Placement And Routing For Reconfigurable Systems

Piotr Stępień

A thesis submitted in partial fulfilment of the requirements of Bournemouth University for the degree of Doctor of Philosophy

September 2009

Bournemouth University

Abstract

Piotr Stępień

Placement And Routing For Reconfigurable Systems

Applications using reconfigurable logic have been widely demonstrated to offer better performance over software-based solutions. However, good performance rating is often destroyed by poor reconfiguration latency - time required to reconfigure hardware to perform the new task. Recent research focus on design automation techniques to address reconfiguration latency bottleneck.

The contribution to novelty of this thesis is in new placement and routing techniques resulting in minimising reconfiguration latency of reconfigurable systems. This presents a part of design process concerned with positioning and connecting design blocks in a logic gate array. The aim of the research is to optimise the placement and interconnect strategy such that dynamic changes in system functionality can be achieved with minimum delay.

A review of previous work in the field is given and the relevant theoretical framework developed. The dynamic reconfiguration problem is analysed for various reconfigurable technologies. Several algorithms are developed and evaluated using a representative set of problem domains to assess their effectiveness.

Results obtained with novel placement and routing techniques demonstrate configuration data size reduction leading to significant reconfiguration latency improvements.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BCT	Bitstream Comparison Tool
CAD	Computer Aided Design
CDS	Configuration and Data Store
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Silicon
DLC	Dynamic Location Cost
EDA	Electronic Design Automation
EPROM	Erasable Programmable Read Only Memory
FO P&R	Frames Optimised Placement and Routing
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
JPR	Java Place & Route
LUT	Look-up Table

MP3	MPEG-1 Audio Layer 3
MPEG	Moving Picture Experts Group
OTP	One Time Programmable
P&R	Placement and Routing
PCB	Printed Circuit Board
PE	Processing Element
RAM	Random Access Memory
RCU	Reconfigurable Control Unit
RLU	Reconfigurable Logic Unit
RS	Reconfigurable System(s)
RTL	Register Transfer Level
SLU	Swappable Logic Unit
SMC P&R	Simultaneous Multi-Context Placement and Routing
SoC	System-on-Chip
SRAM	Static Random Access Memory
UMTS	Universal Mobile Telecommunications System
VHDL	VHSIC HDL
VLSI	Very Large Scale Integration
VPR	Versatile Place & Route

To my parents,

family and friends,

for their love, patience and support.

Acknowledgements

This thesis would have not existed without the efforts of many kind individuals. It gives me a great pleasure to acknowledge their contribution here.

I would like to thank my first supervisor, Dr Milan Vasilko, for his advice and encouragement, generous support, and dedication to see this project through to its successful end. I am very grateful to my second supervisor, Denzil Claremont, for his help, constructive criticism and support on this project.

I would like to thank Dr Jon Cobb for supervising the project in its final stage, especially for his support and dedication towards the successful outcome. I would also like to thank Dr Glyn Hadley for his constructive comments.

Special thanks to Professor Jim Roach and the rest of the management team of the School of Design, Engineering & Computing who have provided generous support throughout my studies.

I am grateful to my past and present colleagues, Petr Voleš, Periklis Chatzimisios, Pi Huang, Adam Polus and many others for their friendship, help and an excellent working environment they have provided over the years.

My work on this research would have never been possible without the support from my family. I am grateful to my parents for their continuing support and encouragement in my pursuit of the university studies and the scientific career.

Declaration

This thesis contains the original work of the author except where otherwise indicated.

Chapter 1

Introduction

In recent years, Field Programmable Gate Arrays (FPGAs) have become one of the most popular implementation technologies for digital circuits. Initially, being just a mesh of programmable logic units and interconnections, they have rapidly evolved to become complex System–on–Chip (SoC) solutions offering dedicated processing, storage and communication blocks to satisfy an ever increasing demand for configurable platforms that can be realised with minimum time to market.

The benefits gained from using FPGAs have been demonstrated across different, often computationally heavy application domains:

Various video processing algorithms have been successfully implemented in FPGAs, (Kulmala et al. (2006); Note et al. (2006); Agostini et al. (2006); Gorgon et al. (2005); Hormati et al. (2008), Kirischian et al. (2008)) often outperforming purely software implementations. A video decoder presented by Mignolet et al. (2003) implemented on reconfigurable hardware, was four times faster than its software implementation.

FPGAs have also proven to be an efficient platform for various image processing algorithms, (Garcia and Navarro (2006); Fahmy et al. (2006); Fahmy et al. (2005); Canto et al. (2009)) especially for the ever increasing image resolutions, and the growing popularity of image recognition based

systems.

The wireless communications application domain has also benefited from using FPGAs (Ahmed and Arslan (2006); Berthelot et al. (2006); Herrero et al. (2006); Esquiagola et al. (2005); Subramanian et al. (2009); Vogt and Wehn (2008)) to efficiently implement a variety of communication modules.

Examples from Mentens et al. (2006); Michalski and Buell (2006), Legat et al. (2009), Glas et al. (2008) and Huffmire et al. (2008) show that data security and cryptology have successfully used FPGAs in order to provide data security.

Yamaguchi et al. (2002) present results from using FPGAs in Reconfigurable Systems (RS) for a high–speed homologycal search. Their study found that the hardware approach was 330 times faster than an equivalent algorithm optimised for software implementation. Other examples from Arteaga et al. (2008), Koch et al. (2009) and Vogt and Wehn (2008) show, that FPGAs offer better performance for many algorithms previously reserved for microcontrollers.

As SRAM-based FPGAs can be programmed to perform different functionalities, they can also support reconfiguration, where the same hardware infrastructure can be reused to perform different functionality at different times. For this reason SRAM-based FPGA technologies can be considered as Reconfigurable Systems.

1.1 The Demand for Reconfigurable Systems

The demand for reconfigurable digital hardware is increasing as manufacturers realise the benefits of reconfiguration, for example:

- Increased functionality without additional hardware.
- Dynamic modification and upgrading of systems with no or minimal

system downtime.

- Realisation of adaptive systems.
- Remote hardware upgrade.

Several approaches have been proposed for reconfigurable systems. A Seamless Communication Platform presented in Vasilko et al. (2001) describes a multi–standard communication system with 'over the air' reconfiguration. Reconfiguration provides platform adaptability to different communication protocols and data coding requirements. Other examples of platforms for reconfigurable designs have been described by Rissa and Niittylahti (2000), Sedcole et al. (2003) and Marescaux et al. (2003).

Wireless communication is a good example of an application area that can benefit from reconfigurable functionality. At the present time, a change in communication protocol or introduction of a new network service often necessitates a user purchasing an upgraded device that supports new protocols. With reconfigurable devices an upgrade can be done in milliseconds (Rana et al., 2009) that is transparent to the user. Examples presented by Herrero et al. (2006), Ahmed and Arslan (2006), Subramanian et al. (2009) and Vogt and Wehn (2008) show that FPGAs can satisfy performance requirements specified by the wireless communication domain.

Apart from end–user devices, mobile base station or satellite transmission systems can also benefit from dynamic hardware reconfiguration. New network services can be introduced rapidly without expensive modifications to network infrastructure. It is especially important in satellite systems, where physical modification of existing satellites in space is usually protracted and costly. With reconfigurable hardware on board, a satellite can be reprogrammed from the control centre to provide new protocols and services. Examples presented by Vladimirova and Wu (2007) and Fiethe et al. (2007)

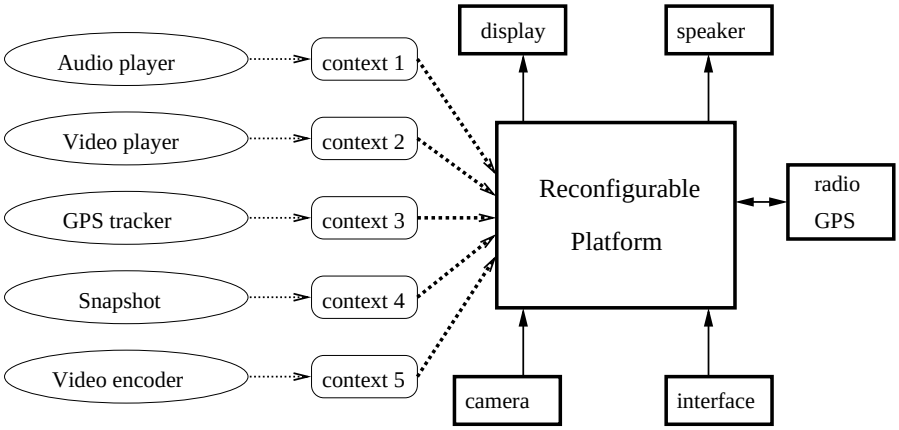

Figure 1: Example of a multi–context design using a single reconfigurable platform.

show that FPGAs can satisfy performance requirements specified by the satellite communication domain.

Another application area of notable interest is multimedia portable devices. Problems have emerged in relation to the variety of data coding/decoding methods and lack of well established standards. Consequently, many devices are limited in respect of the data protocols they can support. Extending support for new data protocols requires hardware modifications which cannot be done through software.

The example a smartphone implemented on FPGA-based reconfigurable system has been presented on Figure 1.

Single context presented on the Figure 1 refers to a portion of configuration data required to program FPGA to perform desired functionality. It is an equivalent of a single application loaded to the smartphone to switch it into desired functionality (e.g. multimedia player, GPS receiver, etc.).

In many situations reconfiguration of system functionality on a scale of a milliseconds (Resano et al., 2008) would not impose a significant burden on system usability and further improvement is not justified. However, in specific application areas, such as, reconfigurable computing, minimising reconfiguration delay is essential to maximise throughput. For example, real

time image processing requires a sequence of processing operations applied in parallel to image blocks. In this type of application optimal throughput is achieved if the images are stored locally in the processing matrix and local hardware reconfigured at each processing step. This overcomes the delay imposed by conventional hardware or sequential processors in moving large amounts of data between different elements of the functional architecture e.g. memory swapping. Moreover, minimalisation of data transfers can lead to significant reduction in power consumption. An example presented by Wang et al. (2006) showed that FPGA dynamic power consumption can be reduced when a modified set of design constrains is used at FPGA placement and routing.

Smit et al. (2002) identified the potential for dynamically reconfigurable mobile communication systems that operate in differing communication environments. In this type of application, additional functionality is limited by the requirement to minimise the physical size of the system to ensure portability and low battery power consumption. By using reconfigurable techniques the system functionality can be extended without adversely affecting these requirements. Similar arguments are presented by Helmschmidt et al. (2003) in their consideration of the realisation of the UMTS rake receiver.

Examples from different application domains illustrate the benefits of using dynamically reconfigurable hardware. An FPGA–based implementation of public–key cryptography algorithm developed by Mazzeo et al. (2003) benefited from a reduced design area and significantly increased performance. Gause et al. (2000) presented two reconfigurable design approaches for two–dimensional Shape–Adaptive Discrete Cosine Transform: one static where configuration does not change during algorithm execution and the second with dynamic reconfiguration enabled. They concluded that the dynamic approach significantly reduced the required FPGA are utilisation, however it was noted reconfiguration latency overhead did not significantly

improve. Vissers (2003) describes parallel processing architectures as a network of ALU–like structures with a corresponding set of instructions and recognises the opportunity for rapid dynamic reconfiguration and instruction multiplexing.

1.2 Limitations of Reconfigurable Hardware

Although FPGAs offers high performace and flexibility, there are a number of limitations that have prevented them from being widely exploited as reconfigurable systems. Rana et al. (2009) addressed reconfiguration latency – time required to switch configuration – as the key issue undermining performance improvements. As modern FPGAs configuration files size is of MBytes due to their complexity it adds a significant reconfiguration overhead. This impose reconfiguration latency and data storage penalty especially in multi-context designs.

Furthermore, configuration data size becomes an issue especially in battery operated, mobile devices, where configuration data storage and its transfer to the FPGA array contribute to the overall power consumption. The significance of this issue has been recently exemplified by the rapid increase in portable systems, such as MP3 players and mobile phones.

1.3 Design Support For Reconfigurable Systems

Efficient use of reconfigurable hardware could be impossible without design tools. They offer fully automated design flow mostly for a single content solutions, able to satisfy variety of design goals, like: routability, meeting timing constraints or power consumption requirements. Typically designs for FPGAs have been described using Hardware Description Language (HDL), proven to be an efficient way to describe hardware. However such an approach

is technology dependant.

Multi-context design as presented in Figure 1 can be processed using currently available design tools as a set of individual contexts. However such an approach is unable to benefit from any configuration overlapping mainly because each context is processed separately. Lack of feasible tools for efficient multi-context design has been addressed by Canto et al. (2009).

1.4 Scope and Objectives

This thesis evaluates the problem of placement and routing (P&R) for reconfigurable system. The reason to focus on placement and routing comes from the fact, that FPGA configuration data content depends on P&R results. Analysis of the current state of design tools for reconfigurable systems presented in Sections 1.1 through 1.3 allows a statement of the principal aim of the thesis which is the development and characterisation of novel placement and routing strategies for FPGAs that reduce reconfiguration latency and minimise reconfiguration data storage requirements.

The principal objective is to develop a method of minimising configuration data size by re–using parts of configuration data from previously loaded contexts. Novel algorithms will be developed and characterised for representative design applications. The results will be analysed to establish if generalised design rules can be established.

Partial result of this research inluding author's novel placement and routing approach has been published in (Stepien and Vasilko, 2006).

1.5 Thesis outline

The current chapter presents an introduction to the field of research and identifies the principal aim of the research project.

Chapter 2 provides a critical review of related literature and relevant background information on reconfigurable systems. Evaluation of state-of-the-art hardware, design methodologies, tools and conceptual approaches have been presented. Several technological approaches to reconfigurable systems together with reconfigurable logic architecture have been examined together with reconfiguration methods in order to demonstrate how the technology–specific features influence the design considerations during reconfigurable system design.

Chapter 3 discusses the methodology used to achieve design goals. It describes details about development framework, testing environment and quality measures used to evaluate the novel placement and routing algorithms. The benchmarking tests used to validate the solutions are also discussed.

Chapter 4 discusses the development of a novel bitstream size reduction methodology for placement and routing of a single context design and its optimisation. It describes how a configuration architecture interface can be incorporated into the placement and routing stage in order to address design goals. Critical evaluation of experimental results has also been presented in this chapter.

Chapter 5 discusses the enhancements introduced for a novel simultaneous multi-context placement and routing methodology. Details of configuration data sharing and its impact on the quality of delivered placement and routing results are also discussed there together with their impact on different FPGA technologies.

Chapter 6 presents the conclusions of the study with the directions of further research.

Chapter 2

Literature review

This chapter provides critical review of reconfigurable systems focusing on FPGAs as a leading technology. To describe the complexity of configuration bottleneck problem FPGA technologies have been discussed leading to the critical evaluation of methodologies used to improve reconfiguration. This chapter also provides critical evaluation of FPGA Computer Aided Design (CAD) tools with a focus on reconfiguration improvement.

2.1 Reconfigurable System Definition

Reconfigurable systems combine features previously existing individually in either software or hardware systems. Consequently, they provide the flexibility of processor–based software systems with performance approaching that of custom hardware circuits (Shirazi et al., 2000). Typical RS architecture contains a Reconfigurable Logic Unit (RLU), Reconfiguration Controller Unit (RCU), and Configuration Data Store (CDS). The RCU manages the RLU reconfiguration. Typically, the RCU is implemented as a micro–controller or microprocessor. It operates the task of reconfiguration control as well as data transfers to and from RLU. Sometimes, the RCU can be an integral

part of the RLU. The CDS unit provides memory for configuration data and application specific data. It can be organised as two separate memory blocks (one for each type of data) or one single memory block shared between configuration and application specific data.

2.2 Reconfigurable Systems Implementation

In the past reconfigurable system required several components: microcontroller, FPGA, DSP, storage. With submicron technology currently available FPGAs from Xilinx, Altera or Actel contain not only the array of configurable blocks but also dedicated microcontroller, DSP, storage and high-speed I/O interace (Xilinx, 2009b; Canto et al., 2009; Lewis et al., 2009; Actel, 2009).

There has been considerable research into possible reconfigurable computing architectures. Alternatives range from systems constructed using standard FPGAs to systems constructed using custom–designed chips. Some technologies like Xilinx XC6200 (Brebner, 1996; Luk et al., 1996; Brebner, 1997a) were introduced to support research on reconfiguration. A review on existing RS architecture approaches by Hauck and DeHon (2008) include discussion of a range of research and commercial technologies including Garp, PipeRench, RaPiD, Chimaera together with FPGA–based PAM, VCC, Splash, Prism and XC6200 architectures.

The variety of hardware solutions arises from the fact, that there is no single universal architecture that fulfils requirements of different applications. SoC has become very application specific in terms of size, available resources and reconfiguration technique, in order to avoid penalties in design speed and efficiency.

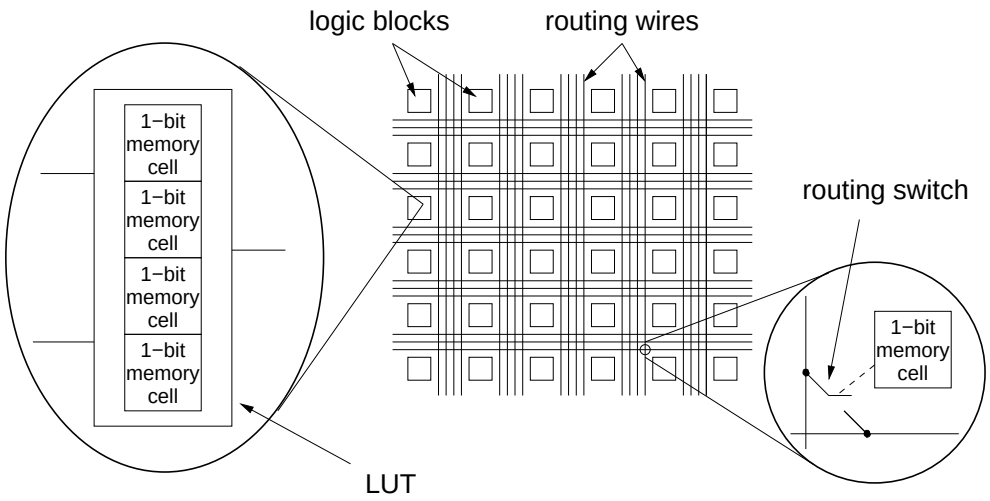

Figure 2: Example of island-style configurable logic architecture based on Betz (1999).

2.3 FPGA Architecture

The popularity of static RAM–based technology pushed forward the development of reconfigurable logic. Modern FPGAs offer not only the equivalent of several millions of logic gates but also dedicated microcontrollers, DSP modules, storage and high-speed I/O interface able to accommodate very complex digital designs (Xilinx, 2009b; Canto et al., 2009; Lewis et al., 2009; Actel, 2009).

A typical example of reconfigurable logic architecture is presented in Figure 2. It contains logic blocks (for logic functions) surrounded by the routing wires to provide connections between logic blocks. Wires are connected together via configurable switches controlled by individual memory cells. During the configuration process appropriate switches are set into on/off position to provide connectivity between logic blocks.

As has been described in Betz et al. (1999) all FPGAs are composed of three fundamental components: logic blocks, I/O blocks and programmable routing. A circuit is implemented in an FPGA by programming each of the logic blocks to hold a portion of a circuit's functionality, and each of the I/O

blocks to provide connectivity with the PCB. The programmable routing is configured to provide necessary connections between logic blocks and I/O blocks.

2.3.1 FPGA Programming Technologies

According to Betz et al. (1999) there are three different approaches to making an FPGA programmable:

- SRAM cells, controlling pass transistors, multiplexers and tri–state buffers.
- Antifuses.
- Floating gate devices.

While antifuse and floating gate devices represent OTP class of devices, SRAM–based FPGAs became the most popular reconfigurable technology used today.

2.3.2 FPGA Logic Block Architecture

The complexity of logic block used in an FPGA is a trade-off between flexibility, performance and has an impact of reconfiguration latency. While many different logic blocks have been used in FPGAs, most commercial FPGAs use logic blocks based on a look–up table (LUT) (Xilinx, 2009b; Canto et al., 2009; Lewis et al., 2009; Actel, 2009). An example of a LUT–based logic block is presented in Figure 3.

LUT example presented in Figure 3 contains 16 to 1 demultiplexer controlled by four LUT inputs. Depending on combination of LUT inputs selected memory cell is connected to the LUT output. Four bit LUT can perform any four input combinatorial function. Output register R is used to implement sequential logic functions. LUT inputs resolution has been

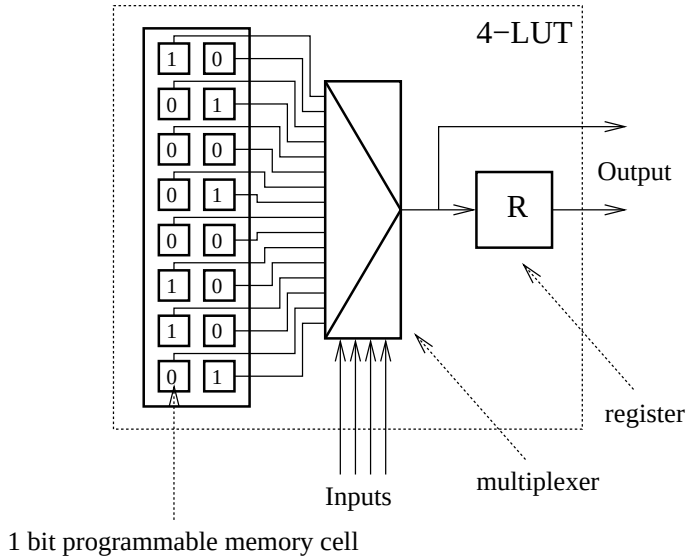

Figure 3: FPGA logic block with 4–input LUT and register.

investigated by various research groups to establish the optimal logic structure. These studies identified a trade–off between area efficiency and the number of LUT inputs. Eventually, the four input LUT became a standard for commercial FPGAs, although recently with silicon technology going into deep–submicron area 6–input LUTs have been used Xilinx (2009b).

In most of the modern FPGAs LUTs are grouped into clusters in order to host more complex functions inside a single logic block. Often LUTs within a single logic block are locally interconnected to enable them to host more complex functions. In Xilinx Virtex-6 4 LUTs form a slice and 2 slices are grouped into a single Configurable Logic Blocks (CLB) (Xilinx, 2001).

2.3.3 FPGA Routing Architecture

According to (Betz et al., 1999) FPGAs can be classified by routing architecture into the following groups:

- Island–style.

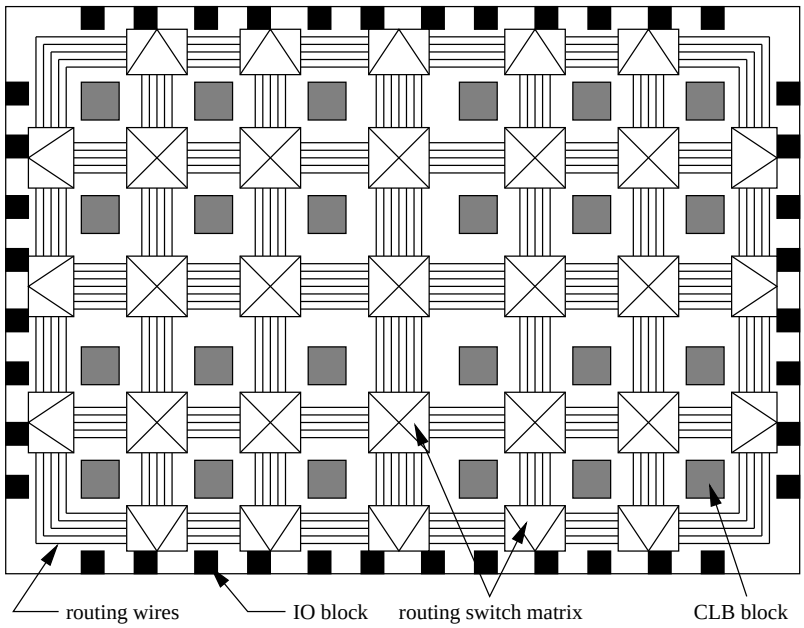

Figure 4: Example of island–style routing based FPGA based on Hauck (2008).

- Row–based.
- Hierarchical.

Island–style routing architecture where a logic block is surrounded by routing resources, can be found in Xilinx and Lucent devices (Xilinx, 2001; Xilinx, 2009b), while Actel's devices are row–based (Actel, 2009), and Altera's devices are based on hierarchical routing architecture (Lewis et al., 2009).

Island–style routing architecture is the most popular approach and is widely employed in commercial FPGAs and research devices. An example of an island–style routing architecture based FPGA has been presented in Figure 4.

As indicated in Figure 4 routing resources are grouped into channels spanning horizontally and vertically alongside logic block locations. Basic routing resources of unit length (i.e. they terminate at the next block location) are used to provide connectivity between logic blocks. Longer

connections can be built–up using a combination of single length routing wires, or long wires (spanning more than a single logic block), or a mixture of both, depending on the distance between logic blocks to be connected.

2.4 Design Automation

The advent of complex configurable devices with more than a million logic gates, made manual circuit design impractical, if systems were to be realised within the time constraints imposed by market forces. To support the design process, numerous design automation tools and methodologies have been proposed.

2.4.1 Design Techniques for One Time Programmable Systems

One Time Programmable (OTP) systems refer to the case when the configuration is loaded at startup and it does not change while the system is active. Single context programmed FPGA is a good example of OTP system. The methodology described in this section describes the design automation associated with this type of architecture. It is included as a basis for understanding several of the challenges associated with the current study. Figure 5 presents the typical design flow for non–reconfigurable system.

During behavioural synthesis an abstract behavioural design model is translated into one of several possible architectural design models, whilst attempting to meet the stipulated design constraints. An architectural design model is often referred to as a 'register–transfer level' (RTL) architecture, because it indicates data transfers between register blocks (Vasilko, 2000a). Subsequently, a logic synthesis process translates the RTL architecture description into a gate–level architecture resulting in a gate and

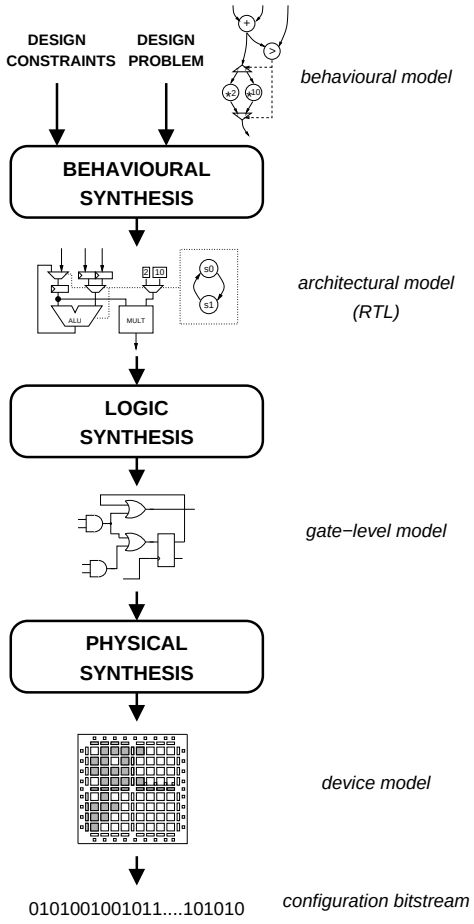

Figure 5: Typical Design Flow for configurable systems based on Vasilko (2000a).

interconnection netlist. This description is generic and a technology mapping process is necessary to map the gate–level netlist onto a specific target technology. The last process is known as the placement and routing (P&R) stage and involves placement of the target technology cells and their routing.

Collectively the steps outlined above form the Electronic Design Automation (EDA) process. Many EDA algorithms and methodologies have been proposed for automatic translation between the various design abstraction levels. For detailed information on these techniques the reader is referred to the literature, Gajski et al. (1992); Sherwani (1995); Gerez (1999) offer comprehensive discussion on these topics.

2.5 Configuration Interface Architecture

Configuration architecture is the underlying physical circuitry that loads the configuration data onto the chip and stores it at the correct location. Configuration architectures can range from a simple serial shift chain to addressable structures that can manipulate configuration information after it is loaded (Hauck and DeHon, 2008).

The process of FPGA configuration is a two stage process involving data transfer from external memory to the FPGA and data distribution to each programmable resource on the chip. The efficiency of the method used to transfer and distribute configuration data determines reconfiguration performance.

