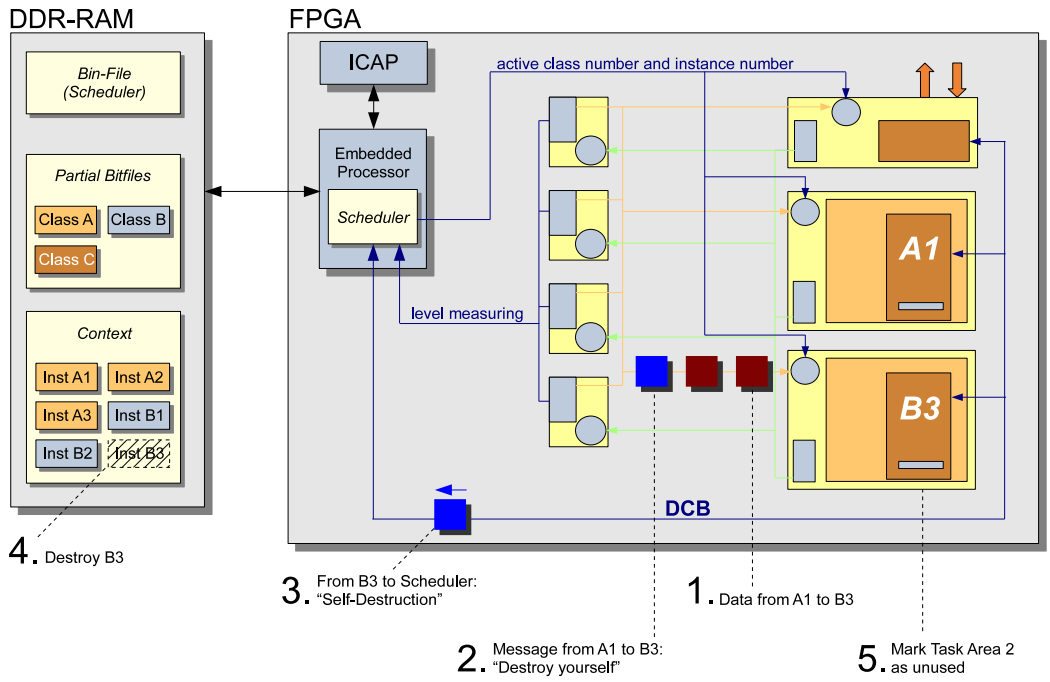


Figure 6.9: Dynamic destruction of instance $B3$ by instance $A1$

6.3.3 Runtime Scheduling

The creation of a new instance and its configuration into a dynamic area are two different events. This is caused by the support of Runtime Scheduling which allows to overmap the FPGA, and in case of overmapping requires an alternating configuration of active instances. In other words: instances can be swapped out although they are still active. For this, the context has to be saved in the DDR-RAM during swap-out and to be restored during swap-in.

Swap-In

If the scheduler decides to swap in an instance, it first has to find an unused Task Area. If no free Task Area is available, one of the active instances has to be swapped out first. After that, the new instance can be configured to the FPGA. This happens in four steps:

1. Configure the CLB frames of the dynamic area (representing the functionality of the instance)
2. Configure the BRAM frames of the dynamic area (representing the Context of the instance)
3. Configure the active multiplexer
4. Start the Instance

The first step can be skipped, if the correct functionality is already loaded to the dynamic area. This happens, if the scheduler replaces an instance by another instance of the same class. Since both belong to the same class, both have the same functionality — only their context is different.

The second step needs some extra effort, if the instance is new (which means it has never been configured to the FPGA before). In this case, the Scheduler creates an initial context which is completely empty except the instance ID stored in the first word of the Context Memory. This way, the Scheduler assigns an ID to a new instance.

The third step tells the active multiplexer, which instance of which class is loaded to the dynamic area. This step is necessary to ensure that the active multiplexer takes the data items from the correct Class Buffer. Finally the instance is started. For this, no extra effort is needed — the Scheduler just activates the Busmacros and releases the Task Area reset.

Swap-Out

If the number of active instances exceeds the number of available Task Areas, the Scheduler has to load the instances alternately. For this, it is important to be able to swap out active instances, which means to save the current state of an instance. Instances do not only store intermediate data in the Context Memory but also use flip-flops, shift-registers,

multipliers and other FPGA components. Before the swap-out can be performed, the instance has to store its intermediate data in the Context Memory, because for swap-in only Instance Buffer and Context Memory are read. To reduce the amount of intermediate data that has to be stored, the instances only can be suspended at particular breakpoints. The most important breakpoint is located at the beginning of the method *calc()*. Here, no variables which are local to *calc()* have to be saved. The amount of intermediate data is minimal (more information will be shown in section 6.4). Additional breakpoints are introduced at each blocking *get()*.

It is pretty obvious that such usage of breakpoints requires an interaction between Scheduler and instance. For this, the DCB is used. First, the Scheduler sends a suspend command to the instance. As a result, the instance uses the next available breakpoint to suspend its execution and to store all intermediate data in the Context Memory. Next, it sends a suspend-acknowledge command to the Scheduler. Now, the Scheduler saves the Context of the instance, deactivates the Task Area (deactivate the Busmacros, activate the Task Area reset), and finally marks the Task Area as unused.

Adaptive Scheduling and Short Reconfiguration

As shown before, the Communication Matrix instantiates one buffer per class — the Class Buffers. This leads to multiple instances of one class sharing a single Class Buffer. The Scheduler has no influence on the order of the data items inside a Class Buffer. Therefore, it is the data coming out of Class Buffer A that determines which instance of class A is needed next. The data coming out of Class Buffer B determines which instance of class B is needed next and so forth. The Scheduler can decide freely which class should be swapped in next, but the concrete instance of this class is defined by the data coming out of the Class Buffer. For this, the scheduler is not only able to read the filling level of the Class Buffer but also to read the target address of the data item coming out of of Class Buffer's FIFO.

The filling level of the Class Buffers mainly influences the decision, which class should be configured to a Task Area next. In principle the Scheduler starts with a simple round-robin scheduling, but adapts the priority of the classes according to the filling levels of the Class Buffers. The higher the filling level is, the higher the priority of the class and the longer its execution time becomes. Due to the ongoing adaption, the scheduling algorithm is called Adaptive Scheduling.

The dependency on the topmost data items in the Class Buffers is a hard limitation of the Scheduler's freedom of choice. However, at this point the usage of object-orientation (especially of classes and instances) helps to avoid a performance problem. Although a data item may force a reconfiguration (e.g. because it is heading for instance $B2$ which is currently swapped out), the corresponding reconfiguration process can be done very quickly. This is caused by the fact that the change from one instance of a class to another instance of the same class only requires a change of the context. Since this only affects some BRAMs, it requires much less time than a full reconfiguration. Therefore it is called

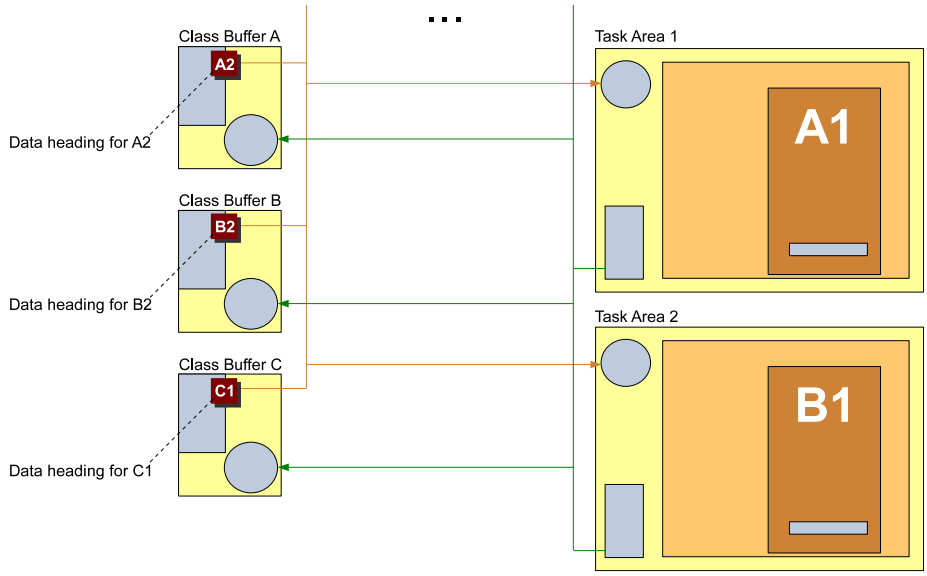


Figure 6.10: Scheduling example with 3 Class Buffers and 2 Task Areas (*System InOut* is not shown): Task Area 1 currently contains instance A1 of class A, while Task Area 2 currently contains instance B1 of class B. Based on the data coming out of the Class Buffers the Scheduler can perform:
- a Short Reconfiguration on Task Area 1 to load instance A2
- a Short Reconfiguration on Task Area 2 to load instance B2
- a Long Reconfiguration on Task Area 1 to load C1 or B2
- a Long Reconfiguration on Task Area 2 to load C1 or A2

Short Reconfiguration. In contrast, the change from an instance of class *A* to an instance of class *B* requires a full reconfiguration and is therefore called Long Reconfiguration. A Short Reconfiguration is approximately 10 times faster than a Long Reconfiguration. The concrete factor depends on the ratio of the size of the dynamic area to the size of the Context Memory.

If a data item is heading for an instance that does not exist, it cannot be processed at all. This erratic behavior can be caused by a connection to an instance that does not exist (any longer). Since such a defective data item would block the whole Class Buffer, the Scheduler removes it from the Class Buffer and reports an error. However, such errors always indicate a misuse of *connect*, *disconnect* or *finish()*.

6.3.4 Processor Subsystem

The Scheduler is running on an embedded processor and makes use of an external DDR-RAM, since the FPGA's internal BRAMs do not provide enough memory to store all the needed partial bitfiles and contexts. In order to implement this feature it is necessary to generate a processor subsystem which is part of the static design and provides a microprocessor, a DDR-RAM controller, a serial interface, an ICAP controller and a bus system which is interconnecting these components. To support the easy creation of such a processor subsystem, Xilinx provides the EDK (Embedded Development Kit). Here, a processor subsystem can be created on system level (even by drag and drop) by using pre-created IP Cores (see figure 6.11). Furthermore the EDK contains the SDK (Software Development Kit), which simplifies the process of writing software for the embedded processor. The Scheduler has been developed in the SDK.

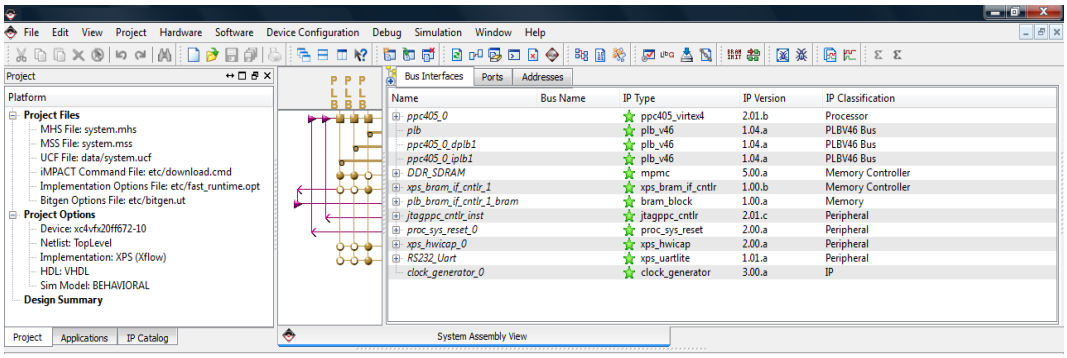


Figure 6.11: Screenshot of the EDK

6.4 POL

POL is used to describe the dynamic part of the target design. It was originally designed to be a subset of Java. During the development of the Framework it turned out, that such a strict approach leads to some serious problems. Reasons for this are, for example, the missing support of operator overloading in Java and the requirement for embedding subcomponents written in VHDL. Thus, the decision was made to slightly change the approach, so that POL is defined as a subset of Java with some minor extensions. Table 6.1 gives a quick overview of the features of POL.

Data types	void, int, Boolean, one-dimensional arrays, object references

Operators	`i++, ++i, -i, a[i], ?:, &&, || ,`
	`<<, >>, <<=, >>=, <, <=, >, >=, ==`
	`+, -, *, /, &, |, ^, =, ~, unary !,`
	`!=, +=, -=, &=, |=, ^=, *=, /=`

Variables	local and private member variables, register and heap based, Slots and Signals
Statements	if-branching, while-, do while-, and for- loop with break and continue, return, new and finish for classes and arrays, component
Functions	member functions and function calls, recursion

Table 6.1: Features of POL [170]

In order to be able to produce efficient hardware, some features of Java have been removed: POL does not support full inheritance, garbage collection and exceptions. Nevertheless, on the statement or expression level POL is similar to Java. [170]

6.4.1 Classes and Interfaces

Every hardware module that is connected to the Communication Matrix is represented by a POL class. The Framework supports two models of classes: dynamic classes which can be instantiated, scheduled, and deleted — and static classes which are singletons. Their base class determines the different models: a class that derives from *ParObj* is dynamic, while a class that derives from *StatObj* is static. Additionally there has to be exactly one class that is derived from *DispObj*. This class is equal to those derived from *ParObj* except that it is automatically instantiated by the Scheduler at start-up.

Each POL class always contains a calc() method which contains the actual functionality. calc() contains usual Java code (as shown in table 6.1) and is translated to VHDL by the POL-Compiler. To represent the characteristics of hardware (continuous execution), the calc() method runs in a loop (in other words: it is automatically called again after it has finished). It is of type void and has no arguments. It can call other methods, but cannot be called by other methods. The creation of a POL instance (via *new*) automatically leads to the start of calc().

Classes which share a common set of *Slots* and *Signals* can make use of a common interface via the keyword *implements*. Interfaces contain no member variables or method signatures, since methods and variables inside classes are always private.

```
interface Arithmetic {
   Slot:   int in1, in2;
   Signal: int out;
}

class Adder extends ParObj implements Arithmetic {
   private: int res = 0;
   void calc() {
     res = in1.get() + in2.get();
     out.emit(res);
   }
}
```

Figure 6.12: POL example using classes and interfaces

In the following, the syntax of POL is described by a regular expression variant of EBNF (Extended Backus Nauer-Form [171]). For better reading, production rules are printed in *italics* and terminal strings are underlined. Table 6.2 illustrates the general syntax of a POL program.

program	=	*include** (*class* | *interface*)+
include	=	include " *filename* "
filename	=	*printable_character* \ "
class	=	class *identifier* extends *identifier* (implements *identifier*)?
		{ (*class* | *member_decl* | *method* | *constructor*)* }
interface	=	interface *identifier* { *member_decl* }
member_decl	=	*modifier*: (*con_token*
		| *final_token*? *type var_decl* (, *var_decl*)*)
modifier	=	private | Slot | Signal
local_decl	=	*final_token*? *type var_decl* (, *var_decl*)*)
var_decl	=	*identifier* (= *assignm_expr*)?
method	=	*type identifier*
		(*parameters*?) *block*
parameters	=	*parameter* (, *parameter*)*
parameter	=	*final_token*? *type identifier*
constructor	=	*identifier* ( ) *block*
con_token	=	con
final_token	=	final

Table 6.2: The syntax of a POL program [170] — *program* is the start rule, statements and expressions are defined in the following sections

Dynamic Creation and Destruction

When the generated system on the FPGA starts up, the first objects which exist physically on the chip are the static classes (including *System InOut*). The first class that is instantiated dynamically is the one derived from *DispObj*. Other objects, which are derived from *ParObj*, are instantiated at runtime via *new*. If the POL code is executed in software (in the Emulator) the creation of a new instance leads to the immediate execution of its calc() method. If the POL code is executed in hardware (in other words: if it has been translated to partial bitfiles) the creation of a new instance at first leads to a new entry in the Scheduler's instance list. At a later point in time the Scheduler can decide to swap in this instance.

To destruct an object the keyword *finish* followed by a reference to the object is used. The *new* statement works synchronously since it has to return the reference of the new instance. The *finish* statement is handled asynchronously since it just generates a finish message (see section 6.3.2) and returns immediately.

```
class main extends DispObj {
   Slot: int in1;
   private:
     int key;
     Adder A;
   void calc() {
      key = in1.get(0);
      if (key==1) A = new Adder();
      if (key==2) finish A;
   }
}
```

Figure 6.13: Dynamic instantiation and destruction

Arrays

From an abstract point of view, arrays are just objects and could be handled like *ParObj*. However, since a partial reconfiguration is very time consuming and the bandwidth between several hardware objects is limited, it makes sense, to distinguish between hardware objects which represent functionality (like the *Adder* in figure 6.13) and objects which are only storing data (like arrays). Due to this, the Framework handles arrays different than hardware objects. Arrays are stored in the heap of an instance, which means they are stored in the instance's context memory (which is a BRAM). The only similarities between arrays and hardware objects are the keywords for creation and deletion: an array is instantiated via *new* and destroyed via *finish*. POL only supports arrays with a fixed size. An array can contain primitive data types or references to POL objects. They provide the method *length()* which returns the size of the array. Since the Scheduler on the FPGA does not provide a garbage collector, the developer has to ensure that every array instance is deleted when it is no longer used. However, if an instance is destroyed (via *finish*) the corresponding arrays are destroyed as well (since the heap is part of the context which is deleted after destroying an instance). *Slots* and *Signals* are allowed to be of type integer array. In this case, the Communication Matrix sends the data entries of the array one after the other. Arrays which have been received from other objects cause an implicit *new*. They have to be deleted explicitly.

6.4.2 Communication

The basic idea of POL is to force the programmer to use multiple objects running in parallel and well-defined communication channels which can be translated to hardware without using a shared memory. Due to this the strongest restrictions of POL affect the inter-module communication and the access control: POL does not offer *public* and *protected* access at all. The only way to communicate with a hardware object is though its *Signals* and *Slots* (which can be seen as public). All other variables and methods are private. POL knows 3 access modifiers:*Signal*, *Slot* and *private*. They are followed by a colon and applied to all following members until another access modifier occurs or the class

```
class main extends DispObj {
   Slot: int index, value, go;
   Signal: int[] out;
   private:
     int[] array = new int[15];

   main() {
     filter fi = new filter();
     this.out.connect(fi.in);
   }
   void calc() {
     while(this.out.connect.con_count()==0);
     array[index.get()] = value.get();
     if (go.get(0)==1) out.emit(out);
   }
}

class filter extends ParObj {
    Slot: int[] in;
    Signal: int[] out;
    private: int[] data;
    void calc() {
       data = in.get();
       //...FILTER...
       out.emit(data);
       finish data;
    }
}
```

Figure 6.14: Arrays in POL

declaration ends. POL supports only 4 data types: integer (*int*), Boolean (*Boolean*), arrays and object-handlers. These 4 types can be used for private variables as well as for *Signals* and *Slots*.

Sending and Receiving Data

To transfer data between hardware objects, each object makes use of its own *Signals* and *Slots*. For this, each *Signal* provides the method *emit()*, which sends a data item to it. The syntax is: *Signal.emit*(*data*). At this, *data* has to be of the same type as the *Signal*. Receiving data is a little more complex, since it can be done in two ways: blocking and non-blocking. A blocking *get()* stops the execution of *calc()* until the corresponding *Slot* contains at least one data item that can be received and returned. A non-blocking *get(default_value)* returns the next data item of the corresponding *Slot*. If the *Slot* does not contain any data, it returns the *default_value*. In addition to blocking and non-blocking *get* methods, POL provides the method *valid()* which returns a Boolean indicating if the corresponding *Slot* does contain data or not.

Connections

Each object only accesses its own *Signals* and *Slots* to send and to receive data. However, at some point the *Signals* and *Slots* have to be connected to each other, so that the data written to Signal x of instance A reaches Slot y of instance B. For this, the method *connect()* is used. It is a method that belongs to each *Signal*. The syntax is: $A.x.connect(B.y)$. The *connect()* method can be used by each object as long as it has the necessary object handler for A and for B. The connected *Signal* and *Slot* have to be of the same type. As usual in Java every object can refer to itself using the keyword *this*. POL allows multiple *Slots* to be connected to one single *Signal*. So, if two *Slots* are connected to one *Signal*, a data item that is written to this *Signal* is send two times (with two different target addresses).

Connections can be removed from a *Signal* using the method *disconnect()* which also belongs to each *Signal*. Its syntax is equal to that of *connect()*. If a *Signal* is not connected to any *Slot* at all, data that is written to this *Signal* is discarded. To avoid data loss caused by a delayed establishment of a connection, each *Signal* provides the method *con_count()*, which returns the number of *Slots* that are connected to it.

Syntax	Argument Type	Return Value Type
Slot.get()	-	message type
Slot.get(default)	message type	message type
Slot.valid()	-	Boolean
Signal.emit(data)	message type	void
Signal.connect(input)	input to connect to	void
Signal.disconnect(input)	input to disconnect from	void
Signal.con_count()	-	int

Table 6.3: Methods of Signals and Slots[170]

The information that a *Signal* shall be connected to a *Slot* (via *connect()*) has to reach the instance this *Signal* belongs to. Theoretically, the DCB could be used for this but this would lead to an asynchronous relation between the actual data and the Connection Messages. Due to this, the decision was made to realize the connection commands as standard messages which are delivered by the Communication Matrix. To distinguish between data messages and Connection Messages, the Connection Messages use a special Slot ID. The corresponding *Slot* is called connection *Slot* and belongs to each hardware class. The Destruction Messages shown in figure 6.9 are a special kind of these Connection Messages.

In section 6.2.3 the Instance Buffer has been shown. This buffer is used to parallelize several *Slots*. Theoretically it can be used to generate a fully parallelized input where each *Slot* has its own message queue but this behavior is not always desired. Let's assume a filter consisting of two sub-filters *A* and *B*, which both are realized as separate hardware classes. *A* sends processed data to *B* via Slot 1 and control messages via Slot 2. During processing the conditions change and *A* now wants *B* to change its functionality. For this, *A* uses control messages. These control messages on Slot 2 must **not** be able to overtake the data messages on Slot 1, since this would unintentionally change the way older data is processed. So, *Slot* 1 and *Slot* 2 shall be synchronized. In other words: they shall share one single message queue inside the Instance Buffer.

To be able to express such a need for synchronization, POL distinguishes between *Slots* which are declared in one single line of code and *Slots* which are declared in individual lines of code. *Slots* that share one line of code also share one message queue.

To be able to synchronize a set of *Slots* with the connection *Slot*, POL provides the keyword *con* which stands for the connection *Slot*. Figure 6.15 illustrates the implementation of *Filter_B*. The *Slots sample*, *control1*, and the connection *Slot* (represented by *con*) are synchronized to each other and therefore share one single message queue of the Instance Buffer. *Slot control2* is not synchronized to the other *Slots* and has its own message queue. Thus, data items send to *control2* are able to overtake data that is send to *sample*, while data items send to *control1* are not.

```
class Filter_B extends ParObj {
   Slot: int sample; int control1; con;
   Slot: int control2;
   ...
}
```

Figure 6.15: Parallel and Synchronized Slots in POL

System InOut / world

The description of the static design and the interconnections to the outside world will be described in the next section (which focuses on the JSB). However, the POL classes have to be able to communicate with this outside world. For this, a special class named *world* provides inputs and outputs, which can be used to connect the program with the surrounding system. *world* refers to *System InOut* which is provided by the Communication Matrix (see chapter 6.2.3). Figure 6.16 illustrates the usage of *world*.

```
class Adder extends DispObj {
   Slot: int in1;
   Slot: int in2;
   Signal: int out;

   Adder() {
     this.out.connect(world.out1);
     world.in1.connect(this.in1);
     world.in2.connect(this.in2);
   }

   void calc() {
     out.emit(in1.get() + in2.get());
   }
}
```

Figure 6.16: Class *world* and how it is used to communicate with the outside world

6.4.3 Embedded VHDL components

To support the reuse of existing VHDL components and the manually optimized implementation of subroutines, POL supports the usage of VHDL components. Figure 6.17 shows an example of how to use a custom VHDL entity that calculates the greatest common divisor of *a* and *b*. The result is stored in the variable *result*. The VHDL entity has two input ports: *in1* and *in2*. Furthermore, it has one output port: *out1*. The first argument of the statement is the name of the VHDL component. Second, the mapping between VHDL input signals and POL expressions is defined. The last mapping denotes which VHDL output signal is connected to which POL variable. To be able to use VHDL components, a POL program has to begin with the *include* directive.

```
include "gcd_vhdl.vhd";

class gcd extends ParObj {
   Slot: int i1;
   Slot: int i2;
   Signal: int o1;
   private: int a, b, result;
   void calc() {
     a = i1.get();
     b = i2.get();
     component(gcd_vhdl : in1 => a, in2 => b : out => result);
     o1.emit(result);
   }
}
```

Figure 6.17: Using VHDL subcomponents

6.4.4 Statements

On statement level, POL is very similar to Java. Differences are POL's additional *component* statement, no support for the *switch* statement, and no support for exceptions. Table 6.4 illustrates the syntax of POL statements.

statement	=	*local_decl* ;
	|	*stmnt_expr* ;
	|	*while_stmnt*
	|	*do_stmnt*
	|	*for_stmnt*
	|	*if_stmnt*
	|	*return_stmnt* ;
	|	*component* ;
	|	*block*
	|	*empty_stmnt*
while_stmnt	=	while (*assignm_expr*) *statement*
do_stmnt	=	do *statement* while (*assignm_expr*)
for_stmnt	=	for (*for_init* ; *assignm_expr* ; *expr_list*) *statement*
for_init	=	*expr_list*
	|	*local_decl*
expr_list	=	*assignm_expr* (, *assignm_expr*)*
if_stmnt	=	if (*assignm_expr*) *statement* (else *statement*)?
return_stmnt	=	return (*assignm_expr*)?
component	=	component (*identifier* : *portmap_in* : *portmap_out*)
portmap_in	=	*port_in* (, *port_in*)*
portmap_out	=	*port_out* (, *port_out*)*
port_in	=	*identifier* => *assignm_expr*
port_out	=	*identifier* => *identifier*
block	=	{ *statement** }
empty_stmnt	=	;

Table 6.4: The syntax of POL statements [170] — for declaration of expressions see next section

6.4.5 Expressions

Table 6.5 lists all available expressions in the order of the precedence of their operators.

stmnt_expr	=	*assignment*
	|	*pre_incr_expr*
	|	*post_incr_expr*
	|	*method_invoc*
	|	*new_inst*
	|	*finish*
assignment	=	*array_access assign_op assignm_expr*
	|	*identifier assign_op assignm_expr*
assignm_expr	=	*assignment*
	|	*cond_expr*
cond_expr	=	*cond_or_expr* (? *assignm_expr* : *cond_expr*)?
cond_or_expr	=	*cond_and_expr* (|| *cond_or_expr*)?
cond_and_expr	=	*incl_or_expr* (&& *cond_and_expr*)?
incl_or_expr	=	*excl_or_expr* (| *incl_or_expr*)?
excl_or_expr	=	*and_expr* (^ *excl_or_expr*)?
and_expr	=	*equ_expr* (& *and_expr*)?
equ_expr	=	*rel_expr* ((== | !=) *rel_expr*)?
rel_expr	=	*add_expr* (*rel_op add_expr*)?
shift_expr	=	*add_expr* ((« | ») *add_expr*)*
add_expr	=	*mult_expr* ((+ | -) *mut_expr*)*
mult_expr	=	*unary_expr* ((* | /) *unary_expr*)*
unary_expr	=	*preincr_expr*
	|	(+ | -) *unary_expr*
	|	*unarynpm_expr*
preincr_expr	=	*incr_op unary_expr*
unarynpm_expr	=	(! | ~) *unary_expr*
	|	*postfix_expr*
postincr_expr	=	(*primary* | *expr_name*) +*incr_op*
postfix_expr	=	*postincr_expr*
	=	*primary*
	=	*expr_name*
primary	=	*new_array*
	|	*primary_nna*
new_array	=	new (*prim_type* | *ref_type* [*assignm_expr*]
primary_nna	=	*literal*
	|	*this_lit*
	|	(*assignm_expr*)
	|	*method_invoc*

	|	*array_access*
	|	*new_inst*
	|	*finish*
array_access	=	*identifier* [*assignm_expr*]
new_inst	=	new *identifier* (( ))?
finish	=	finish *assignm_expr*
method_invoc	=	*identifier* (. *identifier*)* (*arguments*?)
arguments	=	*assignm_expr* (, *assignm_expr*)*
type	=	*array_type* | *prim_type* | *ref_type*
prim_type	=	void | int | boolean
ref_type	=	*identifier*
array_type	=	(*prim_type* | *ref_type*) [ ]
incr_op	=	++ | --
assign_op	=	= | *= | /= | += | -=
rel_op	=	< | <= | > | >=
expr_name	=	*identifier* (. *identifier*)?
identifier	=	*letter* (*letter* | *digit* | _)*
		\ (new | finish | do | while | for | class | interface | if | else | return
		| component | break | continue | extends | include | true
		| false | null | con | this | final |int | boolean | break
		| continue | extends | include | true | false | null | con
		| this | final)
this_lit	=	this
literal	=	*int_lit* | *bool_lit* | *null_lit*
int_lit	=	-? *digit*+
bool_lit	=	true | false
null_lit	=	null
digit	=	[0 - 9]
letter	=	[a - z] | [A - Z]

Table 6.5: The syntax of POL expressions [170]

6.4.6 From Parallel Object Language to VHDL

The POL-Compiler is responsible for the translation from POL to VHDL. Its capabilities determine which features are supported by POL. As mentioned before, this compiler directly translates the POL code to VHDL without using an intermediate step like converting POL to Java bytecode. The concept is to directly transfer the software concurrency (described in Java via threads) to hardware concurrency (described in VHDL via processes). Therefore, each hardware object (namely *ParObj*, *DispObj*, and *StatObj*) is translated directly to a separate VHDL entity. Moreover, classes derived from *ParObj* or *DispObj* are used to generate VHDL entities which serve as basis for partial bitfiles — in other words: for dynamic instantiable hardware. All hardware objects are interconnected by the Communication Matrix which serves as additional layer between automatically generated hardware modules and the FPGA resources.

