TU Delft

BSc THESIS

Interconnect estimation from C-code

Michel Vos (1267825), Rick van Akkeren (1157493) and Silvian Bensdorp

EE-BS-2008-01

Abstract

FPGAs are easy and cheap to produce, a world of new possibilities is opened. One of those is in the area of reconfigurable computing. It is possible to extend normal CPUs with FPGAs for specific tasks, especially for those tasks which requires a lot of computational power. The Delft WorkBench is such a project. In this project, C-code is directly rewritten into a new piece of software and a set of hardware descriptions, suitable to program on a FPGA. In the rewritten part of the software, the computational parts are replaced by simple instructions to control the FPGA. The FPGA will run in parallel with the software and in this way, software can work up to 100 times faster. This thesis focus on the estimation of the required area of interconnect on a FPGA, depending on a given set of software metrics. These metrics are found by a special compiler, based on ELSA, and are specific for each part of C-code. With this estimation, it is possible to say, in an early stage of the whole process, if a certain part of software will fit on the FPGA. The developed model is based on a dataset from 127 kernels and is suitable for the Virtex2 and the Virtex4 platforms.

Faculty of Electrical Engineering, Mathematics and Computer Science

Interconnect estimation from C-code

THESIS

submitted in partial fulfillment of the
requirements for the degree of

BACHELOR OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Michel Vos (1267825), Rick van Akkeren (1157493) and Silvian Bensdorp

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Interconnect estimation from C-code

by Michel Vos (1267825), Rick van Akkeren (1157493) and Silvian Bensdorp

Abstract

Since FPGAs are easy and cheap to produce, a world of new possibilities is opened. One of those is in the area of reconfigurable computing. It is possible to extend normal CPUs with FPGAs for specific tasks, especially for those tasks which requires a lot of computational power. The Delft WorkBench is such a project. In this project, C-code is directly rewritten into a new piece of software and a set of hardware descriptions, suitable to program on a FPGA. In the rewritten part of the software, the computational parts are replaced by simple instructions to control the FPGA. The FPGA will run in parallel with the software and in this way, software can work up to 100 times faster. This thesis focus on the estimation of the required area of interconnect on a FPGA, depending on a given set of software metrics. These metrics are found by a special compiler, based on ELSA, and are specific for each part of C-code. With this estimation, it is possible to say, in an early stage of the whole process, if a certain part of software will fit on the FPGA. The developed model is based on a dataset from 127 kernels and is suitable for the Virtex2 and the Virtex4 platforms.

Bachelor in : Electrical Engineering
Codenumber : EE-BS-2008-01

Committee Members :

Advisor:	R.J. Meeuws M.Sc., CE, TU Delft
Chairperson:	Dr. K.L.M. Bertels, CE, TU Delft
Member:	Dr. ir. A.J. van Genderen, CE, TU Delft
Member:	L. Mhamdi, CE, TU Delft

Contents

List of Figures

List of Tables

Listings

Acknowledgements

This report is written as part of the course 'Bachelor afstudeerproject', ET3905. With this report we hope to fulfill part of our obligations in attaining the degree of Bachelor of Science in Electrical Engineering. After this we hope to proceed with the Master phase.

Readers who are interested or not familiar with the background of this project are referred to Chapter 2 of this report. The readers who are already familiar with the background and are interested in more specific information on the project can go directly to Chapter 3 and 4. Results can be found in Chapter 5 and conclusions and recommendations are given in Chapter 6.

We would like to thank the following people: Koen Bertels (CE, TU Delft), for guiding us in the beginning of the project, Yi Lu (CE, TU Delft), for helping us with the Xilinx ISE, Yana Yankova (CE, TU Delft), for providing us with the VHDL code of the software kernels and all the other people who did something for us, but kept unnoticed

Our special thanks goes out to Roel Meeuws (CE, TU Delft) for supervising us during the whole project. He gave us great support and advice. Also he read the concept versions of our report, providing us with useful comments on how to improve certain things.

Michel Vos (1267825), Rick van Akkeren (1157493) and Silvian Bensdorp
Delft, The Netherlands
July 3, 2008

1 Introduction

Since the beginning of the computer era many different types of computer systems have been developed. Early computer systems were based on the fixed program architecture. These systems could only do simple mathematics, but not such things like word processing or running video games. To change a program the whole architecture had to be re-wired, re-structured or even completely redesigned, often a laborious task.

The idea of a stored program architecture, also called the Von-Neumann architecture [1],[2], changed that. This architecture was divided into several units. A processing unit, a combined data and program memory, and data and control elements between the processing unit and the memory. By creating an instruction set for the processing unit it could fetch data from the memory, making a calculation and storing it back into the memory. Executing instructions sequentially allows a next instruction to use the result of a previous instruction in its calculation. This made the implementation of algorithms much simpler and therefore the variety of programs for this architecture much larger.

However this idea also showed a bottleneck, known as the Von-Neumann bottleneck[2]. Division between the processing unit and the memory, caused that instructions and data continually had to be moved between them, also called the throughput. Because the processing unit could work at a much faster speed than the rate of throughput, the processing unit had to wait continuously for data to be fetched from memory, what lead to longer execution times. Every increase in the amount of data transferred would increase the execution time. To reduce this performance problem, caches were placed between the CPU and the main memory. Despite this bottleneck this architecture is still the dominating architecture in conventional computing, because it offers a lot of flexibility.

Although the computer systems based on the Von-Neumann architecture are dominating conventional computing, other systems are used in specific areas. Application Specific Integrated Circuits (ASICs) for example. ASICs are designed to execute specific applications in hard-wired technology. Though this is a very fast technology for executing applications, it has the same lack of flexibility as the fixed program architecture.

In recent years the amount of data processed in computer systems has increased rapidly. For computer machines based on the Von-Neumann architecture this is becoming a problem. The earlier mentioned bottleneck causes these machines to be insufficient due the increasing execution time by large amounts of data. Therefore, designers increasingly use ASICs to execute applications with large amount of data. Though this is a good solution for speeding up the execution time, the lack of flexibility still remains.

To solve this problem researchers came up with a technology called reconfigurable computing (RC). It combines the flexibility of the Von-Neumann architecture with the speed of ASICs. RC systems consist of software programmable processing units and programmable hardware components, like a Field Programmable Gate Arrays (FPGAs).

These systems provide new possibilities to speed up compute-intensive calculations multiple times. However, the design and applications for RC systems demand a new approach for hardware and software design. This new approach requires knowledge of hardware and software. Current application designers for conventional computing do not have this knowledge. To prevent application designers from ignoring this new technology, because it does not integrate with their systems, a comprehensive high-level development platform is required. This platform should give the application designer an easy way to implement their applications on a reconfigurable computing system.

The DelftWorkBench (DWB) tool chain, in development at the Delft University [3], is such a software design platform. It provides the application designer with an easy way to implement their application. Going through several phases in the tool chain the application will be implemented on the reconfigurable platform. These phases are: Code profiling and cost modeling, code transformations and optimizations, retargetable compiler, retargetable processor and (exchange) registers; all of which are further discussed in section 2.5.

1.1 Problem description

This report will focus on the first phase of the DWB: *Code profiling and cost modeling*. As part of the Hardware/Software Partitioning process this phase is an important step in the development of reconfigurable systems. Hardware/Software Partitioning is nothing more then identifying those parts of an application that should be executed in hardware and those parts that should be executed in software. The goal of Hardware/Software Partitioning is to divide an application (the hardware part and the software part) in such a way that it can offer the optimal performance when implemented on a reconfigurable system. To achieve this goal, parts of the application have to be analyzed and profiled on certain aspects, like area metrics and time delay. To get the best results it is important to know in an early stage some of these aspects, so the process of partitioning becomes easier and better. Therefore one possibility would be to build up a statistical estimation model, that can predict these aspects in an early stage. In addition to this statistical estimation model, a tool should be written that is able to make predictions with the prediction model and earlier determined Software Complexity Metrics (SCMs), see [4] for more details on SCMs.

The original idea of this project was to build up a statistical estimation model to estimate the area used by the inter-modular interconnect between two or more kernels on an FPGA, based on the size of the kernels, the amount of data exchanged between kernels and the amount of kernels on a chip. Unfortunately, the Xilinx ISE [5] was not capable to place multiple kernels on a FPGA in an easy and useful way. Besides that, the necessity of this model was also questionable. At this moment, it is still not possible to use the Delft WorkBench to place multiple kernels on the FPGA. Therefore, such a statistical estimation model of the interconnect is not applicable in the current situation.

Therefore, the assignment was revised. Still, a statistical estimation model for interconnects was required, but now focusing on the intra-modular interconnect of a kernel.

Using a special C-compiler[1], based on ELSA [6], some SCMs become available. These SCMs are used as input for the prediction model. As output will be the probable amount of interconnect needed by the specific piece of C-code. Using this model is important to predict if a kernel will fit on a FPGA and so whether the Hardware/Software Partitioning is successful or not.

To achieve this, many different software kernels written in C-code will be compiled to VHSIC Hardware Description Language (VHDL) using DWARV [7]. Mapping the VHDL from each software kernel onto the FPGA will provide data about the interconnect resources. This data, combined with the SCMs will form the basis for the model.

1.2 Structure

This report consists of six chapters. The next chapter provides information about the background of this project. Detailed are reconfigurable computing, MOLEN: A reconfigurable computing system [8], [9], [10], [11] and the Delft WorkBench tool chain. Subsequently, chapter 3 translates the problem into a set of operational requirements and discusses some related research and related theory. Chapter 4 describes the methodology of the project and shows the experimental setup. In chapter 5 the experimental results are provided. Finally, conclusions and further recommendations are given in chapter 6. In addition, the tools developed during this project are discussed in the appendices.

[1]C is a general-purpose, block structured, procedural, imperative computer programming language. Although C was designed as a system implementation language, it is also widely used for applications.

2 Backgrounds

In the previous chapter conventional computing was described and what kind of problem is occurring at the moment. We briefly mentioned reconfigurable computing as a solution to this problem. Also the Delft WorkBench was mentioned, which is a comprehensive high-level development platform for reconfigurable computing. In this chapter more information is given on the background studies of this project. In section 2.1 we discuss about reconfigurable computing in more detail. Section 2.3 provides some information about FPGAs, used in reconfigurable computing systems. Subsequently, section 2.4 introduces and explains MOLEN, a reconfigurable computing system. Finally, in section 2.5 the Delft WorkBench is discussed in more detail.

2.1 Reconfigurable computing

Reconfigurable computing (RC) has become an important subject in research. This is due to the fact that it allows executing computationally intense parts of an application in hardware to increase the performance significantly compared to software execution while the flexibility of a software solution still remains [12].

Before the rise of RC there were two main approaches in conventional computing. The first approach was to use hard-wired technology such as Application Specific Integration Circuits (ASICs). This approach has the advantage that it is much faster than software implementation, but the disadvantage is its lack of flexibility. Every time an application changes the ASIC should be redesigned and re fabricated, which is an intensive and expensive task.

The second approach was to use a software programmable processing unit such as an Intel processor. This is already a far more flexible solution than using hard-wired technology. An application written in a high-level programming language, for example C, is compiled to lower-level instructions and executed by the processor. Now if the application changes, only part of the application code has to be rewritten and recompiled. However, the disadvantage of this approach is that it is limited by the capabilities of the processor and by the sequential execution of instructions, leading back to the Von-Neumann bottleneck. This causes poor performance when applications become more computationally intensive.

RC combines aspects of both of these approaches and it aims to fill the gap between software and hardware[12], depicted in Figure 2.1. It has some of the speed of hardware and some of the flexibility of software.

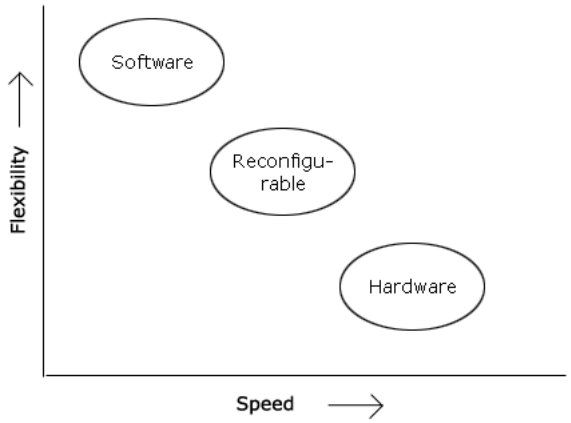


Figure 2.1: The design space between flexibility and performance. Reconfigurable filling the gap between software and hardware

2.1.1 Principles

The basic idea of reconfigurable computing is to combine a General Purpose Processor (GPP) with reconfigurable hardware [13]. An application normally executed on a processor should then be divided into compute-intensive parts and less compute-intensive parts. Executing the compute-intensive parts on the reconfigurable hardware and the other parts on the GPP, should provide significantly better performance.

The early RC systems consisted of two important main components. A GPP and a reconfigurable hardware component, for example a FPGA. These kinds of systems could implement specific functionality on FPGAs rather than on the GPP. The GPP in such a system no longer provided the major computational power, instead the FPGA took over this task. Because the major computations were now done in hardware on the FPGA, it could provide a significantly better performance. These early systems are known as static RC systems [13]. They carry the name static, because only one configuration was loaded onto the FPGA and could not be changed during run-time.

Today, most of the RC systems are run-time RC systems [13]. They have the possibility to reconfigure the FPGA during run-time. Therefore these systems consists of three important main components, the GPP, the reconfigurable part and the Arbiter.

The GPP is a normal processor also used in conventional computing. It executes those parts of the application that are less compute-intensive.

The second component is the reconfigurable part, also called the *reconfigurable unit* (RU). This component actually consists of two subcomponents. The FPGA and the *reconfigurable processor* (RP). The logic of the compute-intensive parts of the application are mapped onto the FPGA, while the task of the RP is to control the logic that is mapped onto the FPGA. Therefore, the RP is extended with an additional instruction set. This instruction set has instructions to set and execute the logic mapped on the FPGA. In section 2.4.3 such an instruction set, used for the MOLEN platform, will be further discussed.

The third component is the so called *arbiter*. The task of the arbiter is to decide, during run-time, whether the application code should be executed by the GPP or by the RU. The arbiter recognizes these additional instructions in the application code and then decides whether it should be executed by the RU or by the GPP. How these additional instructions appear in the application code will be further discussed in section 2.5 on the Delft WorkBench tool chain.

2.1.2 Advantages

RC offers several advantages above conventional computing methods. Combining the speed of hardware execution and the flexibility of software execution the following advantages can be identified.

- *Flexibility*
  Because the FPGA can be reconfigured many times, different configurations can be loaded. Therefore making it unnecessary to change the physical hardware every time an application is updated. This saves a lot of time in redesigning and refabricating of hardware.
- *Cost efficiency*
  Using only one FPGA which can be reconfigured with different configurations, saves a lot of money on purchasing multiple FPGAs for each different configuration.
- *Speed*
  Due to the fact that the computationally intense parts of an application can be done in hardware, rather than on a GPP, the execution time of an application reduces significantly. Due to the reduced execution time the amount of computing power over time increases.
- *Power efficiency*
  Having less execution time for an application, reducing the time the RC system is consuming power. This is making these systems more power efficient.

2.1.3 Disadvantages

Though RC offers several advantages above conventional computing methods, there are also several disadvantages to be mentioned.

- *Application design*
  As mentioned earlier in the introduction, designing an application for RC systems requires a different approach than designing an application for the current conventional computing technology. In such an approach, hardware and software knowledge are required. Because of this, there is a risk that current application designers will ignore reconfigurable technology. In order to prevent this from happening, a comprehensive high-level development platform should be developed. This platform should provide an easy way for the application designer to implement their application. The Delft WorkBench is such a platform and will be further discussed in section 2.5.

- *Speed*
  Although executing an application on a RC system is faster than executing it only on a GPP, the use of ASICs still remains the fastest solution for executing an application in hardware.

- *Reconfiguration delay*
  By requiring multiple reconfigurations to complete a computation, the time it takes to reconfigure the FPGA becomes a significant concern. The systems should be idle during this reconfiguration time, wasting a lot of processing cycles that otherwise could be used for useful work.

2.2 Examples of RC systems

RC systems come in various shapes and sizes. These systems can be classified by four distinctive properties: reconfigurability, granularity, coupling and reconfiguration time. More information about these properties can be found in [14]. In this section we will only focus on three of them. Below are some examples of RC systems given.

2.2.1 Field Programmable Gate Arrays

FPGAs are reconfigurable Very Large Scale Integration[1] (VLSI) components that allow for the implementation of arbitrary sequential and combinatorial circuits which are described in a hardware description language. Therefore, they can be seen as a reconfigurable computing system. The simplest form of an FPGA consists of an array of Configurable Logic Blocks (CLBs), a set of input and output blocks (IOBs) and a programmable interconnect architecture, see also section 2.3.

- *Reconfigurability*
  FPGAs are widely used as an alternative for ASICs, because they are reprogrammable. Therefore, allowing to implement different applications just by reprogramming the FPGA. Examples of applications that can be implemented on an FPGA are real-time digital signal processing and data encryption.

- *Granularity*
  FPGAs can be considered as fine-grained computing systems. The CLBs on a FPGA are fine-grained components. They can be used as a replacement for two to six simple logic gates or a single flip flop in a gate level circuit design. Most of the commercially available FPGAs also contain larger, coarse-grained blocks, such as ALUs, which provide commonly used functionality.

- *Reconfiguration time*
  The reconfiguration time of an FPGA is specific to each FPGA series. FPGAs require a complete reset before a new application can be programmed. The reset is done by an external bit stream, which can take several seconds to perform. This

[1] VLSI is the process of creating integrated circuits by combining thousands of transistor-based circuits into a single chip

makes them statically reconfigurable only, because a delay of several seconds is too long for reconfiguring it during run time. At present, some FPGA series support limited runtime reconfiguration, for example the Xilinx Virtex4 series.

- *Coupling*
  Because an FPGA is a single VLSI component, it cannot be used as a generic computing platform. In order to make it suitable as a generic computing platform, it should be mounted on a printed circuit board that provides I/O facilities. An FPGA mounted on such a board is an example of a fully reconfigurable architecture.