2.5.1 Serial Configuration Data Distribution

The simplest way to manage configuration memory cells in the FPGA is to group them into a single shift–register spanning across entire chip. Data are loaded serially to this register. Once data loading is completed configuration can be activated. Example of serial configuration interface is presented in

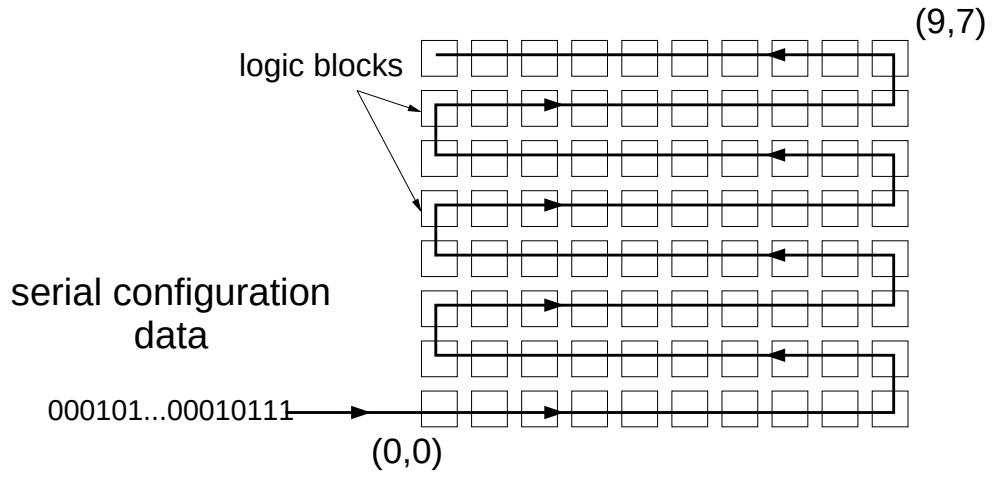

Figure 6: Example of Serial Configuration Data Interface.

Figure 6.

Serial configuration approach is quite fast when the entire chip needs to be reconfigured. It is also silicon area efficient due to minimal resources having to be dedicated to control configuration process. However, in situations where only a part of the configuration needs to be changed, entire chip has to be reconfigured due to the constraints of the configuration interface architecture. The Xilinx XC4000 family is an example of this type and incorporates a single serial configuration data distribution interface (Xilinx, 1999).

2.5.2 Parallel Configuration Data Distribution

The need for a faster and more direct access to the FPGA configuration registers was identified by Kean (1988). In response, parallel configuration data interfaces were implemented by Xilinx in their XC6200 family. With a memory like distribution architecture, single routing switches became directly accessible, and the configuration could be changed without the need to reconfigure other parts of the device. However, a parallel interface requires much more robust circuitry and is still too slow in many applications where a significant portion of the FPGA needs to be reconfigured. An example of

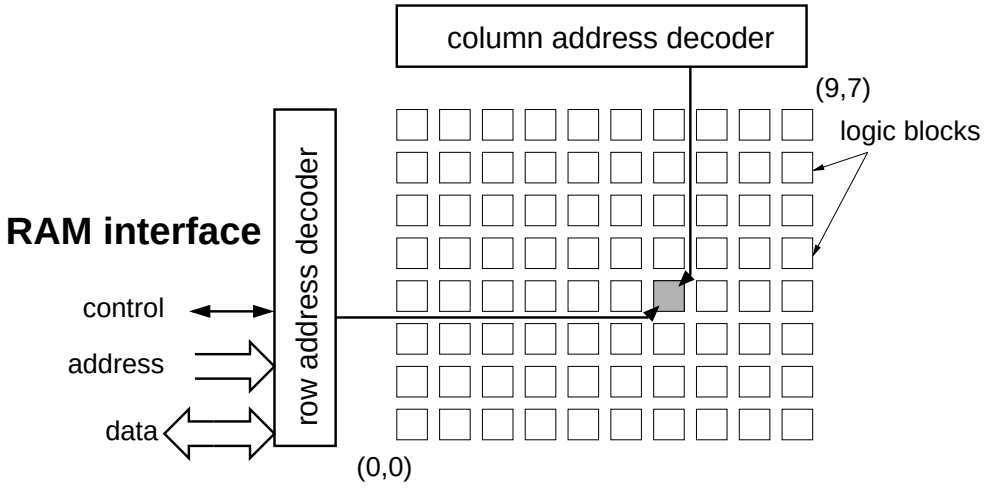

Figure 7: Example of Parallel Configuration Data Interface.

a parallel configuration data interface is presented in the Figure 7.

2.6 Support for Reconfiguration

2.6.1 Partial Reconfiguration

During the early development of FPGAs, reconfiguration was performed at start–up only. Consequently, the serial configuration data interface was a very convenient solution due to its simplicity. With further development of reconfigurable computing, demand for the capability to reconfigure parts of FPGA while other parts continue to operate (dynamic reconfiguration) increased. Lysaght and Dunlop (1994) classify an FPGA as dynamically reconfigurable if it can be partially reconfigured while active. To achieve partial reconfiguration capability, configuration data has been organised into clusters each controlling different parts of the device. Such an approach, allows reconfiguration of a part of a FPGA without interfering with other previously implemented or operational elements of the design.

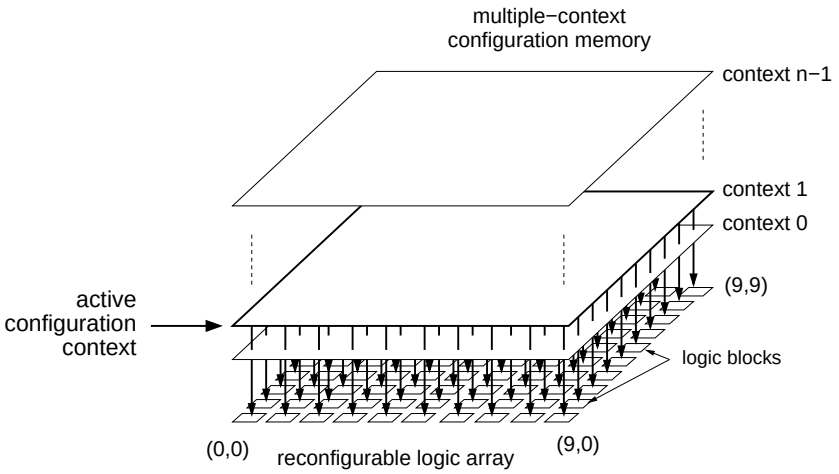

Figure 8: Multiple–context configuration memory from Vasilko (2000a).

2.6.2 Reconfiguration Latency

Different reconfiguration mechanisms demonstrate a tradeoff between the configuration throughput and the area overhead required for the implementation of the reconfiguration subsystem. The time needed for the configuration of a design module will vary with the technology. For partially reconfigurable technologies, the required configuration time will also vary with the current contents of the configuration memory in terms of number of frames which need to be reloaded. A frame in this context denotes the smallest sub–division of the device that can be independently reconfigured.

To accelerate the speed of reconfiguration, a cascade technique involving the connection of several configuration cells to one configurable resource was proposed by Brown et al. (1994) and later refined by Trimberger et al. (1997). These systems provide a configuration memory containing more than one configuration data store so that multiple–contexts can be preloaded. The configuration memory for each context is termed a context layer as presented on Figure 1 in section 1.3. Once the configuration data has been preloaded into the context layers, the desired configuration can be activated by selecting the appropriate configuration cells. This process is illustrated in Figure 8.

The main advantage of this approach is in the speed of reconfiguration. In current technologies entire device configurations can be changed within a few nanoseconds (Vasilko, 2000a). However, fast reconfiguration is achieved at the expense of the large silicon area overhead required for the configuration context memory and its control logic. Power consumption during reconfiguration is also difficult to estimate, which is an important requirement in the design of multi–function portable devices (Trimberger et al., 1997).

Vasilko and Ait-Boudaoud (1996b), investigated an alternate approach to fast reconfiguration which exploited integrated optoelectronics on silicon. Embedded photo–detectors can be used to configure switches and logic block contents instead of memory cells. Such an approach can benefit from faster reconfiguration without the overheads associated with the multi–context approach previously described. However, the ability to integrate high density optoelectronic detectors on silicon remains a challenge and commercial realisation of optically reconfigured FPGAs has not yet materialised.

Another basic approach to resolving the reconfiguration problem is to reduce the amount of data required to configure an FPGA. Modern FPGA architecture offers millions of simple logic blocks to host all sorts of algorithms. To provide connectivity between these logic blocks routing resources are also increasing exponentially. This gives the designer freedom and flexibility, but comes at the price of an increasing the amount of configuration data that needs to be loaded into an FPGA. For example, a typical configuration data file for the Xilinx XCV1000 which offers the equivalent of one million logic gates requires about 770kB of configuration data. Whilst the quantity of data is not high by today's standards the time to load this data serially even at the highest throughput rates is on the order of milliseconds. This is too slow for reconfigurable computing applications where reconfigurations times on the order of a few nanoseconds are required to sustain typical programme execution and data throughput rates.

Several approaches have been proposed to minimise configuration data

size. Dyer et al. (2002) proposed an architecture in which some of the routing, particularly of data buses is fixed. Thus only a part of the routing switches can be reconfigured which significantly decreases configuration data size.

An alternate approach utilises a different architectural paradigm, where logic blocks are much more complex structures e.g. PACT XPP Technologies (2003). This type of architecture decreases the level of flexibility offered by an FPGA although it benefits from a faster reconfiguration time and reduced configuration data size.

2.7 Dynamic Reconfiguration Implementation

The technique of partial reconfiguration subsequently led to the concept of dynamically reconfigurable systems. According to Lysaght and Dunlop (1994), an FPGA is classified as dynamically reconfigurable if it can be partially reconfigured whilst active. With Internal Configuration Access Port (ICAP) introduced on Xilinx Virtex-II FPGAs fabric can be reprogrammed by an internal microcontroller (Eto, 2007). This feature pushed forward research on fault-tolerant and self-reconfiguring design methodologies described in details in (Legat et al., 2009; Sterpone et al., 2008; Pilotto et al., 2008; Hu et al., 2008).

2.8 Design Techniques for Reconfigurable Systems

The introduction of reconfigurable systems brought a whole new set of challenges into design automation. The difficulties with the design of reconfigurable systems have been highlighted by Hadley and Hutchings (1995). As modern FPGAs contain a mixture of programmable hardware and microcontrollers (Hadley and Hutchings, 1995) emerged better support of

hardware/software design tools.

2.8.1 Hardware/Software Design

Introduction of microcontrollers as a part of FPGA fabric emerged the need for new tools linking together two different design methodologies for hardware and software.

Resano et al. (2008) have developed a hybrid design-time/runtime reconfiguration scheduling heuristic that generates its final schedule at runtime but carries out most computations at design-time. They demonstrated that the PowerPC 405 processor embedded on a FPGA generates a very small runtime penalty while providing almost as good schedule as a full runtime approach.

So and Brodersen (2008) explored the design and implementation of BORPH, an operating system designed for FPGA-based reconfigurable computers. Hardware designs execute as normal UNIX processes under BORPH, having access to standard OS services, such as file system support. Hardware and software components of user designs may, therefore, run as communicating processes within BORPH's runtime environment.

Deledda et al. (2008) presented the architecture and associated development tools of an heterogeneous reconfigurable SoC focusing on the chosen communication infrastructure. The SOC integrated units of various sizes of reconfiguration granularity. The included SoC approach demonstrated the scalability for actual and future SoC design.

Huang and Vahid (2008) explored the problem of operating system controlled dynamic management of the loading of coprocessors into the FPGAs for best overall performance or energy.

Bauer et al. (2009) addressed the problem of reconfiguration latency in reconfigurable systems. Comparisons with the most-prominent replacement policies show a speedup of up to 2.26. A parallel hardware implementation of

their MinDeg algorithm demands only 4,440 gate equivalents, which corresponds to 64% of the average requirements of one real-world reconfigurable accelerator.

2.8.2 Temporal Partitioning Problem

Initially temporal partitioning has been introduced to split complex deigns between multiple FPGAs. The temporal partitioning for reconfigurable systems is different from that of multiple FPGA devices (when a design is too complex to be mapped onto a single FPGA). While both problems address the partitioning of design computational and storage elements, temporal partitioning must also consider the temporal relationships between the individual design partitions to ensure that no dependency violations or conflicts occur during execution (Vasilko, 2000a). Temporal partitioning can be performed at different levels within the design automation process flow.

2.8.3 Temporal Partitioning at the Behavioural Level

Starting from a behavioural design model and set of constraints, the temporal partitioning is performed directly on the behavioural model. The product of such a temporal partitioning after the behavioural synthesis is a set of reconfigurable system partitions and a configuration controller for the design (Vasilko, 2000a).

Temporal partitioning at behavioural level makes it possible to explore tradeoffs between different architectural implementation options. Temporal partitioning at behavioural level is also referred to as 'task scheduling'. Several automatic techniques addressing temporal partitioning have been reported and will now be considered in detail.

A list of scheduling based synthesis techniques which support synthesis for partially reconfigurable systems was presented in Vasilko and Ait-Boudaoud

(1996a). The technique assumes a constant reconfiguration time and a simple approach is used for the high–level estimation of the available reconfigurable device area.

The 'online' scheduling technique proposed by Walder and Platzner (2003) is based upon the concept of allocating tasks to a block–partitioned reconfigurable device. Results obtained using this method benefit from a smaller configuration data size and lower reconfiguration latency.

Dynamic reconfiguration is also associated with the problem of on–line resource management. It involves resource allocation to make enough space for incoming tasks. Gericota et al. (2002) present a novel active replication mechanism for configurable logic blocks, able to implement on–line rearrangements without disturbing currently running functions. ElGindy et al. (2000) also studied the problem of tasks rearrangement on partially reconfigurable FPGAs.

2.8.4 Temporal Partitioning at Gate–Level

Once a gate–level design model has been generated, it is not possible to modify the architecture or execution schedule of a design. Temporal partitioning at gate–level becomes attractive if the final gate–level model cannot fit into the target device. One possibility is to 'fold' the implementation of the gate–level model over multiple design configurations (Vasilko, 2000a).

A solution presented in Shirazi et al. (1998) describes an optimisation technique based on graph bi–partitioning, capable of optimising the layout of two configurations. The algorithm maximises the overlap of similar blocks in two configurations in order to minimise the overhead associated with the reconfiguration of the partitions.

A solution proposed by Kielbik et al. (2002) presents a temporal partitioning algorithm based upon splitting a design into two, area balanced,

time independent sub–designs. These sub–designs can be mapped onto a target device with two configuration context layers.

The limitation of this approach is that the architectural implementation of the system, as captured in the gate–level model, cannot be changed. Temporal partitioning at gate–level cannot guarantee that system requirements such as reconfiguration latency dependencies will be satisfied.

2.9 Reconfigurable System Framework

The problems and differences raised in Section 2.8 identified the need for new design frameworks. Brebner (1997b) introduced the Swappable Logic Unit (SLU) as a new computing paradigm to support dynamic reconfiguration and placement in reconfigurable computer systems. This approach was also advocated by Luk et al. (1996).

From this diverse array of reconfigurable hardware platform architectures emerged the need for an architecture independent framework which could provide the designer with sufficient information about system performance across different platforms (Dyer et al., 2002). Several such frameworks have been proposed. A Prototyping Framework proposed by Sawitzki et al. (2001) focuses on design capture, testing and debugging of reconfigurable processor cores.

The DYNASTY framework proposed by Vasilko (2000b) focuses on design visualisation for dynamically reconfigurable systems. The DYNASTY approach to the design offers the advantage of being technology independent. The idea of keeping the design specification separate from a particular technology enables performance comparison across different technologies and platforms enabling further optimisation. Such a solution enables system designers a basis for comparison of design performance on different

technologies and provides reconfigurable hardware developers with improved feedback to assist refinement of the underlying technology.

2.10 Placement Algorithms Background

Placement & routing are the final steps in the design automation flow as illustrated previously in Figure 5. During placement and routing the gate–level model is transformed into final configuration data. Configuration data can then be loaded into the device to configure it. Target technology needs to be defined before placement and routing as these processes are always technology dependent.

2.10.1 Placement

Placement algorithms determine which logic block within an FPGA should implement each of the logic blocks required by a circuit. In general each placement algorithm starts from an initial placement and performs a series of placement iterations to find optimal results according to the placement schedule. Cost function is used to determine the quality of the placement iteration.

2.10.1.1 Initial Placement

The initial placement procedure determines blocks allocation at the beginning of the search for a satisfactory placement solution. It can be done either by random allocation or using deterministic methods to allocate each individual block to a physical location within the device.

Depending on the placement algorithm used, initial placement can influence the quality of the final placement especially when deterministic search based algorithms are employed.

2.10.1.2 Placement Cost Function

A cost function is used by the placement algorithm to determine the quality of a proposed solution. It represents a trade–off between algorithm speed and efficiency, as the cost function needs to be recalculated after each iteration of the placement algorithm. A typical placement cost function depends on either a net bounding box or net wire length as the fastest way to evaluate the quality of the proposed solution.

2.10.1.3 Next Step and Placement Scheduling

Placement scheduling determines placement algorithm mechanisms used to deliver an optimum design placement solution. It controls algorithm performance, solution acceptance criteria, and the next step proposed solution. The next–step determines the way in which a new proposed placement solution will be refined in order to deliver a satisfactory placement result.

2.10.2 Placement Algorithms Overview

The common optimisation goals are to place connected logic blocks close together to minimise the required wiring (wirelength–driven placement), to place blocks to balance the wiring density across the FPGA (routability–driven placement), or to maximise circuit speed (timing–driven placement) (Betz et al., 1999).

Placement algorithms can be divided into two categories:

- Constructive placement – once the location of a block has been determined it remains fixed.
- Iterative placement – all blocks have randomly assigned initial locations and blocks are moved around or swapped in order to get a new configuration.

Most placement algorithms comprise a combination of both approaches: an initial placement is obtained in a constructive way and attempts are made to increase the quality of the placement by iterative improvement (Gerez, 1999).

Constructive placement is represented by the min–cut method. The idea of min–cut placement is to split the circuit into two sub–circuits of more or less equal size while minimising the number of connections between sub–circuits. Two sub–circuits obtained in this way will be placed into separate halves of the FPGA. This type of bi–partitioning is recursively applied to the sub–circuits until further partitioning yields no further benefit against quantifiable criteria. Min–cut placement is based on Fiduccia–Mattheyes algorithm (Fiduccia and Mattheyses, 1984) and respresents a top–down method: it starts with whole circuits and ends with small elementary sub–circuits. The opposite approach to min–cut is clustering which is effectively a bottom–up approach. Clustering starts with single elementary sub–circuits and finds one or more sub–circuits that share nets with it. These sub–circuits are then taken together to form a cluster. The cluster is then placed. The process is repeated until the cluster contains the entire circuit.

Another example of constructive placement is represented by Force-directed algorithm (Shahookar and Mazumder, 1991). Starting from an initial random placement, each logic block is moved to its best location. Best location for each block is determined as the closest available location to the centroid of all the other blocks to which this block is connected. If destination location is occupied, the two blocks are swapped. After moving to the best location the block is locked and its location is considered unavailable. Single placement iteration lasts until all the blocks are locked. If the exit condition has not been met, all the blocks are unlocked and next placement iteration begins.

Iterative placement is based on iterative improvement strategy which perturbs a given placement by changing the position of one or more blocks

and evaluates the result. If the evaluated cost is lower (i.e. the placement is improving) then the new placement replaces the old one and the process continues. Conversely, if the evaluated cost is higher, the modification can still be accepted under certain circumstances to avoid failed convergence arising from being stuck in a local minima of the solution space.

The most popular method of iterative placement is Simulated Annealing (SA). It mimics the annealing process used to gradually cool molten metal to produce high–quality metal objects where crystal structure is very regular. An initial placement is created by assigning logic blocks randomly to the available locations in the FPGA. A large number of moves, or local improvements, are then made to gradually improve the placement (Betz et al., 1999). Following each iteration the cost function is recalculated to determine whether to accept or decline the new placement location.

Mulpuri and Hauck (2001) compared several placement algorithms demonstrating tradeoff between runtime and quality. Results presented by Mulpuri and Hauck (2001) show that Simulated Annealing dominates other placement algorithms.

2.11 Routing

Once locations for all logic blocks in the circuit have been chosen, the router determines which programmable switches should be turned on, to connect all the logic block inputs and outputs defined in the netlist. An example of routing is shown in Figure 9.

As FPGA routing resources are fixed, the routing problem can actually be represented as finding shortest path through a routing–resource graph. In this type of analytical model the graph nodes represent the logic block pins to be connected. Furthermore, the interconnect switches are represented by the vertices and wires are represented by edges. Each wire has a cost attached

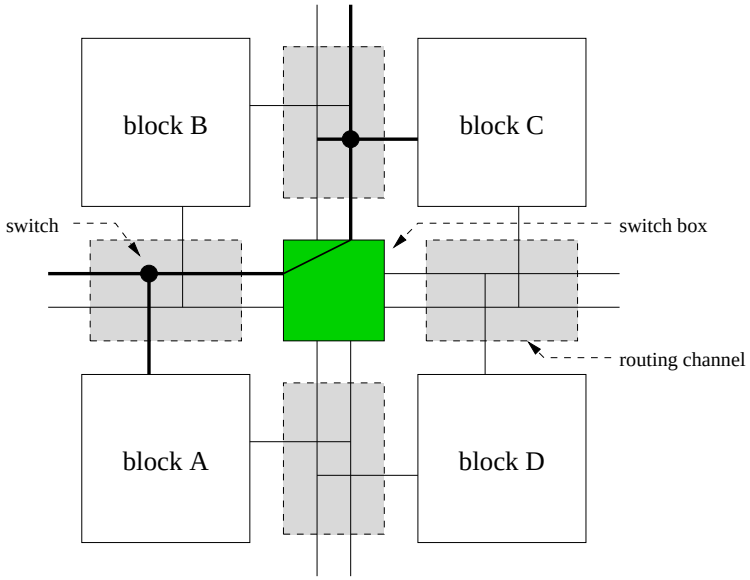

Figure 9: FPGA routing example.

to drive router to find the best way to provide connectivity between selected points. An example is illustrated in Figure 10 for the trivial case of a six node representation.

In the example shown in Figure 10 the cheapest path between v1 and v6 goes through v4 and v5. As the main goal of the FPGA router is to route the design using the shortest possible paths, Dijkstra's algorithm is used (Gerez, 1999; Betz et al., 1999; Hauck and DeHon, 2008). The algorithm determines the shortest path between a source node and sink node in a routing resource graph using the cost associated with each routing wire.

In general, there have been two types of search: (i) breadth-first and (ii) depth-first. Breadth-first search begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal. Depth-first search starts at the root and explores as far as possible along each branch before backtracking.

FPGA routers can be divided into two groups. Combined global–detailed

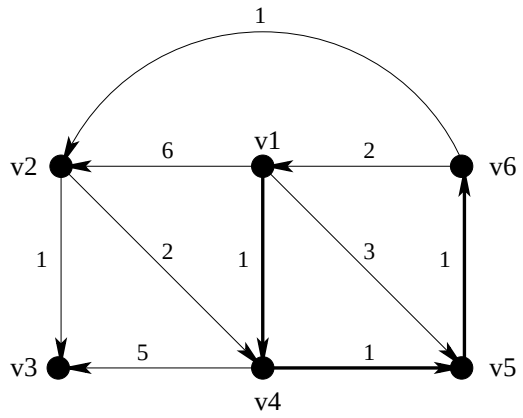

Figure 10: Example of edge–weighted directed graph.

routers determine a complete routing path in a single step. Two–step routing algorithms perform global routing first, to determine which logic block pins and channel routing segments each net will use, and then performs detailed routing to determine which physical wire a net will be allocated to within a specified routing channel segment. The task of an FPGA detailed router is often complicated by limited FPGA routing flexibility and a dependency on decisions made by the global router. For this reason, combined global–detailed routers offer improved potential to fully optimise the routing (Betz et al., 1999).

2.11.1 Pathfinder Router Algorithm

To address the problem of routing bias, arising from the order in which the nets are processed, Betz et al. (1999) introduced Pathfinder router. The Pathfinder is a negotiation–based router, which allows routing resource overuse during the initial routing passes, and then re–routes nets until resource overuse is completely resolved.

2.11.1.1 Initial Routing

During initial routing each net is routed with unconstrained access to routing resources to find the shortest path for each net. Wherever a single routing resource has been claimed by more than one net, routing resource overuse occurs. If at the end of this process illegal routing remains, further iterations are attempted until illegal routing is eliminated.

2.11.2 Congestion Avoidance Schemes

To resolve the routing resource overuse problem, Pathfinder uses a congestion avoidance scheme based on negotiation, where each routing resource has an overuse cost attached. After a routing iteration overusing, an increased cost on a particular net necessitates rerouting of that net.

2.11.3 Delay modelling

To ensure design timing constraints are met, FPGA routers need to determine the delay of each net and ensure all of them are within design timing constraints.

The most common way to model timing of the net has been based on the Elmore delay, which defines each net as a resistance and capacitance tree (RC). The Elmore delay of a source – sink path can be described (Betz et al., 1999) as:

$$\sum_{i \in source-sinkpath} R_i \cdot C(subtree_i) + T_{d,i} \tag{2.1}$$

Where $T_{d,i}$ is the intrinsic delay of a buffer, if it is used as routing element i, R_i represents the resistance of routing element i and $C(subtree)_i$ is the total capacitance of the subtree routed from the routing element i.

2.12 Placement & Routing Methodologies

The problem of automated placement and routing has been widely addressed in PCB and later in VLSI design automation. Work in this area has been reviewed by Gerez (1999); Sherwani (1995). However, P&R for FPGA is significantly different to that of P&R for PCB or VLSI due to the constraints imposed by the fixed location of blocks and routing resources in circuit board and full custom technologies.

With PCB or VLSI placement and routing, the focus is on delivering an optimal solution usually in terms of the smallest PCB or silicon area respectively. In contrast, the main goal of FPGA P&R is to fit the circuit into the FPGA, and to satisfy design timing requirements. In this context, once a satisfactory solution has been derived, further optimisation is usually unnecessary. This fact has resulted in development of heuristic methods of placement and routing for FPGA rather than adaptation of well proven algorithms from the PCB and VLSI CAD domains.

To find an optimal FPGA placement solution, combinatorial optimisation methods can be applied. This is possible as placement can be represented as a problem of allocating a finite number of objects to a finite number of FPGA blocks. A solution presented in Emmert and Bhatia (1999) uses Tabu search to find optimal placement. Tabu search is a heuristic approach to solve optimisation problems that approaches near optimal solutions in a relatively short time. Results of placement using Tabu search optimisation show up to a factor of 20 times improvement in placement execution time compared to the standard Xilinx M1 tool.

Mulpuri and Hauck (2001) researched the problem of runtime and quality tradeoffs in placement and routing. Results demonstrated clearly show the benefits of using Simulated Annealing over other existing methods. Also Betz et al. (1999); Anderson et al. (2000); Abke and Barke (2001) successfully used Simulated Annealing in their P&R frameworks. Simulated

Annealing was also used by Ebeling et al. (1995) used as a placement algorithm for the Triptych FPGA.

In order to support development of P&R algorithms a set of benchmark circuits have been established to determine the quality of proposed solutions in P&R. Several benchmarks have been established – the most popular benchmark reference circuits are those from MCNC (Microelectronics Centre of North Carolina), which have been widely used for the development and refinement of many P&R algorithms (Anderson et al., 2000; Kannan and Bhatia, 2001; Kannan et al., 2001; Abke and Barke, 2001; Kannan et al., 2002).

A very comprehensive solution has been developed by Betz et al. (1999). Their Versalite Place and Route (VPR) is a place and route tool offering several execution options. Placement can be wire–length driven or path–timing driven with different parameters of simulated annealing optimisation. The routing algorithm can be timing–driven or routing–driven with a user defined number of iterations. Targeted architecture is also user defined and allows comparison between different architectures and different P&R methodologies. VPR has been widely used due to the benefits offered by its flexible approach (Anderson et al., 2000; Kannan and Bhatia, 2001; Kannan et al., 2001; Abke and Barke, 2001; Kannan et al., 2002).

Historically, Placement and Routing (P&R) has been considered as a single unified process because of the strong interdependency of the goals of each subprocess (Kannan and Bhatia, 2001). However, through separation placement can be independently controlled in such a way that the subsequent routing can be improved compared to a unified approach.