The POL-Compiler translates *calc()* to one single FSM (Finite State Machine). Each calculation is represented by one state of this FSM. Some optimizations are done (e.g. state merging based on data dependency analysis), but the focus concentrates on the dynamic instantiation, swap-in, swap-out and the dynamic inter-object communication.

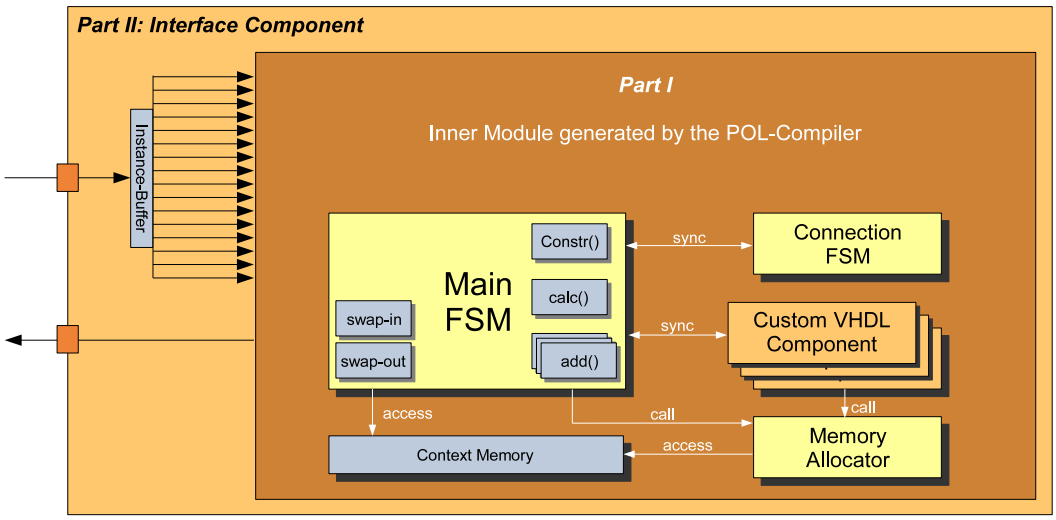


Figure 6.18: Basic Structure of the Generated VHDL Code

Figure 6.18 illustrates the basic structure of the generated VHDL entities, each representing one hardware class. The sequential statements inside the constructor and inside *calc()* are translated to states of the Main FSM. Additional methods called by *calc()* also are translated to states of the Main FSM. Furthermore, the Main FSM contains states responsible for swap-in and swap-out. For swap-out the context is stored in the context memory, for swap-in it is restored from the context memory. The context also contains the current state. This way, the Main FSM can continue to calculate exactly at the point where it has been interrupted.

The Connection FSM is responsible for incoming Connection Messages. It is running in parallel to the Main FSM. Nevertheless the Main FSM and the Connection FSM are synchronized. The Memory Allocator is a handwritten VHDL subcomponent that is responsible for memory allocation in the Context Memory. It manages used and free space in the stack as well as in the heap. The creation and the destruction of an array is realized via the Memory Allocator. If additional VHDL components have been included (using *include* and *component*), these customized VHDL components are also part of the module. A special interface between each VHDL subcomponent and the Memory Allocator allows the subcomponents to store intermediate data in the context memory. This is the only way to safely store intermediate data. Other data, stored in flip-flops etc., is lost during swap-out and swap-in. Please note that the VHDL submodules are not running continuously but are only active when they are called by the Main FSM. More details about the concrete implementation of the POL-Compiler and the generated hardware can be found in chapter 7.

6.5 Merger

In principle, the different parts of the Framework (processor subdesign, POL-Compiler, Communication Matrix and Scheduler) act as independent as possible. However, at some point they need to come together to exchange crucial design parameters. The POL compiler needs to transport information to the Communication Matrix as well as to the Scheduler:

1. Number of classes
2. Mapping between partial bitfile and class ID
3. Type of the classes (*ParObj*, *StatObj* or *DispObj*)

For this, the POL-Compiler creates a configuration file ("task.cfg") containing the relevant attributes of the hardware classes. Figure 6.19 shows an example configuration with two dynamic classes and one dispatcher task.

```
task: filter
   id: 1
   type: dynamic
task: gcd
   id: 2
   type: dynamic
task: main
   id: 3
   type: dispatcher
```

Figure 6.19: Syntax of "task.cfg"

At this point the Merger comes into play. It reads the "task.cfg" and changes files that are used by the Communication Matrix and by the Scheduler. For the Scheduler it creates a file called "pol.h" which contains the number of classes and the class id of the class deriving from *DispObj*. The parameterization of the Communication Matrix is more complex, since it requires the change of several VHDL-Files. For this, predefined VHDL template files ("*filename*.vhdt") are used. These template files contain so called magic comments (syntax: *--POL:command*) which are replaced by the Merger with the correct number and type (dynamic or static) of component declarations and component instantiations (such as Class Buffers or Task Areas). Furthermore, a parameter file ("matrix_components.vhd") is created, which contains the number of classes and the size of the used vectors (such as class ID or instance ID). More details about the Merger and the generated files as well as the corresponding implementation runs can be found in chapter 7.

6.6 Emulator

Since POL is an enriched synthesizable subset of Java, it can be seen as both HDL and software language. Thus, the easiest way to verify the correctness of POL source code is to execute it in software. The Emulator realizes this functionality. POL mainly consists of Java, extended by new communication mechanisms such as *Slot*, *Signal*, *connect*, and *disconnect*. After the translation from POL to VHDL, all communication is buffered by FIFOs (by the Communication Matrix), which make shure that a running object can read input data and write output data even if sender or receiver are currently swapped out. The aim of the Emulator is to resemble this functionality of the Communication Matrix, so that the execution in software and the execution in hardware are identical regarding messages and their delivery. Figure 6.20 shows the different layers of hardware and software execution. The Communication Matrix serves as an additional layer between the generated modules and the actual FPGA hardware. It enables the hardware objects to make use of the message protocol. The Emulator provides the same abstraction layer in software. The exact timing of the underlying layers (JVM and CPU) is completely different from the timing of the FPGA but on message layer both ways of execution are identical. Due to this, verification of POL code in software is sufficient to qualify its functionality.

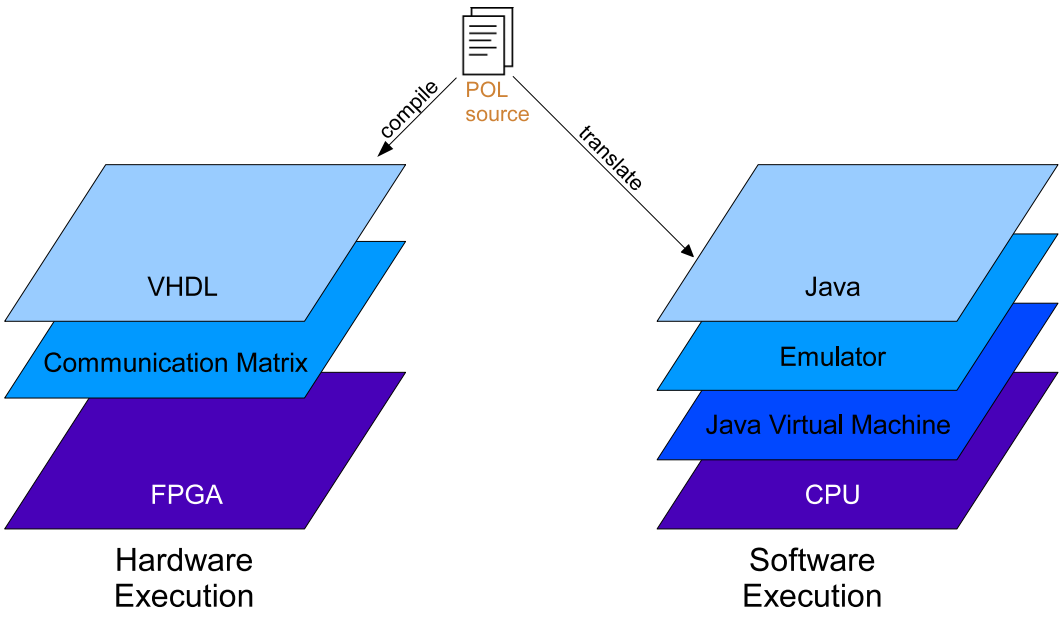


Figure 6.20: Comparison of the software and the hardware layers[172]

6.6.1 The Precompiler

In its first version, POL has been defined to be a strict subset of Java. However, at some points this limitation was too restrictive. Thus, the decision was made to introduce some minor extensions which make POL more powerful and convenient but keep it similar to Java. The extensions are realized via the Precompiler, which translates POL to usual Java, before it can be executed in software. Table 6.6 illustrates the extensions and how they are replaced by the Precompiler. The two most important extensions are operator overloading (needed to support 16-bit integers and not supported by Java) and the access modifiers *Slot* and *Signal*.

Extension	POL	Java

Operator Overloading (for *n*-bit integers)	int a = 40;
	
	a++;
		IntWrapper a
		= new IntWrapper(40);
		post_inc(a);

Access Modifiers	Slot: int a; Boolean b; Signal: Filter f;	Slot<Integer> a = new Slot<Integer>(this); Slot<Boolean> b = new Slot<Boolean>(this); Signal f = new Signal(this);

Table 6.6: The Precompiler's Translation Table from POL to Java (Excerpt)

6.6.2 Emulation vs. Simulation

In the following the difference between simulation and emulation shall be clarified.[172]
Simulation: A computer simulation tries to model a real-life or a theoretical situation so that it can be studied to gain insides into the simulated system. A simulation usually represents certain key characteristics or behaviors of a selected system. The aim of simulation is to take a closer look at the *internal* behavior of the simulated system.
Emulation: An emulator recreates the functionality of a system so that it behaves like the emulated system. The aim of emulation is the exact reproduction of the *external* behavior of the emulated system.

Since the aim of the software execution of POL is to recreate the functionality of the Communication Matrix on message layer, the corresponding process is more an emulation than a simulation. The Emulator does not provide a step by step execution as known from some simulators. Nevertheless, the generated Java code can be debugged as usual Java code. For this purpose it is possible to introduce non-synthesizable Java constructs (like exceptions or console messages) for development and debugging. Of course, these elements have to be removed, before the POL-Compiler is used to generate VHDL code. This method is comparable to the different MOCs[3] in SystemC or to the usage of non-synthesizable constructs like *wait for 40ns;* in VHDL.

[3]Model Of Computation — see chapter 3.3.5

6.6.3 Emulation of the Static Part of the Design

For the emulation of POL code a complete system has to be defined. This means that besides the POL code itself, devices for input and output are required. Thus, the Emulator contains classes that are used to represent the surrounding system (e.g. *FileInput*, *FileOutput*, *UART* or *PushButtons*). They are instantiated in a file called *JSB.java* which also instantiates a class representing the Scheduler and the POL class deriving from *DispObj*. The layout of *JSB.java* is inspired by the Java System Builder (JSB) [173].

Figure 6.21 shows an example of how the Emulator is used to connect the dispatcher object to a file reader input and a file writer output. Some simulation components like push-buttons or a file reader/writer are already included in the Emulator, other customized devices can be added easily (using usual Java).

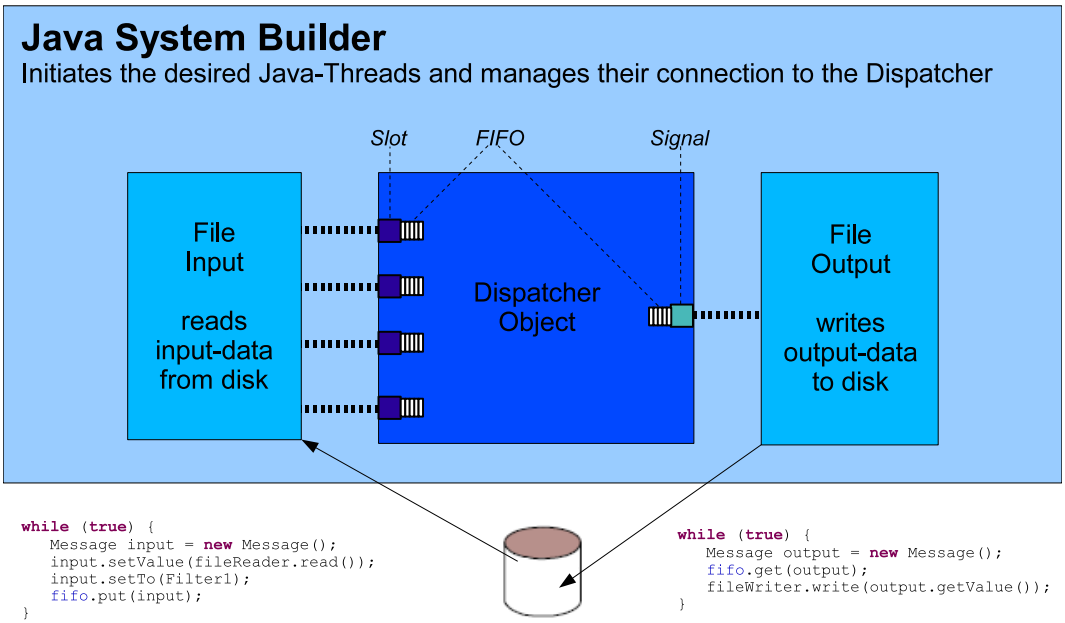


Figure 6.21: Interaction between JSB and Emulator[172]

More details about the Emulator including an example implementation and the corresponding screen shots are shown in the following chapter 7.

7 Implementation of the Framework

In chapter 4 the basic approach has been shown. It has been illustrated that object-orientation in combination with multi-threading is a very good way to describe dynamic hardware. As a consequence, the language POL which is very close to Java has been designed. Chapter 5 described the resulting requirements coming with the specification of POL. In chapter 6 the resulting design of the Framework which contains an Emulator, a POL-to-VHDL compiler as well as the necessary infrastructure on the FPGA has been presented.

This chapter concludes the description of the Framework. It focuses on the concrete implementation of the Framework and describes the way the design has been realized.

Implementation Environment

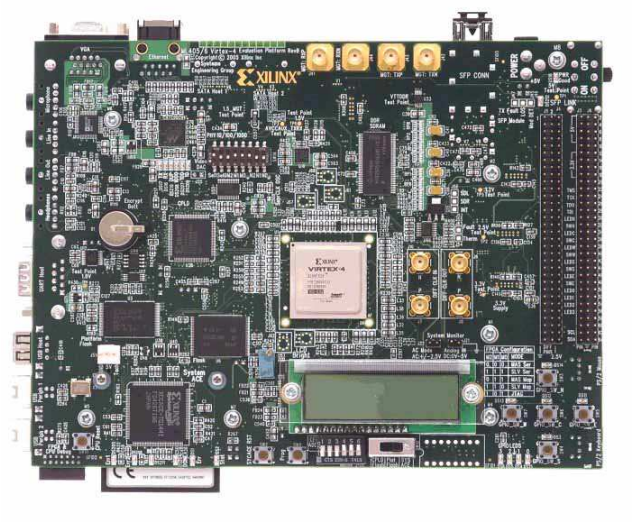


Figure 7.1: ML405

The implementation of the POL-Compiler, the Communication Matrix and the Scheduler was performed on a PC equipped with a 2.5 GHz single-core Intel processor and 4 GB memory, which was running 32-bit Kubuntu Linux. For bitfile generation, the Xilinx tools ISE 9.1i SP2, enhanced by the PREA[1] patch, and *PlanAhead* 10.1i were used. The processor subdesign as well as the Scheduler were developed using the Xilinx EDK 9.1i SP2.

[1] Partial Reconfiguration Early Access

For simulation, Modelsim 6.5 was used. The Emulator was developed on an Intel-PC running 32-bit Windows Vista, using Eclipse Europa (version 3.3.0) and JDK[2] version 6.0.

For testing of the generated bitfiles, the Xilinx Evaluation Board ML405 containing a Virtex-4 FX20 (package FF672, speed-grade 10) was used (see figure 7.1). The Board inter alia contains a 100 MHz oscillator, an RS-232 serial port, an AC97[3] chip which provides audio inputs and outputs, a VGA chip with a monitor interface, push buttons and LEDs [174]. The FPGA contains 8 544 Slices, 68 BRAMs, one PPC[4] and two ICAPs.

Implementation Example

To increase the transparency of this chapter, an example implementation is used as a leitmotif. It consists of 3 classes: *Adder*, *Multiplier* and *Dispatcher*. The *Adder* takes its input data from two *Slots*, adds the received values and emits them to its *Signal*. The *Multiplier* also takes its input data from two *Slots*, multiplies the received values and emits them to its *Signal*. The *Dispatcher* creates two instances of *Adder* and one instance of *Multiplier*. It connects them as shown in figure 7.2. The left part of figure 7.3 shows the complete POL source code used for the adder-multiplier example.

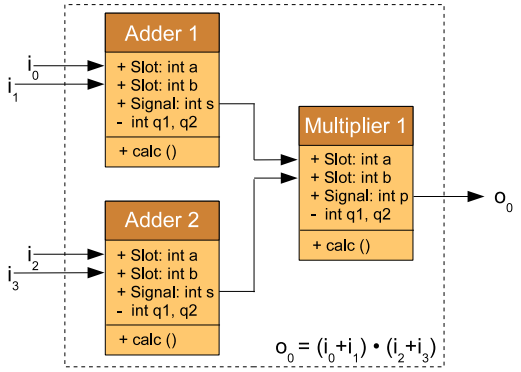


Figure 7.2: Dataflow of the adder-multiplier example

[2]Java Development Kit
[3]Audio Codec '97
[4]PowerPC

POL

```
class Adder extends ParObj {
  Slot:    int a;
           int b;
  Signal:  int s;
  private: int q1;
           int q2;
  void calc() {
    q1=a.get();
    q2=b.get();
    s.emit(q1 + q2);
  }
}

class Multiplier extends ParObj {
  Slot:    int a;
           int b;
  Signal:  int p;
  private: int q1;
           int q2;
  void calc() {
    q1=a.get();
    q2=b.get();
    p.emit(q1 * q2);
  }
}

class Dispatcher extends DispObj {
   Adder A1 = new Adder();
   Adder A2 = new Adder();
   Multiplier M1 = new Multiplier();
   AdderMultiplier() {
       world.in0.connect(A1.a);
       world.in1.connect(A1.b);
       world.in2.connect(A2.a);
       world.in3.connect(A2.b);
       A1.s.connect(M1.a);
       A2.s.connect(M1.b);
       M1.p.connect(world.out0);
   }
   void calc() {}
}
```

Java

```
package application;
import pol.*;

class Adder extends ParObj {
   Slot<Integer> a = new Slot<Integer>(this);
   Slot<Integer> b = new Slot<Integer>(this);
   Signal s = new Signal(this);
   IntWrapper q1 = new IntWrapper();
   IntWrapper q2 = new IntWrapper();
   public void calc() {
      q1.value = a.get();
      q2.value = b.get();
      s.emit(uint16b(q1.value,"+",q2.value));
   }
}

class Multiplier extends ParObj {
   Slot<Integer> a = new Slot<Integer>(this);
   Slot<Integer> b = new Slot<Integer>(this);
   Signal p = new Signal(this);
   IntWrapper q1 = new IntWrapper();
   IntWrapper q2 = new IntWrapper();
   public void calc() {
      q1.value = a.get();
      q2.value = b.get();
      p.emit(uint16b(q1.value,"*",q2.value));
   }
}

class AdderMultiplier extends DispObj {
   Adder A1 = new Adder();
   Adder A2 = new Adder();
   Multiplier M1 = new Multiplier();
   public AdderMultiplier() {
      In[0].connect(A1.a);
      In[1].connect(A1.b);
      In[2].connect(A2.a);
      In[3].connect(A2.b);
      A1.s.connect(M1.a);
      A2.s.connect(M1.b);
      M1.p.connect(Out[0]);
   }
   public void calc() {}
}
```

Figure 7.3: Comparison of POL and the corresponding generated Java code

7.1 Emulator

After creating the POL sources, a user of the Framework can execute the console command *pol* to parse and to compile the POL sources. Since parsing is needed for both the Precompiler and the POL-Compiler, these two components have been implemented in one tool. Thus, invoking *pol* creates both the VHDL code[5] and the Java code. The right part of figure 7.3 shows the Java code that has been generated by the Precompiler. As one can see, the POL specific access modifiers have been replaced by classes called *Signal* and *Slot*. The different types of *Slots* were implemented using Java Generics. Furthermore, the Precompiler introduces a class called *IntWrapper*. This class is used to support 16-bit Integers, which is necessary, since the current implementation of the Framework uses a data format with a payload size of 16 bit. To be able to handle overflows correctly, operations such as an addition or multiplication have to be realized via a special method called *uint16b*. Please note that in the current Emulator implementation the emulation of VHDL subcomponents is not yet implemented.

[5]The parsing process as well as the compile process will be illuminated in detail in section 7.2

7.1.1 Emulator Packages

Package *application*

The package *application* contains the Java files that can and/or must be edited by the user of the Framework. *JSB.java* contains the representation of the surrounding system and thus the static *main* method. *Constants.java* contains a list of constants which determine the output and debug level of the Emulator. Finally, *toEmulate.java* contains the Java code generated by the Precompiler, while the name *toEmulate.java* can be interchanged freely. Figure 7.4 shows an example representation that connects the adder-multiplier example with a UART[6] interface.

```
01 package application;
02 import pol.*;
03 import components.*;
04
05 public class JSB {
06    public static void main(String[] args) {
07       Dispatcher disp1 = new Dispatcher();
08
09       Control control = new Control();
10       control.getControlDevice(Constants.usedOutputs);
11
12       UART uart = new UART();
13       uart.getInputDevice(POL.In[0].myFifo, 0);
14       uart.getInputDevice(POL.In[1].myFifo, 1);
15       uart.getInputDevice(POL.In[2].myFifo, 2);
16       uart.getInputDevice(POL.In[3].myFifo, 3);
17       uart.getOutputDevice(POL.Out[0].myFifo, 0);
18
19       disp1.go();
20    }
21 }
```

Figure 7.4: Example implementation of JSB.java

Line 07 starts the initialization of the *Dispatcher*, while *Dispatcher* is the name of the POL class derived from *DispObj*. The name *Dispatcher* depends on the given POL code. Lines 09 and 10 create and start the *Control* thread which is used to emulate the Scheduler. Line 13 to 16 connect the inputs *In[0]* to *In[3]* of the POL world with the UART component created in line 12, while line 17 connects the output *Out[0]* with this UART component (the class *POL* is a static class which is used to provide the connections between the POL world and the surrounding system). Finally, line 19 starts the emulation.

[6]Universal Asynchronous Receiver Transmitter

Package *components*

The package *components* contains all classes that are used by the Emulator to represent the surrounding system. Examples are *UART* or *PushButtons*. Furthermore this package holds all classes that are needed to emulate the behavior of the Scheduler. The class *UART* shown in figure 7.4 is a typical example for a class that belongs to the package *components*. It provides the methods *getInputDevice* and *getOutputDevice* which provide an emulation of the surrounding system and have to be directly connected to the inputs and outputs which have been described in POL using the class *world*. The class *UART* creates a GUI (Graphical User Interface) for each input and for each output. Figure 7.5 shows a screenshot of the graphical output resulting from the Java code presented in figure 7.4.

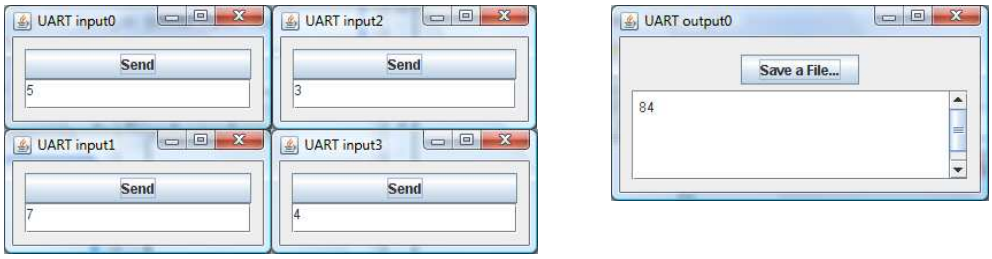


Figure 7.5: Screenshot of the emulation GUI provided by the Emulator class UART

Package *pol*

The Emulator makes use of inheritance and encapsulation to emulate the POL specific objects (such as *Slot*, *Signal*, *ParObj*, ...) and methods (such as *connect*, *disconnect*, *get*, ...). The corresponding classes are integrated in the package *pol*. Table 7.1 gives an overview of these classes and shows how they are used to emulate the functionality of the Communication Matrix.

Class	Description
ParObj	*ParObj* derives from *Thread* and is the heart of the Emulator. Attributes, functions and mechanisms typical for parallel running hardware modules are defined here. Thus, not only the POL objects, but also all objects representing a hardware module of the Communication Matrix (e.g. the Class Buffers) derive from *ParObj*. Due to this the emulation of POL code leads to the execution of dozens of threads.
FIFO	A *FIFO* is a linked list of type *Message*. It is used to represent the FIFOs of the Communication Matrix (e.g. the Task Area FIFO). Although Java provides the possibility to generate potentially infinite lists, each *FIFO* is limited to the correct size which complies with the Communication Matrix' FIFO size. *FIFO* derives from *ParObject*.
Slot	A *Slot* is the only input channel of the POL objects. It provides the methods *get()*, *get(default)* and *valid()*. To support the emulation of synchronized *Slots*, *Slots* can share one single *FIFO*. *Slot* derives from *ParObj*.
Signal	A *Signal* is the only output channel of the POL objects. It provides the methods *emit*, *connect* and *disconnect*. It derives from *ParObj*.
Message	Objects of type *Message* represent the messages which are delivered by the Communication Matrix. They are purely passive data-objects, which therefore do not derive from *ParObj* and are no thread at all. A *Message* contains the class ID, the instance ID and the Slot ID of the destination as well as the actual payload.
POL	*POL* is a static class that provides the connections between the Java code generated from the POL sources and the classes representing the surrounding static system. Since *DispObj* derives from *POL*, the class deriving from *DispObj* can access the global inputs and outputs via *In[x]* and *Out[y]*, while in the class *JSB.java* they are accessed via *POL.In[x]* and *POL.Out[y]*.
DispObj	*DispObj* represents the initial POL object which is automatically instantiated at system start-up. It derives from *POL* and is a singleton. *DispObj* instantiates *ClassFIFO* and provides the connections to the outside world.
ClassFIFO	As shown before, the Communication Matrix instantiates one buffer per class (the Class Buffer). The Emulator represents all Class Buffers with one single object called *ClassFIFO*. It is a singleton, which is automatically instantiated at system start-up by the dispatcher object. The parallelism of several Class Buffers is realized internally.

Table 7.1: The additional classes coming with the Emulator

Figure 7.6 illustrates the internal connection of the Emulator classes using the adder-multiplier example. It shows the *FIFO* objects used, as well as all used *Signals* and *Slots*. Please note that the inputs, outputs and the *ClassFIFO* are implicitly instantiating *Signals* and *Slots* which cannot be found in the original POL code. These additional objects are used to represent the functionality of the Communication Matrix. A user of the Framework does not have to care about them at all.

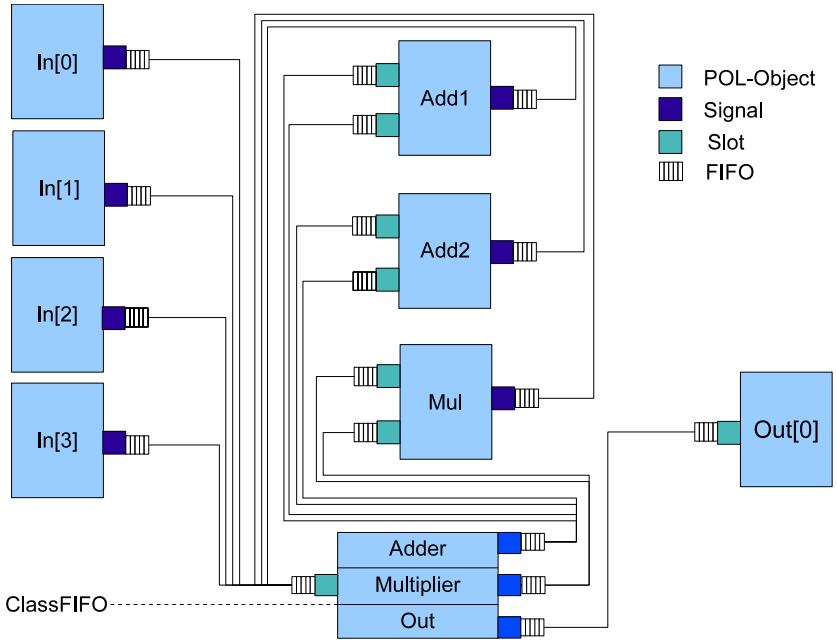


Figure 7.6: Connection diagram showing all instantiated Emulator classes [172]

7.1.2 Emulation of the Scheduler

In order to provide an exact emulation of the Communication Matrix, the Emulator has to recreate the behavior of the Scheduler. To that end, the class *Control* is used. It is a singleton and provides the scheduling algorithm, which determines whether an POL object is virtually loaded to a task area and thus allowed to work. To be able to trace the scheduling decisions and the state of the POL objects as well as the state of the *Signals*, *Slots* and their corresponding *FIFOs*, the *Control* object provides a GUI which displays the actual state of the system and allows the user to freeze single objects or the whole system.

