Optimising and evaluating designs for reconfigurable hardware

Tobias Becker

A thesis submitted for the degree of Doctor of Philosophy
Department of Computing
Imperial College London

Abstract

Growing demand for computational performance, and the rising cost for chip design and manufacturing make reconfigurable hardware increasingly attractive for digital system implementation. Reconfigurable hardware, such as field-programmable gate arrays (FPGAs), can deliver performance through parallelism while also providing flexibility to enable application builders to reconfigure them. However, reconfigurable systems, particularly those involving run-time reconfiguration, are often developed in an ad-hoc manner. Such an approach usually results in low designer productivity and can lead to inefficient designs. This thesis covers three main achievements that address this situation. The first achievement is a model that captures design parameters of reconfigurable hardware and performance parameters of a given application domain. This model supports optimisations for several design metrics such as performance, area, and power consumption. The second achievement is a technique that enhances the relocatability of bitstreams for reconfigurable devices, taking into account heterogeneous resources. This method increases the flexibility of modules represented by these bitstreams while reducing configuration storage size and design compilation time. The third achievement is a technique to characterise the power consumption of FPGAs in different activity modes. This technique includes the evaluation of standby power and dedicated low-power modes, which are crucial in meeting the requirements for battery-based mobile devices.

Statement of originality

I hereby declare that the research presented in this thesis is my own work, and to my best knowledge it contains no material previously published or written by another person, except where explicitly stated, cited or acknowledged otherwise.

Tobias Becker

Acknowledgements

First, I would like to thank my supervisor Professor Wayne Luk for his support, advice and encouragement throughout this thesis. I would also like to thank my second supervisor Professor Peter Y. K. Cheung.

I am extremely grateful to Patrick Lysaght for giving me the unique opportunity to spend several months in Xilinx Research Labs. I would also like to thank all other Xilinx Research Lab members, especially Adam Donlin and Brandon Boldget, for their advice and for making my time in Xilinx an enjoyable and rewarding experience. The idea and initial implementation for the bitstream relocation work that is presented in chapter 5 was developed during my stay at Xilinx and I would like to thank Adam and Brandon for supporting my work with their invaluable technical advice.

The idea for the placement algorithm presented in section 5.3 was developed in joint discussions with Markus Koester. I would like to thank Markus for his insights, and for proofreading and providing comments on the resulting publication.

I would like to thank Tero Rissa of Nokia R&D for arranging the GroundHog low-power benchmarking project. Collaborating with Nokia has been a very valuable experience for me. I also thank Peter Jamieson, who worked with me on this project, for an excellent and productive collaboration. The work presented in chapter 6 represents a part of the GroundHog project that was mainly developed by myself; but it was also influenced by ideas from Tero and Peter.

I would like to thank all my colleagues in the Custom Computing Group, in particular David Thomas and Timothy Todman, for insightful discussions during our regular Friday meetings. I also thank Tim for reading this thesis and providing feedback.

I would like to acknowledge the generous financial support by Xilinx that has funded a large part of this thesis. The financial support of Nokia is also gratefully acknowledged.

Finally, but by no means least, I would like to thank Ellen for her enduring support and for so much more.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In electronic systems we observe an ever-growing demand for computational power. System designers can choose from a wide range of devices and architectures to meet numerous requirements such as computational performance, power consumption and area constraints, time-to-market and cost. General purpose processors (GPPs) and Application Specific Integrated Circuits (ASICs) represent two fundamentally different implementation options. Processors are very flexible and can cover a wide range of applications. However, they are inherently inefficient because every operation requires multiple steps for fetch, decode, execute and write-back. This results in low performance and high power consumption. In contrast, ASICs are specialised for one particular use resulting in high performance and low power consumption, but they provide no flexibility.

Programmable logic such as Field-Programmable Gate Arrays (FPGAs) presents a third means of computation that can combine programmability with performance and power efficiency. FPGAs provide an array of programmable logic and routing resources and hence allow designers to implement similar circuits to those in an ASIC. The functionality is not defined at the time of chip manufacturing. Instead, a user can program the device by loading a configuration bitstream into the device. Most FPGAs are based on SRAM technology and allow reloading of new configuration bitstreams over time, updating or changing the functionality of the device. However, this flexibility comes at the cost of performance and power overheads when compared to ASICs. This overhead is incurred by the circuitry that provides the configurability of the device. Thus, FPGAs represent a trade-off between flexibility and efficiency. This is illustrated in figure 1.1.

Traditionally, FPGAs were mostly used for rapid prototyping and glue logic applications but with increasing speed, size and transistor density, they have emerged as powerful

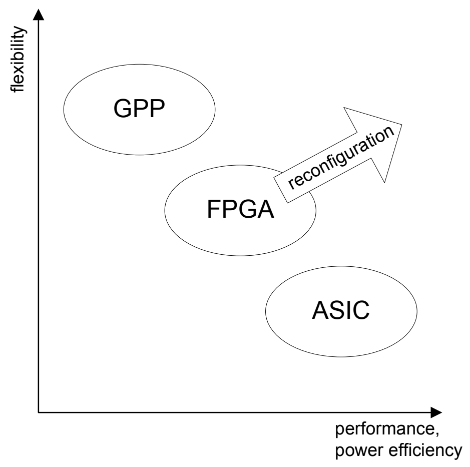


Figure 1.1: Flexibility, performance and power efficiency of digital processing devices. In FPGAs, reconfiguration can help to improve these parameters.

computational devices that can be a viable alternative to ASICs. The introduction of embedded RAMs, DSPs and processors within FPGAs, as well as peripheral functionalities such as Ethernet MACs, PCI Express and high-speed serial IOs has made FPGAs powerful platforms that can implement a wide range of high-performance applications. With the pressure of fast product development cycles and the increasing risk and cost of ASIC design, using programmable devices becomes increasingly appealing. Time-to-market, non-recurring engineering (NRE) cost and risks are reduced by targeting pre-fabricated, programmable devices. Design iterations only involve re-running the design implementation tools, while debugging can be done by adding debug logic as needed. It is also possible to provide design fixes or new features after the device has been deployed.

FPGAs can also serve as accelerators in combination with processors. A computationally expensive task can be offloaded to the FPGA which provides a dedicated hardware implementation of this task. This implementation is often many times faster than the software version. However, this approach requires analysing and partitioning the application, and mapping the hardware part to the FPGA. This is more complex than traditional software design but the effort can be justified by significant overall speed-up [32, 104].

The reconfigurability of FPGAs can be exploited to improve their flexibility, performance and power efficiency. This is illustrated in figure 1.1. By reloading a new configuration bitstream it is possible to change the functionality at run time. For example, it is possible to design several accelerator circuits that are loaded into the device based on the demand of the application. One can also design a circuit specialised to a particular condition. This circuit will be faster and more efficient than a general purpose circuit. When the condition changes, the circuit can be updated using run-time reconfiguration.

If employed systematically, reconfiguration can narrow the power and performance gap between FPGAs and ASICs, and make FPGAs a viable high-performance co-processing solution for processors.

However, designing reconfigurable systems requires additional efforts over fixed FPGA design, and is considerably more complex than traditional software development. Many reconfigurable systems are currently designed in an ad-hoc process which has several disadvantages: At the beginning of the design process it may not be clear what the efficiency of the final implementation will be, and design choices may be largely based on intuitive decisions. Once the design is implemented it may be unclear if the current solution is optimal, or it may require time-consuming iterations to improve the design. In order to become a mature design discipline, reconfigurable design needs an approach that improves both productivity and design efficiency. Key to such a design approach are simple and effective analytical models that capture the essential aspects of system behaviour. Based on such models we can devise techniques to judge the impact of design choices early on, or calculate optimal design parameters.

An orthogonal concern to efficient design is the improvement of the underlying architecture itself. Especially power and energy efficiency are increasingly important factors for both mobile and grid-connected applications. This needs to be addressed through improvements in device architecture, and mobile applications also require dedicated low-power modes.

1.2 Contributions and thesis outline

In this thesis we aim to exploit the reconfigurability of FPGAs in order to improve designs in several ways. Reconfiguration can improve performance, area and energy consumption. We develop a parametric model that allows us to improve designs for performance, area and energy consumption, including the exploration of parallelism in the implementation of an algorithm. Furthermore we want to enhance the relocatability of bitstreams on heterogeneous architectures that contain dedicated resources within the regular programmable fabric. Relocation allows us to reduce storage size and design time. We also consider the implications on floorplanning and placement, and present a technique to floorplan a design that makes use of relocatable modules. Finally, we want to characterise the power efficiency of FPGAs in different activity modes. This specifically includes the evaluation of standby power and dedicated low-power modes, and allows us to evaluate current devices and helps us to develop future devices with improved power efficiency. The key contributions of this thesis are:

- A parametric model for reconfigurable applications (chapter 4).
- Performance, area and energy optimisations based on the parametric model (chapter 4).
- A technique to enhance the relocatability of partial bitstreams (chapter 5).
- A floorplanning and placement technique for modules with enhanced relocatability (chapter 5).
- A power benchmarking framework and FPGA fabric characterisation (chapter 6).
- Case studies evaluating the proposed techniques (chapter 4, 5 and 6).

The thesis outline is as follows: Chapter 2 provides the background of this thesis. Chapter 3 presents software-defined radio as the motivating application. We analyse how software-defined radio can benefit from reconfiguration, and outline several reconfiguration scenarios. Chapter 4 shows the optimisation technique based on application, implementation and device parameters that is used to improve performance, area and power. Chapter 5 presents the module relocation technique that enhances the relocatability of partial bitstreams. Chapter 6 describes the challenges in FPGA power benchmarking and presents a technique to characterise the power consumption in FPGAs. Finally, chapter 7 presents the conclusions.

1.3 List of publications

The following publications are based on this thesis:
Chapter 3 and 4:

T. Becker and W. Luk and P.Y.K. Cheung, "Parametric Design for Reconfigurable Software-Defined Radio". In *ARC '09: Proceedings of the 5th international workshop on Reconfigurable Computing*, pp. 15–26. Springer, 2009

Chapter 4:

T. Becker and W. Luk and P.Y.K. Cheung, "Energy-Aware Optimisation for Run-Time Reconfiguration". In *Field-Programmable Custom Computing Machines*, pp. 55–62. IEEE Computer Society, 2010

Chapter 5:

T. Becker, W. Luk and P.Y.K. Cheung, “Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration”. In *Field-Programmable Custom Computing Machines*, pp. 35–44. IEEE Computer Society, 2007

T. Becker and B. Blodget, “Enhancing Relocatability of Partial Configuration Bitstreams” U.S. Patent 7,673,271 B1, Issued March 2, 2010

T. Becker, M. Koester and W. Luk, “Automated Placement of Reconfigurable Regions for Relocatable Modules”. In *International Symposium on Circuits and Systems*, pp. 3341–3344, IEEE, 2010

Chapter 6:

T. Becker, P. Jamieson, W. Luk, P.Y.K. Cheung and T. Rissa, “Power Characterisation for the Fabric in Fine-Grain Reconfiguralbe Architectures”. In *Southern Conference on Programmable Logic*, pp. 77–82, IEEE, 2009

T. Becker, P. Jamieson, W. Luk, P.Y.K. Cheung and T. Rissa, “Power Characterisation for Fine-Grain Reconfigurable Fabrics”. In *International Journal of Reconfigurable Computing*, vol. 2010, pp. 1–9, Hindawi, 2010

The following papers were co-authored during this thesis and provide context for chapter 6:

T. Becker, P. Jamieson, W. Luk, P.Y.K. Cheung and T. Rissa, “Towards Benchmarking Energy Efficiency of Reconfigurable Architectures”. In *International Conference on Field Programmable Logic and Applications*, pp. 691–694, IEEE, 2008

P. Jamieson, T. Becker, W. Luk, P.Y.K. Cheung, T. Rissa and T. Pitkänen, “Benchmarking Reconfigurable Architectures in the Mobile Domain”. In *Field-Programmable Custom Computing Machines*, pp. 131–138, IEEE Computer Society, 2009

P. Jamieson, T. Becker, P.Y.K. Cheung, W. Luk, T. Rissa and T. Pitkänen, “Benchmarking and Evaluating Reconfigurable Architectures Targeting the Mobile Domain”. In *ACM Transactions on Design Automation of Electronic Systems*, vol. 15, no. 2, pp. 1–24, ACM, 2010

Chapter 2

Background

This chapter presents a review of existing research and relevant background for this thesis. We begin with giving a general overview of FPGA technology and current devices. Next, we review relevant research and previous approaches related to our work. Section 2.1 gives a brief introduction into FPGA technology and compares several commercially available devices. In section 2.2, we give an overview over different types of reconfiguration. Run-time reconfiguration is often conceptualised as switching between several functions or modules, but by reviewing a range of previous applications, four scenarios that utilise the configurability of the FPGA in different ways can be identified. Section 2.3 presents the technical details of the reconfiguration architecture in current FPGAs and shows how these details influence the realisation of the different reconfiguration types. In the following, several aspects of reconfigurable design itself are outlined. In section 2.4, we explain the role of the configuration controller as an essential part of every reconfigurable design. It can have various responsibilities and it can be implemented within or outside the chip. A reconfigurable design also needs communication infrastructures connecting the reconfigurable and non-reconfigurable parts of the design. Several structures such as point-to-point links, bus systems or networks-on-chip are possible. This is covered in section 2.5. Designing a reconfigurable system also involves additional design efforts as explained in section 2.6. We point out the general aspects of creating a design that uses run-time reconfiguration and show the specific steps of the standard design flow and the current reconfigurable design flow. We also outline several advanced design techniques proposed in recent research. Section 2.7 discusses simulation and design space exploration for reconfigurable designs. Current tools do not support the simulation of reconfigurable designs, but design space exploration can be used to reduce time-consuming design iterations. Section 2.8 covers flexible placement and module relocation. Current design tools produce configurations that are specific to one location on the FPGA, yet often it is desirable to have more flexible configurations that can be relocated and used in several locations on the chip. We review several approaches for relocation that have been proposed, and we explain the impact of heterogeneous device resources on relocation. In section 2.9, we outline how a design can be evaluated in terms of its area, performance and power. Reconfiguration may improve any of these aspects but overheads have to be considered. Section 2.10 addresses power consumption in FPGAs and reviews several power optimisation approaches that are currently being employed. We also compare existing devices with low-power modes. Finally, section 2.11 summarises this chapter.

2.1 FPGA technology

Field-programmable gate arrays are a type of programmable logic device (PLD). PLDs provide a number of programmable logic functions and programmable interconnect that can be used to configure the device to a certain functionality. PLDs range in their complexity and flexibility, and FPGAs are amongst the most complex and flexible PLDs. FPGAs are characterised by a highly flexible interconnect that allows to implement a large range of functions. Figure 2.1 illustrates a typical FPGA architecture. The device is composed of tiles which contain a logic block and a switch matrix. The logic block contains programmable logic and connections to other logic can be made by configuring the switch matrix. Xilinx calls such a tile a Configurable Logic Block (CLB) and Altera uses the term Logic Array Block (LAB). The interconnect architecture inside a CLB or LAB allows routing to all neighbouring tiles and also to a number of logic blocks that are located several tiles away. A limited number of long range connections are also available. The interconnect takes up most of the FPGA area and represents a significant overhead in terms of chip area, signal delay and power compared with non-programmable hardware such as ASICs.

A logic block contains look-up tables (LUTs) and flip flops as basic logic elements. Combinatorial logic is implemented using LUTs, while flip flops are used to create registers. A LUT/flip flop pair is called logic cell or logic element and a logic block usually contains several of these logic cells. Figure 2.2 shows the simplified structure of a logic cell. A logic cell typically contains a look-up table (LUT) that can implement a 4-bit logic function. A 4-input LUT can usually also be configured as a 16x1 RAM or a shift register of up 16 bits in length. The LUT output is available directly, or registered through the flip flop. Logic cells also contain additional arithmetic logic and carry chains that allow the implementation of fast adder and subtractor circuits. Many FPGAs have traditionally

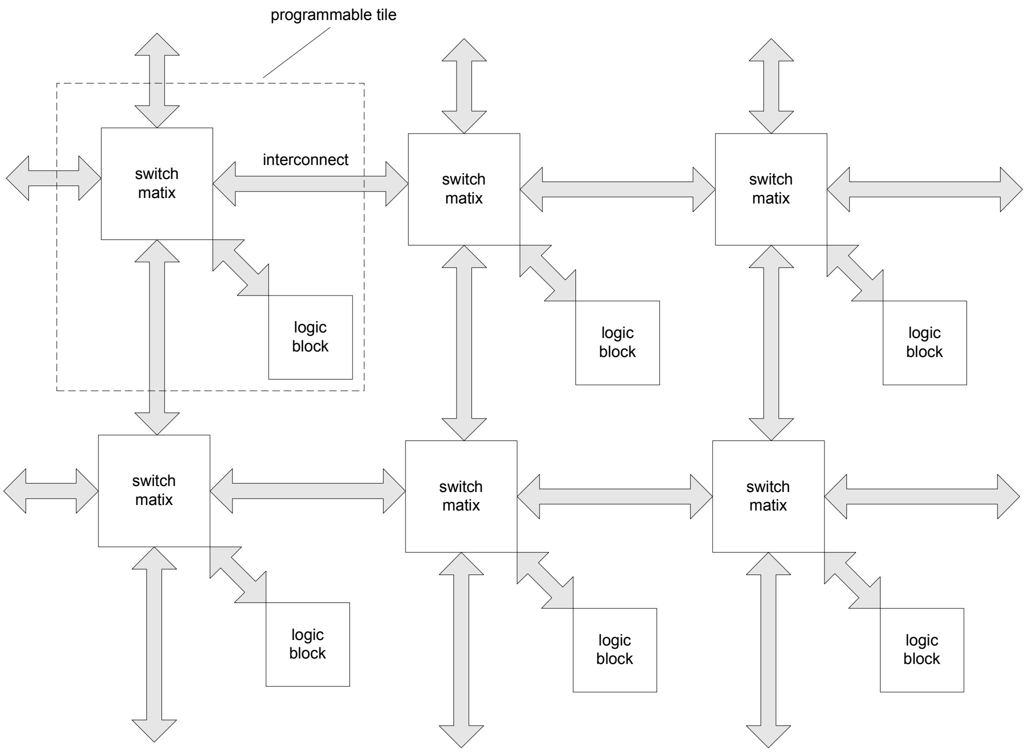


Figure 2.1: FPGA architecture.

used 4-input LUTs but more recent research suggests that LUTs with up to 6 inputs can be more efficient [3]. FPGA vendors have moved to more complex and higher-input LUT structures. Altera introduced 8-input Adaptive Logic Modules (ALMs) in their Stratix-II series FPGAs [4]. An ALM acts as a combination of two LUTs with variable input sizes. Xilinx moved to 6-input LUTs in Virtex-5 and Spartan-6 series FPGAs [129, 136]. These 6-input LUTs can also be operated as dual 5-input LUTs with 4 shared inputs.