2.2.2 PipeRench

PipeRench is Carnegie Mellon University's answer to the reconfigurable computing challenge. The system, which is particularly suitable for stream based media processing applications, was designed with the benefit of hindsight on FPGA based reconfigurable computing systems. The PipeRench fabrics aims to beat FPGA based computing machines on five points, as described in [14].

- *Reconfigurability*
  The PipeRench design consists of pure reconfigurable fabrics and thus in itself is a fully reprogrammable system. However, unlike FPGAs, the PipeRench is not suitable for complete System-on-Chip (SoC) implementations. The designers of the system envisioned PipeRench as a co processor. It requires an additional computing system in order to be used as a full computing platform.

- *Granularity*
  The configurable logic units in PipeRench are designed for computation purposes. The PipeRench chip consists of so called stripes which represent pipeline stages in a computation. A stripe contains a number of configurable processing elements such as ALUs. Therefore, PipeRench can be seen as a coarse-grained fabric.

- *Reconfiguration time*
  A powerful feature of the PipeRench is its extremely short configuration time. Its short reconfiguration time enables the PipeRench architecture to simulate long virtual pipelines on hardware with minimal overhead.

- *Coupling*
  PipeRench was designed to function as an attached co processor in a general purpose computing system. The co processor is intended to be used as a loosely coupled system, which processes longer instruction sequences with relative autonomy.

2.3 FPGAs

There are several manufacturers that produce FPGAs based on static memory (SRAM). A couple of these are Xilinx [15], Altera [16] and Atmel [17]. Each of these manufacturers have their own architectural implementation, but the basics are the same. This report

will refer to the architectural implementation of Xilinx, because the Xilinx ISE, discussed in Section 2.2.3, is used to synthesize the software kernels.

The basics of an FPGA architecture are depicted in Figure 2.2 [18].

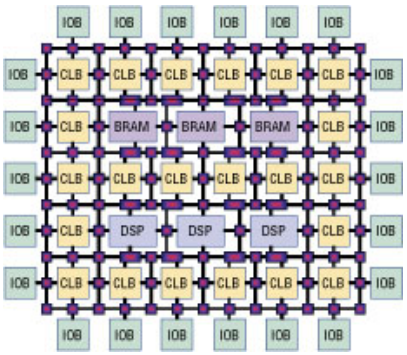


Figure 2.2: FPGA internal structure based on the Xilinx architecture style.

A FPGA is a semiconductor device consisting of configurable logic blocks (CLBs), interconnects, and input/output blocks (IOBs) that allow implementing complex digital circuits. On the outside of the FPGA the IOBs form a ring for connection I/O pins that are situated on the exterior of the FPGA. Inside this ring lies a rectangular array of logic blocks. A typical FPGA logic block consists of a four-input lookup table (LUT) and a flip-flop. Modern FPGA devices also include higher-level functionality such as Digital Signal Processing (DSP), high-speed IOBs, embedded memories (BRAM) and embedded processors. The programmable interconnect wires are required to connect CLBs to other CLBs and CLBs to IOBs.

A slice (Xilinx terminology) contains a small set of building blocks (LUTs, flip-flops and control elements). This is the basic unit area when determining the FPGA-based design size. CLBs consists of several number of slices. Modern FPGAs consists of tens of thousands of CLBs and a large programmable interconnection network. In Figure 2.3 a CLB is depicted.

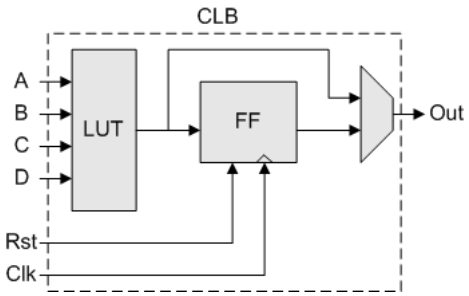


Figure 2.3: Internal structure of a CLB.

Unlike ASICs which perform only one specific function, FPGAs can be reprogrammed many times to perform a different function. For programming the FPGA, code written

in a Hardware Description Language (HDL) is used, for example VHDL.

Since their introduction in 1985, FPGAs [19] have been used in various systems implementing a broad range of applications. In most of these systems, they are used to implement certain logic, providing high-level integrated circuits without the expenses and risks that are involved with using ASICs.

Where the performance of FPGAs increased rapidly, their use in RC systems payed considerable attention [19]. With the inheritance of speed and parallelism from a hardware solution, FPGA-based co-processors are used to execute compute-intensive tasks while maintaining the flexibility of a software programmable solution. Due to this fact, the first systems with FPGA-based co-processors that compete with parallel computers and even supercomputers have started to emerge.

2.3.1 Interconnect on a FPGA

As described above, there is a huge amount of logical blocks on a FPGA. In order to realize a working system, it is necessary to connect those blocks to each other. The whole process of creating and programming this interconnect structure is called routing. This process is very important for the speed of the design, because the wiring is responsible for a big part of the delays on a FPGA. This is due to the capacitors between the wiring and the ground.

On a FPGA, a reprogrammable interconnect system is available to take care of this. About 80% of the FPGA area is reserved for this system [20].

The system exists of metal wires, laying on the chip. All inputs and outputs of the logical blocks on the chip are connected to one of those wire segments. Those wire segments are one or more logical blocks wide. At last, the wires segments are connected to each other with so called *configuration interconnect points* (CIPs). This is shown in Figure 2.4.

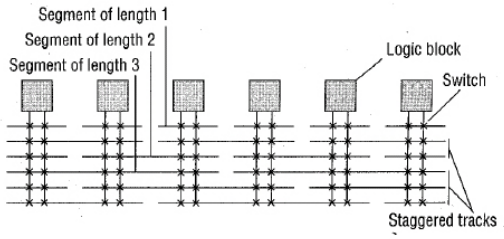


Figure 2.4: Wire segments in an interconnect structure

There are several types of CIPs, but a basic structure consists of a transistor which couples two wire segments. The gate of this transistor is controlled by configuration memory bits. The different types of CIPs can be found in [21]. There are multiple layers of wires present on the chip. The amount of layers can be more than eight. In one of the layers, wires will run from up to down, while in one other layer, the wires will run from left to right. Where the wires cross each other, there'll be a switch point (i.e. CIP).

Figure 2.5 [22] shows how CIPs will connect all wire segments from multiple, individual layers. Also, you can see a extensive version of a CIP, using six transistors to be able to connect every wire segment to another.

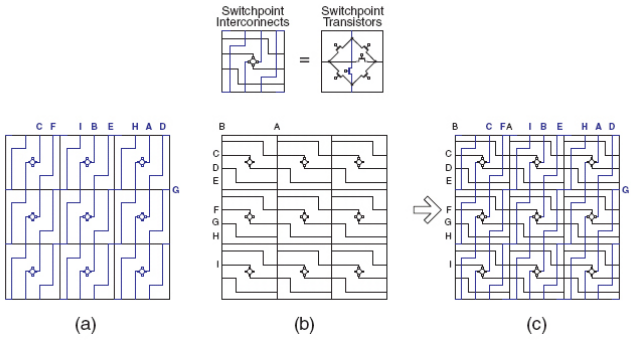


Figure 2.5: A typical interconnect structure with wires and switch points

In this thesis you will read more about *nets*. It can be that only two logic blocks are connected, but mostly, the output from one block is used as input for several others. One complete connection from one logic block to one, or multiple others, is called a net.

2.3.2 Xilinx: Profile and design flow

Xilinx leads the Programmable Logic Device (PLD) market - one of the fastest growing segments of the semiconductor industry [15]. This market features a technology called FPGAs. Xilinx offers a lot of different FPGA series and design environments for these series. One of these environments is the Xilinx ISE. This design environment provides the designer with easy-to-use built-in tools for synthesizing and implementing HDL for all leading FPGA series. The design flow for synthesizing and implementing HDL is depicted in Figure 2.6 [5].
The scope of this project will mainly focus on the synthesis and implementation stage, because these parts provide the necessary data for the statistical estimation model.

- *Synthesis*
  The input of the synthesis stage is a HDL design file, for example VHDL source code. During this stage, behavioral information in the HDL design file is translated into a structural netlist and optimized for the selected FPGA series. The output, a netlist (NGC) file, is then translated into a Xilinx Native Generic Database (NGD) file. This file contains a logical description of the design in terms of logic elements, such as AND gates, OR gates, decoders, flip-flops, and RAMs. This file is further used in the Mapping stage.

- *Mapping*
  The input file is the NGD file created in the synthesis process. During the mapping stage a logical design is mapped to the selected FPGA series. First it performs

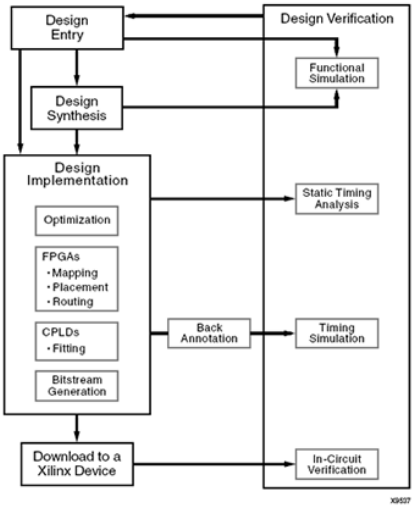


Figure 2.6: Xilinx design flow

a logical Design Rule Check (DSC) on the design in the NGD file. After that, it maps the design logic to the components (CLBs, IOBs and other components) on the selected FPGA series. The output of this stage is a Native Circuit Description (NCD) file. A NCD file is a physical representation of the design mapped to the components in the selected FPGA series. This file is further used in the Place and Route stage.

- *Place and Route*
  The input file of the Place and Route stage is the NCD file created in the mapping process. The Place and Route stage places all components and routes the interconnect on certain constraints (timing, costs). The output file is another NCD file, but a little more optimized due to the constraints. This file can then be put through a Timing Reporter And Circuit Evaluator (TRACE) to verify if the timing constraints were met.

2.4 MOLEN: A reconfigurable computing system

As discussed in the previous chapter, reconfigurable computing bears great promises for the future. Therefore it is necessary to do research in what kind of configuration reconfigurable computing systems perform best. Different kinds of proposals have been done, but all with shortcomings, mentioned in [8, 9]. These shortcomings are listed below.

- *Opcode space explosion*
  Every time a new application is mapped on an FPGA, new instructions are required. Each new instruction requires a new opcode. Therefore if a wide variety of

applications will be mapped on an FPGA, a large amount of opcodes are needed. However, every proposed architecture has only a limited amount of space for opcodes available.

- *Limitation of the number of parameters*
  In a number of proposals the applications mapped on an FPGA could only have a limited number of in and output parameters. Therefore the amount of parallelism is limited to a certain level.

- *No modularity*
  In each proposal the configuration is bounded to one specific reconfigurable technology or design. Therefore, the applications designed for this configuration could not be easily ported to another configuration. Also there is a lack of mechanisms that allows designers to design applications independent of the reconfigurable technology used.

- *Support for parallel execution*
  Many of the proposed architectures do not support executing sequential data-independent operations or configurations in parallel.

2.4.1 Overview

At the Delft University of Technology, some members of the Computer Engineering group have developed a reconfigurable computing paradigm together with an accompanying platform, called the MOLEN platform. This platform aims to overcome the shortcomings mentioned above. An overview of the MOLEN platform is depicted in Figure 2.7 below.

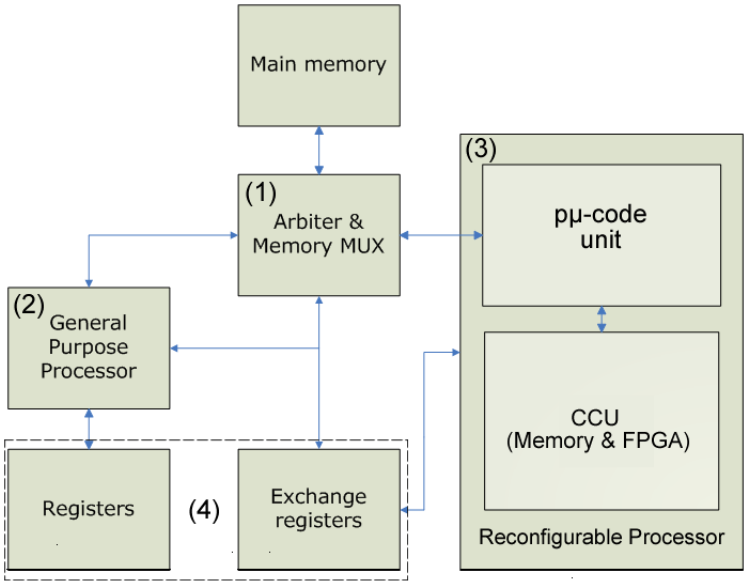


Figure 2.7: Overview of the MOLEN platform with four main components.

As can be seen in Figure 2.7 the MOLEN platform exists of four main components. First the instructions are fetched from the memory. The *arbiter* then decides whether the instruction should be send to the *general purpose processor* (GPP) or to the *reconfigurable processor* (RP). If sent to the RP, the microcode[2] unit ($\rho\mu$-code unit) interprets and executes the instruction. The actual configuration is mapped on the FPGA. When done executing, the results can be stored in the exchange registers making them available for the GPP to use.

2.4.2 Components

- *The arbiter*
  The MOLEN platform is extended with an additional limited instruction set of eight instructions, called the polymorphic Instruction Set Architecture (πISA) [8]. This extension provides instructions for setting and executing configurations, passing results and parameters to the exchange registers, and the possibility to execute different configurations in parallel. When an instruction is fetched from the memory, it is the task of the arbiter to decide, by distinguishing one of these eight additional instructions, if the instruction must be sent to the GPP or to the RU. The eight additional instructions are discussed in section 2.4.3.

- *The general purpose processor*
  The general purpose processor is a normal processor also used in conventional computing systems. It executes the instructions that are not send to the RP. For the MOLEN platform a PowerPC 405 processor is used [11].

- *The reconfigurable processor*
  The reconfigurable processor (RP) consists of two main parts. The CCU, a combination of memory and an FPGA, where the actual configuration is mapped, and the $\rho\mu$-code unit, which interprets and executes the incoming instructions. In section 2.4.3, a set of eight additional instructions for the RP is described. These instructions are implemented in microcode. The $\rho\mu$-code unit interprets the incoming microcode and depending on what kind of instruction it receives, it communicates with the CCU, the Arbiter and/or the Registers and Exchange registers.

- *Normal registers and exchange registers*
  Two kinds of registers or register files are used in the MOLEN platform to store and exchange data. The GPP stores its data in the registers. To exchange data between the GPP and the RU the exchange registers are used. For this purpose the additional instructions `movfx` and `movtx`, as mentioned in the section 2.4.3, can be used.

[2]Microcode is used in microprogramming that can be employed to implement machine instructions in a CPU relatively easily.

2.4.3 Programming paradigm

The arbiter uses a special instruction set, called the polymorphic Instruction Set Architecture (πISA), to control the FPGA. This set contains eight instructions; six to control the hardware and two instructions are used to control the registers. Below, these instructions are listed [23].

- `p-set` - controls the setting of those configurations that cover common parts of multiple functions and/or frequently used functions.
- `c-set` - controls the setting of configurations of the remaining blocks on the FPGA (not covered by the p-set) to complete the FPGA functionality
- `execute` - controls the execution of the operations implemented on the FPGA. These implementations are configured onto the FPGA by the set instructions.
- `set prefetch` - prefetches the needed microcode responsible for FPGA reconfigurations into a local on-chip storage facility in order to possibly diminish microcode loading times.
- `execute prefetch` - The same reasoning as for the set prefetch instruction holds, but now relating to microcode responsible for FPGA executions
- `break` - Used for synchronization between the GPP and the RU, when executing in parallel. It halts the execution of instructions following the break statement, allowing the execution of instructions in parallel.
- `movtx` - Moving content from a GPP register to an exchange register
- `movfx` - Moving content from an exchange register to a GPP register

Figure 2.8 shows how a normal compiler handles a certain part of C-code and how the special MOLEN compiler does this. As you can see the call in the normal code is replaced by `set` and `execute` instructions.

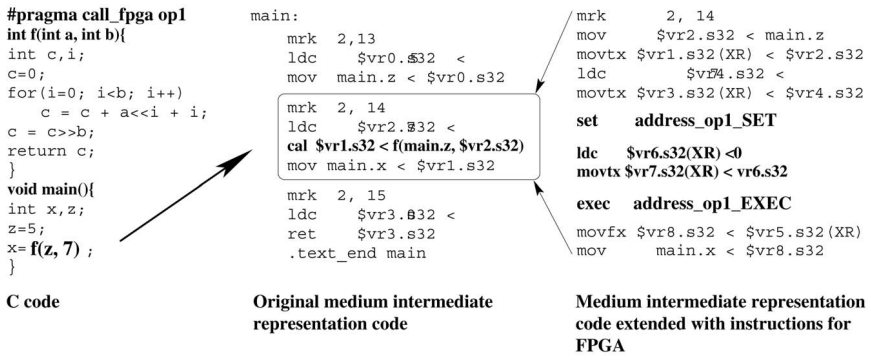


Figure 2.8: Medium intermediate representation code [8]

2.5 The Delft WorkBench toolchain

As mentioned earlier in the introduction, designing an application for RC systems requires a different approach than designing applications for conventional computing systems [24]. In such an approach, the required knowledge from hardware forms an obstacle for current application designers to use RC. Therefore, the need for a comprehensive high-level development platform that supports application development is essential in order to prevent designers from ignoring the reconfigurable technology.

2.5.1 Overview

At the Delft University of Technology a part of the Computer Engineering group has developed a tool chain, called the Delft WorkBench (DWB) [3]. The DWB covers the entire design cycle for designing an application for a RC system. An overview of the DWB is depicted in Figure 2.9 below.