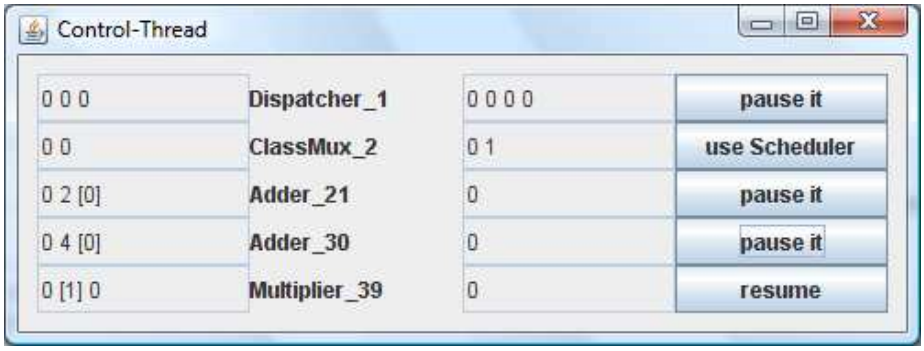


Figure 7.7: Screenshot of the Control GUI

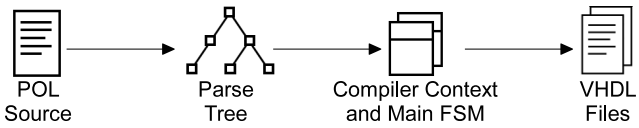
Figure 7.7 shows a screenshot of the *Control* GUI displaying 5 threads: the *Dispatcher*, the two *Adder* instances, the *Multiplier* and the *ClassFIFO* (called *ClassMux*). This output represents the current state of the emulated system and is updated whenever an instance is created or destroyed. Monitoring of the threads is realized via the Java method *Thread.enumerate* which fills a given *Thread* array with all threads that have been started by the current Java program. *Control* uses this list of threads and filters out all threads that do not represent a POL instance.

The numbers in the first column of the *Control* GUI represent the filling levels of the *Slots* of a thread, the numbers in the third column represent the filling levels of the *Signals* of a thread. Please note that each *Slot* and each *Signal* shown in figure 7.6 as well as the connection Slots are displayed.

The button “pause it” allows the user to freeze a single thread. The button “resume” continues a frozen thread. Finally, the button “use Scheduler” allows the user to activate the scheduler emulation, which automatically stops and restarts the threads as the Scheduler would do on the FPGA.

7.2 From Parallel Object Language to VHDL

The POL compiler translates the POL code to VHDL without using an intermediate step such as a conversion from POL to Java bytecode. The idea is to directly transfer the software concurrency (using threads) to hardware concurrency (using processes). The compiler is written in C++. The process of compilation is implemented in three steps. First, the syntactic structure of the POL code is analyzed by a parser, which results in a parsing tree. Secondly, the tree is recursively traversed and scanned for declared and instantiated variables, methods and other language elements. The outcome is the Main FSM and the so called Compiler Context which represents the needed declarations and definitions. Finally, the Main FSM and the Compiler Context are used generate the VHDL entities. [170]



7.2.1 The Parser

Instead of writing a proprietary parser, the decision was made to use the Spirit[175] parser framework which is part of the open-source Boost Libraries[176]. Spirit makes use of a set of predefined primitives, and overloads C++ built-in operators (+, *, >>) to provide a C++ representation of the grammar that is very close to EBNF[7]. Only some operator positions are different from those of the EBNF operators. For example, the unary operators + and * are prefix operators in Spirit while they are postfix operators in EBNF. Table 7.2 shows the operators in Spirit compared to their EBNF notation. Table 7.3 shows the parser primitives provided by Spirit. Figure 7.8 shows a C++ snippet using these operators and primitives to define the parser rule describing the syntax of a *while* loop.

Spirit operator	EBNF	Description
a = *b*	*a* = *b*	Production
a >> *b*	*a b*	Matches *a* followed by *b*
a | *b*	*a* | *b*	*a* or *b*
**a*	*a**	*a* zero or more times
+*a*	*a*+	*a* one or more times
a % *b*	*a* {*b a*}*	*a* one or more times, separated by *b*

Table 7.2: C++ operators overloaded by Spirit compared to EBNF [170]

[7] Extended Backus Nauer Form - see chapter 6.4.4

Primitive	Description
char_p('*x*')	a single character *x*
alpha_p	an alphabetical character
digit_p	a single digit
uint_p	an unsigned integer
print_p	any printable character
eol_p	a line break
end_p	end of input
str_p("*string*")	*string*
white_p	space, tabular or line break
comment_p("//")	a comment starting with '//' and ending with a line break
comment_p("/*", "*/")	a comment starting with '/*' and ending with '*/'

Table 7.3: Parser primitives provided by Spirit [170]

```
while_stmnt = str_p("while") >> *white_p >> ch_p('(') >> *white_p
              >> expression >> *white_p >> ch_p(')') >> *white_p >> statement;
```

Figure 7.8: Parser rule definition in Spirit [170]

POL does not depend semantically on spaces between keywords, identifiers, literals and operators. Denoting these possible spaces explicitly makes the definitions long and hard to read. Therefore, Spirit provides the possibility to define a so called skipper which consumes superfluous spaces before the actual parser is called. Figure 7.9 illustrates the syntax definition of a *while* loop using a skipper.

```
while_stmnt = str_p("while") >> ch_p('(') >> expression >> ch_p(')') >> statement;
```

Figure 7.9: Parser rule definition in Spirit using a skipper [170]

Beyond primitives and overloaded operators, Spirit provides the usage of so called directives, which can be used to control the behavior of the parser. An important directive is *lexeme_d* which makes it possible to disable the skipper for a defined part of the tree. Another useful directive is *leaf_node_d* which is used to restrain the number of generated leaf notes inside the parse tree. By default, each primitive (even a digit or a character) creates a leaf node. However, the POL compiler does not need such a fine granularity. Therefore, *leaf_node_d* is used to achieve a coarser granularity. Table 7.4 lists all available directives. Figure 7.10 shows an example parse rule using *lexeme_d* and *leaf_node_d* to declare a POL identifier as white-space free leaf node.

Directive	Description
leaf_node_d	Creates a single leaf node for all enclosed rules
discard_node_d	Discards the node and its children
lexeme_d	Disables the skip parser
as_lower_d	Transforms the input to lower case before parsing

Table 7.4: Spirit's parser directives [170]

```
identifier = lexeme_d[leaf_node_d[alpha_p >> *(alpha_p | digit_p | ch_p('_')]];
```

Figure 7.10: Example parser rule using parser directives [170]

The whole POL syntax is described in one single *struct* containing all Spirit parser rules as illustrated previously. This *struct*, the skipper, and an iterator referencing the POL source code are handed over to the Spirit function *pt_parse* which uses the given rules to parse the source code and return the parse tree. The parse tree is used by the Precompiler to generate Java code for the Emulator as well as by the POL compiler to generate VHDL for synthesis.

Error Reporting

Errors must be reported in a way that allows the user of the Framework to find the error source and to correct it. The parser uses assertions to ensure the correctness of the given code. If an assertion fails, the parser throws an exception that is caught and the corresponding error is reported to the user. Figure 7.11 illustrates the assertion used to report a missing bracket. Figure 7.12 shows the respective compiler output.

```
assertion<const char*> exp_bracket_open("'(' expected");
```

Figure 7.11: Assertion used to report a missing bracket [170]

```
# pol example.pol
example.pol:12:11: error: '(' expected
example.pol:1:1: error: parsing failed
```

Figure 7.12: Error output caused by a missing bracket

7.2.2 The Compiler

Basic Structure of the Generated VHDL Code

In section 6.4.6, the basic structure of the generated VHDL code has been described. It has been shown that each POL class (be it *ParObj*, *DispObj* or *StatObj*) is translated to a separate VHDL entity in a separate VHDL file. The name of the entity corresponds to the name of the POL class and appending "_csym". The file name is the entity name plus ".vhd". Each VHDL entity consists of a Main FSM, a Connection FSM, a Memory Allocator, the Context Memory and optional VHDL subcomponents. The Main FSM is the only element that is generated from the given POL code. All other elements have been implemented in VHDL.

The Main FSM is, as the name implies, a finite state machine which consists of states that represent the functionality of the POL code. Furthermore it contains states that are used for initialization, swap-in and swap-out. In the following, a set of states which belong together and represent a self-contained piece of functionality (e.g. swap-out) is called a State Sequence. Figure 7.13 illustrates the basic structure of the Main FSM.

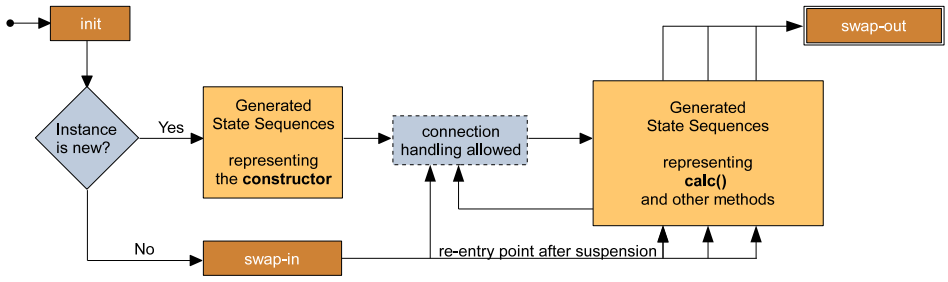


Figure 7.13: Basic structure of the Main FSM

The Context Memory is divided into two halves: the stack and the heap. The heap begins at address *0x00* and grows in positive direction. It is used to store general instance data (instance number, re-entry point, memory allocation map, stack pointer) and private variables during suspension as well as arrays. The stack begins at address *0xff* and grows in negative direction. It is used to store local variables, function parameters, return values, return states and stack base pointers.

Basic Expressions

The POL-Compiler creates two data structures: the Main FSM which contains the generated State Sequences, and the Compiler Context which contains the needed VHDL variables, signals and enumerations. Table 7.5 describes the correlation between basic POL expressions and VHDL State Sequences.

POL	Realization in VHDL
Variables	All variables are translated to VHDL registers. Thus, reading the value of a variable does not consume a state. In contrast, writing a value to a variable is realized as a write to the corresponding VHDL register (on the rising edge of the clock) and therefore consumes one state. During swap-in the values of these registers are read from the Context Memory. During swap-out these values are written to the Context Memory. Thus, the size of the State Sequences used for swap-out and for swap-in depends on the number of *private* POL variables. The value of *this* represents the instance number. It is read from the Context Memory (address *0x00*) during *init* (see figure 7.13).
Array access	POL arrays are not represented by VHDL arrays. Instead, their elements are solely stored on the heap. A write to an element of an array requires one state (BRAM address and data value can be send concurrently). A read from an array requires two states: one for sending the BRAM address to the Context Memory and one for processing the read value.
Get	A *get()* call needs special treatment since its blocks until new data arrives on the input it waits for. To be able to suspend an instance, although it is waiting for data, the State Sequence used to realize the blocking *get()* also has to check for a swap-out request. This requires 5 states. The non-blocking variant *get(default)* returns always immediately and is therefore represented by one single state.
New instance	A DCB request containing the class ID of the new instance is send to the Scheduler. The Main FSM waits until the Scheduler responds the new instance ID which is stored in a VHDL signal representing the object handler. This requires one state.
Finish instance	A special kind of Connection Message is send to the instance that shall be destroyed. This requires one state.
New array	Memory is requested from the Memory Allocator. The expression's value is a reference (BRAM address) to the array. This requires 2 states.
Finish array	The corresponding memory is deallocated by the Memory Allocator. This requires 1 state.
Method invocations	A State Sequence handles the stack management. It's size depends on the size of the context that has to be stored on the stack.

Table 7.5: POL vs. VHDL

Operators

Table 7.6 lists all available POL operators and how they are translated to VHDL. The translation of some operators is very simple, since they have a one-to-one counterpart in VHDL. In contrast, other operators require a complete State Sequence. In order to keep the generated VHDL code understandable and easy to read, some operators were implemented via VHDL functions.

Description	POL Symbol	Implementation in VHDL
Simple assignment	=	State Sequence
Compound assignment	*=, /=, %=, +=, -=	disassembled to arithmetical operators and simple assignments
Ternary condition	? :	VHDL function and State Sequence
Conditional logic	||, &&	State Sequence
Binary bitwise logic	|, &, ^	`or`, `and`, `xor`
Equality operators	==, !=	VHDL functions based on =, /=
Relational operators	<, <=, >, >=	VHDL functions based on <, <=, >, >=
Shift operators	«, >>	VHDL functions based on `sll` and `srl`
Binary arithmetic	*, /, +, -	*, /, +, -
Unary arithmetic	+, -	VHDL functions based on binary minus
Prefix operators	++a, --a	disassembled to arithmetical operators and simple assignment
Postfix operators	a++, a--	disassembled to arithmetical operators and simple assignment
Bitwise complement	~	`not`
Logical complement	!	`not`
Brackets	( )	( )

Table 7.6: Translation of POL operators to VHDL [170]

In VHDL the equality operators as well as the relational operators return a Boolean. However, in order to store the returned values in VHDL registers, std_logics are needed. To solve this problem, VHDL functions are instantiated that compare the two values and return a std_logic. Additionally a VHDL function is instantiated that allows the usage of an unary minus operator.

The ternary condition as well as the conditional logic require a State Sequence, since the execution of the second operand depends on the result of the first operand. This behavior has not been parallelized due to possible side effects of the second operand.

Data Types

POL supports unsigned integers, Booleans and object handlers. Furthermore it supports one-dimensional arrays of these three data types. Table 7.7 illustrates the actual encoding of these data types. The encoding of *port* is only used for *Signals*, *Slots* and the corresponding methods. In the current version, the Communication Matrix makes use of messages with a 32 bit size. Of those, 16 bits are used for addressing and 16 bits are used as payload. The class ID and the slot ID utilize 4 bits. The instance ID utilizes 6 bits. These sizes are parameters of the Framework and can be modified. Please note that such a modification influences all parts of the Framework: the Emulator, the Compiler, the Communication Matrix and the Scheduler.

POL type	VHDL type	Encoding
integer	std_logic_vector(15 downto 0)	big endian
Boolean	std_logic	'0': false, '1': true
array reference	std_logic_vector(15 downto 0)	bit 15-08: heap address bit 07-00: array size
object reference	std_logic_vector(15 downto 0)	bit 15-12: class ID bit 11-06: instance ID bit 05-00: unused
port	std_logic_vector(15 downto 0)	bit 15-12: class ID bit 11-06: instance ID bit 05-00: input ID

Table 7.7: Translation of POL data types to VHDL

Statements

Figure 7.14 illustrates the State Sequences that are used to realize if-branches and loops. In contrast to Java, unreachable code (caused by unsatisfiable conditions) only leads to a warning, not to an error.

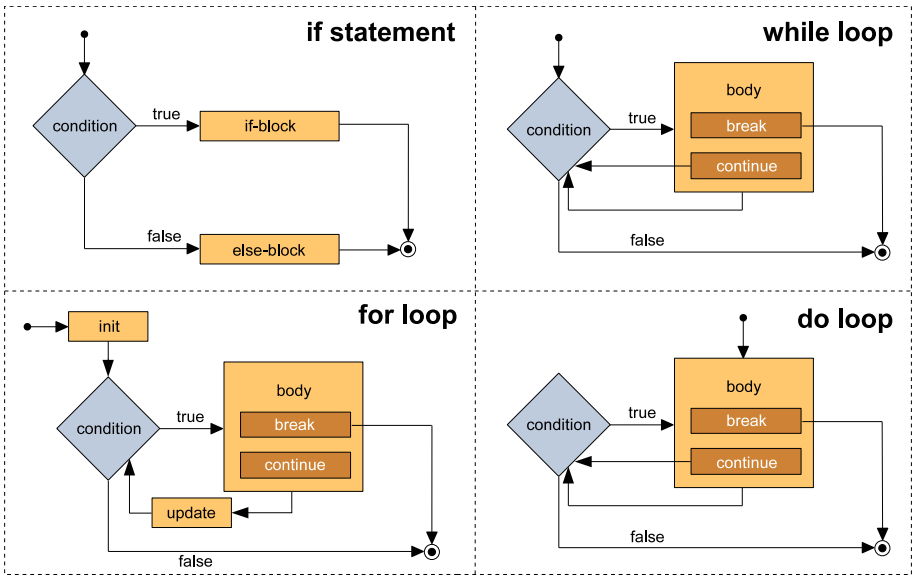


Figure 7.14: State Sequences for if-branches and loops

7.2.3 Connection Management

The communication between several instances is realized via *Signals* and *Slots*. From the compiler's point of view the *Slots* are passive registers, which provide a validity flag and can be read via a blocking or a non-blocking *get* (which resets the validity flag). The message delivery from the Task Area FIFO to the correct *Slot* is performed by the Communication Matrix and does not affect the compiler at all. Please note, that in the current implementation of the Communication Matrix the Instance Buffer has not been implemented, yet. In many cases, this makes it necessary to avoid using the blocking *get()* method.

In contrast to *Slots*, *Signals* are much more complex elements. This is mainly due to the fact that one *Signal* can be connected to multiple *Slots*. Thus, a *Signal* is represented by a State Sequence, which looks for established connections and generates a message for each *Slot* connected to the *Signal*. For this, each *Signal* is assigned to 16 BRAM addresses which are part of the heap and are used to store the addresses of the connected *Slots*.

Connection Messages

The methods *connect* and *disconnect* create Connection Messages. These messages are always addressed to port 15 and consist of two data words. Table 7.8 illustrates the structure of the Connection Messages. In contrast to normal *Slots* the Connection Slot

Word nr.	Encoding of the payload	
1.	bit 15-05: bit 04: bit 03-00:	zeroed disconnect flag the ID of the affected Signal
2.	bit 15-12: bit 11-06: bit 05-02:	the class ID of the Slot the instance ID of the Slot the input ID of the Slot

Table 7.8: Encoding of the Connection Messages

(Slot ID 15) requires special treatment. In fact, the handling of Connection Messages is realized by an additional state machine (see figure 6.18). This state machine is synchronized to the Main FSM. It is only allowed to handle Connection Messages, after the constructor or *calc()* have finished and before the first *get* or *emit* have been executed. As long as the Connection FSM is processing a request, *emit* and *get* calls must wait until it has finished.

If the *disconnect flag* is set, the Connection FSM tries to terminate a connection. Otherwise it establishes a connection. The ID of the affected *Signal* informs the Connection FSM, to/from which *Signal* a receiver shall be added/removed. The class ID, instance ID and Slot ID transmitted in the second word inform the Connection FSM which *Slot* shall be added/removed to/from the list of receivers. If a *Slot* shall be removed from the list of receivers but is not stored in this list, the disconnection fails without reporting an error.

Serialization

Before data elements can be transmitted via *emit*, they have to be transformed to (a sequence of) 16-bit words. This is trivial for integers and object references, since they are already implemented as 16-bit vectors. Booleans are converted to a 16-bit vector consisting of 15 leading zeros and one bit containing the actual value.

Arrays are more complex, since their size exceeds the size of a single data message. Thus, they must be serialized. The first data packet contains the size of the array. Subsequently, every element of the array is transmitted in the order they are stored on the heap. The receiver of the array executes an implicit *new*, which allocates the necessary space on the heap. Finally, the array is filled with the received values.

In a way, the Connection Messages can be seen as a special kind of serialized messages. All serialized messages make use of the *aligned bit* which is part of the message's address and tells the Communication Matrix to handle this series of messages as an uninterrupt-

ible block. This prevents the interruption of a message series by other messages which are heading for the same class.

7.2.4 The Memory Allocator

The memory allocator is a handwritten VHDL component that is used to manage the allocation of memory in the heap. For this, the heap is separated into 128 sub units which are represented by a 128-bit vector (called the allocation map). A '1' inside this vector represents a used sub unit. A '0' represents a free sub unit. Figure 7.15 shows the interface of the memory allocator.

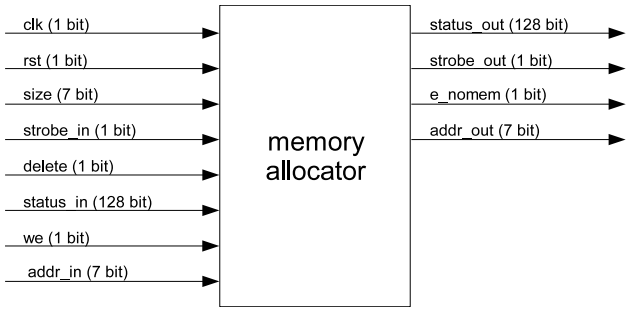


Figure 7.15: Interface of the memory allocator

To **allocate** a block of memory, *size* is set to the size of the block and *delete* is set low. *strobe_in* must be set high for one clock cycle. As a result, the allocator searches for a pattern of *size* zeros in its allocation map. If it was successful, it returns the corresponding BRAM address (using *addr_out*), sets *e_nomem* low and *strobe_out* high for one clock cycle. Moreover, it sets the corresponding bits in the allocation map to '1'. If it was not successful, it sets *e_nomem* high and *strobe_out* high for one clock cycle.

To **deallocate** a block of memory, *addr_in* is set to the address of the block, *size* is set to the size of the block and *delete* is set high. *strobe_in* must be set high for one clock cycle. As a result, the allocator sets the corresponding bits of the allocation map to '0'. Finally, it sets *strobe_out* high for one clock cycle.

If the instance is **swapped in** or started the first time, the allocation map has to be initialized with the correct values. For this, *status_in* is set to the desired value of the allocation map and *we* is set high.

If the instance is **swapped out**, the allocation map has to be stored in the Context Memory. For this, the value of *status_out* can be used. It represents the current allocation map.

7.2.5 Dynamic Hardware

As shown in section 6.3, the actual configuration of an instance corresponds to a swapping process which includes the interaction between the Scheduler and the instance. Furthermore, the instance needs to know if it has been started the first time, in which case it has to execute the constructor. In order to be able to determine in which state the instance has been interrupted, so called interruption points are used. These interruption points are defined as enumerated interruptible states. Each interruption number corresponds to one single interruptible state of the Main FSM. The interruption number is stored in the Context Memory at address 0x1.

Creation

If the interruption number is zero, the instance is executed the first time. Thus, it initializes the stack pointers and the memory allocator. Next, it executes the State Sequences representing the member initializers and the constructor. After this, it executes the State Sequences representing the *calc* method.

Swap-Out

If an instance shall be swapped out, it is informed by the Scheduler via the DCB. As soon as the Main FSM reaches an interruptible state, the instance stores the interruption number and calls the swap-out handler (which is a special State Sequence of the Main FSM). The swap-out handler saves all VHDL registers that represent a POL variable in the Context Memory. Furthermore, it stores the heap allocation map and the interruption number. Next, it informs the Scheduler via the DCB that it is ready to be deactivated. Finally, the Main FSM transitions to its final state. This state can only be left via a reset.

Swap-In

If the interruption number is not zero, the instance continues to execute after a swap-out. Thus, the swap-in handler (which is a special State Sequence of the Main FSM) is called. It reads back the heap allocation map as well as the values of the VHDL registers that represent a POL variable. Subsequently, it executes the State Sequences representing the *calc* method.

Termination

An instance can terminate itself or other instances. In any case, the call of *finish* at first leads to the generation of a special Connection Message, called Destruction Message. This Destruction Message only consists of one single data word: *0x7FFF*. The usage of Destruction Messages has been introduced to synchronize the destruction of an instance with data that is heading for this instance and is still stored in the buffers (see chapter 6.3.2).

The actual termination of an instance is realized by its Connection FSM which receives the Destruction Message. It forces the Main FSM to transition to its final state as soon as it reaches an interruptible state. No data is stored (the swap-out handler is skipped) and the instance informs the Scheduler via the DCB that it can be deleted.

7.2.6 VHDL Subcomponents

VHDL subcomponents implemented in POL are directly used as VHDL components. Syntax and behavior of the subcomponents are not touched by the compiler at all. Only the available input signals and output signals are analyzed. Table 7.9 shows the input and output signals that must be provided by each VHDL subcomponent. Additionally, every user-defined output and input signal leads to the need for an corresponding VHDL output or input signal with a size of 16 bit. The signal *strobe_in* is used by the Main FSM to call the subcomponent. The signal *strobe_out* indicates that the subcomponent has completed its calculation and that the user defined output signals are valid.

Direction	Port name	Port width	Description
in	`clk`	1	the global clock signal
in	`rst`	1	the task reset signal
in	`strobe_in`	1	starts the calculation of the VHDL entity
out	`strobe_out`	1	indicates the end of the calculation

Table 7.9: Minimal interface of a VHDL subcomponent [170]

Optionally, the VHDL subcomponent can implement a BRAM interface. In this case, the POL complier establishes a connection between the VHDL subcomponent and the heap. Required space can be allocated as integer array in POL. Table 7.10 shows the BRAM interface of the VHDL subcomponents.

Direction	Port name	Port width	Description
in	`bram_dout`	16	data out
out	`bram_addr`	8	address
out	`bram_din`	16	data in
out	`bram_we`	1	write enable

Table 7.10: Additional interface ports for BRAM access [170]

7.2.7 Code Optimization

To decrease execution time and to save FPGA resources, the compiler performs some optimizations based on the merging of states. Two succeeding states can be merged to one state, if the first state is the only direct predecessor of the second state and the second state does not read/write from/to resources the first state writes to. Furthermore, *get()* calls on different input queues are bundled if they are part of the same expression. The optimizations take place, after the main state machine has been created. Please note that the consecutive access to elements of an array cannot be optimized, since arrays are solely stored in the Context Memory.

7.2.8 Generated VHDL

After a POL class has been parsed and the corresponding Compiler Context has been created, the VHDL file is generated. Registers that represent VHDL variables are named as the POL identifier plus "_sym". If a variable name is used twice (in different scopes), the name of the VHDL signal also contains the line and the column number of the variable's definition. The state names in the Main FSM consist of two parts. The first part is an identifier representing the basic functionality of the state. The second part is the line number of the POL expression and, if needed, the column number. As a result, a comparison between the generated VHDL code and the POL source code is very easy. This was important for development and debugging of the POL compiler. Figure 7.16 illustrates the layout of the generated VHDL file. Some code examples are shown to demonstrate how POL expressions are translated to VHDL. Beyond the VHDL files, the compiler creates the configuration file *task.cfg*, which lists the classes and the corresponding class types and class IDs (see chapter 6.5).

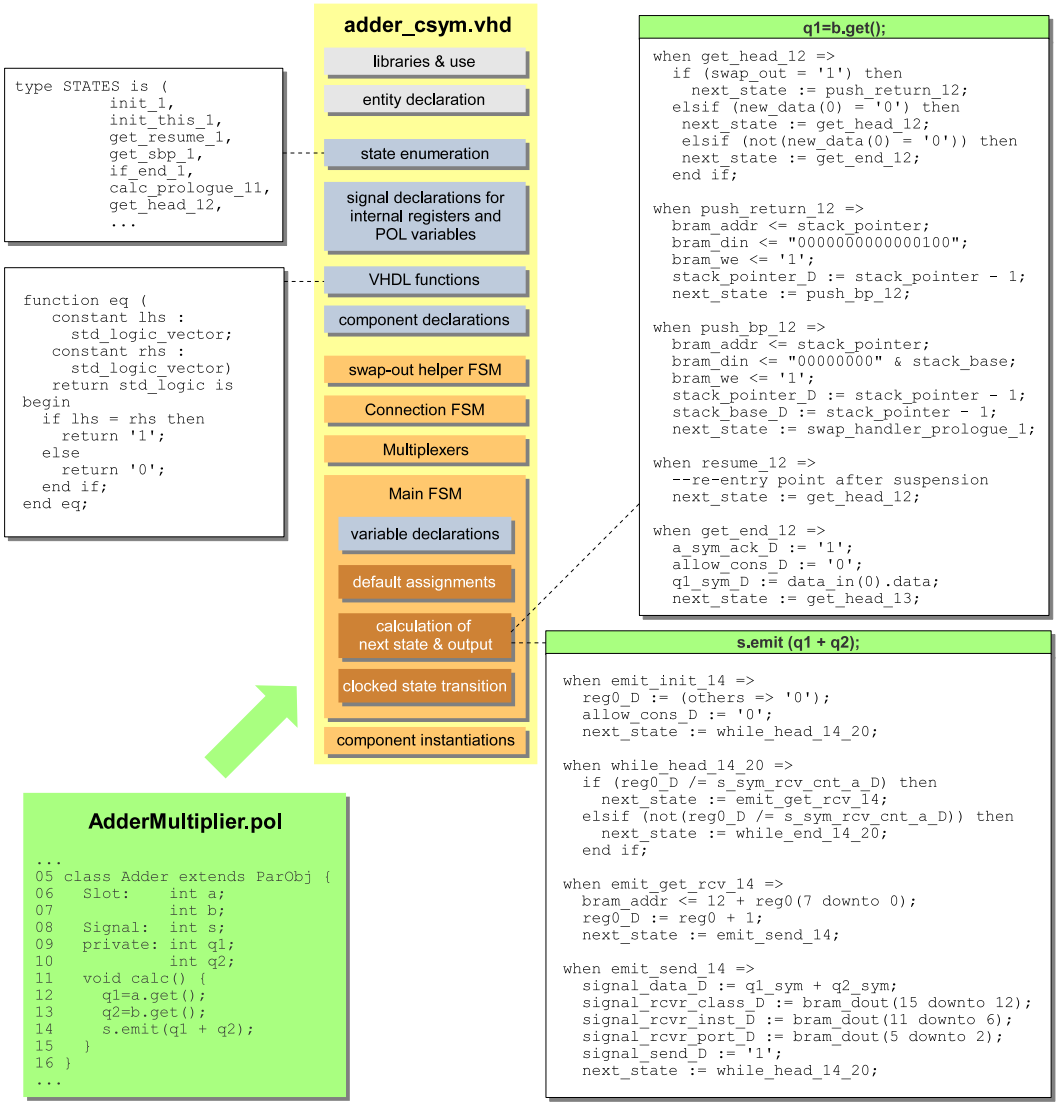


Figure 7.16: Layout of the generated VHDL files

7.3 Communication Matrix

The Communication Matrix interconnects the dynamic parts and the static parts of the design in a very flexible and highly concurrent way. It serves as an additional layer between the generated modules and the actual FPGA hardware. In the following section, the implementation of the Communication Matrix is described.