As an example, Fast Generic Routing Demand Estimation for Placed FPGA Circuits (fGREP) presented in Kannan et al. (2002) speeds up placement and routing process using routing demand estimation in order to measure quality of placement. Extensive routing estimation provides additional information for the placer which results in improved design routing.

Conversely, Gambit, a proof–of–concept presented in Karro and Cohoon (2001) suggested a need for simultaneous P&R as a means of providing better results than sequential routers.

Another solution for improved router performance has been proposed in McCulloch and Cohoon (2003) and is based on negotiation–based routing. Each net is given an 'initial fund' for biding in 'pin–auctions'. A net is considered to have a complete detailed route if the set of pin–auctions that it is currently winning comprise a path that would be a legal detailed route for the net.

2.13 Summary

Methodologies and algorithms developed for placement and routing for FPGA, even though constantly improving, are still generally optimised for one time configuration rather then reconfiguration.

At the present time, multi–context designs are realised by designing each configuration layer separately using conventional P&R tools. With this approach any optimisation across configuration layers can only be achieved manually by the designer. This manual process is time consuming and does not guarantee improvement.

Typically, each configuration layer is designed separately and existing P&R methodologies optimise the design only in terms of occupied area or timing constraints within a single configuration layer.

Existing P&R methodologies do not take into account reconfiguration interface architecture and the way in which reconfiguration is performed. Evaluation of a variety of different reconfiguration interfaces and different FPGA architectures by the author has confirmed the need for new P&R methodologies which can place and route designs so that they are optimised for reconfiguration.

Automated placement and routing for reconfigurable systems can speedup reconfigurable systems design. Successful P&R for RS methodologies and algorithms with a focus on configuration overlapping will benefit from smaller reconfiguration latency, smaller configuration memory size and decreased power consumption. These benefits will be realised through a reduction in the quantity of configuration data that needs to be transferred during a change of operational context. Such new methodologies will provide the required process continuity in frameworks targeted for reconfigurable systems. In the broader application context this will enable the development of portable devices offering limitless flexibility while providing high performance at low power.

Chapter 3

Design Goal and Research Methodology

This chapter provides detailed description of the methodology used to achieve design scopes and objectives.

3.1 Configuration Data Optimisation Problem

To demonstrate how placement contributes to frames utilisation, a simple FPGA placement in two variants has been presented in Figure 11. It presents an example of a different implementation of the same design resulting in a very different number of frames required for configuration.

Figure 12 presents an example of reconfigurable platform supporting partial reconfiguration with two designs to be performed on the platform – one at a time.

Shared configuration represents the case, where FPGA resources controlled by the shared frames are configured exactly the same in two or more designs. In this way it can be assumed, that there is a group of blocks of the same content and/or nets of the same topology in each of the

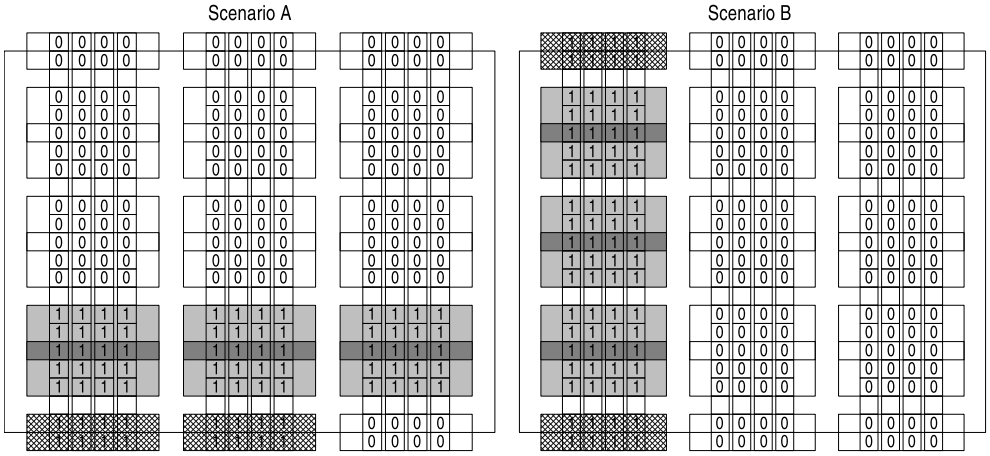

Figure 11: Two design implementations with different configuration data sizes: Scenario A = 12 frames, Scenario B = 4 frames.

designs. Furthermore, it can be assumed, that these shared blocks and/or nets represent sub–functionality identical in every design.

Bitstream content depends on the placement and routing decisions taken, therefore to obtain specific frames content, placement and routing needs to be modified. The question which then arises is how to determine the maximum similarity of two functionally different designs?

Solving the generic problem of reconfiguration latency optimisation at the placement and routing stage is difficult for a number of reasons:

- Lack of standardisation in FPGA reconfiguration procedures.
- Lack of configuration interface architecture models.
- Unavailability of configuration interface architecture and data structure details.

As described in Section 2.5 each FPGA family uses different configuration interface therefore different reconfiguration techniques apply to each of them. FPGAs offering partial reconfiguration come with different configuration data structures, which make it difficult to build a universal configuration model

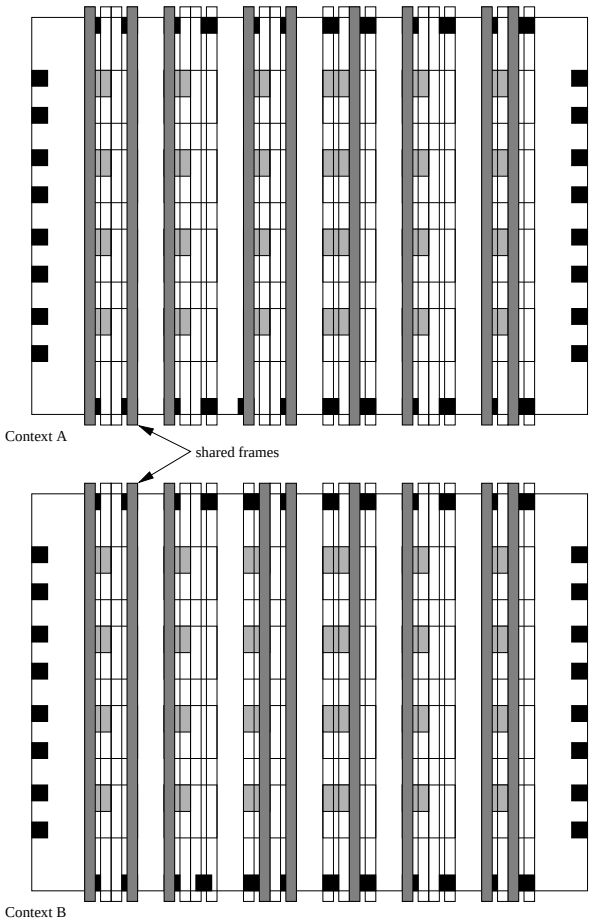

Figure 12: Example of multi–design project using the same platform.

for placement and routing purposes. Such a model has to be created for the purpose of this project.

3.2 Design Goal

The goal of this project was to provide solutions to minimise reconfiguration latency and storage for a specified FPGA technology given a set of design contexts. This was achieved by introducing new placement and routing optimisation techniques. Placement and routing optimisation focused on re–using parts of configuration data between designs in order to minimise reconfiguration latency and configuration data storage overhead. A proposed solution was required to demonstrate its practicality and needed to be applicable to different reconfigurable architectures.

As has been demonstrated in the section 2.3, currently used placement and routing algorithms and methodologies use models, which do not include configuration interface architecture information. They are therefore unable to deliver configuration data sets with the content necessary to address the configuration data sharing problem.

To achieve the design goal, the FPGA architectural model used during placement and routing has to include configuration interface architecture information, and be able to deliver configuration data with the desired content. Any new model therefore, has to be evaluated to investigate its impact on existing placement and routing optimisation goals such as performance and complexity as well as optimisation for reconfiguration.

3.3 Research Methodology

The entire research problem presented in section 3.2 was divided into two subproblems: (i) automated single-context bitstream size reduction during

placement and routing and, (ii) automated multi–context bitstream sharing during placement and routing.

Bitstream size reduction at the placement and routing stage demonstrates the impact of placement and routing algorithms on configuration data content. Solving the bitstream size reduction subproblem provided a modified FPGA P&R technique that enabled development of a multi–context placement and routing algorithm for dynamic reconfiguration.

Since an FPGA bitstream results from mapping of a design netlist onto s specific FPGA technology the unique architectural features of the target device has to be known in advance.

3.3.1 Development Framework

In many earlier projects in the area of FPGA placement and routing Versalite Place and Route (VPR) was used (Abke and Barke, 2001); (Kannan et al., 2001); (Kannan et al., 2002); (McCulloch and Cohoon, 2003); (Barreiros and Costa, 2003).

Although VPR represents a framework suitable for placement and routing algorithms evaluation, its placement and routing algorithms do not include support for configuration interface architecture and bitstream generation as required in the present study. It proved difficult to build a new FPGA architecture model which included configuration interface architecture and bitstream protocol support as (i) FPGA vendors were unwilling to provide the required interface specification in order to protect their intellectual property and, (ii) Given that the bitstream is the final output from the design tools the need for further modification was not a requirement.

However, a solution to the preceding problem was available from Xilinx with its JBits (Xilinx, 2009a) product developed to support the reconfigurable FPGA research community. JBits is a set of Java classes supporting bit–level bitstream manipulation on Xilinx Virtex FPGA devices. JBits was

considered a suitable post–processing optimisation tools for the proposed study as it offered full flexibility to change the bitstream content according to specific optimisation requirements.

3.3.1.1 Target Architecture

To demonstrate applicability of the new placement and routing methodology development has been based on the real-world FPGA architecture. The choice of bitstream manipulation tools required for the development is limited only to Xilinx JBits (Xilinx, 2009a) as the only available bitstream manipulation tool. JBits version 2.8 used in this research project supports Xilinx Virtex technology only. Although Xilinx Virtex FPGA family has been replaced by more advanced architectures, it is still a good example of well documented partially reconfigurable technology.

To simplify the process of evaluating new placement and routing methods development was targeted to the XCV50 – the smallest device in the Xilinx Virtex family (Xilinx, 2001). The XCV50 has an 16 x 24 array of CLBs controlled by 1152 configuration frames, each containing 324 configuration bits (Xilinx, 2002; Xilinx, 2000; Xilinx, 2001).

The Xilinx Virtex FPGA is an example of an island–style FPGA. A single Virtex Combinatorial Logic Block (CLB) cell contains two slices – each containing a four bit LUT and routing switch matrix. Figure 13 illustrates a Xilinx Virtex programmable cell.

Configuration of a single Xilinx Virtex CLB cell is controlled by 864 bits (Xilinx, 2002; Xilinx, 2000). Half of these bits are used to configure the routing switch matrix itself, whilst the other half control LUT configuration, CLB input and the output switch matrix.

From the configuration architecture point of view the same cell can be represeted as an array of registers, each controlling configuration of a part of the programmable cell, as presented in Figure 14.

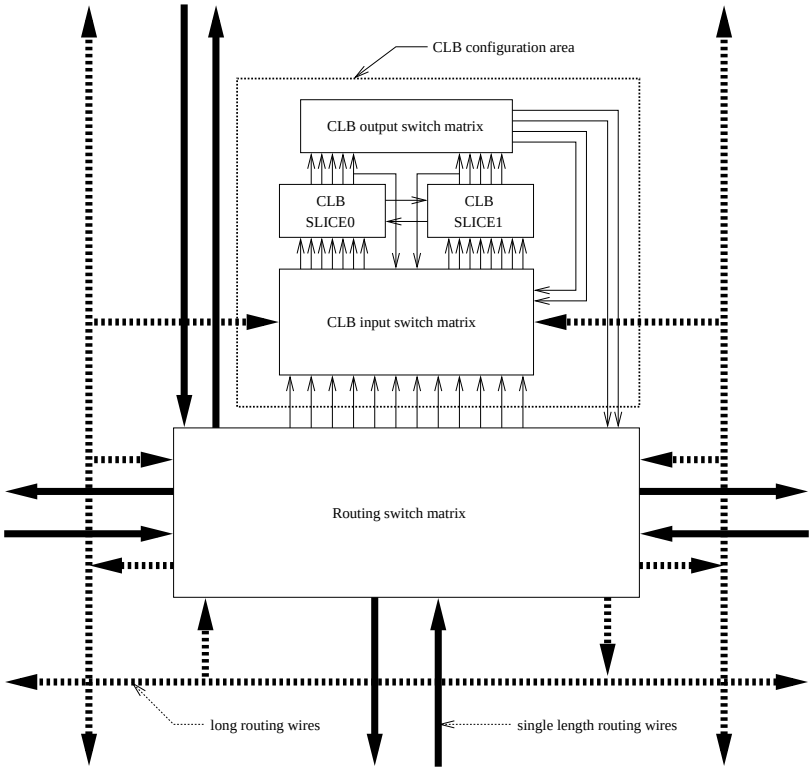

Figure 13: Example of Xilinx Virtex single programmable cell.

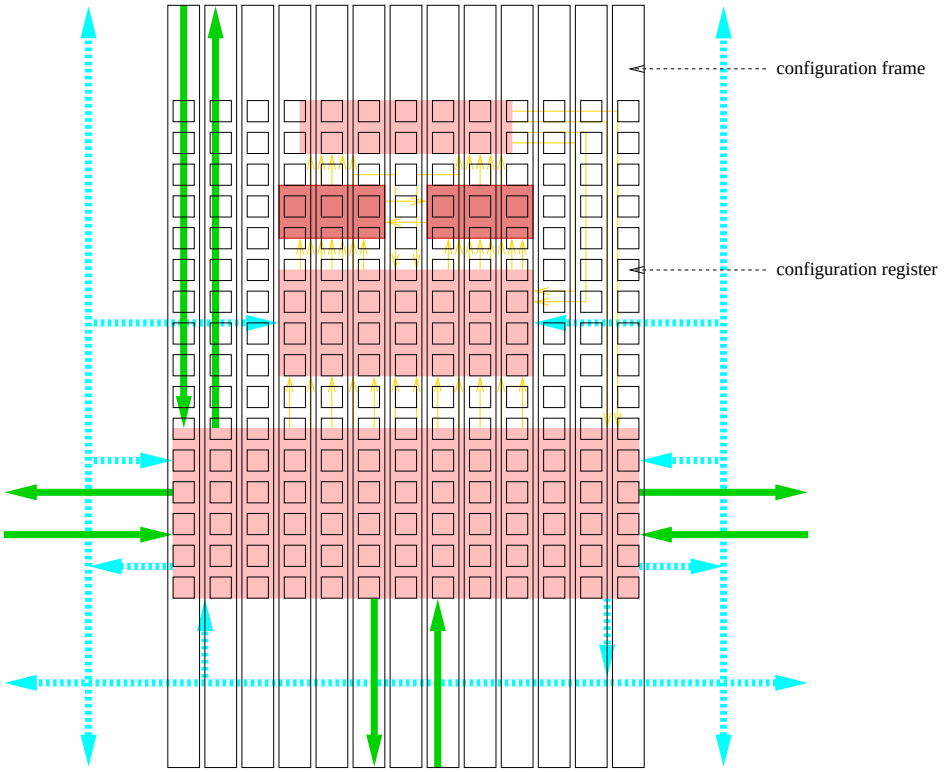

Figure 14: Example of Xilinx Virtex configuration architecture controlling a single programmable cell.

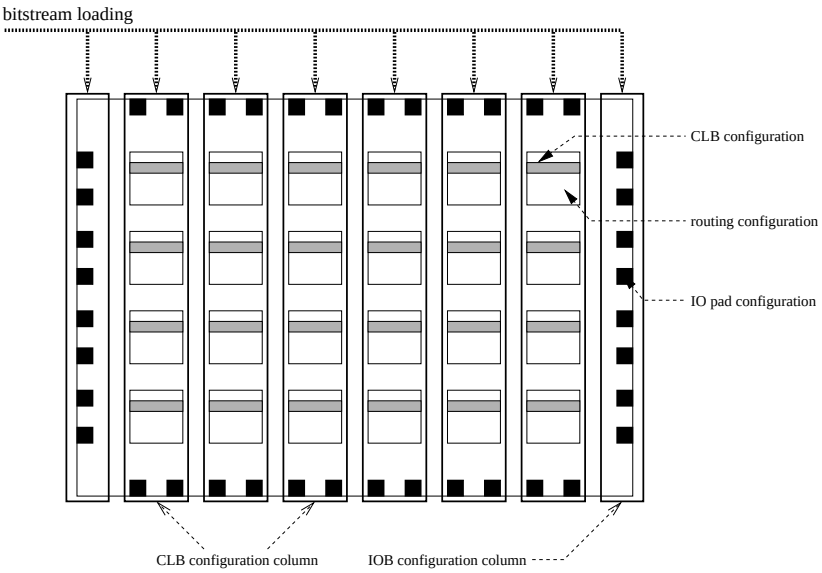

Figure 15: Example of Xilinx Virtex FPGA configuration columns of frames architecture.

Access to a single configuration register requires complex configuration interface circuitry, therefore single configuration registers are grouped into clusters called configuration frames. A single configuration frame is the smallest amount of configuration data loaded into an FPGA at a time. In the Xilinx Virtex FPGA, the smallest configuration data block is represended by a configuration frame spanning through an entire CLB column. An example of the distribution of the configuration data in a Xilinx Virtex FPGA is given in Figure 15.

To configure a single FPGA programmable cell, all the frames covering the particular cell need to be loaded into the FPGA. A detailed register mapping depends on the cell configuration which describes the logic/routing resources that need to be configured. Therefore the content of an FPGA configuration data, bitstream, is dependent on placement and routing results and can vary.

To illustrate the complexity of the single CLB column configuration process for a Xilinx Virtex FPGAs, summary of configuration data for the

Table 1: Xilinx Virtex FPGA configuration data size summary.

Device name	CLB Array size	Frame size	column size
XCV50	16 x 24	324	15,552
XCV1000	64 x 96	1,188	57,024
XCV3200E	104 x 156	1,908	91,584

Xilinx Virtex FPGA family is presented in Table 1.

As demonstrated in Table 1 an increase of CLB columns requires more data to be transferred during the configuration process even if the design complexity remains unchanged. Thus, if placement is not optimised to a single column there can be a significant increase in storage size and configuration latency.

To configure the entire FPGA all frames need to be loaded. Design reconfiguration can be performed by either re–loading the entire bitstream or by re–loading only those frames which contain data different from that of a previous design (using partial reconfiguration).

It is the partial configuration which offers the opportunity to reduce the size of configuration data, even more so if the design can be placed and routed so that it uses minimal number of frames.

Figure 16 demonstrates how placement and routing determine configuration bitstream content. From the configuration architecture point of view, an FPGA can be considered as a two–dimensional array of CLB locations as described in Figure 16.

Following the approach of Figure 16 each individually implemented design can be described as a two–dimensional array of free/taken CLB locations as shown in Figure 17.

A CLB cell is considered as free when any of the resources controlled by configuration registers within this cell are not part of an implemented design. Any CLB cell which does not fulfil this requirement is considered as taken.

Each CLB contains four look–up tables (LUTs) organised into two slices.

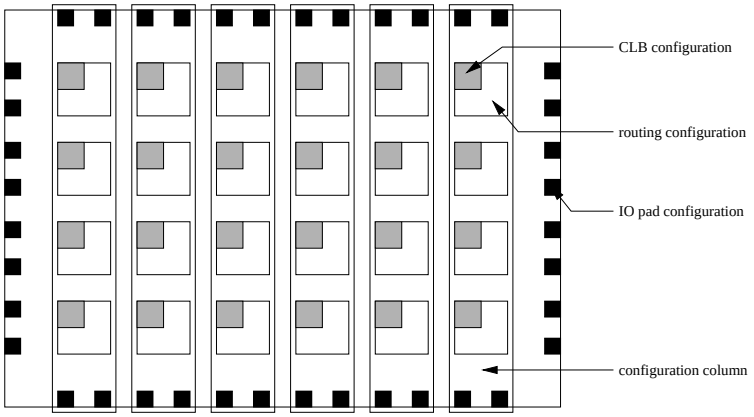

Figure 16: Example of an FPGA CLB array seen from the configuration data architecture point of view.

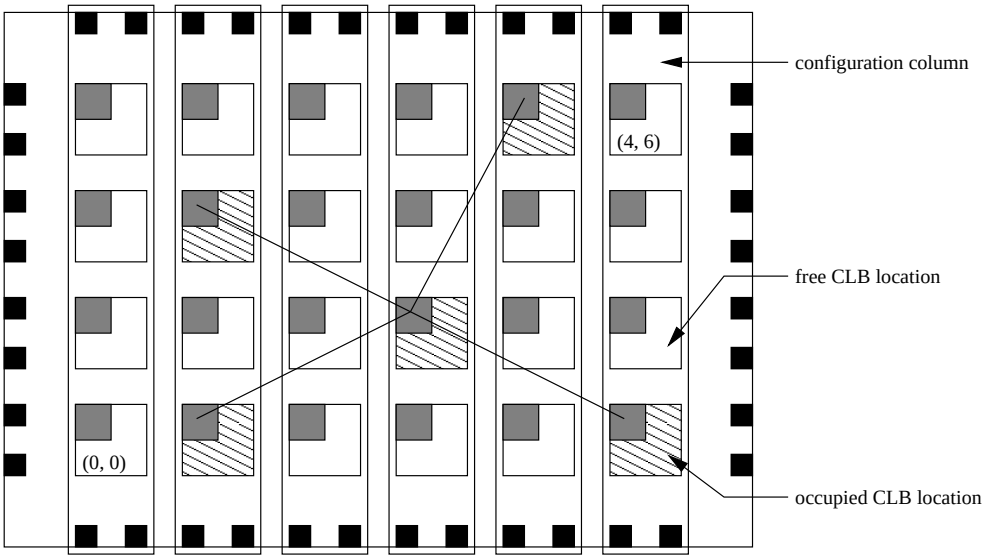

Figure 17: Example of implemented design considered as taken/free CLB cells.

Each CLB column is controlled by 48 configuration frames. Each configuration frame controls part of top and bottom IO blocks and part of each CLB within the column. Left and right IO blocks are controlled by a dedicated set of configuration frames similar to distributed RAM blocks. Full details of Xilinx Virtex XCV50 can be found in the data sheet (Xilinx, 2001).

3.3.1.2 Bitstream Comparison Tool

To determine the frames utilisation characteristic of each XCV50 bitstream, a Bitstream Comparison Tool (BCT) was developed in C using Kdevelop (KDevelop, n.d.) running under Mandrake Linux OS on dual Xeon 2GHz/512kB 1GB RAM Dell PC. BCT takes two bitstreams, scans through their content and and performs frame to frame comparison. BTC delivers set of statistics showing percentage of similar frames and distribution of frames differences categorised into difference groups, each covering a range of ten bits difference. See Appendix for main.c

3.3.1.3 Java Place and Route

Using the Xilinx Virtex FPGA architecture together with the JBits tools a new placement and routing framework was developed with JBuilder compiler running on dual Xeon 2GHz/512kB 1GB RAM Dell PC with Debian Linux operating system. The developed process takes a Xilinx Virtex bitstream as a design source and extracts the netlist to perform placement and routing according to the designer specified rules. Placement and routing results are saved as a new output bitstream. In this way the original and modified bitstreams can be compared to measure the size reduction ratio. The detailed framework architecture is presented in Figure 18.

After loading a bitstream JBits routines are used to scan through all the routing resources to identify the nets. Any net terminal is identified as a

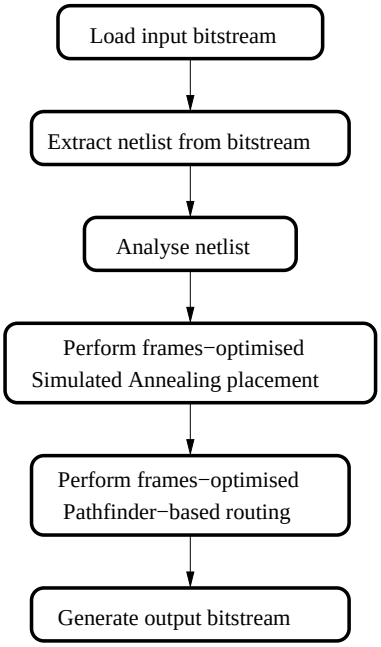

Figure 18: Java Placement and Routing framework data flow.

block.

As JBits does not provide any placement and routing routines, these have been implemented based on VPR (Betz et al., 1999) as a reference algorithms before implementing frames optimised placement and routing.

3.4 Simulated Annealing Placement

As described in the section 2.10.2 simulated annealing has been successfully adapted to FPGA placement. Simulated annealing placement has been described by the pseudo–code of Figure 19.

3.4.1 Initial Placement

Initial placement determines allocation of blocks on the FPGA layout at the beginning of simulated annealing placement. According to the approach

```
S = RandomPlacement();
T = RandomTemperature();
R_limit = InitialR_limit();

while(ExitCriterion() == False){    /* "Outer loop" */
while(InnerLoopCriterion() == False){     /* "Inner loop" */
S_new = GenerateViaMove(S,R_limit);
ΔC = Cost(S_new) – Cost(S);
r = random(0,1);
if(r<e^-ΔC/T){
S = S_new;
}
} /* End "inner loop" */

T = UpdateTemp();
R_limit = UpdateR_limit();
} /* End "outer loop" */
```

Figure 19: Pseudo–code of a simulated annealing placer from (Betz et al., 1999).

used, initial placement can be based on a random block allocation or based on a deterministic method of assigning block locations according to the set of rules applied. It can also be based on the existing placement if simulated annealing placement is applied to existing placement results. The best results have been achieved with random block allocation (Betz et al., 1999).

3.4.2 Placement Cost Function

A cost function is used to evaluate the quality of logic block placement. A typical simulated annealing placement cost function is based on the concept of a net bounding box. The placement cost function proposed by Betz et al. (1999) represents a combination of bounding box and routing channel capacity to determine quality of placement:

$$Cost = \sum_{i=1}^{N_{nets}} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)^{\beta}} + \frac{bb_y(i)}{C_{av,y}(i)^{\beta}} \right] \tag{3.1}$$

Where bb_x and bb_y denote horizontal and vertical spans of net's bounding box, $C_{av,x}(i)$ and $C_{av,y}(i)$ are the average channel routing capacities. For the standard bounding box $\beta = 0$.

The new placement iteration is done by swapping two randomly selected blocks. Freedom of movement depends on the current annealing temperature and is determined by the following equation:

$$R_{limit}^{new} = R_{limit}^{old} \cdot (1 - 0.44 + \alpha) \tag{3.2}$$

Where α represents the fraction of moves accepted, and $1 \leq R_{limit} \leq maxFPGAdimensions$.

Following (Betz et al., 1999) the number of placement iterations within the same annealing temperature is defined by:

$$MovesPerTemperature = InnerNum \cdot (N_{blocks})^{4/3} \tag{3.3}$$

Where the value of $InnerNum$ has been set to 10 (Betz et al., 1999).

Once the number of moves within temperature reaches $MovesPerTemperature$ the temperature changes according to the following formula:

$$T_{new} = \gamma \cdot T_{old} \tag{3.4}$$

Where the value of γ is a function of α and depends on the fraction of accepted moves at T_{old}. Betz et al. (1999) determined values of γ following series of experiments. Results has been presented in Table 2.

Table 2: Temperature update schedule.

α	γ
$\alpha > 0.96$	0.5
$0.8 < \alpha \leq 0.96$	0.9
$0.15 < \alpha \leq 0.8$	0.95
$\alpha \leq 0.15$	0.8

The annealing process terminates when:

$$T < \epsilon \cdot \frac{Cost}{N_{nets}} \tag{3.5}$$

Where the value of ϵ is set to 0.0005 based on (Betz et al., 1999).