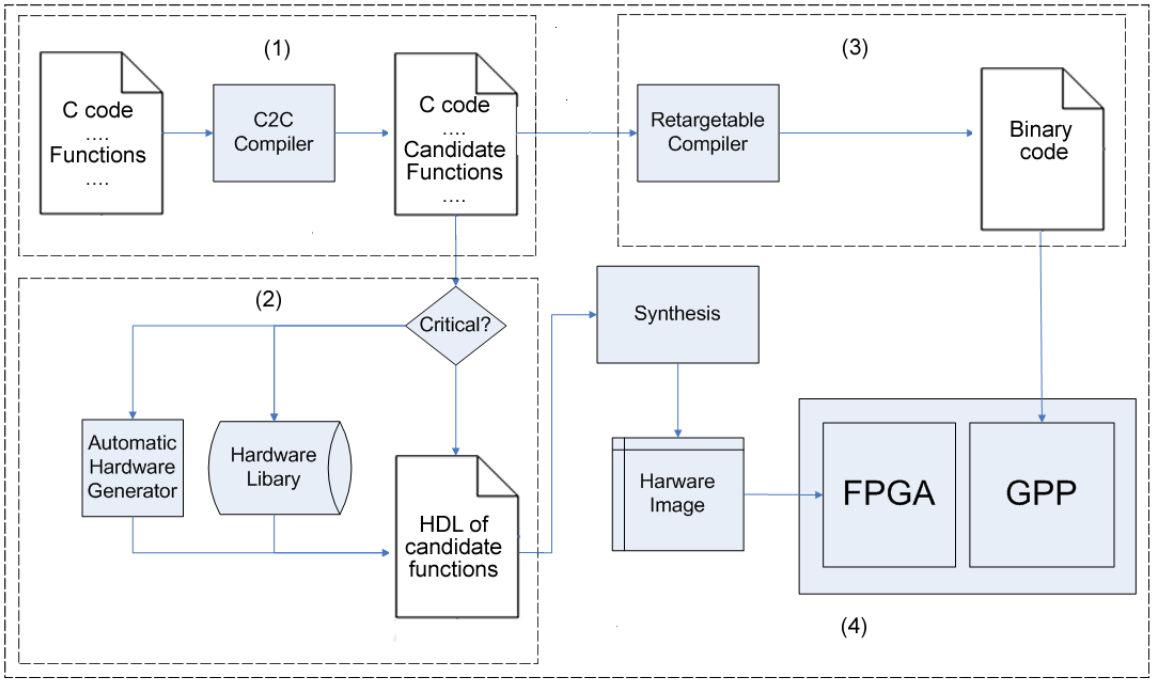


Figure 2.9: Overview of the Delft WorkBench tool chain which covers the whole design process.

As can be seen in Figure 2.9 the DWB exists of several phases. In phase 1 candidate functions are selected and transformations and optimizations are performed on them. Subsueqently, in phase 2, the candidate functions are translated to VHDL code. This can be done in three ways which are further discussed in 2.5.2 item VHDL generation. In phase 3 the Retargetable Compiler translates the C-Code to the right instructions for

the reconfigurable platform. Finally in phase 4, everything will be integrated and the results will be validated. The DWB mainly focuses on the MOLEN platform.

2.5.2 Design flow

Within the DWB, applications written in C-code have to go through several phases before they can be executed on the MOLEN platform. These phases are listed below [3].

- *Code profiling and cost modeling*
  An application often consists of several functions. By characterizing each function on performance and area metrics, this phase aims to find a set of candidate functions that offers the optimal increase of performance when implemented in hardware. To support this phase a profiler is offered that collects and analyzes the data to predict the area and performance. This information can then be used for characterizing each function. At the end of this phase a set of candidate functions is determined for implementation in hardware.

- *Code transformations and optimizations*
  In this phase, the candidate functions are optimized and eventually transformed. The compiler will focus on two main optimizations, namely graph restructuring and loops parallelization. Loops can be seen as an important candidate for code optimization. Loops parallelization and optimization searches inside loops (f.e. `for` and `while` statements) for which parts can be executed parallel. More information about loop transformation, parallelization and optimization is in [25] and [26].

- *Retargetable compiler*
  The DWB tool chain provides a compiler to compile the application to the MOLEN platform. Each function in the optimal set will be translated to the correct instructions for the FPGA. These instructions are mentioned earlier in subsection 2.4.3. Then, the compiler retargets these instructions in the original compiled code. The compiler also has to deal with the scheduling problems that arise when executing the functions in parallel with the GPP. A `set` instruction causes the FPGA to reconfigure itself. This will introduce some reconfiguration latency that has to be taken into account while scheduling the instructions.

- *VHDL generation*
  When the optimal set of functions is determined, they have to be translated to VHDL. The DWB tool chain offers three methods for translating the C-code to VHDL. The first method for the designer would be to translate the C-code to VHDL manually. This requires knowledge of hardware, but it offers the best quality designs. The second method is to use existing VHDL from a hardware library. Such an hardware library consists of VHDL segments, IP-cores and design patterns, from often used algorithms. The third method is to automatically generate VHDL with the DWARV[3] compiler. This is the easiest and fastest method. However, it lacks good optimization of the VHDL code.

[3] DelftWorkbench Automated Reconfigurable VHDL Generator, an automated VHDL generator designed for the DWB tool chain [7]

- *Integration and validation*
  After the optimal set of functions is compiled and re targeted by the Retargetable Compiler it is ready to be integrated on the MOLEN platform. Running it on the MOLEN platform will provide statistics about the performance. These statistics are used to validate the process and refine the code profiling and cost modeling stage. This can be iterated many times until the designer is satisfied with the results.

Interconnect estimation from C-code

3

In the previous chapter we described the background of the project. In this chapter we discuss some more details about the problem, described in the introduction. In section 3.1 we discuss the operational requirements of the statistical estimation model and the Prediction Tool. Section 3.2 presents some related research, followed by section 3.3 where the Quipu model is described. Finally, in the last section some related theory is provided.

3.1 Operational requirements

To find a suitable solution for the problem as discussed in Section 1.1, a set of operational requirements were formulated. This set describes what the prediction model and the Prediction Tool should do and how it should be made. This set of requirements follows from the way how our model will be used in the DWB tool chain.

3.1.1 Operational requirements for the prediction model

- Given a predefined set of software metrics, the prediction model should be able to predict the amount of interconnect resources (nets) on the FPGA, used by this piece of software. This model should be accurate enough to use in the DWB tool chain.
- The model should be based on partial least squares regression.
- The model should be generated in R, a programming language specialized in statistical problems.
- The model should be represented in an XML-format file.
- The model should be able to make predictions for the Virtex2 platform as well as for the Virtex4 platform.

3.1.2 Operational requirements for the accompanying tool

- Given the specification of the model and a set of software complexity metrics and FPGA specifications, the accompanying tool should be able to make predictions of the interconnect resources needed.
- The accompanying tool should come along with a built-in help function or a readme.txt
- The accompanying tool should work cross-platform.

- The accompanying tool should be command line based.
- The accompanying tool should run standalone as well as in an automated tool chain.
- The source code of this tool should be commented as much as possible, in the English language.
- The input and output files of the accompanying tool should be in an XML-format.
- The accompanying tool should be able to calculate the utilization for different FPGA devices which are described in an XML-format file.

3.2 Related research

Many approaches in prediction of interconnect resources have been presented in the past. Most of these approaches are based on a lower level than C-code.

Many prediction models were initially based on Rent's rule [27]. Rent's rule was first described by Russo and Landman in 1971. Rent's rule describes the relation between the number of pins at the boundaries of integrated circuits and the number of components (e.g. logical blocks).

That interconnect problems shows up in other subjects, is made clear in [28]. The authors presented a new method of high-level synthesis for devices that accomplish computationally dense operations, like complex ASIC systems. In order to make that work, they tried to allocate and optimize the interconnections on the chip.

In [29], several methods for wire length estimation are discussed. Those methods are based on Rents exponents of partitioning or placement. Wire length estimation is difficult for large circuits and depends on several factors, such as the placement algorithm in the design flow and the parameters of the global router. The authors encountered that extensive data is required to make precise wire length estimations.

In [30], an early estimation scheme from C-code is presented. The authors are presenting a custom-built model for execution time based on the SPARK C-to-VHDL compiler. Using a set of 9 kernels from 2 applications they end up with an error of 39.3% and 44.4% compared to the actual execution time.

In [31], the authors propose an a priori interconnect and wire length estimation methodology for producing half perimeter wire length estimates for every single net in the circuit. A priori techniques estimate these parameters without actually performing circuit placement. Using properties of the circuits together with the FPGA limitations the authors derive important parameters about the wire length. Their method has an average error of 11.6% in wirelength compared to the actual measurements.

In [32], a multi-dimensional quantitative prediction model for hardware-software partitioning (Quipu) is presented. Quipu offers a prediction model based on linear regression between software metrics, from a set of 127 software kernels written in C-code, and hardware measures from their corresponding design. Currently Quipu still has a large error compared to lower level prediction models, but it offers fast and early predictions and

supports a wide variety of applications. At the moment Quipu only takes area measures like Slices, Flip-Flops, or LUTs into account.

Interestingly, the authors in [33] investigated the statement that the industries do not make use of interconnect prediction models. Existing models are too inaccurate to be useful for industry. They give four main reasons for that inaccuracy. At last they put forward, that it would be very useful if a tool was designed that could predict routing requirements directly from HDLs or higher level languages. Due to the absence of the use of wires in HDLs, designers do not have much notion of this routing requirements. If those requirements does not fit the available amount of wiring on the FPGA, it will reveal in a late stage in the design process (after synthesis and mapping, see section 2.3.2).

So that means that our project is relevant to the industries. In our project we will try to extend the Quipu model with the area measures for the interconnect. The area measures for the interconnect are quiet important, because the available wires on an FPGA is limited. Kernels with a huge amount of interconnect can inhibit the use of other kernels, despite the sufficient amount of computational elements that might be available.

3.3 Quipu

Quipu is a quantitative prediction model based on Multi Dimensional Linear Regression using Software Complexity Metrics. This model provides estimates of certain resources for implementing applications on hardware from a high-level programming language like C. The model is based on the hypothesis that software and hardware complexity are related. In [4] is shown that for area measures, like slices and flip flops, this hypothesis is true. For other resources this hypothesis has not been profen yet.

The earlier mentioned Software Complexity Metrics (SCMs) represent different aspects of the complexity from computer programs and functions. SCMs are not a new concept, they already have been used in software development processes to predict development time or the number of errors. Therefore, the advantage of using SCMs in Quipu, is that some SCMs are already available. Those which are not already available are often quiet easily to determine in a relatively short time. For the Quipu model 24 SCMs are gathered. A detailed description of these SCMs can be found in [4]. These SCMs cover a wide range of applications from encryption to multimedia purposes. As shown in [4] some of these metrics correlate with hardware, but some of these metrics also correlate with other metrics. For example the Average Path Length, Maximum Path Length and Statements are all a measure for the length of an application. This correlation among the SCMs is called linear interdependence or multicollinearity. Because for classical linear regression independent variables are required, the set of metrics are transformed using Principal Component Analysis (PCA). More details about PCA can be found in [4].

Quipu focuses on the MOLEN platform. Because MOLEN is a reconfigurable computing platform, Quipu should not only predict area measures, it should also cover delays, such as reconfiguration delay. At the moment Quipu only covers area measures. Therefore, extending the Quipu model with delay estimation could be an improvement. This is not the only improvement that can be made for Quipu. A list of other improvements

can be found in [4]. As mentioned earlier this report focuses on the estimation of the interconnect resources, one of the improvements mentioned in [4].

3.4 Related theory

3.4.1 Statistical Modeling

The results obtained from the interconnect modeling should somehow give insight in how many nets a kernel will approximately contain once it is mapped to an FPGA. A common approach to problems such as finding the interconnect from certain C-code is statistical modeling. Statistical modelling is widely applied throughout a diverse set of problems including those that arise from social sciences, environmental sciences and economical sciences. Statistical modelling allows us to find any kind of empirical relation between causes, or *predictors*, and effect, or *response*.

The drawback of modelling is that the mathematics behind the modeling theory will always fit the predictors to the response. It does not matter whether a relation between predictor and response actually exists, or is just mere coincidence. There is also a risk of overfitting, which happens when there are to many predictors compared to the number of responses. If the number of predictors is large enough, there will be a linear combination of the predictors that suit the response well with an acceptable small error.

Fortunately, there are a number of tests to check to which degree a model is useful. A well-known method to check the goodness-of-fit of a linear model is the Aikike Information Criterium (AIC) [34]. Other uses are Mallows' C_p [35] and the adjusted R^2. These functions calculate a number based on the residuals and the number of parameters. Their formulas are:

$$AIC = 2k + N\left(\ln\frac{2\pi RSS}{N} + 1\right) \tag{3.1}$$

$$C_p = 2k - N + \frac{\sum_{i=1}^{N}(Y_i - Y_{i,pred})}{\overline{RS}^2} \tag{3.2}$$

$$Adj.\ R^2 = 1 - \left(1 - \frac{\mathrm{var}(RSS)}{\mathrm{var}(Y_{pred})}\right)\frac{N-1}{N-k-1} \tag{3.3}$$

where k is the number of parameters, N the number of observations, RSS the residual sum of squares, Y the response, Y_{pred} the predicted response and $\overline{RS}$ the residuals mean, which, when unbiased, is the same as the squared *Root Mean Square Deviance* (RMSD, see formula (3.4)), for all number of model components.

$$RMSD = \sqrt{\frac{\sum_{i=i}^{n}(x_{true,i} - x_{pred,1})^2}{n}} \tag{3.4}$$

.

We can also see that all the criteria are based on the number of parameters and the total error (the structure of the adjusted R^2 is comparable).

A model can be selected using one of the variables stated above. These parameters are important to select a best model, since they take the number of parameters into

account. A bigger model will always fit at least as good compared to a smaller model, since the additional parameter does not have to be used. However, if it can gain even the slightest increment in the quality of the fit, an exhaustive algorhitm will include it in a model.

The reason that the goodness of fit calculations punish the inclusion of extra model predictors is twofold. First, each extra predictor adds extra noise to the model, so too many predictors will render a model useless, despite excellent fitting. The other reason is simplicity. A model is used to get insight in a relation between response and predictors. Too many parameters will diminish the understandability of the model. If a model is not well understood, it is hard to tell whether it makes any sense.

3.4.2 Linear regression

A common way tot fit data is the use of linear regression. A linear regression algorithm will try to fit data by minimizing the residual squares of sums. A linear model will have the general form of

$$Y_i = \beta_1 f(X_{1,i}) + \beta_2 f(X_{2,i}) + \ldots + \beta_n f(X_{n,i}) + \epsilon_i \tag{3.5}$$

where β_j are the parameters of the regression model, X_j the model predictor and $f(X_j)$ are predictor transformations. Any model that can not be written in the form of (3.5) is not a linear model. Transformations are useful when e.g. a linear predictor results in an exponential response. There is no easy way to choose which transformation to use on the data. Usually a visual analysis of the data helps to decide if a transformation is useful. The least squares method used by the R `glm` function. It assumes the error ϵ_i is normally distributed, which is a good assumption when there is no thorough understanding of the underlying relations in the data set. We have chosen to use R 2.7.0 for statistical calculations. R [36] has a very good support for statistical graphics output and statistical testing on data.

3.4.3 OLS and PLS models

There a several types of models. We have chosen to make an OLS and a PLS model. OLS stands for Ordinary Least Squares where PLS stands for Partial Least Squares. In an OLS model, it is tried to minimize the total sum of squared errors. By squaring the errors, they will add up instead of compensate each other (in the case of positive and negative errors). Also, by squaring the errors, big errors will have more influence in the model rather than small errors.

A PLS model model relies on the theory of principal components. These are found by taken the orthogonal eigenvalues of the covariance matrix of the predictor set. The benefit of these principal components is that the try to descripe the data with only the *important* predictors. PLSR modelling is better suited towards prediction then OLSR modelling. Since OLSR is the de facto standard way of elementary modelling, but PLSR will theorethically perform better, both methods will be worked out in the results chapter.

3.4.4 Validation of statistical models

The art of fitting is not to let a computer do a number crunching job. The art is to find models that do actually make sense for the purpose they are needed and designed for. Therefore, the model needs to be validated in some way. There are a number of ways to do this.

The Root mean square deviance (RMSD) is a tool which is often used as a measure comparable to the standard deviance of the model, which gives in indication of how well the model fits.

Since the model is used for predicting, this also must be measured. The RMSD can also be used for checking the cross-validated models. This makes the OLS and the PLSR models directly comparable in the field of predictablilty. The downside of this approach is that the RMSD for cross validation for different models is calculated by two different packages, so there could be differences in calculations.

4 Methodology

Chapter 3 presented more details about the problem. In this chapter, we describe the different steps that need to be taken to eventually build up the statistical estimation model.

4.1 Generating VHDL code from the kernels

The first step in this process is to generate VHDL from the kernels written in C-code. This is done by a compiler in the Delft Workbench tool chain. The compiler is called DWARV and is developed by the Computer Engineering group at Delft University [7]. Due to some complications, we were not able to compile the kernels ourselves, so we were provided with a set of already compiled versions of the kernels. This set existed of 127 kernels.

4.2 Synthesizing & Implementing the VHDL code

The second step in the process is to synthesize and implement the VHDL code from each kernel. This is done with the Xilinx ISE, a design environment that provides the designer with easy-to-use built-in tools for synthesizing and implementing VHDL, see section 2.3.2. With the Xilinx ISE, it is only possible to synthesize and implement the kernels one by one. Because of the large number of kernels, it is a time consuming process doing this for each kernel manually. Therefore, a tool has been written that automates the whole Xilinx ISE process of synthesizing and implementing. This tool is called the Xilinx ISE Tool, discussed in Appendix A. During the Xilinx ISE process the tool produces several log files containing data necessary for building up the statistical estimation model.

4.3 Extracting necessary data from the log files

From the log files, created during the Xilinx ISE process, the necessary data needs to be extracted. Doing this manually for each log file is a time consuming process. Therefore, another tool has been written called the log files tool, discussed in Appendix B, that extracts all data from the log files. The data extracted from the log files is written to an output file.