7.3.1 Data Format

In order to keep the VHDL code clear and easy to read, the Communication Matrix makes use of nested VHDL arrays and records to define the interfaces of Class Buffers, Task Areas and other elements. These records are declared in *matrix_components.vhd*. It determines the structure and the size of the connecting signals.

The most important record is *object_word*, since it defines the structure of the messages sent by the *Signals*, stored in the buffers and received by the *Slots*. The structure of the DCB interface is defined by the record *DCB_word*. Both records are illustrated in figure 7.17.

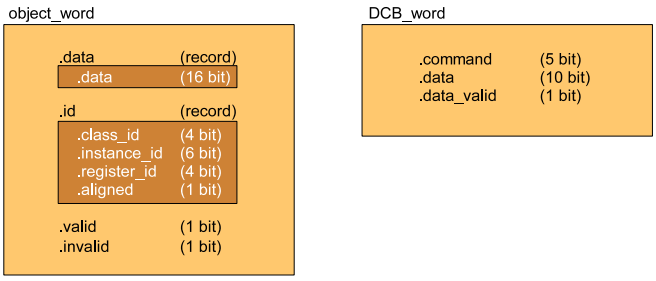


Figure 7.17: Structure of the records object_word and DCB_word

The record *object_word* consists of four parts: the subrecord *.data* which contains the message's payload, the subrecord *.id* which contains the receiver address as well as the *aligned*-bit (used for serialized messages — see section 7.2.3) and the control bits *valid* and *invalid*.

7.3.2 System Templates

In order to represent the static part of the design (and thus all pre-defined parameters like the number of provided Task Areas or the size of a message), System Templates are used. Figure 7.18 illustrates the directory structure of a template and explains which data is stored at which position.

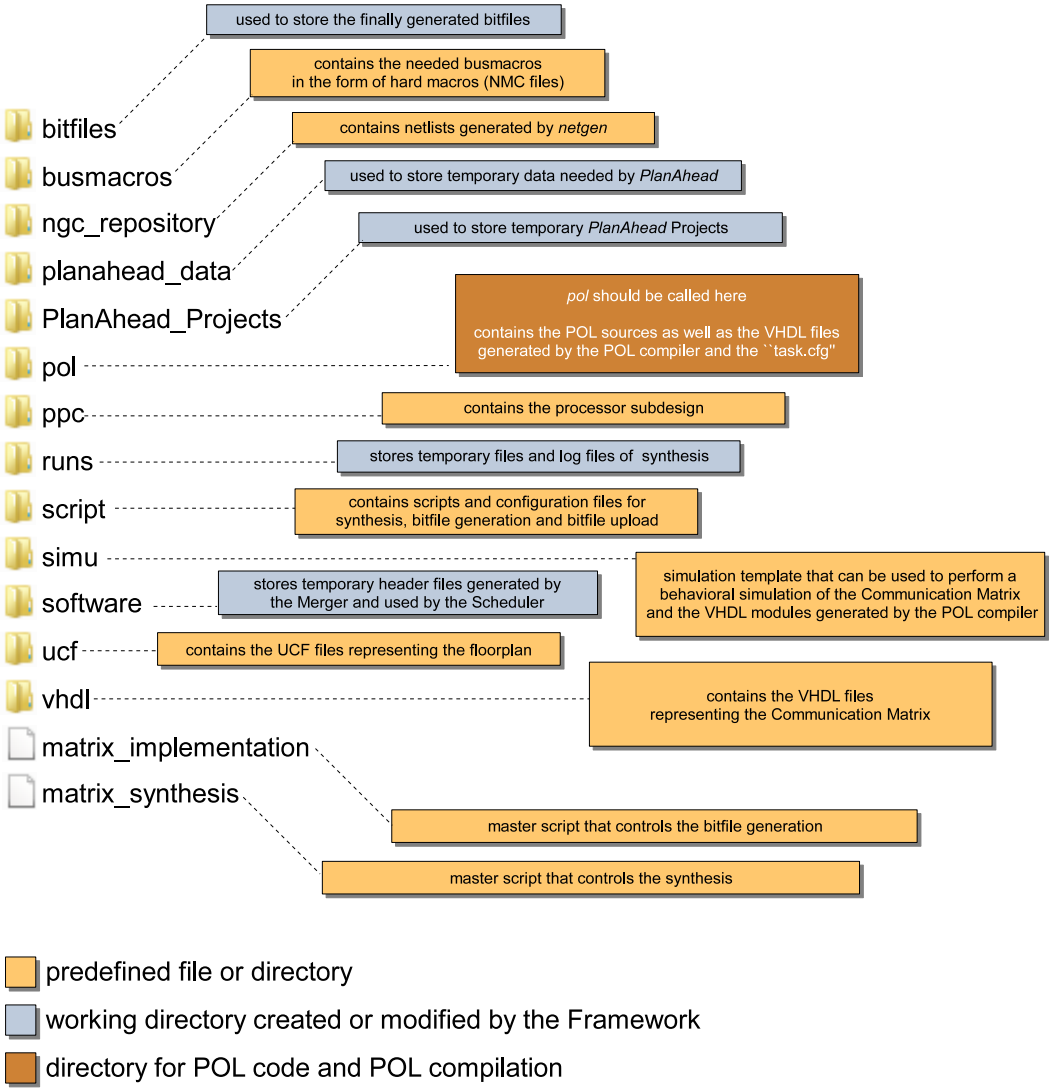


Figure 7.18: Directory structure of the System Templates

7.3.3 Structure of the VHDL code

Usually, the structure of the VHDL code highly corresponds to the logical structure of the design as presented in chapter 6.2.4 in figure 6.6. Unfortunately, this haptic approach cannot be used for the implementation of the Framework since *PlanAhead* is used as the orchestrator for the creation of the (partial) bitfiles. *PlanAhead* dictates a structure of the VHDL code that is geared to the partitioning into static and dynamic parts. The top VHDL component must contain the static part as subcomponent, the dynamic areas as subcomponents and the busmacros (as black box subcomponents[8]). Optionally, it can contain I/O-buffer instantiations. All other logic must be placed in the subcomponents. Busmacros are not allowed to be placed in the subcomponents at all.

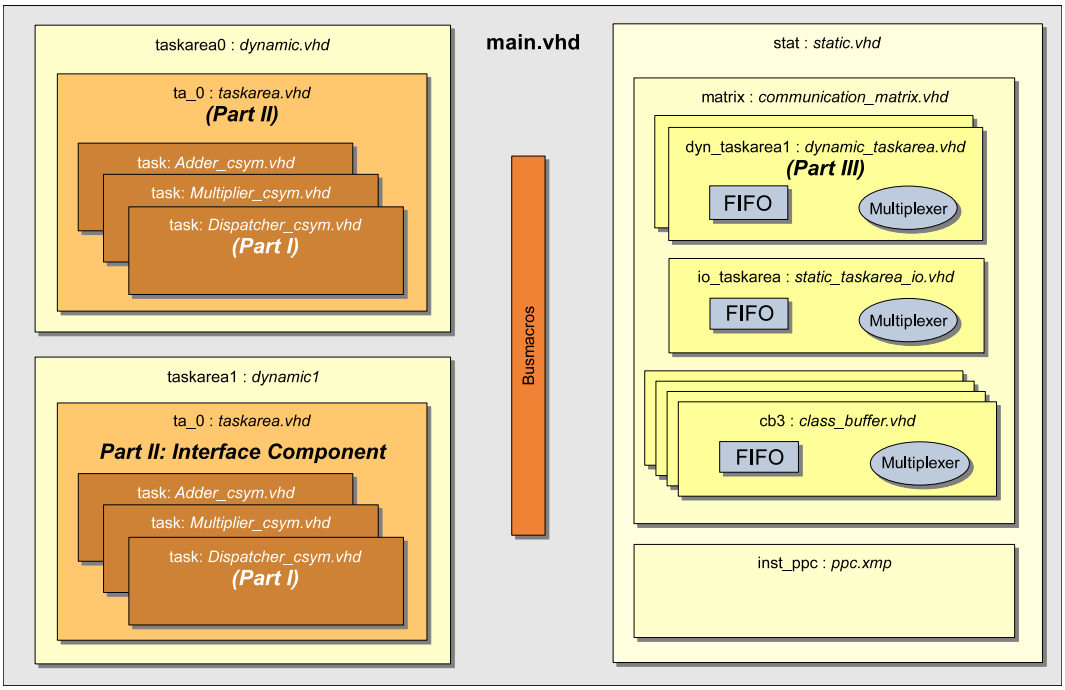


Figure 7.19: Structure of the VHDL code

As a consequence, although a dynamic module is a logical subcomponent of a static component (as it is the case in the Framework — see figure 6.6) it cannot be implemented as a VHDL subcomponent of this static component. Rather, it has to be implemented as a subcomponent of the dynamic part. The corresponding signals between the static component and the dynamic subcomponent have to be passed through enclosing VHDL components and must be routed through the busmacros in the top VHDL component. Figure 7.19 illustrates the resulting structure of the VHDL code.

[8]The busmacros are added to the design as pre-routed and pre-placed hardmacros (NMC files) during the place and route process

The Matrix

The Communication Matrix has to connect each Task Area FIFO with each Class Buffer multiplexer and each Class Buffer FIFO with each Task Area multiplexer. For this, nested VHDL *generates* are used. Figure 7.20 shows the implementation of the interconnection between Class Buffers and Task Areas. Both, *data_in* and *data_out* are of type *object_word*. If less than 16 POL classes are used, the unneeded signals are not connected to a component and thus removed by the synthesis tool.

```
gen_taskareas : for area in 0 to 15 generate begin
  gen_taskarea : for buff in 0 to 15 generate begin
    process (reset, ...)
    begin
      if (reset='1') then
        taskarea(area).data_in(buff) <= no_data;
        class_buffers(buff).data_out_read_enable(area) <= no_read;
        class_buffers(buff).data_in(area) <= no_data;
        taskarea(area).data_out_read_enable(buff) <= no_read;
      else
        taskarea(area).data_in(buff) <= class_buffers(buff).data_out;
        class_buffers(buff).data_out_read_enable(area)
              <= taskarea(area).data_in_read_enable(buff);
        class_buffers(buff).data_in(area) <= taskarea(area).data_out;
        taskarea(area).data_out_read_enable(buff)
              <= class_buffers(buff).data_in_read_enable(area);
      end if;
    end process;
  end generate;
end generate;
```

Figure 7.20: Nested *generates* used to interconnect the Class Buffers with the Task Areas

The Active Multiplexers

In order to support the usage of *active* multiplexers which decide, whether a message is to be accepted or not, all FIFOs in the system are of type FWFT (First Word Fall Through). This means that the multiplexers see the topmost message stored in the FIFO as well as a control bit (*valid*) indicating whether the FIFO output is valid or not. If it is valid, the corresponding multiplexer processes the output and sends an acknowledge signal to the FIFO (in figure 7.20 this acknowledge signal is called *data_in_read_enable*). In addition to the *valid* bit, the messages contain an *invalid* bit. This bit is used by the FIFOs to inform the multiplexers that another multiplexer has read the data and that it is therefore no longer valid. This is necessary, since the actual *valid* bit is a single pulse and the information regarding validity is cached by all multiplexers. Setting the *invalid* bit to '1' clears this cached *valid* bit.

The third control bit is the *aligned* bit. It is used by the POL compiler to tag serialized messages. In normal operating mode, a multiplexer checks all connected FIFOs for new data. If multiple FIFOs send concurrent messages a multiplexer, the data from the FIFO

with the lowest number is taken. So, it could happen that a message stream coming from Task Area 2 is interrupted by a message coming from Task Area 1. For serialized messages this behavior is undesirable. Therefore, if the *aligned* bit of a message is set to '1', the multiplexer that processed this message only checks for messages coming from the same FIFO. The POL compiler sets all *aligned* bits except that of the last word of a serialized message to '1'.

The FIFOs

As previously explained, all FIFOs inside the Communication Matrix are of type FWFT. Since the interface of a FIFO supports bit vectors, but no records, the messages need to be converted to normal bit vectors. All bits of a message with the exception of the *valid* and the *invalid* bits are stored in the FIFOs. The width of the FIFOs is 32 bit. The depth of each FIFO is 512 words. Thus, each FIFO matches exactly one single BRAM.

Dynamic Elements

The number of needed Class Buffers depends on the POL source code (namely on the number of instantiated POL classes). Thus, the Communication Matrix makes use of VHDL templates which provide keywords that are replaced by the Merger with the correct number of Class Buffer instantiations before synthesis. Furthermore, multiple synthesis runs are needed to create all netlists needed by *PlanAhead*. The runs for the dynamic modules depend on the instantiation of the correct VHDL component representing the correct POL class. This is realized by the script *matrix_synthesis* which is described in detail in section 7.5.

The number of Task Areas is an attribute of the used System Template. A change of the number of available Task Areas requires a change of the template. For development and testing of the Framework, two System Templates have been implemented: a template providing one Task Area and a template providing two Task Areas. The following sections focus on the latter.

7.4 Behavioral Simulation

For the development of the Communication Matrix and of the POL-Compiler an additional testing layer has been introduced: behavioral simulation. Before the VHDL code is used for bitfile generation, it can be simulated using Modelsim. This makes it possible to detect and remove errors at an early stage. Please note that this additional step was only necessary for the development of the Framework. A user of the Framework does not have to care about behavioral simulations at all. The correctness of the generated code and of the Communication Matrix is assured by the Framework itself.

The biggest problem regarding behavioral simulation comes with the usage of DPR. Today, there is no simulation tool that directly supports runtime reconfiguration. Thus, the reconfiguration has to be substituted by a comparable but simulatable process. For the behavioral simulation of the Framework the file *taskarea.vhd*[9] has been adopted. Normally, it instantiates the classes alternately. The correct class ID is set by the Merger — n classes require n synthesis runs. At each run, only one single class is instantiated[10]. This behavior has been changed for simulation. Here, all classes are instantiated simultaneously. Which class is virtually loaded to the FPGA is decided by a multiplexer which emulates the runtime reconfiguration.

The processor subsystem is *not* part of the simulation. Only the matrix control interface (see section 7.6.2) is simulated. The behavior of the Scheduler is emulated by do-files (which actually are tcl[11]-scripts) that assign the correct signal sequences to the matrix control interface:

peek	Reads one data item from the Communication Matrix.
poke	Writes one data item to the Communication Matrix.
reconf.do	Emulates a reconfiguration.
swapin.do	Activates an instance.
swapout.do	Deactivates an instance.
schedule.do	Activates an instance for a certain time interval.
scenario1.do	Used to simulate a complete Scheduler cycle. It first initializes the Communication Matrix. Then it answers to new-instance requests (caused by the constructor of the initial class). Next, it sends some data messages to the Communication Matrix and alternately activates the instances (using *schedule.do*). Finally it uses *peek* to receive produced output messages.

[9] see figure 7.19
[10] More details about bitfile generation will be shown in the following section 7.5
[11] Tool Command Language

Figure 7.21 illustrates the content of *reconf.do*. Figure 7.22 shows a screenshot of the simulation of the adder-multiplier example realized via *scenario1.do*.

```
force -freeze /tb_vhd/uut/stat/busmacro_enable_i(0) '0'
force -freeze /tb_vhd/uut/stat/taskarea_reset_i(0) '1'
run 100 ns

force -freeze /tb_vhd/uut/taskarea0/ta_0/select_task $1
force -freeze /tb_vhd/uut/taskarea0/ta_0/task1/this $2
force -freeze /tb_vhd/uut/taskarea0/ta_0/task1/select_instance $2
force -freeze /tb_vhd/uut/stat/active_class_0_i(7 downto 6) $1
force -freeze /tb_vhd/uut/stat/active_class_0_i(5 downto 0) $2
run 100 ns

force -freeze /tb_vhd/uut/stat/busmacro_enable_i(0) '1'
force -freeze /tb_vhd/uut/stat/taskarea_reset_i(0) '0'
run 100 ns
```

Figure 7.21: *reconf.do*

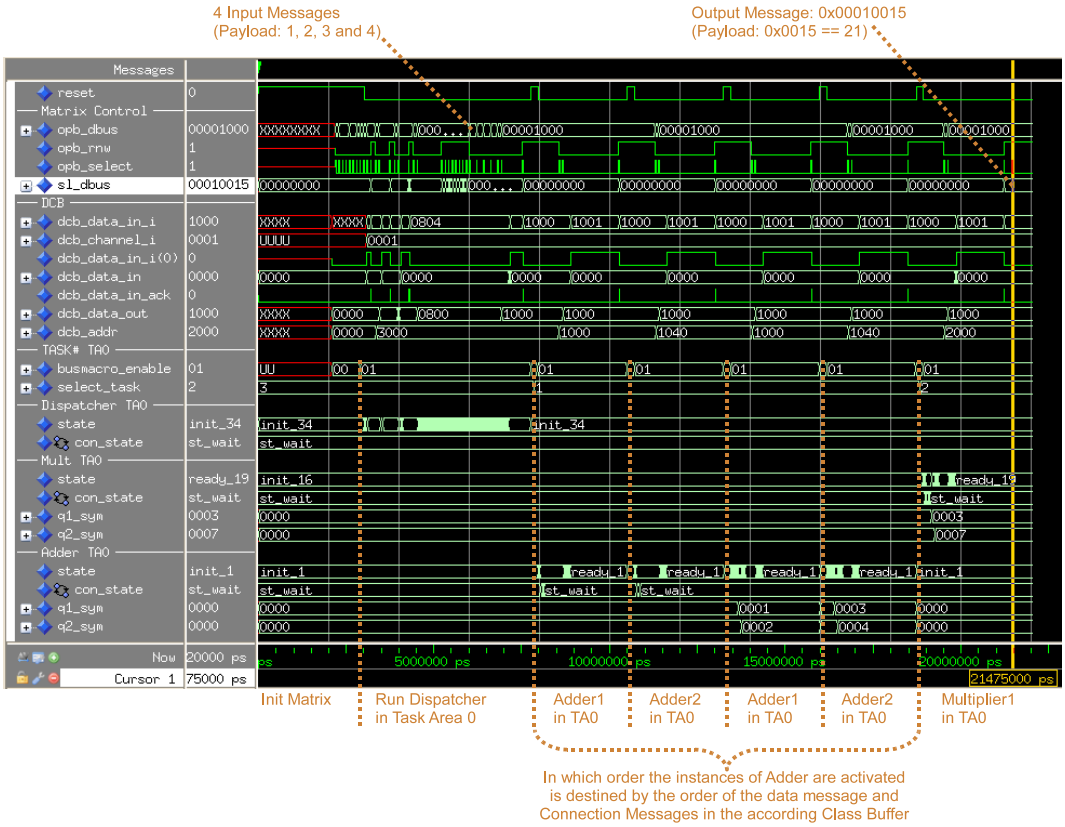


Figure 7.22: Simulation screenshot of the adder-multiplier example
Performed calculation: $(1+2)\cdot(3+4)=21$

7.5 Bitfile Generation

In the following section, the creation of the full and the partial bitfiles is described. The sources of the bitfile generation are the VHDL files generated by the POL compiler, the netlists representing the processor subdesign, and the VHDL files representing the Communication Matrix.

7.5.1 Synthesis

The usage of dynamic partial reconfiguration and the utilization of *PlanAhead* as orchestrator of the bitfile generation have a strong influence on the structure of the VHDL code (see section 7.3.3) as well as the way the synthesis is performed. If n is the number of used POL classes and a is the number of Task Areas, *PlanAhead* expects $a \cdot n + 2$ NGC files (see section 2.6.1): $n \cdot a$ NGC files representing the dynamic modules, one NGC file representing the static part and one NGC file representing the top module. The generation of these NGC files has to be done in the following way:

- For the synthesis of the top module, the busmacros, the dynamic components (representing the dynamic areas) and the static component have to be included as black boxes.
- For the synthesis of the static part, the static module has to be declared as the top module. The actual top module as well as the dynamic modules are left out.
- For the synthesis of n dynamic modules, $n \cdot a$ synthesis runs are needed. The corresponding algorithm can be explained best in pseudo code:

```
for (i=1; i<=a; i++) {
  USE DYNAMIC AREA i AS TOP MODULE
  for (j=1; j<=n; j++) {
      INSTANTIATE DYNAMIC MODULE j AS "PART I"
      DO SYNTHESIS
  }
}
```

For the alternate instantiation of the dynamic modules and for the instantiation of the correct number of Class Buffers, VHDL templates are used. The templates contain the actual VHDL code and so called magic comments. These magic comments are replaced by the Merger with the needed VHDL component instantiations. The syntax of these comments is:

--POL: *command*

The available magic comment commands are shown in table 7.11. The Merger uses the file *task.cfg* (which has been created by the POL compiler) as an input file to assign the correct class name and class type to a class ID.

Command	VHDL inserted by the Merger
static_taskarea_number	an integer constant counting the static taskareas
dynamic_taskarea_number	an integer constant counting the dynamic taskareas
dynamic_task_instantiation	instantiation of a particular dynamic module (the module number is an argument of the Merger)
task_declarations	declaration of all dynamic modules
task_instantiations	instantiation of all dynamic modules
static_taskarea_instantiations	instantiation of all static Task Areas

Table 7.11: Template commands the merger substitutes

The Merger is a Linux binary and is called via the command *merger*. It provides several options that control the behavior of the Merger and determine which magic comments shall be replaced. The Merger uses the current directory as input directory, parses each VHDL template file (ending with ".vhdt") and translates it to a usual VHDL file. Please note that this means that the ".vhdt" files are the actual source files, not the corresponding ".vhd" files. Table 7.12 lists the available Merger options.

Merger option	Description
--static_tasks	return a list of the names of all static classes
--dynamic_tasks	return a list of the names of all dynamic classes
--expand_static	expand all static task templates
--expand_dynamic *class*	expand all dynamic module templates with class *class*
--create_pol_h	create header file 'pol.h' for the Scheduler

Table 7.12: Command line options for the merger.

The whole synthesis process is controlled by the bash script *matrix_synthesis*. In the following part, the synthesis will be illustrated step by step. Please note that a user of the Framework does not have to deal with these implementation details at all. He or she just calls the synthesis script.

Preparation

Firstly, the synthesis script clears all temporary directories used for synthesis. These are *runs* and *planahead_data*. Next, it builds up the necessary temporary directory structure. The directory *runs* is used to store all temporary synthesis files. The directory *planahead_data* is used to store the created NGC files. Next, the Merger is used to create a list of all dynamic modules. The corresponding command is:

```
dynamic_tasks=‘merger --dynamic_tasks ../pol/‘
```

This list is used to copy all VHDL files representing a POL class from the directory *pol* to the directory *vhdl*. Finally, the file *full_files.txt* is created. It contains a list of all VHDL files which have to be synthesized.

The Static Part

For the generation of the static part, the Merger is used to expand all static task templates. The corresponding command is:

```
merger --expand_static
```

This option causes the Merger to replace all magic comments that contain one of the following commands:
static_taskarea_number, *dynamic_taskarea_number* or *static_taskarea_instantiations*
The affected files are *communication_matrix.vhdt* and *matrix_components.vhdt*. Finally, the synthesis of the static part is started. The file *create_static.txt* determines the synthesis options. It inter alia defines the static component as top module. The corresponding command is:

```
xst -intstyle xflow -ifn ../script/create_static.txt \
    -ofn ../runs/report/static.txt
```

The Dynamic Parts

As previously mentioned, the synthesis of n dynamic modules for a dynamic areas requires $a \cdot n$ synthesis runs. For the instantiation of the correct dynamic module, the Merger is used. Since the number of available dynamic Task Areas is an attribute of the System Template, the runs needed for each Task Area are denoted explicitly. The corresponding commands are:

```
for dynamic_task in $dynamic_tasks; do
    merger --expand_dynamic $dynamic_task --expand_static
    xst -intstyle xflow -ifn ../script/create_dynamic.txt \
        -ofn "../runs/report/$dynamic_task.0.txt"
    xst -intstyle xflow -ifn ../script/create_dynamic1.txt \
        -ofn "../runs/report/$dynamic_task.1.txt"
done
```

The option *expand_dynamic* causes the Merger to replace all magic comments that contain one of the following commands:
dynamic_taskarea_number or *dynamic_task_instantiation*
The affected files are *dynamic.vhdt*, *dynamic1.vhdt* and *taskarea.vhdt*. Figure 7.23 lists the content of *dynamic.vhdt* and the corresponding generated *dynamic.vhd*.

dynamic.vhdt

```
library IEEE;
...

entity dynamic is
...
end dynamic;

architecture Behavioral of dynamic is

component taskarea is
...
end component taskarea;

signal reset : std_logic;

begin

reset <= not reset1;

-------------------------------------
-- POL: dynamic_task_instantiation
-------------------------------------

end Behavioral;
```

dynamic.vhd

```
library IEEE;
...

entity dynamic is
...
end dynamic;

architecture Behavioral of dynamic is

component taskarea is
...
end component taskarea;

signal reset : std_logic;

begin

reset <= not reset1;

-------------------------------------
-- dynamic task instance, generated by merger

-- dynamic task 2instance

ta_0 : taskarea generic map (
          class_id      => 2
)port map (
          clk           => sys_clk,
          reset         => reset,
          data_in       => totaskarea_data,
          data_out      => fromtaskarea_data,
          DCB_data_in   => DCB_data_in,
          DCB_addr_in   => DCB_addr_in,
          DCB_data_out  => DCB_data_out,
          DCB_data_ack  => DCB_data_ack,
          read_enable   => fromtaskarea_read);

-------------------------------------

end Behavioral;
```

Figure 7.23: Comparison of the VHDL template file *dynamic.vhdt* with the generated VHDL file

The generated VHDL line of paramount importance is the generic map determining the value of the generic *class_id*. In the file *taskarea.vhdt* each POL class is instantiated in an if-generate depending on the value of *class_id*. It is the generic *class_id* that determines the POL class that is actually used for synthesis.

The file *dynamic1.vhdt* is a one-to-one copy of *dynamic.vhdt*. Different names have been introduced as a workaround for a *PlanAhead* bug: *PlanAhead* confuses the areas if both are implemented as an instance of the same VHDL component.

The Top Module

In order to generate the top module, the Merger is not needed. The file *create_top.txt* determines the synthesis options. It defines the top module and reduces the set of used VHDL source files to *main.vhd* and *matrix_components.vhd*. Thus, all subcomponents are implemented as black boxes as expected by *PlanAhead*.

```
xst -intstyle xflow -ifn ../script/create_top.txt \
    -ofn ../runs/report/top.txt
```

Completion

After all synthesis runs have been performed, the generated NGC files are copied to the directory *planahead_data*. It contains $4 + n$ subdirectories (this directory structure is also geared to the needs of *PlanAhead*):

top	only contains the top module
static	contains the static part including the processor subdesign
busmacros	contains the hardmacros representing the busmacros
ucf	contains the constraints file representing the floorplan
Adder_csym	contains the files *dynamic.ngc* and *dynamic1.ngc* representing the POL class *Adder* in the first and the second dynamic area
...	(since two dynamic areas are used, each POL class is represented by two files called *dynamic.ngc* and *dynamic1.ngc*)

Next, the Merger creates the file *pol.h* which is used by the Scheduler to determine the number of POL classes and the class ID of the initial class. Finally, a summary of all synthesis reports is shown, giving the developer a chance to check quickly if the synthesis runs have been successful. If an error has occurred, the synthesis reports stored in the directory *runs/report* can be used to determine the error source.

7.5.2 Place, Route and Merging

The creation of the partial bitfiles is realized with Xilinx ISE 9.1i SP2 enhanced by the PREA patch. *PlanAhead* is used as orchestrator. This means that the ISE commands are not called directly, but via scripts generated by *PlanAhead*. This final step can be done in two ways. First, the graphical user interface of *PlanAhead* can be used. This comes with the advantage of a user-friendly interface that makes debugging much easier but depends on user-interaction and demands a lot of time-consuming clicks. Secondly, *PlanAhead* can be started in batch mode, which makes it possible to fully automate the bitfile generation via tcl[12]-scripts.

The Framework makes use of the batch mode, but all steps are implemented in a way that makes it possible to fall back to the graphical version whenever needed (e.g. if an error occurs). The whole process is controlled by the script *matrix_implementation* which has to be called with one argument. This argument determines the name of the *PlanAhead* project that will be created as well as the name of the subdirectory of *PlanAhead_Projects* where the project is stored. In the following part the script is described in detail.

Preparation

Firstly, the Merger is used to generate a list of all dynamic modules. This list is stored in a shell variable. Next, the file *task.cfg* is parsed to determine the name of the class that is configured initially to the dynamic areas (in the following, this class is called the dispatcher). Finally, the project directory is removed and re-created as blank directory.

Floorplaning and the Static Part

After preparation, *PlanAhead* is started in batch mode. The corresponding command is:

```
planAhead -mode batch -source $pwd/script/static.tcl
```

The file *static.tcl* is a tcl-script that controls *PlanAhead* in batch mode. *PlanAhead* uses the namespace *hdi* to provide special functions regarding floorplanning and bitfile generation. In order to be able to parameterize commands as well as arguments without the constraints and restrictions of tcl, templates (named ".tclt") have been used. These templates contain special placeholders that are replaced via the Linux tool sed[13]. They are used to generate the actual tcl-scripts. Table 7.13 lists these placeholders and how they are replaced.