Figure 2.3 illustrates the organisation of a CLB in Xilinx Vitex-4 FPGAs. It contains eight logic cells which are organised in four *slices*. In Virtex-5 FPGAs, CLBs also contain eight logic cells but they are organised in two slices with four logic cells each. In addition to logic blocks, FPGAs also contain IO blocks, clock buffers and clock managers as well as several dedicated clock distribution networks.

FPGAs are considered fine-grain configurable because their logic and routing architecture can be configured on bit level. This provides the highest possible flexibility, but it comes at the cost of higher performance, area and power overheads. Coarse-grain configurable devices with higher-level logic functions such as ALUs and byte-wide routing can be more efficient than FPGAs, but they are also less flexible. Hence, coarse-grain devices are usually specific to a particular domain such as signal processing and communication applications [42, 77, 95].

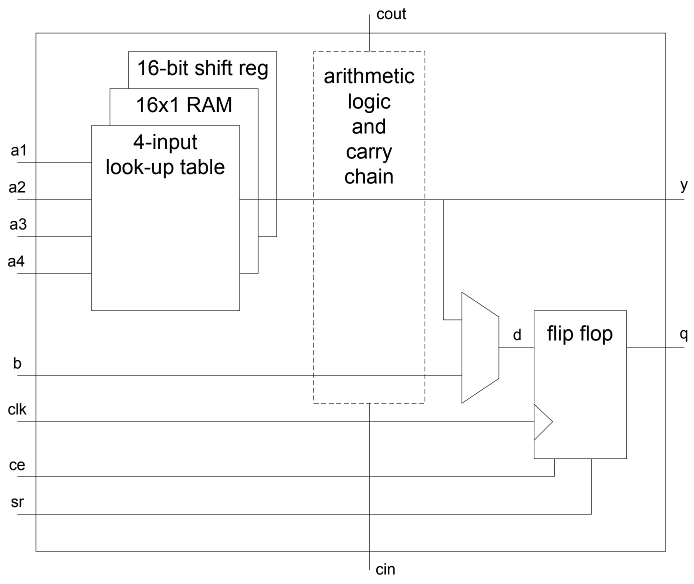


Figure 2.2: Basic structure of an FPGA logic cell.

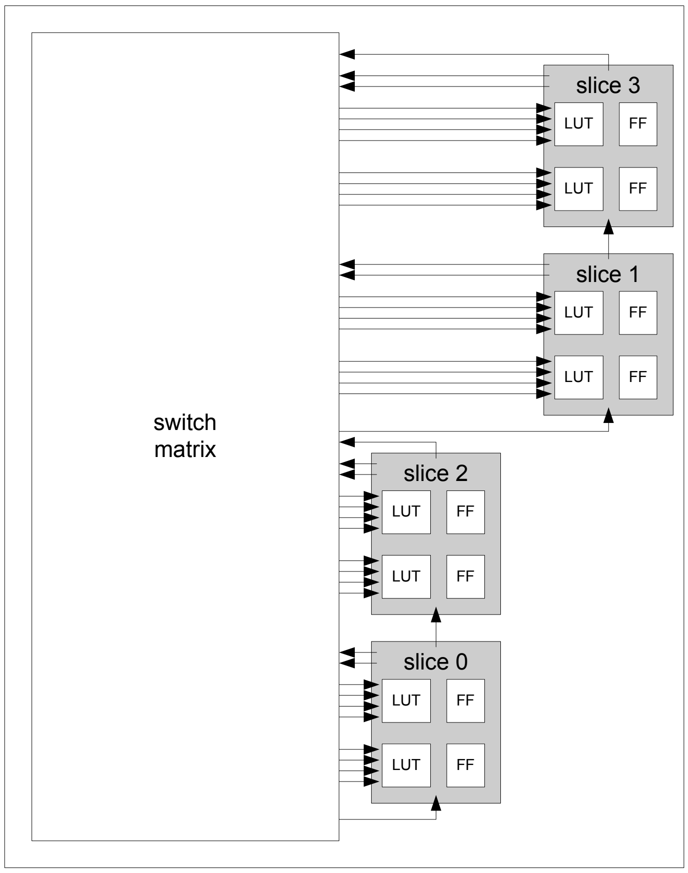


Figure 2.3: Simplified structure of a Xilinx Virtex-4 configurable logic block (CLB) containing eight logic cells organised into four slices.

Modern FPGAs also provide dedicated hardened blocks that provide a number of commonly required functionalities more efficiently than using the FPGA logic resources. This can alleviate some of the overhead incurred by fine-grain reconfigurability. For example, using a dedicated multiplier is more efficient than implementing a multiplier in FPGA logic. However, if the application does not require multiplication then the multipliers remain unused and consequently also represent an overhead. Examples of such dedicated blocks that are commonly found in current devices are RAM blocks, multipliers, DSPs and processors. These blocks are usually spread throughout the FPGA's logic fabric. The location of dedicated blocks has implications on the design floorplan, making floorplanning more complicated than in fully homogeneous FPGA fabrics. Other dedicated blocks such as Ethernet MACs, PCI express endpoints and memory controllers are usually found on the perimeter of the device. Table 2.1 compares the basic features of current commercial FPGAs.

FPGA design traditionally requires a designer to describe the application in a hardware description language (HDL) such as VHDL or Verilog. FPGA synthesis tools transform such a description into a netlist that is subsequently mapped, placed and routed on the FPGA using vendor-provided software. The design is then transformed into a configuration bitstream, usually simply referred to as bitstream, that can be loaded into the FPGA. Today, HDL-level design is deemed to be too time-consuming and complex and a wide body of research is available that targets higher-level design specifications and synthesis [20, 74, 101].

The configuration of an FPGA can be stored using antifuse, flash or SRAM technology [58]. Antifuse technology uses two metal conductors separated by an insulating silicon layer. The insulation can be permanently turned into a conductive silicon-metal alloy by applying a programming current. This technology is very area efficient and non-volatile. Furthermore, it is very difficult and expensive to observe the state of antifuses. These devices therefore offer very good protection against design cloning or reverse engineering. On the other hand, antifuse technology requires a non-standard and complex fabrication process and devices can only be programmed once, thus ruling out any form of reconfiguration. These disadvantages have relegated antifuse FPGAs to niche applications.

Flash-based FPGAs use floating-gate transistors to store the configuration and control the programmable elements of the device. Charges trapped inside the floating gate change the threshold voltage of the transistor. Flash-based FPGAs are non-volatile and low-power. They can also be reconfigured by overwriting the flash memory but configuration speeds are slow compared to SRAM-based devices. The number of programming cycles is also limited by the number of flash write cycles. This is typically lower than in regular

Manufacturer	Device	Technology	Logic Elements	Dedicated Blocks	Special Features	Reconfiguration
Actel	Axcelerator	antifuse	4-input MUX	RAM	low-power mode	no
	IGLOO, ProASIC3	flash	3-input LUTs	RAM	low-power mode	slow, full
Altera	Cyclone II, III, IV	SRAM	4-input LUTs	RAM, multiplier		full
	Stratix II	SRAM	8-input adaptive logic module	RAM, DSP		full
	Stratix III, IV	SRAM	8-input adaptive logic module	RAM, DSP	programmable power	full
Lattice	LatticeSC	SRAM	4-input LUTs	RAM		partial
	LatticeECP	SRAM	4-input LUTs	RAM, DSP		full
	LatticeXP2	SRAM	4-input LUTs	RAM, DSP	internal flash	full
SiliconBlue	iCE65	SRAM	4-input LUTs	RAM	internal flash, low-power mode	full
Xilinx	Spartan-3/3E/3A	SRAM	4-input LUTs	RAM, multiplier		partial, 1D
	Spartan-3AN	SRAM	4-input LUTs	RAM, multiplier	internal flash, low-power mode	partial, 1D
	Spartan-6	SRAM	6-input LUTs or dual 5-LUTs	RAM, DSP	internal flash, low-power mode	partial, 2D
	Virtex-II Pro	SRAM	4-input LUTs	RAM, multiplier PowerPC 405		partial, 1D
	Virtex-4	SRAM	4-input LUTs	RAM, DSPs PowerPC 405		partial, 2D
	Virtex-5, 6	SRAM	6-input LUTs or dual 5-LUTs	RAM, DSPs PowerPC 440		partial, 2D

Table 2.1: Comparison of commercial FPGAs.

flash memory, and some devices only allow 500 programming cycles [1]. Reconfigurability of these FPGAs is therefore limited to occasional in-field upgrades.

In SRAM-based FPGAs, a six-transistor static RAM cell is used to control the state of a programmable element. This technology is the most common type of FPGA today and is used in devices such as Altera Stratix and Cyclone device families, and Xilinx Virtex and Spartan device families. These devices can be manufactured in a standard CMOS process. Taking advantage of highly advanced CMOS-technology allows the creation of FPGAs with high performance and high complexity. SRAM-based FPGAs are volatile and have to be reprogrammed every time they are powered on. Some FPGAs, such as Xilinx Spartan-3 and Lattice XP2 devices, also provide an integrated flash memory for configuration storage. After power-on, the configuration is transferred from flash memory into SRAM, and the device exhibits similar instant-on behaviour to flash or antifuse devices.

Using six-transistor SRAM cells to control all programmable elements causes a significant overhead in terms of area, delay and power when compared to non-programmable devices. Kuon and Rose show that logic circuits implemented in a 90-nm CMOS FPGA can be approximately 35 times larger and 4 times slower than in an ASIC implementation using the same technology node [57]. On the other hand, it is possible to exploit the SRAM-based configuration architecture of FPGAs as an additional benefit over non-programmable devices. SRAM memory allows fast read and write access to the device's configuration even while the device is operational. This can simply be used to adapt the functionality of a design over time. It is also possible to modify a design during its run-time, or even to create designs that modify themselves in order to adapt to their environment. In the following section we provide a detailed overview over different reconfiguration types and techniques.

2.2 Reconfiguration types

The ability to reconfigure an FPGA can be used in various ways. A simple and basic technique is an *in-field upgrade* where a configuration with new features or a bug-fix is uploaded to a device after deployment. This provides additional flexibility not available in fixed hardware. In a more far-reaching interpretation, hardware reconfiguration can be seen as a software-like property: the reconfiguration of a hardware block can be compared to a context switch in a processor. Brebner introduced the idea of virtual hardware, where the FPGA is treated like memory and managed by an operating system [16]. DeHon and Wawrzynek show that traditional hardware and software boundaries are blurred by the notion of reconfigurable computing [26]. There are various concepts of how the reconfigurability of an FPGA can be exploited, and we can classify these into the following four categories:

1. Design and debug
2. In-field upgrade
3. Self-repair
4. Dynamic switching

Reconfigurability can be used for **design and debug**. Unlike ASIC-design, which requires extensive simulation and testing before tape-out, FPGAs can be frequently reprogrammed if the design does not function as desired. This enables greatly simplified and shortened design cycles: the design can be recompiled within a matter of hours or minutes and immediately loaded onto the device. Hence, FPGA design has significantly lower risk, cost and time-to market than ASIC design. The short design cycle also allows making changes late into the design process. Reconfigurability can also be used to aid the debugging of a design. Access to the device configuration allows users to probe registers or make changes to the design if necessary. Donlin *et. al.* map the content of the configuration memory into a virtual file system [31]. This enables a user to probe and modify live configuration data through a simple file system abstraction layer. All design registers can be read and set, and state can be rolled back when an error occurs. The debug scenario differs from other types of reconfiguration in that the state of configuration memory is read back. Other types of reconfiguration usually just overwrite the entire or parts of the configuration memory.

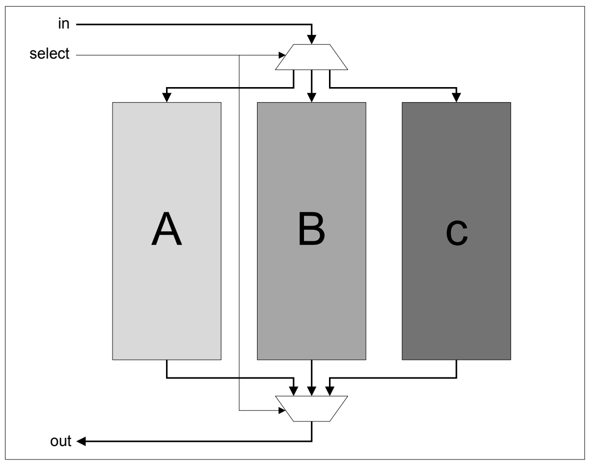
In-field upgrade is used to update, fix or extend the functionality of a design after the device has been deployed. In a set top box for example, it might be necessary to upload a new decryption or decompression standard. A base station for mobile telephone service can require an update of a communication standard. Such a field upgrade is done only occasionally and it usually involves reconfiguring the entire device. The device is powered down and then rebooted with a new configuration. An in-field upgrade can also be provided remotely. For example, Xilinx internet reconfiguration logic (IRL) allows sending a new configuration to an FPGA over a TCP/IP network [116]. It is also possible to perform a field upgrade without rebooting the entire device. If the functionality that needs reconfiguration is separated from the rest of the system and only this part of the device is reconfigured, then the rest of the system can function without interruption.

Self repair is a scenario where run-time reconfiguration is used to repair hard or soft errors on the FPGA. Hard errors are physically damaged elements of the FPGA. In such a case the circuit can be repaired by rewiring the faulty parts to alternative resources [140]. Soft errors occur when the content of a configuration memory cell is toggled into a faulty state. This is also known as single-event upset and is usually caused by high-energy charged particles. The design can be repaired by simply rewriting the original configuration to the device. It is possible to scrub the device of single-event upsets by constantly reconfiguring the device with the same configuration [120]. Such a configuration scrubbing requires a glitchless configuration memory architecture where access to a certain configuration memory location does not disturb the function of an element that is controlled by this memory location. Reconfiguration can also be used to increase the yield when process variation causes signal delays to vary from their nominal value [90].

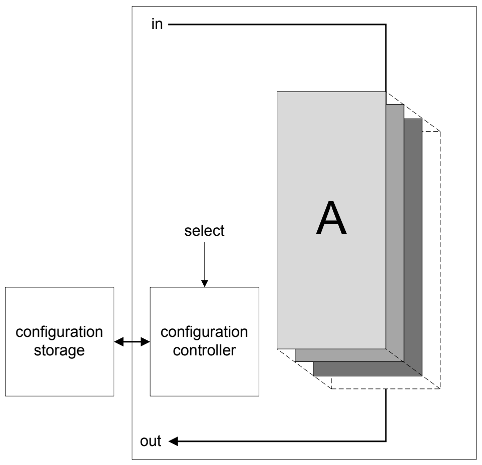
In **dynamic switching**, the design or part of the design is swapped during its run time. This is also commonly referred to as run-time reconfiguration. Dynamic switching can be seen as an analogy to a context switch in a processor: multiple hardware blocks can share the same area on the FPGA, similar to how multiple processes can share a single processor. The motivation for swapping hardware blocks is to be able to adapt to a changing environment or processing requirement. Dynamic switching is different from the previous scenarios in that reconfiguration is not just a supportive feature but that it becomes part of the designs functionality. For designs using dynamic switching we can distinguish between two different approaches:

- Module multiplexing
- Module specialisation

In *module multiplexing*, we try to identify several mutually exclusive blocks in an application. Instead of having all blocks available in hardware all the time, blocks are time-shared within the FPGA and swapped on demand. This is illustrated in figure 2.4. This type of reconfiguration has been formalised with the concept of virtual multiplexers or isolation switches [65, 68]. In essence, blocks that are multiplexed in hardware can be turned into reconfigurable blocks and the reconfiguration controller becomes the equivalent of the multiplexer. The motivation behind this is simply to save area. As a side effect, it can also lead to power reduction; the design may fit in a smaller device which consumes less

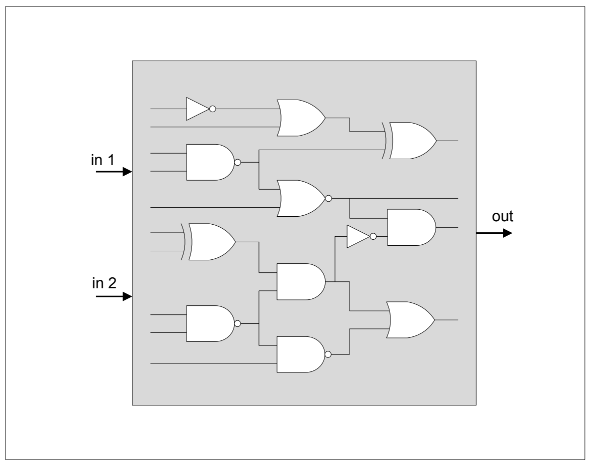


(a) Exclusive modules

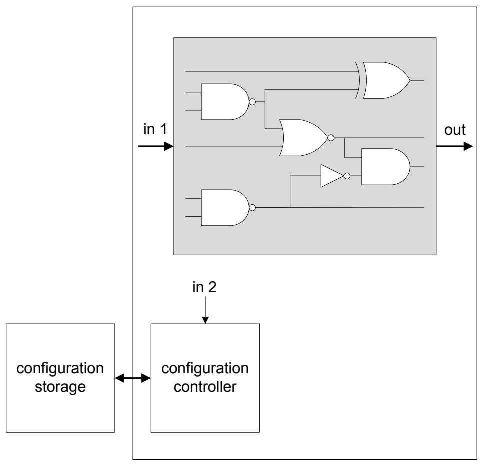


(b) Reconfigurable version

Figure 2.4: Module multiplexing: mutually exclusive blocks can be swapped through reconfiguration.



(a) Two input



(b) Specialised

Figure 2.5: Module specialisation: a circuit can be specialised for one input. Reconfiguration is used to change the input.

static power. An example of module multiplexing is an FPGA-based software-defined radio where blocks realising different communication waveforms can be loaded in the FPGA depending on the communication requirements [108].

The idea behind *module specialisation* is to take a general purpose circuit and specialise it to a set of particular circumstances or inputs. For each circumstance, one can generate a specialised circuit that is more efficient than the general purpose circuit. When the circumstances change, the circuit can be swapped against another specialised circuit. A practical example is a multiplier circuit. If one of the inputs only ever changes between a few values, then the multiplier can be specialised for each of these possible inputs. In the specialised version, the input is treated as constant and this results in a simplified fixed-coefficient multiplier circuit. To change the input, reconfiguration is used to load another specialised multiplier. This is illustrated in figure 2.5. Singh *et. al.* describe this mechanism formally as partial evaluation of hardware similar to the partial evaluation of a software function that can be specialised to a number of known inputs [97]. Advantages of a specialised circuit can be area and power reductions as well as improved performance. For example, Rissa *et. al.* demonstrate that a reconfigurable FIR filter with constant-coefficient multipliers can improve area, power and performance over a design with regular multipliers [84]. Hence, using reconfiguration for module specialisation can mitigate some of the overheads associated with reconfigurable hardware. In another example, a power-optimised Viterbi decoder can be reconfigured to adapt to the signal-to-noise ratio [100]. In this case, the design is not specialised to an input. Instead one can choose between several power-efficient but less powerful decoders for good signal-to-noise ratios or less power-efficient but stronger decoders for worse conditions.