4.4 Building up the statistical estimation model

Since modeling takes a lot of computations, this task has to be done with mathematical software. We choose to make exclusive use of the R programming environment for this

task due to its flexibility and powerful statistics engine.

To build the model, two types of regression analysis have been considered, namely *ordinary least squares* (OLS) and *partial least squares* (PLS) ([37], chapter 9.2 and section 3.4.3 of this report). To build a model with OLS we have used the `regsubsets` package to select which parameters to include in the model. To build and analyze the PLS model we have made extensive use of the `pls` library, which automatically fits a PLS model and calculates the RMSD.

The purpose of the model is to find a relation between certain C-code parameters and the number of nets in the FPGA design for a specific FPGA architecture, in this thesis the Xilinx Virtex-II Pro and the Virtex-4. Since the goal of the modeling is to find a model that makes good predictions over physical understanding of the relation between C-code and interconnection, we will focus on the PLS modeling technique ([37], chapter 9.2). A PLS model can be generated in R as follows with the so-called *leave one out* cross validation:

```
library(pls)
model <- plsr(response ~ predictors, validation= "LOO")
```

With the `validationplot` the relation between RMSD model parameters can be visualized to select the PLS model with the smallest error.

4.4.1 Transformations on the data

Although the technique of linear modeling assumes that there are linear relations in the form of $\mathbf{y} = \beta\mathbf{X}$, this is not a general principle in the relation between predictors and response. Some relations take a form which closer resembles an exponential, logarithmic or polynomial relation.

There is no way to obtain these relations in a structured manner, so they have to be found by inspection. This can be done by plotting a predictor against the response. If there is a visible relation between the two, but not a linear one, a transformation can be tried. Testing whether a predictor is effectively linearized against the response can be done by checking for normality when the response is divided by the appropriate β for the predictor it is tested against. There are a number of tests to check for normality, each having their own application. Easier would be to verify by inspection that the relation is linear, since there is no guaranty whatsoever that the model residuals are normally distributed.

4.5 Analyzing the model

Fitting a model is relatively easy, since calculations can be performed by software. More difficult is understanding and correctly interpreting the steps involved in verifying the models.

Unfortunately, there is no single strategy or methodology for generating a linear model. Each situation requires an application specific approach to the problem. A computer makes every predictor fit to a response with a least squares method, even

when no such relation exists at all. There are, however, dozens of tests to see how models compare to each other. For the OLS case, several plots for criterion based model selection (i.e. the best predictors for a predefined number of parameters) have been included in the figures section, such as selection on the Bayesian information criterion, Mallows' C_p and the adjusted R^2 (see [37] chapter 8). Testing tools in the `pls` package lack these criteria, making them somewhat hard to directly compare.

One way to select which model works best is to look at the confidence intervals of the reported error. Since the sample contains only 118 units $U_1, U_2, \ldots, U_{118}$ over a theoretically infinite sized population $\mathcal{U}$, we need some method to enlarge our sample space $U_1, U_2, \ldots, U_{118}$. The general way to do this is by applying resampling methods. The bootstrap method (see [38] for a thorough explanation) is a general applicable method for resampling. The R programming environment offers a good library routine for bootstrapping. For this particular case, we are interested in the model with the smallest error and confidence interval for those errors. A good hands-on introductory tutorial on bootstrapping and permutation tests, another resampling method, can be found in [39].

Another way to make good use of the bootstrapping package is exploiting its capabilities for cross validation for generalized linear models , using the `cv.glm` function:

```
library(boot)
cv.err <- cv.glm(data, linear.model)
```

Nowadays computer power is not a big problem any more, so both modeling techniques OLS and PLS can be validated by the leave-one-out algorithm.

The confidence interval will be calculated on the standard deviation of the errors of the PLS and OLS model. There are no hard rules for selection either model based on the confidence interval and cross-validation information, but the one with the lowest cross validation error will presumably do.

4.5.1 Elimination of data outliers

As with almost any data set, there are values which in some way do not seem to fit in the model. There can be two reasons. First, a measurement or data collection error could have been made. Second, some kernels might exhibit some anomalous behavior, translating into deviating SCM measurements. We examined some outliers, namely that ones that appear on the Virtex2 platform as well as on the Virtex4 platform. These outliers are `havalTransform3`, `intfdct`, `g721_body`, `gost_encrypt`, `cast128_decrypt` and `cast128_encrypt`. Because we want to figure out what types of C-code causes these types of outliers, it is necessary to know which metrics are responsible for those kernels being outlied. Therefore we investigated the total amount of nets divided by the value of one of the metrics and compared them to the normal kernels. Unfortunately, there was not a clear metric which we could depict as being the cause. The only two metrics which show some bit of exceptions were the AICC and the Oviedo.DU.pairs. All the outliers have relatively high nets per AICC or per Oviedo.DU.pairs. Nevertheless, we could not find something specific in the C-code. Extensive analysis on this outliers is one of our recommendations (see chapter 6).

4.6 Making predictions using the prediction tool

With the parameters of the build up model stored in a .xml file and the earlier determined SCMs, also stored in a .xml file, we can now make predictions with the prediction tool. Details on the working of the prediction tool can be found in Appendix C.

5 Experimental setup and results

5.1 Experimental setup

5.1.1 Used Software

For synthesizing the VHDL code the application Xilinx ISE: Release version 9i, Application version J.30 has been used. This application offers to synthesize VHDL code for different FPGA series. In this experiment the VHDL code is synthesized for two different FPGA series. The Virtex-II Pro, device type XC2VP30, and the Virtex-4, device type XC4VFX60. The statistical estimation model is produced with the R 2.7.0 programming environment

5.1.2 Used hardware

The synthesizing of the VHDL code with Xilinx ISE was executed on a Sony Vaio laptop. The laptop specifications are listed below.

- **Processor:** Intel Core 2 Duo-processor T5500 running at 1.66 Ghz
- **Memory:** 2x1024 Mb of DDR2 SDRAM
- **Harddisk:** SATA 120 Gb, 5400 rpm
- **Operating System**: Windows Vista Home Premium

5.2 Experimental results

The MOLEN platform is currently designed to work with the Xilinx Virtex-II Pro platform. As technology strides on, the upcoming versions will work with the newer Virtex-4 platform. To give our research relevance now and in the future, interconnect models for both architectures are included in this report.

5.2.1 Prediction Model for the Virtex-II Pro design

5.2.1.1 Ordinary Least Squares Model

The results for the linear model were obtained using the `regsubsets` function in the R `leaps` library. The algorithm tries to find the best combinations for a predefined number of parameters from 1 to 9. The working of these algorithms are thoroughly explained in [40]. The significance results for the Virtex-II Pro are plotted in Table 5.3. This provides no information on how well these predictors actually perform in the model. To check whether the extra parameters actually help to improve the model instead of just adding

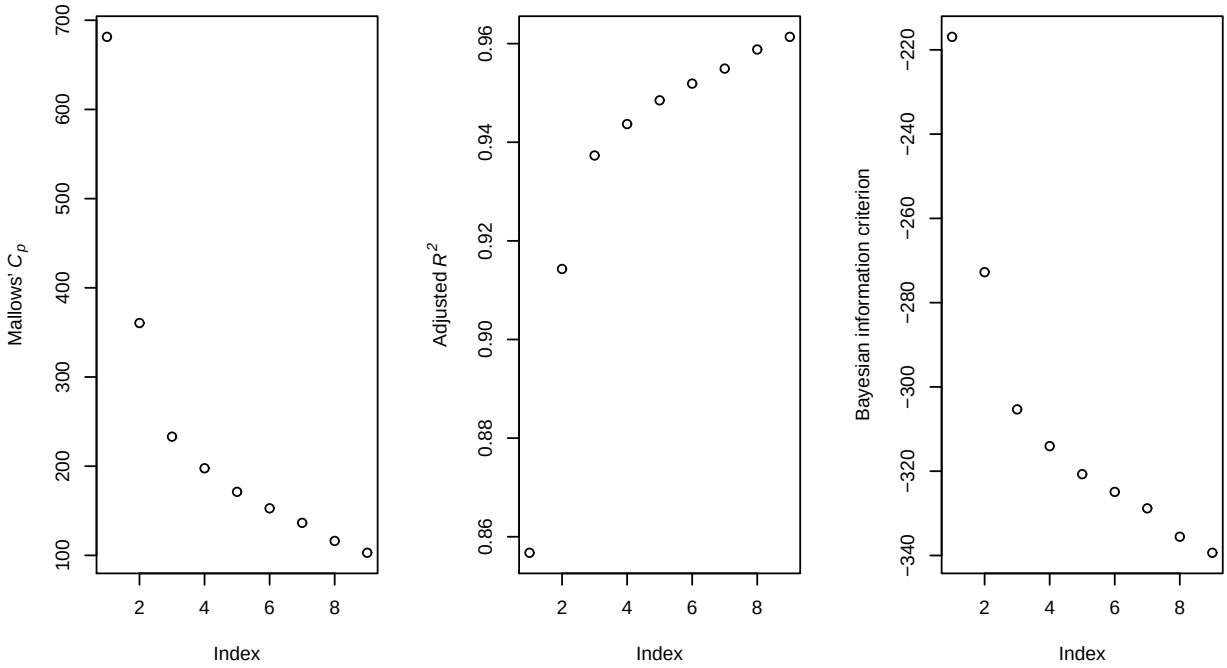


Figure 5.1: Three criterion statistics for the number of predictors to include in the OLSR model,Virtex-II Pro

noise and provoke overfitting, some useful test statistics, mentioned in Chapter 4, are used. Figure 5.1 gives some insight in how many model parameters to use based on these statistics. The plots indicate that the full 9 predictors calculated in `regsubsets` must be used. To get this information we actually need to make the fit with the given parameters, which gives us the following results, which already excludes some parameters who turn out to have no good significance levels when actually fitted:

The t value is the number of times the standard error fits into the estimate, the `Pr(>t|)` is the chance that the predictor does not explain any relation between response and predictor. In this case, we will leave the intercept in because of the physical nature of the DWARV compiler which always generates some amount of wiring even if there is no program attached.

To improve results, two data outliers have been removed from the regression set in order to make a better fit. In Figure 5.2 no values show extreme deviance from the model. The problematic units fit poorly in all four plots for the linear model. Note that with the exclusion of two non conforming units causes the favorable subset selection to change with some two to three predictors, depending on the number of prediction parameters.

Table 5.1: R results for the linear fit of the OLSR model, Virtex-II Pro

Metric	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-3223.474	2226.267	-1.448	0.15053
AICC	1704.188	749.096	2.275	0.02488
BBcurMaxExpression	3.253	1.390	2.340	0.02112
BBnumBlocks	135.272	52.231	2.590	0.01092
BBtotalExpressions	3.225	1.369	2.356	0.02027
BINMultiplications	-632.168	203.621	-3.105	0.00243
BINMultiplyBits	29.594	4.955	5.973	3.03e-08
Oviedo.DU.pairs	15.584	1.749	8.910	1.39e-14

The statistics of this fit are shown in Table 5.2. An average error of 47 % is not all that bad, but the next section shows that the partial least squares method will perform a bit better. Figure 5.3 plots the actual number of nets against the predicted number of nets. The OLSR modelling technique tries to reduce the total amount of error, not taking in account relative errors, which could be a good subject for further research. Figure 5.3 gives a graphical notion of how well the fitted values compare to the observed values, i.e. the closer a point is to the diagonal line, the better it fits in the model. Table 5.2 gives some of the basic statistics of the OLSR model. The error percentage is calculated by:

$$err. = \frac{\overline{(|y_{i,pred} - y_{i,mes}|)}}{y_{i,mes}} \tag{5.1}$$

This gives a general view of the relative overall fitting error of the model. RMSD and bootstrap[1] intervals are explained in Chapter 4. The RMSD values are useful to compare different models, they contain no information when used without context.

Table 5.2: Statistics of the OLSR model, Virtex-II Pro

No. of components	7
Error percentage	46.7 %
Root mean square deviation	4326.362
Cross-validation RMSD	5714.076
Bootstrapped 95 % BC_α conf. interval	3254 – 5693
Bootstrapped conf. interval length	2439

[1] All bootstrap simulations are based on 10000 bootstrapped samples

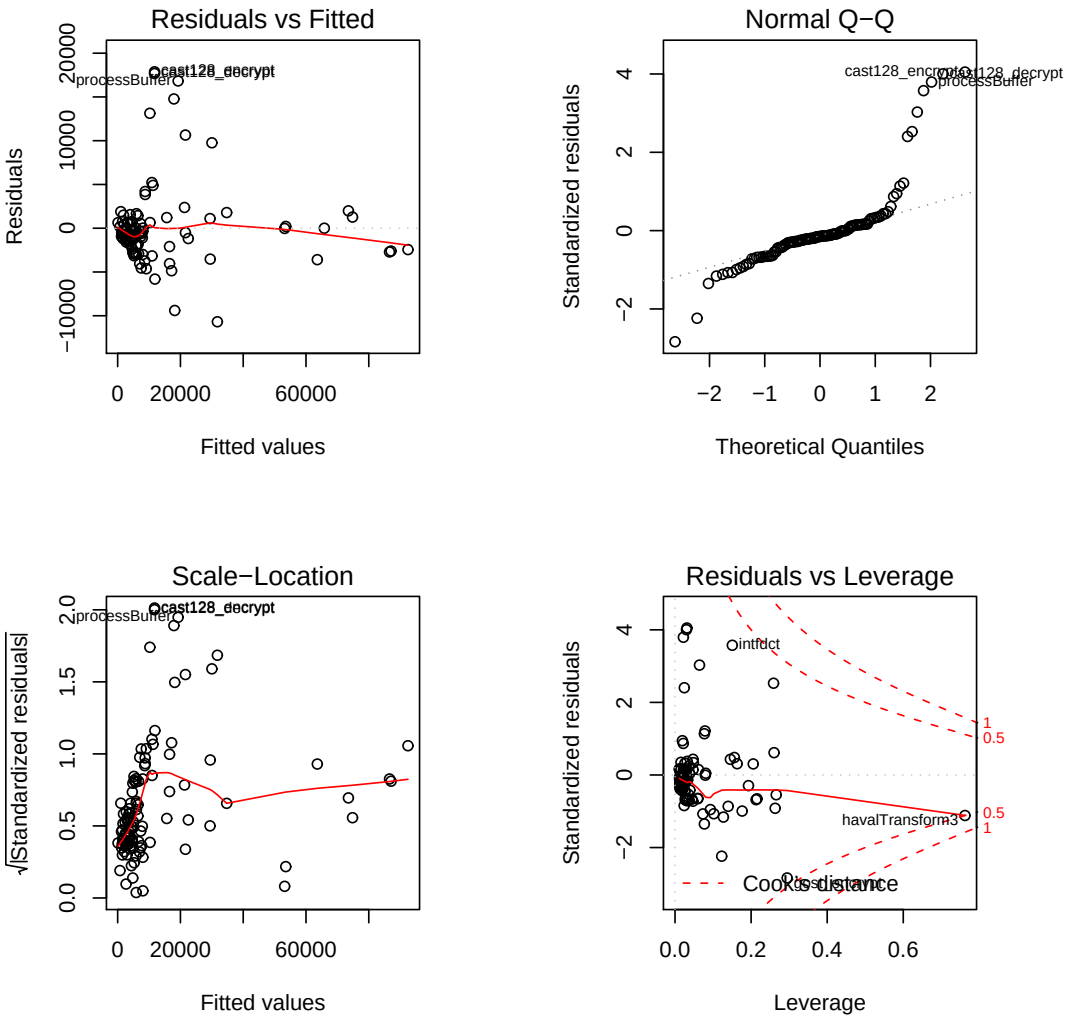


Figure 5.2: Standard R test statistics plots for the OLSR model, Virtex-II Pro

5.2.1.2 Partial Least Squares

Partial least squares regression is a somewhat different approach to the fitting problem then the Ordinary Least squares, as it is based on so called principal components. Principal components are predictors who are selected to be the most suitable predictors for a fitting set. They are found by covariance analysis . PLS regression in R can be done with the `pls` library. The `plsr` function natively supports cross-validation, which is the only good way to compare the PLS and the OLS. Figure 5.4 shows the RMSD for models with different validation methods (no validation, leave one out validation and multiple

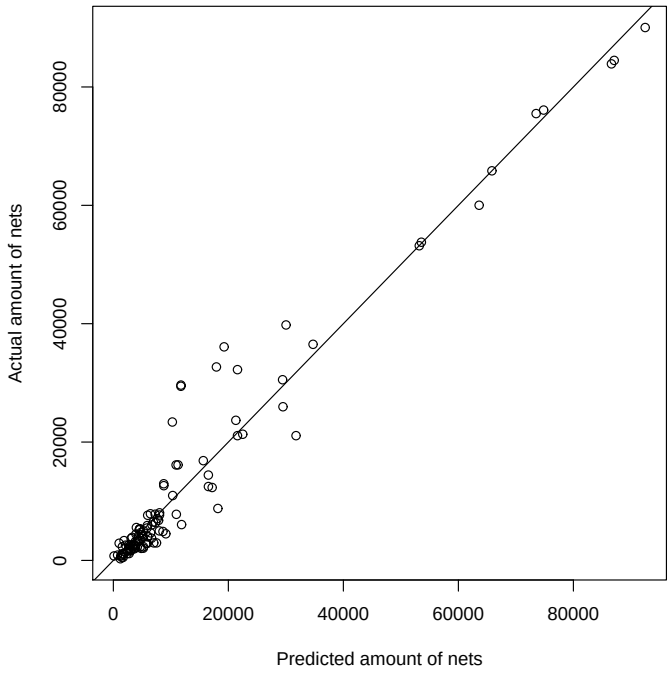


Figure 5.3: Measured values plotted against the predicted values of the OLSR model, Virtex-II Pro

K cross validation).

Again some extreme values are eliminated from the model to make a better fitting. It is interesting that the first plot in Figure 5.4 shows lower values for higher component numbers, since the lack of predictability testing makes it more likely that overfitting will occur, since the predictors will adept to the data. This is the mayor advantage over OLS, where models are not generated with testing for predictability in mind. Also the number of components needed for a good model is lower for, thereby including less noise into the model.