[12] Tool Command Language
[13] Stream EDitor

Placeholder	Replaced by
$pname	the project name
$dir	the project directory
$dispatcher	the entity name of the dispatcher
$list	the list of all dynamic modules

Table 7.13: Placeholders and their replacement

The script *static.tcl* first creates a new project via *hdi::project new*. The *main.ngc* is defined as top module and the subdirectories under *planahead_data* are used as NGC source directories. Next, the script sets the correct target architecture. Then it reads the predefined floorplanning informations from the file *main.ucf*. This file contains the placement of the I/O-pins, the definitions of the clock nets, the placement of the BRAMs and the placement of the dynamic areas. The latter is implemented via so called area constraints which are illustrated in figure 7.24.

```
INST "taskarea0/*" AREA_GROUP=task0;
AREA_GROUP "task0" RANGE = SLICE_X40Y127:SLICE_X71Y64;
AREA_GROUP "task0" RANGE = RAMB16_X3Y15:RAMB16_X3Y15;
AREA_GROUP "task0" RANGE = DSP48_X0Y16:DSP48_X0Y31;

INST "taskarea1/*" AREA_GROUP=task1;
AREA_GROUP "task1" RANGE = SLICE_X40Y63:SLICE_X71Y0;
AREA_GROUP "task1" RANGE = RAMB16_X3Y7:RAMB16_X3Y7;
AREA_GROUP "task1" RANGE = DSP48_X0Y0:DSP48_X0Y15;
```

Figure 7.24: Area constraints placing the dynamic areas

After creating the project, a DRC check is performed. Then, the project is declared as a partial reconfiguration project. From then on, *PlanAhead's* DPR functionality can be used. The corresponding procedure is:

```
hdi::pr setProject -name $pname
```

Before the project has been defined as a DPR project, *taskarea0* and *taskarea1* have been interpreted as usual floorplanning areas used for better chip utilization. Now, the procedure *hdi::pr setInstance* can be used to define these two areas as dynamic areas. Next, additional NGC files (that means all dynamic modules but the dispatcher) are assigned to these dynamic areas (the dispatcher is already assigned to the dynamic areas, since it has been used at project creation). Finally the creation of the static netlist is invoked. This is realized by an independent script called *launch1.sh*. While this script is running, the project is saved and *PlanAhead* is closed. Control returns to *matrix_implementation* which waits until *launch1.sh* has finished. The result is an NCD file representing the placed and routed netlist of the static part.

The Dynamic Parts

In order to create the dynamic modules, *PlanAhead* is started in batch mode again. The corresponding command is:

```
planAhead -mode batch -source $pwd/script/dynamic.tcl
```

The script *dynamic.tcl* makes use of the procedure *hdi::run schedule* to schedule the creation of the NCD files representing the dynamic modules. Next, the procedure *hdi::run launch* is used to start it. An separate script called *launch2.sh* is started. While this script is running, the project is saved and *PlanAhead* is closed. Control returns to *matrix_implementation* which waits until *launch2.sh* has finished.

The scripts *launch1.sh* and *launch2.sh* both make use of subscripts called *runme.sh* which contain the actual functionality and are stored in the subdirectories that belong to the static and the dynamic modules. These subscripts are called once for each module. Figure 7.25 illustrates how *runme.sh* uses the ISE tools to generate the NCD files.

```
ngdbuild -intstyle ise -modular module -active dynamic -uc "dynamic.ucf" "top.edn"
map -intstyle ise "top.ngd"
par -intstyle ise "top.ncd" -w "top_routed.ncd"
```

Figure 7.25: Part of *runme.sh* that calls the ISE tools

PR Assemble

After creation of the dynamic parts, the *PlanAhead* project contains a set of NCD files either representing the static part or a dynamic module. Finally, these parts have to come together. For this, *PlanAhead* is started in batch mode again. The corresponding command is:

```
planAhead -mode batch -source $pwd/script/assemble.tcl
```

The script *assemble.tcl* uses the procedure *hdi::pr assemble* to create the shell script *assemble.sh*. Next, the project is saved, *PlanAhead* is closed and *assemble.sh* is called. This script collects all created NCD files in one directory and calls the ISE tools *PR_verifydesign* and *PR_assemble* to merge the NCD files.

PR_assemble merges the NCD files in a way that allows the usage of feed-through routes. It generates the merged NCD files as well as the corresponding bitfiles. These are:

static_full.bit	the full bitfile containing the static part as well as the initial dynamic modules
task0_blank.bit	a partial bitfile clearing the first dynamic area (it contains nothing but the feed-through routes)
task1_blank.bit	a partial bitfile clearing the second dynamic area (it contains nothing but the feed-through routes)
taskarea0_Adder_csym.bit	a partial bitfile representing the *Adder* in the first dynamic area
taskarea1_Adder_csym.bit	a partial bitfile representing the *Adder* in the second dynamic area
...	(since two dynamic areas are used, each POL class is represented twice)

Completion

After the bitfiles have been created, they are copied to the directory *bitfiles*. This is realized via the program *bitcopy* which additionally removes the dispensable preamble (located before the synchronization word *0xFFFFFFFF*) from the bitfiles since this preamble would disturb the ICAP controller. Furthermore the elf-file representing the Scheduler is copied to this directory. Next, the file *xmd.ini* (which is used to upload the partial bitfiles to the correct position in the DDR-RAM) is adapted to the bitfile names. Finally, all log files are displayed. If an error occurred, PlanAhead can be started in GUI mode for further investigations.

If the bitfile generation was successful and the target FPGA is connected, a user of the Framework must simply navigate to the directory *bitfiles* enter the following commands:

```
# impact -batch < usbdownload.dat
# xmd
```

The configuration files *usbdownload.dat* and *xmd.ini* automate the upload of the full bitfile via *iMPACT* and the upload of the partial bitfiles via the *xmd*[14] tool.

Thereafter, the Framework should be up and running.

[14] Xilinx Microprocessor Debugger

7.6 Scheduler

The Scheduler is a software program running on an embedded processor, which is part of the FPGA design. It is responsible for the reconfiguration management. That means, it conducts the dynamic creation as well as the dynamic destruction of instances and controls the Short Reconfiguration as well as the Long Reconfiguration. Furthermore, it monitors the Communication Matrix. For this, it provides a Scheduler Shell that allows the user of the Framework to observe and to control the behavior of the generated system.

7.6.1 Processor Subsystem

Since the Scheduler is a software program, it is dependent on the implementation of an embedded processor and the corresponding infrastructure (namely buses and controllers). Xilinx offers two types of embedded processors: PPC and MircoBlaze. The PPC is a hard-core processor. That means it is a hardwired subcomponent of the FPGA which is not reconfigurable (see chapter 2.1). The utilization of a PPC depends on its availability (e.g. only the FX chips of the Virtex-4 series contain a PPC). In contrast, the MicroBlaze is a soft-core processor. That means it is implemented entirely in the configurable parts of the FPGA. Therefore it can be utilized on every FPGA — but in contrast to the PPC it consumes reconfigurable logic.

The Scheduler has been implemented in a way that allows its usage on the MicroBlaze as well as on the PPC. Due to the fact, that all example designs have been tested on the ML405 board, equipped with a Virtex-4 FX20 (which contains a PPC) the implemented System Templates make use of the PPC and the corresponding infrastructure. The processor subdesign contains the following components:

- the PPC itself
- a DDR-RAM controller (used to store the software as well as the partial bitfiles)
- a BRAM controller (used for booting)
- a JTAG debug interface (used by the xmd tool to upload the partial bitfiles)
- a RS232 UART user interface (used to provide a Scheduler Shell)
- an ICAP controller (used to perform the partial reconfigurations)
- a matrix control interface (used to observe and to manage the Communication Matrix)

The processor subdesign has been created using the EDK (Embedded Development Kit). The generated NGC files are used as direct input for PlanAhead. The script *matrix_synthesis* copies them from the directory *ppc/implementation* to the directory *planahead_data/static*. Figure 7.26 illustrates the structure of the generated system including the processor subdesign as well as the Communication Matrix.

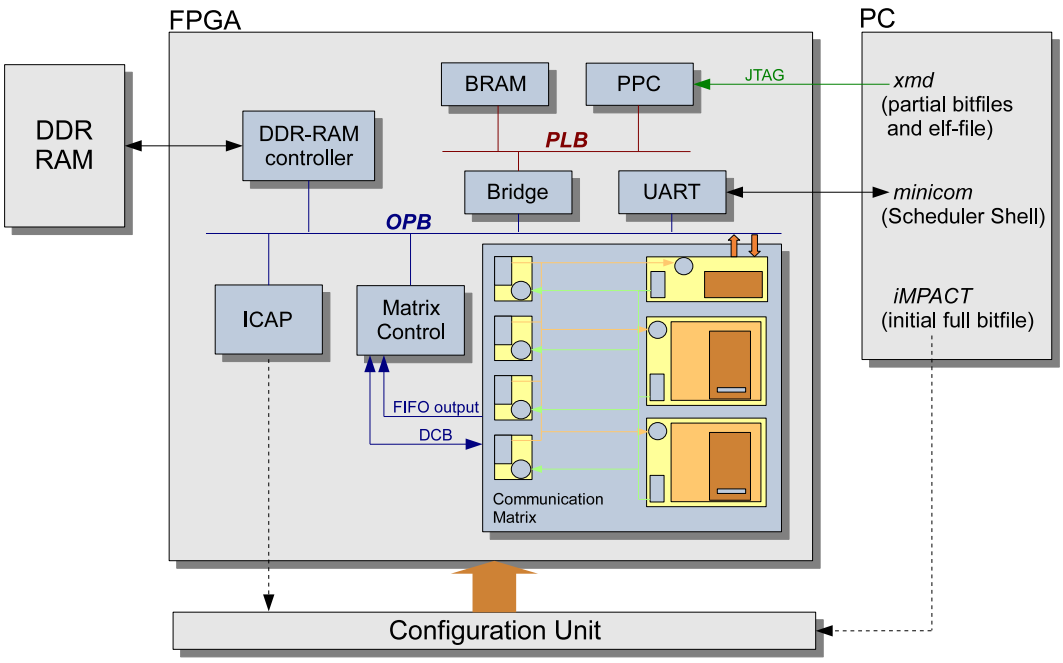


Figure 7.26: Generated system including the processor subsystem as well as the Communication Matrix

7.6.2 The Matrix Control Interface

The Scheduler has to be able to interact with the Communication Matrix. This is realized via the matrix control interface. It maps bus addresses to VHDL signals and therefore allows the Scheduler software to observe and to control the behavior of particular components via address accesses. The managed components are:

- the FIFOs — the Scheduler is able to read the filling level as well as the topmost messages; this information is used as input for the scheduling algorithm (especially for Short Reconfigurations)
- the active multiplexers of the Task Areas — the Scheduler is able to set the class ID and the instance ID of the object loaded to the dynamic area; this is used whenever a Long Reconfiguration is performed
- the DCB — used for the interactions between Scheduler and hardware objects, which is needed for the dynamic creation and destruction of objects as well as for the swap-out process (see chapter 6.3.1).

DCB

If the Scheduler wants to swap out a running instance, this instance has to be notified. The reason is that a hardware object can only be interrupted at particular interruptible states. Furthermore, parts of the context have to be saved to the Context Memory before an instance can be swapped out (see chapter 7.2.5). The Scheduler makes use of the DCB to send the corresponding object the DCB command *ready for reconfiguration*. Then it waits, until the instance sends a *ready for reconfiguration* command via the DCB. The Scheduler waits for a preassigned period (e.g. 3 seconds). If the instance fails to answer in time, the Scheduler deactivates the instance without saving its context and reports an error. This situation is comparable to a *kill -9* under Linux. After such a rigorous kill of the instance the Framework cannot assure the correct behavior of the system any longer.

If an instance wants to create a new object, it uses the DCB to send the Scheduler the command *new instance request*. The corresponding data is the class ID of the new instance. As response, the Scheduler checks for a free instance ID and adds a new entry in the instance list. Next, the Scheduler sends the DCB command *new instance reply*. The corresponding data is the instance ID of the new instance.

If an instance receives a Destruction Message, it ultimately stops its execution. At the very end, it uses the DCB to send the Scheduler the command *self deletion*. In answer to this, the Scheduler removes the instance from the list of active instances and marks the corresponding dynamic area as unused. Table 7.14 lists the available DCB commands and the corresponding DCB data.

Each DCB message sent to the Scheduler consists of a DCB command, DCB data and the number of the Task Area from which it comes. Each DCB message sent to an hardware object consists of a DCB command, DCB data and the corresponding class ID and instance ID. The Scheduler has to assign the Task Area number to the correct class ID and instance ID to be able to process the incoming data correctly.

DCB Command	Description	DCB Data
	Instance to Scheduler	
00001	new instance request	new class ID
00010	ready for suspension	-
00011	self deletion	-
	Scheduler to Instance	
00001	new instance reply	instance ID
00010	prepare for suspension	-

Table 7.14: List of available DCB commands

7.6.3 ICAP Controller

The ICAP controller coming with the Xilinx EDK has been designed to be flexible and easy to handle. However, it does not provide a high throughput. Therefore, using the Xilinx ICAP controller leads to very long reconfiguration times. This is primarily caused by the fact that each data packet has to be transported multiple times (see figure 7.27).

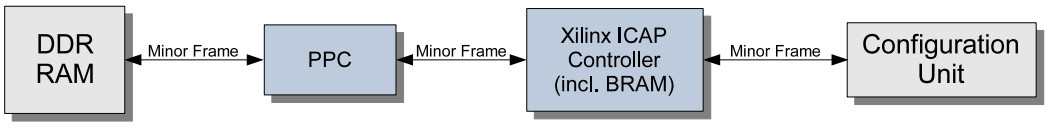


Figure 7.27: Xilinx ICAP controller - Each Minor Frame has to be transported three times: from the DDR-RAM to the PPC, from the PPC to the BRAM of the ICAP Controller, and finally from this BRAM to the ICAP

For Runtime Scheduling, the reconfiguration speed has to be as high as possible (see chapter 5). Due to this, the decision was made to implement a customized ICAP controller based on [40] which makes use of DMA (Direct Memory Access). Thus, each Minor Frame is transported only once (see figure 7.28), which significantly increases the reconfiguration speed.

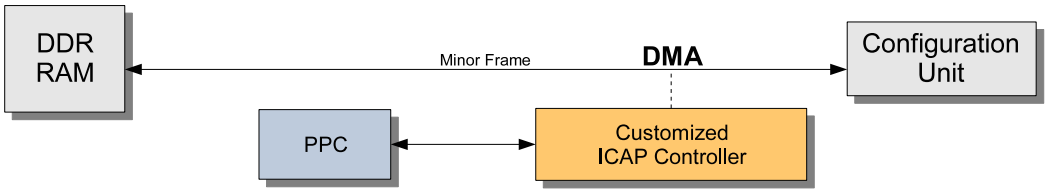


Figure 7.28: Customized ICAP controller - Each Minor Frame is transported only once

7.6.4 The Software

The Scheduler software has been created using the SDK (Software Development Kit) which is part of the EDK. It is written as a C program which is running directly on the embedded processor (no underlying operating system, no multithreading, no virtual memory). It consists mainly of two parts: First, the Scheduler Shell which allows user interaction and direct control of the Communication Matrix as well as the Scheduler. Second, an Orchestrator which performs the necessary operations (instance management, DCB interaction, Short Reconfiguration, Long Reconfiguration and connection management) automatically.

Instance Management

For instance management, the Scheduler makes use of a struct named *matrix_data* to store all informations about classes and instances (see figure 7.29).

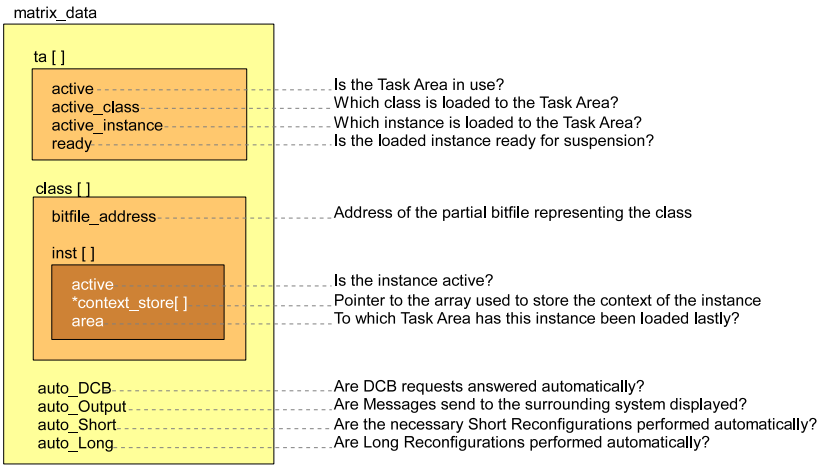


Figure 7.29: Struct *matrix_data* storing all informations about classes and instances

Every class is related to a partial bitfile which is stored in the attached DDR-RAM. Several instances of the same class share one single bitfile. To enable the upload of these partial bitfiles via *xmd*, the partial bitfiles are not stored in a dynamic array but in a dedicated area of the DDR-RAM. This area begins at address *0x03000000* and is not touched by the linker at all. In contrast, the context of the instances is stored in the DDR-RAM as a dynamic array (created on demand).

The dynamic instantiation of an instance leads to the search for an unused (inactive) instance ID. If such an ID is found, the corresponding instance is set to active and the context is initialized (which means that the corresponding array storing the context is created, completely zeroed and the bits representing the first word of the Context Memory are configured to represent the instance ID). This can be done automatically by the Orchestrator, or manually using the Scheduler Shell: the command *addInst* adds an instance to a given class. The command *context* displays the context of a given instance. In order to dynamically destroy an instance, the corresponding instance and the corresponding Task Area are set to inactive and the array storing the context is deleted. The Scheduler Shell offers the command *disable* which allows to deactivate a Task Area manually. Furthermore, it provides the command *ls sw* which returns the actual content of *matrix_data*.

DCB

The DCB bus can either be controlled by the Orchestrator or by the Scheduler Shell. In the first case, the Orchestrator automatically responds to DCB requests coming from running instances. If an instance is asked by the Scheduler to suspend itself, the corresponding answer (*ready for suspension*) is received by the Orchestrator which sets the ready bit of the instance to *1*. This way, the Scheduler is informed that it can swap out the corresponding instance.
The Scheduler Shell provides the following commands to control the DCB bus manually:

writeDCB class inst command	Writes the DCB command *command* with destination class *class* and instance *inst* to the DCB
readDCB	Reads a single word from the DCB
DCBnewInst	Special command to send a *new_instance_reply*

Short Reconfiguration

A Short Reconfiguration is a change from an instance of a class to an other instance of the same class. This requires a read-back of the old context (swap-out) and a configuration of the new context (swap-in). A reconfiguration of the whole dynamic area is not needed at all. If an instance has been loaded to Task Area 1, suspended and shall now be loaded to Task Area 2, the corresponding context has to be adopted. This is due to the fact that the bits of the upper BRAMs are organized other than those of the lower BRAMs. The context adaption is realized by the function *context_move()*.

The Orchestrator is able to perform necessary Short Reconfigurations automatically. If it is configured to do so, it monitors the outputs of the Class Buffer FIFOs. If an instance of class A is currently active and the topmost message of Class Buffer A is heading for an inactive instance of this class, a Short Reconfiguration loading the addressed instance is triggered. The Scheduler Shell offers the command *shortReconf* which allows to invoke a Short Reconfiguration manually. If a manually suspended instance does not answer in time, the user is asked to press *ESC* to kill this instance, or to press an other key to keep it running. The commands *ls hw* and *ll hw* return a list of all used FIFOs and the corresponding attributes (filling level and topmost message).

Long Reconfiguration

A Long Reconfiguration is a change from an instance of a class to an instance of another class. This requires a read-back of the old context (swap-out), a reconfiguration of the dynamic area and a configuration of the new context (swap-in). The Orchestrator is able to perform the Long Reconfigurations automatically. In contrast to Short Reconfigurations, the time a Long Reconfiguration has to take place is not strictly determined. The

Scheduler uses the resulting degrees of freedom to minimize the number of needed reconfigurations as well as to prevent data loss caused by full FIFOs. For this, it monitors the development of the filling levels of the FIFOs. The more data sent to a class the higher its priority.
The Scheduler Shell offers the command *longReconf* to invoke a Long Reconfiguration manually.

Connections and Messages

In order to keep the number of consumed hardware resources as small as possible, only particular parts of *System InOut*[15] are implemented in hardware. All messages (data messages as well as Connection Messages) sent to or coming from the Scheduler are handled in software. Only particular *Signals* and *Slots* that are connected to a hardware component (e.g. an audio codec) are managed in hardware. This significantly reduces the size (in LUTs) of the Connection Manager. If and how channels are handled in hardware is determined by the used System Template. In order to handle the messages correctly, the Scheduler contains functions to tag outgoing and to untag incoming messages.

The Orchestrator is able to receive and to display incoming data. For this it makes use of the function *check_Output()* which is part of the file *application.c*. This file contains functions which can/shall be modified in an application specific way. The raw version of *check_Output()* just displays all received data.

Table 7.15 lists the commands the Scheduler Shell provides for managing connections and messages manually.

ls con	Lists all connections on input *Signals* established by the software part of the Connection Manager
connect	Sends a Connection Message establishing a connection
disconnect	Sends a Connection Message dissolving a connection
put data sig	Sends a message via *Signal sig* (depends on the established connections)
put data class inst slot	Sends a message to Slot *slot* of instance *inst* of class *class* (independent from the Connection Manager)

Table 7.15: Shell commands for managing connections and messages

Furthermore, the Scheduler Shell offers the command *play* which leaves the Shell and hands control over to the application specific function *Application_Interface()* which is part of *application.c*. This function can/shall be used to provide an application specific interface. Its raw version just returns control to the Shell.

[15] see chapter 6.2.3

The Big Picture

The Orchestrator and the Scheduler Shell can be used concurrently without any restrictions. This is realized by a function called *polling()* which subsequently starts the different parts of the Orchestrator and is called whenever the Scheduler Shell is waiting for a keystroke. The Shell allows to activate and deactivate the Orchestrator via *CTRL+Y* and *CTRL+X*. The command *setConfig* allows a more fine-grained control. It makes it possible to activate and deactivate particular parts (Long Reconfiguration, Short Reconfiguration, Message control, DCB control) of the Orchestrator. Furthermore, the Scheduler Shell offers the command *debug* which can be used to activate and deactivate debug messages (e.g. the Orchestrator can be configured to report an info message for each performed Long Reconfiguration). The command *reset* can be used to reset the Scheduler, the Communication Matrix or the whole Framework.

The Scheduler is compiled with the cross compiler *powerpc-eabi-gcc*. The C sources are stored in *ppc/SDK_projects/Scheduler*. The output of the compiler is a PPC binary called *Orchestrator.elf*. It is copied to *bitfiles/executable.elf* and uploaded via *xmd* at system start-up. Figure 7.30 illustrates the corresponding output of the Scheduler Shell using the adder-multiplier example.

At first, the Orchestrator is activated (pressing CTRL+Y). Thus, the class *Dispatcher* (which has been configured to the dynamic areas via the full bitfile) is started. It creates two instances of *Adder* and one instance of *Multiplier*. Next, it establishes the connections between them. Four Connection Messages are sent to *System InOut*. They are processed by the software part of the Connection Manager. The command *ls con* displays the corresponding established connections. The command *ls sw* lists the content of *matrix_data* which can be used to monitor the created instances. At this point, *put* is used to send four data messages (*1*, *2*, *3* and *4*) to the four input *Signals*. The data is processed by the Framework and the corresponding output is send to the surrounding system (and thus to the Scheduler). The received value *21* is printed out.

```
Welcome to the OO-DPRF Version 1.0.1
Initializing...

Starting Shell...

Orchestrator is deactivated.
Type [CTRL-Y] to start it.

Type 'help' to get help.

DPRF>
Added new connection to signal 0. (class: 1,
inst: 0, port: 0)
Added new connection to signal 1. (class: 1,
inst: 0, port: 1)
Added new connection to signal 2. (class: 1,
inst: 1, port: 0)
Added new connection to signal 3. (class: 1,
inst: 1, port: 1)

DPRF> ls con
connections
        Signal 0:
                class: 1, inst: 0, port: 0
        Signal 1:
                class: 1, inst: 0, port: 1
        Signal 2:
                class: 1, inst: 1, port: 0
        Signal 3:
                class: 1, inst: 1, port: 1

DPRF> ls sw
matrix_data
        auto_DCB: 2
        auto_Output: 1
        auto_Short: 2
        auto_Long: 2
        ta[0]
                active: yes
                active_class: 3
                active_instance: 0
                busmacro: on
                reset: off
                ready: no
        ta[1]
                active: yes
                active_class: 1
                active_instance: 0
                busmacro: on
                reset: off
                ready: yes
        class[0]
                bitfile_address[0]: 3000000
                bitfile_address[1]: 3250000
        class[1]
                bitfile_address[0]: 3025000
                bitfile_address[1]: 3275000
                instance[0]
                        area: 1
                instance[1]
                        area: 1

                    . . .

                    . . .

        class[2]
                bitfile_address[0]: 304A000
                bitfile_address[1]: 329A000
                instance[0]
                        area: 1
        class[3]
                bitfile_address[0]: 306F000
                bitfile_address[1]: 32BF000
                instance[0]
                        area: 0


DPRF> put 1 0
DPRF> put 2 1
DPRF> put 3 2
DPRF> put 4 3

Data: 21


DPRF> play
(0+1)*(2+3) = 5 (5)
(4+5)*(6+7) = 117 (117)
(8+9)*(10+11) = 357 (357)
(12+13)*(14+15) = 725 (725)
(16+17)*(18+19) = 1221 (1221)
(20+21)*(22+23) = 1845 (1845)
(24+25)*(26+27) = 2597 (2597)
(28+29)*(30+31) = 3477 (3477)
(32+33)*(34+35) = 4485 (4485)
(36+37)*(38+39) = 5621 (5621)
(40+41)*(42+43) = 6885 (6885)
(44+45)*(46+47) = 8277 (8277)
(48+49)*(50+51) = 9797 (9797)
(52+53)*(54+55) = 11445 (11445)
(56+57)*(58+59) = 13221 (13221)
(60+61)*(62+63) = 15125 (15125)
(64+65)*(66+67) = 17157 (17157)
(68+69)*(70+71) = 19317 (19317)
(72+73)*(74+75) = 21605 (21605)
(76+77)*(78+79) = 24021 (24021)
(80+81)*(82+83) = 26565 (26565)
(84+85)*(86+87) = 29237 (29237)
(88+89)*(90+91) = 32037 (32037)
(92+93)*(94+95) = 34965 (34965)
(96+97)*(98+99) = 38021 (38021)
(100+101)*(102+103) = 41205 (41205)


DPRF> setConfig 0 0 0 0

DPRF> longReconf 1 2 0
WARNING:
Waiting for Suspend Acknowledge from Class 2, Inst 0.
Press ESC to kill..........

Class 2, Instance 0 has been killed!

DPRF>
```

Figure 7.30: The Scheduler Shell

The application specific command *play* sends a sequence of data values to the Communication Matrix and compares the returned value to the result calculated in software. To demonstrate the functionality of the Shell, *SetConfig* is used to deactivate the Orchestrator. Next, a Long Reconfiguration is invoked manually. Since the DCB unit of the Orchestrator has been deactivated the *ready*-bit of the running instance is not set to *1* although the instance has sent a *ready_for_suspension* message via DCB. Thus, a warning message is printed out and the Scheduler waits for user interaction. In figure 7.30 ESC is pressed to kill the instance.

8 Results

In order to evaluate the design and the implementation of the Framework, two example applications have been used. This chapter presents these two examples as well as the corresponding measurements. Furthermore some extrapolations are used to illustrate the implementation-independent possibilities and limitations of the Framework. Finally, the measured and calculated results are interpreted.

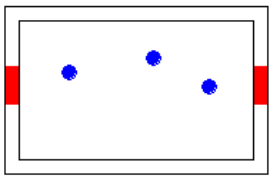
8.1 Example Implementations

The example implementations have been realized on the same hardware as presented in chapter 7. The POL-Compiler was running on a Linux PC. For bitfile generation, the Xilinx tools ISE 9.1i SP2 enhanced by the PREA[1] patch and *PlanAhead* 10.1i have been used. The Emulator was running on a Windows PC which was equipped with an Intel Core 2 CPU running on 1,67 GHz. For testing of the generated bitfiles, the Xilinx Evaluation Board ML405 containing a Virtex-4 FX20 has been used. All implementations are realized based on a System Template providing two dynamic areas.

8.1.1 Pong Game

The first example implementation focuses on the functionality of the Framework itself. The dynamic instantiation of reconfigurable modules, their alternating execution (swap-out, swap-in) and the final removal of instances has to work correctly in order to be able to do Runtime Scheduling. To verify this functionality, the intuitive and playful example Pong has been chosen. Pong is is one of the earliest arcade video games and was one of the first video games to reach mainstream popularity [177]. It is a tennis-inspired game for two players. Each player can move a bar up and down on his edge of the screen and tries to bounce the ball back when it arrives at his side. The implementation used to evaluate the Framework is able to contain more than one ball. New balls can be added to the game by simply pressing a key. A ball can also leave the game when it is missed by a player. To verify the Framework, the balls and bars were implemented as POL classes. The adding of a new ball correlates to the instantiation of a new instance. The removal of a ball correlates to the destruction of an instance.