Most reconfigurable designs have in common that the configurations are generated before run time. This is due to the relatively long compilation time that usually takes several minutes to hours. There are, however, some approaches where configuration data is generated at run-time. If the number of cases one wants to specialise for is very large, then it might be impractical to pre-compile and store all possible design variations. For example, an AES core can be optimised for one particular key but it would seem impossible to design and store optimised circuits for all possible encryption keys. As a solution one can create a circuit that can be adapted by making only a few fine-grain changes. A new input can be mapped directly to some changes in the logic or routing configuration rather than re-running the entire complication process. This reduces the complexity and the time required to generate a new configuration. An example are reconfigurable DES and AES encryption cores where changes to the encryption keys are directly translated into changes of LUT configurations [78, 73]. Key to this is a Java-based API called JBits that allows access to individual FPGA logic resources in Xilinx Virtex-II devices [40].3 In a similar approach, Bruneel *et. al.* present a tunable LUT (TLUT) concept to implement reconfigurable circuits that can be modified by changing LUT configurations [18]. This is demonstrated on the example of a reconfigurable FIR filter. It has also been proposed to reconfigure the routing between logic resources [51]. The disadvantage of this is that routing is a more complex task, and rewiring parts of the circuit at run time is usually too time consuming. Derbyshire *et. al.* present techniques to reduce the time consumed by run-time circuit generation [27].

2.3 Reconfiguration in current devices

In this section we explain the technical aspects of reconfiguration in current commercial FPGAs. We show how configuration granularity, glitching and preservation of state influence the configuration techniques presented in the previous section.

One important property of an FPGA is the granularity of the configuration memory, or, in other words, the smallest element that can be reconfigured individually. We can identify several types of configuration granularities:

- Full reconfiguration
- Partial reconfiguration
  - Module-based reconfiguration
  - Fine-grain reconfiguration

Full reconfiguration is possible in all commercial SRAM-based FPGAs. It is simply in the nature of such devices that they can be loaded with new configurations. The only exceptions are devices that use a tamper protection mechanism. Such a mechanism can stop unauthorised bitstreams from being loaded and, is available in Altera Stratix-III/IV devices [6, 7] and Xilinx Virtex-6 [137] and Spartan-6 [136] devices. Full configuration is usually used in the *design* and *in-field upgrade* scenarios. Full configuration is simple to do and does not require any additional design considerations. However, a full reconfiguration usually involves a reset. The functionality will be interrupted and all state will be lost. Full reconfiguration can also be used for *dynamic switching* but there are several disadvantages. Because of the interruption of functionality in the FPGA, this is only suitable for systems where the FPGA is a peripheral processing component that is managed by an independent controller. Reconfiguring the entire device can take a long time for large devices, and in many situations it is desirable to minimise the interruption time.

Approaches where only part of the device is reconfigured are commonly described as partial reconfiguration. Partial reconfiguration is currently only supported in Xilinx Spartan and Virtex series FPGAs but the technical details differ between device families and generations. LatticeSC FPGAs also support partial reconfiguration but there is very limited information available about further details. Partial reconfiguration is a general term and we can make several distinctions as follows.

Module-based reconfiguration is a method where reconfigurable modules can be swapped in and out in a dedicated reconfigurable region while the rest of the system

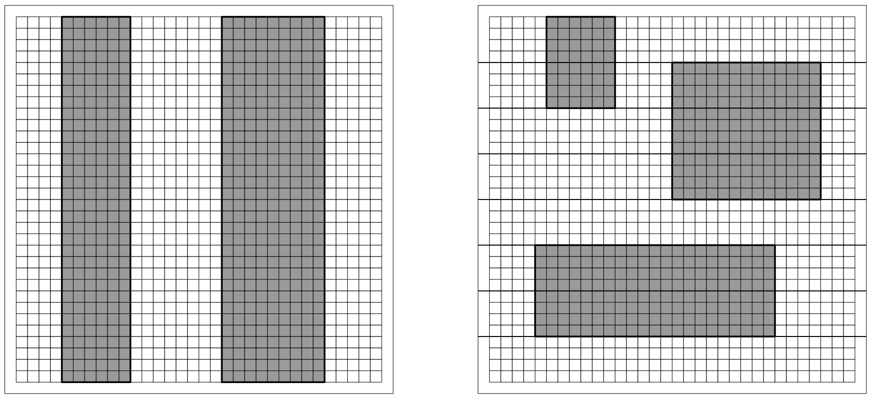

(a) One-dimensional reconfiguration (b) Two-dimensional reconfiguration

Figure 2.6: Different types of configuration granularity

continues to operate undisturbed. The part of the system which in never reconfigured is called *static*. It is possible to have several reconfigurable regions residing inside one FPGA. There are also approaches that do not require dedicated reconfigurable regions. Instead they allow modules to be loaded anywhere into the current free space. This is further explained in section 2.8. Module-based reconfiguration is performed by writing a *partial bitstream* to the FPGA's configuration memory. This type of reconfiguration is usually used for *dynamic switching*. The configuration control mechanism can also be integrated into the FPGA. This allows self-reconfigurable systems, further described in section 2.4. Module-based reconfiguration can also be used for *in-field upgrades* where a part of the system should remain operational or carries out the reconfiguration.

The granularity of reconfigurable modules and regions is determined by the configuration memory architecture, and Xilinx FPGAs exhibit two different principles. In Virtex-II, Virtex-II Pro and Spartan-3 devices, the configuration memory is organised in *frames* that span the entire height of the device. Consequently, reconfiguration always affects a full column of the device. This results in a one-dimensional arrangement of reconfigurable regions as illustrated in figure 2.6a. Virtex-4 and Virtex-5 have a finer granularity that enables two-dimensional reconfiguration. This is shown in figure 2.6b. Configuration frames cover rows of 16 CLBs in Virtex-4, and rows of 20 CLBs in Virtex-5. Reconfigurable regions within these boundaries will not affect logic above or below the region.

Fine-grain reconfiguration allows access to individual logic or routing elements. This is currently not supported in Xilinx FGPAs. However, fine-grain access can be emulated by employing a read-modify-write mechanism. This involves reading back an entire frame of configuration data, modifying the data that is associated with the element that needs to be changed and writing back the modified frame to the device.

Another aspect of the configuration architecture is glitching. An architecture is called *glitchless* if overwriting a logic or routing element with the same configuration does not cause this element to glitch, i.e. to toggle or interrupt its functionality. This is important if static logic or routing is present within a frame under reconfiguration. For example, in a one-dimensional configuration architecture one might need several static wires crossing the reconfigurable region to connect the adjacent parts. All Xilinx Virtex families are glitchless and allow static logic or routing in a reconfigurable region. However, in Virtex-II and Virtex-II Pro FPGAs, LUTs are not glitchless when used in shift-register or RAM mode [89]. Newer Virtex devices have no such restriction. Xilinx Spartan-3 FPGAs are not glitchless, and any logic or routing in a reconfigurable region will be interrupted by reconfiguration [79].

Finally we consider the preservation of state during reconfiguration. The state in a device's flip flops and memories is lost if a global reset is initiated. Current Altera FPGAs only allow full reconfiguration involving a reset and hence, cannot preserve any state. Current Xilinx FPGAs can be reconfigured without a global reset. This is an important precondition for creating designs with *module-based reconfiguration* where a part of the design remains operational during reconfiguration. In Xilinx Virtex families, the state in flip flops will be preserved even if they are located in a region that is being reconfigured. This can be used for *self-repair* through configuration scrubbing: overwriting the FPGA with the same configuration will not affect processing in the device [120]. State in block RAMs (BRAMs) can be changed through reconfiguration. This can be used as a feature if required, or BRAMs can be skipped during reconfiguration to preserve their state.

2.4 Configuration controller

A reconfigurable system usually consists of a reconfigurable FPGA, a configuration repository and a configuration controller that loads configurations from the repository into the FPGA. In its simplest form, a configuration controller can be a state machine that streams configuration data from memory into the FPGA's configuration interface. However, configuration controllers are often more complex.

Shirazi *et. al.* introduce a basic concept that consists of a loader and a monitor [94]. The loader detects requirements for reconfiguration and maintains information about the system status. If reconfiguration is required, it instructs the loader to read configuration data from the repository and load it into the device. The separation between monitor and loader is only formal and does not mean that these components have to be implemented

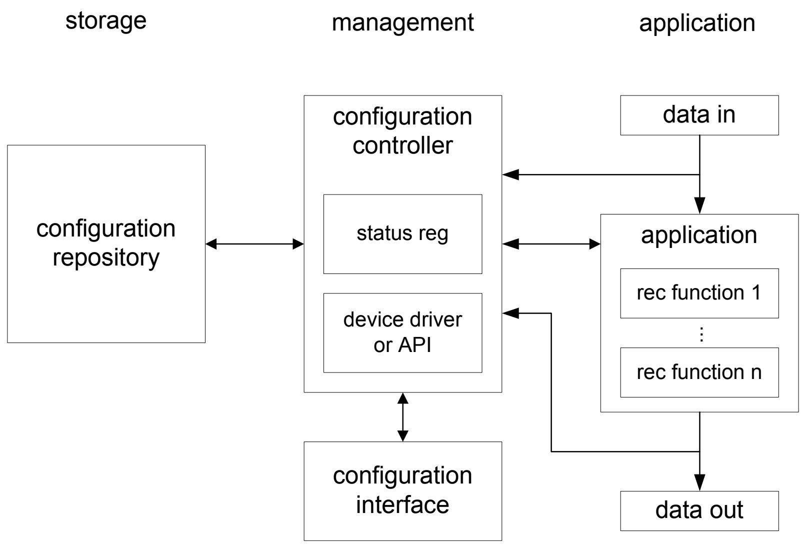


Figure 2.7: Structure of a typical reconfigurable system including configuration controller.

separately. Robinson *et. al.* present an advanced configuration controller scheme [85]. The controller includes a scheduler, pre-emption interface and layout transformer. The scheduler collects configuration requests and processes these into an output queue depending on their priority. Task pre-emption allows low-level tasks to be pre-empted by higher priority ones. The layout transformer can modify the configuration data in order to write a configuration to a different target location. Brebner *et. al.* present a configuration controller and management system that is located on the FPGA itself [17]. The controller allows flexible allocation of modules with various sizes to current free space. It can also perform defragmentation by relocating modules. Diessel *et. al.* consider a two-dimensional dynamic scheduling and placement problem that would arise from being able to freely place a reconfigurable module anywhere inside the FPGA. In a similar approach, Platzner *et. al.* analyse one-dimensional and two-dimensional scheduling and placement problems that the configuration controller has to manage [28]. The controller is implemented on a host CPU that manages the FPGA. Huebner *et. al.* propose to use configuration compression and add a decompressor module to the configuration controller [47].

Figure 2.7 shows the structure of a reconfiguration controller in a typical reconfigurable system. This illustration is conceptual and does not mean that these components have to be physically separate. Configuration can be initiated by inputs or outputs of the application, or by a direct request of the application. The controller can then perform several tasks such as checking if reconfiguration is possible, scheduling the reconfiguration, allocating a module to a free location, stalling the application and starting of the configuration process. The configuration process itself can involve retrieving configuration data from the repository, decompression of configuration data, modification or retargeting configuration data and finally loading the data into the configuration interface. The controller can make use of a device driver or device API. The following list shows tasks that a configuration controller may have to perform:

- Handshaking with the application. A module under reconfiguration cannot perform any computation and its outputs are unknown. To handle this situation the controller can disable the module outputs, stall the application or instruct the application to buffer incoming data.

- Compilation of configuration data. If it is impossible or impractical to pre-compile and store all possible configurations, the controller needs to compile configuration data.

- Configuration Scheduling. If configuration requests cannot be executed ad-hoc the controller has to schedule configurations based on their priority.

- Placement of modules. The controller may need to place reconfigurable modules on the FPGA. Placement can be fully flexible or limited to pre-defined reconfigurable regions (see section 2.8).

- Defragmentation. Fully flexible placement can cause fragmentation of the device.

- Transformation of configuration data. Relocation and flexible placement of reconfigurable modules can require a transformation of the address field or of the configuration itself.

- Decompress configuration data. If compression is used to reduce the size of the configuration storage then the configuration must be decompressed before loading it into the FPGA.

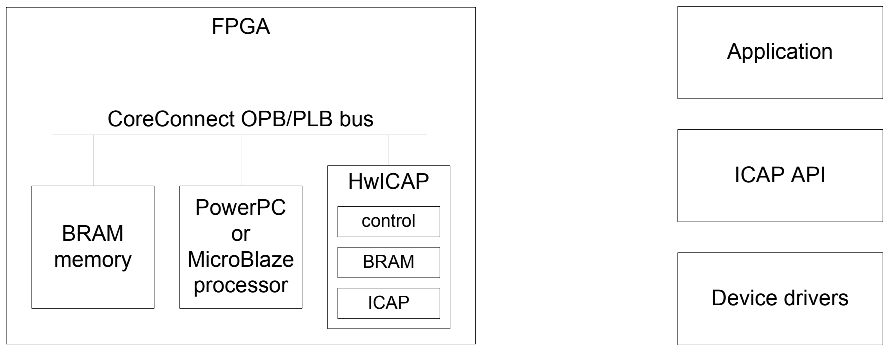
- Context save and restore. If a module is pre-empted then its state has to be saved and later restored to continue execution.

A configuration controller can be an external or internal component. External configuration controllers are usually used in systems where the FPGA is a component within a larger host system [69]. An external controller is also required for devices that support only full reconfiguration. Devices with partial reconfiguration can make use of an integrated configuration controller. This allows to create a self-modifying system with an

FPGA as the only processing component. For example, McGregor *et. al.* demonstrate the concept of an integrated controller and show that such a system would need non-volatile memory for configuration storage as the only external component [72]. Donlin proposes a reconfigurable processor where the configuration memory is mapped into the processor memory space [30]. Thus, the processor can modify itself in the same manner it accesses application data. This concept demonstrates that the boundaries between reconfiguration controller and reconfigurable application can be blurred or completely removed. Early reconfigurable research was often concerned with converting non-reconfigurable designs into reconfigurable ones and creating the necessary infrastructure. A modern approach however is to create the application under the reconfigurable paradigm. We observe that the distinction between configuration controller and application is less applicable in recent work.

Blodget *et. al.* present a self-reconfigurable platform implemented on a Xilinx Virtex-II Pro FPGA [13]. This platform makes use of the internal configuration access port (ICAP) that is available in all current Xilinx Virtex series FPGAs. ICAP is, as the name suggests, an internal configuration interface that can be directly connected to the FPGA design in order to perform partial reconfiguration of the device. Hence, it allows to create self-modifying or self-reconfigurable designs. ICAP is similar to the external SelectMAP interface and can be used with the same configuration bitstreams. In the self-reconfigurable platform, ICAP is connected to an embedded processor. The processor can reconfigure parts of the FPGA and also perform other tasks. Some devices such as Spartan-3 do not feature an internal configuration interface such as ICAP. Embedded self-reconfiguration is still possible if the external configuration interface is connected to an internal processor with some external wiring [79].

HwICAP is a Xilinx IP core developed based on the above framework [119]. This IP core is part of the Xilinx Embedded Development Kit (EDK) [121]. Xilinx EDK allows designers to develop embedded systems based on MicroBlaze soft-core processors [123] or IBM PowerPC hard-core processors [135]. These processors use IBM OPB and PLB CoreConnect bus systems and can be connected to a wide range of IP cores to create embedded processing systems. The HwICAP core encapsulates ICAP so that it can be connected to a CoreConnect bus. It provides a low-level device driver and a high-level API that provide several levels of abstraction ranging from register accesses to changing CLB configurations, writing frames or writing partial bitstreams to the FPGA. The hardware and software structure of a self-reconfigurable platform with HwICAP is illustrated in



(a) Hardware platform (b) Software platform

Figure 2.8: Self-reconfigurable platform with HwICAP IP core [13].

figure 2.8. Xilinx embedded processors and HwICAP can be used to implement a pure configuration controller or reconfigurable application with integrated configuration control. This approach is used for many self-reconfigurable systems [12, 21, 33, 59, 67, 89, 106, 114].

2.5 Communication infrastructures

An important aspect of reconfigurable designs is the communication infrastructure between the static part and reconfigurable modules or between several reconfigurable modules. Xilinx recommends to use *bus macros* as an interface between a reconfigurable region and the static part of the system [131]. A bus-macro is an 8-bit wide fixed routing structure that is placed straddling the boundary of a reconfigurable region. It creates a fixed interface for all reconfigurable modules that can be loaded into this reconfigurable region. Figure 2.9 illustrates the structure of a bus macro. It is comprised of two CLBs and signals are routed through slices in order to create a fixed routing structure between these slices and across the boundary. Xilinx bus macros are unidirectional point-to-point interfaces and multiple instances have to be placed around the region's boundary according to input and output bit width that is required.

In addition to placing bus macros across the boundaries of reconfigurable regions one has to consider the overall connectivity in the system. This is particularly important if multiple reconfigurable regions are used. A simple way of providing a global connectivity in the system is leaving free channels between the reconfigurable regions. This is illustrated in figure 2.10a. These external communication channels can then be used for various communication architectures such as point-to-point links, shared buses or networks-on-chip. External communication channels are used in [14, 56].