3.5 Pathfinder Routing

As described in the previous chapter the Pathfinder router has been successfully adopted to FPGA routing. The Pathfinder routing algorithm can be described by the pseudo–code presented in Figure 20.

```
Let: RT(i) be the set of nodes, n, in the current routing of net (i).

Crit(i,j) = 1 for all nets i and sinks j;

while(overused resource exists) {/* Illegal routing */

  for(each net, i){
    rip-up routing tree RT(i) and update affected p(n) values;
    RT(i) = NetSource(i);

    for(each sink, j, of net(i) in decreasing crit(i,j) order) {
      PriorityQueue = RT(i) at PathCost(n) = crit(i,j)*delay(n) for each node n in RT(i);
while(sink(i,j) not found) {
   Remove lowest cost node, m, from PriorityQueue;
   for(all fanout nodes n of node m) {
      Add n to PriorityQueue at PathCost(n) = Cost(n) + PathCost(m);
   }
      }

for(all nodes, n, in path from RT(i) to sink(i,j) { /*backtrace*/
   Update p(n);
   Add n to RT(i);
      }
    }
  }
  Update h(n) for all n;
  Perform timing analysis and update Crit(i,j) for all nets i and sinks j;
}/* End of one routing iteration */
```

Figure 20: Pseudo–code of the Pathfinder routing algorithm.

3.5.1 Routability–driven Routing Cost

When the routability–driven routing algorithm is used, the cost of using routing resource n when it is reached by routing resource m has been calculated according to the following formula:

$$Cost(n) = b(n) \cdot h(n) \cdot p(n) + BendCost(n, m) \tag{3.6}$$

Where $b(n)$ is a base cost, $h(n)$ is a historical congestion and $p(n)$ represents present congestion and $BendCost(n, m)$ penalises bends when global routing is performed.

Congestion is calculated according to the following equations:

$$p(n) = 1 + max(0, [occupancy(n) + 1 - capacity(n)] \cdot p_{fac}) \tag{3.7}$$

$$h(n)^i = 1, i = 1 \tag{3.8}$$

$$h(n)^i = h(n)^{i-1} + max(0, [occupancy(n) - capacity(n)] \cdot h_{fac}), i > 1 \tag{3.9}$$

Where $occupancy(n)$ is a number of nets claiming to use routing resource n and $capacity(n)$ represents maximum number of nets that can legally use resource n. The values of h_{fac} and p_{fac} define routing schedule and according to Betz et al. (1999) the best router performance has been achieved for $0.2 < h_{fac} < 1$ and $p_{fac} = 0.5$ during the first routing iteration, and then 1.5 to 2 times its previous value in each subsequent iteration.

Routing resource base cost values used by Betz et al. (1999) are presented in Table 3.

3.5.2 Timing–driven Routing Cost

When the timing–driven routing algorithm is used, the cost of using routing resource n when is reached by routing resource m has been calculated

Routing resource type	$b(n)$
wire segment	1
logic block output pin	1
logic block input pin	0.95
net source	1
net sink	0

Table 3: Base costs of different types of routing resource.

according to the following formula:

$$Cost(n) = Crit(i,j) \cdot delay_{Elmore}(n, topology) + [1 - Crit(i,j)] \cdot b(n) \cdot h(n) \cdot p(n) \tag{3.10}$$

Where $Crit(i,j)$ has been defined as:

$$Crit(i,j) = max\left([MaxCrit - \frac{slack(i,j)}{D_{max}}]^{\eta}, 0\right) \tag{3.11}$$

and $delay_{Elmore}(n, topology)$ has been defined as:

$$delay_{Elmore}(n, topology) = T_{d,intrinsic}(switch) + R(n) \cdot C_{total}(n) \tag{3.12}$$

Where $T_{d,intrinsic}(switch)$ represents the delay of a routing switch and $R(n) \cdot C_{total}(n)$ represents the delay of a routing segment (wire) based on its physical electrical parameters.

According to (Betz et al., 1999) the best algorithm performance has been achieved for η equal to 1 and $MaxCrit$ equal to 0.99.

The routing algorithm described above, although very successful at solving the FPGA routing problem is unable to deliver particular configuration data content. This is due to the fact that switch configuration bits location are not taken into account when searching for the route between the net's source and the sink, as illustrated in Figure 21.

Reference Simulated Annealing placement starts with random blocks allocation and then performs series of block swaps to find optimal solution.

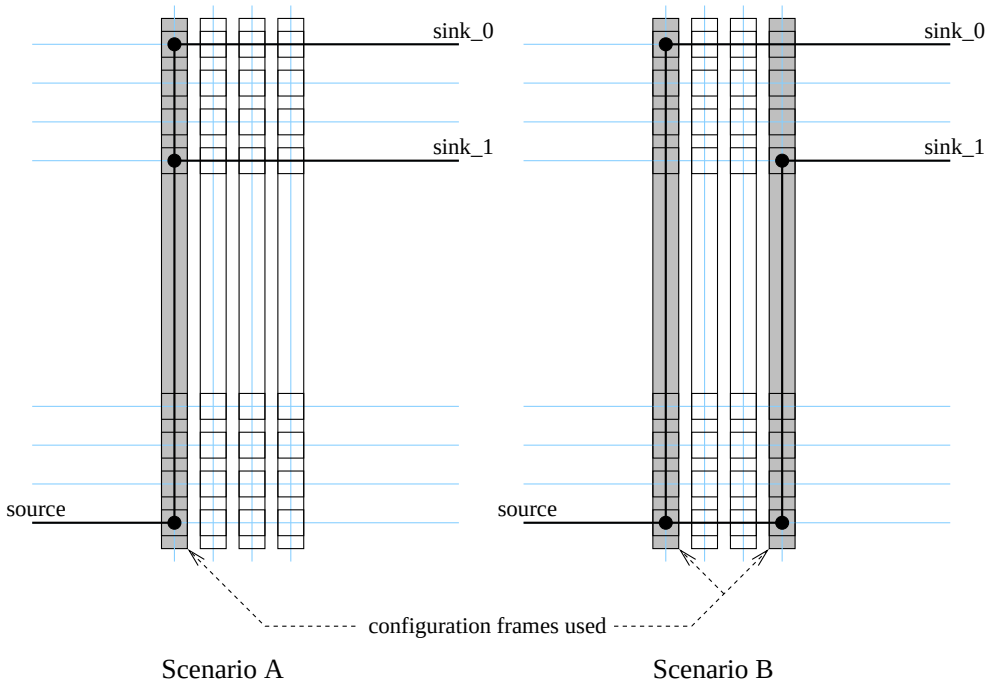

Figure 21: Routing implementation and its impact on configuration data content.

After each iteration the cost is calculated based on each net bounding box. Depending on the cost function change and acceptance probability, the current placement can accepted or rejected. Number of moves per temperature has been based on the equation used by Betz et al. (1999). As the temperature cools down blocks freedom movement changes from FPGA boundary at the beginning upto just a neighbouring location at the end of the annealing process. Annealing is completed when temperature reaches exit temperature.

With a typical simulated annealing placement algorithm the candidate for relocation is randomly selected. Then its destination location is selected within the area determined by the 'movement freedom factor' – a function of the annealing temperature. At the beginning of the annealing process, this freedom factor allows relocation within the entire FPGA area, and gradually shrinks to allow moves only to neighbouring locations at the end of the annealing process.

In order to find optimal placement each combination of placement algorithm evaluated the cost function to determine the quality of proposed placement.

Bounding box base cost function used initially in JPR has been based on VPR's bounding box based cost function (Betz et al., 1999) and has been presented below:

$$Cost = \sum_{i=1}^{N_{nets}} [bb_x(i) + bb_y(i)] \tag{3.13}$$

where bb_x and bb_y denote horizontal and vertical spans of net's bounding box. The cost function from Equation 3.13 when applied to the example from Figure 11, will give the same bounding box, therefore it cannot be directly applied for shape–oriented placement.

3.5.3 Benchmarks Suite

To determine the quality of proposed placement and routing approaches representative test circuits were collected. As a consequence of the selected technology, benchmarks had to be compatible with the Xilinx Virtex XCV50 bitstream structure and cover different sizes. Size selection was based on the number of CLB slices used by the circuit. Benchmark design sizes were divided into ten size categories (1% to 10%, 11% to 20% and so on upto 91% to 99%). Each size category was represented by at least five circuits to cover different block/routing ratios and different I/O pins requirements. A set of real–world FPGA benchmarks was developed using Verilog and VHDL sources available from ITC99 benchmark suite and free IP cores available at www.opencores.com. The LeonardoSpectrum compiler (Mentor Graphics) was deployed to compile circuit sources to Xilinx Virtex XCV50 netlists. Bitstreams were generated for each netlist using Xilinx ISE, executing on a dual Intel Xeon 2GHz/512k Dell PC under MS Windows 2000 SP4. In both packages default settings were used (e.g. commercial temperature).

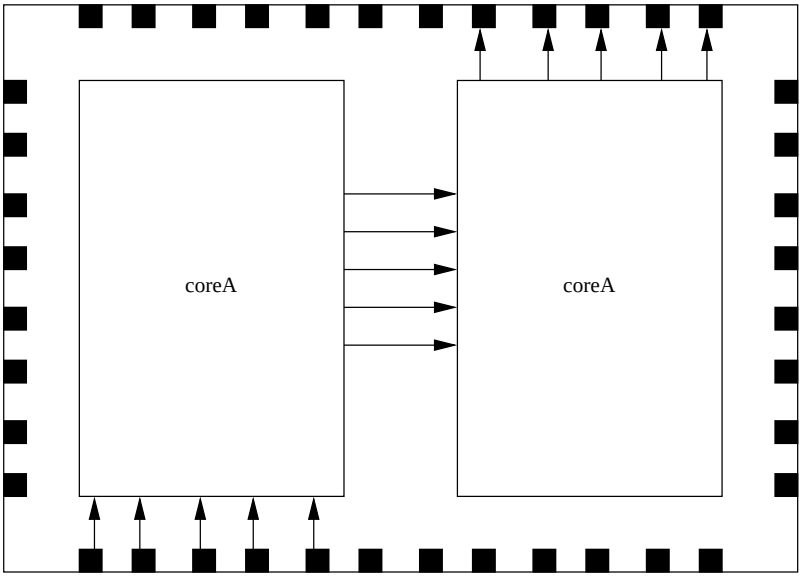

Figure 22: Example of serial cores implementation: coreA_chain_x2.

Each circuit was assigned to the appropriate size category according to the number of CLB slices used. As the aim was to have at least five circuits in each of the size category, a group of custom circuits were created using multiple instances of smaller circuits to fill up certain size categories. Base circuits with a similar number of inputs and outputs were connected as in a serial mode, where output of the first core is feeding input of the second core, while others were connected in parallel mode, where single cores are only sharing ENABLE and RESET signals, where appropriate. An example of the topology utilised is presented in Figure 22 and 23.

The list of benchmark prototypes used based on those described by Corno et al. (2000) are presented in Table 4.

The list of benchmark prototypes used from *http://www.asics.ws* (2005) is presented in Table 5.

The list of benchmark prototypes used from Opencores (2005) is presented in Table 6.

The list of benchmark prototypes from HLSynth95 is presented in Table 7.

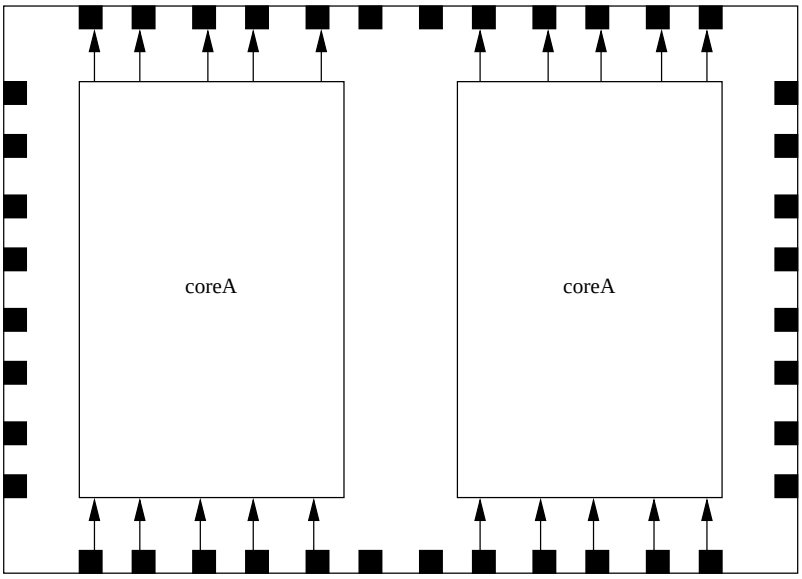

Figure 23: Example of parallel cores implementation: coreA_coreA.

Table 4: FPGA Benchmark Prototypes from ITC99 Benchmark Suite.

Name	Size [%]	Description
b01	1	FSM comparing serial flows
b02	1	FSM that recognises BCD numbers
b06	1	Interrupt handler
b03	2	Resource arbiter
b09	2	Serial to serial converter
b08	3	Find inclusions in a sequence of numbers
b10	4	Voting System
b07	6	Count points on a straight line
b13	6	Interface to meteo sensors
b11	8	Scramble string with variable cipher
b04	11	Compute min and max
b05	15	Elaborate the content of a memory
b12	30	1 Player Game (guess a sequence)

Table 5: FPGA Benchmark Prototypes from www.asics.com

Name	Size [%]	Description
wbif_68	1	Motorola DragonBall/68K Wishbone
sasc	2	Simple Asynchronous Serial Controller
ss_pcm	6	Single Slot PCM Controller
usb_phy	10	USB 1.1 PHY
i2c	15	I2C Master Controller
ata	30	ATA/ATAPI Controller

Table 6: FPGA Benchmark Prototypes from www.opencores.com

Name	Size [%]	Description
cf_ldpc	9	Low Density Parity Check
cf_fp_mul_c_3_4	10	Floating Point Multiplier
cf_interleaver_6_8	13	Memory Interleaver
cf_fir_3_8_8	23	FIR Filter
cf_fp_mul_c_5_10	34	Floating Point Multiplier
cf_fp_mul_p_5_10	43	Floating Point Multiplier
cf_fir_7_16_8	86	FIR Filter
cf_interleaver_6_64	86	Memory Interleaver

Table 7: FPGA Benchmark Prototypes from HLSynth95.

Name	Size [%]	Description
barcode	8	Barcode Reader
fmu	91	Multiplier

As a part of the developed benchmark suite, an empty bitstream for a blank design has been created using '-bitgen' option in the FPGA Floorplanner (part of Xilinx ISE 5.1). Such an empty bitstream represents the FPGA registers state after power–up, with default SRAM configuration memory content.

3.5.4 Bitstream Size Reduction at P&R

Once the JPR framework had been developed and the set of suitable benchmark designs established evaluation of bitstream size reduction during placement and routing became possible. As both placement and routing contribute to the bitstream content, and routing also depends on the results of placement the entire problem needed to be split into two separate subproblems: (i) bitstream size reduction during placement and, (ii) bitstream size reduction during routing. Each is described in detail in the following sections.

3.5.4.1 Bitstream Size Reduction During Placement Approach

Before the evaluation of any developed algorithms was possible it was necessary to define the measurement criteria that would be used to assess their performance.

With the selected technology and framework established it was apparent that column–based (frames-optimised) placement was the best choice, due to the fact, that configuration frames in Xilinx Virtex FPGA span horizontally through entire CLB columns. To calculate the minimum number of columns required to place the design, equation 3.14 was used.

$$Columns_{min} = \frac{N_{CLB\_slices}}{CLB_{column\_capacity}} \tag{3.14}$$

Where N_{CLB_slices} represents number of CLB slices involved in the design and $CLB_{column\_capacity}$ is the capacity of a single column of CLB slices in the

FPGA. For the selected Xilinx Virtex XCV50 the value of $CLB_{column\_capacity}$ is 16.

To achieve column–based placement several modifications to the simulated annealing placement algorithm were evaluated, first using a simplified FPGA model, and then tested on JPR with the selected benchmark designs. Modifications included different initial placement rules, selection of placement cost functions and next step rules. Initial evaluation allowed maximum algorithm freedom by focusing on optimising the number of CLB columns used. Thus, IO pins were set to floating to allow the placement algorithm to place them in any appropriate location. For each algorithm variation the number of configuration columns was determined by calculation to justify the quality of the approach tested.

3.5.4.2 Bitstream Size Reduction and Routing Problem

Once successful placement had been achieved each design was routed using the standard Pathfinder–based routing algorithm incorporated to JPR from VPR. The main task for the router was to validate placement results by providing successful routing. As a result of successful routing an output bitstream was then generated. Using a net tracing facility available in Xilinx JBits, a search for the longest path was performed on the set of input and output bitstreams to compare timing performance. The achieved bitstream size reduction ratio was then defined by equation 3.15.

$$SizeReductionRatio = \frac{N_{STD}}{N_{FO}} \tag{3.15}$$

where N_{FO} and N_{STD} represent the number of frames obtained using Frames Optimised Placement and Routing (FO) and using standard simulated annealing Placement and Routing (STD), respectively. In both cases, the number of frames used was established with BTC by comparing the relevant bitstream against an empty bitstream representing an empty design reference.

A simple timing model for the FPGA architecture was used in the experiments, which represents the delay of each net as the number of configuration switches used by the net. This simplification was considered reasonable as the delay introduced by a routing wire is significantly smaller when compared to a routing switch delay. Based on this assumption the following equation has been used to calculate the Critical Path Change:

$$CriticalPathChange = (\frac{T_{FO}}{T_{STD}} - 1) \cdot 100\%, \tag{3.16}$$

where T_{FO} and T_{STD} represent the number of routing switches on the critical path after FO P&R and VPR P&R respectively.

3.5.4.3 Placement and Routing Algorithms Critical Evaluation

Results obtained from the previous stage, subsequently led to determination of further size reduction–oriented placement and routing algorithm improvement. It was evident that the development should proceed along two main directions: (i) improving performance by minimising the number of placement iterations and, (ii) improving routing by reducing the longest path whilst keeping the design routable.

Further algorithm improvements were evaluated using the same JPR framework and using BCT to determine the quality of size reduction ratio achieved.

3.5.4.4 Problem of IO Pins Allocation

To determine the affect of IO pin allocation on the bitstream size reduction ratio a set of benchmarks were evaluated using JPR with fixed and floating IO pins.

3.5.4.5 Scalability and Feasibility for Other Technologies

Once the developed placement and routing solution had been successfully validated with Xilinx Virtex technology, the scalability of the solution and its feasibility across different FPGA technologies was considered. This included possible changes to configuration interface architecture in order to further improve bitstream size reduction, during placement and routing.

3.5.5 The Context Similarity Optimisation Problem

The vast majority of currently available EDA tools and methodologies focus on a single context optimisation, therefore they are unable to benefit from multi-context similarities.

To resolve this problem, placement and routing for reconfigurable systems should process multiple contexts together. One such approach employs a master/slave arrangement also termed semi–parallel P&R. In master/slave approach, the master context has to be selected and placed and routed first. Any other context will have to be placed and routed in such a way, that they share a subset of the bitstream of the master context.

The decision as to which context should be the master is crucial to the overall bitstream sharing ratio, so the decision criteria have to be carefully specified. Furthermore, the placement and routing cost function has to be based on bitstream comparison in order to determine the quality of the solution. Consequently, a single placement iteration will include fully–detailed routing information and bitstream generation. However, as typically thousands of iterations are performed during placement, such a method is unacceptable due to the massive compilation time overhead.

An alternative proposed solution is to perform placement and routing on multiple contexts at the same time, and measure similarities during P&R iterations as a part of an overall cost function. To obtain the exact similarity figure based on bitstream comparison after each placement iteration all

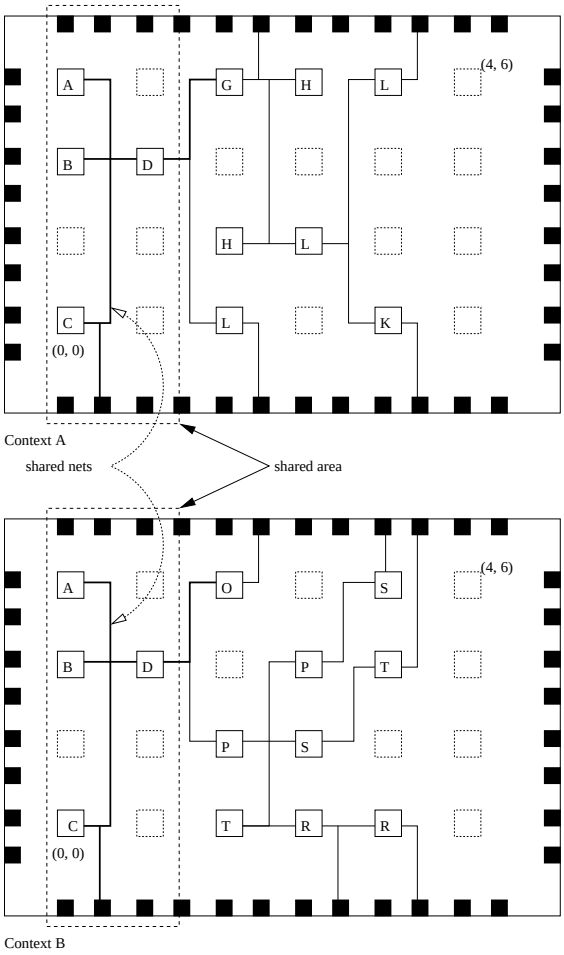

Figure 24: Placement of the design using shared and non–shared area approach.

contexts would have to be routed. This would make the approach unsuitable due to problem complexity.

Alternatively, the placer can utilise additional infomation about block similarity and divide an FPGA array into shared and non–shared area. The shared area will host only the blocks shared between contexts, whilst all the remaining blocks will be placed in the non–shared area. Within the shared area it is important to place the same blocks in each design in exactly the same location in each implementation so the entire column configuration can remain the same. An example of such an approach is presented in Figure 24

3.6 Designs Similarity Problem

As multi-context design can be implemented in such a way that contexts share a portion of common configuration, a key idea is to split each context into shared and non–shared parts. The place and route implementation can then benefit from design similarities.

Standard single–context placement and routing algorithms operate a list of blocks and nets, treating all the blocks the same way. In order to obtain partially shared bitstreams P&R algorithms need extra information to distinguish between shared and unshared sections of the circuitry. This requires that earlier steps in the design flow need additional processing steps to obtain an overall optimisation that results in maximally shared logic and routing infrastructure.

3.6.1 Multi-Context Placement and Routing Approach

A solution to the problem of multi–context placement and routing was developed based on the lessons learned during the bitstream size reduction study.

Initially the entire problem was reduced to the placement and routing of two unrelated design contexts oriented purely on configuration data sharing. To indicate the problem and the need for improvement, benchmark bitstreams were grouped into pairs and compared. As expected the level of similarity was very low on the order of a few percent.

3.6.2 Multi-context Benchmarks

To test the feasibility of the developed solution for bitstream size reduction a selection of circuits were collected. The selection criteria for these circuits was not constrained and real–world circuits, covering a spectrum of different CLB sizes, representing different functionalities, and different similarity ratios were

used. They were compiled as XCV50 bitstreams using the author's JPR tool.

Selected circuits were coupled into pairs to simulate run–time reconfiguration from one circuit to another. As run–time reconfiguration can be applied to any two circuits design similarity and size were selected as the primary focus to get pairs covering the most common cases.

Using the JPR bitstream extraction feature, netlists from each bitstream pair were compared against each other to find and identify CLB blocks with the same configuration. Designs similarity was the key criteria for placement and routing quality measurement. It was assumed that shared CLB blocks would share configuration data.

3.6.3 Methodology

A set of experiments were performed to evaluate the feasibility of simultaneous multi–context placement and routing. The number of frames in the bitstream delivered by JPR have been compared with an empty bitstream (configuration data for blank design) provided by JBits in order to establish the number of frames required to implement the design.

To determine the quality of delivered frames improvement ratio the following equation has been used:

$$F = \left(\frac{FS_{SMC} - FS_{SA}}{F_{TOTAL}} \right) \cdot 100\%, \tag{3.17}$$

where F represents Frames Improvement Ratio, FS_{SA} and FS_{SMC} represent the number of shared frames between two designs using Single–Context Placement and Routing (SC) and Simultaneous Multi–Context Placement and Routing (SMC) respectively, and F_{TOTAL} the total number of frames.

An important parameter quantifying any placement and routing approach is critical path delay, which represents the longest path between two logic blocks on the FPGA. Critical path determines maximum frequency of a

design. As precise timing requirements were not available for each benchmark circuit, the impact of FO P&R on timing was established by comparing critical path delays of the circuit placed and routed using both FO P&R and VPR P&R.

The simplest way to compare two nets is to count the number of routing switches involved. This is possible as the delay introduced by a routing wire is significantly smaller than that of a routing switch delay and can therefore be ignored. The critical path delay was then represented as the number of routing switches required to connect a critical path net. Under this assumption the Critical Path Change was determined using equation 3.18.

$$CP = \left(\frac{T_{SMC}}{T_{SC} - 1}\right) \cdot 100\% \tag{3.18}$$

where CP represents Critical Path Change, and T_{SMC} and T_{SC} represent the number of routing switches on the critical path using SA and SMC respectively.

3.6.3.1 Multi-context Placement Approach

The problem of multi-context placement refers not only to the placement algorithm itself, but also to the way designs are dealt with, whether they are processed separately in master/slave mode, or simultaneously. In the master/slave approach bitstream size reduction can be utilised during placement and routing for a single design, although final results depend significantly on which design is selected as master. When simultaneous placement is used, such a decision is not required, therefore simultaneous placement was selected for ease of evaluation.

A placement algorithm was developed based on Simulated Annealing with enhancements added during bitstream size reduction development. A selection of different initial placements, cost functions and next steps were

tested to achieve an initial multi-context placement algorithm.

3.7 Multi-context Placement Algorithm Evaluation

The placement algorithm required information about cross–design block similarities in order to deliver sharing–optimised placement. If only a netlist is provided, then it is down to the placer to cross–compare designs netlists to distinguish between shared and non–shared blocks.

The number of shared blocks is required to calculate the minimum area to host shared blocks. This decision depends on the configuration interface architecture, as the shared area requires a minimum number of configuration frames. When a column–based configuration interface is used such as that of the Xilinx Virtex technology the placer needs to calculate the minimum number of columns to host shared blocks. In similarity with row–based configuration interface architectures the minimum number of rows had to be determined.

Once the FPGA was divided into two different areas the simulated annealing placement algorithm was applied. In the more general case of multiple designs ($n > 2$) further modifications are likely to be required to ensure optimal results.

3.7.1 Multi-context Initial Placement

With the FPGA divided into two different areas completely random initial placement was inappropriate. However, it can be still used as a two phase process: random placement of shared blocks within a shared area, and random placement of non–shared blocks in the non–shared area. If random placement is applied to the shared area, it can be done only once for an entire project and replicated for every design, to keep the shared area layout exactly the same. Otherwise this area will not be able to share configuration data, as

only an entire column of the same blocks can share configuration – for a column–based configuration architecture FPGA technology.

3.7.2 Multi-context Next Step Criteria

Modifications proposed by the placer should be oriented on bringing improvements to timing and should not interfere with shared and non–shared area division. Simulated annealing was again found to be the most suitable algorithm for placement. Again random selection of the location of the master block was found to be unacceptable due to the placement restrictions imposed by the technology.

As the placement area had been divided into two different sub–areas, different modification rules were applied to each one. Blocks located in a shared area could still be swapped or moved, but they had to fulfil the following requirements:

- any placement modification had to be applied to all designs,
- blocks from shared area could only be swapped with the blocks from a non–shared area only if they both had got exactly the same content.

Blocks located in the non–shared area were relocatable within an entire non–shared are. However, by using bitstream size reduction placement methods the number of frames required to implement a non–shared part of the design was minimised.

3.7.2.1 Multi-context Routing Approach

To validate the placement results obtained from the developed tool, the standard Pathfinder–based router was used to route each design. Both input and output bitstreams were then compared for each approach to find the level of similarity and quantify the longest path delay.

3.8 Multi-context Routing Algorithm Evaluation

As presented in the previous chapter, the aim of routing is to find a path through FPGA routing resources, to provide connectivity for every design net. The location of a net's terminals is specified by the placement procedure. Subsequently, the router has to prove, that every design can be successfully routed with the specific placement given. Therefore routing can be performed either by routing each design separately, or by routing multiple designs simultaneously.

3.8.1 Multi-context Routing Algorithm Criteria

From the router's point of view nets can be divided into three categories depending on their terminal allocation. Figure 24 illustrates typical placement for different net examples. Nets belonging entirely to the non–shared area can be routed using a router used for a single design routing such as a frame–optimised Pathfinder router.

Nets which belong entirely to a shared area and cross–area nets can potentially improve frame sharing if routed in such a way, that they overlap each other as shown in the example of Figure 24.