[1] Partial Reconfiguration Early Access



```
01 class Ball extends ParObj {
02  Slot:
03    int bar1, bar2, clk_in;
04  Signal:
05    int x_pos, y_pos; Ball dead;
06  private:
07    int x=40,y=20,vx=1,vy=1;
08  void calc() {
09    int clk = clk_in.get(0);
10    int b2 = bar2.get(b2);
11    int b1 = bar1.get(b1);
12    if (clk!=0) {
13      x = x + vx;
14      y = y + vy;
15      if (y==0) vy=1;
16      if (y==40) vy=-1;
17      x_pos.emit(x);
18      y_pos.emit(y);
19      if (x==0) {
20        if ((b1 >= y-2) && (b1 <= y+2)) vx=1;
21        else dead.emit(this);
22      }
23      if (x==80) {
24        if ((b2 >= y-2) && (b2 <= y+2)) vx=-1;
25        else dead.emit(this);
26      }
27    }
28  }
29 }
```

```
30 class Bar extends ParObj {
31   Slot:
32     int change_pos, clk_in;
33   Signal:
34     int y_pos, clk_out;
35   private:
36     int y=0;
37   Bar() {
38     y = 20;
39   }
40   void calc() {
41     int c = change_pos.get(0);
42     if ((c==1)&(y<35)) y++;
43     if ((c==2)&(y>05)) y--;
44     int clk = clk_in.get(0);
45     if (clk!=0) {
46        y_pos.emit(y);
47        clk_out.emit(1);
48     }
49   }
50 }
```

```
51 class Dispatcher extends DispObj {
52   Slot:
53     int keys, bar1_y, bar2_y;
54     int ball_x, ball_y, clk_in;
55     Ball ball_dead;
56   Signal:
57     int data, bar1_changepos;
58     int bar2_changepos, clk_out;
59
60   private:
61     Bar bar1 = new Bar();
62     Bar bar2 = new Bar();
63     int b1=0, b2=0;
64     int ball_count=0;
65     int recv=2;
66     int ball_number=0;
67     int[] ballx = new int[15];
68     int[] bally = new int[15];
69
70
71   Dispatcher() {
72     data.connect(world.out0);
73
74     bar1.clk_out.connect(clk_in);
75     bar1.y_pos.connect(bar1_y);
76     bar1_changepos.connect(bar1.change_pos);
77     clk_out.connect(bar1.clk_in);
78
79     bar2.clk_out.connect(clk_in);
80     bar2.y_pos.connect(bar2_y);
81     bar2_changepos.connect(bar2.change_pos);
82     clk_out.connect(bar2.clk_in);
83
84     world.in0.connect(keys);
85   }
86
87   void calc() {
     . . .
101    int t = keys.get(0);
102    if(t==1) bar1_changepos.emit(1);
103    if(t==2) bar1_changepos.emit(2);
104    if(t==3) bar2_changepos.emit(1);
105    if(t==4) bar2_changepos.emit(2);
106    if((t==5)&(ball_count<10)) {
107      Ball b = new Ball();
108      bar1.y_pos.connect(b.bar1);
109      bar2.y_pos.connect(b.bar2);
110      clk_out.connect(b.clk_in);
111      b.x_pos.connect(ball_x);
112      b.y_pos.connect(ball_y);
113      b.dead.connect(ball_dead);
114      ballx[ball_number]=40;
115      bally[ball_number]=20;
116      ball_count++;
117      ball_number++;
118    }
119
120    if(ball_dead.valid()) {
121      Ball b = ball_dead.get();
122      bar1.y_pos.disconnect(b.bar1);
123      bar2.y_pos.disconnect(b.bar2);
124      clk_out.disconnect(b.clk_in);
125      b.x_pos.disconnect(ball_x);
126      b.y_pos.disconnect(ball_y);
127      b.dead.disconnect(ball_dead);
128      ball_count--;
129      ball_number--;
130      finish b;
131    }
132  }
133 }
```

Figure 8.1: POL source code of the Pong example

Figure 8.1 shows the POL source code of the Pong example. It consists of three classes: *Ball, Bar* and *Dispatcher*. An instance of *Ball* represents a ball. An instance of *Bar* represents a bar. *Dispatcher* derives from *DispObj* and is therefore automatically instantiated at system start-up. It manages the dynamic creation and destruction of balls as well as the communication of the objects with each other. The two bars are created at system start-up in the constructor of *Dispatcher*. They are not destroyed at all. The *Signals* called *clk_out*

and the *Slots* called *clk_in* are used to synchronize the objects at message layer. These are usual messages and should not be confused with VHDL clock signals. Please note that the *clk_out-Signals* make use of the possibility to add more than one receiving *Slot* to a *Signal* (see line 110 and line 124). The part of the *Dispatcher* that manages the sending and the receiving of synchronization signals via *clk_in* and *clk_out* has been omitted in figure 8.1.

The *Dispatcher* is connected to the outside world via a *Slot* called *keys* and a *Signal* called *data* — *keys* is used to receive keystrokes, while *data* is used to report the position of the balls. If a ball leaves the playing field (size 80x40) on the left or the right side, it sends its object reference to the *Dispatcher* via the *Signal dead* (line 21 and line 25). In answer to this the *Dispatcher* removes all connections from or to the ball via *disconnect* and then removes the ball via *finish* (line 120 – line 130).

Although Pong is a very playful example, the here presented implementation is a real stress test for the Framework. The creation and the removal of balls is correlated to the instantiation and the destruction of instances. Furthermore Pong depends on the establishing and the dissolving of connections at runtime. Since the used System Template contains two dynamic areas, the usage of one *Dispatcher* instance and two *Bar* instances results in the usage of Runtime Reconfiguration. Each instantiated *Ball* increases the demands and leads to more necessary Runtime Reconfigurations. The synchronization via the *clk*-messages has been done in a way that causes a deadlock as soon as a data message is lost. Thus, erroneous implementations of the instance handling, the context management, the connection management or the message transport would be visible immediately. In other words: a running Pong game indicates that all components of the Framework are working correctly.

```
package application;
import pol.*;
import components.*;

public class JSB {
         public static void main(String[] args) {
                Dispatcher disp1 = new Dispatcher();
                Control control = new Control();
                control.getControlDevice(Constants.usedOutputs);

                PongIn PongButtons = new PongIn();
                PongButtons.getInputDevice(POL.In[0].myFifo);

                PongOut pong = new PongOut();
                pong.getOutputDevice(POL.Out[0].myFifo);

                disp1.go();
        }
}
```

Figure 8.2: Implementation of JSB.java for the Pong example

Emulation

After writing the POL sources, the POL-Compiler is started. It generates the VHDL files used for synthesis as well as the Java files used for emulation. In order to be able to emulate the Pong example, a class emulating the outside world is needed. Figure 8.2 illustrates the implementation of this class. In order to provide an user interface that blends in well with Pong, the input class *PongIn* and the output class *PongOut* have been implemented. *PongIn* provides a set of push buttons which can be used to move the bars and to create a new ball. *PongOut* provides a graphical view of the playing field. Figure 8.3 shows a screenshot of the Pong emulation.

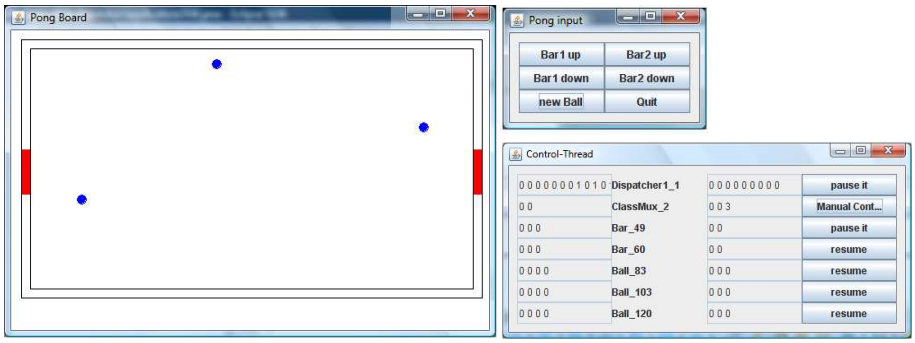


Figure 8.3: Screenshot of the Emulator

In its first implementation, the Pong source code contained a few errors in data message handling and the synchronization via the *clk*-messages. All these errors could be found early, during the emulation, since the Emulator exactly reproduces the behavior of the hardware at message layer. For this, the GUI of the *Control-Thread* emulating the Scheduler was very important. For synchronization, it made it possible to detect the exact error source, since it shows the filling level of all used buffers.

For the performance evaluation of the Emulator, the emulation was monitored using the JConsole tool which is part of the JDK[2]. After starting the Pong example, 92 threads are running. This is due to the fact that each functional block of the Emulator (e.g. the buffers) is represented by a thread in order to represent the functionality of the hardware. The 92 threads are:

[2]Java Development Kit [178]

- 3 POL objects (*Dispatcher* and two instances of *Bar*)
- 18 *Slots*
- 11 *Signals*
- 35 FIFO objects
- 1 *ClassFIFO* representing the Communication Matrix
- 1 *Control-Thread* representing the Scheduler
- additional internal Java classes (such as TimerQueue)

The instantiation of a *Ball* caused the creation of 6 additional *Slots* (3 *Slots* directly visible in POL and 3 additional internal *Slots*), 4 additional *Signals*, 9 additional FIFO objects, and of course 1 additional POL object representing the new ball.

The memory usage was quite independent from the number of instantiated objects: it stayed at about 5 Mb. The CPU usage primarily depended on the usage of the exact Scheduler emulation. As long as all POL objects were allowed to run all the time (Scheduler emulation turned off) the CPU usage was about 60% plus 3% per ball. Once the exact Scheduler emulation was activated, the CPU usage was increased by about 20% (see figure 8.4).

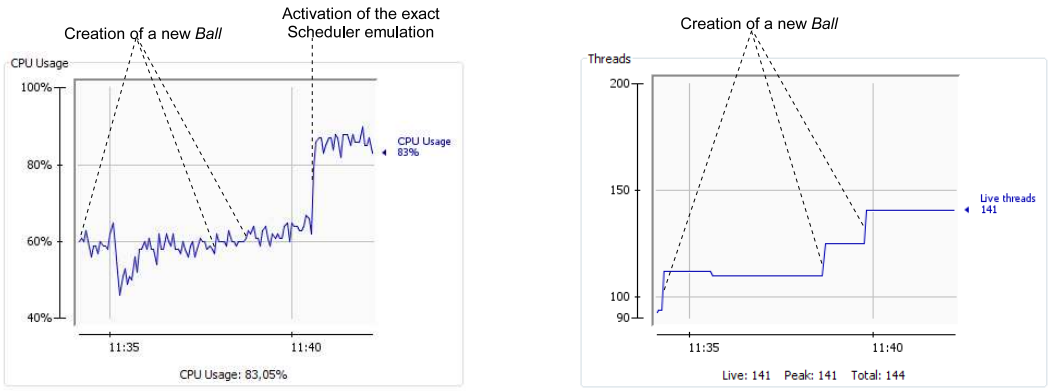


Figure 8.4: Overview of the CPU and memory usage in JConsole

High Level Synthesis

The first step of the synthesis is the translation of the POL sources to VHDL. This is realized by the POL-Compiler and takes about 0.5 seconds. As described in chapter 6.4, the POL-Compiler translates each POL class to a FSM which is located in a separate VHDL entity. The number of used states is optimized via state merging (two subsequent calculations are merged into one state as long as they do not depend on each other). The Pong example is translated as follows:

	Number of states before optimization	Number of states after optimization	Number of states used for context-management
Ball	137	100	44
Bar	90	67	36
Dispatcher	350	233	64

Before the VHDL code is used for bitfile generation, it can be simulated using Modelsim. Figure 8.5 illustrates the first start of the *Dispatcher*. In order to simulate DPR, an additional multiplexer has been introduced. This multiplexer decides, which instance is running when. It is controlled by tcl-scripts emulating the behavior of the Scheduler, since the processor subsystem is not part of the behavioral simulation.

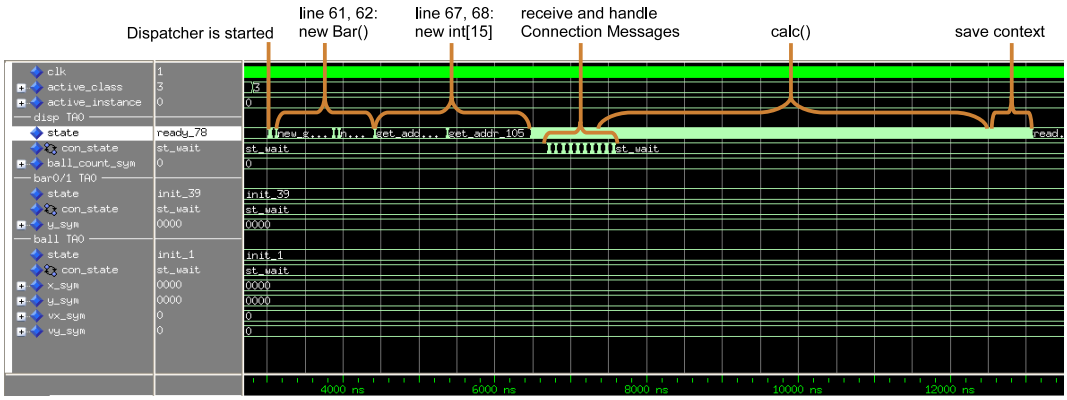


Figure 8.5: First execution of the *Dispatcher*

Table 8.1 lists a more detailed analysis of the number of clock cycles required by the generated VHDL code.

The large number of clock cycles needed for an *emit* is caused by the possibility to connect multiple *Slots* to one *Signal*. Due to this, each *emit* requires at least one BRAM access to read the number of connected *Slots* and additional BRAM accesses to get the addresses of all connected *Slots*. As a consequence, the minimum number of required clock cycles to handle one single data item (from receiving to transmitting) is 10 (*get*: 2, assign: 1, *emit*: 7).

Operation	Time consumption in clock cycles
assignment	1
while loop	2 per iteration
do loop	3 per iteration
get	2
connect	2
emit	3 · #receivers + 4

Table 8.1: Time consumption of the generated VHDL code

All buffers and multiplexers in the Communication Matrix require 2 clock cycles per data item. In the current implementation the Communication Matrix is running with 100 MHz. The payload of the transported messages is 16 bit. This means that the Communication Matrix has a maximum throughput of 100 MB/s. In contrast, the generated VHDL code has a maximum throughput of 20 MB/s. Hence, in the current version of the Framework the generated VHDL code is the bottleneck.

Figure 8.6 illustrates the behavioral simulation of the Pong example. It shows the execution of two so called turns. One cycle consists of the subsequent execution of all active instances and requires 16 μs. Please note, that the times needed to perform a partial reconfiguration are not considered in figure 8.6, at all. However, if the reconfiguration times are known, they can simply be added to the behavioral simulation. The corresponding dead times just have to be added to the tcl-scripts that emulate the Scheduler.

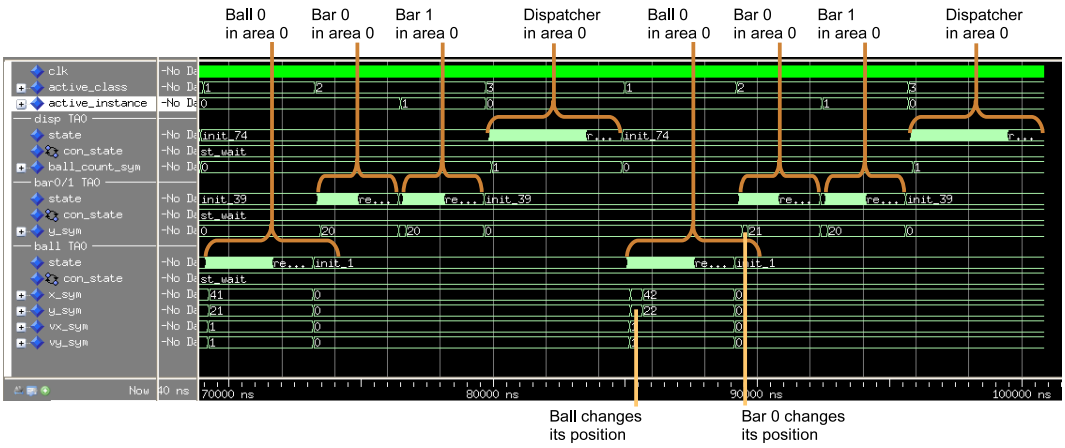


Figure 8.6: Behavioral simulation of the Pong example

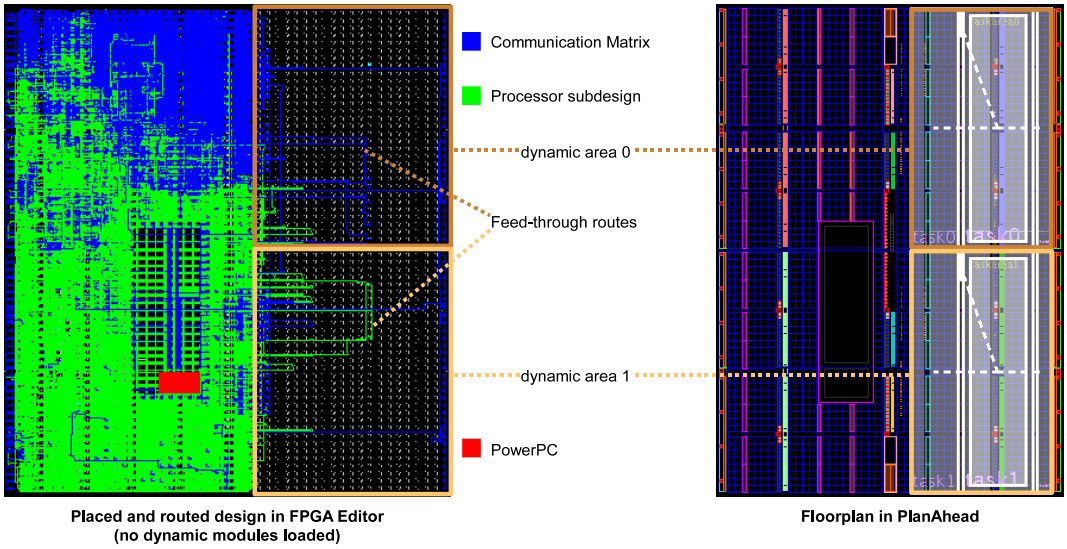


Figure 8.7: Partitioning of the Virtex-4 FX20

Bitfile Generation

In order to execute the generated design on an FPGA it has to be translated from VHDL to (partial) bitfiles. This has been realized using ISE 9.1 and PlanAhead. Before the bitfile generation can be started, the target FPGA has to be partitioned into dynamic and static areas and these areas have to be placed. Figure 8.7 shows the partitioning of the Virtex-4 FX20. The FPGA contains 8 544 Slices, 68 BRAMs, one PowerPC and two ICAPs. Each dynamic area contains 2 048 Slices and 16 BRAMs. In consequence, the static area contains 4 448 Slices and 36 BRAMs. The PowerPC is part of the static area. The bitfile generation (low level synthesis, place and route) of the Pong example took 91 minutes.

Component	#Slices	Part of the FPGA	#BRAMs
Virtex-4 FX20	8 544	100%	68
static area	4 448	52%	36
Communication Matrix	1 695	20%	7
Processor subsystem	2 753	31%	3
dynamic area 0	2 048	24%	16
dynamic area 1	2 048	24%	16
Dispatcher	2 016	24%	1
Bar	1 119	13%	1
Ball	1 316	15%	1

Table 8.2: Resource consumption of the generated VHDL code

The Communication Matrix requires 20% of the Slices. Please note that the size of the Communication Matrix (especially the number of used BRAMs) depends on the POL source code, since the Matrix contains one Class Buffer per POL class. The processor subsystem consumes 31% of the Slices. Table 8.2 shows an overview of the ressource consumption. The Pong example supports the instantiation of up to 10 *Balls*. Using the POL compiler in combination with conventional hardware (without utilizing DPR) this would lead to the consumption of approximately $1\,695 + 2\,016 + 2 \cdot 1\,119 + 10 \cdot 1\,316 =$ 19109 Slices, which are 223% of the FPGA (which means that the design would not fit onto a FX20).

Hardware Execution

All input and output channels of *System InOut* were connected to the OPB and thus to the Scheduler running on the PowerPC. Due to this, the Scheduler Shell could be used to monitor the output of the Pong example and to produce input data. The application specific output parser of the Scheduler receives the incoming data messages (y-position of *Bar0*, y-position of *Bar1*, number of *Balls* and the positions of the *Balls*) and prints out a message, if a value has changed.

```
DPRF>

Added new connection to signal 0.
(class: 3, inst: 0, port: 0)

Left Bar: 20
Right Bar: 20
0 Ball(s).

DPRF> ls con
connections
        Signal 0:
                class: 3, inst: 0, port: 0

DPRF> put 1 0
Left Bar: 21

DPRF> put 1 0
Left Bar: 22

...

DPRF> put 1 0
Left Bar: 30

DPRF> put 3 0
Right Bar: 21

DPRF> put 4 0
Right Bar: 20

                    ...
```

```
                              ...

DPRF> put 5 0

1 Ball(s) x: 40 y: 20
1 Ball(s) x: 41 y: 21
1 Ball(s) x: 42 y: 22

...

1 Ball(s) x: 78 y: 22
1 Ball(s) x: 79 y: 21
1 Ball(s) x: 80 y: 20
1 Ball(s) x: 79 y: 19

...

1 Ball(s) x: 3  y: 24
1 Ball(s) x: 2  y: 23
1 Ball(s) x: 1  y: 22
1 Ball(s) x: 1  y: 21
1 Ball(s) x: 0  y: 20

Self delete detected!
(Area:1, Class:1, Inst:0)

0 Ball(s).

DPRF>
```

Figure 8.8: Scheduler Shell output of the Pong example

The Pong example makes use of the standard ICAP controller coming with the EDK. Table 8.3 shows the measured reconfiguration times. The time for the Short Reconfiguration is the sum of the times for swap-in, swap-out and the DCB interaction between Scheduler and reconfigurable module.

Swap-Out:	671 μs
Swap-In:	518 μs
Short Reconfiguration:	1 209 μs
Long Reconfiguration:	19 146 μs

Table 8.3: Measured reconfiguration times

The Scheduler loaded the *Dispatcher* to Task Area 0. The instances of *Bar* and the instances of *Ball* have been loaded to Task Area 1. Figure 8.9 illustrates one scheduling cycle with 3 instances of *Ball*.

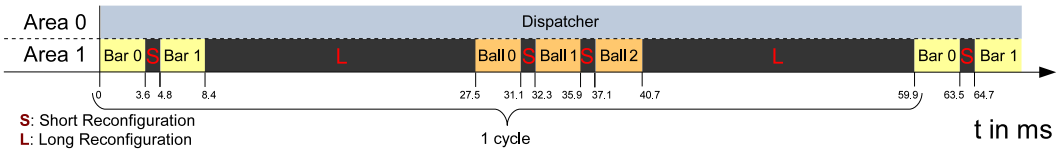


Figure 8.9: Scheduling cycle using the example of 3 active balls

Since the Scheduler Shell does not provide a very haptic user interface for Pong, the final step regarding the Pong example was to implement a Java program that connects to the serial interface and displays the received coordinates in a graphical interface. Figure 8.10 shows a screenshot of this Java program. One can see that the behavior of the hardware is identical to that of the software emulation.

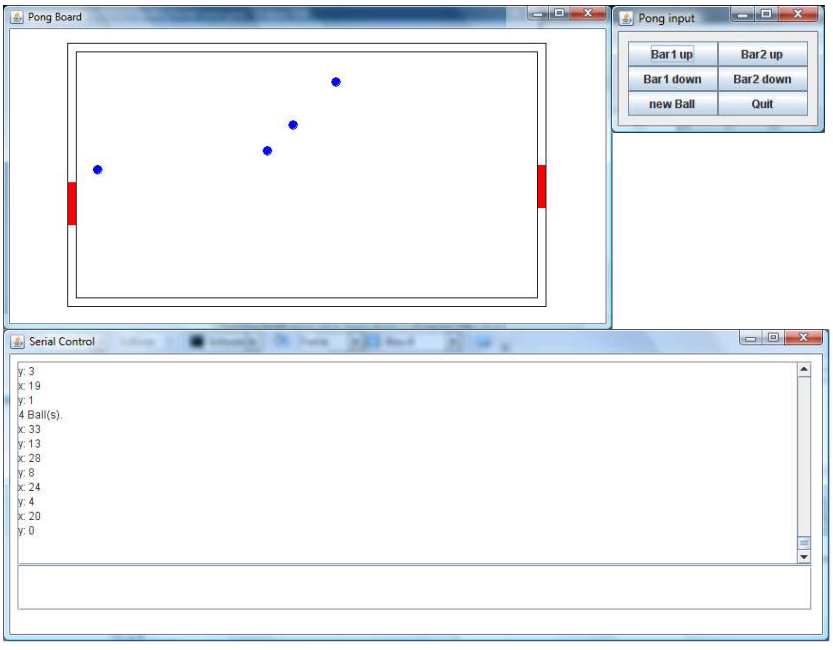


Figure 8.10: Java serial interface output of the Pong example

8.1.2 Audio Filter

As shown in chapter 5, FPGAs usually are used to handle data streams with a very high throughput. To evaluate the performance of the Framework regarding data streams, an audio DSP application has been implemented. It makes use of the AC97[3] chip which provides stereo audio inputs and outputs. All inputs and outputs have a size of 16 bit. The audio example contains two hardware objects representing two different effects: a high pass filter and a low pass filter. The filter objects are instantiated to activate the corresponding effect on a single channel and removed to deactivate it. Due to the two channels (left and right), each filter object can be instantiated twice. The used System Template provides two Task Areas. So, if the user activates all effects on both channels, the Scheduler has to use Runtime Reconfiguration. Figure 8.11 shows the POL source code of the audio example. Please note, that in contrast to the Pong example, the *Dispatcher* is not needed for normal operation. The audio inputs and outputs are directly linked to the filter objects. The *Dispatcher* is solely used to instantiate and to destroy the filter objects and to establish the corresponding connections.

```
01 class high extends ParObj {
02  Slot:
03    int audio_in;
04  Signal:
05    int audio_out;
06  private:
07    int x, y=0;
08    int x1=0, x2=0;
09    boolean neu=true;
10
11  void calc() {
12    if (audio_in.valid())
13        x=audio_in.get(0);
14    else return;
15
16    if (neu==false) {
17      y = (x<<2) - (x1<<1) - (x2<<1);
18      x2 = x1;
19      x1 = x;
20    } else {
21      y = x; x2 = x; x1 = x;
22      neu=false;
23    }
24    audio_out.emit(y);
25  }
26 }
```

```
27 class low extends ParObj {
28  Slot:
29    int audio_in;
30  Signal:
31    int audio_out;
32  private:
33    int x, y=0;
34    boolean neu=true;
35
36  void calc() {
37    if (audio_in.valid())
38        x=audio_in.get(0);
39    else return;
40    if (neu==false)
41        y=((y)-(y>>7)+(x>>4));
42    else {
43        y=x; neu=false;
44    }
45    audio_out.emit(y);
46  }
47 }
```

```
50 class Dispatcher extends DispObj {
51   Slot:
52       int audio_in;
53       int keys;
54   private:
55       high h1,h2;
56       low  l1,l2;
57       int high_active=0;
58       int low_active=0;
59
60   Dispatcher() {
61       world.in0.connect(world.out0);
62       world.in1.connect(world.out1);
63       world.in2.connect(keys);
64   }
65
66   void calc() {
67           int t=keys.get();
68           if ((t==3)&(high_active==0)) {
69                     if (low_active==0) {
70                       world.in0.disconnect(world.out0);
71                       world.in1.disconnect(world.out1);
72                       h1 = new high();
73                       h2 = new high();
74                       h1.audio_out.connect(world.out0);
75                       h2.audio_out.connect(world.out1);
76                       world.in0.connect(h1.audio_in);
77                       world.in1.connect(h2.audio_in);
78                       high_active=1;
79                     } else {
80                       h1 = new high();
81                       h2 = new high();
82                       h1.audio_out.connect(world.out0);
83                       h2.audio_out.connect(world.out1);
84                       l1.audio_out.disconnect(world.out0);
85                       l2.audio_out.disconnect(world.out1);
86                       l1.audio_out.connect(h1.audio_in);
87                       l2.audio_out.connect(h2.audio_in);
88                       high_active=1;
89                     }
90
91           }
92

               ...

163    }
164   }
```

Figure 8.11: Source code of the audio filter example

[3] Audio Codec '97

Emulation

In order to provide an user interface that suits the audio filters, the input class *AudioIn* has been created. It is producing an input signal that represents a sum of two sinus waves — one with low and one with high frequency. Furthermore, the input class *AudioButtonIn* and the output class *AudioOut* have been implemented. *AudioButtonIn* provides a set of push buttons which can be used to activate and to deactivate a filter. *AudioOut* provides a graphical view of the output wave. Figure 8.12 shows a set of screenshots of the audio emulation.

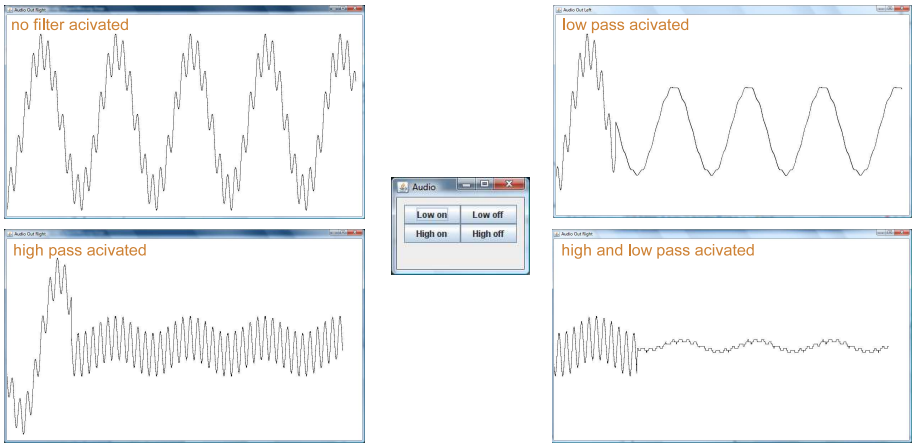


Figure 8.12: Emulation of the audio filters

After startup, the Emulator was using of 48 threads. The memory usage was between 25 Mb and 35 Mb. After activating the lowpass filter, the number of threads increased to 65. The memory usage was not influenced at all. With both filters active, the number of used threads was 82. This is the maximum number of started threads. Figure 8.13 illustrates the CPU and memory usage during emulation.