In one-dimensional configuration architectures such as Xilinx Virtex-II Pro it is not

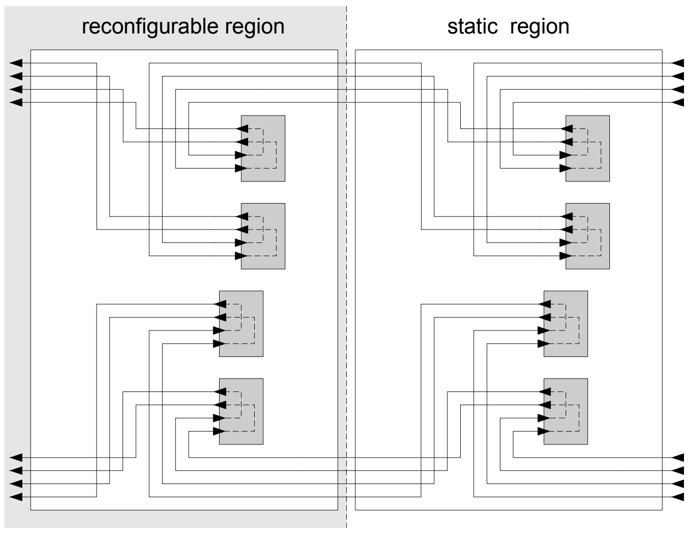
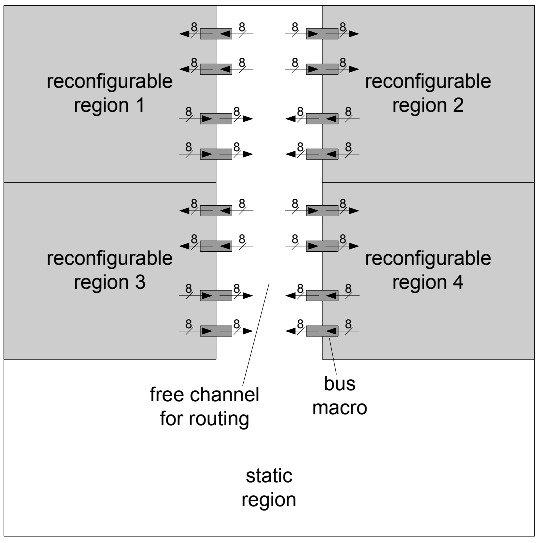
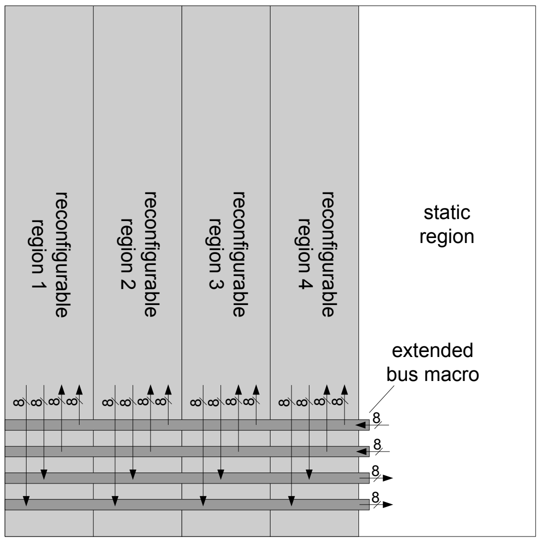


Figure 2.9: Xilinx 8-bit bus macro comprised of two CLBs [131].



(a) External communication channel



(b) Integrated communication structure

Figure 2.10: Different types of communication infrastructures.

possible to create free routing channels above or below the reconfigurable regions. Bus macros can be used at the region boundary, but communication through a region will be interrupted if this region is reconfigured with a new module. As a solution, an integrated communication structure comprised of extended bus macros can be used. This is shown in figure 2.10b. Huebner *et. al.* develop extended bus macros that stretch across four reconfigurable regions [46]. These bus macros allow uninterrupted communication across a region under reconfiguration. This scheme also includes a bus arbiter for region addressing and access control. Similar bus systems are used in [41, 49, 53].

However, bus architectures do not scale well with a growing number of modules. In larger systems, bus systems can become a throughput bottleneck. A solution can be a network-on-chip that sends packets between router nodes which are attached to reconfigurable modules. Marescaux *et. al.* propose a network with 2D torus architecture and wormhole routing on a Virtex-II FPGA [71]. The 2D torus is folded into a 1D arrangement to accommodate for the one-dimensional configuration architecture of the device. Bobda *et. al.* develop a dynamic network-on-chip with 2D grid architecture as a technique to communicate between modules in a system with many reconfigurable regions [14]. The routing algorithm supports reconfiguration of router nodes. Pionteck *et. al.* propose a configurable network-on-chip that allows to change the network topology at run time [81]. However, networks-on-chip have the disadvantage of requiring additional logic for the router nodes. In a comparison of several communication architectures, Mak finds the overheads of networks-on-chips not justified for current design sizes [70].

Koh *et. al.* develop a framework that provides an application-specific communication structure instead of a general purpose network-on-chip or bus [56]. The communication structure is generated based on a communication graph that specifies communication bandwidth requirements between tasks. The resulting communication infrastructure is then placed in free routing channels between reconfigurable modules.

Reconfiguration can also be used to change the connectivity of a communication infrastructure. Keller presents a Java-based routing API called JRoute [51]. The API can be used to reroute wires at run-time. This approach is generally not practical for reconfigurable systems because of the complexity of the routing problem. Guccione and Levi propose stitcher cores as a more lightweight alternative [39]. Stitcher cores have a predefined pattern of connectivity and can easily be changed. Young *et. al.* develop a reconfigurable crossbar switch [139]. The complexity of making connections at run time is reduced by using a regular pattern of reconfigurable multiplexers. The crossbar is a stand-alone design and is not used to connect any modules inside the FPGA.

2.6 Designing reconfigurable systems

Designing a reconfigurable system generally involves additional steps over a standard, non-reconfigurable design. The three main aspects are design input, configuration control and floorplanning. First, the design description for a reconfigurable design requires some form of partitioning. A design description is split into several reconfigurable modules and a static part that remains unchanged during the runtime of the system. The static system does not necessarily have to reside inside the FPGA. If the FPGA is used as a co-processor or accelerator then it may be used for reconfigurable modules only. In the simplest form of design partitioning, an existing hardware description is analysed for exclusive blocks of logic that are not used simultaneously [65]. These exclusive blocks can be implemented as reconfigurable modules. There are also numerous high-level approaches that consider partitioning and scheduling of a task graph. Tasks can be mapped to static logic and reconfigurable modules inside the FPGA, or to a combination of software tasks running on a processor and reconfigurable hardware tasks on the FPGA. High-level approaches face several challenges such as requiring a design description that can be efficiently mapped to both hardware and software, or necessitating realistic annotations of tasks with cost and performance parameters for their respective software and hardware implementations. There is a wide range of research that considers various aspects of the task graph partitioning and mapping problem. Banerjee *et. al.* propose a method for automatic partitioning of a task graph into hardware and software tasks [10]. Hardware tasks are mapped to a one-dimensional reconfigurable architecture and software tasks are simply executed in software. This work focusses on the partitioning and scheduling of tasks given a number of architectural constraints. In a similar approach, Cordone *et. al.* partition task graphs in order to create reconfigurable systems that support one-dimensional and two-dimensional architectures [24]. However, both [10] and [24] do not address how the functionality of a task is specified and how it can be mapped to hardware. Koh *et. al.* present a method for automatically mapping an application specified as a communication graph onto a predefined reconfigurable template [56]. The focus of this approach is the automatic generation of a communication infrastructure and the reconfigurable template is inspired by the Virtex-4 configuration architecture.

Instead of automatically partitioning a task graph or inferring reconfigurability from a high-level description, it is also possible to specify reconfigurability explicitly. Many practical reconfigurable designs are created in such a way. For example, the automotive control system presented in [12] is created with the a priori knowledge of which functions need to be reconfigured. The productivity for such designs is often low not because the designer has to choose how and when to use reconfiguration, but because of the many manual design steps involved and the lack to supporting tools. Sedcole *et. al.* propose to increase the productivity of creating reconfigurable designs through the concept of modularity and reuse [91]. Lee *et. al.* present high-level language extensions that can be used to generate reconfigurable designs from a C-like description [62]. Tasks are explicitly made reconfigurable by using a reconfigurable specifier.

Partitioning-based approaches usually result into designs with module multiplexing. Creating a design with module specialisation requires a slightly different approach. As explained in section 2.2, such a design does not involve switching between functions, but specialising one function for a certain condition and reconfiguring when the condition changes. Such a design can be created by using the partial evaluation approach presented by Singh *et. al.* where the design is specialised for one input [97]. Deciding what component should be specialised is generally left to the designer.

Designing a reconfigurable system also involves creating a configuration control mechanism. If the FPGA is part of a larger system or acts as a co-processor, then a configuration mechanism may already be present that can also be used for run-time configuration control. For standalone systems, a configuration controller has to be created and incorporated. This is often a manual process, and the mechanisms used range from simple state machines to embedded processors. The details of this subject are further explained in section 2.4. Some approaches aim to automate the generation of a configuration controller, or include its design into an overall design methodology. For example, Robinson *et. al.* create a DCS (dynamic circuit switching) framework that supports synthesising a configuration controller [85]. Designs are specified in VHDL with additional configuration schedule information. However, synthesis is not fully automated and requires some intervention by the designer. Hagemeyer *et. al.* develop a framework that maps a reconfigurable design onto an architecture that consists of multiple FPGAs [41]. The framework includes the generation of a Linux-based system host and configuration manager as well as the necessary communication infrastructure. However, design partitioning has to be done manually.

Reconfigurable designs require additional efforts for floorplanning. The only exception to this are designs that use full-device reconfiguration. In such a case, the chip is configured as a whole and each configuration can be treated independently. Designs with partial run-time reconfiguration are more complex to floorplan because the FPGA area is used by different modules over time. Floorplanning can therefore be treated as a 3D problem with time as the third axis. This is shown by Kastner *et. al.* [11]. Singhal *et. al.* propose to combine the partitioning and floorplanning problem for a known sequence of modules [98]. For this sequence, the minimum number modules that have to be reconfigured is identified. The modules are then automatically floorplanned using simulated annealing. There are also numerous approaches that shift the floorplanning problem into a placement problem that has to be solved at run time. Another common approach is the use of pre-defined reconfigurable regions which are incorporated into regular floorplanning. This is further

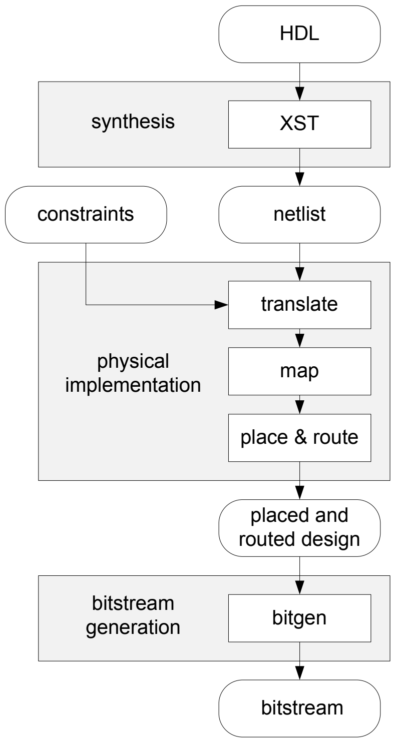


Figure 2.11: Standard Xilinx design flow.

explained in section 2.8.

In the following we highlight the differences between a standard and a reconfigurable design in the Xilinx design tools. A standard design flow with Xilinx ISE design tools [122] is outlined in figure 2.11. The design process usually starts with a description of the design in a hardware-description language (HDL) such as VHDL or Verilog. This description is then synthesised into a netlist with XST synthesis. The designer then has to set several design constraints. Typical constraints are the location of IO pins, timing constraints based on the required design performance and area constraints based on the design floorplan. The design can then be physically implemented based on the netlist and design constraints. Physical implementation involves translation of the netlist to a proprietary database of basic logic elements, mapping of these logic elements to FPGA primitives and finally, placement and routing where primitives are placed and nets between primitives are routed. The result of this process is a file that describes the physically implemented design. Based on this, a configuration bitstream is generated. Not shown are additional steps for design simulation and verification that accompany this design process. The synthesis and implementation process has to be repeated if changes to the HDL description or to the design constraints are necessary.

A reconfigurable design adds several steps to the design flow. The following summarises

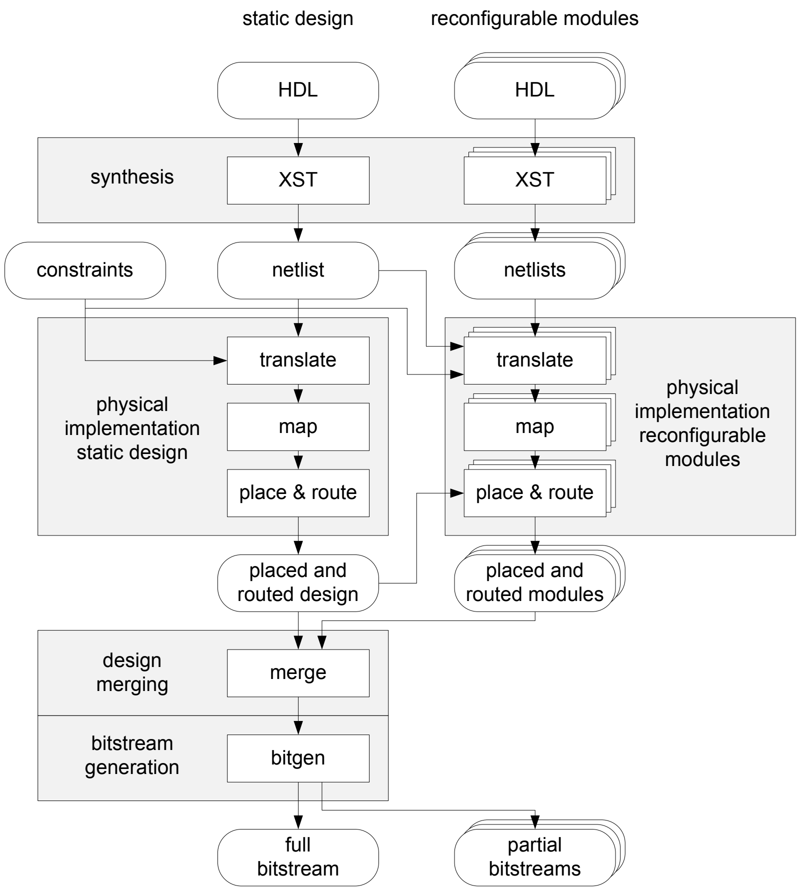


Figure 2.12: Xilinx partial reconfiguration design flow.

the design flow that is described in the Xilinx Early Access Partial Reconfiguration User Guide [131]. Early access partial reconfiguration software is a number of patches for Xilinx ISE tools which add support for creating reconfigurable designs. The design flow is illustrated in figure 2.12. The design process begins with the design description that is partitioned into a static part and several reconfigurable modules. The static part also acts as a top-level design that contains one or several reconfigurable black boxes. These black boxes act as a placeholder for reconfigurable modules and will later form reconfigurable regions. The Xilinx tool flow does not provide guidelines or mechanisms for design partitioning. Once the HDL description is partitioned, bus macros have to be inserted between the static part and the reconfigurable black boxes. The static part and all reconfigurable modules are synthesised separately. Design constraints have to be set for IO pins and timing similar to non-reconfigurable designs. In reconfigurable designs, additional constraints are necessary for the placement of reconfigurable regions as well as for the placement of bus macros. Next, the static design is physically implemented. This implementation will leave the reconfigurable region empty. Based on this static design, all reconfigurable modules are subsequently implemented within the reconfigurable region. The physical implementation of the static design is then merged into combinations with all reconfigurable modules, and full and partial bitstreams are generated. A full bitstream configures the entire device and can be used to initialise the design. Partial bitstreams only affect the reconfigurable region and can be used to swap modules at run time. For a reconfigurable design it may also be necessary to iterate the design process and make changes to the design description or the design constraints. Since all reconfigurable modules are implemented based on the static design, any changes to static design or the constraints will require a reimplementation of all reconfigurable modules. In order to minimise time-consuming design iterations it is also possible to test and verify the design first by building non-reconfigurable versions of the design that consist of the static part in combination with one of the reconfigurable modules. The Xilinx Early Access Partial Reconfiguration design flow also offers some guidelines and examples on how to create an integrated configuration controller based on embedded MicroBlaze or PowerPC processors. Such a control mechanism uses a stand-alone software application to manage and control configuration. However, there is no general co-design methodology for self-reconfigurable systems.

A key aspect in the above partial reconfiguration design flow is the overall floorplanning including the placement of reconfigurable regions. The placement of such regions is influenced by the granularity of the configuration architecture. As explained in section 2.3, devices have different configuration frame sizes. Traditionally, reconfigurable regions encompass full configuration frames. A floorplan is therefore limited to a one- or two-dimensional pattern determined by the configuration granularity. Sedcole *et. al.* propose to use smaller reconfigurable regions that do not cover full configuration frames [89]. In a Virtex-II Pro FPGA, a device with full column frames as illustrated in figure 2.6a, a system is developed that contains static logic above and below the reconfigurable region. This is possible because the configuration architecture is glitchless and hence, the functionality of logic is not interrupted if overwritten with identical configuration data. However, LUTs must not be used in RAM or shift-register mode. The current Xilinx Early Access Design Flow allows reconfigurable regions with arbitrary boundaries regardless of the configuration granularity. For Virtex-II and Virtex-II Pro the tools automatically add placement restrictions for static logic using LUT RAM or shift registers above or below a reconfigurable region. In Virtex-4 and Virtex-5 it is also possible to create regions that are not aligned with configuration frame boundaries. There are no restrictions for the use of LUT RAM or shift registers but configuration bitstreams will be larger because they include static logic. It is therefore recommended to restrict the dimensions of reconfigurable regions to frame boundaries in the configuration architecture in order to minimise configuration size and time overheads. When floorplanning reconfigurable regions the designer must also ensure that a region is large enough for all modules to be implemented within this region. Finally, the designer must place bus macros that are associated with a region such that they straddle the region boundary. All these steps can be done manually by setting constraints in a text editor. Xilinx also offers a graphic floorplanning tool called PlanAhead [124]. This tool simplifies floorplanning of reconfigurable designs with graphic visualisations and design rule checks.

2.7 Simulation and design space exploration

Simulation and design space exploration allow a designer to verify the design and evaluate the impact of design choices early on. Simulation can help finding design bugs early, without having to run through the entire design flow. The Xilinx design flow as described in section 2.6 lacks the possibility to simulate reconfigurable designs as such. Instead it is recommended to build all non-reconfigurable design versions with static part in all possible combinations of reconfigurable modules [131]. These non-reconfigurable designs can then be used to verify functionality and evaluate performance.

Other work focusses on simulating reconfigurable designs directly. A simple simulation model for reconfiguration itself are control blocks that can be implemented as real multiplexers or virtual multiplexers through reconfiguration [65]. This model can be used to describe a reconfigurable design and minimise the amount of logic that needs to be reconfigured when switching between two modules. However, the multiplexer model in [65] does not consider the time required for the reconfiguration process. Furthermore, the model cannot express the disruption of functionality during reconfiguration. The DCS framework expresses reconfiguration through isolation switches which can model the absence of a module during the reconfiguration process [85]. The framework allows to simulate designs that have been annotated with reconfiguration schedule information. A netlist post processor inserts isolation switches around the annotated components and the design can then be simulated with a standard simulation tool. The framework allows to include area and performance estimates into the simulation. It is also proposed to make simulation more accurate by back-annotating timing information from a physical implementation of the design [86].