3.8.2 Multi-context Routing Cost Function

Similar to the routing algorithm described in the previous chapter, the cost function used while traversing the routing resource tree should include the cost of switches used, try to utilise any frames already allocated and avoid using empty frames. To test the performance of the developed placement and routing methodology, a framework based on JBits was created as described in the following section.

3.8.3 Framework Description

To demonstrate the feasibility of bitstream sharing at the placement and routing level, Simulatenous Multi–Context Java Placement and Routing tool (SMC JPR) was developed. This was based on an earlier system produced by the author and discussed in detail in Stepien and Vasilko (2006). SMC JPR performs simultaneous placement and routing on two designs, based on design information extracted from a Xilinx Virtex bitstream using JBits. The SMC JPR design flow is presented on Figure 25.

Placement and routing algorithm have been incorporated from the VPR placement and routing tool developed by (Betz et al., 1999). Of the algorithms available in VPR, the Simulated Annealing placement algorithm and Pathfinder maze–router were selected as the best algorithms used for investigating novel placement and routing strategies for the present study.

The developed Multi–Design Java Place and Route software takes two bitstreams as a design entry and performs a series of comparisons to extract design similarities. Based on these results, the placement space is divided into two areas: one for shared design blocks and the other for all the remaining designs blocks. During initial placement similar blocks were placed in exactly the same location in both designs. The reason for this was that by having an entire column of the same blocks in both designs, the probability that frames of the column with shared blocks would be similar was maximised. Block swaps or moves within a shared area were allowed only if it did not influence the content of shared columns.

The router was also modified to avoid using empty frames but to use any area which had already been used by the placer.

3.8.3.1 Placement & Routing Algorithms Critical Evaluation

Results obtained in the previous stage resulted in identification of further potential placement and routing algorithm improvement, such as: increased

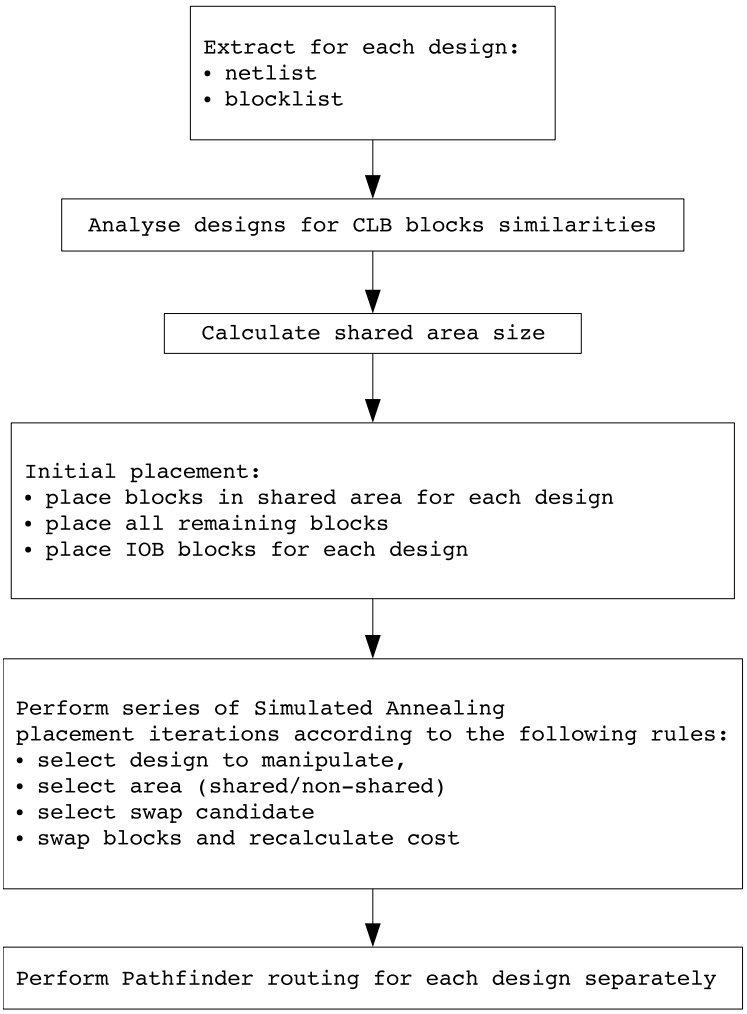

Figure 25: Simultaneous Multi–Design Placement and Routing – Design Flow.

performance by minimising the number of placement iterations; an improving level of similarity utilisation by the algorithm; and optimal routing by minimising longest path length whilst keeping the design routable. Further algorithm improvements were evaluated using the same JPR framework and using BCT to determine the quality of the developed multi-context placement and routing technique.

3.8.3.2 Problem of IO Pins Allocation

Separate consideration has been given to the issue of IO pin allocation. As the proposed P&R solution needs to demonstrate its practicality IO pin allocation is usually determined by PCB design prior to FPGA design development. To determine IO pin allocation on the bitstream size reduction ratio, a set of benchmark designs have been placed using JPR with fixed and floating IO pins respectively. Results obtained were used to inform IO pins allocation on multi-context placement and routing.

3.8.3.3 Scalability and Feasibility for Other Technologies

Once the developed multi-context placement and routing solution had been successfully validated on the Xilinx Virtex technology, the scalability of the solution and its feasibility across different FPGA technologies was evaluated. These included possible changes to configuration interface architecture so as to further improve bitstream size reduction during placement and routing.

3.9 Summary

Due to the lack of available tools and methodologies to investigate placement and routing impact on configuration data size, new complete placement and routing framework has been develop. Based on JBits it offers placement and routing environment for Xilinx Virtex FPGAs. Reference placement and

routing algorithms (Simulated Annealing placer and Pathfinder router) has been implemented based on the work of Betz et al. (1999) and their VPR solution.

As these algorithms are not supporting configuration data size reduction, new placement and routing methodology has been introduced to deliver configuration data size reduction. Quality measure criteria have also been established to determine the quality of proposed methodology.

For better understanding of the impact of placement and routing on reconfiguration latency the problem has been divided into two sub-problems: (i) configuration data size reduction for single-context design and (ii) configuration data sharing for multi-context design. The aim is to investigate impact of placement and routing on configuration data size first and then use the results in multi-context placement and routing.

Chapter 4

Bitstream Size Reduction Implementation

This chapter describes detailed benchmark bitsreams characteristics to verify the need for improvement. Series of modifications applied to the JPR to achieve desired bitstream size reduction have been described in detail to

4.1 Benchmark Bitstreams Analysis

Benchmark bitstreams have been placed and routed using JPR with standard placement and routing algorithms described in the section 3.4 and section 3.5. Each bitstream generated by JPR has been compared against an empty bitstream to establish number of frames used by each bitstream. Results are presented in Tables 8 and 9.

The results presented in Table 8 and Table 9 indicate that circuits representing CLB size from 40% upwards, are using 100% of bitstream frames. Also circuit b04 representing 11% CLB size requires 57% of configuration frames to be loaded.

To illustrate the problem of bitstream redundancy detailed analysis of

Table 8: FPGA Benchmarks Summary – size 1% – 40%.

Name	CLB Size [%]	Frames Summary [%]	
		Free	Used
b01	1	81	19
b01_b02	1	78	22
b02	1	92	8
b06	1	81	19
wbif_68	1	29	71
b03	2	81	19
b09	2	75	25
sasc	2	71	29
b08	3	66	34
b10	4	60	40
b07	6	80	20
b13	6	42	58
ss_pcm	6	46	54
b11	8	55	45
barcode	8	33	67
cf_ldpc	9	24	76
cf_fp_mul_c_3_4	10	50	50
usb_phy	10	29	71
b04	11	43	57
cf_interleaver_6_8	13	42	58
b05	15	30	70
i2c	15	13	87
b10_chain_x4	17	51	49
b04_chain_x2	22	33	67
b10_chain_x5	22	33	67
cf_fir_3_8_8	23	12	88
b10_chain_x6	26	32	68
ata	30	12	88
b10_chain_x7	30	26	74
b12	30	10	31
b11_chain_x4	32	1	99
b04_chain_x3	33	13	87
cf_fp_mul_c_5_10	34	4	96
b10_chain_x8	35	16	84
b10_chain_x9	39	16	84
b11_chain_x5	40	4	96

Table 9: FPGA Benchmarks Summary – size 41% – 99%.

Name	CLB Size [%]	Frames Summary [%]	
		Free	Used
cf_fp_mul_p_5_10	43	4	96
b10_chain_x10	44	4	96
b04_chain_x4	45	0	100
b05_b12	45	0	100
b10_chain_x11	47	0	100
b11_chain_x6	48	0	100
b10_chain_x12	51	0	100
b04_b05_b12	55	0	100
b04_chain_x5	56	0	100
b10_chain_x13	56	0	100
b11_chain_x7	57	0	100
b05_b05_b05_b05	59	0	100
b10_chain_x14	61	0	100
b12_chain_x2	61	0	100
b10_chain_x15	64	0	100
b04_chain_x6	67	0	100
b12_chain_x2_b11	68	0	100
b11_chain_x9	73	0	100
b12_chain_x2_b11_b07	73	0	100
b12_chain_x2_b05	75	0	100
b10_chain_x18	77	0	100
b04_chain_x7	79	0	100
b11_chain_x10	81	0	100
b10_chain_x19	82	0	100
b10_chain_x20	86	0	100
cf_fir_7_16_8	86	0	100
cf_interleaver_6_64	86	0	100
b04_chain_x8	90	0	100
b10_chain_x21	91	0	100
fmu	91	0	100
b10_chain_x22	95	0	100
b10_chain_x23	99	0	100
b11_chain_x12	99	0	100

Table 10: FPGA Benchmarks Frames Utilisation Summary – size 1% – 20%.

Name	CLB Size [%]	Used Bits per Frame [%]					
		0	1–10	11–20	21–30	31–40	over 41
b01	1	81	19	0	0	0	0
b01_b02	1	78	21	1	0	0	0
b02	1	92	8	0	0	0	0
b06	1	81	18	1	0	0	0
wbif_68	1	29	71	0	0	0	0
b03	2	81	14	5	0	0	0
b09	2	75	22	3	0	0	0
sasc	2	71	23	5	1	0	0
b08	3	66	30	4	0	0	0
b10	4	60	31	9	0	0	0
b07	6	80	13	2	2	1	2
b13	6	42	52	6	0	0	0
ss_pcm	6	46	45	7	2	0	0
b11	8	55	33	4	6	2	0
barcode	8	33	55	11	1	0	0
cf_ldpc	9	24	57	17	2	0	0
cf_fp_mul_c_3_4	10	50	31	16	3	0	0
usb_phy	10	29	51	19	1	0	0
b04	11	43	40	14	3	0	0
cf_interleaver_6_8	13	42	37	19	2	0	0
b05	15	30	40	25	5	0	0
i2c	15	13	54	26	6	1	0
b10_chain_x4	17	51	14	17	13	5	0

bitstream frames content were obtained running BTC on the benchmark bitstreams. Comparison results are presented in Tables 10 through to 12.

The results presented in Tables 10 to 12 indicate that small and medium size circuits are configured mostly by frames carrying only few configuration bits. Circuit b04 representing 11% CLB size requires 57% of configuration frames to be loaded, whereas almost half of of all configuration frames contain only 1 – 10 configuration bits as presented in Figure 26.

Using JPR, a map of final blocks placement allocation results for b04 circuit was obtained and is presented in Figure 27.

Table 11: FPGA Benchmarks Frames Utilisation Summary – size 21% – 60%.

Name	CLB Size [%]	Used Bits per Frame [%]					
		0	1–10	11–20	21–30	31–40	over 41
b04_chain_x2	22	33	27	24	11	4	1
b10_chain_x5	22	33	25	22	14	6	0
cf_fir_3_8_8	23	12	45	27	11	4	1
b10_chain_x6	26	32	13	25	25	5	0
ata	30	12	36	21	20	7	4
b10_chain_x7	30	26	20	17	22	12	3
b12	30	10	31	34	21	4	0
b11_chain_x4	32	1	31	39	24	5	0
b04_chain_x3	33	13	33	30	17	7	2
cf_fp_mul_c_5_10	34	4	29	42	19	7	1
b10_chain_x8	35	16	17	25	29	12	1
b10_chain_x9	39	16	17	25	29	12	1
b11_chain_x5	40	4	14	42	27	11	2
cf_fp_mul_p_5_10	43	4	20	37	24	9	6
b10_chain_x10	44	4	12	35	33	12	4
b04_chain_x4	45	0	22	41	29	7	1
b05_b12	45	0	15	42	26	13	4
b10_chain_x11	47	0	7	36	35	16	5
b11_chain_x6	48	0	11	31	32	17	8
b10_chain_x12	51	0	4	28	41	24	4
b04_b05_b12	55	0	6	30	35	20	9
b04_chain_x5	56	0	10	33	35	17	5
b10_chain_x13	56	0	6	26	30	23	15
b11_chain_x7	57	0	7	22	35	23	13
b05_b05_b05_b05	59	0	4	24	29	26	17

Table 12: FPGA Benchmarks Frames Utilisation Summary – size 61% – 99%

Name	CLB Size [%]	Used Bits per Frame [%]					
		0	1–10	11–20	21–30	31–40	over 41
b10_chain_x14	61	0	4	15	36	29	15
b12_chain_x2	61	0	7	24	37	19	13
b10_chain_x15	64	0	2	14	31	32	21
b04_chain_x6	67	0	5	23	37	25	10
b12_chain_x2_b11	68	0	2	20	35	25	18
b11_chain_x9	73	0	0	10	26	33	30
b12_chain_x2_b11_b07	73	0	1	13	30	32	24
b12_chain_x2_b05	75	0	1	12	27	34	26
b10_chain_x18	77	0	0	7	20	36	37
b04_chain_x7	79	0	2	13	31	32	22
b11_chain_x10	81	0	0	4	21	32	43
b10_chain_x19	82	0	0	5	19	33	43
b10_chain_x20	86	0	0	4	16	28	53
cf_fir_7_16_8	86	0	4	12	24	28	32
cf_interleaver_6_64	86	0	0	5	25	36	33
b04_chain_x8	90	0	1	8	27	33	31
b10_chain_x21	91	0	0	3	13	26	59
fmu	91	0	0	1	12	24	64
b10_chain_x22	95	0	0	2	11	22	65
b10_chain_x23	99	0	0	1	8	22	69
b11_chain_x12	99	0	0	1	6	20	73

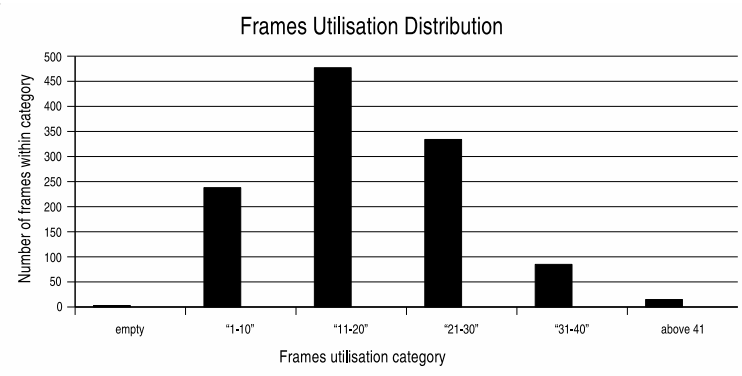

Figure 26: Frames utilisation for a design implementation occupying 45% of device CLBs.

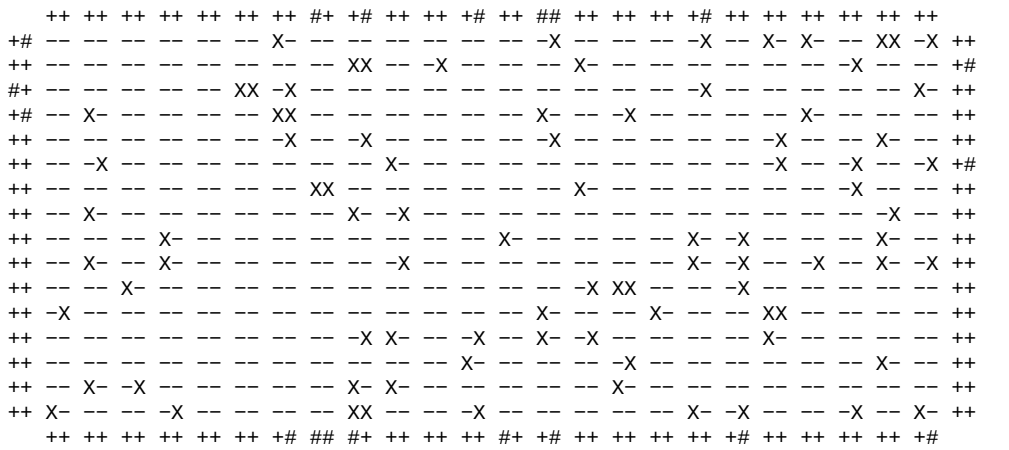

Figure 27: b04 Placement Results.

As presented in Figure 27 b04 circuit occupies 41 CLB columns therefore frames controlling each used column have to be loaded. Looking at frames utilisation chart presented in Figure 26, it is evident, that the majority of frames carry only between 1 and 40 bits to configure the design.

The remaining 284–323 bits are not used, but they still need to be stored and loaded onto the FPGA. This suggests that it should be possible to increase the utilisation of frames while concentrating the design implementation bits into a smaller number of frames. This will, however, require the placement and routing process to be modified to optimise the frame utilisation in addition to other existing design constraints, for example, timing and area utilisation.

4.1.1 Bitstream Size Reduction Implementation

Bitstream analysis presented in section 4.1 proved that placement and routing algorithms used are unable to deliver frames-optimised solution. Using JPR and BCT series of experiments have been performed to find a new set of algorithms modifications able to deliver frames-optimised solution.

4.2 Placement Algorithm Evaluation

At first the placement and routing have been separated in order to obtain column-aligned blocks assignment.

4.2.1 Placement Algorithm Criteria

To obtain an implementation which considers column–based structure of the configuration distribution (Figure 11), the placement algorithm needs to evaluate locations which are more likely to lead to a reduced number of configuration frames. Furthermore, the placement cost evaluation for each iteration should include a factor determining quality in terms of frames utilisation.

4.2.2 Dynamic Location Cost Schedule

To address the problem of column–oriented placement, a new placement algorithm feature has been introduced – Dynamic Location Cost (DLC). DLC determines the cost of each placement location. This cost is based on *ColumnUtilisationCost* calculated after each placement iteration according to the following equation:

$$ColumnUtilisationCost = 1 - \frac{NumOfLocsOccupied}{ColumnCapacity} \tag{4.1}$$

where $NumOfLocsOccupied$ is a number of locations occupied by CLB/IOB blocks within a single column and $ColumnCapacity$ is a maximum number of CLB/IOB blocks which can be placed within a single column. For Xilinx XCV50 $ColumnCapacity$ equals to 18 (16 CLBs and 2 IOBs) (Xilinx, 2000).

The cost of all the locations within a single column is the same and equals to *ColumnUtilisationCost*. As blocks allocation changes every placement iteration, *ColumnUtilisationCost* needs to be re-calculated every time the placement changed.

The initial cost for each location is calculated based on column utilisation after the initial placement. Then after each placement iteration has been accepted, the cost of each location is re–calculated. Placement algorithm should keep the cost of a column low or equal to 1. If $ColumnUtilisationCost$ equals to 1 the column is considered empty and therefore all frames controlling that column do not need to be loaded during FPGA reconfiguration. Any empty columns are excluded from overall column utilisation cost.

4.2.3 Placement Cost Function

An initial modification of the cost function was found to be necessary to penalise a horizontal bounding box span for column–based placement. The following equation proved to deliver column–oriented nets:

$$Cost = \sum_{i=1}^{N_{nets}} [PenaltyFactor \cdot bb_x(i) + bb_y(i)] \tag{4.2}$$

where the $PenaltyFactor$ has been empirically set to 10 following a series of experiments, and bb_x and bb_y denote horizontal and vertical spans of net's bounding box.

Applying cost function from Equation 4.2 to the example from Figure 11 delivers two different costs for scenario A and scenario B.

Evaluation of this modified cost function suggests that it can force the placement algorithm to realise all nets as column–oriented. However, the entire design will still be occupying most of the FPGA as presented by the example shown in Figure 28.

To address the problem presented in Figure 28 the cost function should be factorised by the overall bounding box, in which the horizontal span will be penalised. This indicates that a placement strategy that uses less columns is preferable.

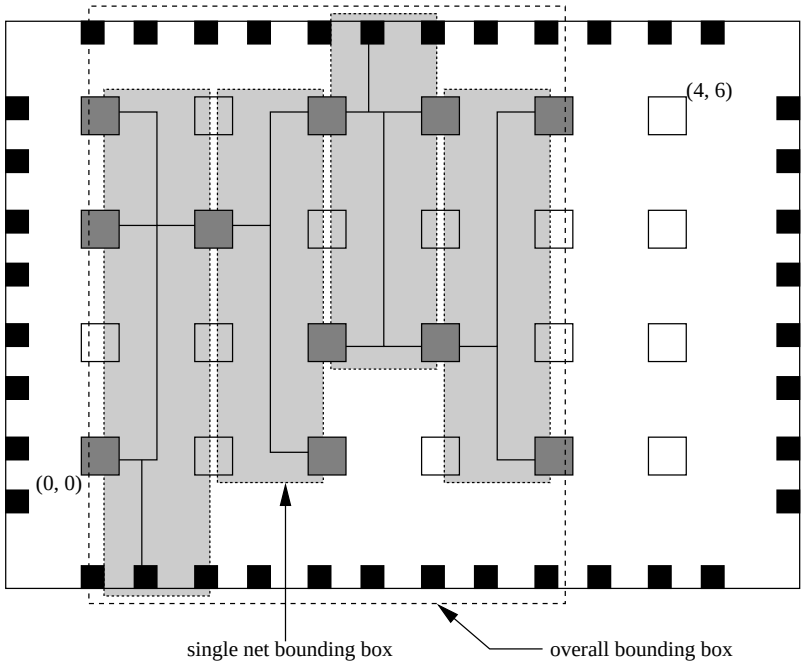

Figure 28: Example of placement result with modified cost function used.

$$Cost = NumOfColumnsUsed \cdot \sum_{i=1}^{N_{nets}} [PenaltyFactor \cdot bb_x(i) + bb_y(i)] \quad (4.3)$$

where $PenaltyFactor$ has been set to 10, $bb_x(i)$ and $bb_y(i)$ denote bounding box length for a single net i and $NumOfColumnsUsed$ denotes the total number of CLB columns used to place the circuit. Adding new constraint to the cost function resulted in prioritising column–based nets with minimum number of columns necessary to obtain frames–optimised placement.

4.2.4 Next Step Criteria

For column–based placement out of any possible moves there is only a limited number which can improve frames utilisation. As the aim of any placement algorithm is to find an acceptable solution within a minimum number of iterations it is important to perform only those moves which offer the

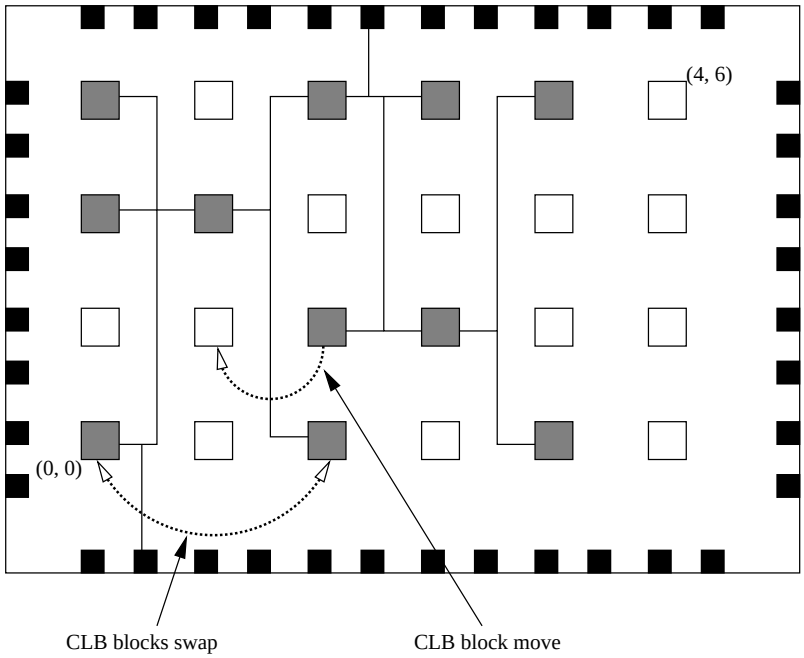

Figure 29: Example of placement with next step move resulting in net wirelength penalty.

reduction possibility in placement cost.

As simulated annealing allows acceptance of moves in a less optimal placement (with a certain probability), it was found to be beneficial to keep this freedom in column–oriented placement strategy. That is why the placement modification step needed to be divided into two types: block move and block swap. A block move determines column utilisation while block swap determines timing constraints via the net bounding box.

During the block move, the block is relocated into an empty location. Using placement DLC, the block on the most expensive location is moved to the cheapest empty location. Such a step was found to improve the overall location cost and drive an algorithm to place blocks closer to each other in the column–shaped order.

Using only this type of move will introduce a net wirelength penalty, as it allows cases like the one presented in Figure 29.

The reason for this is that once the block is not located at the most expensive location it will not be moved.

Block swap is similar to the typical simulated annealing next step iteration, as it does not change columns utilisation, although it has an impact on a net bounding box.

4.3 Routing Algorithm Evaluation

Once placement has been completed, the aim of frames–optimised router is to successfully route the design using minimum number of configuration frames.

4.3.1 Routing Algorithm Criteria

Any routing algorithm suitable for bitstream size reduction needs to be aware of which switches need to be configured to implement each net. As each switch is associated with a configuration frame, and each frame has a cost depending on its utilisation after placement, the router should use 'cheaper' switches when looking for the cheapest connection. Figure 30 shows an example of how routing influences the number of frames and bitstream size.

4.3.2 Routing Cost Function

The Dijkstra algorithm which is used in a search for the best connection between blocks needs a cost associated with every routing resource to find the shortest path. This cost is usually calculated based on the wirelength or delay caused by the single routing resource. The cost of each path represents the sum of all routing resources building up the path. To incorporate frames utilisation to the routing process new connection cost function has been introduced according to the equation 4.4.

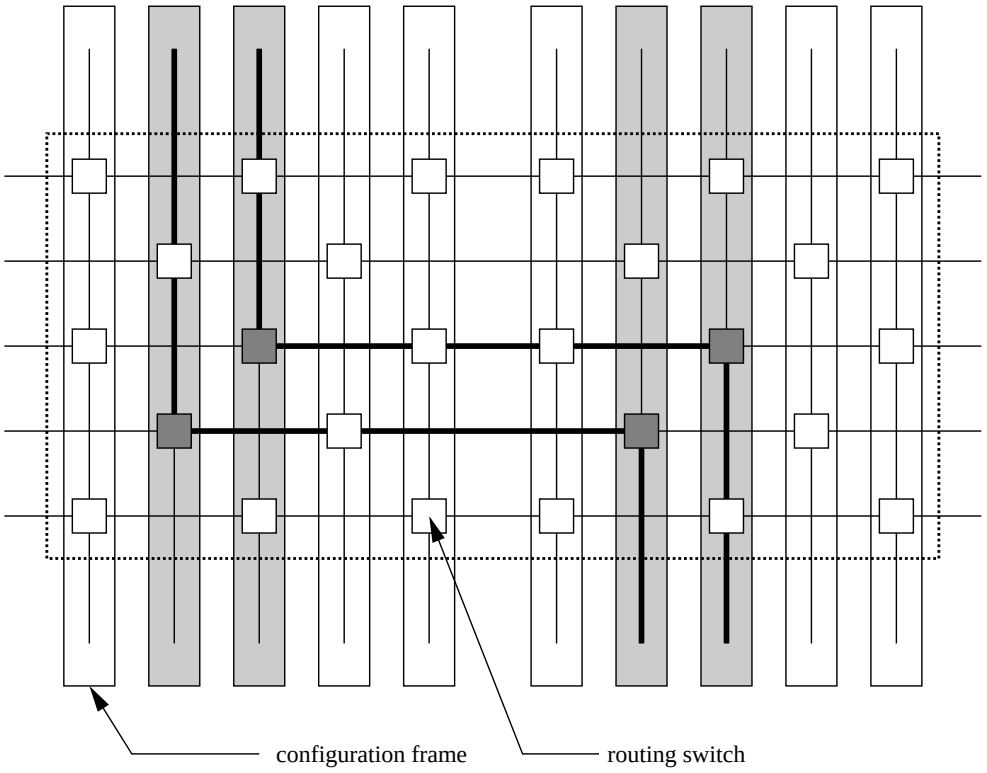

Figure 30: Example of routing search and its impact on bitstreams size.