The statistics of the PLSR model are shown in table 5.4. The PLSR model gives two indications on what amount of parameters to use. Figure 5.4 hints the use of only four predictors, while Figure 5.5 indicates to use nine predictors. This is probably due to the fact that error percentages allow easier over fitting, and that the RMSD more heavily penalizes extreme outliers because of its mean square nature.

Based on the data from table 5.4, it is hard to say which model performs best. The model with 9 parameters clearly performs better on all fronts except on the cross validation, which gives the important quantification of predictability. This problem could be resolved by bootstrapping cross validation (see [38] chapter 17), more on this is in the recommendations section in chapter 6. The error percentage of the nine predictor

Table 5.3: Best predictor set for Virtex-II Pro OLS given the number of parameters

Metric	1	2	3	4	5	6	7	8	9
AICC								×	
BBavgExpPerBlock							×	×	×
BBavgExpPerStatement					×	×			
BBcurMaxExpression							×	×	×
BBmaxExpPerStatement						×			
BBnumBlocks								×	
BBtotalExpressions								×	
BINMultiplications								×	×
BINMultiplyBits		×	×	×	×	×	×	×	×
Loads					×	×			
Maximum.Nesting.Depth									×
Maximum.Path.Length							×		×
Oviedo.DU.pairs			×	×	×	×	×	×	
Statements							×		×
UNYBitNot									×
nOperands	×	×							
nOperator			×	×					
nUOperands				×	×	×	×		×

Table 5.4: Statistics of the PLSR model, Virtex-II Pro

No. of components	4	9
Error percentage	60.3 %	31.6 %
Root mean square deviation	4726	4157
Cross-validation RMSD	5593	6247
Bootstrapped 95 % BC_α conf. interval	$3722 - 6037$	$3358 - 5255$
Bootstrapped conf. interval length	2325	1897

model is clearly better, but is performs worse on predictability. The bootstrapped 95 % interval for the errors also looks better for the nine pridictor model, since the upper bound of the confidence interval is better.

The results for the PLS regression methods, such as the used empirical transformations and the coefficients can be found in the appendix. Figure 5.7 shows the difference in fitting between the PLS and the OLS model, whereas the OLSR data is represented

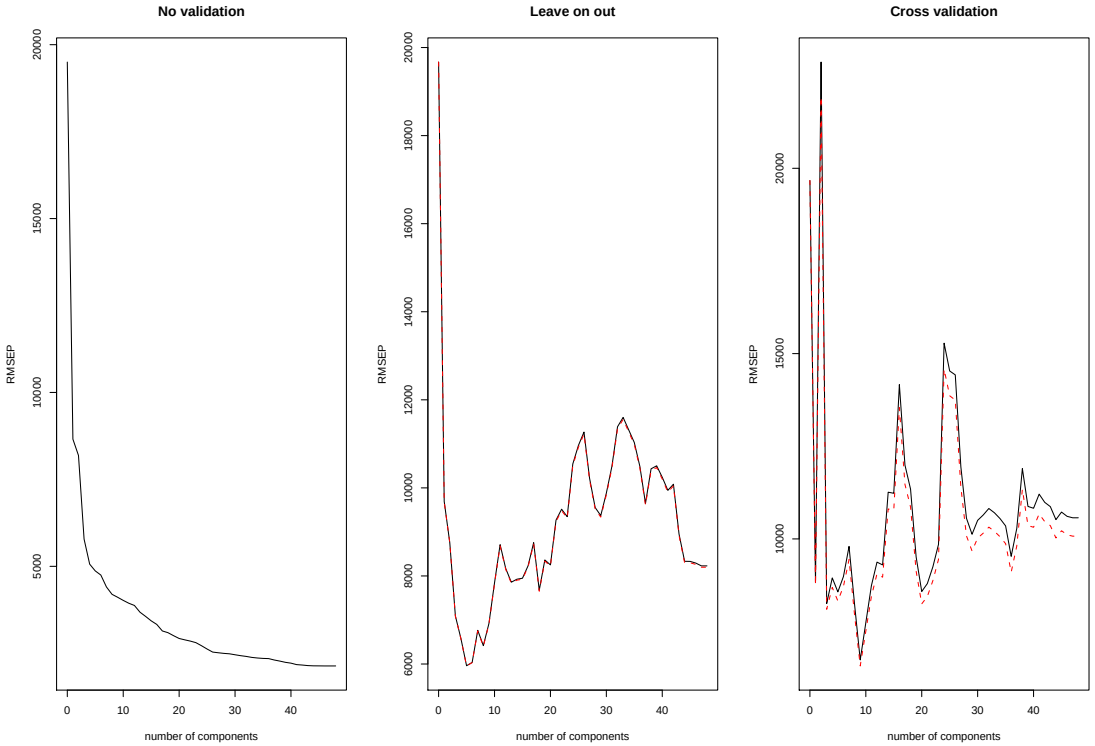


Figure 5.4: RSMD for number of components in model for different validation methods, Virtex-II Pro

by the ∘ and the PLSR data by the ×.

5.2.1.3 Data transformation

To build a linear model, one needs in some way linear relations between the response **y** and the predictors **X**. When relations are not linear, they can be made by applying a function to the data to fit better. We have chosen to use polynomial transforms on the data sets, and only when there was a visible kind of relation between predictor and response.

The data transforms result in a prediction quality which is, as expected, better than the quality of the non-transformed model. Figure 5.6 visualizes the difference in predicting quality of the transformed and the non-transformed data set, where the RMSD is used as test statistic. All the calculations in the PLSR section were done using the transformed data.

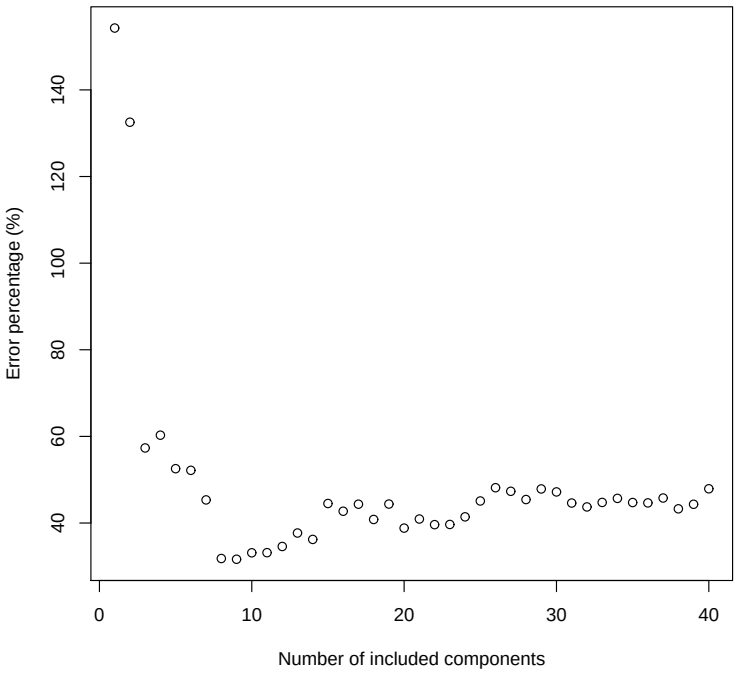


Figure 5.5: Relative error of the PLSR model for different number of prediction parameters, Virtex-II Pro

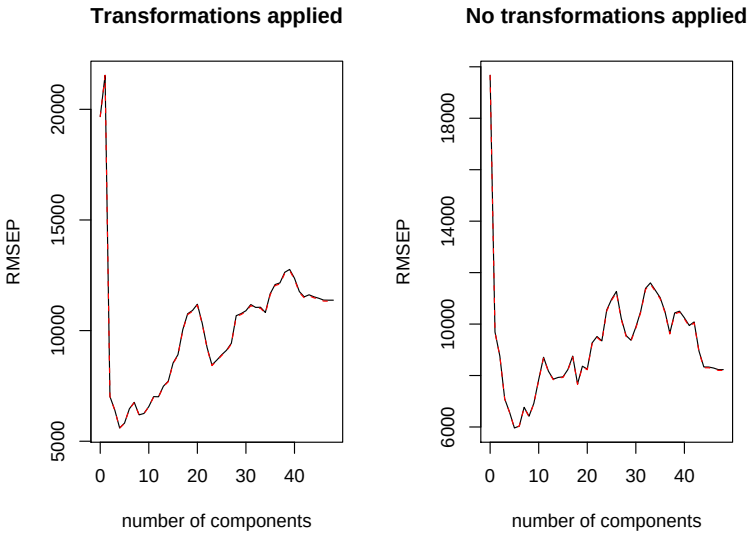


Figure 5.6: RMSD for transformed and non-transformed predictors, Virtex-II Pro

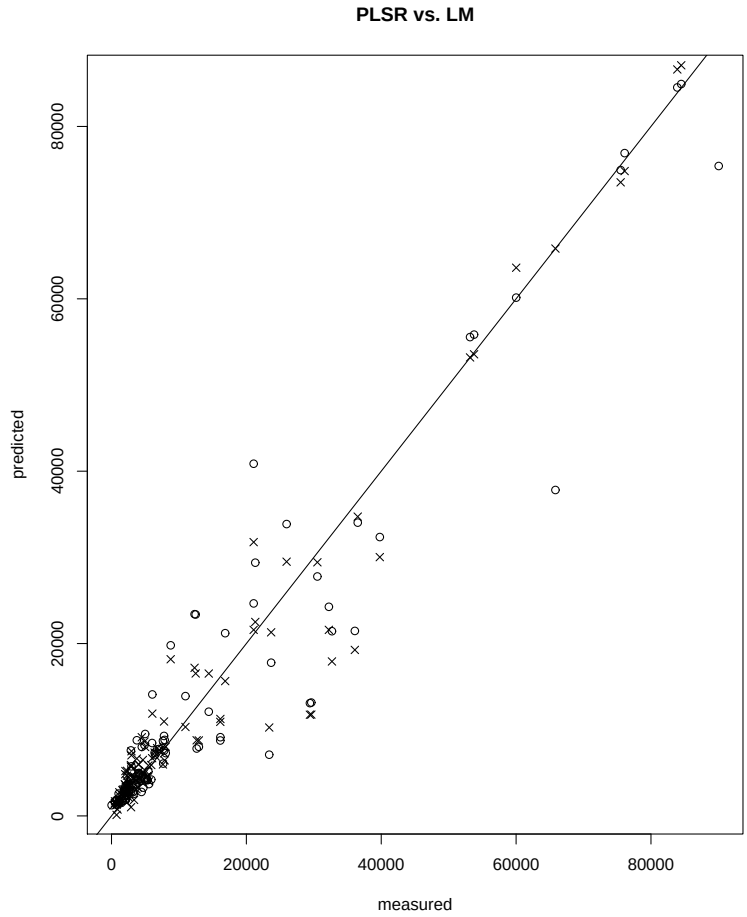


Figure 5.7: The OLSR model vs. the PLSR model when fitted,Virtex-II Pro

5.2.2 Prediction Model for the Virtex-4 design

To cope with the upcoming Xilinx Virtex-4 Architecture for the MOLEN Platform fits for this architecture have also been made. Since the methodology is the same, this section omits most of the comments in the previous section.

When fitted with R, the transformed model for the Virtex-4 OLS gives the following result:

The results of the linear fit, with transformed, i.e. linearized data, data is shown in table 5.5 The criterion plots for the transformed data with removed outliers are in Figure 5.8. The main statistics of the transformed model without outliers are in table 5.6. Table 5.7 gives an indication of which parameters to include in the model for a given number of model predictors. A performance indication of the linear model is given in Figure 5.9.

Due to the architectural differences separated models for the Virtex-II Pro and the Virtex-4 have to be developed. The number of nets differ considerably in the two different

Table 5.5: R results for the linear fit of the OLSR model, Virtex-4

Metric	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-1489.0434	531.3429	-2.802	0.00604
BBavgExpPerStatement	223.5130	26.0995	8.564	9.99e-14
BBcurMaxStatement	-50.9074	6.0973	-8.349	2.98e-13
BBtotalStatements	24.1130	3.7276	6.469	3.20e-09
BINMultiplyBits	3.7384	1.5406	2.427	0.01695
Oviedo.DU.pairs	7.6370	1.3582	5.623	1.56e-07
PlusMinus	15.2837	0.9212	16.592	$<$ 2e-16
Statements	62.3702	5.5913	11.155	$<$ 2e-16
Tai.DU.pairs	-12.3500	1.8704	-6.603	1.69e-09
VSArgVarCount	631.7507	188.1093	3.358	0.00109

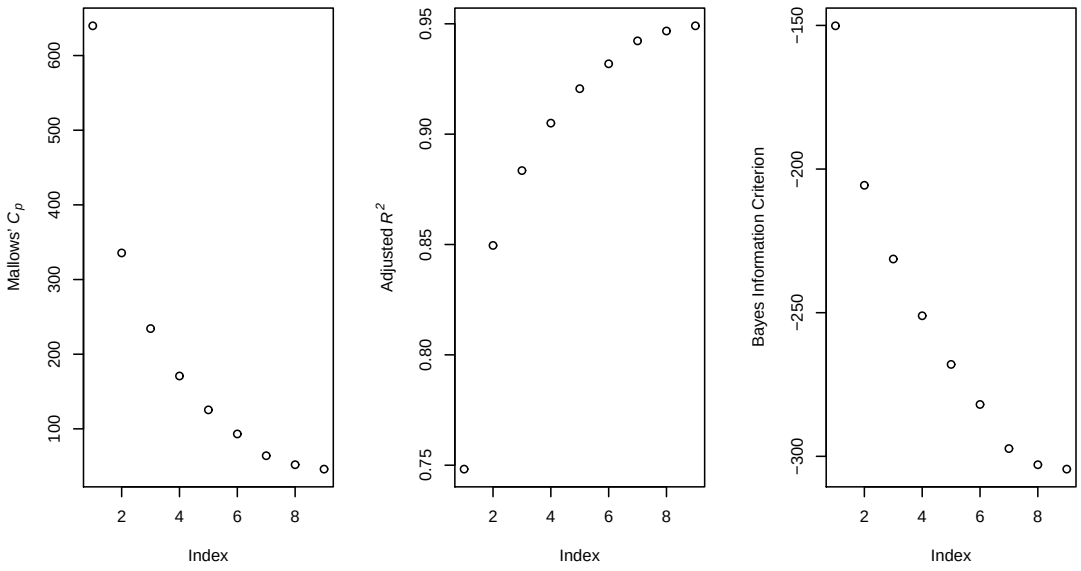


Figure 5.8: Three criterion statistics for the number of predictors to include in the OLSR model,Virtex-4

architectures, despite that the VHDL code is the same. This is most likely to be caused by different futures of the FPGA's, where the number of multiplexers, memory cells and physical layout can differ sub sequentially.

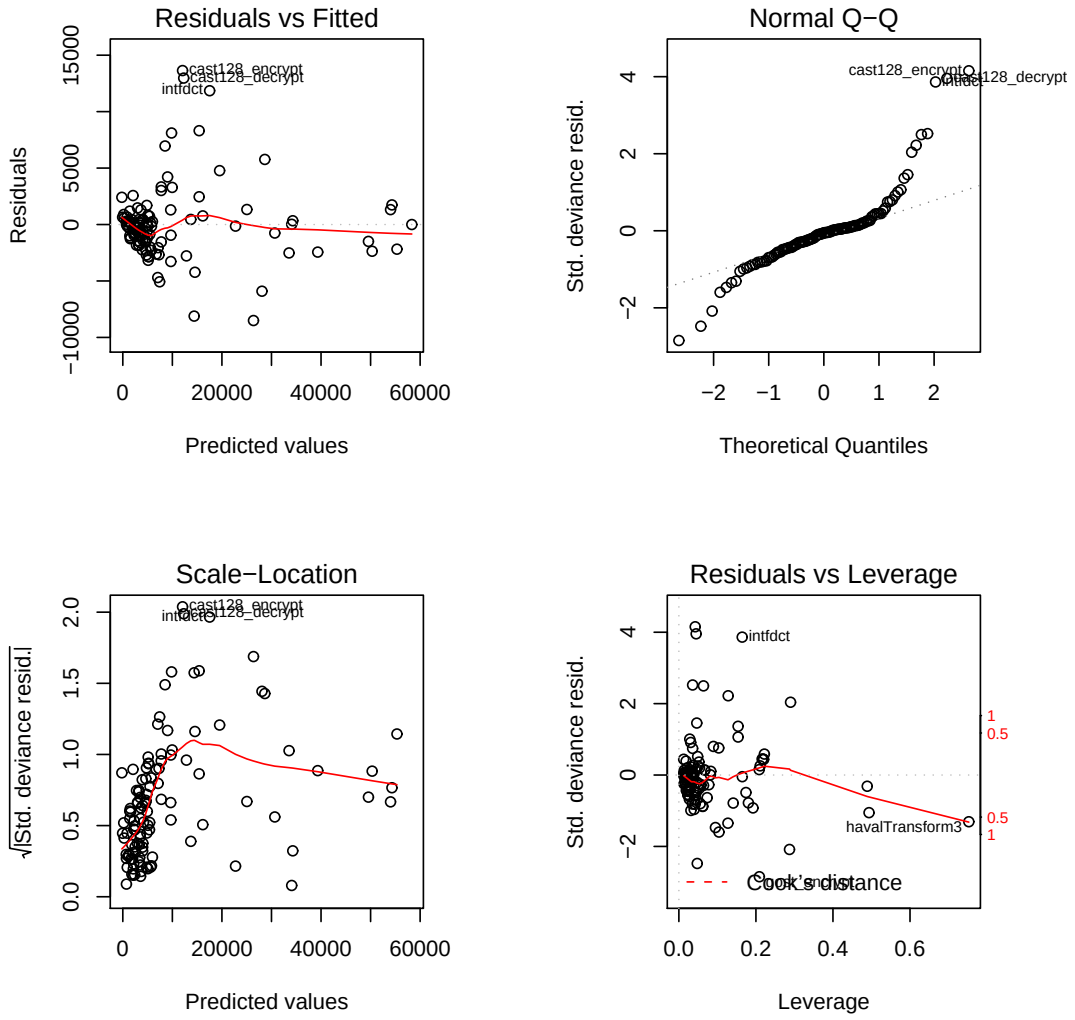


Figure 5.9: Some standard R test statistics for the linear model, Virtex-4

Table 5.6: Statistics of the transformed OLSR model, Virtex-4

No. of components	9
Error percentage	36.02 %
Root mean square deviation	2720.3
Cross-validation RMSD	3465.3
Bootstrapped 95 % BC_α conf. interval	2041 − 3788
Bootstrapped conf. interval length	1747