Several more recent simulation approaches are based on SystemC. SystemC is a set of C++ classes and libraries that allows event-driven simulation of systems. However, SystemC can generally not be synthesised. Schallenberg *et. al.* create a language based on SystemC to model and simulate reconfigurable systems [87]. The language uses a synthesisable subset of SystemC so that a design can be simulated and synthesised from the same description. In order to obtain accurate performance models, the design has to run through the implementation tools and timing information has to be back-annotated to the simulation model. The configuration controller also has to be created manually. Amicucci *et. al.* develop a SystemC-based tool that allows to simulate designs targeted for the Caronte architecture [8]. The tool is used to simulate the performance of a design based on workload and number of reconfigurable elements but its applicability is limited to the proposed architecture.

For an accurate simulation model it is always necessary to back-annotate design parameters from a physically implemented design. Simulation can therefore also suffer from incurring long delays through hardware compilation. Design space exploration can help to reduce design iterations by trying to evaluate the influence of some design choices early in the design phase. For example, it may be possible to estimate the impact of design aspects such as bit-width or the degree of pipelining on the design without having to build and measure all possible choices. The accuracy of an estimation may be improved by building one design and extrapolating from there.

Potter *et. al.* propose a design space exploration technique where a datapath can be partitioned into smaller blocks [82]. Blocks can share a hardware resource or be implemented independently. The method is used to explore possible implementations of a JPEG encoder with regard to their area and performance and to find Pareto optimal points in the area-performance space. Area and performance values are high-level estimates based on the number of required blocks and cycles. Schumacher *et. al.* present a method for fast resource estimation [88]. The estimation is performed based on an RTL description of the design and it is significantly faster than obtaining the number of required resources through synthesis of the design.

2.8 Flexible placement and bitstream relocation

A regular reconfigurable design that is created through the Xilinx partial reconfiguration design flow as described in section 2.6 employs location-specific reconfigurable modules. If a design contains several reconfigurable regions, independent instances of a module have to be implemented for each region even if these regions are identical. This is illustrated in

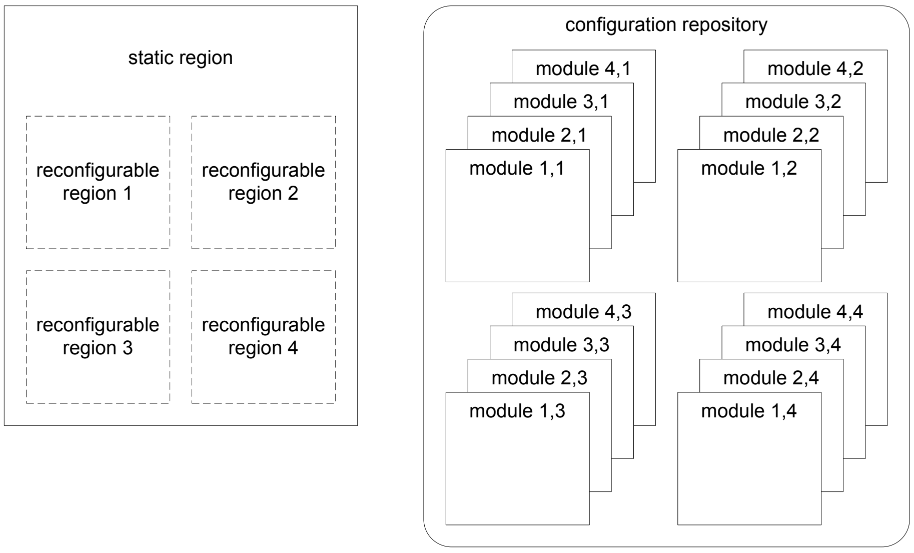


Figure 2.13: A standard reconfigurable design with non-relocatable modules. Each region needs its own set of modules.

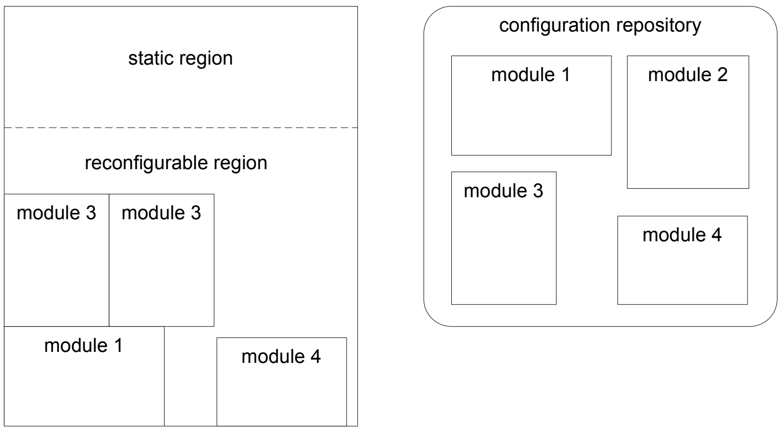


Figure 2.14: A reconfigurable design with fully flexible placement of relocatable modules.

figure 2.13. Having to implement separate modules for each region causes an overhead in compilation time and storage size.

A wide range of research considers a more flexible approach where modules are not fixed in their location. One possibility is the concept of fully flexible modules that can be placed anywhere within the free reconfigurable space. This is shown in figure 2.14. As an alternative, designs can have pre-defined reconfigurable regions similar to a non-relocatable design shown in figure 2.13. However, the modules are relocatable between these regions. This is illustrated in figure 2.15.

Fully flexible placement is often treated as a combined placement and scheduling problem where tasks represented by hardware modules have to be placed within the current free space of the device. For static schedules the placement can be solved offline but for

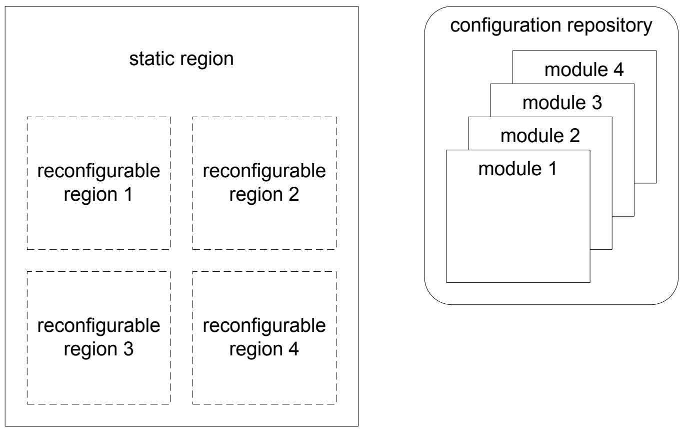


Figure 2.15: A reconfigurable design with regions allowing relocation. Modules can be assigned to any region.

dynamic schedules placement has to be done at run time. In all cases, device fragmentation is an important issue. Kastner *et. al.* present an offline 3D floorplanning technique with time as the third dimension [11]. Reconfigurable modules are treated as cuboids where the base rectangle represents the module's dimensions and the height the module's run time. It is proposed to solve the 3D placement problem with a greedy algorithm or simulated annealing. A run time version of the problem is also considered, and several algorithms are compared with regard to quality of placement and speed. Diessel *et. al.* study a two-dimensional dynamic scheduling and placement problem [28]. The analysis is based on rearrangement of modules in order to free up space. Three different compaction methods are compared on an abstract tiled architecture. It is shown that rearrangement can be a viable method to reduce queuing delays. However, the model assumes that tasks can be stopped and restarted at any time without additional overhead. Brebner *et. al.* consider the problem of flexible module placement in a one-dimensional arrangement [17]. Defragmentation is proposed to reclaim contiguous free space in the device. A concept for communication along the device is also introduced. Kalte *et. al.* create a reconfigurable system with one-dimensional module relocation [49]. This work includes a mechanism that relocates a module by performing several address translations in the bitstream at run time. Steiger *et. al.* study several heuristics for one-dimensional and two-dimensional placement and scheduling problems [99]. Tasks cannot be pre-empted because the associated overhead is considered to be unacceptable. Defragmentation is also considered impractical because of the time overhead to unload and reload modules into the FPGA. The heuristics are evaluated in a reconfigurable system consisting of an FPGA managed by a host CPU. The results show that the scheduling time is small compared to the configuration

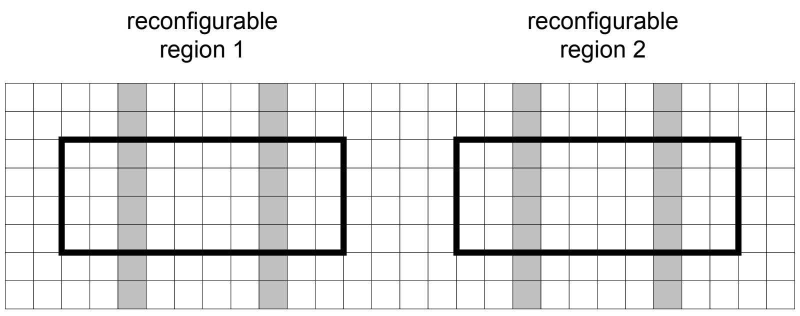


Figure 2.16: Relocation in a heterogeneous device. Reconfigurable regions have an identical footprint.

time of a module. Lu *et. al.* introduce a model where placement is not fully flexible [64]. Instead, the FPGA is partitioned in coarse tiles and modules can be placed covering one or several tiles. This reduces the complexity of the placement problem and hence, also reduces scheduling times and rejections caused by fragmentation. All these methods assume a fully homogeneous FPGA architecture without dedicated blocks. This is a precondition for fully flexible placement. Fully flexible placement always suffers from fragmentation issues. Another important aspect is the inclusion of a communication infrastructure which is often not considered.

Current commercial FPGAs are not homogeneous and contain at least one type of dedicated block. The most common type of dedicated resources are embedded RAMs and these are usually spread throughout the FPGA fabric. Approaches with fully flexible placement assuming a homogeneous FPGA fabric are therefore not applicable for modern FPGAs. Relocation and flexible placement is also possible in heterogeneous devices but the placement options have to be restricted. A simple technique to enable relocation in heterogeneous devices is to identify regions that have an identical footprint, i.e. they provide an identical pattern of resources as shown in figure 2.16.

Horta *et. al.* develop the PARBIT tool that can relocate partial bitstreams within a device or across device types [44]. The tool is used offline and cannot be used for run-time relocation. The self-reconfigurable platform presented by Blodget *et. al.* also provides routines to relocate a module [13]. Unlike in Horta's tool, relocation is here performed at run-time. Both tools need identical regions as shown in figure 2.16 in order to perform a relocation, but do not consider the problem of how to place a module at run time. Koester *et. al.* present a module placement technique for heterogeneous architectures [55]. The method finds suitable target regions for a module by identifying identical footprints in the fabric. Two algorithms are presented that aim to optimise the placement based on the probability that tiles will be used by competing modules. This work includes a mechanism for relocating modules by modifying the address fields in the bitstream similar to the one in [49]. Sedcole's reconfigurable video processing platform is an example of a practical application that uses relocatable modules on a heterogeneous device [91]. Another application is the reconfigurable platform by Hagemeyer *et. al.* [41]. It provides a flexible bus system that allows modules to be relocated and placed anywhere along the bus. In both cases, the system allows flexible 1D placement provided that a suitable identical target region can be found. However, it is not explained how the placement problem is solved.

In a slightly different approach, Koester *et. al.* present a placement technique where the device is pre-partitioned into tiles [54]. Modules can then cover one or several tiles. The concept is similar to the approach by Lu *et. al.* [64] but targets heterogeneous devices. Such an approach effectively coarsens the granularity and simplifies the placement problem.

Another possibility is to pre-partition the device into fixed reconfigurable regions as illustrated in 2.15. This type of approach has several advantages. It simplifies the placement problem so that modules only have to be allocated to a free region. Furthermore there are no issues with fragmentation of the device and finally, it is easy to include a communication infrastructure if the possible locations of modules are known at design time. Sedcole *et. al.* develop such a system with two reconfigurable regions on Virtex-II Pro and Virtex-4 [89]. The regions have an identical footprint and allow relocation. The system presented by Koh *et. al.* has the entire device partitioned into fixed-size reconfigurable regions [56]. The motivation is the simplification of scheduling, placement and the inclusion of communication links. However, relocatability is very limited in this system because of the heterogeneity of the targeted Virtex-4 device: most regions do not have an identical footprint.

The limitations on relocatability caused by device heterogeneity forces many current reconfigurable systems not to use relocation. A reconfigurable system for automotive control functions presented by Becker *et. al.* does not use relocation [12]. The system contains four large reconfigurable regions, but they cannot be floorplanned such that they provide identical footprints. A reconfigurable system for high-performance computing reported by Araby *et. al.* encounters similar problems and features two large regions without relocation [32].

2.9 Quality metrics for reconfigurable designs

The quality of a design can generally be evaluated in terms of area, performance and power. With reconfiguration one can try to improve any of these parameters, or several together. As explained in section 2.2, one of the most basic motivations behind reconfiguration is the reduction in area. Area can be saved by switching between modules or by specialising a module to a certain condition. Area reduction through reconfiguration is used in a range of applications such as encryption [73, 78], image compression [106], automotive [12], software-defined radio [108], circuit switched communication [139] and signal processing [18, 36, 52]. In [18] for example, a FIR filter is specialised for a set of coefficients resulting in a 40% area reduction. However, making a design reconfigurable can also incur a certain area overhead. This can be caused by a reconfiguration controller that has to be added, or other necessary infrastructure. If an embedded or external processor system is already present then configuration control can be added with very little overhead.

Reconfiguration can also be used to improve performance. Increased performance is usually obtained through module specialisation. This has been demonstrated for encryption [78] and signal processing [84]. For example, the reconfigurable DES core in [78] is 60% faster than the best previous implementation. However, when evaluating the performance of a reconfigurable design one has to take the time overhead of the reconfiguration process into account. FPGA reconfiguration is much slower than a context switch in a processor and can take hundreds of microseconds to hundreds of milliseconds depending on the configuration speed and the area being reconfigured [66]. This can have significant impact on performance if the system is frequently reconfigured. It has been proposed to use configuration pre-fetching or caching in order to reduce the configuration overhead [43, 63], but this comes at the cost of increased area usage. Another possibility to significantly reduce the configuration time is to use a multi-context FPGA. Such a device can hold several configurations at once, and it is possible to switch between configurations quickly [103]. However, the overall area overhead of such an architecture has barred multi-context FPGAs from commercial success.

Reconfigurable hardware is also often used to accelerate software functions. This can provide speed-ups of several orders of magnitude [92]. There is extensive work on reconfigurable accelerators attached to embedded processors [106], embedded processors with reconfigurable custom instructions [29, 92], reconfigurable co-processors [109] and reconfigurable accelerators for high-performance computing [32]. In order to achieve good overall performance, hardware speed-ups have to be carefully balanced against reconfiguration overheads [32]. Seng *et. al.* introduce the configuration ratio r as simple metric to judge the efficiency of a reconfigurable accelerator [92]:

$$r = \frac{t_{sw}}{t_{hw} + t_r} \tag{2.1}$$

In the above equation, t_{sw} denotes the time that a software kernel requires to compute a certain function, t_{hs} is the time to compute the same function in hardware and t_r is the reconfiguration time. For configuration rates $r < 1$ a reconfigurable system will be faster than a pure software version.

Wirthlin *et. al.* propose to evaluate the efficiency of a reconfigurable system by functional density [115]:

$$d = \frac{1}{A \cdot t} \tag{2.2}$$

Functional density d is the inverse product of the total area and time required for a computation. It is shown that for several neural network and template matching applications, functional density can be improved by up to four times in the best case, but the results vary with configuration time and workload size between configurations. Functional density is also used in [19] to evaluate a specialised FIR filter. It is shown that, depending on dataset size, run-time generation of a reconfigurable circuit can deliver higher functional density than a non-reconfigurable design.

Finally, reconfiguration can also be used to improve power consumption. Total power is a combination of static power and dynamic power. Static power is mainly influenced by the area of the chip and temperature, while dynamic power is related to switching activity. A reconfigurable design with module switching can reduce static power simply by reducing area. However, dynamic power is expected to be similar to a non-reconfigurable design because that same activity takes place in a smaller area of time-shared FPGA fabric. Module specialisation can reduce static and dynamic power; the design will be smaller but also more efficient. Power improvements in reconfigurable designs have been demonstrated for signal processing [84] and error correction [76, 100]. For example, the Viterbi decoder in [100] can choose a low-power version for conditions with good signal-to-noise ratios and low bit-error rates. When the conditions deteriorate, the system can be reconfigured with a more powerful but less power-efficient decoder. While power during processing can be improved, there is also a power overhead associated with the reconfigurability of the design.

This overhead can be caused by the configuration process itself, and by the presence of a configuration controller. The work in [76, 84, 100] simply measures the optimised design but not the configuration power. Hübner *et. al.* measure the power during the reconfiguration process in a Virtex-E FPGA. In this measurement, power ramps up during initial device configuration but remains fairly constant during run-time reconfiguration. The work is limited to measuring configuration power and does not relate the results to the power in a reconfigurable application.

2.10 Low-power FPGAs

Low power consumption and energy efficiency is a crucial aspect in battery-based mobile applications to extend the battery life of the device. Mobile devices have stringent power constraints and these can bar FPGAs from being used in such a device. Power and energy is also important for applications that are connected to the grid. Lower power leads to less heat, reduces the complexity of cooling systems and power supplies and consequently reduces energy and system costs.

Reconfiguration is one technique to reduce power consumption but is important that devices are developed for low power. This is addressed through research in FPGA architecture, circuit design and comparative evaluation. Shang *et. al.* measure the dynamic power consumption of a Xilinx Virtex-II FPGA using a Xilinx internal benchmark that represents a large industrial circuit [93]. Using this internal benchmark and input stimuli, the switching activity of the design is calculated. This allows to estimate power based on the calculated switching activity and the effective capacitance of each resource on the FPGA. This involves using low-level models of the FPGA, but such models are usually not publicly available. Kuon *et. al.* assess the gap between FPGAs and ASICs [57]. This work includes a comparison of dynamic and static power consumption between the two technologies. The evaluation is based on power estimation tools provided by the FPGA vendor, and uses either testbenches or estimates of net activity. The results show that FPGAs can consume between 7 and 14 time more dynamic power than ASICs. The comparison of static power is less conclusive but is estimated to be within one order of magnitude. FPGA clock networks can be responsible for a significant amount of power consumption. Lamoureux *et. al.* examine clock-aware placement techniques, and investigate the trade-off between the power consumption of the clock network and the impact of the constraints imposed by the design automation tools [60].