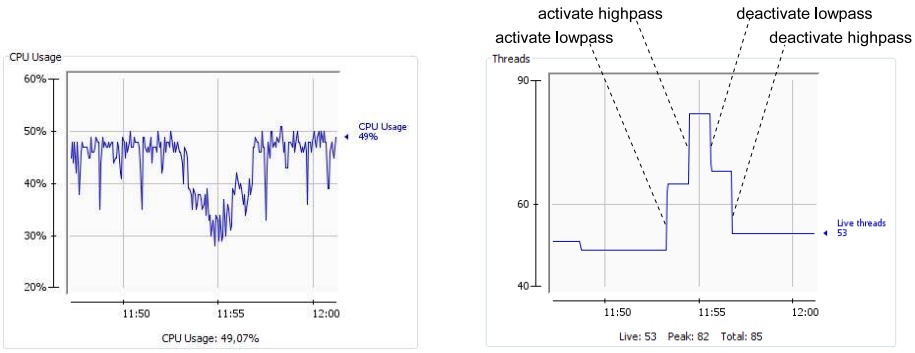


Figure 8.13: Overview of the CPU and memory usage in JConsole

High Level Synthesis

After verifying the functionality of the filters in the Emulator, the audio example has been translated to VHDL by the POL-Compiler. Table 8.4 shows the corresponding number of FSM states.

	Number of states before optimization	Number of states after optimization	Number of states used for context-management
high	79	59	40
low	71	55	36
Dispatcher	249	204	42

Table 8.4: Number of FSM states

Figure 8.14 shows the behavioral simulation of the lowpass filter. One can see that processing one audio sample takes 110 ns (11 states). The highpass filter also processes one audio sample in 11 states. This is made possible by the optimization step performed by the POL compiler which merges line 17, 18 and 19 into one single state.

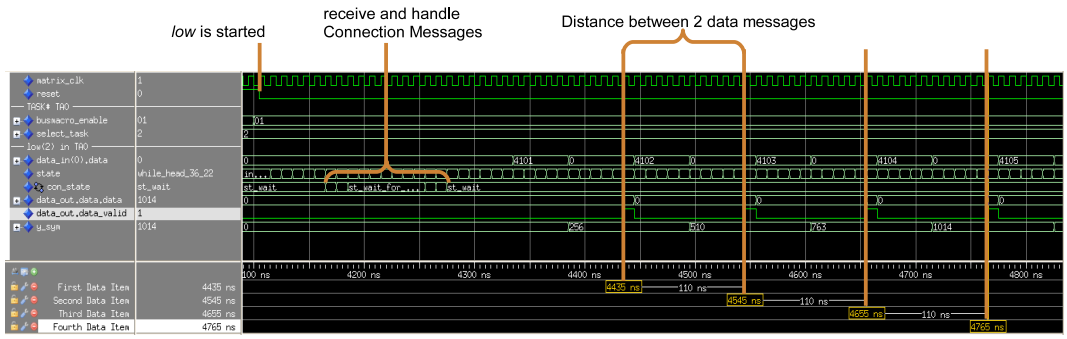


Figure 8.14: Behavioral simulation of the lowpass filter

Bitfile Generation

For the Pong example, all outputs and inputs were linked to the OPB. Such a design is not very useful regarding streaming applications. Thus, for the generation of the audio system, *System InOut* has been modified to transfer data which was send to *Slot* 0 directly to the right output channel of the AC97 and to transfer data which was send for *Slot* 1 directly to the left output channel of the AC97. All other data messages are forwarded to the OPB and are handled in software by the Scheduler.

The input channels of the AC97 were also linked directly to *System InOut*. For this, *System InOut* analyzes the Connection Messages send to *Slot* 15. If they concern *Signal* 0 or *Signal* 1 of *System InOut*, they are processed in hardware. Otherwise, they are forwarded

to the Scheduler and handled in software. To keep the hardware simple and fast, *Signal* 0 and *Signal* 1 of *System InOut* only support one single receiver.

System InOut contains 4 additional FIFOs: one for each of the input channels and one for each of the output channels. The input FIFOs collect the data samples coming from the AC97. Once 125 samples have been collected, those from the right input are sent to the Matrix. Then, those from the left input are transmitted. The output FIFOs collect the data coming from the Communication Matrix. They only send a new output sample to the AC97 if both (left and right output FIFO) contain data. Figure 8.15 illustrates the resulting chip design.

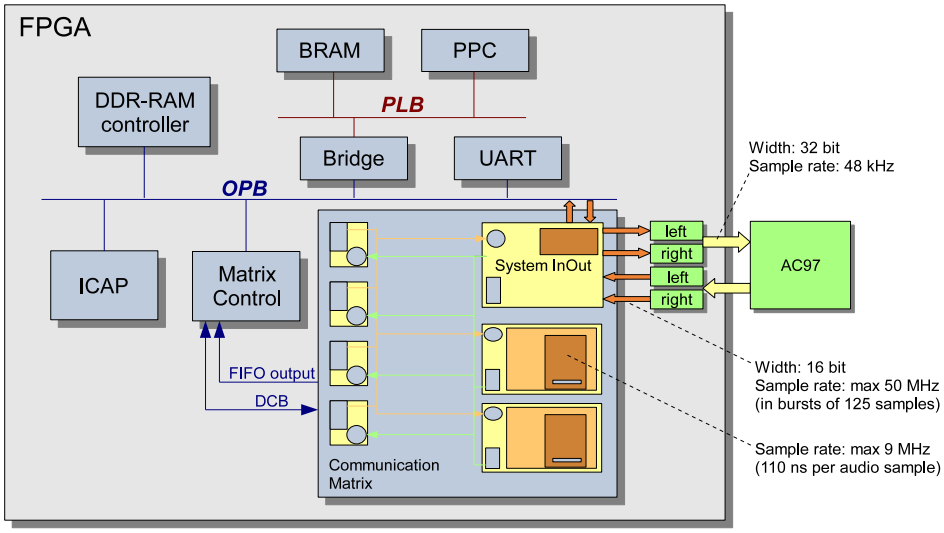


Figure 8.15: Chip design using the AC97-specific version of *System InOut*

The Xilinx ICAP controller used for the Pong example is very slow. Due to this the reconfiguration times are very long — too long for streaming applications. Hence, for the audio example a customized ICAP controller using DMA was utilized for reconfiguration (see chapter 7.6.3). The new ICAP controller contains 3 BRAMs (the Xilinx ICAP controller contained only one) which are used as ROM for reconfiguration command sequences.

During the implementation of the audio example, the first attempt to generate the bitfiles failed during place and route. The possibility to fall back to the graphical version of PlanAhead allowed a quick debugging. It turned out that due to the modification of *System InOut* and the new ICAP controller, the static design increased in size. Due to this, the floorplanning had to be re-done. The size of the dynamic areas had to be reduced in order to increase the size of the static area. Figure 8.16 illustrates the resulting partitioning of the FPGA for the audio example. One can see that the new floorplan has a large number of feed-through routes.

The bitfile generation of the audio example took 69 minutes. Table 8.5 shows an overview of the resource consumption.

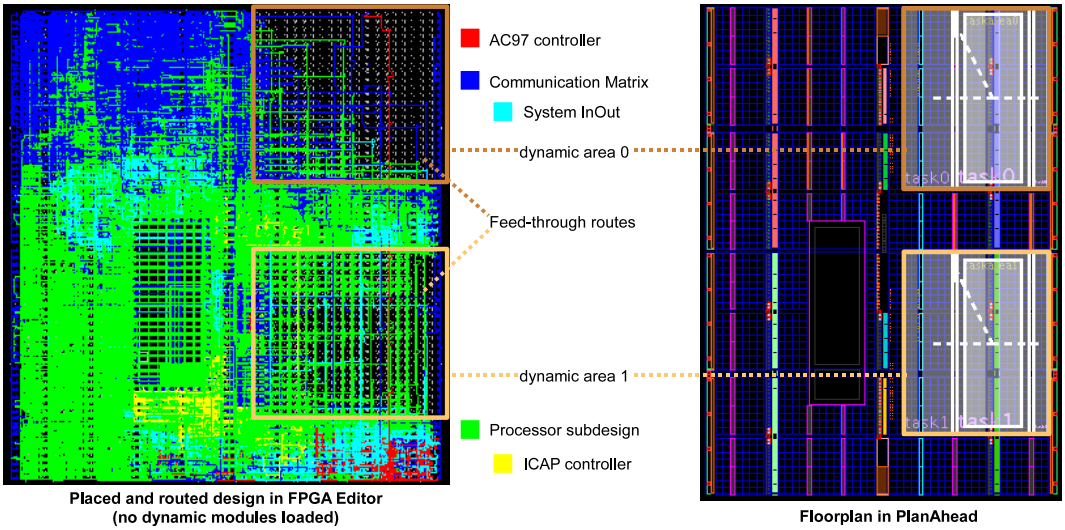


Figure 8.16: Partitioning of the Virtex-4 FX20

Component	#Slices	Part of the FPGA	#BRAMs
Virtex-4 FX20	8 544	100%	68
static area	5 472	64%	44
Communication Matrix	2 158	25%	11
Processor subsystem	3 037	36%	5
dynamic area 0	1 536	18%	12
dynamic area 1	1 536	18%	12
Dispatcher	1 088	13%	1
high	998	12%	1
low	966	11%	1

Table 8.5: Resource consumption

Hardware Execution

Table 8.6 illustrates the reconfiguration times coming with the usage of the customized ICAP controller compared to the reconfiguration times coming with the usage of the Xilinx ICAP controller. With the customized ICAP controller the reconfiguration speed of Short Reconfigurations has been increased by a factor of 6.

	Customized ICAP driver	Xilinx ICAP driver
Swap-Out:	83 μs	671 μs
Swap-In:	104 μs	518 μs
Short Reconfiguration:	215 μs	1 209 μs
Long Reconfiguration:	5 815 μs	19 146 μs

Table 8.6: Measured reconfiguration times

The times corresponding to the Xilinx controller are the same as in the Pong example. The time for the Short Reconfiguration is the sum of the times for swap-in, swap-out and DCB interaction. It only depends on the size of the Context Memory, not on the POL source code. The time for the Long Reconfiguration is determined by the number of used minor frames. Although the size of the dynamic areas has been reduced, the number of affected minor frames has not changed (since one minor frame utilizes one fourth of the FPGA's height and thus the change of the area size was smaller than the minor frame granularity).

If both filters are activated, the Scheduler loads the instances of *low* to Task Area 0 and the instances of *high* to Task Area 1. The *Dispatcher* is loaded on demand (when a filter is requested to be activated or deactivated) to Task Area 0. Figure 8.17 illustrates the minimum time for one scheduling cycle if both filters are activated and the *Dispatcher* is not called to change the filter setup. The time at which an instance is loaded to a dynamic area depends on the receiver address of the messages in the corresponding Class Buffer. Due to the specific design of *System InOut*, a sequence of 125 audio samples is sent to an instance of a filter class. The behavioral simulation showed that the filters need 110 ns to process one audio sample. Thus, they need about 14 μs to process 125 audio samples. A Short Reconfiguration needs 215 μs. Since the two dynamic areas can only be reconfigured successively, the minimum time an instance is loaded to a Task Area is determined by the reconfiguration times. Hence, the maximum sample rate of the runtime reconfigured system is 145 kHz.

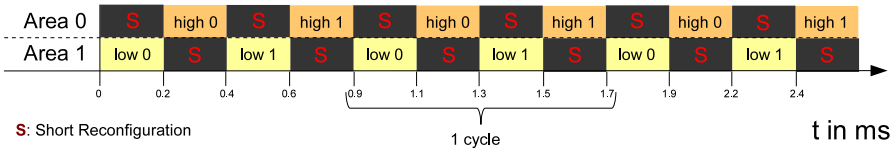


Figure 8.17: Minimum time for one scheduling cycle with both filters activated

The sampling rate of the AC97 is 48 kHz. In consequence, every 20.83 μs a new audio sample is stored in the input FIFOs. The input FIFOs store 125 audio samples before they forward them to the Communication Matrix. This takes 2.6 ms. The Communication Matrix processes the $2 \cdot 125$ audio samples in 0.86 ms. Next, it waits for new input data. Due to this, the real cycle time is 2 604 μs. Figure 8.18 illustrates the corresponding measured times.

In the performed tests, all calculations presented here have been verified. The audio stream (a song) was not interrupted, even when both filters were activated for both channels and thus Short Reconfigurations were performed permanently as shown in figure 8.18. In contrast, the activation as well as the deactivation of a filter and the resulting Long Reconfigurations caused an audible disruption of the audio stream.

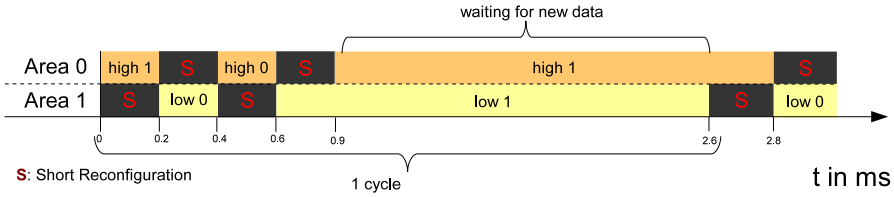


Figure 8.18: Real time for one scheduling cycle with both filters activated

8.2 Extrapolations

In the following section some extrapolations are shown to illustrate the possibilities of the Framework based on the usage of a faster ICAP controller and bigger FPGAs. As shown in figure 8.17, the minimum cycle times and the resulting maximum sample rate of a runtime reconfigured system are determined primarily by the reconfiguration times. The customized ICAP controller used for the audio example had a maximum throughput of 26 MB/s. This is caused by the relatively slow OPB-DDR-RAM controller coming with the EDK, which cannot provide a higher data rate. However, using the ICAP controller and the DDR-RAM controller presented in chapter 3.1.7, a maximum throughput of 400 MB/s can be achieved on Virtex-4. Such a throughput would lead to reconfiguration times of 13.12 μs for a Short Reconfiguration and 378.88 μs for a Long Reconfiguration. Figure 8.19 illustrates the corresponding minimum time for one scheduling cycle with both filters activated.

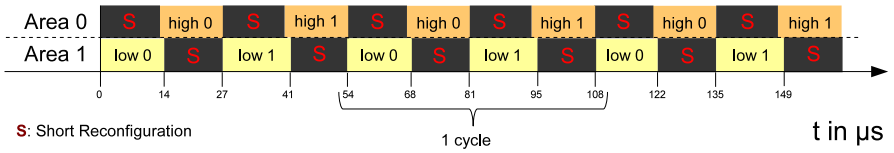


Figure 8.19: Minimum time for one scheduling cycle using an improved ICAP controller with 400 MB/s throughput

One can see that the minimum cycle time is determined by both the reconfiguration time (13 μs) and the filter processing time (14 μs). For two audio channels (left and right) this amounts to about 54 μs. Thus, using such a fast ICAP controller the maximum sample rate of the runtime reconfigured system would be 2 314 kHz. Based on these values, the audio example could support up to 96 audio channels with a sample rate of 48 kHz each ($2\,604\mu s / \frac{27\mu s}{channel} = 96\,channels$).

The Virtex-4 FX20 that has been used for the example implementations is one of the smallest FPGAs of the Virtex-4 series. Due to this, the static design consumed a big part of the FPGA. However, if a bigger FPGA is used, the number of Slices needed for the static part does not increase. That means that the size of the FPGA's part that can be used for dynamic areas increases linearly with the size of the whole FPGA. Figure 8.20 illustrates the Virtex-4 FX20 and the FX140 in proportion to each other. Furthermore, it shows the floorplan of the FX140 using 4 dynamic areas. The Communication Matrix consumes only 4% of the FPGA while each of the dynamic areas on the FX140 is bigger than the whole FX20.

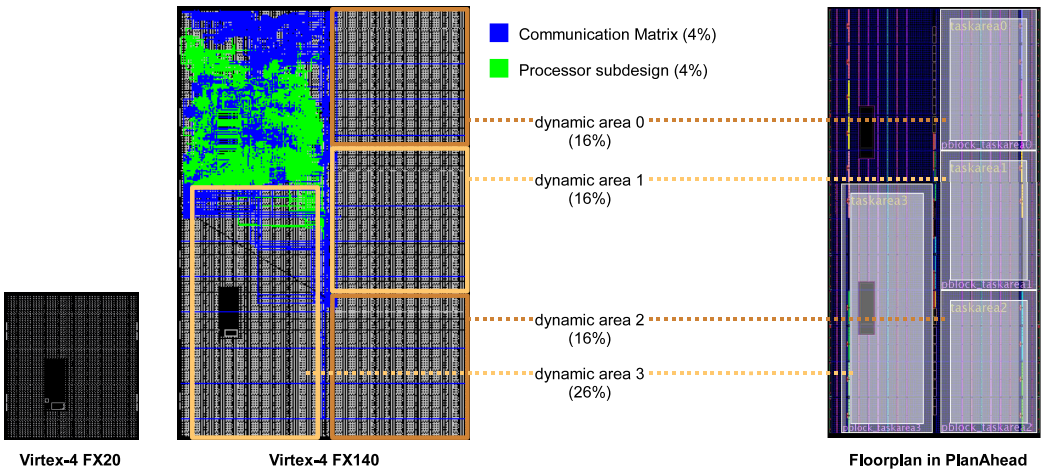


Figure 8.20: Floorplan of a Virtex-4 FX140 compared to the size of a Virtex-4 FX20

Figure 8.21 illustrates the cycle times resulting from the usage of 4 Task Areas and 4 audio filters (lowpass, highpass, echo and distortion). One can see that the reconfiguration times are determining the length of one scheduling cycle, even though the assumed time for one single Short Reconfiguration is based on the usage of the very fast ICAP controller providing 400 MB/s throughput. This is caused by the fact that the dynamic areas only can be reconfigured one after another. Due to this, each additional dynamic area that is part of the Runtime Scheduling increases the minimum cycle times and thus decreases the maximum sample rate of the system.

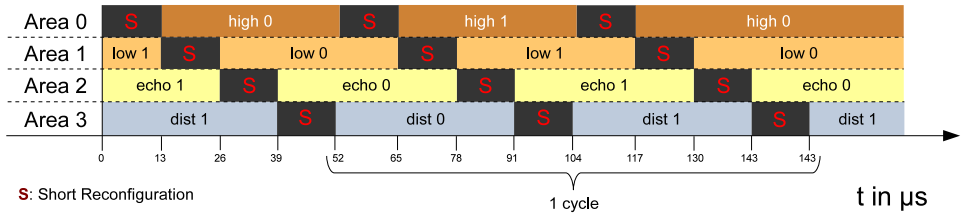


Figure 8.21: Minimum time for one scheduling cycle using an improved ICAP controller with 400 MB/s throughput and 4 different filter classes in 4 dynamic areas

Please note, that the time for a single Short Reconfiguration is not influenced by the size of the dynamic areas since it only depends on the size of the Context Memory. In contrast, the time for a Long Reconfiguration is proportional to the size of the dynamic areas. For an area with the size of 10 240 Slices (as shown in figure 8.20) the time for a Long Reconfiguration would be 2 526 μs (using the very fast ICAP controller).

8.3 Interpretation

Area

The Pong example has proven that all parts of the Framework are working correctly. The POL-compiler produces valid Java code for emulation as well as valid VHDL code for synthesis. The Emulator provides a platform for quick debugging on message layer where it is identical to the hardware execution. The Communication Matrix transports the messages without data loss, even if the system is reconfigured at runtime. The Scheduler is capable of Update Scheduling, Scenario Based Scheduling and Runtime Scheduling. The latter made it possible to overmap the FPGA (instantiating 10 balls on conventional hardware would lead to a FPGA utilization of 223%). Thus, the question if DPR can be used to increase the capacity utilization of hardware created via HLS can definitely be answered with yes.

Speed

The audio example shows that the Framework is able to handle streaming applications, even if Runtime Scheduling is being used. This has primarily been enabled by the usage of messages and the corresponding buffers for inter-module communication. Many of today's reconfigurable hardware accelerators make use of function calls (see chapter 3.2). That means that a piece of hardware is called like a function in procedural programming.

The problem of this approach is the potential absence of a called instance that leads to a delay of the function call until the instance has been loaded onto the FPGA. The Qt-like object-oriented and message-based inter-object communication helps to avoid these delays. Instance *A* is able to process a set of messages (e.g. the 125 audio samples) and to send it to instance *B* although instance *B* is currently not active. If and when instance *B* is loaded to the FPGA by the Scheduler does not influence instance *A* at all — which is very important for Runtime Reconfiguration.

The behavioral simulation has shown that the maximum sample rate of the filters is 1 sample per 110 ns. This is primarily caused by the functionality of the *Signals* which allow to connect more than one *Slot* to one single *Signal*. In order to improve the maximum sample rate, it would make sense to introduce a fourth access modifier called *Stream* which is used as a fast *Signal* and does only allow the connection of one single *Slot*. This way the needed clock cycles for *emit* could be reduced to 1.

However, nowadays these optimizations of the POL-Compiler are secondary, since the maximum throughput of a runtime reconfigured system is primarily determined by the reconfiguration times. These times have been reduced significantly by the introduction of the Short Reconfiguration. This method increases the reconfiguration speed by a factor of 54 (using dynamic areas with a size of 1 536 Slices) and even by a factor of 360 (using dynamic areas with a size of 10 240 Slices), since only the Context Memory has to be read back and/or configured. Furthermore, the corresponding Long Reconfigurations only have to configure the dynamic area. A read back of the whole dynamic area is not necessary at all. Hence, the measured maximum sample rate of 145 kHz highly depends on the usage of Short Reconfigurations. Without this technology the maximum sample rate would have been less than 4 kHz and thus Runtime Scheduling could not have been used in combination with the audio example.

The usage of Short Reconfiguration is enabled by the usage of object-orientation: changing from one instance of an object to another instance of the same object only requires a change of the object's state. The functionality stays the same. To put it in a nutshell, the object-oriented approach helped to significantly decrease the reconfiguration times.

Having said that, it is important to underline that the maximum throughput of the ICAP still is a very limiting factor. Even using Short Reconfiguration in combination with the fastest available ICAP controller cannot change the fact that it are the reconfiguration times which primarily limit the maximum sample rate of a runtime reconfigured system. Due to this, Runtime Reconfiguration is ready to be used in environments with a data rate of $\sim$100 000 samples/s (e.g. audio streaming) but it is not suitable for very high data rates, like 100 MB/s (e.g. video streaming). Nevertheless, the underlying scheduling technology can be used to automate the Scenario Based Scheduling. The measured values show that Update Scheduling and Scenario Based Scheduling are ready to be used in streaming applications with date rates of $\sim$100 000 000 samples/s (such as video-streaming or data-acquisition in high energy physics).

Productivity

As shown in the introduction of this thesis, the major motivation to use HLS is productivity. In many cases productivity is more important than efficiency. Thus, a very important question is:

- Does the Framework increase the productivity?

Before this question can be answered, it has to be said that productivity measurement is a very challenging task [179]. Productivity is commonly understood as the ratio of produced outputs to consumed resources. This definition has many degrees of freedom. For example, outputs can be measured in terms of delivered products or functionality. Resources can be measured in terms of effort or monetary costs. The major problem is that it is almost impossible to quantify "functionality" or "effort" through objective measurements. The IEEE standard 1045 defines the functionality as the number of lines of code or function points, but recommends variations (e.g. to address software re-use) [180]. Due to this, the decision was made, not to try to answer the productivity question in a quantitative but in a qualitative way.

Without using the Framework or a comparable development environment, the use of DPR leads to struggling with architectural details of the FPGAs and the corresponding synthesis and implementation tools. In consequence, the average time to introduce a student who had already worked with VHDL or Verilog to DPR was about 1 to 3 months. In contrast, POL allows to use DPR without going into detail with the FPGA architecture. A developer uses *new* just like in software development. He or she does not have to care about the underlying technology (e.g. whether DPR is used or not). The Framework completely encapsulates these implementation details. Performed tests show, that a student who has already worked with Java is able to make use of DPR within one day, if he or she is using POL.

However, these values cannot be used to determine an exact factor. For example the change from one Virtex-4 series (e.g. FX20) to another (e.g. FX40) requires a re-do of the floorplanning, which demands a deeper understanding of the underlying system and decreases the productivity. Furthermore a change of *System InOut* (as it has been performed for the audio example) requires VHDL or Verilog skills and also decreases the productivity.

In conclusion, it is not possible to denote an exact factor, but the question if the Framework can help to increase the productivity can definitely be answered: yes, it does.

9 Conclusions and Outlook

9.1 Conclusions

This study focuses on the combination of dynamic partial reconfiguration (DPR) and high level synthesis (HLS). As shown in chapter 3, both are very important future trends regarding hardware design and have already proven to be very useful. However, the combination of DPR and HLS is still in its infancy. Most of today's software-to-hardware compilers focus on conventional hardware and therefore have to remove dynamic aspects — while most of today's DPR tools work on the lowest possible layer regarding FPGAs: the bitfile level. There are hundreds of projects focusing on DPR and also hundreds of projects focusing on HLS, but only 3 projects (namely OSSS+R, JHDL and MORPHEUS) could be found which focus on a combination of both.

In chapter 4, the programming paradigms as well as the existing programming languages have been analyzed. It turned out that object-oriented programming (OOP) in combination with multithreading is a very good way to describe dynamic hardware. The reasons are the elegant way to express DPR in OOP via *new* and *finish* and the well-established way to express concurrency explicitly via objects derived from a common *Thread* class. Beyond that, the *Signals* and *Slots* as well as the methods *connect* and *disconnect* which are part of Qt, provide a very elegant way to establish a message-based inter-object communication. All these language constructs are well-known by software developers and do not depend on the introduction of cumbersome low-level statements like *par* or *delay* in Handel-C.

In order to force the developer to make use of multi-threading, encapsulation and *Signals* and *Slots*, an enriched subset of Java was introduced: POL (Parallel Object Language). It allows the developer to express coarse-grained concurrency explicitly via parallel running objects, while potential fine-grained concurrency (between single statements) is detected automatically by the POL-Compiler.

Chapter 5 analyzed the requirements coming with the specification of POL. For this, typical FPGA applications have been evaluated and it has been shown that flexible streaming algorithms are the application niche of FPGAs. Due to this, the hardware generated from the POL sources has to be flexible (what is obviously in line with the utilization of DPR) and it has to support streaming. Thus, the utilization of a central memory or a simple bus for inter-object communication was not an option. In consequence, a flexible but standardized network on chip was designed: the Communication Matrix. It introduces an additional abstraction layer: messages containing both payload and receiver address. These messages are stored in Class Buffers. Each Class Buffer collects all mes-

sages heading for the instances of one single class. Thus, the Communication Matrix has to instantiate as much Class Buffers, as POL classes are used. The whole messaging system perfectly fits the way *Signals*, *Slots*, *connect* and *disconnect* are working.

All the concepts presented in chapter 6 with the exception of the Instance Buffer have been implemented and are operational. The POL-Compiler takes POL source code and produces a Java file for emulation as well as VHDL files for synthesis. The Emulator makes it possible to perform early debugging on message layer. In fact, once the Framework was running and teething troubles had been removed, the usage of the behavioral simulation was not necessary any longer. A well running emulation was directly correlated to a well running hardware design. Especially deadlocks coming with the misuse of *connects* or *disconnects* were found and eliminated quickly in the Emulator.

In the FPGA, the dynamic modules produced by the POL-Compiler, the Communication Matrix and the Scheduler perfectly worked together. To demonstrate the Framework, chapter 8 showcases two example applications: Pong and an audio filter. The Pong example showed, that all parts of the Framework are working correctly. Furthermore it proved that it is possible to overmap the FPGA using Runtime Scheduling. The audio example was used to analyze the behavior of the Framework regarding data streams. It showed that Runtime Reconfiguration can be used in environments with a data rate of $\sim$100 000 samples/s, while Update Scheduling and Scenario Based Scheduling can be used in streaming applications with data rates of $\sim$100 000 000 samples/s. There is a multitude of applications that can benefit significantly from these scheduling strategies. Examples are video streaming and data-acquisition in high energy physics which have been illuminated in detail in chapter 5.2.

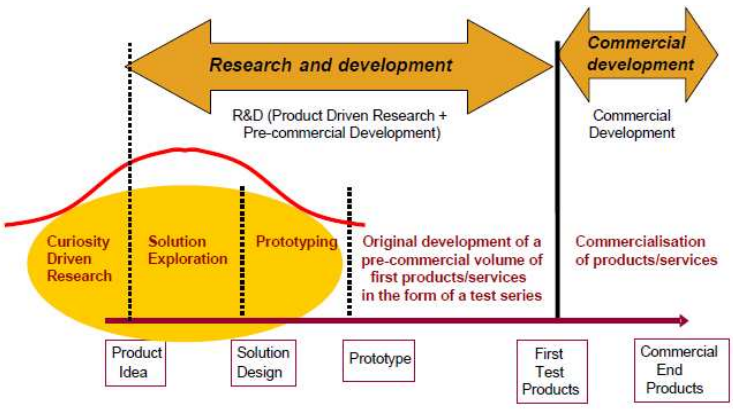


Figure 9.1: Typical innovation live cycle [181]

At this point it is important to emphasize, that the real implementation of the Framework is an essential feature of this thesis. The Framework is automatically producing hardware that makes use of Runtime Scheduling from a language operating on algorithmic level. It has not only been conceived or simulated but is actually running and the presented measured results are reproducible at any time. One resulting question is: is the Framework ready to be used in real world applications (e.g. as commercial product)?

Figure 9.1 illustrates the live cycle of an innovation. At the beginning there is academic curiosity which leads to a concrete idea. Next, the problems coming with the idea are analyzed and solutions are developed. This results in the creation of a so called Solution Design that represents a first running implementation. The corresponding measured results are used to improve the design and to implement a first prototype. At this step, the basic approach can still be modified (due to encountered possibilities or limitations). Next, the prototype is refined to a first test product. The experiences of users of this product are used to perform further refinement steps.

The Object-Oriented Framework for DPR which is object of this thesis represents a Solution Design. It is a first implementation that solves the problems coming with the combination of DPR and HLS. It has proven that it is possible to combine DPR and HLS and that this combination enables developers to write programs on algorithmic level which are actually executed in hardware and make use of DPR and all the advantages coming with it (e.g. more flexibility, update management and overmapping). The next step is to evaluate the measured and extrapolated results and to focus on the implementation of a prototype. In the following section some possible further developments are discussed.