Table 5.7: Best predictor set for OLSR given the number of parameters, Virtex-4

Metric	1	2	3	4	5	6	7	8	9
BBavgExpPerStatement					×	×	×	×	×
BBcurMaxStatement						×	×	×	×
BBtotalExpressions	×								
BBtotalstatements								×	×
BINMultiplyBits									×
BINShift			×		×				
Basili.Hutchens			×				×		
Oviedo.DU.pairs				×		×	×	×	×
PlusMinus		×	×		×	×	×	×	×
Statements				×		×	×	×	×
Stores		×			×	×			
Tai.DU.pairs							×	×	×
UNYBitNot				×					
VSArgVarCount								×	×
nOperator				×					
nOperands					×				

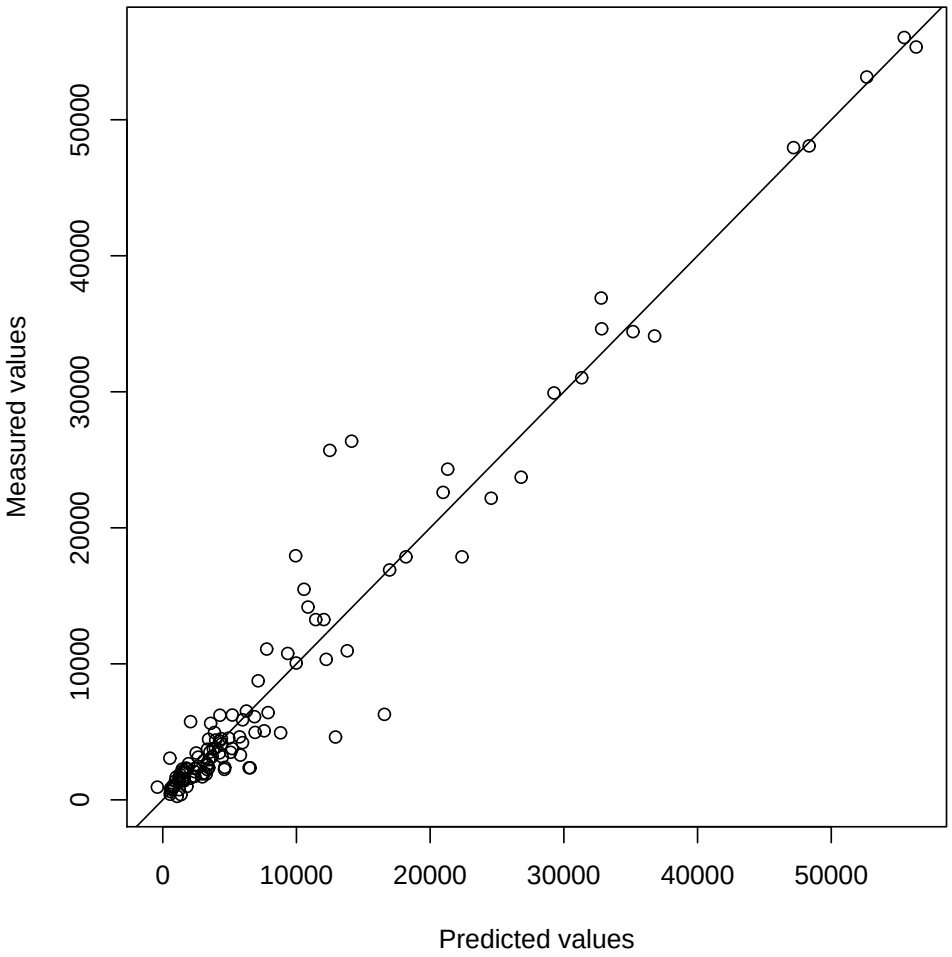


Figure 5.10: Fitted values vs. the measured values of the OLSR model, Virtex-4

Conclusions and recommendations

6

The results section shows that the model we have built can make predictions with acceptable error. These results can help to make early decisions in the entire design from C*-code to FPGA layouts regarding the placement of the design in hardware.*

6.1 Conclusions

As reconfigurable computing is likely to become the next big revolution in the field of computing, more and more research is done towards all facets of the problem. The department of Computer Engineering at the TU Delft has several research programs related to reconfigurable computing research. Our thesis is part of the software traject for the TU Delft reconfigurable computing platform, the Delft WorkBench.

A chain of tools is required to change C-code into a design that can be programmed onto an FPGA and work in a coherent way with a general purpose processor. The tool made for this thesis handles an early estimation of the number of interconnects that is needed in the final FPGA design layout. To make these predictions, statistical models in the R programming environment are built. These models give no exact answers, but rather approximations with confidence intervals for the errors. Our model is built for the amount of nets. It shows that there is a linear relation between the software complexity metrics and the interconnect.

The results obtained from our research give some decent approximation of about 35%. Although not very accurate, this result can be very helpful in the early stages of design to decide whether a design will fit in an FPGA architecture or not. Due to the way the model is built, the results for the larger kernels are generally more reliable than the results for small kernels (see Further recommendations on the model bullet number 1). Since research on multiple kernels on one design was dropped in an early phase of the project, this is no problem for the scope of our project.

For the building of the model, two types of modeling have been studied. The first is the ordinary least squares method (OLSR), with first users such as Gauss, the second is the partial least squares (PLSR), a method which is around for about only thirty years. The OLSR method is well suited to explore the relation between predictors and responses, but it does not evaluate the models prediction quality. Therefore, we have also used the PLSR model, which has built in support for cross-validation.

The only way to compare the two models is by their cross-validation. The predicting quality for the Virtex-II Pro design is better with the PLSR model than with the OLSR technique, which is no surprise since the former was designed to optimize predictability.

The models for the Virtex-4 are harder to compare. An unresolved problem caused the RMSD of the cross-validated model not to reach sufficient values, so no motivated models could be selected from these values. Since the linear model works comparably

well, also on predictability, compared with the Virtex-II Pro data, this model can be recommended for use in the developed tool.

6.2 Further recommendations on the model

The models we viewed in this thesis were standard models using Partial or Ordinary Least Squares. Although results were fairly acceptable with an average error of 35 %, there are ways to improve the models. The presented models in this thesis are known to have some flaws. Possible solutions for these flaws are stated below.

- The least squares algorithm tries to minimize the total squared errors. This is a method which is widely used. In the case of the interconnection estimation, it is questionable if this is the right way to fit a model. Misfits for kernels with small interconnections are absolutely equally penalized as misfits for kernels with large interconnections. This diminishes the usefulness of the relative prediction error. The model will generate fits which have big relative errors in small kernels, ranging easily towards 50% - 100%, where large kernels will have errors of around 10%. A way to deal with this is the introduction of heteroscedastic modeling, i.e. models where the variance is a function of some parameter, in our case the number of nets. An example of how to build such models can be found in [41]. This should give a more balanced view on the error of the model, since a fit to a heteroscedastic model will try to keep the error percentage independent of the amount of nets.

- The goal of the thesis is to create a tool set which can process metrics data from C-code and predict, based on those metrics, the number of nets that the final design will contain. The algorithm for OLS tries to fit the predictors best suited to the data. The drawback is that this technique gives no information how well the model can actually predict a value. Cross validation is used to verify the predictability. In our models cross validation, especially for the OLS models, is only briefly used. The correct way find the best model is to use bootstrapped cross validation (see [38] chapter 17) for all possible models. A model has the best prediction quality if the model that suits the desired task the best

- As with all sample sets, the kernels that were used the fit the model on contains outliers who do not seem to fit in the model. Since the number of nets is a direct function from C-code [1] other metrics, or more detailed metrics could build a model with a much improved quality, and give a better explanation and approximation to outliers, to make the model more accurate. The presented model can make good predictions in most cases, but in some occasions the model is wrong by some 30.000 nets. If the prediction tool is used to check beforehand if a design will fit on an FPGA, results could be disappointing.

[1] The C-code is mapped to an FPGA design with a number of nets in a fixed a consistent matter (i.e. the mapping results are always the same), so a function which describes this relation *must* exist. This function would by far be to complicated to find, so statistical modeling is used instead

Bibliography

[1] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commmun. ACM 21 (1978) no.8*, pages 613–641, 1978.

[2] Wikipedia. Von neumann architecture. *http://en.wikipedia.org/wiki/Von_Neumann_architecture*, unspecified.

[3] The Delft Workbench. The delft workbench. *http://ce.et.tudelft.nl/DWB/*, unspecified.

[4] R.J. Meeuws; Y.D. Yankova; K.L.M. Bertels; G.N. Gaydadjiev; S. Vassiliadis. A quantitative prediction model for hardware/software partitioning. In *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, pages 735–739, August 2007.

[5] Xilinx. Development system reference guide. *http://www.xilinx.com/support/sw_manuals/xilinx92/download/dev.zip*, unspecified.

[6] S. McPeak. Elkhound: A glr parser generator and elsa: An elkhound-based c++ parser. *http://www.cs.berkeley.edu/ smcpeak/elkhound/*, unspecified.

[7] Y.D. Yankova; G.K. Kuzmanov; K.L.M. Bertels; G. N. Gaydadjiev; Y. Lu; S. Vassiliadis. Dwarv: Delft workbench automated reconfigurable vhdl generator. In *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, pages 697–701, August 2007.

[8] S. Vassiliadis; S. Wong; G.N. Gaydadjiev; K.L.M. Bertels; G.K. Kuzmanov; E. Moscu Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, pages 1363–1375, November 2004.

[9] G.K. Kuzmanov; G.N. Gaydadjiev; S. Vassiliadis. The molen media processor: Design and evaluation. In *Proceedings of the International Workshop on Application Specific Processors*, pages 26–33, September 2005.

[10] E. Moscu Panainte; K.L.M. Bertels; S. Vassiliadis. The molen compiler for reconfigurable processors. *ACM Transactions in Embedded Computing Systems (TECS)*, page 18, February 2007.

[11] MOLEN. Molen platform. *http://ce.et.tudelft.nl/MOLEN/*, unspecified.

[12] K. Compton; S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Survey*, pages 171–210, June 2002.

[13] L. Zhiyuan. *Configuration Management Techniques for Reconfigurable Computing.* PhD thesis, Northwestern University, June 2002. A dissertation submitted to the graduate school in partial filfullment of the requirements.

[14] D. de Leeuw Duarte. Reconfigurable computing: A survey of architectures and synthesis tools. October 2005.

[15] Xilinx. Xilinx. *http://www.xilinx.com/company/about.htm*, unspecified.

[16] Altera. Altera. *http://www.altera.com*, unspecified.

[17] Atmel. Atmel. *http://www.altmel.com*, unspecified.

[18] D. Buell; T. El-Ghazawi; K. Gai; V. Kindratenko. High-performance reconfigurable computing. *IEEE Computer society, Computer: innovative technology for computer proffesionals, vol. 40 nr 3*, page 25, March 2007.

[19] B.K. Fawcett and J. Watson. Reconfigurable processing with field programmable gate arrays. In *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on 19-21 Aug.*, pages 293–302, August 1996.

[20] Mehdi Baradaran Tahoori. Application-dependent testing of fpga interconnects. *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03)*, page 403, 2003.

[21] C. Stroud; S. Wijesuriya; C. Hamilton; M. Abramovici. Built-in self-test of fpga interconnect. pages 404–411, Oct. 1998.

[22] H. Schmit; V. Chandra. Fpga switch block layout and evaluation. pages 11–18, 2002.

[23] E. Moscu Panainte. The molen compiler for reconfigurable architectures. 2007.

[24] K.L.M. Bertels; S. Vassiliadis; E. Moscu Panainte; Y.D. Yankova; C. Galuzzi; R. Chaves; G.K. Kuzmanov. Developing applications for polymorphic processors: the delft workbench. page 7, January 2006.

[25] O.S. Dragomir; E. Moscu Panainte; K.L.M. Bertels. Loop parallelization for reconfigurable architectures. *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC07)*, November 2007.

[26] Z. Nawaz; O.S. Dragomir; T. Marconi; E. Moscu Panainte; K.L.M. Bertels; S. Vassiliadis. Recursive variable expansion: A loop transformation for reconfigurable systems. In *proceedings of International Conference on Field-Programmable Technology 2007*, December 2007.

[27] R.L. Russo; B.S. Landman. On a pin versus block relationship for partitions of logic graphs. *IEEE transactions on computers*, pages 1469–1479, 1971.

[28] C.A. Papachristou; H. Konuk. A linear program driven scheduling and allocation method followed by an interconnect optimization algorithm. *Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 77–83, 1991.

[29] X. Yang; E. Bozorgzadeh; M. Sarrafzedeh. Wirelength estimation based on rent exponents of partitioning and placement. *Proceedings of the 2001 international workshop on System-level interconnect prediction*, pages 25–31, 2001.

[30] M. Holzer; M. Rupp. Static estimation of execution times for hardware accelerators in system-on-chips. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 62–65, 2005.

[31] S. Balachandran; D. Bhatia. A priori wirelength and interconnect estimation based on circuit characteristics. *IEEE Transactions on computer-aided design of integrated circuits and systems, vol. 24 no. 7*, July 2005.

[32] R.J. Meeuws; K. Sigdel; Y.D. Yankova; K.L.M. Bertels. Quantitative prediction for early design space exploration in delft workbench: An outlook. In *proceedings of ProRisc 2007*, November 2007.

[33] L. Scheffer; E. Nequist. Why interconnect prediction doesn't work. *Proceedings of the 2000 international workshop on System-level interconnect prediction*, pages 139–144, 2000.

[34] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control 19*, 6:716–723, 1974.

[35] C.L. Mallows. Some comments on cp. *Technometrics*, 15:661–675, 1973.

[36] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2008. ISBN 3-900051-07-0.

[37] J.L. Faraway. *Linear Models with R*. Texts in Statistical Science Series. Chapman & Hall/CRC, 2005.

[38] B. Efron and R.J. Tibshirani. *An Introduction to the Bootstrap*, volume 57 of *Monographs on Statistics and Applied Probability*. Chapman & Hall/CRC, 1993.

[39] P.I. Good. *Resampling Methods*. Birkhäuser, 3rd edition, 2006.

[40] A. Miller. *Subset Selection in Regression Analysis*. Chapman & Hall/CRC, 2nd edition, 2002.

[41] J. S. Long; L. H. Ervin. Correcting for heteroscedasticity with heteroscedasticity consistent standard errors in the linear regression model: Small sample considerations. *American Statistician*, 54, 2000.

Appendices

Xilinx ISE Tool

A

The Xilinx ISE Tool, written in Java, is a tool that helps to automate the whole Xilinx process of synthesizing, mapping and place and routing. Provided with the VHDL files, it creates the necessary directories, copies the necessary files to each directory and runs the whole xilinx process for each VHDL file. A global overview of the Xilinx ISE Tool is depicted in Figure A.1. The source code of this tool can be found on the CD-ROM provided with this document.

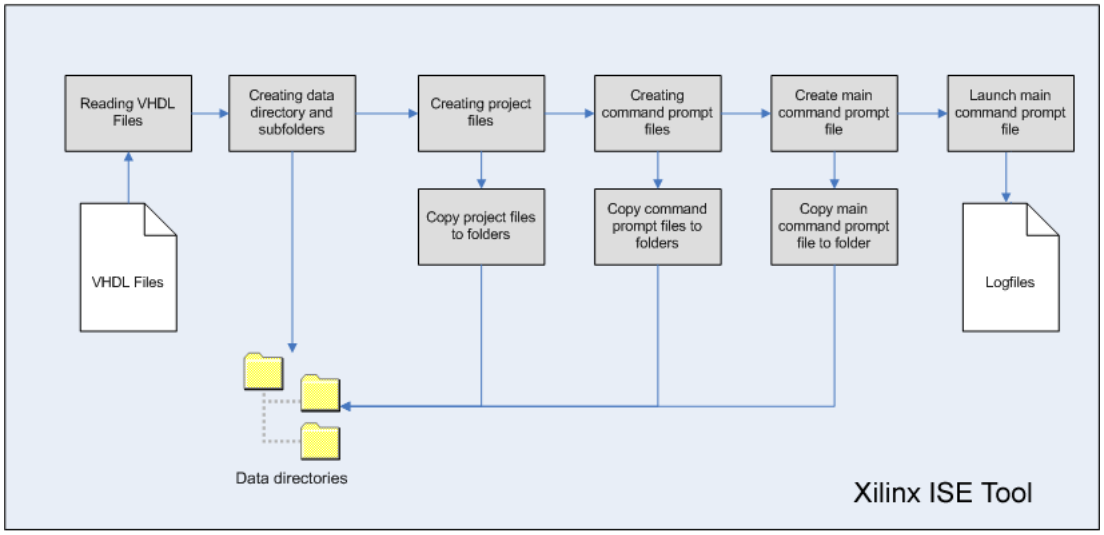


Figure A.1: Global overview of the Xilinx ISE Tool.

A.1 Work flow

First, the Xilinx ISE Tool reads in all the VHDL files and creates a data directory with for each VHDL file a separate folder and the necessary sub folders: *script*, *log* and *vhdl*. When finished creating all the directories it copies each VHDL file to its own directory into the sub folder *vhdl*. For each VHDL file two project files are created and copied into the sub folder *script*. These project files are required for the Xilinx process. Then it creates, for each VHDL file, six command prompt files, that will run each part of the Xilinx process. These files are copied into the folder that belongs to the VHDL file. Finally, it creates and launches a main command prompt file. This command prompt file will run, for every VHDL file, each of the six command prompt files. During each part of the process, log files, containing specific data, are created and placed into the sub

folder *log*. These log files will be used by the Logfiles Tool, discussed in Appendix B, to extract the necessarry data required to build up a statistical estimation model. Below, examples of the project files and command prompt files are shown.