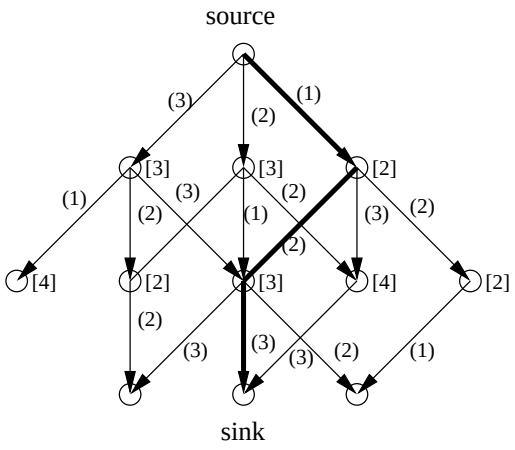

Figure 31: Example of routing search and its impact on bitstreams size.

$$ConnectionCost_{(A,B)} = \sum_{i=A}^{i=B} [Cost_{wire}(i) + Cost_{switch}(i, i+1)] \qquad (4.4)$$

where $Cost_{wire}$ for each routing resource is calculated using equation 3.10 and $Cost_{switch}$ connecting two routing resources i and $i+1$ i based of the current frames utilisation cost associated with this switch. Frame utilisation cost has been calculated according to the equation 4.5:

$$FrameUtilisationCost(i) = \left[1 - \frac{NumOfBitsSet}{FrameLength}\right] \qquad (4.5)$$

where $NumOfBitsSet$ denotes number of bits set within the frame and $FrameLength$ denotes total number of configuration bits available within the frame. For XCV50 $FrameLength$ equals 324 (Xilinx, 2001).

With the cost function from equation 4.4 the cost of two paths presented on Figure 31 is different.

4.4 Timing Analysis

As benchmark circuits are not characterised by any timing constraints a timing comparison can be done by comparing the longest nets between the

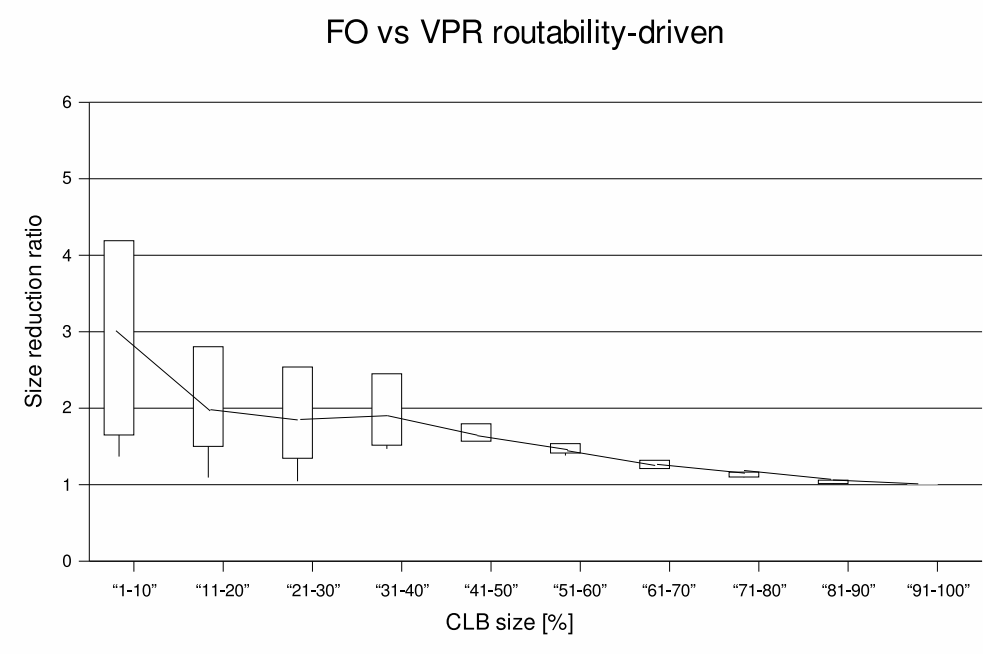

Figure 32: Size Reduction Ratio: FO vs STD routability–driven P&R.

original (STD) and frame–optimised (FO) bitstreams.

4.5 Experimental Results

To test feasibility of the new approach FO P&R was performed on the set of benchmarks with maximum flexibility given to the algorithm and no pin constraints. Cost functions used in placement and routing were purely frame optimisation oriented without any other constraints. Figure 32 presents bitstream size reduction ratio as a function of design complexity represented as percentage of FPGA CLB size. FO P&R and STD P&R were both routability driven with floating pins option enabled.

The results presented in Figure 32 show, that it is possible to obtain a frame–optimised bitstream that is better than that offered by currently available P&R approaches. Not surprisingly, the achieved size reduction ratio

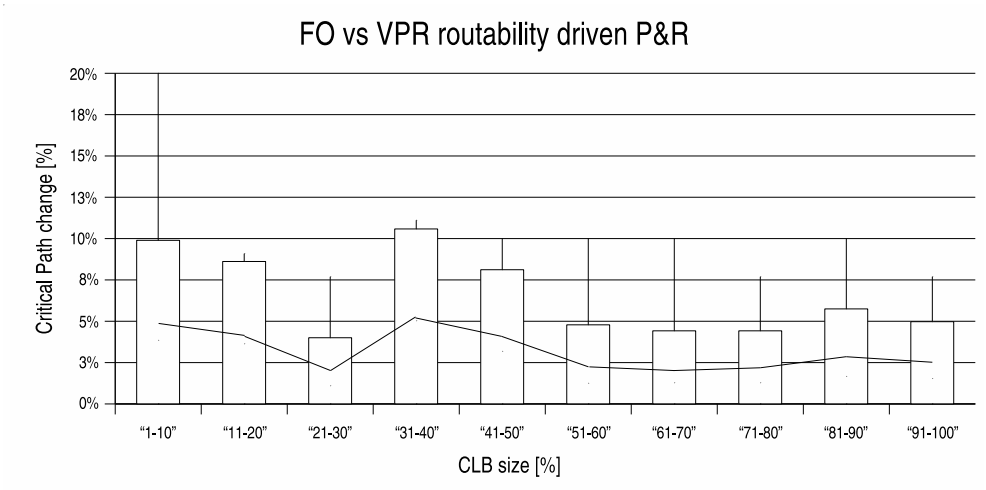

Figure 33: Critical Path:FO vs STD routability–driven P&R.

is considerable for small circuits. In this case up to 5.5 times less frames can be used to implement the design. However, the results also demonstrate that an approach using configuration frame–based optimisation during placement and routing can provide significant benefits for circuits up to 60% FPGA usability.

To check the impact of FO P&R on the circuits timing, the change in critical path has been calculated. Results are presented in Figure 33.

Further analysis indicated that the critical path in a frame optimised design is longer than when using STD P&R. An average decrease of 8% was obtained with a maximum timing decrease of up to 20% for very small circuits.

To improve timing performance of FO P&R timing–driven algorithms have been employed. Results for varying size reduction ratios as presented in Figure 34 and Figure 35 indicate the critical path change obtained over the range of circuit complexity.

Results for the timing–driven version of FO P&R show, that a size reduction ratio up to 4.8 can be achieved, although not as high as for routability–driven FO P&R. Critical path delay analysis presented in

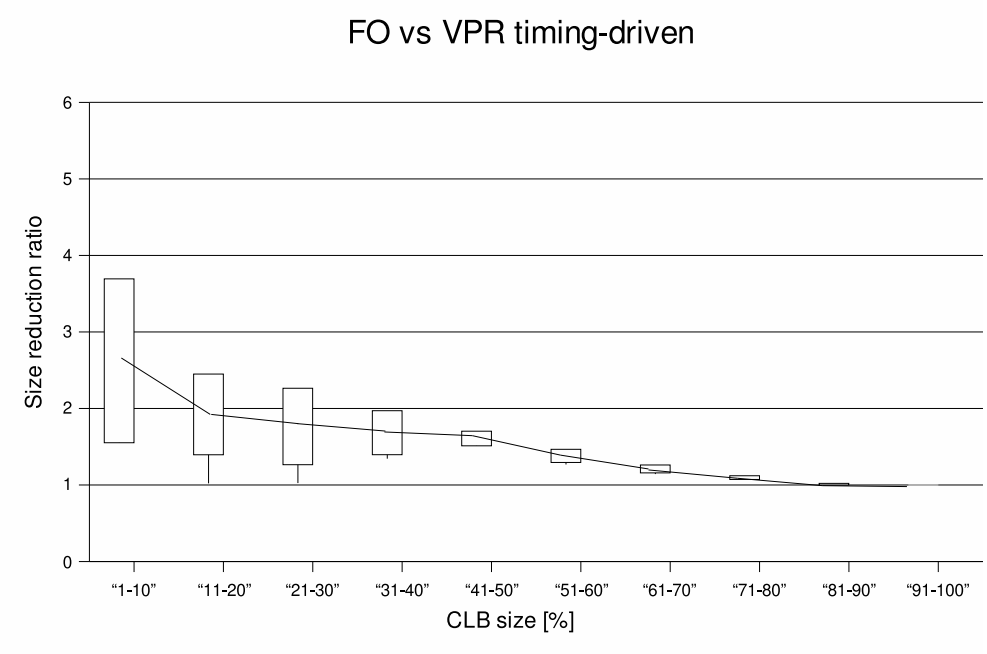

Figure 34: Size Reduction Ratio: FO timing–driven vs STD timing–driven P&R.

Figure 35 shows, that critical path delay increased on average 15% with a maximum increase up to 33% for small circuits.

4.6 Pins allocation

Analysis of the results presented in Figures 32 to 34 suggests that for small designs (up to 40% FPGA usability) size reduction ratio varies from 1.2 to 2.8 and in the smallest size circuits group, the range is even bigger from 1.4 to 5.5. The reason for some of the circuits having low size reduction ratios relates to the problem of pin allocation and the Xilinx Virtex configuration architecture. Each FPGA IO pin is controlled by an IO block. IO blocks located on the left and right side of Xilinx Virtex FPGA are controlled by a separate set of 96 frames (Xilinx, 2000). IO blocks located on the top and bottom size of the FPGA chip are controlled by the same frames which

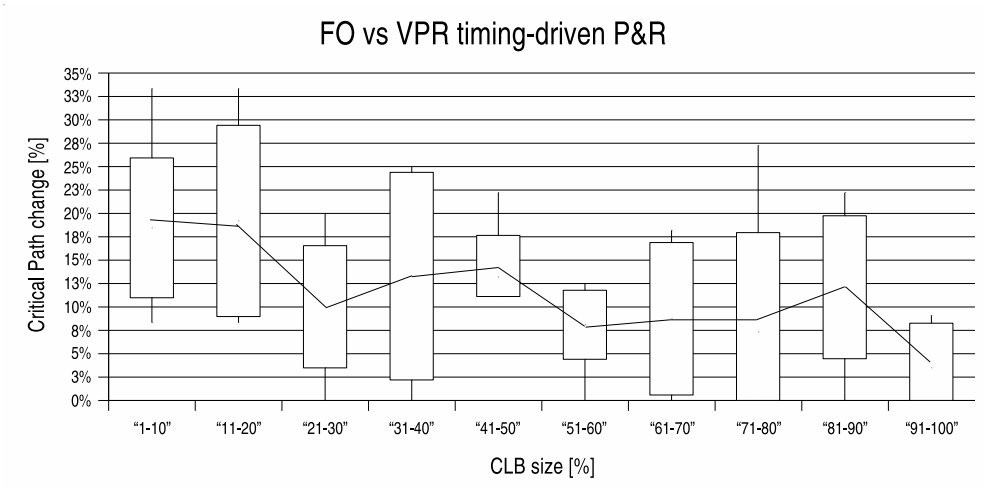

Figure 35: Critical Path: FO vs STD timing–driven P&R.

control CLB content and routing. For small circuits if the ratio of IO blocks to CLB blocks is close to unity and IO pins are allocated to the top and the bottom of the FPGA situation presented in Figure 36 was found to occur.

The situation shown in Figure 36 (top diagram) presents placement result for pins fixed to the top and bottom locations. Consequently, all 52 columns (single column representing 24 frames per FPGA location) have to be configured despite any CLBs relocation. The same circuit placed and routed with the floating pins option enabled (i.e it is left to the placement algorithm to determine suitable IO allocation) requires only 8 columns to configure the FPGA.

4.7 Summary

Novel placement and routing approach presented in this chapter demonstrated its ability to deliver frames optimised solution. The size of the configuration bitstream can be reduced considerably. As a result of bitstream size reduction, storage requirements, reconfiguration latency and power consumption can be improved. The size reduction ratio depends on the complexity of the design

```
Fixed Pins Placement Result:

   ++ ++ ++ #+ +# ## ## ++ ++ ++ ++ ++ #+ ++ ++ ++ ++ #+ ++ +# ++ ++ +# ++
+# -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ++
+# -- -- -- -- -X -- X- X- -- -- -- -- -- X- -X -- -- -- X- -X X- -- X- -- ++
#+ X- -- -- -- -X -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- X- -- X- ++
++ -X -- -- -- -X -- -- -- -- -- -- -- -- -- X- X- X- X- -- -- -- -- X- -- #+
++ -- X- -- -- -- -- -- -- -- -X -- -- -- -X -- -- -- X- -- -X -- -- -X -- #+
++ -- X- X- -- -- -- -X X- -X -- -- -- -X X- -- -- -- -- -- -- -- -- -X -- -- +#
++ -- -- -- -- -- -- -- -- -- -- -X -X -- -- -- X- -- -X -- -- -X -- -- -X ##
#+ -- -- -X -- -- -- -- -- -- -- -- X- -- -- -- -- -- -- -- -- -- -- -- -X -- ++
++ X- -- -- -- -- -X -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X -- -X ++
++ X- -- -- -- X- X- -X X- -- -- -- -- -- -- X- -- -X -- -- -- -- -- -- -- -- ++
+# -- -X -- -- -- X- -- -- -- -- -- -- -- -- -X -- -- -- -- -- -X -- -X -- ++
++ -- -- -X X- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ++
++ -- -- X- -- -- X- -- -X -- X- -- -- -- -- -- -- -- X- -- X- X- -- -- -X +#
++ -- -- -- -X -- -- -- -- X- -- -- -- -- -- -- -- -- -- -- X- -- -- -- -- #+
++ -- X- X- -- -- -- -- -X -- -- -- -- -- X- -- -- -- -- X- -- -- X- -- -- ++
++ -- X- -X X- -- -- -- -- -- -- -- -- -- -- -- X- -X -- -X -- -- -- -- -- ++

   ++ ++ #+ ++ ++ ++ #+ ++ ++ +# #+ ++ ++ #+ +# +# ++ ++ ++ ++ ++ ++ #+ #+
Number of columns used in random placement: 52


Floating Pins Placement Result:

   ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ +# ## ##
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX ##
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX ##
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- XX XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- XX XX XX ##
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- XX XX XX ##
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- XX XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX ##
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#
++ -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -X XX XX +#

   ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ +# ## ##
Number of columns used: 8
```

Figure 36: Placement result for fixed and floating pins: # – used IO block, X – used CLB block.

implementation, which was expected.

Frames optimised placement and routing can be applied to multi–context designs, where contexts are not equally size–balanced and the size of the FPGA is usually selected to accommodate the biggest context. In this time of scenario FO P&R can significantly improve switching between small and medium size contexts.

Although FO P&R has been implemented for Xilinx Virtex technology only, presented approach can be easily modified to support other reconfigurable technologies. Novel DLC introduced at placement can be applied to any FPGA array as long as FPGA technology supports partial reconfiguration and configuration data follow regular patterns (vertical spanning frames, horizontal spanning frames, etc.).

Bitstream size reduction comes at the cost of timing deterioration. In Xilinx Virtex technology long routing resources can only be accessed from certain locations only therefore with highly compacted placement router is forced to use single length resources. That is the reason why longest paths after FO P&R use more switches. With different FPGA technology this limitation might be removed.

The fact, that in Xilinx Virtex CLB frames control top and bottom IOBs makes FO P&R reduction efficiency sensitive to IO pins allocation. Bitstream size reduction for high I/O pins count designs might be impossible as number of frames to load is determined by I/O pin allocation. Yet again this limitation is a feature of Xilinx Virtex architecture and might not occur with different FPGA technology.

Chapter 5

Multi–Context Placement and Routing Implementation

This chapter presents details about the implementation of multi–content placement and routing. New framework is described together with the results.

5.1 Designs Similarity Analysis

Using BCT collection of benchmarks has been analysed to derive initial level of similarity between members of each design pair. These similarities were used as a starting point to determine the quality of the novel multi–context simulatenous placement and routing approach. Results from this cross–comparison are presented in Table 13.

The results summarised in Table 13 show, that although there are pairs of circuits with certain number of similar CLBs, their overall bitstreams similarity is less then 1%. This is due to the fact, that they have all been placed and routed individually.

Table 13: Designs similarity summary.

Design pair	Size [%]	Shared blocks	Shared frames
b04_out.bit	11		
tb_04_chain_x02.bit	22	25	239
b12_out.bit	30		
tb_12_chain_x2.bit	60	63	4
tb_05_12.bit	45		
tb_04_chain_x4.bit	45	58	0
cf_fp_mul_c_5_10_out.bit	34		
cf_fp_mul_p_5_10_out.bit	43	113	0
cf_interleaver_6_8_out.bit	13		
cf_interleaver_6_64_out.bit	86	74	0
tb_10_chain_x20.bit	86		
tb_11_chain_x6.bit	48	21	0

5.2 Placement Algorithm Evaluation

Unlike traditional placement which starts from initial placement, novel approach has been applied based on placing similar blocks in the same location in both placements. Simultaneous placer needs first to determine the size for shared and non–shared blocks. This is done by cross–comparing the netlists to get the number of shared slices (set of two LUT within the same location).

5.2.1 Placement Algorithm Criteria

With the FPGA area divided into shared and non–shared areas, random initial placement is performed on the shared area and then on each non–shared area separately.

To obtain an implementation which considers column–based structuring, the configuration distribution set of modifications developed during bitstream size reduction described in section 4.2 placement have been used.

5.2.2 Dynamic Location Cost Schedule

To address the problem of column–oriented placement, Dynamic Location Cost (DLC), developed for bitstream size reduction was used. Any change to the cost of shared locations has to be applied to both netlists, while any change to the non–shared area applies only to the currently processed netlist.

5.2.3 Placement Cost Function

Following the bitstream size reduction placement methodology, the simulated annealing placement cost function based on column–oriented net bounding box was used. Placement iteration cost is calculated to all blocks within the design regardless of their assignment to a shared or non–shared area. Similar to bitstream compression placement, this cost function penalises horizontally spanned nets.

5.2.4 Next Step Criteria

Because of the processors sequential execution, only one context can be processed at a time. Therefore at the beginning of every placement iteration, selection of which design is going to be modified is based on a random number generator.

Once the design has been selected, there is another selection to be made between those areas where a proposed change will be applied. Similar to the previous selection case, this one is based on the random based function as well. Within the area selected, modification is done either by block swap or block move similar to the previously described in section 4.2.4 bitstream size reduction placement.

As a result of the FPGA area being divided into shared and non–shared areas, block manipulations are only allowed within the same area. However, there is an exception for the shared area: the block in the shared area can

still be swapped with the block in the non–shared area if their LUT content is exactly the same. Such a swap does not influence the content of shared columns (as the LUT content remains unchanged) but it has an impact on the net bounding box, and potentially influences the net's routing.

5.3 Routing Algorithm Evaluation

The main goal of the routing algorithm, is to validate placement by providing connectivity between specified block input and output while using a minimum number of configuration frames.

5.3.1 Routing Algorithm Criteria

Based on the results of bitstream size reduction routing, the same method has been selected to perform detailed routing. Because of the nets sharing complexity problem, each design has been routed independently.

5.3.2 Routing Cost Function

In order to keep the number of frames to a minimum, the cost function used during the search for the shortest–path has been modified to include the cost of the routing switch location. In this way the router will try to use routing switches, whose configuration is driven by the frames with higher utilisation.

5.4 Experimental Results

5.4.1 Simultaneous Placement and Routing Results

To test the feasibility of the proposed approach Simultaneous Multi–Context (SMC) P&R was performed on the set of benchmark designs. Cost functions

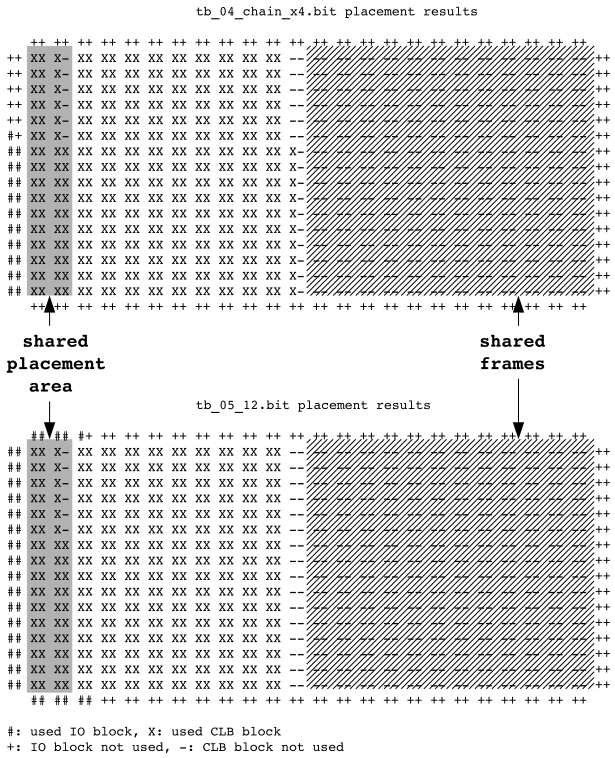

Figure 37: Example of SMC P&R performed on two-contexts design.

used by placement and routing algorithm were purely frame–sharing oriented without any other constraints. Experimental results are presented in Table 14, and an example of placement layout after SMC P&R is presented in Figure 37.

5.5 Frames Sharing Analysis

Results presented in Table 14 show, that it is possible to increase the number of shared frames compared to currently used P&R approaches. Not surprisingly the achieved improvement ratio is considerable for small circuits (51% for 20–30% circuits size category). However, the results also demonstrate that the approach is also beneficial for the bigger circuits (33% for upto 60% circuit size category).

Table 14: FPGA Multi–design benchmarks summary.

Design pair	Size [%]	SB	SF_{SA}	SF_{SMC}	F [%]	CP_{SA}	CP_{SMC}	CP [%]
A1	11					9	12	-25
A2	22	25	239	821	51	12	14	-14
B1	30					10	13	-23
B2	60	63	4	382	33	12	14	-14
C1	45					9	10	-10
C2	45	58	0	527	46	8	11	-27
D1	34					9	12	-25
D2	43	113	0	518	45	9	11	-27
E1	13					8	11	-25
E2	86	74	0	102	9	9	11	-18
F1	86					8	10	-20
F2	48	21	0	65	6	7	9	-22

Design pair	filename
A1	b04_out.bit
A2	tb_04_chain_x02.bit
B1	b12_out.bit
B2	tb_12_chain_x2.bit
C1	tb_05_12.bit
C2	tb_04_chain_x4.bit
D1	cf_fp_mul_c_5_10_out.bit
D2	cf_fp_mul_p_5_10_out.bit
E1	cf_interleaver_6_8_out.bit
E2	cf_interleaver_6_64_out.bit
F1	tb_10_chain_x20.bit
F2	tb_11_chain_x6.bit

5.6 Timing Analysis

The results of the timing analysis demonstrated, that the critical path used on average 25% more routing resources despite using the Pathfinder routing algorithm, which aims to find the shortest path for each net. This situation is a result of additional restrictions applied to the block allocation during placement, and the fact, that long wire connections are only available for certain block allocation combinations.

Additional constraints applied to the placement algorithm, where shared and non–shared blocks are assigned to the different FPGA areas were likely to contribute to the net bounding box size and therefore increase the number of routing resources required to successfully route the net.

The timing results presented describe a worst–case scenario, where placement and routing is purely compression and frames sharing oriented. As a result of the configuration interface architecture used (where all the blocks within the shared CLB column have to be exactly the same) any linear trade–off between timing and sharing might be difficult to achieve in practice.

5.6.1 CLB Blocks Sharing

Results of CLB block sharing between two designs are presented in Table 14 and suggest that there are a number of blocks with the same content (up to 15%). This is despite the fact that all the benchmark designs were compiled separately without any intention of sharing resources. Thus, the number of shared CLB blocks can be further improved by using resource–sharing methodologies developed in previous steps of the design flow.

5.7 Pins Allocation

As both designs are targeting the same FPGA platform, IO pin allocation will be determined by the PCB layout. However due to the unique nature of IO pin configuration (top and bottom IO pins are configured by CLB frames) IO pins have been left floating, to allow the algorithm to derive column–oriented placement.

5.8 Scalability Of The Approach

The presented SMC JPR approach has been tested for the limited case of two-context designs, but it can easily be scaled to handle three and more designs. The overall frame sharing ratio will then depend on the netlist similarity factor and it is likely to decrease as the number of designs increases.

5.9 Frames Overlapping Problem

Unlike bitstream size reduction, SMD P&R distinguishes between two types of shared frames: shared empty frames and shared design frames. Shared design frames represent non–empty frames with the same content. The detailed contribution of both types into the total number of shared frames has been presented in Table 15.

Results presented in Table 15 clearly show that the majority of shared frames are empty frames. Despite CLB similarity and their specific placement, the content of the configuration frames covering that area is still different. This is caused by the specific configuration frame structure, where a single frame is in control of a part of the CLB content as well as routing in that area. Although sharing LUT content can be arranged during placement, any constraints applied to the way the design is routed could result in unroutable nets.

Table 15: Summary of shared empty and design frames.

Design filename	Shared empty frames	Shared design frames
b04_out.bit tb_04_chain_x02.bit	821	2
b12_out.bit tb_12_chain_x2.bit	380	4
tb_05_12.bit tb_04_chain_x4.bit	523	4
cf_fp_mul_c_5_10_out.bit cf_fp_mul_p_5_10_out.bit	516	2
cf_interleaver_6_8_out.bit cf_interleaver_6_64_out.bit	92	10
tb_10_chain_x20.bit tb_11_chain_x6.bit	64	1

5.10 Configuration Interface Analysis

To demonstrate the impact of configuration interface architecture on bitstream sharing in a multi–design approach, simulation of a selection of different configuration interface architectures was performed. Because developed framework is based on JBits supporting only Xilinx Virtex FPGA technology, selection of different frames allocation patterns was used to emulate different reconfigurable architectures.

5.10.1 LUT Specific Frames

A single CLB column in XCV50 is configured by 48 frames. 18 bits in each frame are used to hold a part of the configuration of a single CLB location. Currently at least 32 frames are required to configure LUT, as a single CLB frame holds only two LUT bits per location. To configure LUTs within a single CLB location 64 bits are required. Therefore, if there are LUT–dedicated frames, to configure a LUT in entire CLB column requires four frames. An example of a proposed configuration interface architecture is

Table 16: Summary of shared empty and design frames for CLB specified frames.

Design filename	Shared empty frames	Shared design frames
b04_out.bit tb_04_chain_x02.bit	821	10
b12_out.bit tb_12_chain_x2.bit	380	20
tb_05_12.bit tb_04_chain_x4.bit	523	20
cf_fp_mul_c_5_10_out.bit cf_fp_mul_p_5_10_out.bit	516	34
cf_interleaver_6_8_out.bit cf_interleaver_6_64_out.bit	92	30
tb_10_chain_x20.bit tb_11_chain_x6.bit	64	8

presented in Figure 38.

It can be assumed that shared CLB column will have at least four frames similar, and it will not depend on the routing results. Bitstream sharing after using modified frames distribution has been presented in Table 16.

5.10.2 Single Block Configuration

In the Xilinx Virtex XCV50 single configuration frame has 384 bits and each CLB location is configured by 864 bits. Therefore, to configure an entire CLB location (LUT and surrounding routing switches), 2.25 frames are required. The main advantage of this architecture was that shared blocks do not have to be placed within a column. They only need to be similar within a single location. Use of this approach would therefore result in better routing, as blocks are not forced into columns, which relaxes routing congestion.