9.2 Outlook

During design and implementation of the Framework the focus was on the development of the Communication Matrix, on the elegant expression of coarse grained granularity, and on the efficient expression and realization of dynamic hardware instantiation and the corresponding dynamic inter-object communication. Projects like MORPHEUS (see chapter 3.4.3) show that combining DPR with HLS is a huge topic which easily keeps a consortium consisting of 18 partners from several universities all over Europe busy for years. Therefore, during the design of the Framework, it was very important to stay focused and not to get lost in details or in secondary areas. So, one of the topics that were out of scope is the optimized translation from *calc()* to VHDL. There are many research groups which did or still do focus on the optimal translation of sequential algorithm descriptions to efficient hardware (see chapter 3.3). Thus, the decision was made to keep this part simple. The idea was to prove the elegance and power of POL regarding dynamic hardware instantiation and communication, and to combine this approach with an efficient C-to-VHDL or Java-to-VHDL compiler (handling sequential and purely static code) at a later point in time (e.g. as a follow-up thesis). Already established cooperations between the Kirchhoff Institute for Physics Heidelberg and industrial partners are very promising and show that the industry is highly interested into the here presented way to combine DPR and HLS.

Since the Framework represents a Solution Design, the basic approach has to be revisited for further development. At the moment the POL-Compiler solely creates VHDL for hardware execution. In chapter 3.2 several approaches combining software and hardware have been presented. It has been shown that the combination of a CPU with an FPGA that is loading hardware accelerators on demand is a very promising approach. Thus, a possible further development of POL could be the introduction of a fourth base class called *SoftObj*. Objects deriving from *SoftObj* would be translated to software which is running on an embedded processor that is part of the FPGA (just like the Scheduler) or that is placed on the same board as the FPGA. Furthermore, the introduction of a base class called *SoftParObj* could enable the POL-Compiler to translate a corresponding class to both software and hardware. This way, the Scheduler could decide at runtime if an instance shall be instantiated in hardware or in software.

Nowadays, the usage of Runtime Scheduling is limited to systems with a sample rate of $\sim$100 000 samples/s. If Xilinx or any other vendor improves the time for the reconfiguration process, Runtime Scheduling will instantly become very interesting for systems with a higher throughput (such as video streaming or data-acquisition) as well.

Bibliography

[1] K. Zuse, *Der Computer - Mein Lebenswerk*. Berlin, Germany: Springer, 1993.

[2] US Department of Commerce, *ENIAC - A Survey of Domestic Electronic Digital Computing Systems - Computers with names starting with E through H*, 1955.

[3] M. Irvine, *Early Digital Computers at Bell Telephone Laboratories*. IEEE Annals of the History of Computing, 2001.

[4] C. Brunelli *et al.*, *Reconfigurable hardware: The holy grail of matching performance with programming productivity*, FPL08, Heidelberg, Germany, Sep. 2008.

[5] *MAX II Product Backgrounder*, Altera inc.
URL http://www.altera.com/literature/pr/mx2_backgrounder.pdf, Nov. 2010.

[6] *Flash FPGAs in the Value-Based Market*, Actel inc.
URL http://www.actel.com/documents/ValueFPGA_WP.pdf, Nov. 2010.

[7] D. T. R. Walker, *A Model of Design Representation and Synthesis*, DAC85, Las Vegas, Nevada, USA, 1985.

[8] C. Kao, *Benefits of Partial Reconfiguration*, Xcell Journal, www.xilinx.com, 2005.

[9] C. Claus and J. Zeppenfeld, *Using Partial-Run-Time Reconfigurable Hardware to accelerate Video Processing in Driver Assistance System*, in *DATE07*, Nice, France, Mar. 2007.

[10] J. Becker, M. Huebner *et al.*, *Automotive Control Unit Optimisation Perspectives: Body Functions on-Demand by Dynamic Reconfiguration*, in *DATE07*, Munich, Germany, Mar. 2005.

[11] R. Sinnappan *et al.*, *A reconfigurable approach to packet filtering*, in *FPL01*, Belfast, UK, Aug. 2001.

[12] J. Henkel *et al.*, *A Computation- and Communication-Infrastructure for Modular Special Instructions in a Dynamically Reconfigurable Processor*, in *FPL08*, Heidelberg, Germany, Sep. 2008.

[13] S. Chau *et al.*, *In-system partial run-time reconfiguration for fault recovery applications on spacecrafts*, in *IEEE International Conference on Systems, Man, and Cybernetics*, Oct. 2005.

[14] U. K. N. Abel, *Increasing Design Changeability using Dynamic Partial Reconfiguration*, IEEE Journal, 2010.

[15] K. O. R. Helaihel, *Java as a specification language for hardware-software systems*, International Conference on Computer Aided Design, 1997.

[16] *SystemC Synthesizable Subset*, http://www.systemc.org, 2009.

[17] *Spartan and Spartan-XL FPGA Families Data Sheet*, Xilinx inc., www.xilinx.com, Jun. 2008.

[18] *Virtex-4 FPGA Configuration User Guide*, Xilinx inc., www.xilinx.com, Jun. 2009.

[19] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, Xilinx inc., www.xilinx.com, Nov. 2007.

[20] *Virtex-5 FPGA Configuration User Guide*, Xilinx inc., www.xilinx.com, Aug. 2009.

[21] *Virtex-6 FPGA Configuration User Guide*, Xilinx inc., www.xilinx.com, Nov. 2009.

[22] *Configuration and Readback of Spartan-II and Spartan-IIE FPGAs Using Boundary Scan*, Xilinx inc., www.xilinx.com, Jun. 2005.

[23] *Spartan-3 Generation Configuration User Guide*, Xilinx inc., www.xilinx.com, Oct. 2009.

[24] *Spartan-6 FPGA Configuration User Guide*, Xilinx inc., www.xilinx.com, Jun. 2009.

[25] W. S. L. B. M. H. J. B. C. Claus, B. Zhang, *A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput*, FPL08, Heidelberg, Germany, Sep. 2008.

[26] C. Claus, *AutoVision - Reconfigurable Hardware for video-based Driver Assistance Systems*, Erlangen, Germany, Sep. 2009.

[27] M. Berg, *A Comparison of Xilinx SRAM Based Configuration Scrubbing Methodologies*, Workshop on Fault-Injection and Fault-Tolerance tools for reprogrammable FPGAs, Nordwijk, Netherlands, Sep. 2009.

[28] H. Gustafsson *et al.*, *The ALICE experiment at the CERN LHC*, Institute of physics publishing and SISSA, Aug. 2008.

[29] P. Senger, *The Compressed Baryonic Matter Experiment*, GSI, Darmstadt, Germany, Jun. 2002.

[30] *Actel RTAX FPGAs*, Actel inc.
URL http://www.actel.com/products/milaero/rtaxs/default.aspx, Nov. 2010.

[31] *Xilinx Virtex-4QV FPGAs*, Xilinx inc.
URL http://www.xilinx.com/products/v4qv/index.htm, Nov. 2010.

[32] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. LaBel, M. Friendlich, H. Kim, and A. Phan, "Effectiveness of Internal vs. External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA: Design, Test, and Analysis ," *IEEE Transactions on Nuclear Science*, no. 4, pp. 2259–2266, Aug 2008.

[33] U. K. J. Gebelein, H. Engel, *An approach to system-wide fault tolerance for FPGAs*, FPL09, Prague, Czech Republic, Sep. 2009.

[34] *Single-Event Upset Mitigation for Xilinx FPGA Block Memories*, Xilinx inc., Mar. 2008.

[35] U. K. J. Gebelein, H. Engel, *Verbesserung der Strahlentoleranz von FPGAs für Experimente der Hochenergiephysik*, ZuE09, Stuttgart, Germany, 2009.

[36] *DS 280 - OPB HWICAP Product Specification*, Xilinx inc., www.xilinx.com, Mar. 2004.

[37] *Early Access Partial Reconfiguration User Guide*, Xilinx inc., www.xilinx.com, Mar. 2008.

[38] *Partial Reconfiguration Early Access Software Tools*,
URL http://www.xilinx.com/support/prealounge/protected/index.htm, Nov. 2010.

[39] H. Walder and M. Platzner, *Online Scheduling for Block-Partitioned Reconfigurable Devices*, DATE03, Munich, Germany, Mar. 2003.

[40] N. Abel, *Schnelle dynamische partielle Rekonfiguration in Hardware mit Inter-Task-Kommunikation*, Diploma thesis, Leipzig University, Germany, 2005.

[41] M. Huebner, *Dynamic and Partial Reconfigurable Hardware System Architecture with Real-time On-demand Functionality*, DATE07, Nice, France, Apr. 2007.

[42] C. Bobda, *Introduction to Reconfigurable Computing*. Springer, 2007.

[43] P. Lysaght and B. Blodget, *Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs*, FPL06, Madrid, Spain, Aug. 2006.

[44] N. A. W. Gao *et al.*, *DPR in High Energy Physics*, DATE09, Nice, France, Mar. 2009.

[45] M. Majer, J. Teich *et al.*, "The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer," *VLSI Signal Processing*, vol. 47, Apr. 2007.

[46] D. Morris, D. Evansa, and P. Green, "Object oriented computer system engineering," *Springer*, 1996.

[47] U. K. N. Abel, *Parallel Hardware Objects for Dynamically Partial Reconfiguration*, FPL08, Heidelberg, Germany, Sep. 2008.

[48] *Qt Homepage*, URL http://qt.nokia.com/, Nov. 2010.

[49] *Qt everywhere*, URL http://www.nokia.com/, Nov. 2010.

[50] N. W. M. Platzner, J. Teich, *Dynamically Reconfigurable Systems – Architectures, Design Methods and Applications*. Springer, 2010.

[51] *Department of Computer Science – Hardware-Software-Co-Design*, URL http://www12.informatik.uni-erlangen.de/, Nov. 2010.

[52] C. Bobda, *ESM: The Erlangen Slot Machine - Architecture and Development Tools*, DATE05, Munich, Germany, Mar. 2005.

[53] J. T. C. Bobda *et al.*, *The Erlangen Slot Machine (ESM): A Flexible Platform for Dynamic Reconfigurable Computing*, DATE05, Munich, Germany, Mar. 2005.

[54] ——, *The Erlangen Slot Machine: Increasing the Flexibility in FPGA-Based Reconfigurable Platforms*, FPT05, Singapore, Dec. 2005.

[55] ——, *The Erlangen Slot Machine: A Highly Flexible FPGA-Based Reconfigurable Platform*, FCCM05, Napa, CA, USA, Apr. 2005.

[56] J. T. M. Majer *et al.*, *The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-Based Computer*, Journal of VLSI Signal Processing Systems, Mar. 2007.

[57] J. A. D. K. S. Fekete, J. van der Veen and J. Teich, *No-Break Dynamic Defragmentation of Reconfigurable Devices*, FPL08, Heidelberg, Germany, Sep. 2008.

[58] H. K. D. K. A. Ahmadinia, C. Bobda and J. Teich, *FPGA Architecture Extensions for Preemptive Multitasking and Hardware Defragmentation*, FPT04, Brisbane, Australia, Dec. 2004.

[59] J. T. C. Bobda *et al.*, *A Flexible Reconfiguration Manager for the Erlangen Slot Machine*, DRS06, Frankfurt/Main, Germany, Mar. 2006.

[60] M. M. D. Göhringer and J. Teich, *Bridging the Gap between Relocation and Available Technology: The Erlangen Slot Machine*, Dagstuhl, Germany, 2006.

[61] J. T. C. Bobda *et al.*, *Erlangen Slot Machine: An FPGA-Based Dynamically Reconfigurable Computing Platform*, Dynamically Reconfigurable Systems - Architectures, Design Methods and Applications, Feb. 2010.

[62] J. T. J. Angermeier *et al.*, *The Erlangen Slot Machine - A Platform for Interdisciplinary Research in Reconfigurable Computing*, it - Information Technology, 2007.

[63] J. Grembler, *Dynamisch rekonfigurierbare Videomodule auf der Erlangen Slot Machine*, Diploma thesis, University Erlangen-Nuremberg, Germany, Nov. 2006.

[64] E. Sibirko, *Paralleles Sortieren auf der Erlangen Slot Machine*, Student Research Project, University Erlangen-Nuremberg, Germany, Dec. 2009.

[65] U. Batzer, *Hardware-Software-Co-Design von Echtzeitbilderkennungsalgorithmen für die Erlangen Slot Machine (ESM)*, Student Research Project, University Erlangen-Nuremberg, Germany, May 2008.

[66] *ReCoBus*, URL http://www.recobus.de/, Nov. 2010.

[67] C. B. D. Koch and J. Teich, *Efficient Reconfigurable On-Chip Buses for FPGAs*, FCCM08, Palo Alto, California, USA, Apr. 2008.

[68] D. Koch and J. Teich, *Platform-Independent Methodology for Partial Reconfiguration*, CF04, Ischia, Italy, Apr. 2004.

[69] C. B. D. Koch and J. Teich, *ReCoBus-Builder - a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs*, FPL08, Heidelberg, Germany, Sep. 2008.

[70] C. H. D. Koch, T. Streichert and J. Teich, *Efficient Reconfigurable On-Chip Buses*, European Patent EP07017975, Sep. 2007.

[71] ——, *Logic Chip, Logic System and Method for Designing a Logic Chip*, Patent PCT/EP2008/007342, Sep. 2008.

[72] *Institut für Technik der Informationsverarbeitung*, URL http://www.itiv.kit.edu/, Nov. 2010.

[73] M. K. J. B. M. Huebner, C. Schuck, *New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits*, IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006.

[74] J. B. M. Huebner, C. Schuck, *Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs*, 20th International Parallel and Distributed Processing Symposium, 2006.

[75] M. H. J. B. C. Schuck, M. Kuehnle, *A framework for dynamic 2D placement on FPGAs*, IEEE International Symposium on Parallel and Distributed Processing, Miami, USA, 2008.

[76] A. Thomas, *Design of a Dynamic Reconfigurable Multi-Grained Hardware Architecture with Adaptive Runtime Routing*, FPL05, Tampere, Finland, 2005.

[77] J. B. A. Thomas, *Multi-grained Reconfigurable Harware Architecture with Online-Adaptive Routing Techniques*, IFIP VLSI-SOC 2005, Perth, Australia, 2005.

[78] ——, *Dynamic Adaptive Routing Techniques In Multigrain Dynamic Reconfigurable Hardware Architectures*, FPL04, Leuven, Belgium, 2004.

[79] ——, *Multi-grained Reconfigurable Datapath Structures for Online-Adaptive Reconfigurable Hardware Architectures*, ISVLSI 2005, Tampa, Florida, USA, 2005.

[80] J. B. C. C. W. S. M. Huebner, L. Braun, *Physical Configuration On-Line Visualization of Xilinx Virtex-II FPGAs*, IEEE Computer Society Annual Symposium on VLSI, 2007.

[81] *Development System Reference Guide*, Xilinx inc., www.xilinx.com, 2005.

[82] J. B. M. Huebner, T. Becker, *Real-time LUT-based Network Topologies for dynamic and partial FPGA Self-Reconfiguration*, SBCCI04, Porto de Galinhas, Brasil, 2004.

[83] J. B. M. Huebner, *Tutorial on Macro Design for Dynamic and Partially Reconfigurable Systems*, RC-Education, Karlsruhe, Germany, 2006.

[84] J. B. A. Donlin, M. Huebner, *Models and Tools for the Dynamic Reconfiguration of FPGAs*, In IEEE-SOCC05, Washington, USA, 2005.

[85] M. H. C. S. J. B. W. S. C. Claus, B. Zhang, *An XDL-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems*, ISVLSI07, Porto Alegre, Brazil, May 2007.

[86] *Institute for Integrated Systems*,
URL http://www.lis.ei.tum.de/, Nov. 2010.

[87] W. Stechele, *Entwicklung von VLSI Architekturen für die Videosignalverarbeitung in Multimedia-Endgeräten*, Habilitation Thesis, TU Munich, 2000.

[88] ——, *Algorithmic Complexity, Motion Estimation and a VLSI Architecture for MPEG-4 Core Profile Video Codecs*, VLSI-TSA, Taiwan, Apr. 2001.

[89] ——, *VLSI architecture for MPEG-4 core profile video codec with accelerated bitstream processing*, VLSI Circuits and Systems conference, Maspalomas, Spain, May 2003.

[90] A. H. W. Stechele, S. Herrmann, *Towards a Dynamically Reconfigurable System-on-Chip Platform for Video Signal Processing*, ARCS04, Augsburg, Germany, Mar. 2004.

[91] S. H. J. L. S. W. Stechele, L. Alvado Carcel, *A Coprocessor for Accelerating Visual Information Processing*, DATE05, Munich, Germany, Mar. 2005.

[92] S. H. W. Stechele, *Reconfigurable Hardware Acceleration for Video-based Driver Assistance*, Workshop on Hardware for Visual Computing, Tübingen, Germany, Apr. 2005.

[93] W. S. A. Herkersdorf, *AutoVision - Flexible Processor Architecture for Video-assisted Driving*, DATE06, Munich, Germany, Mar. 2006.

[94] W. S. C. Claus, H. C. Shin, *Tunnel Entrance Recognition for video-based Driver Assistance Systems*, International Conference on Systems, Signals and Image Processing, Budapest, Hungary, Sep. 2006.

[95] W. Stechele, *AutoVision: A Run-time Reconfigurable MPSoC Architecture for Future Driver Assistance*, ECSI Workshop on Reconfigurable Systems-on-Chip, Paris, France, Jan. 2007.

[96] W. S. N. Alt, C. Claus, *Hardware/software architecture of an algorithm for vision-based real-time vehicle detection in dark environments*, DATE08, Munich, Germany, Mar. 2008.

[97] *Taillight Detection Video*,
URL http://www.lis.ei.tum.de/uploads/media/VehicleLights-Video.wmv, Nov. 2010.

[98] *Optical Flow Video*,
URL http://www.lis.ei.tum.de/uploads/media/31x31_01.wmv, Nov. 2010.

[99] W. S. C. Claus, J. Zeppenfeld, *Using Partial-Run-Time Reconfigurable Hardware to accelerate Video Processing in Driver Assistance Systems*, DATE07, Nice, France, Apr. 2007.

[100] S. B. J. B. K. Paulsson, M. Huebner, *Exploitation of Run-TIme Partial Reconfiguration for Dynamic Power Management in Xilinx Spartan III-based Systems*, ReCoSoc07 Montpellier, France, 2007.

[101] *JTAG Programmer Guide*, Xilinx inc., www.xilinx.com, 2000.

[102] G. A. M. D. L. C. J. B. K. Paulsson, M. Huebner, *Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self-Reconfiguration on Xilinx Spartan III FPGAs*, FPL07, Amsterdam, Netherlands, 2007.

[103] M. H. J. B. K. Paulsson, U. Viereck, *Exploitation of the External JTAG Interface for Internally Controlled Configuration Readback and Self-Reconfiguration of Spartan 3 FPGAs*, VLSI08, 2008.

[104] C. B. D. Koch and J. Teich, *Bitstream Decompression for High Speed FPGA Configuration from Slow Memories*, ICFPT07, Kokurakita, Kitakyushu, Japan, Dec. 2007.

[105] F. H. M. W. S. C. Claus, J. Zeppenfeld, *A new framework to accelerate VirtexII Pro dynamic partial self-reconguration*, Reconfigurable Architectures Workshop, Long Beach, USA, Mar. 2007.

[106] N. W. T. Vogt, *A Reconfigurable Application Specific Instruction Set Processor for Convolutional and Turbo Decoding in a SDR Environment*, DATE08, Munich, Germany, Mar. 2008.

[107] L. Bauer, M. Shafique, and J. Henkel, *Cross-architectural design space exploration tool for reconfigurable processors*, DATE09, Nice, France, 2009.

[108] S. Sawitzki, A. Gratz, R. G. Spallek, and R. Rsrc, *CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism*, 1998.

[109] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, *CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit*, in *In Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM Press, 2000, pp. 225–235.

[110] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, *The MOLEN Processor Prototype*, in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 296–299.

[111] G. Amdahl, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS67, 1967.

[112] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, *RISPP: rotating instruction set processing platform*, DATE07, New York, NY, USA, 2007.

[113] J. H. L. Bauer, M. Shafique, *A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor*, FPL08, Heidelberg, Germany, Sep. 2008.

[114] R. Lysecky, *Low-Power Warp Processor for Power Efficient High-Performance Embedded Systems*, DATE07, Nice, France, Apr. 2007.

[115] R. Lysecky, G. Stitt, and F. Vahid, *Warp Processors*, in *Proceedings of the 41st annual Design Automation Conference*. New York, NY, USA: ACM, 2004, pp. 659–681.

[116] ——, "Warp Processors," *ACM Trans. Des. Autom. Electron. Syst.*, pp. 659–681, Jun. 2004.

[117] A. Putnam *et al.*, *CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures*, FPL08, Heidelberg, Germany, sep 2008.

[118] *Handel-C Language Reference Manual*, Agility Design Solutions Inc., 2010.

[119] S. A. Edwards, *The Challenges of Hardware Synthesis from C-like Languages*, DATE05, Munich, Germany, Mar. 2005.

[120] S. Kurra, N. Singh, and P. Panda, *The Impact of Loop Unrolling on Controller Delay in High Level Synthesis*, DATE07, Nice, France, Apr. 2007.

[121] K. B. O. Dragomir, T. Stefanov, *Loop Unrolling and Shifting for Reconfigurable Architectures*, DATE08, Munich, Germany, Mar. 2008.

[122] S. P. Y. Chong, *Automatic Apllication Specific Floating-Point Unit Generation*, DATE07, Nice, France, Apr. 2007.

[123] T. K. J. Ulm, J. Kim, *Layout-driven resource sharing in high-level synthesis*, ICCAD02, San Jose, CA, USA, 2002.

[124] M. S. P. Brisk, A. Kaplan, *Area-efficient instruction set synthesis for reconfigurable systems-on-chip designs*, DAC04, San Diego, CA, USA, 2004.

[125] T. Grötker *et al.*, *System Design with SystemC*. Kluwer, 2002.

[126] G. M. D. Ku, *HardwareC: A Language for Hardware Design*. Stanford University, USA, 1990.

[127] D. Gajski *et al.*, *SpecC: Specification Language and Methodology*. Kluwer, 2000.

[128] H. G. G. Ferizis, *Mapping Recursive Functions to Reconfigurable Hardware*, FPL06, Madrid, Spain, Aug. 2006.

[129] Y. P. D. Sonderman, *Implementing C algorithms in reconfigurable hardware using C2Verilog*, FCCM98, Napa, CA, USA, 1998.

[130] K. V. Zhi Guo *et al.*, *Optimized Generation of Data-Path from C Codes for FPGAs*, DATE05, Munich, Germany, Mar. 2005.

[131] *IEEE Standard SystemC Language Reference Manual*, http://www.systemc.org, 2006.

[132] *Celoxica*, URL http://www.celoxica.com, Nov. 2010.

[133] *Catapult-C*, URL http://www.mentor.com/catapult, Nov. 2010.

[134] D. Galloway, *The Transmogrifer C hardware description language and compiler for FPGAs*, FCCM95, Napa, CA, USA, 1995.

[135] J. Frigo *et al.*, *Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective*, FPGA01, Monterey, CA, USA, 2001.

[136] *DIMEtalk 3.1 User Guide NT*, Nallatech, 2006.

[137] D. B. S. Huang, A. Hormati and R. Rabbah, *Liquid metal: Object-oriented programming across the hardware/software boundary*, ECOOP 2008, Paphos, Cyprus, Jul. 2008.

[138] B. Niemann, *C-Based Design – Myth or Reality?*, DATE09, Nice, France, Mar. 2009.

[139] P. Bellows and B. Hutchings, *JHDL - An HDL for Reconfigurable Systems*, FCCM98, Napa, CA, USA, 1998.

[140] *JHDL Homepage*, URL http://www.jhdl.org, Nov. 2010.

[141] *BYU EDIF Tools Homepage*, URL http://reliability.ee.byu.edu/edif/, Nov. 2010.

[142] P. Hartmann *et al.*, *Languages for Embedded Systems and their Applications*. Springer, 2009.

[143] *ANDRES Homepage*, URL http://andres.offis.de/, Nov. 2010.

[144] *OSSS+R Getting Started*,
URL andres.offis.de/infomaterial/material/osssr_getting_started.pdf, Nov. 2010.

[145] *Fossy synthesizer*, URL http://www.system-synthesis.org/, Nov. 2010.

[146] A. Schallenberg *et al.*, "PolyDyn - Object-Oriented Modelling and Synthesis Targeting Dynamically Reconfigurable FPGAs," in *Dynamically Reconfigurable Systems*.

[147] *MORPHEUS Homepage*, URL http://www.morpheus-ist.org/, Nov. 2010.

[148] *PACT XPP Homepage*, URL http://www.pactxpp.com, Nov. 2010.

[149] *ARCES Homepage*,
URL http://www.arces.unibo.it/content/view/31/290/, Nov. 2010.

[150] G. Pulini and D. Hulance, "Flexeos Embedded FPGA Solution," in *Dynamic System Reconfiguration in Heterogeneous Platforms*.

[151] M. H. N. Voros, A. Rosti, *Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach*. Springer, 2009.

[152] P. Bonnot *et al.*, *MORPHEUS project overview*, DATE08, Munich, Germany, Mar. 2008.

[153] ——, *Exploitation of reconfiguration for increased run-time flexibility and self-adaptive capabilities in future SOCs*, DATE10, Dresden, Germany, Mar. 2010.

[154] P. Alfke, *20 Years of FPGA Evolution – from Glue Logic to Major System Component*, HOT CHIPS 19, Aug. 2007.

[155] *Unlock New Levels of Productivity for Your Design Using ISE Design Suite 12*, Xilinx inc., May 2010.

[156] *Introducing Innovations at 28 nm to Move Beyond Moore's Law*, Altera, Apr. 2010.

[157] M. Santarini, *Xilinx Architects ARM-Based Processor-First, Processor-Centric Device*, Xcell Journal, Apr. 2010.

[158] S. Kilts, *Advanced FPGA Design – Architecture, Implementation, and Optimization*. Wiley-Interscience, Hoboken, USA: Wiley, 2007.

[159] G. Smit, *A mathematical approach towards hardware design*, Dagstuhl Seminar, Germany, Jul. 2010.

[160] S. Manz, *Development and Implementation of an MotionJPEG Capable JPEG Decoder in Hardware*, Diploma thesis, Heidelberg University, Germany, 2008.

[161] *Silicon Software*, URL http://www.silicon-software.de/, Nov. 2010.

[162] C. K. T. Armbruster, P. Fischer, *Front-end electronics for CBM*, GSI, Darmstadt, Germany, 2009.

[163] T. Anticic *et al.*, *Commissioning of the ALICE data acquisition system*, Victoria, Canada, 2007.

[164] W. Mueller *et al.*, *DAQ and Online Event Selection for CBM*, GSI, Darmstadt, Germany, 2008.

[165] *CERN*, URL http://www.cern.ch/, Nov. 2010.

[166] *nXYTER Homepage*,
URL http://cbm-wiki.gsi.de/cgi-bin/view/Public/PublicNxyter, Nov. 2010.

[167] F. L. N. Abel, W. Gao, *Design and implementation of a hierarchical DAQ network*, DPG conference, Darmstadt, Germany, 2008.

[168] U. K. N. Abel, W. Gao, *DPR in CBM: an Application for High Energy Physics*, DATE09, Nice, France, 2009.

[169] N. Meier, *Development of a Framework for Dynamic Partial Reconfiguration serving the Object Oriented Hardware Programming Language POL*, Diploma thesis, Heidelberg University, Germany, 2009.

[170] F. Gruell, *The Parallel Object Language - Development and Implementation of an Object Oriented Language for Partial Dynamic Reconfiguration on FPGAs*, Diploma thesis, Heidelberg University, Germany, 2009.

[171] *Information technology - sytanctic metalanguages - extended BNF*, ISO/IEC 14977, Oct. 1996.

[172] A. Beyer, *Development of an Emulator for the execution of POL-Code inside a Java-Environment*, Bachelor thesis, Heidelberg University, Germany, 2008.

[173] J. Gebelein, *System-Specification of Embedded Systems in Java for Synthesis*, Diploma thesis, Leipzig University, Germany, 2007.

[174] *ML40x Getting Started Tutorial*, Xilinx inc., Jun. 2006.

[175] *Spirit*, URL http://boost-spirit.com/, Nov. 2010.

[176] *Boost*, URL http://www.boost.org/, Nov. 2010.

[177] *The Pong Museum*, URL http://www.pongmuseum.com/, Nov. 2010.

[178] *Sun Java Homepage*, URL http://java.sun.com, Nov. 2010.

[179] D. Card, *The Challenge of Productivity Measurement*, Pacific Northwest Software Quality Conference, Portland, USA, Oct. 2006.

[180] *IEEE Standard for Software Productivity Metrics*, IEEE Std. 1045.

[181] P. Tsarchopoulos, *European Research Prospects for Reconfigurable Computing Systems*, DATE10, Dresden, Germany, Mar. 2010.

Erklärung zur selbständigen Verfassung

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

Bei der konkreten Implementierung des Frameworks wurde ich durch Diplomanden und einen Bachelor unterstützt. Die Implementierung des POL-Compilers wurde von Frederik Grüll im Rahmen einer Diplomarbeit durchgeführt. Der Emulator wurde von Andreas Beyer im Rahmen einer Bachelor-Arbeit implementiert. Eine Vorversion der Communication Matrix wurde von Johannes Nick Meier im Rahmen einer Diplomarbeit implementiert. All diese Arbeiten wurden von Prof. Dr. Udo Kebschull und mir betreut.

Die Studien des aktuellen Forschungsstandes, die Erarbeitung des konkreten Ansatzes, das Design des Frameworks, die initialen Implementierungen der Komponenten sowie die endgültige zusammenfassende Implementierung wurden von mir selbst realisiert. Die im Kapitel "Results" präsentierten Ergebnisse und Berechnungen wurden von mir selbst erhoben bzw. gemessen.

Heidelberg, im November 2010

Norbert Abel