Listing A.1: projectfile.prj

```
#example of one of the kernels: adpcm_coder

set -xsthdpdir adpcm_coder/xst
run
-ifn adpcm_coder/script/adpcm_coder_work.prj
-ifmt VHDL
-ofn adpcm_coder/adpcm_coder_CCU.ngc
-ofmt NGC
-p xc2vp30-7-ff896
-opt_mode Speed
-opt_level 1
-top CCU
-iobuf YES
```

Listing A.2: projectfilework.prj

```
#example of one of the kernels: adpcm_coder

vhdl work ../vhdl/adpcm_coder.vhd
vhdl work ../vhdl/synth_param_pckg.vhd
```

Listing A.3: synthesize.cmd

```
#example of one of the kernels: adpcm_coder

xst -intstyle silent -ifn adpcm_coder/script/adpcm_coder.prj -ofn
    adpcm_coder/logs/synth_adpcm_coder.log
```

Listing A.4: ngdbuild.cmd

```
#example of one of the kernels: adpcm_coder

ngdbuild -active CCU adpcm_coder/adpcm_coder_CCU.ngc adpcm_coder/
    adpcm_coder_CCU.ngd > adpcm_coder/logs/ngdbuild.log
```

Listing A.5: map.cmd

```
#example of one of the kernels: adpcm_coder

map /adpcm_coder_CCU.ngd adpcm_coder/adpcm_coder_CCU_map.pcf -o adpcm_coder
    /adpcm_coder_CCU_map.ncd  > adpcm_coder/logs/map.log
```

Listing A.6: par.cmd

```
#example of one of the kernels: adpcm_coder

par -w adpcm_coder/adpcm_coder_CCU_map.ncd adpcm_coder/
    adpcm_coder_CCU_routed.ncd  > adpcm_coder/logs/routed.log
```

Listing A.7: trce.cmd

```
#example of one of the kernels: adpcm_coder

trce adpcm_coder/adpcm_coder_CCU_routed.ncd  > adpcm_coder/logs/trace.log
```

Listing A.8: xdl.cmd

```
#example of one of the kernels: adpcm_coder

xdl -ncd2xdl adpcm_coder/adpcm_coder_CCU_map.ncd adpcm_coder/
    adpcm_coder_xdl.xdl  > adpcm_coder/logs/xdl.log
```

Logfiles Tool

B

The Logfiles Tool, written in Java, is a tool to extract the necessary data from the log files that were created using the Xilinx ISE Tool, see Appendix A. A global overview of the Logfile Tool is depicted in Figure B.1

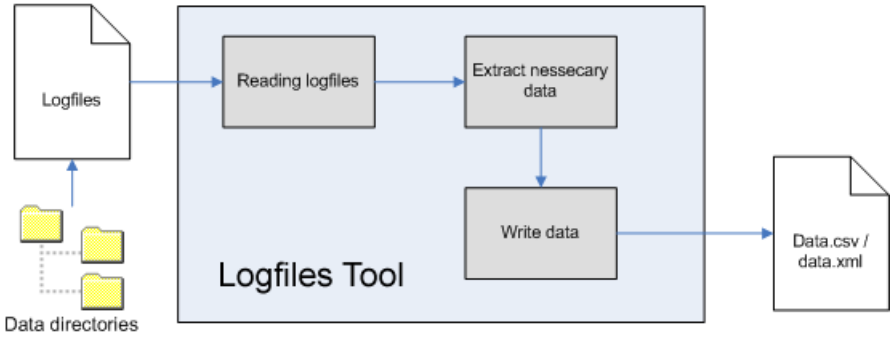


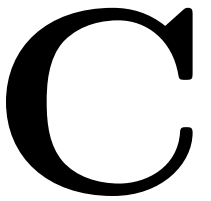
Figure B.1: Global overview of the Logfiles Tool.

B.1 Work flow

The Logfiles Tool reads in all the log files in the data directories created with the Xilinx ISE Tool. These log files contain specific information necessary for building up a statistical estimation model. With the help of regular expressions[1], the Logfiles Tool, extracts the necessary data from the log files and writes it to an output file.

[1]Regular expressions provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters

Prediction Tool



The Prediction Tool, written in Java, is a tool to predict certain aspects of a software kernel and based on this predictions calculate the utilization for a specified FPGA series. To predict these aspects and to calculate the utilization, the Prediction Tool is provided with three files in XML-format. The first file contains a list of software complexity metrics. The second file contains a list of models that correspond with a certain aspect and finally, the third file contains a list of FPGA descriptions and their resources. Further on in this report they will be referred as *measurments.xml*, *model.xml* and *fpga.xml* respectively. A global overview of the Prediction Tool is depicted in Figure C.1. The source code of this tool can be found on the CD-ROM provided with this document.

C.1 Work flow

The Prediction Tool should be executed through the command line. The user has two options for executing the tool. These options are listed below.

1. *java -jar Prediction_tool.jar help*
2. *java -jar Prediction_tool.jar run -command1 argument1 -command2 argument2...*

Option one will read in the readme.txt, provided with the tool, and prints the context in the command line. Option two will start calculating the predictions and based on the predictions calculate the utilization for the specified FPGA series. The user has a number of command and argument options available. Three command options are obligatory, for the tool to run, and the others are optional. In table C.1 an overview of the command and argument options is depicted.

Table C.1: Overview of the command and arguments options

Command	**Argument**	
`-model`	path to your model.xml file	obligatory
`-measurement`	path to your measurement.xml file	obligatory
`-output`	path where you want the output.xml file to be placed	obligatory
`-device`	device type of the FPGA series you are using	optional
`-xsdmodel`	path to your own xsd schema file for model.xml	optional
`-xsdmeasurement`	path to your own xsd schema file for measurement.xml	optional
`-negval`	specifies if negative predictions must be set to zero, the standard value is `off`, it can be changed to `on`	optional

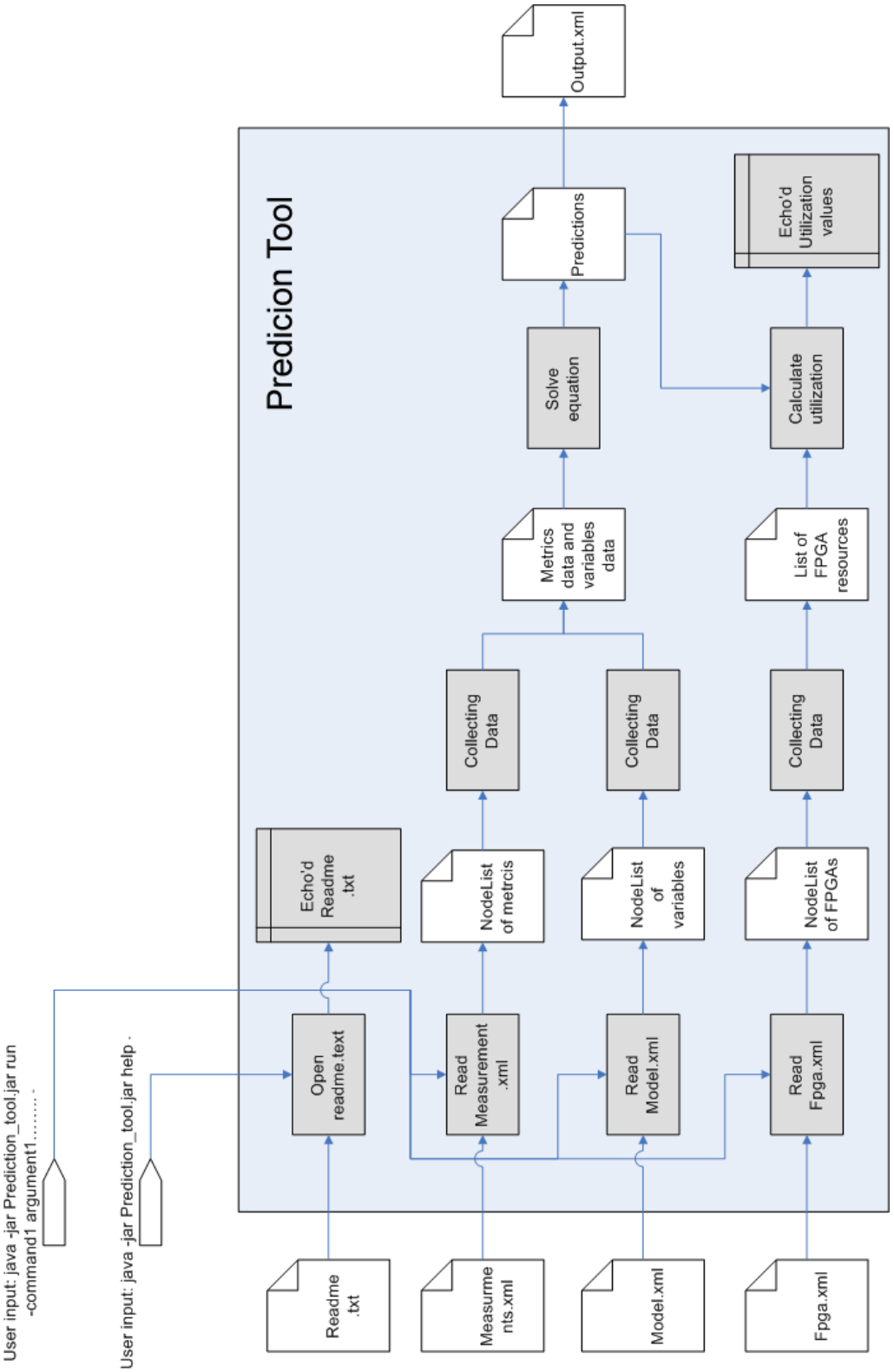


Figure C.1: Global overview of the Prediction Tool.

When the user has specified the obligatory commands and executes the Prediction Tool, it reads in the *model.xml* and the *measurements.xml* file. The *model.xml* file exists of a list of models that correspond with a certain aspect. A model also has a `name` and a `target` attribute that corresponds with a resource of an FPGA and a device type of one of the FPGAs described in *fpga.xml* respectively. Each model can exist of an `intercept` with a `numeric value` and a list with n-number of variables. A `variable` has a `name` that should correspond with one of the specific names of a metric. Each variable can exist of a `transformation`, with a `string value`, and a `coefficient` with a `numeric value`. An example of this file is listed in Listening C.1.

Listing C.1: model.xml

```
<model name="modelName" target="deviceType">
  <intercept>numeric value</intercept>
  <variables>
    <variable name="variableName(1)">
      <transformation>string(1)</transformation>
      <coefficient>numeric value(1)</coefficient>
    </variable>
    <variable name="variableName(2)">
      <coefficient>numeric value(2)</coefficient>
    </variable>
      .
      .
      .
    <variable name="variableName(n)">
      <coefficient>numeric value(n)</coefficient>
    </variable>
  </variables>
</model>
  .
  .
  .
```

The *measurements.xml* file exists of a list with n-number of software metrics from a certain software kernel. Each `metric` has a specific `name` and a `numeric value`. An example of this file is listed in Listening C.2. More detailed versions of both of these files can be found on the CD-ROM provided with this document.

Listing C.2: measurements.xml

```
<metric name="metricName(1)">numeric value(1)</metric>
<metric name="metricName(2)">numeric value(2)</metric>
  .
  .
  .
<metric name="metricName(n)">numeric value(n)</metric>
```

Before proceeding, the Prediction Tool first checks if both, *model.xml* and *measurement.xml*, files are valid XML and meet the constraints in the default xsd schema's which are described in *default_model.xsd* and *default_measurements.xsd*. These files can be overwritten by specifying the `-xsdmodel` and `-xsdmeasurement` command. If both

files are valid and met the constraints, it collects a list of models from *model.xml* and list of metrics from *measurements.xml*. For each metric in the list of metrics it collects the specific names and numeric values. For each model in the model list it collects a list of variables and checks if each variable corresponds with one of the collected metrics. If so, it then collects, for each variable, the string values of the transformations and the numeric values of the coefficients. With all the collected data it then predicts the aspects, that correspond with the models in *model.xml*, by solving the following equation.

- *prediction = intercept + (coefficient(1) * transformation(1)(metric value(1))) + (coefficient(2) * transformation(2)(metric value(2)).....(coefficient(n) * transformation(n)(metric value(n))*

The prediction for each model is written to an output file. If the command option `-negval` is specified, negative predictions will be set to zero in the output file. If the command option `device` is specified the Prediction tool then starts with calculating the utilizations. It reads in the *fpga.xml* file. The *fpga.xml* file exists of a list with n-number of FPGAs with a `device name` and a `device type`. Each FPGA has a list with n-number of `resources`. These resources can correspond with the `name` of a model in *model.xml*. For each model, where the `target` attribute corresponds with the specified device and the `name` of the model corresponds with one of the resources of the FPGA, the Prediction Tool calculates the utilization and prints it in the command line.

C.2 Class diagram

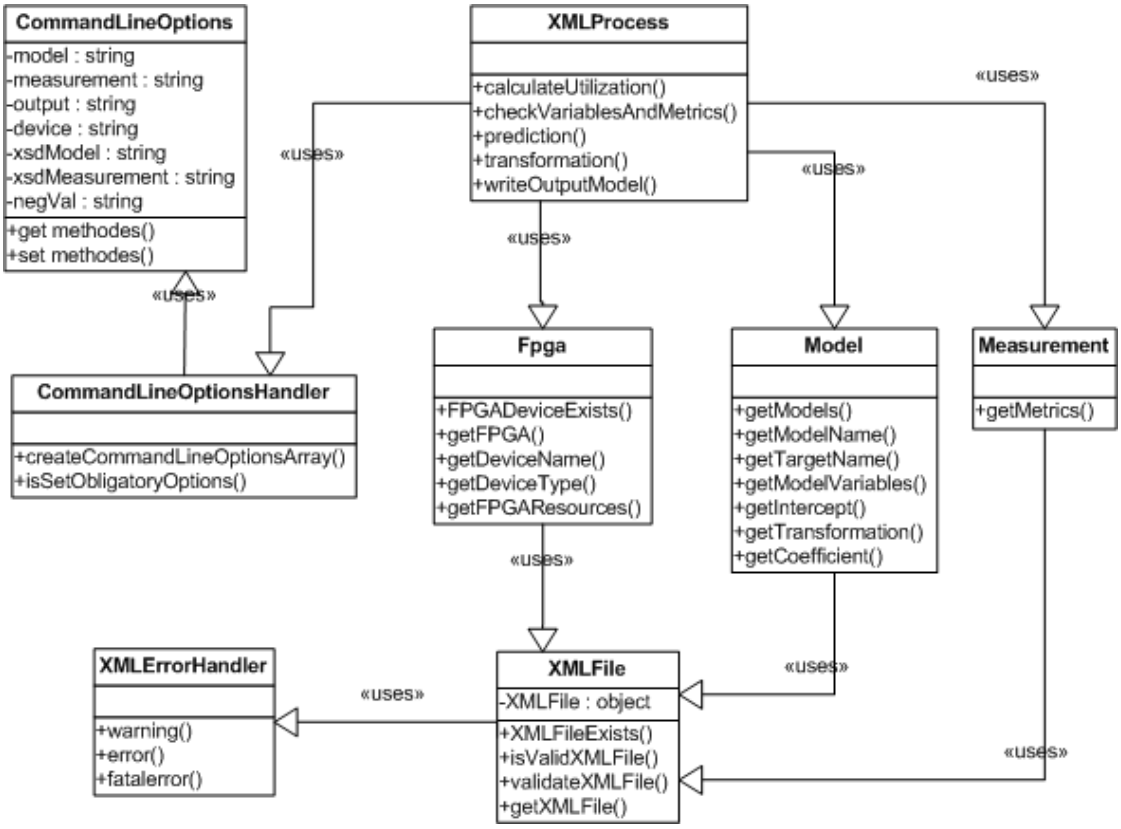


Figure C.2: Class diagram of the Prediction Tool.

Extracted data from log files

D

This appendix contains two tables with data extracted from the log files that were created during the process of synthesizing, translating, mapping and place and routing of the software kernels. The data is used to build up the statistical estimation model. The first section contains a table with data from the Virtex2P architecture and the second sections contains a table with data from the Virtex4 architecture.