George *et. al.* create a low-power FPGA through architecture and low-level circuit design [35]. The optimisation includes a reduction in connectivity to neighbouring tiles. A comparison of this FPGA with Xilinx and Altera devices based on simulation with Synopsis Powermill shows improvements of more than an order of magnitude. Gayasen *et. al.* propose an FPGA architecture with two supply voltages where the lower voltage is used for all non-critical path components [34]. The efficiency of their architecture is evaluated with MCNC benchmarks which provide a range of simple circuits and state machines. Tuan *et. al.* present an experimental low-power FPGA with several power optimisations such as voltage scaling, leakage reduction of configuration memory cells, and power gating of tiles with preservation of state and configuration [105]. An implementation on a 90 nm process technology demonstrates a reduction of 46% in active power and 99% in standby power. These values are not specific to a particular FPGA configuration.

Most current commercial FPGAs have limited low-power capabilities. Traditionally, FPGA power optimisations target operational power in order to reduce power and heat in systems that operate continuously. Xilinx introduced triple-oxide technology, which uses less leaky transistors with thicker gate oxide for configuration memory cells and interconnect pass gates [117]. Altera developed a programmable transistor back-biasing feature that allows selecting high-performance logic with high power consumption only for timing-critical components in the design [5]. However, these improvements alone are insufficient to enable the use of FPGAs in mobile, battery-based applications. These applications do not only have power constraints for active processing. They are also characterised by long periods of inactivity and require dedicated low-power modes that reduce the total power by several orders of magnitude. With most current devices, the only methods of saving power during inactivity are to employ clock gating or to turn off the entire device. The first method has limited potential whereas in the latter case, state and configuration are lost.

Recently, some devices with additional low-power capabilities were introduced. Xilinx Spartan-3A and Spartan-3AN FPGAs support a suspend mode in which auxiliary clock circuitry can be stopped and powered down [125]. Lattice Mach XO devices provide a sleep mode [61] that can reduce the standby power by a factor of 100. However, the application state is lost when using this sleep mode. Another example are Actel IGLOO devices which feature a sleep state that does retain the application state [1]. Silicon Blue provides iCE65 FPGAs [96] that are optimised for low static power. Table 2.2 shows a comparison of these devices and their low-power features. These modes vary widely in their behaviour and effectiveness, and it is difficult to perform a simple comparison.

device manufacturer	Xilinx	Lattice	Actel	Silicon Blue
device name	Spartan-3A, Spartan-3AN	Mach XO	IGLOO	iCE65
device size [LUTs]	1.6k - 53k	256 - 2.3k	192 - 36k (*)	1.8k - 17k
low-power feature	suspend mode	sleep mode	flash freeze	iCEgate
wake-up time [μs]	100-500	1000	1	instant
retain state	yes	no	yes	yes

(*) device does not provide LUT/FF pairs like all other devices, reconfigurable tiles can be used as LUT or FF.

Table 2.2: Comparison of devices and low-power features.

2.11 Summary

FPGAs contain an array of programmable logic and routing elements and most commercial devices use SRAM technology to store the device configuration. This can be used to rapidly change the configuration even when the device is in use. There are many ways how the reconfigurability of FPGAs can be exploited. Scenarios include fast design and debug, in-field upgrades, self-repair and dynamic switching. Most scenarios involve some form of reconfiguration during run time, but the latter scenario is the one that is commonly termed “run-time reconfiguration”.

Run-time reconfiguration is a powerful technique that allows a design or part of a design to be modified while it is operational. Hardware modules can be swapped in and out, and this concept is often compared to a context switch in a processor. However, a context switch is a mechanism that is readily available in the software world, and making use of FPGA run-time reconfiguration is much more complex. A reconfigurable design takes more time to design and compile, and requires manual intervention. Reconfiguration can also incur overheads that could limit or possibly even outweigh the improvements. This makes careful planning and early design space exploration necessary. However, if a system is designed well, run-time reconfiguration can bring compelling improvements in performance, area and power efficiency.

Relocatability is another important aspect because it increases the flexibility of reconfigurable modules and reduces compile time and storage requirements. Relocatability is currently not supported in the Xilinx partial reconfiguration design flow but there is a range of research that proposes several relocation techniques. Fully flexible placement is not applicable on heterogeneous devices. Relocation on heterogeneous devices is currently limited to identical regions. This can be a severe restriction and many practical reconfigurable systems do not use relocation.

Finally, we find that the power consumption in FPGAs needs to be reduced to make them more competitive for mobile applications. Several evaluations and comparisons measure the power in FPGAs and it is shown that current devices are significantly less efficient than ASICs. Power can be addressed through architectural improvements and advances in process technology but reductions in operational power alone are not sufficient. Mobile applications also need dedicated low-power modes for periods of inactivity. We find that devices need improvements in providing advanced low-power modes. We also need new methods to evaluate the power efficiency of devices for mobile scenarios.

Chapter 3

Software-defined radio

This chapter introduces software-defined radio as the motivating application of this thesis. In section 3.1, we provide a general overview of software-defined radio, and explain the challenges one faces when developing a digital radio. Modern radios demand flexible and efficient target architectures with high design productivity and short development times. Section 3.2 shows a comparison of three common architectures and points out several trade-offs. FPGAs are an appealing target architecture because they combine performance and flexibility, but power consumption and chip costs have to be reduced. In section 3.3, we analyse how reconfiguration in FPGAs can be used as an additional advantage. We introduce four reconfiguration scenarios and explain what benefits they provide for a radio design. In the following section 3.4, we compare several current radio standards and analyse how our reconfiguration scenarios can be used to implement a flexible multi-mode radio system. Section 3.5 concludes this chapter and presents challenges that need to be addressed in order to give FPGAs a broader appeal as a solution for modern commercial radio systems.

3.1 Overview of software-defined radio

Software-defined radio is an emerging technology that is loosely defined as processing digital radio signals by means of software techniques. Today's mobile applications feature an abundance of radio standards with ever increasing bandwidth, and standards keep evolving quickly. This leads to a demand in both increased design productivity and flexibility. Software techniques are seen as a solution to quickly design flexible mobile applications. There are many understandings of what is considered a "software technique". It can mean employing software-programmable processors or DSPs in a digital radio. The implementation of a radio standard then becomes a software task rather than hardware design. Others

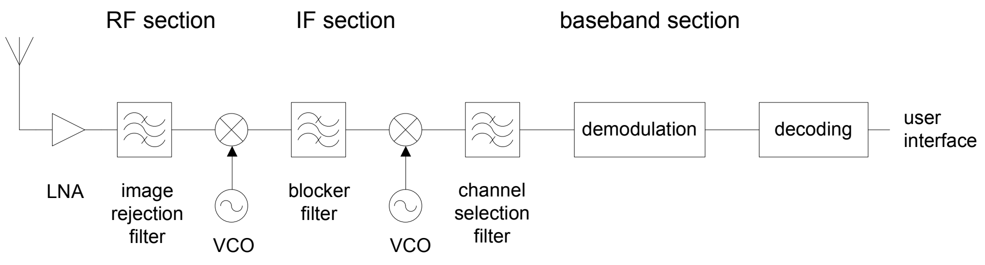


Figure 3.1: Structure of a classic receiver architecture.

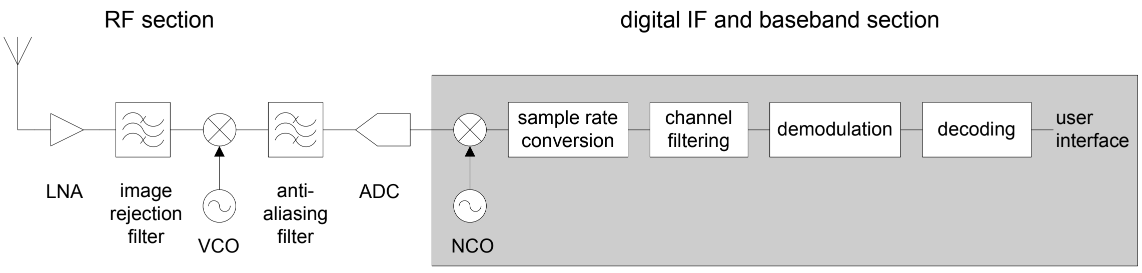


Figure 3.2: Receiver architecture of a software-defined radio.

consider reconfigurable hardware architectures, which offer hardware-like performance and efficiency with software-like flexibility.

There are also multiple levels where the term software-defined radio can be applied. The most intuitive interpretation is the physical implementation of the radio itself but software techniques can also be applied to higher levels such as the network operator level or the service provider level. In the following we focus on the radio implementation.

In modern radio systems, digital technology is continuously moving closer to the antenna. Early radio systems had digital components performing basic control functions such as medium access control or channel select. Digital components then moved into the user interface and baseband processing. Today, digital technology can also include processing at intermediate frequency (IF) range. Ultimately, digital technology could be moved up directly to the antenna and all processing including the radio frequency (RF) processing is performed by digital technology. Currently this is deemed not to be efficient due to the very high sampling and processing rates involved. For example, today's 3G communication standards operate at frequencies of up to 2.5 GHz and according to the Shannon sampling theorem this would require sampling rates of 5 GHz/s if signals are digitised at radio frequency. However, the performance and productivity of digital technology continues to increase exponentially whereas the productivity of analogue design only increases slowly. Therefore, a fully digital radio could become realistic in the future.

Figure 3.1 shows the structure of a receiver in a standard mobile phone. The incoming signal is amplified and filtered before it is mixed down to an intermediate frequency by a superheterodyne receiver. Frequencies outside the wanted band are blocked and the signal is then converted down to a complex baseband signal. After selecting the desired channel the signal is demodulated and decoded. Figure 3.2 shows a receiver where the signal is digitised directly after being mixed to an intermediate frequency.

However, a digital radio is not synonymous with a software-defined radio. In commercial mobile phones, digital baseband processing is usually performed by inflexible but power-efficient ASICs. In order to support multiple standards, multiple pieces of IP have to be assembled into a complex system that subsequently needs to be tested and manufactured. In contrast, a software-defined radio can support multiple standards by employing a software-programmable architecture that is flexible in its processing capabilities. This concept was first proposed by Mitola [75] and it is based on the observation that software embodiments of traditional radio hardware can significantly improve efficiency and design productivity. The main motivations are:

1. Increased interoperability of a communication device with multiple communication standards.

2. Increased spectral or energy efficiency of a communication device.

3. Reduced size and complexity of the device.

4. Ease of design and increased productivity by moving from custom hardware to a programmable device.

The first aspect is very often cited as the key motivation for software-defined radio [107]. Current commercial wireless systems are based on a wide range of frequency bands and modes. A device with multiband, multimode capabilities can enable unlimited worldwide roaming by supporting all common standards such as GSM, CDMA, PDC or PHS. Another scenario is the support of a wide range of 2G and 3G standards such as GSM, EDGE, DECT, UMTS and Bluetooth. Supporting a wide range of standards in a conventional digital radio usually involves assembling IP cores in a chip or implementing them as separate radios. This becomes impractical as the number of standards increases. A programmable software radio architecture on the other hand can support various standards with significantly fewer components. It also allows to update the radio if a new revision of a communication standard becomes available.

Software-defined radios can also have the benefit of higher efficiency. Such a system can increase the spectral efficiency by managing the spectrum in wideband technologies.

It is also possible to increase the energy efficiency by adapting to the signal-to-noise ratio of the channel or to the required bandwidth.

Especially mobile handheld devices have strict area constraints. Supporting multiple standards often requires implementing separate radios. Placing them on a small circuit board can be challenging. Using a software-defined radio can reduce the number of chips on the circuit board and hence, lower the system complexity.

Finally, software-defined radios can also help to simplify and speed up the design of a communication device. The design of digital as well as analogue custom circuits is becoming increasingly expensive, risky and time-consuming. Relying on pre-fabricated programmable components can reduce risk, cost and time-to-market.

However, the term software-defined radio is a general concept and not a strict definition of a particular platform or architecture. In practice, there are several possibilities to implement flexible, programmable radios.

3.2 Architectures for flexible radios

A simple and straightforward approach to implement a software-defined radio is to use a general-purpose processor. For example, *GNU radio* is a software development toolkit that provides the signal processing runtime and processing blocks to implement communication standards on a PC [37]. It requires an FPGA-based front end, but baseband processing functions are implemented in C++ and executed on the processor. This lowers the entry barrier and enables academics and hobbyists to work with modern communication standards, but the implementation itself is very inefficient in that it requires a modern high-performance CPU to perform the same function as a dedicated baseband processor. DSPs are more efficient for signal processing, and they are widely used for radio prototyping systems. However, programmable instruction set architectures such as DSPs or general-purpose processors (GPPs) are generally not very efficient and they usually violate the constraints of mobile devices or base stations.

Another approach is to use programmable hardware such as FPGAs. These offer performance and efficiency closer to dedicated hardware while still being flexible. Table 3.1 compares the advantages and disadvantages of GPPs, DSPs, FPGAs and ASICs.

GPPs and DSPs are limited in their performance and they are usually not a viable solution as a stand-alone system. FPGAs can be very fast and offer similar performance to ASICs. Modern FPGAs also provide dedicated DSP cores which help to accelerate arithmetic functions. In terms of power efficiency, the four architectures can also be ranged

	performance	power	flexibility	design time / NRE	chip cost
GPP	- -	- -	++	++	- -
DSP	-	-	+	++	-
FPGA	+	0	++	+	0
ASIC	++	++	- -	- -	++

Table 3.1: Comparison of architectures for digital radios.

from low for GPPs and DSPs to high for ASICs. FPGAs represent an intermediate solution but they are not as competitive in power as they are in performance. The functionality of ASICs is specified at the time of manufacturing and they do not provide any flexibility other than what is built in at design time. A DSP on the other hand is flexible as it can be re-programmed, but the functional units inside the DSP cannot be changed and the programmability is typically limited to signal processing functions. GPPs provide better programmability than DSPs but they also have fixed functional units. An FPGA is very flexible in that it can be re-programmed but it also allows to customise the functional units in the data-path. ASIC design is expensive, risky and time-consuming. It requires extensive testing before tape-out after which no changes can be made. Costs are continuing to go up as process technology advances to smaller geometries. Programming a GPP or DSP on the other hand is very simple. It usually involves writing C or assembly code that can easily be debugged. Designing for an FPGA is more complicated. Writing HDL code is less productive than high-level languages and compilation and debugging takes more time. However, it is still significantly less complex than designing an ASIC. Special toolkits and IP core libraries for signal processing can also be used to simplify the design of such applications. The costs for the individual chip are very low for ASICs while DSPs are too expensive for most consumer applications. Using a GPP to implement radio functions is also not cost effective. Currently, FPGAs lack competitive pricing for consumer devices as they are usually not sold in very high volumes. If the technological challenges are solved, then higher production volumes could also lead to lower FPGA prices. In order to make FPGAs a more attractive solution for software-defined radios, performance and power efficiency needs to be improved, with power being the bigger challenge.

3.3 Opportunities for reconfiguration

In section 2.2, we outline four different approaches of how reconfigurability in FPGAs can be exploited: *design and debug*, *in-field upgrade*, *self-repair* and *dynamic switching*. We can identify several scenarios of how these reconfiguration types can be used to improve an FPGA-based radio design:

1. **Pre-deployment**. The programmability of the target architecture is used to make changes late in the design process before the device is deployed. Generic programmable devices also simplify the design by reducing the number of components needed. This is an application of *design and debug*.

2. **In-field upgrade**. The device is updated to support a new standard of feature that was not included at deployment. In this case, the device will be rebooted and downtime during reconfiguration does not matter. This is an application of *in-field upgrade*.

3. **Reconfiguration per call or session**. The device is reconfigured at the beginning of a call or data transfer session e.g. to select the most efficient or cheapest service available at the moment. In this case, the reconfiguration downtime has to be short, preferably below the level of human perception but no data has to be buffered. The reconfiguration overhead is usually insignificant and may be hidden by the application. This is an application of *dynamic switching*.

4. **Reconfiguration during a call or session**. The device is reconfigured during a call or data session e.g. to hand over from one service to another. Reconfiguration downtime time has to be very short and ‘1streaming data has to be buffered during reconfiguration. The influence of the reconfiguration overhead on performance and energy efficiency is significant and needs to be considered. This is also an application of *dynamic switching*.

Table 3.2: Scenarios showing opportunities for reconfiguration in software-defined radio.

Although reconfigurability certainly has promising aspects in all four scenarios, we are mainly interested in the latter two. We do not consider *self-repair* a key technique for software-defined radio although it could also potentially be used. In the following we analyse how these reconfiguration scenarios may be used to support multiple communication standards.

3.4 Radio communication standards

Table 3.3 gives an overview of some of the most important communication standards that are currently being employed. These standards vary widely in their technical details and satisfy different needs. Highly mobile standards such as GSM, EDGE and UMTS are used mainly in mobile phones and offer low to medium data rates. On the other hand, IEEE 802.11g offers high data rates with limited mobility and high power consumption. Other standards with limited mobility such as DECT and Bluetooth offer low to medium data rates with very low power consumption. This list could be extended with broadcasting standards such as DAB and DVB-T or standards for international roaming such as PHS, PDC and CDMA2000.

A radio device that can support all these standards has to very flexible. In current mobile devices, multiple standards are realised by using separate radios, and by assembling the IP on ASICs and switching between blocks. This however becomes impractical as the number of standards increases and ASIC manufacturing becomes increasingly expensive. A radio based on reconfigurable hardware could easily support a range of standards simple by changing the configuration.

In the following we focus on the functionalities that are typically found in radio, and analyse how they might be reconfigured in a receiver as illustrated in figure 3.2. All considerations apply to the transmitter and receiver in a similar way since the receiver reverses all the steps that are performed by the transmitter. If we focus on the receiver side we can observe the following tasks that have to be performed:

- digitisation of a real signal at IF level
- down-conversion into a complex (I/Q) baseband signal
- fractional sample rate conversion
- flexible channelisation filter with integer decimation
- optional FFT for OFDM standards
- optional de-spreading for spread-spectrum standards (UMTS)
- channel estimation
- equalisation
- optional rake-receiver

	GSM	EDGE	DECT	UMTS TDD	UMTS FDD	Bluetooth	IEEE 802.11g
Frequency band	900, 1800, 1900 MHz	900, 1800, 1900 MHz	1.9 GHz	1.9, 2 GHz	1.9, 2.1 GHZ	2.4 GHz	2.4 GHz
Channel bandwidth	200 kHz	200 kHz	1728 kHz	1.6 or 5 MHz	5 MHz	1 MHz	25 MHz
Channel access	FDMA/TDMA	FDMA/TDMA	FDMA/TDMA	TD-CDMA	DS-CDMA	TDMA	FDMA/TDMA
Duplex	FDD	FDD	TDD	TDD	FDD	TDD	TDD
Modulation	GMSK	GMSK, 8-PSK	GMSK	QPSK	QPSK	GFSK	OFDM
Pulse shaping	Gauss (BT = 0.3)	Linearised Gaussian	Gauss (BT = 0.5)	Root-raised cosine ($\beta = 0.22$)	Root-raised cosine ($\beta = 0.22$)	Gauss (BT = 0.5)	various
Error correction	CRC, convolutional	CRC, convolutional	CRC	CRC, convolutional, turbo	CRC, convolutional, turbo	-	convolutional
Bit or chip rate	270.83 kbit/s	270.83 kbit/s	1152 kbit/s	1.28 or 3.84 Mchip/s	3.84 Mchip/s	1 Mbit/s	54 Mbit/s
Net bit rate	13 kbit/s	up to 473.6 kbit/s	32 kbits/s	up to 2 Mbit/s	up to 2 Mbit/s	1Mbit/s	25Mbit/s
Power consumption	medium	medium	low	high	high	low	very high
Service	voice	voice, data	voice, data	voice, data	voice, data	voice, data	data

Table 3.3: Comparison of different mobile communication standards.