Using smaller configuration frames can further improve the configuration data sharing ratio. As illustrated in Figure 39 a single CLB location has been configured using nine horizontally spanning configuration frames. Such

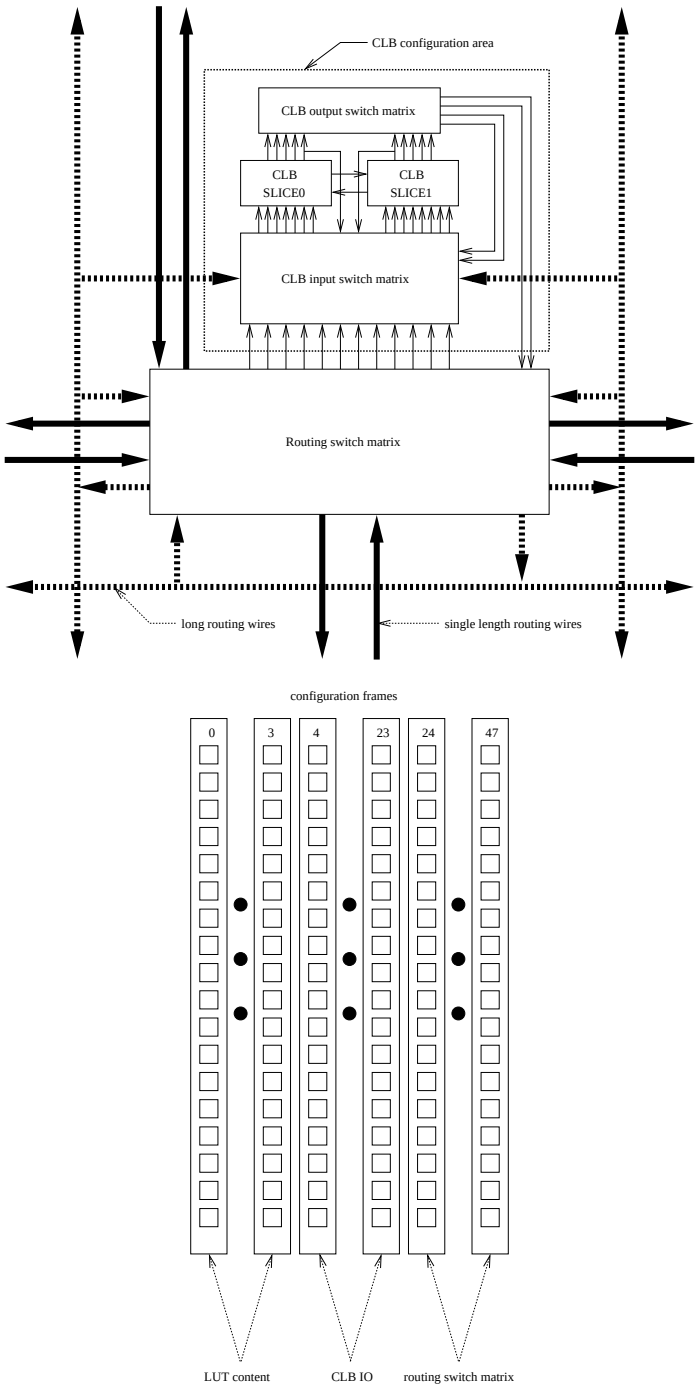

Figure 38: Example of Xilinx Virtex single programmable cell.

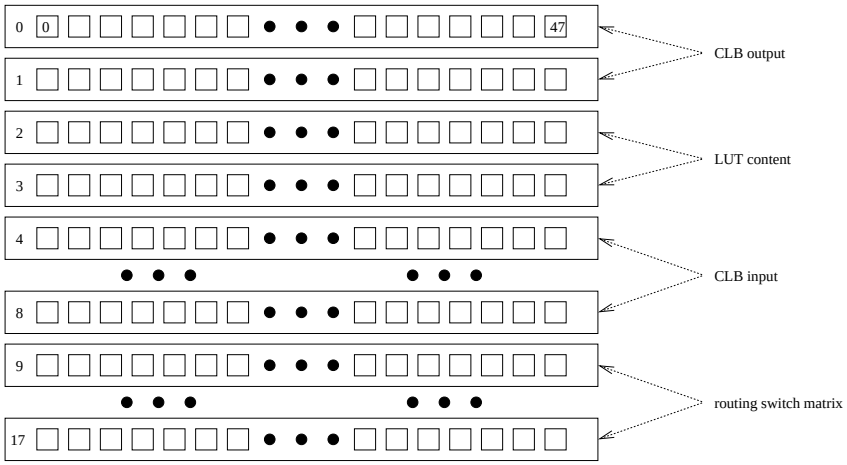

Figure 39: Example of proposed configuration interface architecture for single Xilinx Virtex programmable cell.

a frame alignment allows separate configuration of LUT content and related I/O routing from external routing. Also if the CLB is used for routing only, the LUT part can be still shared with other empty LUTs.

Based on the placement and routing results of the benchmark circuits an estimate for configuration data sharing was obtained and is presented in Table 17.

Smaller configuration frames will increase configuration architecture complexity, but offer on average 10% improvement compared to a column based configuration interface architecture. It also does not require similar CLB blocks to be placed together, as similarity is done purely on a one–to–one basis. A further advantage comes from the routing improvements as CLB block allocation is less constrained.

An additional post–processing run can be applied to the netlist in order to increase the number of shared LUTs. Content of the single LUT can be modified as indicated in Figure 40.

With fine grain configuration interface architecture ‘wildcarding’ can be an option especially for the big CLB arrays, as the number of similar LUT blocks is likely to increase due to a limited combination of 16–bit wide LUT

Table 17: Designs similarity summary.

Design pair	Shared design blocks	Shared empty blocks	Shared frames	SF [%]
b04_out.bit tb_04_chain_x02.bit	25	597	5398	78
b12_out.bit tb_12_chain_x2.bit	63	306	2637	38
tb_05_12.bit tb_04_chain_x4.bit	58	412	3766	54
cf_fp_mul_c_5_10_out.bit cf_fp_mul_p_5_10_out.bit	113	406	3767	54
cf_interleaver_6_8_out.bit cf_interleaver_6_64_out.bit	74	107	1037	15
tb_10_chain_x20.bit tb_11_chain_x6.bit	21	102	939	14

content.

5.11 Summary

The novel simultaneous placement and routing methodology for multi–context designs presented in this chapter demonstrates the feasibility of increasing the number of shared configuration data frames to improve the speed of the FPGA reconfiguration process.

The achieved improvement ratio is considerable for small circuits (51% for 20–30% circuits size category). However, the results also demonstrate that the approach is also beneficial for the bigger circuits (33% for upto 60% circuit size category). This is beneficial especially in multi-context designs where single contexts differ is size and complexity.

The results of the timing analysis demonstrated, that the critical path used on average 25% more routing resources despite using the Pathfinder routing algorithm, which aims to find the shortest path for each net. This situation is a result of additional restrictions applied to the block allocation

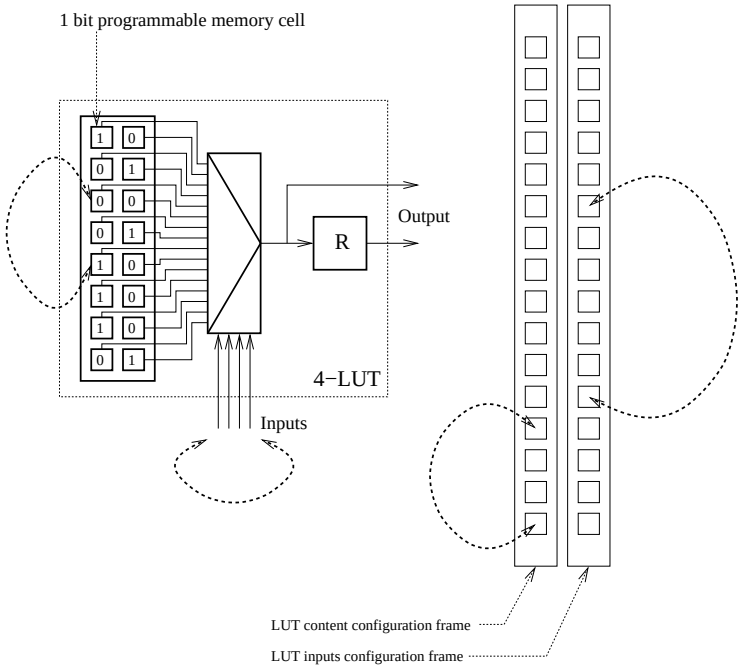

Figure 40: Example of Xilinx Virtex single programmable cell.

during placement, and the fact, that long wire connections are only available for certain block allocation combinations.

Additional constraints applied to the placement algorithm, where shared and non–shared blocks are assigned to the different FPGA areas were likely to contribute to the net bounding box size and therefore increase the number of routing resources required to successfully route the net.

The timing results presented describe a worst–case scenario, where placement and routing is purely compression and frames sharing oriented. As a result of the configuration interface architecture used (where all the blocks within the shared CLB column have to be exactly the same) any linear trade–off between timing and sharing might be difficult to achieve in practice.

The ability to transform netlist similarities into configuration data similarities works well for the difficult case, where configuration data frame

consists of bits controlling parts of different FPGA resources (e.g. IOB, CLB, routing), as any modification to placement or routing applies to a number of configuration frames at once. In similarity to single–context placement and routing methodology, additional restrictions to the placement algorithm were found to increase the critical path net delay, although developed methodology can be still attractive for non–timing critical, power supply limited application domains.

The presented placement and routing methodology has been tested for the limited case of two-context designs, but it can easily be scaled to handle three and more context designs. The overall frame sharing ratio depends on the netlist similarity factor and it is likely to decrease as the number of designs increases.

The limited information available on alternative configuration interface architectures make it difficult to evaluate the proposed methodology, although simulation of alternative configuration interface architectures demonstrate even better improvement in configuration data sharing and timing.

Newly developed multi-context placement and routing methodology proved to work with real-world Xilinx Virtex FPGA - challenging technology due to its configuration frames architecture. Simulation of different frames arrangement demonstrate impact of configuration interface architecture on bitstream sharing and as such can be valid test methodology in the search for better configuration interface architecture.

Chapter 6

Conclusions and Future Work

The thesis are concluded in this chapter together with summary of future areas of development.

6.1 Summary of the Contribution

The work presented in this thesis has explored the area of placement and routing for FPGAs in the search for improvements to the FPGA reconfiguration problem bottleneck – one of the main obstacles towards wider popularity of FPGAs as reconfigurable systems.

It has been demonstrated, that currently available placement and routing methodologies, although delivering quality solutions, especially in the timing optimisation domain, suffer a significant overhead in utilising FPGA configuration data resources, which contribute towards reconfiguration latency and storage penalties.

6.1.1 Single Context Approach

It has been demonstrated, that by incorporating configuration data architecture into placement and routing, the configuration data utilisation

can be improved by using up to five times less configuration data resources to implement the design over traditional approaches. Such an improvement is an important contribution towards minimising FPGA reconfiguration storage and latency overhead, and also power consumption, which is very important especially for battery–operated devices.

A placement and routing framework presented in this thesis has been validated with commercial Xilinx Virtex FPGA technology to show its practicality for real–world applications.

The developed novel bitstream size reduction placement and routing for FPGAs when applied to the Xilinx Virtex FPGAs, demonstrates a trade–off between compression and design timing. It has been demonstrated, that compression data utilisation improvement comes with design timing degradation of up to 30% more routing switches used by the net on the critical path, due to the higher block concentration within the placement area, which limits router flexibility. For highly timing–constrained designs this method might be not applicable, although there are other application domains like battery operated mobile devices, where the benefits of using hardware design implementation is the key to power saving by lowering operating frequency – a key element in determining power consumption.

6.1.2 Multi–Context Approach

The contribution from bitstream size reduction placement and routing provided essential background for development of a more generic placement and routing methodology optimising configuration data sharing between contexts compiled for a particular reconfigurable platform.

Critical evaluation of currently available placement and routing methodologies deliver bitstreams with very low level of configuration data similarity, making re–using of configuration data almost impossible. Analysis of the design flow of these methodologies indicated areas of possible

improvement to increase levels of configuration data sharing.

It has been demonstrated, that using simultaneous placement and routing of two contexts provides upto five times more shared configuration data frames, which contributes towards reconfiguration latency and storage overhead improvement.

The developed multi–context simultaneous placement and routing for FPGAs, when applied to the Xilinx Virtex FPGAs, demonstrates trade–off between configuration data sharing and design timing improvement. It has been demonstrated, that configuration data sharing improvement comes with design timing degradation up to 30%. This problem arises as more routing switches are used by the net on the critical path, due to the additional placement constraints, which limits router flexibility. For highly timing–constrained designs this method might be not applicable, although there are other application domains like battery operated mobile devices, where savings in power consumption during the process of reconfiguration are highly desirable.

6.2 Solution Scalability

The developed placement and routing methodology optimised for configuration data utilisation, successfully tested with Xilinx Virtex FPGA technology, can be applied to any island–style FPGA with regular configuration interface architecture (row–based, diagonal–based). The Dynamic Location Cost developed for placement methodology is a key element linking FPGA configuration interface architecture with FPGA placement methodology.

Multi–context methodology has been based on design netlist similarity and as such can be applied to an unlimited number of designs, known at compile time. The overlapping ability will depend on netlist similarity, and

in the worst case scenario (when no similarities can be picked up) it will deliver a set of individually compressed bitstreams, able to share part of the unassigned FPGA area.

Following the current trends where FPGAs are becoming stand–alone Systems–on–Chip offering much more then just configurable logic, the developed methodology can still be applied to minimise the size of configuration data required to implement IP cores in the FPGA configurable logic area.

6.3 Areas of Improvement and Future Directions

The configuration data optimised placement and routing methodology presented in this thesis represents an alternative approach to EDA, and research in this area can deliver further improvements to the problem of FPGA reconfiguration bottleneck. Research should be done in several directions:

- placement and routing algorithm optimisation,
- configuration data architectures,
- high–level design synthesis.

The trend to incorporate configuration interface architecture into placement and routing can be expanded onto any available FPGA architecture, subject to detailed FPGA architectural information being available.

As it has been demonstrated, the achieved configuration data sharing ratio depends on the configuration data architecture. There is a need for more research to evaluate the most feasible configuration interface architecture for dynamically reconfigurable FPGAs, so that the entire reconfiguration process including placement and routing can be improved even further.

Analysis of multi–context placement and routing show that the quality of the delivered solution is strongly dependent on netlist similarity. As design netlists are a result of design compilation process, the area of simultaneous design compilation is one of the key areas for future research in order to improve netlist similarity and therefore bring further improvements in configuration data sharing.

Appendix A

Appendix

A.1 Bitstream Comparison Tool

Code size: 1070 lines

```
/*****************************************
    main.c  -  description
    ----------------------
    begin     : Tue Feb  3 12:39:42 GMT 2004
    copyright: (C) 2004 by Piotr Stepien
    email     : pstepien@bournemouth.ac.uk
 *****************************************/

/*****************************************
 *                                       *
 *   This program is free software;      *
 *   you can redistribute it and/or modify *
 *   it under the terms of the GNU General *
 *   Public License as published by      *
 *   the Free Software Foundation;       *
 *   either version 2 of the License, or *
 *   (at your option) any later version. *
 *                                       *
```

```
 ****************************************/

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>


#define XCV50

#if defined(XCV1000)
  #define ROWS 64
  #define CLB_COLUMNS 96
  #define WORDS_PER_FRAME 39
  #define STREAM_SIZE 766047
#endif

#if defined(XCV50)
  #define ROWS 16
  #define CLB_COLUMNS 24
  #define WORDS_PER_FRAME 12
  #define STREAM_SIZE 69970
#endif




#define CLB_FRAMES 48



unsigned char bitstream_data[2][STREAM_SIZE];
unsigned int
    frame_start_addr[2][4],
```

```
    number_of_words[2][4];

unsigned int
    subblocks_data[2][CLB_COLUMNS][48][ROWS+3],

    clk_data[2][1][8][ROWS+3],
    clb_data[2][CLB_COLUMNS][48][ROWS+3],
    iob_data[2][2][54][ROWS+3],
    bsramint_data[2][2][27][ROWS+3],
    bsramcontent_data[2][2][64][ROWS+3];

#include "functions_collection.h"


int main()
{
   /* Variables definition */

   unsigned char bit_filename[255], decision;
   unsigned int
      words_rom[37],
      read_word[37],
      passed_Bytes,
      word_index,
      int_words_ctr,
      bitstream_file_number,
      bitstream_byte_index;



      FILE *bitstream_filePtr; /* Bitstream file pointer */

  /* Bitstream Header and Configuration Options (XAPP138v2.7
     Table 8)*/
  words_rom[0] = 0xffffffff; /* Dummy word following bitstream
     */
```

```
words_rom[1] = 0xaa995566; /* Synchronisation word */
words_rom[2] = 0x30008001; /* Packet Header: Write to SMD
    register */
words_rom[3] = 0x00000007; /* Packet Data: RCPC */
words_rom[4] = 0x30016001; /* Packet Header: Write to FLR
    register */
words_rom[5] = 'z'; /* Packet Data: Frame Length for XCV1000
    */
words_rom[6] = 0x30012001; /*Packet Header: Write to COR */
words_rom[7] = 'z'; /* Packet Data: Configuration Options */
words_rom[8] = 0x3000c001; /* Packet Header: Write to MASK */
words_rom[9] = 'z'; /* Pocket Data: CTL mask */
words_rom[10] = 0x30008001; /* Packet Header: Write to CMD
    register */
words_rom[11] = 0x00000009; /* Packet Data: SWITCH */
words_rom[12] = 0x30002001; /* Packet Header: Write to FAR
    register */
words_rom[13] = 0x00000000; /* Packet Data: Frame address */
words_rom[14] = 0x30008001; /* Packet Header: Write to CMD
    register */
words_rom[15] = 0x00000001; /* Packet Data: WCFG */

/* Bitstream Data Frames and CRC (XAPP138v2.7 Table 9) */
words_rom[16] = 0x30004000; /* Packet Header: Write to FDRI */
words_rom[17] = 'z'; /* Pocket Header Type 2: Data Words */

/* Continuing after frames loading */
words_rom[18] = 0x30002001; /* Pocket Header: Write to FAR
    register */
words_rom[19] = 'z'; /* Pocket Data: Next frame address */
words_rom[20] = 'z'; /* Write to FDRI */
words_rom[21] = 'z'; /* Packet Header Type 2: Data words */
words_rom[22] = 0x30002001; /* Packet Header: Write to FAR
    register */
words_rom[23] = 'z'; /* Pocket Data: Next frame address */
words_rom[24] = 'z'; /* Packet Header: Write to FDRI */
```

```
words_rom[25] = 'z'; /* Pocket Header Type 2: Data Words */
words_rom[26] = 0x30008001; /* Packet Header: Write to CRC */
words_rom[27] = 'z'; /* Packet Data: CRC value */
words_rom[28] = 0x30008001; /* Packet Header: Write to CMD
    register */
words_rom[29] = 0x00000003; /* Packet Data: LFRM */
words_rom[30] = 'z'; /* Packet Header: Write to FDRI */

/* Final CRC and Start-up */
words_rom[31] = 0x30008001; /* Packet Header: Write to CMD
    register */
words_rom[32] = 0x00000005; /* Packet Data: START */
words_rom[33] = 0x3000a001; /* Packet Header: Write to CTL */
words_rom[34] = 'z'; /* Packet Data: Control commands */
words_rom[35] = 0x30000001; /* Packet Header: Write to CRC */
words_rom[36] = 'z'; /* Packet Data: CRC value */
```

```
/* Welcome message */

printf("Bitstream analyzer\n");
printf("Version 2.0\n");
printf("Autor: Piotr Stepien\n\n");


/* Loading Bitstreams */

bitstream_file_number = 0;

if((bitstream_filePtr = fopen("null50GCLK0.bit", "r")) ==
    NULL)
{
   printf("File could not be opened.\n");
```

```
  return 0;
}//end if
else
{
  fread(bitstream_data[bitstream_file_number], sizeof(
      unsigned char),
   STREAM_SIZE, bitstream_filePtr);
  fclose(bitstream_filePtr);
}//end else

printf("Bitstream %d loaded successfully.\n\n",
    bitstream_file_number);
bitstream_file_number++;

if((bitstream_filePtr = fopen("xcv50GCLK0_route.bit", "r"))
    == NULL)
{
  printf("File could not be opened.\n");
  return 0;
}//end if
else
{
  fread(bitstream_data[bitstream_file_number], sizeof(
      unsigned char),
   STREAM_SIZE, bitstream_filePtr);
  fclose(bitstream_filePtr);
}//end else

printf("Bitstream %d loaded successfully.\n\n",
    bitstream_file_number);
bitstream_file_number = 0;

printf("Reading verification:\n");
printf("Bitstream 1 %d\n", bitstream_data[0][1]);
printf("Bitstream 2 %d\n", bitstream_data[1][1]);
```

```
/*    printf("Locate frames? (y/n) ");

    scanf("%s", &decision);
    if(decision == 'n')
    {
      return 0;
    }//end if
*/

    /* Extract frames from the Bitstream */

    bitstream_file_number = 0;
    do
    {
      printf("Frames extraction from Bitstream %d.\n\n",
       bitstream_file_number);
      bitstream_byte_index = 0;

      /* Locate Dummy word in the file */
      passed_Bytes = FindSpecific(bitstream_file_number,
       bitstream_byte_index, words_rom[0]);
      read_word[0] = words_rom[0];
      word_index = 1;
      int_words_ctr = 0;
      printf("Dummy word located.\n");
      bitstream_byte_index = passed_Bytes;
      printf("The number of Bytes passed is: %d\n", passed_Bytes)
          ;

      /* Check if words of block 1 are in the right position */
      do
      {
        int_words_ctr++;
        read_word[word_index] = Read_32BitWord(
            bitstream_file_number,
         bitstream_byte_index);
```

```
    bitstream_byte_index = bitstream_byte_index + 4;
    if((words_rom[word_index]) != 'z')
    {
      if(read_word[word_index] == words_rom[word_index])
      {
        printf("Word %d 0x%x", word_index, words_rom[
            word_index]);
        printf(" in the position.\n");
        word_index++;
      }//end if
      else
      {
        printf("ERROR!!! - Word %d 0x%x", word_index,
         words_rom[word_index]);
        printf(" not found. \n");
        return 0;
      }//end else
    }//end if
    else
    {
      printf("Word %d 0x%x", word_index, read_word[word_index
          ]);
      printf(" read from the bitstream.\n");
      word_index++;
    }//end else
  }//end do
  while(int_words_ctr < 16);

  /* * * * * * * * * * * * * * */
  /* First block of frames */
  /* * * * * * * * * * * * * * */

  /* Read Packet Header Type 2: Data words */
  read_word[word_index] = Read_32BitWord(
      bitstream_file_number,
   bitstream_byte_index);
```

```
bitstream_byte_index = bitstream_byte_index + 4;
number_of_words[bitstream_file_number][0] = read_word[
    word_index];
word_index++;
frame_start_addr[bitstream_file_number][0] =
    bitstream_byte_index;
number_of_words[bitstream_file_number][0]<<=5;
number_of_words[bitstream_file_number][0]>>=5;
printf("Start address of the first block of frames: %d\n",
 frame_start_addr[0][bitstream_file_number]);
printf("Number of words to read: %d\n",
 number_of_words[bitstream_file_number][0]);
printf("Number of frames to read: %d\n",
 (number_of_words[bitstream_file_number][0])/
    WORDS_PER_FRAME);
printf("Bitstream %d Header 1 loaded successfully\n\n",
 bitstream_file_number);

/* Check if words of block 2 right position */

int_words_ctr = 0;
bitstream_byte_index = bitstream_byte_index +
 (4 * number_of_words[bitstream_file_number][0]);
do
{
  int_words_ctr++;
  read_word[word_index] = Read_32BitWord(
      bitstream_file_number,
   bitstream_byte_index);
  bitstream_byte_index = bitstream_byte_index + 4;
  if((words_rom[word_index]) != 'z')
  {
    if(read_word[word_index] == words_rom[word_index])
    {
      printf("Word %d 0x%x", word_index, words_rom[
          word_index]);
```

```
      printf("␣in␣the␣position.\n");
      word_index++;
    }//end if
    else
    {
      printf("ERROR!!!␣␣–␣Word␣%d␣0x%x", word_index,
       words_rom[word_index]);
      printf("␣not␣found.␣\n");
      return 0;
    }//end else
  }//end if
  else
  {
    printf("Word␣%d␣0x%x", word_index, read_word[word_index
        ]);
    printf("␣read␣from␣the␣bitstream.\n");
    word_index++;
  }//end else
}//end do
while(int_words_ctr < 2);

/* * * * * * * * * * * * * */
/* Second block of frames */
/* * * * * * * * * * * * * */

/* Read Packet Header Type 2: Data words */
read_word[word_index] = Read_32BitWord(
    bitstream_file_number,
 bitstream_byte_index);
bitstream_byte_index = bitstream_byte_index + 4;
number_of_words[bitstream_file_number][1] = read_word[
    word_index];
word_index++;
frame_start_addr[bitstream_file_number][1] =
    bitstream_byte_index;
number_of_words[bitstream_file_number][1]<<=21;
```

```
number_of_words[bitstream_file_number][1]>>=21;

if(number_of_words[bitstream_file_number][1] == 0)
{
  read_word[word_index] = Read_32BitWord(
      bitstream_file_number,
   bitstream_byte_index);
  bitstream_byte_index = bitstream_byte_index + 4;
  number_of_words[bitstream_file_number][1] = read_word[
      word_index];
  word_index++;
  frame_start_addr[bitstream_file_number][1] =
      bitstream_byte_index;
  number_of_words[bitstream_file_number][1]<<=5;
  number_of_words[bitstream_file_number][1]>>=5;
}//end if
else
{
  word_index++;
}//end else

printf("Start address of the second block of frames: %d\n",
 frame_start_addr[bitstream_file_number][1]);
printf("Number of words to read: %d\n",
 number_of_words[bitstream_file_number][1]);
printf("Number of frames to read: %d\n",
 (number_of_words[bitstream_file_number][1])/
     WORDS_PER_FRAME);
printf("Bitstream %d Header 2 loaded successfully\n\n",
 bitstream_file_number);

/* Check if words of block 3 right position */

int_words_ctr = 0;
bitstream_byte_index = bitstream_byte_index +
 (4 * number_of_words[bitstream_file_number][1]);
```

```
do
{
  int_words_ctr++;
  read_word[word_index] = Read_32BitWord(
      bitstream_file_number,
   bitstream_byte_index);
  bitstream_byte_index = bitstream_byte_index + 4;
  if((words_rom[word_index]) != 'z')
  {
    if(read_word[word_index] == words_rom[word_index])
    {
      printf("Word %d 0x%x", word_index, words_rom[
          word_index]);
      printf(" in the position.\n");
      word_index++;
    }//end if
    else
    {
      printf("ERROR!!! - Word %d 0x%x", word_index,
       words_rom[word_index]);
      printf(" not found. \n");
      return 0;
    }//end else
  }//end if
  else
  {
    printf("Word %d 0x%x", word_index, read_word[word_index
        ]);
    printf(" read from the bitstream.\n");
    word_index++;
  }//end else
}//end do
while(int_words_ctr < 2);

/* * * * * * * * * * * * * */
/* Third block of frames */
```

```
/* * * * * * * * * * * * * */

/* Read Packet Header Type 2: Data words */
read_word[word_index] = Read_32BitWord(
    bitstream_file_number,
 bitstream_byte_index);
bitstream_byte_index = bitstream_byte_index + 4;
number_of_words[bitstream_file_number][2] = read_word[
    word_index];
word_index++;
frame_start_addr[bitstream_file_number][2] =
    bitstream_byte_index;
number_of_words[bitstream_file_number][2]<<=21;
number_of_words[bitstream_file_number][2]>>=21;

if(number_of_words[bitstream_file_number][2] == 0)
{
  read_word[word_index] = Read_32BitWord(
      bitstream_file_number,
   bitstream_byte_index);
  bitstream_byte_index = bitstream_byte_index + 4;
  number_of_words[bitstream_file_number][2] = read_word[
      word_index];
  word_index++;
  frame_start_addr[bitstream_file_number][2] =
      bitstream_byte_index;
  number_of_words[bitstream_file_number][2]<<=5;
  number_of_words[bitstream_file_number][2]>>=5;
}//end if
else
{
  word_index++;
}//end else

printf("Start address of the third block of frames: %d\n",
 frame_start_addr[bitstream_file_number][2]);
```

```
printf("Number of words to read: %d\n",
 number_of_words[bitstream_file_number][2]);
printf("Number of frames to read: %d\n",
 (number_of_words[bitstream_file_number][2])/
    WORDS_PER_FRAME);
printf("Bitstream %d Header 3 loaded successfully\n\n",
 bitstream_file_number);

int_words_ctr = 0;
bitstream_byte_index = bitstream_byte_index +
 (4 * number_of_words[bitstream_file_number][2]);
do
{
  int_words_ctr++;
  read_word[word_index] = Read_32BitWord(
      bitstream_file_number,
   bitstream_byte_index);
  bitstream_byte_index = bitstream_byte_index + 4;
  if((words_rom[word_index]) != 'z')
  {
    if(read_word[word_index] == words_rom[word_index])
    {
      printf("Word %d 0x%x", word_index, words_rom[
          word_index]);
      printf(" in the position.\n");
      word_index++;
    }//end if
    else
    {
      printf("ERROR!!!  - Word %d 0x%x", word_index,
       words_rom[word_index]);
      printf(" not found. \n");
      return 0;
    }//end else
  }//end if
  else
```

```
  {
    printf("Word %d 0x%x", word_index, read_word[word_index
        ]);
    printf(" read from the bitstream.\n");
    word_index++;
  }//end else
}//end do
while(int_words_ctr < 4);

/* * * * * * * * * * * * * */
/* Fourth block of frames */
/* * * * * * * * * * * * * */

/* Read Packet Header Type 2: Data words */
read_word[word_index] = Read_32BitWord(
    bitstream_file_number,
 bitstream_byte_index);
bitstream_byte_index = bitstream_byte_index + 4;
number_of_words[bitstream_file_number][3] = read_word[
    word_index];
word_index++;
frame_start_addr[bitstream_file_number][3] =
    bitstream_byte_index;
number_of_words[bitstream_file_number][3]<<=21;
number_of_words[bitstream_file_number][3]>>=21;
printf("Start address of the forth block of frames: %d\n",
 frame_start_addr[bitstream_file_number][3]);
printf("Number of words to read: %d\n",
 number_of_words[bitstream_file_number][3]);
printf("Number of frames to read: %d\n",
 (number_of_words[bitstream_file_number][3])/
    WORDS_PER_FRAME);
printf("Bitstream %d Header 4 loaded successfully\n\n",
 bitstream_file_number);

printf("Frames in Bitstream %d located successfully.\n\n",
```

```
        bitstream_file_number);

/*      printf("Continue (y/n) ?\n");
      scanf("%s", &decision);
      if(decision == 'n')
      {
        return 0;
      }//enf if
*/
      bitstream_file_number++;
    }//end do
    while(bitstream_file_number<2);

    /* * * * * * * * * * * * * * * * * * */
    /* Save selected columns content */
    /* * * * * * * * * * * * * * * * * * */
    FramesStats();

    ConvertBitstreamToBlocks();
    PrintCLBColumns();
    CompareCLBFrames();
    CompareCLBBlocks();
    CompareCLBFrames_octave();
    CompareCLBBlocks_octave();
    CompareCLBSubBlocks_octave();
    PrintGCLKColumn();
    CompareGCLKFrames();
    CompareGCLKBlocks();
    CompareGCLKSubBlocks();
    CompareGCLKFrames_octave();
    CompareGCLKBlocks_octave();
    CompareGCLKSubBlocks_octave();
    CompareCLBSubBlocks();
    return 0;
}
```