D.1 Virtex2P

Kernel name	Category	Slices	Slice Flip Flops	LUTs	MULTs	Nets
adpcm_coder	Multimedia	1514	1934	2411		4656
adpcm_decoder	Multimedia	1356	1685	2035		4015
apply_butterflies	ECC	14633	16962	20000		38886
arcfour_encrypt	Cryptography	783	669	1262		2312
bdist1	Multimedia	1775	2330	2014	8	5172
bdist2	Multimedia	1414	1784	1773	5	4148
binarySearch	Other	674	849	991		2060
bitreversal1	DSP	829	886	1409		2635
bitreversal2	DSP	194	277	199		516
blowfish_decrypt	Cryptography	1026	1288	1285		2880
blowfish_encrypt	Cryptography	1008	1280	1311		2913
BubbleSort	Other	607	677	828		1835
bytesum	Mathematics	67	116	45		284
calcLCS	Other	1180	1359	1707	6	3753
cast128_decrypt	Cryptography	13898	8031	24890		29424
cast128_encrypt	Cryptography	14585	8077	25955		29630
compress_image	Compression	1142	1482	1476		3290
decompress_image	Compression	795	1137	822		2226
delta_forward	Multimedia	357	443	440		1143
delta_inverse	Multimedia	327	409	407		1156
dist1	Multimedia	5155	6221	6850		14424
dist2	Multimedia	2933	3718	3592	10	8044
divisiblebythree	Mathematics	691	934	890		1844
DotProduct	Mathematics	518	575	573	4	1559
enblf_noswap	Cryptography	821	955	1299		2576
enigma_encrypt	Cryptography	2161	2375	2858	2	5473
f	Cryptography	902	766	1327		2446
factorial	Mathematics	250	310	268	3	755
FIR	DSP	982	1054	1381	20	3700
fix_fft	DSP	2914	3168	4202	4	7791
floyd_warshall	Mathematics	1698	2048	2218	12	4992
form_component_prediction	Multimedia	6602	7878	7509	8	16146
g721_body	Multimedia	15033	12236	27181	18	38653
gcd	Other	1102	1165	1698		2662
generate_set	Mathematics	2124	2378	2171	39	6022
gosthash_compress	Cryptography	11940	13751	17467		32235
gost_decrypt	Cryptography	8616	8875	10568		21082
gost_encrypt	Cryptography	8616	8875	10568		21082
haar_predict	Multimedia	647	799	862		2098
haar_update	Multimedia	651	800	864		2099
Hamming1	ECC	166	263	183		502
Hamming2	ECC	264	377	284		753
havalTransform3	Cryptography	26598	44550	42001		90043
hw_boyer_moore_search	Other	2893	3339	3961		7670
hw_derrivative_x_y	Multimedia	2025	2065	2851	3	5347
hw_encode	ECC	1887	2489	2181		4725
hw_md2_transform	Cryptography	838	888	1304		2617
hw_mdct_bitreverse	Multimedia	2533	2496	3616	32	7881
hw_non_max_supp	Multimedia	20753	9773	36657	132	65832

Kernel name	Category	Slices	Slice Flip Flops	LUTs	MULTs	Nets
hw_ripemd128_transform	Cryptography	16029	27932	21852		53171
hw_ripemd160_transform	Cryptography	25670	45513	32672		83903
hw_ripemd256_transform	Cryptography	16289	28113	22246		53766
hw_ripemd320_transform	Cryptography	25934	45687	33135		84489
hw_sobel	Multimedia	1863	2519	1867	10	4894
hw_viterbi	ECC	8667	7902	15465		23389
idct	Multimedia	4455	4052	6507	26	12320
intersect_triangle	Mathematics	5539	3702	9738	96	16858
intfdct	Multimedia	12616	5980	24344	48	32692
intmatmult	Mathematics	2279	2892	2420	4	5557
intmatmult3x3	Mathematics	3318	2474	5223	108	12485
iquant1_intra	Multimedia	714	749	1045	5	2312
iquant1_non_intra	Multimedia	614	752	745	6	2066
iquant_intra	Multimedia	670	671	1123	5	2299
iquant_non_intra	Multimedia	626	659	835	6	2037
line_predict	Multimedia	927	1283	1357		3187
line_update	Multimedia	955	1312	1345		3221
matrixTranspose	Mathematics	723	974	636	6	2059
MD4Transform	Cryptography	7262	12013	10277		23675
MD5Transform	Cryptography	9689	16743	16047		36515
mdct_butterfly_16	Multimedia	2338	2405	4105	8	7630
mdct_butterfly_32	Multimedia	6683	6933	10907	56	21324
mdct_butterfly_8	Multimedia	734	835	1423		2901
mdct_butterfly_generic	Multimedia	3421	3050	4849	64	10965
merge	Other	1680	1887	2287		4450
mhash_adler32	Cryptography	683	951	875		2122
mhash_crc32	Cryptography	559	721	664		1643
mhash_crc32b	Cryptography	559	721	664		1643
movingfilter	DSP	1003	1209	1713		3349
multiply	Mathematics	2555	2647	3521	3	6764
Parity	ECC	399	238	718		900
permute_fp	Cryptography	1440	866	2585		3901
polygonArea	Mathematics	1119	1226	1561	12	3635
polynomial	Mathematics	778	576	1274	19	2875
power	Mathematics	288	374	277	3	811
powerefficient	Mathematics	367	472	368	6	1061
pred_comp	Multimedia	6596	7677	7685	8	16133
processBuffer	Cryptography	14389	17774	18613		36087
PseudoPolarize	Mathematics	1380	1549	2448		4386
PseudoRotate	Mathematics	1672	1778	2954		4995
QuickSort	Other	1329	1533	1869		3735
radixsort	Other	1612	1673	2366		4497
rc2_decrypt	Cryptography	1228	1454	2129		4065
rc2_encrypt	Cryptography	1239	1482	2061		3982
rijndael128_decrypt	Cryptography	5192	3902	8784		12954
rijndael128_encrypt	Cryptography	4907	3851	8553		12629
safer128_decrypt	Cryptography	2172	2256	3652		6392
safer128_encrypt	Cryptography	2184	2273	3776		6496
safer64_decrypt	Cryptography	2498	2069	4195		7055

Kernel name	Category	Slices	Slice Flip Flops	LUTs	MULTs	Nets
safer64_encrypt	Cryptography	2703	2345	4607		7802
saferplus_decrypt	Cryptography	15475	14274	21934		39788
saferplus_encrypt	Cryptography	10147	6549	16000		25967
serpent_decrypt	Cryptography	25999	39809	36284		76097
serpent_encrypt	Cryptography	26069	39888	35750		75505
sha_transform	Cryptography	17213	29391	25802		60015
shellsort	Other	990	1123	1309		2730
snefru	Cryptography	2249	1949	3833		5898
Sqrt	Mathematics	579	890	863		1977
threeway_decrypt	Cryptography	2872	4221	4252		8779
threeway_encrypt	Cryptography	2013	3073	2699		6053
twofish_decrypt	Cryptography	11179	12821	15387		30531
variance	Multimedia	921	1217	1070	4	2573
VectorSum	Mathematics	336	475	348		1120
vorbis_coslook_i	Multimedia	290	381	338	2	1088
vorbis_invsqlook_i	Multimedia	592	583	832	4	1798
wake_decrypt	Cryptography	1844	2257	2772		5277
wake_encrypt	Cryptography	1881	2269	2827		5312
xtea_decrypt	Cryptography	1013	1465	1161		2961
xtea_encrypt	Cryptography	1014	1465	1161		2960

D.2 Virtex4

Kernel name	Category	Slices	Slice Flip Flops	LUTs	DSP48s	Nets
adpcm_coder	Multimedia	1501	1943	2383		3497
adpcm_decoder	Multimedia	1336	1693	2022		3066
apply_butterflies	ECC	14586	16951	19983		29915
arcfour_encrypt	Cryptography	784	669	1279		2082
bdist1	Multimedia	1646	2266	1792	8	4206
bdist2	Multimedia	1410	1736	1744	5	3455
binarySearch	Other	674	848	987		1592
bitreversal1	DSP	837	895	1432		2257
bitreversal2	DSP	194	277	199		415
blowfish_decrypt	Cryptography	997	1288	1288		2361
blowfish_encrypt	Cryptography	982	1287	1282		2339
BubbleSort	Other	587	677	826		1523
bytesum	Mathematics	67	116	45		256
calcLCS	Other	1124	1327	1629	6	3292
cast128_decrypt	Cryptography	13950	8033	24916		25261
cast128_encrypt	Cryptography	14585	8057	25962		25693
compress_image	Compression	1146	1484	1478		2659
decompress_image	Compression	826	1137	823		1877
delta_forward	Multimedia	359	443	440		968
delta_inverse	Multimedia	328	409	408		987
dist1	Multimedia	5168	6233	6850		10959
dist2	Multimedia	2845	3615	3468	10	6528
divisiblebythree	Mathematics	690	934	884		1390
DotProduct	Mathematics	433	543	462	4	1331
enblf_noswap	Cryptography	794	962	1270		2007
enigma_encrypt	Cryptography	2171	2390	2921		4537
f	Cryptography	904	766	1307		2073
factorial	Mathematics	224	310	240	3	761
FIR	DSP	636	894	826	20	2951
fix_fft	DSP	2862	3102	4175	4	6417
floyd_warshall	Mathematics	1679	2014	2114	12	4496
form_component_prediction	Multimedia	6495	7846	7311	8	13251
g721_body	Multimedia	14953	11941	27068	18	31032
gcd	Other	1130	1172	1825		2222
generate_set	Mathematics	1491	2249	1143	39	5076
gosthash_compress	Cryptography	11959	13814	17469		23720
gost_decrypt	Cryptography	8640	8912	10570		17867
gost_encrypt	Cryptography	8640	8912	10570		17867
haar_predict	Multimedia	633	799	862		1728
haar_update	Multimedia	636	800	864		1727
Hamming1	ECC	167	266	183		384
Hamming2	ECC	264	377	281		609
havalTransform3	Cryptography	26489	44574	41917		53141
hw_boyer_moore_search	Other	2880	3341	3961		6236
hw_derrivative_x_y	Multimedia	1999	2065	2820	3	4453
hw_encode	ECC	1873	2489	2139		3722
hw_md2_transform	Cryptography	839	897	1304		2312
hw_mdct_bitreverse	Multimedia	1975	2240	2728	32	6221
hw_non_max_supp	Multimedia	19138	8841	34263	124	58319

Kernel name	Category	Slices	Slice Flip Flops	LUTs	DSP48s	Nets
hw_ripemd128_transform	Cryptography	15951	27936	21865		34103
hw_ripemd160_transform	Cryptography	25579	45525	32673		55347
hw_ripemd256_transform	Cryptography	16214	28116	22247		34631
hw_ripemd320_transform	Cryptography	25893	45701	33165		56046
hw_sobel	Multimedia	1816	2447	1830	8	4116
hw_viterbi	ECC	8857	8093	15802		17944
idct	Multimedia	4303	3824	6276	26	10329
intersect_triangle	Mathematics	4258	3174	7284	96	14174
intfdct	Multimedia	12306	6230	23669	48	29377
intmatmult	Mathematics	2194	2860	2307	4	4628
intmatmult3x3	Mathematics	1851	2154	2226	108	10059
iquant1_intra	Multimedia	651	701	952	5	1904
iquant1_non_intra	Multimedia	503	663	610	6	1691
iquant_intra	Multimedia	619	623	1018	5	1950
iquant_non_intra	Multimedia	516	570	701	6	1680
line_predict	Multimedia	926	1284	1328		2467
line_update	Multimedia	956	1312	1377		2563
matrixTranspose	Mathematics	679	940	584	6	1910
MD4Transform	Cryptography	7206	12017	10283		15481
MD5Transform	Cryptography	9606	16738	16047		22606
mdct_butterfly_16	Multimedia	2242	2289	3991	8	5885
mdct_butterfly_32	Multimedia	6133	6309	10092	56	16902
mdct_butterfly_8	Multimedia	743	838	1435		2243
mdct_butterfly_generic	Multimedia	2342	2521	3074	64	8751
merge	Other	1681	1894	2276		3753
mhash_adler32	Cryptography	679	951	872		1669
mhash_crc32	Cryptography	559	723	659		1409
mhash_crc32b	Cryptography	559	723	659		1409
movingfilter	DSP	1030	1210	1744		2586
multiply	Mathematics	2536	2606	3515	3	5626
Parity	ECC	427	265	745		819
permute_fp	Cryptography	1448	877	2582		3513
polygonArea	Mathematics	865	1034	1229	12	2875
polynomial	Mathematics	432	496	831	19	2389
power	Mathematics	244	342	249	3	752
powerefficient	Mathematics	315	472	310	6	1073
pred_comp	Multimedia	6493	7645	7486	8	13254
processBuffer	Cryptography	14380	17741	18594		26369
PseudoPolarize	Mathematics	1396	1551	2455		3446
PseudoRotate	Mathematics	1653	1778	2924		3821
QuickSort	Other	1278	1542	1850		3156
radixsort	Other	1624	1680	2389		3773
rc2_decrypt	Cryptography	1216	1456	2118		3175
rc2_encrypt	Cryptography	1230	1484	2050		3138
rijndael128_decrypt	Cryptography	5393	3933	8897		11087
rijndael128_encrypt	Cryptography	5073	3874	8657		10763
safer128_decrypt	Cryptography	2165	2268	3750		4929
safer128_encrypt	Cryptography	2236	2302	3858		4961
safer64_decrypt	Cryptography	2522	2092	4195		5748

Kernel name	Category	Slices	Slice Flip Flops	LUTs	DSP48s	Nets
safer64_encrypt	Cryptography	2664	2347	4543		6115
saferplus_decrypt	Cryptography	15522	14329	21857		34423
saferplus_encrypt	Cryptography	10162	6567	15978		22177
serpent_decrypt	Cryptography	26021	39826	36269		47950
serpent_encrypt	Cryptography	26098	39908	35769		48077
sha_transform	Cryptography	17050	29391	25808		36893
shellsort	Other	980	1125	1278		2241
snefru	Cryptography	2294	1977	3858		4953
Sqrt	Mathematics	579	890	863		1437
threeway_decrypt	Cryptography	2847	4222	4219		6286
threeway_encrypt	Cryptography	2001	3074	2698		4619
twofish_decrypt	Cryptography	11212	12872	15400		24312
variance	Multimedia	912	1217	1040	4	2142
VectorSum	Mathematics	336	475	345		943
vorbis_coslook_i	Multimedia	265	340	313	2	936
vorbis_invsqlook_i	Multimedia	534	542	769	4	1551
wake_decrypt	Cryptography	1852	2261	2813		4338
wake_encrypt	Cryptography	1866	2274	2839		4398
xtea_decrypt	Cryptography	1044	1467	1190		2358
xtea_encrypt	Cryptography	1045	1467	1190		2357

Software Complexity Metrics

E

This appendix contains a list of Software Complexity Metrics with their description. In our thesis we did not use the whole list of Software Complexity Metrics for our model, but only a subset.

Metricname	Description
Slices	Number of CLBs on a Xilinx FPGA (hardware measure)
Related.Slices	Number of CLBs on a Xilinx FPGA which contain only related logic(hardware measure)
Unrelated.Slices	Number of CLBs on a Xilinx FPGA which contain also unrelated logic(hardware measure)
Slice.Flip.Flops	Number of slice flip-flops in use (there are 2 flip-flops per slice in Virtex2pro)(hardware measure)
Total.LUTs	Number of Look-up Tables in use (there are 2 4-input LUTS per slice in Virtex2pro)(hardware measure)
Logic.LUTs	Number of Look-up Tables in use for logic (there are 2 4-input LUTS per slice in Virtex2pro)(hardware measure)
Route.LUTs	Number of Look-up Tables in use for routing purposes (there are 2 4-input LUTS per slice in Virtex2pro)(hardware measure)
Multipliers	Number of Multipliers used (There are 136 multipliers in our Virtex2pro's)(hardware measure)
AICC	Average Information Content Complexity a measure capturing the Information entropy expressed by the kernel
Average.Nesting.Depth	considering the nesting depth of all statements, calculate the average depth of the kernel
Average.Path.Length	Of all paths through the code, calculate the average length of those paths
BBavgExpPerBlock	Average number of expressions per basic block
BBavgExpPerStatement	Average number of expressions per statement
BbavgStaPerBlock	Average number of statements per block
BBcurMaxExpression	Maximum number of expressions in any block
BBcurMaxStatement	Maximum number of statements in any block
BBmaxExpPerStatement	Maximum number of expressions in any statement
BBnumBlocks	Number of Basic Blocks
BBtotalExpressions	Total number of expressions (subexpressions DO NOT count)
BBtotalStatements	Total number of statements
BINBitLogic	Number of Binary bit logic operations
BINDivisions	Number of divisions
BINLogic	Number of Binary Logic operations
BINMod	Number of Modulo operations
BINMultiplications	Number of multiplications
BINMultiplyBits	Number of multiplications considering datawidth of operands
BINShift	Number of shifts
Cumulative.Nesting.Depth	The nesting depth of each statement added to eachother
Cyclomatic	The number of decisions in the code (if statement, loop, or logical statement(lazy evaluation))
Loads	The number of memory loads
Maximum.Nesting.Depth	The maximum nesting depth in the kernel
Maximum.Path.Length	The length of the longest path among all possible paths
NPATH	The number of possible paths through the code (assuming loops run only one time)
Oviedo.DU.pairs	The number of data-use pairs in the code

Metricname	**Description**
Piwowarski	Based on cyclomatic complexity adjusted with nesting depth
PlusMinus	The number of ALU operations
Prather.s.mu	A measure capturing the number of required tests to validate code
Statements	The Number of Statements in the code
Stores	The number of memory stores in the code
Tai.DU.pairs	The number of data-use pairs, based on a control flow abstraction of the code
UNYBitNot	Number of unary bit level Not operations
UNYNot	Number of unary logical Not operations
VSArgMemCount	Number of Function Argument bytes
VSArgVarCount	Number of Function arguments
VSAvgMemPerVar	Average number of bytes per variable
VSMemCount	Number of bytes in variables
VSVarCount	Number of variables
Variable.Declarations	Number of variable declarations
nOperands	Number of operands
nOperator	Number of operators
nUOperands	Number of unique operands
nUOperator	Number of unique operators
MultsCSE	Number of multipliers used, counted after common subexpression elimination

Contents of CD-ROM

F

This appendix contains an overview of the contents on the CD-ROM provided with this document. The following sections represent the different directories on the CD-ROM

F.1 compiled tools/

This directory contains the compiled versions of the Logfiles Tool, the Prediction tool and the Xilinx ISE Tool.

- `Logfiles Tool/`
  This directory contains the compiled version of the Logfiles Tool and the other required files the tool uses.
- `Prediction Tool/`
  This directory contains the compiled version of the Prediction Tool and the other required files the tool uses
- `Xilinx Tool/`
  This directory contains the compiled version of the Prediction Tool, a sub folder *vhdls* for the vhdl files and the other required files the tool uses

F.2 source code tools/

This directory contains the source code of the Logfiles Tool, the Prediction Tool and the Xilinx ISE Tool.

- `Logfiles Tool/`
  This directory contains the java files with the source code of the Logfiles Tool
- `Prediction Tool/`
  This directory contains the java files with the source code of the Prediction Tool
- `Xilinx Tool/`
  This directory contains the java files with the source code of the Xilinx ISE Tool

F.3 vhdl files/

This directory contains the original vhdl files. These files can be placed in the sub folder *vhdls* of the Xilinx Tool. When running the Xilinx Tool, the files in this sub folder will be synthesized and implemented.