- demodulation of GMSK, GFSK and various PSK and QAM modulation schemes
- CRC, convolutional (Viterbi) and turbo error correction schemes
- optional decryption

The first part of the receiver chain consists of an analogue superheterodyne receiver that mixes the input signal down to an intermediate frequency (IF). At this stage, the signal can be digitised. Typical sampling frequencies are in the range of tens of megahertz. In order to support a wide dynamic range, samples usually have a resolution of 14 - 16 bit. This corresponds to a dynamic range of 84 - 96 dB.

As the first digital processing step, the real signal is down-converted into a complex (I/Q) baseband signal. This is done by multiplying the time-discrete IF signal x_{IF} with a complex rotating phasor of the frequency f. The frequency f is selected so that the channel of interest is moved into the baseband. The inphase (I) and quadrature (Q) components are generated with the following equations:

$$I(n) = x_{IF}(n) \cdot \cos(2\pi f n t) \tag{3.1}$$

$$Q(n) = -x_{IF}(n) \cdot \sin(2\pi f n t) \tag{3.2}$$

One way of creating the sine and cosine function in digital logic is storing all their values in a ROM table. The disadvantage is that this table might be very large if the period of the mixing frequency f is long. The frequency can be changed by reconfiguring the ROM table or using a RAM instead. The Cordic algorithm represents an alternative method of generating sine and cosine functions [110]. Since Cordic is an iterative algorithm, a pipelined version would most likely be required to provide sufficient throughput. It is also possible to use an optimised, reconfigurable Cordic processor to achieve higher performance and less area requirements [50]. The down-converter must be able to adapt to a different frequency in less than 1 ms in order to support a change of channel between transmission frames. This corresponds to reconfiguration scenario 4 in table 3.2.

The next part of the receiver converts the sample-rate from the A/D converter to an integer factor of the baseband processing rate. In a simple radio, the A/D conversion can be run synchronised with the baseband processor and sample-rates can be converted by decimation. In a flexible radio this is not always possible, since different standards require various baseband processing rates. In this case, a fractional sample-rate conversion can be done by over-sampling (zero-padding), low pass filtering and subsequent under-sampling (decimation) [25]. After sample-rate conversion, the sample rate is still very high but can now be divided by an integer factor to the baseband rate. This allows designers to employ simple filters in the following channelisation filter.

Channelisation is done by filtering all other channels and non-interesting signals. This can be done by a low pass filter since the channel of interest is located in the baseband. IIR filters are usually avoided due to instability issues and their non-linear phase. FIR filters are used instead, but these can require a high order to achieve sufficient attenuation in the stop band. Filtering a narrowband-signal at high sample rate can require a large filter with many coefficients. The size of a filter can be reduced by using a cascaded integrator-comb filter (CIC) with following FIR filter stages and sample-rate reduction between stages [25].

Filter architectures are usually specific to one radio standard and cannot be easily tuned to substantially different requirements. It is possible to identify similarities between different filter architectures and develop a parametrised circuit that can switch between two standards [113]. However, it becomes difficult to parametrise a design for a large number of varying standards. It can be more practical to develop optimised filters for each standard and reconfigure on demand. Both, sample-rate conversion and channelisation filtering are specific to a communication standard and only need to be reconfigured if the standard is switched, e.g. between a phone call and a data session. This corresponds to reconfiguration scenario 3 in table 3.2.

OFDM and spread-spectrum CDMA standards require extra functional blocks before further baseband processing. In case of spread-spectrum standards, a signal with low bit-rate is multiplied with high bit-rate scramble codes called *chips*. This is done before transmission to spread the signal energy over a wide spectrum. In the receiver this has to be reversed and signal is de-spread by multiplying it with the same scramble codes. In CDMA standards a rake receiver can also be used to counter the effects of multipath signal propagation. The receiver consists of several *fingers* which can be tuned to the delay of the different signal paths. These can then be added back together to recover the maximum signal energy. OFDM standards use orthogonal carriers that have to be separated by an FFT. The aforementioned functions are specific to some standards and are not required otherwise. The level of reconfigurability is according to scenario 3 in table 3.2.

Channel estimation is a function that determines the channel impulse response based on a known number of bits in the received data. The channel impulse response is used in the following demodulator to remove inter-symbol interference that is caused by multipath signal propagation. Channel estimators are usually based on the Least Squares method. The details of the implementation vary between radio standards. The demodulation methods vary in keying techniques and pulse shaping. GSM for example uses Gaussian minimum shift keying (GMSK) which is based on frequency-shift keying and Gaussian pulse-shaping. UMTS uses quadrature phase-shift keying (QPSK) with root-raised cosine pulses. Again, it is possible to parametrise a demodulator for two standards [113], but a range of methods as indicated in table 3.3 would most likely require the reconfiguration of the modulator/demodulator. This can be categorised as scenario 3.

The next step in the receiver chain is error correction. Different methods of cyclic redundancy checks (CRC), convolutional codes and turbo codes are employed. GSM for example uses CRC for the most important bits and a convolutional decoder for less important bits. UMTS also uses turbo codes for error correction. Again, the reconfigurability can be categorised as scenario 3 if the error correction is changed between standards. However, it is also possible to change an error correction to adapt to different channel conditions during a call or data transmission. This case corresponds to scenario 4.

Finally there could be an encryption scheme at the end of the receiver chain. This is completely independent from any particular radio standard and could be reconfigured during or between transmissions.

We conclude that radio standards exhibit many similarities in employed components such as filtering, demodulation and decoding. The implementation details of these components, however, vary widely. Some components such as de-spreading, FFT or decryption are specific to certain standards and not always present. For reconfiguration of filter coefficients or Cordic parameters, module specialisation may be used. On the other hand, module multiplexing can be used to swap entire blocks such as a filter or a demodulator. The reconfigurability also varies between reconfiguration-per-call and reconfiguration-during-call. The latter scenario has hard real-time constraints and usually requires buffering of data during reconfiguration. Even if higher level protocols can tolerate lost data, it is usually preferable to not lose data from the perspective of quality-of-service.

3.5 Summary

Fast evolving standards in today's radio systems encounter problems with cost and productivity when relying on traditional fixed hardware. The need to efficiently and dynamically manage bandwidth and quality of service also calls for more flexible platforms.

Reconfigurable architectures such as FPGAs are a promising option for developing software-defined radio systems: they deliver performance through parallelism while being reconfigurable and therefore very flexible. They are superior to general-purpose processors and DSPs in performance and power efficiency. FPGAs are already used in base stations and radio prototyping systems, but so far they have not seen widespread use in mobile or other consumer devices. The standard for power and performance is set by ASICs, and FPGAs lag behind in both aspects. Especially the gap in power efficiency is crucial and needs to be addressed.

Reconfigurability is an inherent feature of SRAM-based FPGAs and it can be exploited to improve designs in various aspects. We identify four scenarios of how reconfigurability can be used in a software-defined radio. When reconfiguring during the run time of the application, the frequency of reconfiguration plays an important role. If reconfiguration is used frequently during a call or session (scenario 4) then reconfiguration overheads need to be considered and balanced carefully.

When designing a reconfigurable system, we want to improve our application as much as possible while keeping the overheads low. With reconfiguration, improvements can be made in performance, area and power or energy efficiency. Possible overheads are reconfiguration time, reconfiguration energy, configuration storage and extra design efforts.

In the following chapter 4, we present techniques how performance, area and energy efficiency can be improved, taking into account configuration overheads. Furthermore we develop a method to reduce design time and external storage for configurations by increasing the relocatability of partial bitstreams. This is covered in chapter 5.

To bridge the gap in power efficiency, though, reconfiguration alone is not sufficient. Further work on low-power architectures with dedicated low power modes is necessary. We therefore develop a technique to evaluate the power efficiency of reconfigurable architectures in different modes of activity. This is presented in chapter 6.

Chapter 4

Parametric optimisation

Developing a reconfigurable application can be a time-consuming process. As explained in section 2.6, the design flow requires considerable effort and careful planning. This could be alleviated by developing tools that automate or simplify some of the design steps, but the key issue remains the ad-hoc nature of the design process. Since reconfiguration can improve some design parameters while introducing overhead at the same time, it is often unclear how the final implementation will perform.

Efficiency and productivity can be improved by using an analytical model that captures the behaviour of the targeted system. Such a model does not necessarily have to be fully accurate; simplicity and usability are often more important. For example, Ohm's law does not account for the temperature dependency of resistance. Nevertheless, it is a very useful abstraction for many electronic circuits. In the following we present a model for reconfigurable implementations of iterative algorithms. These types of algorithms are often used in signal processing applications. The model allows us to optimise the design for performance and energy efficiency by varying the degree of parallelism, and it provides insight into the trade-offs involved.

In section 4.1 we introduce the parameters used in our model. We consider application parameters and target device parameters, as well as implementation parameters that result from a particular mapping of the application onto the target device. In section 4.2, 4.3 and 4.4 we show how our parameters relate to configuration storage size, reconfiguration time and application data buffering. Reconfiguration time plays an important role in the overall efficiency of a reconfigurable design, and it is the key part in the following optimisations. In section 4.5, we apply our parametric model to improve performance. First, we analyse when a reconfigurable design becomes more efficient than a non-reconfigurable one. In the following we further improve the reconfigurable design. A key contribution is the exploration of parallelism presented in section 4.5.2. A large, parallel design delivers fast processing, but reconfigures slowly. Reducing the degree of parallelism slows down processing but speeds up reconfiguration. This trade-off can be used to optimise the overall execution time. A further key contribution is an optimisation approach that leads to the optimal design based on calculations derived from one initial implementation. This is more productive than exploring the degree of parallelism by building a wide range of designs. Section 4.5.3 presents simple optimisation steps that predict the optimal degree of parallelism, and section 4.5.4 gives guidelines on how to refine the results if the design does not scale as expected.

We can also apply our parametric optimisation approach to improve energy efficiency. This is shown in section 4.6. Again, we begin by comparing reconfigurable and non-reconfigurable designs. Next, we adapt the exploration of parallelism for energy efficiency (section 4.6.2). We find, similar to the trade-off for performance, that parallelism is beneficial for energy efficiency during processing but it increases the energy required for reconfiguration. In section 4.6.3, we devise similar optimisation steps to find the optimal degree of parallelism for energy efficiency. Our energy optimisations also depend on accurate power measurements, and section 4.7 presents several practical aspects of how such measurements can be obtained on an FPGA. We explain a simple and effective setup that can be used on most FPGA boards.

Section 4.8 presents case studies that evaluate our optimisation approach. In section 4.8.1, we perform both performance and energy optimisations on a reconfigurable FIR filter and study the effects in detail. In section 4.8.2, we build a reconfigurable Cordic processor and compare the design to its modelled behaviour. Section 4.9 presents a summary of this chapter and discusses the results.

4.1 Parameters for optimisation

Our parametric optimisation approach is based on application, device and implementation parameters. Application parameters are:

- Number of processing steps s in the algorithm
- Number of packets or items of data n that are processed between reconfigurations
- Required data throughput ϕ_{app}

Application parameters can be obtained from the specification of the application. Our approach is mainly targeted for signal processing applications where a fixed workload has to be processed between reconfigurations, and the number of processing steps in the algorithm is also fixed. The parameters s and n do therefore not change dynamically. The device is characterised by the following parameters:

- Available area A_{max}
- Data throughput of the configuration interface ϕ_{config}
- Configuration size per resource or unit of area Θ

Device parameters can be obtained from the device data sheet. Finally, we use the following implementation parameters:

- Area requirement A
- Configuration storage size Ψ
- Application data buffer size B
- Reconfiguration time t_r
- Processing time t_p for one packet or datum
- Power during processing P_p
- Computation power P_c, a component of P_p
- Power overhead P_o, a component of P_p
- Power during reconfiguration P_r

Implementation parameters require for a design to be implemented. Area and performance related parameters can usually be obtained directly from the implementation tools. Power parameters can also be estimated by the implementation tools but this may not be accurate enough. Power measurements are more accurate, but they are also more complicated. This is further explained in section 4.7. Additionally, we use the amount of parallelism p in the implementation as an implementation attribute that can be scaled. Changing p allows the designer to modify and optimise implementation parameters. Parallelism p can usually be scaled between 1 and s.

4.2 Storage requirements

All designs for FPGAs with volatile configuration memory require external storage for the full configuration bitstream. Designs using partial run-time reconfiguration also need additional storage for the pre-compiled configuration bitstreams of the reconfigurable modules. The partial bitstream size and storage requirement Ψ (in *bytes*) of a reconfigurable module is directly related to its area A. Configuration bitstreams often contain a header h. In most cases, this can be neglected because the header size is very small:

$$\Psi = A \cdot \Theta + h \approx A \cdot \Theta \tag{4.1}$$

A is the size of a reconfigurable module in FPGA tiles (e.g. CLBs) and Θ is a device specific parameter that specifies the number of bytes required to configure one tile. Our approach is mainly intended for partial run-time reconfiguration where the module area A can be scaled freely. However, it can also be used for devices that only support full device reconfiguration. In this case, the entire chip is reconfigured and the module area A is equivalent to the chip area. Design choices are then quantised to available chip sizes.

4.3 Configuration time

The time overhead of run-time reconfiguration can consist of multiple components, such as scheduling, context save and restore as well as the configuration process itself. In our case there is no scheduling overhead as modules are loaded directly as needed. There is also no context that needs to be saved or restored since signal processing components do not contain a meaningful state once a dataset has passed through. Even in systems that perform additional steps such as scheduling of context save and restore, it is usually found that reconfiguration time is the dominant component of the overall time required by the configuration controller [99]. The reconfiguration time is proportional to the size of the partial bitstream and can be calculated as follows:

$$t_r = \frac{\Psi}{\phi_{config}} \approx \frac{A \cdot \Theta}{\phi_{config}} \tag{4.2}$$

ϕ_{config} is the configuration data rate and measured in *bytes/s*. This parameter not only depends on the native speed of the configuration interface but also on the configuration controller and the data rate of the memory where the configuration data are stored.

4.4 Buffering

As outlined in table 3.2, we can distinguish between run-time reconfigurable scenarios where data do not have to be buffered during reconfiguration, and scenarios where data buffering is needed during reconfiguration. For the latter case we can calculate the buffer size B depending on reconfiguration time t_r and the application data throughput ϕ_{app}:

$$B = \phi_{app} \cdot t_r = \frac{\phi_{app}}{\phi_{config}} \cdot \Psi \qquad (4.3)$$

Table 4.1 outlines the buffer size for several receiver functions and a range of reconfiguration times. The presented reconfiguration times cover a range that is usually found in practical reconfigurable systems [66]. We can observe that the data rate is reduced through all stages of the receiver. Hence, a reconfiguration-during-call scenario becomes easier to implement towards the end of the receiver chain. Obviously, the buffer size also increases with the bandwidth of the communication standard and the duration of the reconfiguration time.

function	application data throughput Φ_{app}	buffer size B for a given reconfiguration time t_r		
		100 ms	10 ms	1 ms
down-conversion (16 bit)	800 Mbit/s	80 Mbit	8 Mbit	800 kbit
down-conversion (14 bit)	700 Mbit/s	70 Mbit	7 Mbit	700 kbit
demodulation UMTS	107.52 Mbit/s	10.75 Mbit	1.07 Mbit	107 kbit
demodulation GSM	7.58 Mbit/s	758 kbit	75.8 kbit	7.58 kbit
error correction UMTS	6 Mbit/s	600 kbit	60 kbit	6 kbit
error correction GSM	22.8 kbit/s	2.28 kbit	228 bit	22.8 bit
decryption UMTS	2 Mbits/s	200 kbit	20 kbit	2 kbit
decryption GSM	13 kbit/s	1.3 kbit	130 bit	13 bit

Table 4.1: Buffer size for various functions and reconfiguration times.

A buffer can be implemented with on-chip or off-chip resources. Most modern FPGAs provide fast, embedded RAM blocks that can be used to implement FIFO buffers. For example, Xilinx Virtex-5 FPGAs contain between 1 to 10 Mbit of RAM blocks [128]. Larger buffers have to be realised with off-chip memories.

4.5 Performance optimisation

For designs that make frequent use of reconfiguration such as scenario 4 in table 3.2 *(Reconfiguration during a call or session)*, it is important to consider the impact of the configuration downtime as it can influence or limit the performance of the system. In a module multiplexing scenario, processing times are usually equal for non-reconfigurable and reconfigurable versions of the module. The same module is either switched by real multiplexers or by reconfiguration. However, the reconfigurable system will incur a configuration overhead while a switch in a multiplexer can be assumed instantaneous. Thus, for module multiplexing, reconfiguration will save area, while the reconfiguration downtime will lead to a degradation of overall performance. It has been shown that a reconfigurable design becomes more efficient the more data items n are processed between reconfigurations [115]. The speed-up S of a reconfigurable design can be expressed as $t_{total,nrec}$, the total processing time of the non-reconfigurable system, divided by $t_{total,rec}$, the total processing time of the reconfigurable system:

$$S = \frac{t_{total,nrec}}{t_{total,rec}} = \frac{t_p \cdot n}{t_p \cdot n + t_r} = \frac{n}{n + \frac{t_r}{t_p}} \tag{4.4}$$

The reconfigurable system becomes more efficient by processing more data between configurations and by improving the ratio of reconfiguration time t_r to processing time t_p. However, the speed-up is always < 1 and only approaches 1 for very large datasets. The benefit of such a design is reduced area.