A.2 Java Place and Route

Code size: 15 082 lines

```
package dualPR;

/**
 * <p>Title: Simultaneous Placement and Routing for 2 bitstreams
    </p>
 *
 * <p>Description: </p>
 *
 * <p>Copyright: Copyright (c) 2005</p>
 *
 * <p>Company: </p>
 *
 * @author Piotr Stepien
 * @version 1.0
 */

import java.io.*;
import java.util.*;

public class Modificator {

    Framework framework; //keeps all parameters required for
        placement and routing
    Designs designs; //keeps all designs - netlist and blocklist
         of each circuit
    PrintStream ps; //for "Modificator.txt"

    public Modificator(Framework frameworkLoaded, Designs
        designsLoaded) {

        framework = frameworkLoaded;
        designs = designsLoaded;
```

```
        initialisePrintStream();

        if(ps != null)
        {
            ps.println("Welcome to designs modificator.");

        }
    }


    private void printClbFramesStats()
    {
        Iterator itr;
        SingleDesign design;

        if(ps != null)
        {
            itr = (Iterator) designs.getDesignsIterator();

            while(itr.hasNext() == true)
            {
                design = (SingleDesign) itr.next();

                ps.print(design.getName() + " " + design.
                    getBitstream().getClbFrames().
                    getNumberOfEmptyClbFrames() + "/" + design.
                    getBitstream().getClbFrames().
                    getNumberOfSharedClbFrames() + "   ");
            }
            ps.println();
        }

    }

    private void initialisePrintStream() {
        String filename;
```

```
        ps = null;
        filename = designs.getMasterDesignsPath() + "/
            Modificator.txt";

        if (framework.getVerboseLevel() > 0) {
                    try {
                        ps = new PrintStream(new
                            FileOutputStream(filename));
                    } catch (FileNotFoundException fnfe) {
                        System.out.println("Exception: " + fnfe)
                            ;
                    }

                }

    }

    public void processDesigns()
    {
        if(ps != null)
        {
            ps.println("Welcome to Dual Bitstream Modificator.")
                ;
        }

        designs.processOriginalBitstreams();

        if(ps != null)
        {
            ps.println("Initial frames statistics:");
        }

        designs.analyseBitstreams();
        if(ps != null)
        {
```

```
        ps.println("Best Frames Search completed.");
    }

    designs.analyseSimilarities();
    designs.analyseTimingSeparately();

    System.out.println("Processing started.");

    designs.placeDesigns();

    System.out.println("Placement completed.");
    designs.printCurrentPlacement(); // prints out placement
        results for each design
    designs.optimisePlacementSeparately();
    designs.printCurrentPlacement();
    designs.routeSeparately();
    designs.implementDesigns();
    designs.generateBitstreams();
    designs.processBitstreams();

    if(ps != null)
    {
        ps.println("Random placement and routing frames
            statistics:");
    }

    designs.analyseBitstreams();
    designs.analyseTimingSeparately();

    if(ps != null)
    {
        ps.println("Best Frames Search completed.");
    }

    designs.placeRandomly(); // random allocation of blocks
        in each design
```

```
designs.printCurrentPlacement(); // prints out placement
    results for each design
designs.routeSeparately(); // route each design
designs.implementDesigns(); // saves clb blocks settings
    to jbits
designs.generateBitstreams(); // generate modified
   bitstreams
designs.processBitstreams();


if(ps != null)
{
    ps.println("Random placement and routing frames
        statistics:");
}

designs.analyseBitstreams();

if(ps != null)
{
    ps.println("Best Frames Search completed.");
}


designs.placeRandomly(); // random allocation of blocks
   in each design
designs.printCurrentPlacement(); // prints out placement
    results for each design
designs.routeSeparately(); // route each design -
   temporarily disabled
designs.implementDesigns(); // saves clb blocks settings
    to jbits
designs.generateBitstreams(); // generate modified
   bitstreams
designs.processBitstreams();
```

```
        if(ps != null)
        {
            ps.println("Random placement and routing frames
                statistics:");
        }

        designs.analyseBitstreams();

        if(ps != null)
        {
            ps.println("Best Frames Search completed.");
        }



        designs.generateOriginalBitstreams();
        designs.processBitstreams();
        designs.compareClbFrames();
        printClbFramesStats();

        designs.placeRandomly();
        designs.routeSeparately();
        designs.implementDesigns();
        designs.generateBitstreams();
        designs.processBitstreams();
        designs.compareClbFrames();
        printClbFramesStats();
    }
}
```

References

Abke, J. and Barke, E. (2001). A New Placement Method for Direct Mapping into LUT-based FPGAs, In: G. Brebner and R. Woods (editors), *Field-Programmable Logic and Applications*, Springer, pp. 27–36.

Actel (2009). Actel Corp., http://www.actel.com.

Agostini, L. V., Filho, A. P. A., Rosa, V. S., Berriel, E. A., Santos, T. G. S., Bampi, S. and Susin, A. A. (2006). FPGA Design Of A H.264/AVC Main Profile Decoder For HDTV, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 501–506.

Ahmed, I. and Arslan, T. (2006). A Reconfigurable Viterbi Decoder For A Communication Platform, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 435–440.

Anderson, J., Saunders, J., Nag, S., Madabhushi, C. and Jayaraman, R. (2000). A Placement Algorithm for FPGA Designs with Multiple I/O Standards, In: R. W. Hartenstein and H. Grunbacher (editors), *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, Springer, pp. 211–220.

Arteaga, R., Tobajas, F., Esper-Chain, R., de Armas, V. and Sarmiento, R. (2008). GMDS: hardware implementation of novel real output queuing

architecture, In: *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, ACM, New York, NY, USA, pp. 1450–1455.

Barreiros, J. and Costa, E. (2003). Global Routing for Lookup-Table Based FPGAs Using Genetic Algorithms, In: P. Y. K. Cheung, G. A. Constantinides and J. T. de Sousa (editors), *Field-Programmable Logic and Applications*, Springer, pp. 141–150.

Bauer, L., Shafique, M. and Henkel, J. (2009). MinDeg: a performance-guided replacement policy for run-time reconfigurable accelerators, In: *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ACM, New York, NY, USA, pp. 335–342.

Berthelot, F., Nouvel, F. and Houzet, D. (2006). Partial and dynamic reconfiguration of FPGAs: a top down design methodology for an automatic implementation, In: *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*.

Betz, V., Rose, J. and Marquardt, A. (1999). *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers.

Brebner, G. (1996). A Virtual hardware Operating System for the Xilinx XC6200, In: R. W. Hartenstein and M. Glesner (editors), *Field Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL Proceedings)*, Springer, pp. 327–336.

Brebner, G. (1997a). Automatic Identification of Swappable Logic Units in XC6200 Circuitry, In: W. Luk, P. Y. K. Cheung and M. Glesner (editors), *Field-Programmable Logic and Applications*, Springer, pp. 173–182.

Brebner, G. (1997b). The Swappable Logic Unit: a paradigm for virtual hardware, In: *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 77–86.

Brown, J., Chen, D., Eslick, I., Tau, E. and DeHon, A. (1994). DELTA: Prototype for a first-generatrion dynamically programmable gate array, In: *Transit Note 112*, MIT Artificial Intelligence Laboratory.

Canto, E., Fons, M., Lopez, M. and Ramos, R. (2009). Acceleration Of Complex Algorithms On A Fast Reconfigurable Embedded System On Spartan-3 , In: *Field-Programmable Logic and Applications, FPL 2009.*

Corno, F., Reorda, M. S. and Squillero, G. (2000). RT-level ICT'99 benchmarks and first ATPG results, *IEEE Design & Test of Computers* 17(3): 44–53.

Deledda, A., Mucci, C., Vitkovski, A., Bonnot, P., Grasset, A., Millet, P., Kuehnle, M., Ries, F., Huebner, M., Becker, J., Coppola, M., Pieralisi, L., Locatelli, R., Maruccia, G., Campi, F. and DeMarco, T. (2008). Design of a HW/SW communication infrastructure for a heterogeneous reconfigurable processor, In: *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, ACM, New York, NY, USA, pp. 1352–1357.

Dyer, M., Plessl, C. and Platzner, M. (2002). Partially Reconfigurable Cores for Xilinx Virtex, In: M. Glesner, P. Zipf and M. Renovell (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 292–301.

Ebeling, C., McMurchie, L., Hauck, S. and Burns, S. (1995). Placement and routing tools for the triptych fpga, *IEEE Transactions on VLSI Systems* 3(4): 473–482.

ElGindy, H., Middendorf, M., Schmeck, H. and Schmidt, B. (2000). Task Rearrengement on Partially Reconfigurable FPGAs with Restricted Buffer, In: R. W. Hartenstein and H. Grunbacher (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 379–388.

Emmert, J. M. and Bhatia, D. K. (1999). Tabu Search: Ultra-Fast Placement for FPGAs, In: P. Lysaght, J. Irvine and R. Hartenstein (editors), *Field-Programmable Logic and Applications*, Springer, pp. 81–90.

Esquiagola, J., Ozari, G., Teruya, M., Strum, M. and Chau, W. (2005). A Dynamically Reconfigurable Bluetooth Base Band Unit, In: T. Rissa, S. Wilton and P. Leong (editors), *2005 International Conference on Field Programmable Logic and Applications*, pp. 148–152.

Eto, E. (2007). Xapp290: Difference-based partial reconfiguration, *Technical report*, Xilinx.

Fahmy, S. A., Bouganis, C.-S. and Cheung, P. Y. K. (2006). Efficient Realtime FPGA Implementation Of The Trace Transform, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 555–560.

Fahmy, S. A., Cheung, P. Y. K. and Luk, W. (2005). Novel FPGA-Based Implementation Of Median And Weighted Median Filters ForImage Processing, In: T. Rissa, S. Wilton and P. Leong (editors), *2005 International Conference on Field Programmable Logic and Applications*, pp. 142–147.

Fiduccia, C. and Mattheyses, R. (1984). A linear time heuristic for improving network partitions, In: *Design Automation Conference*, pp. 175–181.

Fiethe, B., Michalik, H., Dierker, C., Osterloh, B. and Zhou, G. (2007). Reconfigurable system-on-chip data processing units for space imaging

instruments, In: *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, EDA Consortium, San Jose, CA, USA, pp. 977–982.

Gajski, D., Dutt, N., Wu, A. and Lin, S. (1992). *High Level Synthesis Introduction to Chip and System Design*, Kluwer Academic Publishers.

Garcia, M. L. and Navarro, E. F. C. (2006). FPGA Implementation Of A Ridge Extraction Fingerprint Algorithm Based On Microblaze And Hardware Coprocessor, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 79–83.

Gause, J., Cheung, P. Y. K. and Luk, W. (2000). Static and Dynamic Reconfigurable designs for a 2D Shape-Adoptive DCT, In: R. W. Hartenstein and H. Grunbacher (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 96–105.

Gerez, S. H. (1999). *Algorithms for VLSI Design Automation*, John Wiley and Sons.

Gericota, M. G., Alves, G. R., Silva, M. L. and Ferreira, J. M. (2002). On-line defragmentation for Run-Time Partially Reconfigurable FPGAs, In: M. Glesner, P. Zipf and M. Renovell (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 302–311.

Glas, B., Klimm, A., Sander, O., Müller-Glaser, K. and Becker, J. (2008). A system architecture for reconfigurable trusted platforms, In: *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, ACM, New York, NY, USA, pp. 541–544.

Gorgon, M., Cichon, S. and Pac, M. (2005). Real-Time Handel-C Based Implementation of DV Decoder, In: T. Rissa, S. Wilton and P. Leong

(editors), *2005 International Conference on Field Programmable Logic and Applications*, pp. 130–135.

Hadley, J. D. and Hutchings, B. L. (1995). Design menthodologies for partially reconfigured systems, In: *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 138–146.

Hauck, S. and DeHon, A. (editors) (2008). *Reconfigurable Computing. The Theory And Practice Of FPGA-Based Computation*, Morgan Kaufmann Publishers.

Helmschmidt, J., Schuller, E., Rao, P., Rossi, S., di Matteo, S. and Bonitz, R. (2003). Reconfigurable Signal Processing in Wireless Terminals, In: D. Sciuto and D. Verkest (editors), *Designers' Forum (DATE '03)*, IEEE Computer Society, March, pp. 244–249.

Herrero, A. F., Jimenez-Pacheco, A., Caffarena, G. and Quiros, J. C. (2006). Design And Implementation Of A Hardware Module For Equalisation In A 4G Mimo Receiver, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 765–768.

Hormati, A., Kudlur, M., Mahlke, S., Bacon, D. and Rabbah, R. (2008). Optimus: efficient realization of streaming applications on FPGAs, In: *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, ACM, New York, NY, USA, pp. 41–50.

http://www.asics.ws (2005).

Hu, Y., Feng, Z., He, L. and Majumdar, R. (2008). Robust FPGA resynthesis based on fault-tolerant Boolean matching, In: *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, IEEE Press, Piscataway, NJ, USA, pp. 706–713.

Huang, C. and Vahid, F. (2008). Dynamic coprocessor management for FPGA-enhanced compute platforms, In: *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, ACM, New York, NY, USA, pp. 71–78.

Huffmire, T., Brotherton, B., Callegari, N., Valamehr, J., White, J., Kastner, R. and Sherwood, T. (2008). Designing secure systems on reconfigurable hardware, *ACM Trans. Des. Autom. Electron. Syst.* 13(3): 1–24.

Kannan, P. and Bhatia, D. (2001). Tightly Integrated Placement and Routing for FPGAs, In: G. Brebner and R. Woods (editors), *Field-Programmable Logic and Applications*, Springer, pp. 233–242.

Kannan, P., Balachandran, S. and Bhatia, D. (2001). fGREP - Fast Generic Routing Demand Estimation for Placed FPGA Circuits, In: G. Brebner and R. Woods (editors), *Field-Programmable Logic and Applications*, Springer, pp. 37–47.

Kannan, P., Balachandran, S. and Bhatia, D. (2002). Rapid and Reliable Routability Estimation for FPGAs, In: M. Glesner, P. Zipf and M. Renovell (editors), *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, Springer, pp. 242–252.

Karro, J. and Cohoon, J. (2001). Gambit: A Tool for the Simultaneous Placement and Detailed Routing of Gate-Arrays, In: G. Brebner and R. Woods (editors), *Field-Programmable Logic and Applications*, Springer, pp. 243–253.

KDevelop (n.d.). Kdevelop3, http://www.kdevelop.org/.

Kean, T. A. (1988). *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*, PhD thesis, University of Edinburgh.

Kielbik, R., Moreno, J. M., Napieralski, A., Jablonski, G. and Szymanski, T. (2002). High-Level Prtitioning of Digital Systems Based on Dynamically Reconfigurable Devices, In: M. Glesner, P. Zipf and M. Renovell (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 271–280.

Kirischian, V., Geurkov, V. and Kirischian, L. (2008). A multi-mode video-stream processor with cyclically reconfigurable architecture, In: *CF '08: Proceedings of the 5th conference on Computing frontiers*, ACM, New York, NY, USA, pp. 105–106.

Koch, D., Beckhoff, C. and Teich, J. (2009). Hardware Decompression Techniques for FPGA-Based Embedded Systems, *ACM Trans. Reconfigurable Technol. Syst.* 2(2): 1–23.

Kulmala, A., Hamalainen, T. D. and Hannikainen, M. (2006). Reliable GALS Implementation of MPEG-4 Encoder with Mixed ClockFIFO on Standard FPGA, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 495–500.

Legat, U., Biasizzo, A. and Novak, F. (2009). Partial Runtime Reconfiguration of FPGA, Applications and a Fault Emulation Case Study, *International Review on Computers & Software* 4(6): 606–611.

Lewis, D., Ahmed, E., Cashman, D., Vanderhoek, T., Lane, C., Lee, A. and Pan, P. (2009). Architectural enhancements in Stratix-III and Stratix-IV, In: *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, New York, NY, USA, pp. 33–42.

Luk, W., Guo, S., Shirazi, N. and Zhuang, N. (1996). A Framework for Developing Parametrised FPGA Libraries, In: R. W. Hartenstein and

M. Glesner (editors), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, Springer, pp. 24–33.

Lysaght, P. and Dunlop, J. (1994). Dynamic Reconfiguration of FPGAs, In: W. R. Moore and W. Luk (editors), *More FPGAs*, Abingdon EECS Books, pp. 82–94.

Marescaux, T., Mignolet, J. Y., Bartic, A., Moffat, W., Verkest, D., Vernalde, S. and Lauwereins, R. (2003). Networks on Chip as Hardware Components of an OS for Reconfigurable Systems, In: P. Y. K. Cheung, G. A. Constantinides and J. T. de Sousa (editors), *Field-Programmable Logic and Applications*, Springer, pp. 595–605.

Mazzeo, A., Romano, L., Saggese, G. P. and Mazzocca, N. (2003). FPGA-based Implementation of a serial RSA processor, In: N. Wehn and D. Verkest (editors), *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, IEEE, IEEE Computer Society, pp. 582–587.

McCulloch, S. T. and Cohoon, J. P. (2003). Quark Routing, In: P. Y. K. Cheung, G. A. Constantinides and J. T. de Sousa (editors), *Field-Programmable Logic and Applications*, Springer, pp. 131–140.

Mentens, N., Sakiyama, K., Batina, L., Verbauwhede, I. and Preneel, B. (2006). FPGA-Oriented Secure Data Path Design: Implementation Of A Public Key Coprocessor, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 133–138.

Michalski, E. A. and Buell, D. A. (2006). A Scalable Architecture For RSA Cryptography On Large FPGAs, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 145–152.

Mignolet, J. Y., Nollet, V., Coene, P., Verkest, D., Vernalde, S. and Lauwereins, R. (2003). Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip, In: N. Wehn and D. Verkest (editors), *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, IEEE, IEEE Computer Society, pp. 986–991.

Mulpuri, C. and Hauck, S. (2001). Runtime and quality tradeoffs in fpga placement and routing, In: *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 29–36.

Note, J.-B., Shand, M. and Vuillemin, J. E. (2006). Real-Time Video Pixel Matching, In: A. Koch, P. Leong and E. Boemo (editors), *2006 International Conference on Field Programmable Logic and Applications*, pp. 507–512.

Opencores (2005). http://www.opencores.org.

PACT XPP Technologies (2003). http://www.pactcorp.com, PACT XPP.

Pilotto, C., Azambuja, J. R. and Kastensmidt, F. L. (2008). Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications, In: *SBCCI '08: Proceedings of the 21st annual symposium on Integrated circuits and system design*, ACM, New York, NY, USA, pp. 199–204.

Rana, V., Murali, S., Atienza, D., Santambrogio, M. D., Benini, L. and Sciuto, D. (2009). Minimization of the reconfiguration latency for the mapping of applications on FPGA-based systems, In: *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ACM, New York, NY, USA, pp. 325–334.

Resano, J., Clemente, J. A., Gonzalez, C., Mozos, D. and Catthoor, F. (2008). Efficiently scheduling runtime reconfigurations, *ACM Trans. Des. Autom. Electron. Syst.* 13(4): 1–12.

Rissa, T. and Niittylahti, J. (2000). A Hybrid Prototyping Platform for Dynamically Reconfigurable Designs, In: R. W. Hartenstein and H. Grunbacher (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 371–378.

Sawitzki, S., Kohler, S. and Spallek, R. G. (2001). Prototyping Framework for Reconfigurable Processors, In: G. Brebner and R. Woods (editors), *Field-Programmable Logic and Applications (FPL)*, Stringer, pp. 6–16.

Sedcole, N. P., Cheung, P. Y. K., Constantinides, G. A. and Luk, W. (2003). A Reconfigurable Platform for Real-Time Embedded Video Image Processing, In: P. Y. K. Cheung, G. A. Constantinides and J. T. de Sousa (editors), *Field-Programmable Logic and Applications*, Springer, pp. 606–615.

Shahookar, K. and Mazumder, P. (1991). Vlsi cell placement techniques, *ACM Computing Surveys* 23(2): 143–220.

Sherwani, N. (1995). *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers.

Shirazi, N., Luk, W. and Cheung, P. Y. (2000). Framework and tools for run-time reconfiguration, In: *Computers Digital Technics*, Vol. 3, IEE, pp. 147–152.

Shirazi, N., Luk, W. and Cheung, P. Y. K. (1998). Automatic production of run-time reconfigurable designs, In: *6th Symposium on Field-Programmable Custom Computing Machines*, pp. 147–156.

Smit, G. J. M., Havinga, P. J. M., Smit, L. T., Heysters, P. M. and Rosien, M. A. J. (2002). Dynamic Reconfiguration in Mobile Systems, In: M. Glesner, P. Zipf and M. Renovell (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 171–181.

So, H. K.-H. and Brodersen, R. (2008). A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH, *ACM Trans. Embed. Comput. Syst.* 7(2): 1–28.

Stepien, P. and Vasilko, M. (2006). On Feasibility of FPGA Bitstream Compression During Placement and Routing, In: *Proceedings of 2006 International Conference on Field Programmable Logic and Applications (FPL)*, pp. 749–752.

Sterpone, L., Aguirre, M., Tombs, J. and Guzmán-Miranda, H. (2008). On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications, In: *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, ACM, New York, NY, USA, pp. 336–341.

Subramanian, P., Patil, J. and Saxena, M. K. (2009). FPGA prototyping of a multi-million gate System-on-Chip (SoC) design for wireless USB applications, In: *IWCMC '09: Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing*, ACM, New York, NY, USA, pp. 1355–1358.

Trimberger, S., Carberry, D., Johnson, A. and Wong, J. (1997). A time-multiplexed FPGA, In: *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pp. 22–28.

Vasilko, M. (2000a). *Design Synthesis for Dynamically Reconfigurable Logic Systems*, PhD thesis, Bournemouth University.

Vasilko, M. (2000b). Design Visualisation for Dynamically Reconfigurable Systems, In: R. W. Hartenstein and H. Grunbacher (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 131–140.

Vasilko, M. and Ait-Boudaoud, D. (1996a). Architectural Synthesis Techniques for Dynamically Reconfigurable Logic, In: R. W. Hartenstein and M. Glesner (editors), *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, Springer, pp. 290–296.

Vasilko, M. and Ait-Boudaoud, D. (1996b). Optically Reconfigurable FPGAs: Is This a Future Trend?, In: R. W. Hartenstein and M. Glesner (editors), *Field-Programmable Logic. Smart Applications, New Paradigms and Compilers*, Springer, pp. 270–279.

Vasilko, M., Machacek, L., Matej, M., Stepien, P. and Holloway, S. (2001). A rapid prototyping methodology and platform for seamless communication systems, In: *12 International Workshop on Rapid System Prototyping*, pp. 70–76.

Vissers, K. A. (2003). Parallel Processing Architectures for Reconfigurable Systems, In: N. Wehn and D. Verkest (editors), *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, IEEE, IEEE Computer Society, pp. 396–397.

Vladimirova, T. and Wu, X. (2007). A Reconfigurable System-on-Chip Architecture for Pico-Satellite Missions, In: A. A. McEwan, S. A. Schneider, W. Ifill and P. H. Welch (editors), *The 30th Communicating Process Architectures Conference, CPA 2007, organised under the auspices of WoTUG and the University of Surrey, Guildford, Surrey, UK*, Vol. 65 of *Concurrent Systems Engineering Series*, IOS Press, pp. 493–502.

Vogt, T. and Wehn, N. (2008). A reconfigurable application specific instruction set processor for convolutional and turbo decoding in a SDR environment, In: *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, ACM, New York, NY, USA, pp. 38–43.

Walder, H. and Platzner, M. (2003). Online Scheduling for Block-partitioned Reconfigurable Devices, In: N. Wehn and D. Verkest (editors), *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, IEEE, IEEE Computer Society, pp. 290–295.

Wang, L., French, M., Davoodi, A. and Agarwal, D. (2006). FPGA Dynamic Power Minimization through Placement and Routing Constraints, *EURASIP Journal on Embedded Systems*.

Xilinx (1999). Xc4000, http://www.xilinx.com/support/documentation/xc4000.htm.

Xilinx (2000). Virtex series configuration architecture user guide, *Application Note XAPP151*, Xilinx.

Xilinx (2001). Virtex 2.5V Field Programmable Gate Arrays, Product Specification, *Data sheet DS003*, Xilinx. http://www.xilinx.com/support/documentation/data_sheets/ds003.pdf.

Xilinx (2002). Virtex FPGA Series Configuration and Readback, *Application Note XAPP138*, Xilinx.

Xilinx (2009a). JBits SDK, http://www.xilinx.com/products/jbits/index.htm.

Xilinx (2009b). *Xilinx Virtex-5 User Guide*, Xilinx, May.

Yamaguchi, Y., Miyajima, Y., Maruyama, T. and Konagaya, A. (2002). High Speed Homology Search Using Run-Time Reconfiguration, In: M. Glesner, P. Zipf and M. Renovell (editors), *Field-Programmable Logic and Applications (FPL)*, Springer, pp. 281–291.