4.5.1 Improving performance

Reconfiguration can also increase performance as explained in section 2.9. This means that processing times t_p are not equal for reconfigurable and non-reconfigurable versions. If reconfigurable hardware is used as an accelerator for software functions, performance is usually improved significantly and it is easy to achieve overall speed-ups despite the configuration overhead [92]. Performance increases can also be obtained through module specialisation but speed-ups are usually less dramatic. A speed-up has to be carefully traded off against the reconfiguration penalty in order to achieve an overall performance gain. We can calculate the total processing time of a non-reconfigurable design $t_{total,nrec}$ as:

$$t_{total,nrec} = t_{p,nrec} \cdot n + t_{load} \approx t_{p,nrec} \cdot n \tag{4.5}$$

The above equation takes into account that even a non-reconfigurable design might have a time overhead t_{load} to load new parameters between processing datasets. Unlike multiplexed modules which can often be switched instantaneously, modules that allow specialisation may require a parameter reload in the unspecialised, non-reconfigurable version. However, reloading parameters through logic is usually significantly faster than reconfiguring a circuit. Hence, this can usually be neglected unless the dataset size n is very small. Similarly, the total processing time of a reconfigurable design can be calculated as:

$$t_{total,rec} = t_{p,rec} \cdot n + t_r \tag{4.6}$$

With equation 4.2 we obtain:

$$t_{total,rec} = t_{p,rec} \cdot n + \frac{A \cdot \Theta}{\phi_{config}} \tag{4.7}$$

As explained above, a reconfigurable design can be faster than a non-reconfigurable version, i.e. $t_{p,rec} < t_{p,nrec}$. It is therefore possible to amortise the configuration time with shorter processing times and archive an overall speed-up, given that:

$$\begin{aligned} t_{total,nrec} &> t_{total,rec} \\ t_{p,nrec} \cdot n &> t_{p,rec} \cdot n + t_r \\ t_{p,nrec} - t_{p,rec} &> \frac{t_r}{n} \\ t_{p,nrec} - t_{p,rec} &> \frac{A \cdot \Theta}{\phi_{config} \cdot n} \end{aligned} \tag{4.8}$$

For a reconfigurable design to provide a performance benefit, the reduction in processing time must be larger than the reconfiguration overhead incurred. The larger the gap between non-reconfigurable processing time $t_{p,nrec}$ and reconfigurable processing time $t_{p,rec}$, the easier it becomes to amortise the configuration time t_r. Processing more data n between reconfigurations also reduces the impact of configuration time. Configuration time itself can be reduced by using high configuration speeds ϕ_{config} and configuring the smallest possible area A. Area can be reduced by packing logic as much as possible and choosing a reconfigurable region such that the boundaries are aligned with the granularity of the configuration memory. If a region is not aligned accordingly, then reconfiguration will also cover static areas surrounding the region, unnecessarily increasing the configuration size and configuration time. The placement of regions is further covered in section 5.3.

Given a reconfigurable and a non-reconfigurable version of a design, we can calculate the dataset size n at which the reconfigurable design shows a performance benefit:

$$n > \frac{t_r}{t_{p,nrec} - t_{p,rec}} = \frac{A \cdot \Theta}{(t_{p,nrec} - t_{p,rec}) \cdot \phi_{config}} \tag{4.9}$$

4.5.2 Exploring parallelism for performance

We now propose a more detailed analysis that also considers the effect of parallelism on processing time and reconfiguration time. Many applications can be scaled between a small and slow serial implementation, and a large and fast parallel implementation. Most signal processing applications are characterised by iterative algorithms. Such algorithms provide the opportunity for unrolled, parallel implementations, or for serial implementations that use shared operators. FIR filters [83], FFTs [23], AES encryption [2] and Cordic [110] are examples of such algorithms.

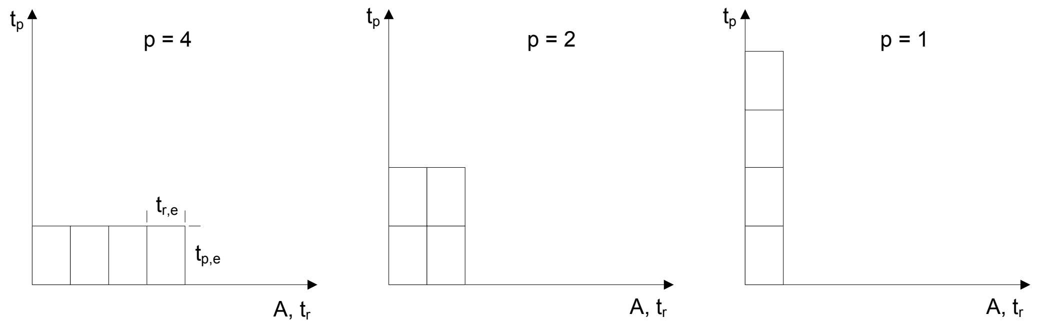


Figure 4.1: Different spatial and temporal mappings of an algorithm with $s = 4$ steps.

The degree of parallelism can be adjusted in both reconfigurable and non-reconfigurable designs. In a non-reconfigurable design, parallelism reduces processing times as multiple steps can be computed concurrently. A fully parallel implementation can therefore be assumed to be the highest performing one. The situation is more complex for reconfigurable designs because parallelism has opposite effects on processing times and reconfiguration times. This is illustrated in figure 4.1. Shown are three different spatial and temporal mappings of an algorithm with regard to processing time, area and reconfiguration time. Rectangles represent basic processing elements, and a processing element is characterised by its elementary processing time $t_{p,e}$ and its elementary reconfiguration time $t_{r,e}$. A fully parallel design for an algorithm with s steps employs s processing elements working concurrently, and therefore produces one result for each time period $t_{p,e}$. A serial design uses one time-shared processing element and requires a time of $s \cdot t_{p,e}$ to compute one result. It is also possible to create implementations with intermediate degrees of parallelism. Like in a non-reconfigurable design, parallelism speeds up processing: the processing time per datum t_p is inversely proportional to the degree of parallelism p:

$$t_p = \frac{t_{p,e} \cdot s}{p} \tag{4.10}$$

On the other hand, parallelism slows down reconfiguration because a parallel implementation is larger than a sequential one and reconfiguration time is directly proportional to the area as shown in equation 4.2. The fully parallel design requires to reconfigure s elements resulting a reconfiguration time of $s \cdot t_{r,e}$. The serial design contains only one processing element and requires a total reconfiguration time of $t_{r,e}$. Hence, the reconfiguration time t_r is directly proportional to the degree of parallelism p:

$$t_r = t_{r,e} \cdot p \tag{4.11}$$

We can now calculate the total processing time for a workload of n data items:

$$t_{total} = n \cdot t_p + t_r = \frac{t_{p,e} \cdot s \cdot n}{p} + t_{r,e} \cdot p \tag{4.12}$$

The above equation normalised to n gives the processing time per data item:

$$t_{total,n} = \frac{t_{p,e} \cdot s}{p} + \frac{t_{r,e} \cdot p}{n} \tag{4.13}$$

Figure 4.2 illustrates how parallelism can affect the optimality of the processing time. We consider an algorithm with $s = 256$ steps. The plots are normalised to processing time per datum and we assume that the reconfiguration time $t_{r,e}$ of one processing element is 5000 times the processing time $t_{p,e}$ of one processing element. This value can vary depending on the application and target device but we estimate that at least the order of magnitude is realistic for current devices. We observe that fully sequential implementations are beneficial for small workloads. In this case, the short configuration time outweighs the longer processing time. However, the overall time is still high due to the large influence of the configuration time. Large workloads benefit from a fully parallel implementations since the processing time is more dominant than reconfiguration time. In case of medium workloads, the degree of parallelism can be tuned to optimise the processing time. An analysis of how different configuration speeds can influence the optimal implementation is shown in section 4.5.6. Based on equation 4.13 we can perform two optimisation approaches: fastest design or smallest design that satisfies a given application throughput requirement.

Fastest design

In order to find the fastest design we search for the minimum in equation 4.13. This can be done by calculating the partial derivative of equation 4.13 with respect to p:

$$\frac{\partial t_{total}}{\partial p} = -\frac{t_{p,e} \cdot s}{p^2} + \frac{t_{r,e}}{n} \tag{4.14}$$

To find the minimum, we set equation 4.14 to 0 and solve for p:

$$p_{opt} = \sqrt{\frac{s \cdot n \cdot t_{p,e}}{t_{r,e}}} \tag{4.15}$$

The result p_{opt} is usually a real number which is not a feasible value to specify parallelism. In order to determine a practical value for p, p_{opt} can be interpreted according to table 4.2.

$0 < p_{opt} \leq 1$	fully serial implementation, $p = 1$
$1 < p_{opt} < s$	choose p such that $s/p \in \mathbb{Z}$ and $\lvert p_{opt} - p \rvert$ minimal
$s \leq p_{opt}$	fully parallel implementation, $p = s$

Table 4.2: Interpretation of p_{opt} to determine a practical value for p.

If an application throughput requirement Φ_{app} is given then it is necessary to check if this requirement is met by the hardware implementation:

$$\frac{1}{t_{total,n}} = \Phi_{hw} \geq \Phi_{app} \tag{4.16}$$

The resulting area requirement A also has to be feasible within the total available area A_{max}. If the design meets the performance and area requirements then we can calculate the buffer size according to equation 4.3.

Smallest design for given throughput requirement

If an application throughput requirement is given then it may not be necessary to implement the highest performing design. Instead one can try to find the smallest design that provides sufficient performance according to the requirement. To achieve a throughput of Φ_{app}, each data item has to be processed in:

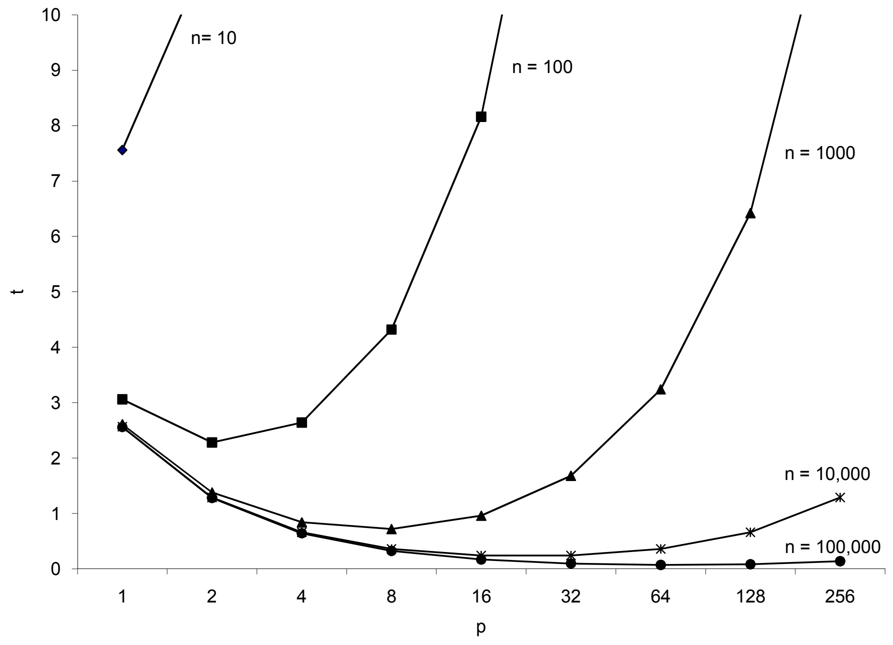


Figure 4.2: Normalised processing times for a range of workload sizes n and different levels of parallelism p. The number of steps s is set to 256 and we assume $t_{r,e} = 5000 t_{p,e}$.

$$t_{app} = \frac{1}{\Phi_{app}} \tag{4.17}$$

To find designs that provide exactly the required throughput, we set equation 4.13 equal to the application processing time t_{app} and solve for p. If multiple solutions for p exist, then the smallest solution also represents the smallest design that provides the required throughput.

$$\begin{aligned} t_{app} &= t_{total,n} \\ \frac{1}{\Phi_{app}} &= \frac{t_{p,e} \cdot s}{p} + \frac{t_{r,e} \cdot p}{n} \end{aligned} \tag{4.18}$$

This results in a quadratic equation for p:

$$\Phi_{app} \cdot t_{r,e} \cdot p^2 - n \cdot p + \Phi_{app} \cdot t_{p,e} \cdot s \cdot n = 0 \tag{4.19}$$

The roots of equation 4.19 are:

$$p = \frac{n \pm \sqrt{n^2 - 4 \cdot \Phi_{app}^2 \cdot t_{r,e} \cdot t_{p,e} \cdot s \cdot n}}{2 \cdot \Phi_{app} \cdot t_{r,e}} \tag{4.20}$$

If equation 4.20 has more than one real solution, then the smaller solution represents p_{min}. Again, p_{min} may not be a feasible value for parallelism and we have to interpret p_{min} according to table 4.3. For this interpretation we also require p_{max}, the larger solution of equation 4.20 which represents the largest design that satisfies the throughput requirement.

$0 < p_{min} < 1$ and $0 < p_{max} < 1$	real design not feasible
$0 < p_{min} < 1$ and $p_{max} \geq 1$	fully serial implementation, $p = 1$
$1 \leq p_{min} \leq s$	choose p such that $s/p \in \mathbb{Z}$, $p \geq p_{min}$ and $p - p_{min}$ minimal
$s < p_{min}$	real design not feasible

Table 4.3: Interpretation of p_{min} to determine a practical value for p.

Again, we can dimension the buffer according to equation 4.3. Since the design is now minimal in size, the buffer size is also minimal.

4.5.3 Optimisation steps

In the previous section we describe how a variation in the degree of parallelism of the implementation can be used to optimise a design for performance or size given required performance. It is of course possible to perform an optimisation by simply building designs for all possible variations of p and evaluating their performance but this would be a very time consuming process. Instead we propose to build one design, derive the necessary parameters and calculate the optimum based on our equations. In the following it is demonstrated how a design can be optimised for performance but the same procedure applies to optimisation for size. For performance, we solve equation 4.15 based on the parameters from the initial design. We then interpret p according to table 4.2 and build a design for this value.

In some designs the parameters may not scale exactly as assumed in our model. For example, some designs may incur an area overhead for serial implementations while in other cases, parallelising a design may cause an area overhead. Even if parameters do scale not exactly as assumed, we can still use our exploration to guide us to the optimum. We build a design for the predicted optimum and then reiterate the process if necessary: If the design performs worse than predicted, p has to be moved closer to the value of the original implementation. If the design is more efficient, then p can be moved further out. In summary, to implement an optimised design according to our model, the following ten steps have to be carried out:

1. Derive Φ_{app}, s and n from application.

2. Obtain Φ_{config} and Θ for target device.
3. Implement one design and determine t_p and A.
4. Calculate t_r, $t_{p,e}$ and $t_{r,e}$ using equations 4.2, 4.10 and 4.11.
5. Find p_{opt} from equation 4.15 and determine a feasible value according to table 4.2.
6. Calculate $t_{total,n}$ using equation 4.13 and verify throughput using equation 4.16.
7. Implement design with p from step 5.
8. Verify if design matches predicted parameters.
9. If necessary, refine results by moving p up or down.
10. Calculate buffer size B using equation 4.3, storage size Ψ using equation 4.1 and check $A \leq A_{max}$.

If a design scales well, then the above method will find the optimum right away. If parameters vary, a reiteration of steps 7 and 8 may be necessary. This is still considerably less effort than building all possible design variations.

4.5.4 Trends for processing and reconfiguration time variations

In our model we assume that the elementary processing time $t_{p,e}$ and the elementary reconfiguration time $t_{r,e}$ are constant when scaling the degree of parallelism. In practice this may not always be the case. When verifying the design in step 8 of our optimisation method we may find that $t_{p,e}$ or $t_{r,e}$ have changed. If this is the case, then the design will not scale as assumed, and the actual optimum can be different from our predicted value.

We now analyse how these variations affect p_{opt}. An exact analytical solution to this problem is not practical as the functions that influence $t_{p,e}$ and $t_{r,e}$ are generally not known. Deriving these functions would require building a wide range of designs and analysing the effects in detail. To maintain the simplicity of our approach, we study the general trends with varying processing and reconfiguration times. These trends can then be used as guidelines to refine a design if variations in $t_{p,e}$ and $t_{r,e}$ are observed in practice.

First, we consider a scenario where $t_{p,e}$ varies when changing the degree of parallelism. We begin with a fully parallel design to derive the initial parameters. When the design is serialised and p reduced, $t_{p,e}$ could increase because a shared processing element might be slower than a specialised element in the parallel design. Another possibility is a reduction

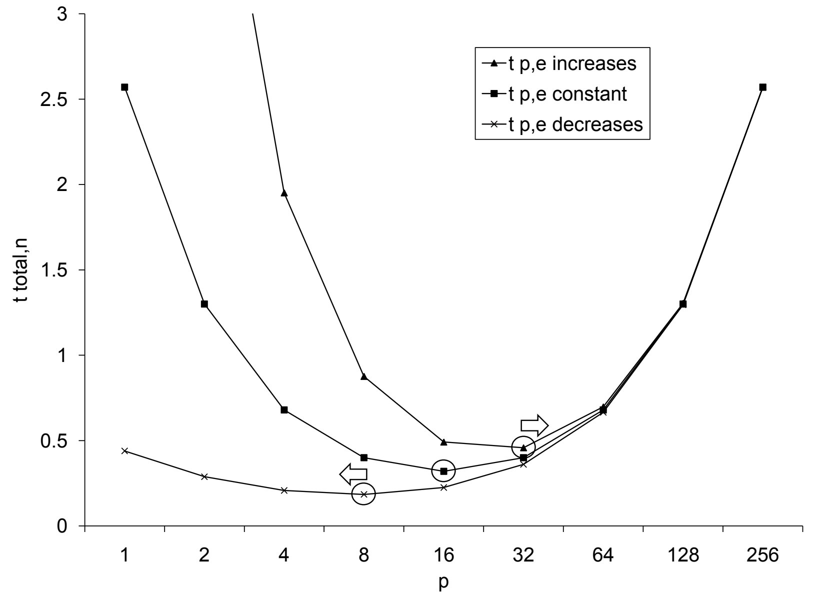


Figure 4.3: Trends for p_{opt} when the elementary processing time $t_{p,e}$ varies.

in $t_{p,e}$ when the design is serialised. A large parallel design may be hampered by long propagation delays and reducing the size of the design could make it faster. Figure 4.3 shows the influence of these two effects on the total processing time in comparison to a design where $t_{p,e}$ is constant. Shown are the cases where $t_{p,e}$ is increased or reduced by 20% for each step serialisation. We observe that a continuous reduction of $t_{p,e}$ has a significant influence on the total processing time as p decreases. The design is faster than projected and p_{opt} is reduced. On the other hand, a continuous increase in $t_{p,e}$ leads to higher total processing times, and p_{opt} is shifted to a higher value.

In our model we assume that each processing element requires a fixed amount of area, but if this area varies with p then this will also lead to changes in $t_{r,e}$. Again, we begin with a fully parallel design to derive the initial parameters for A and $t_{r,e}$. When the design is serialised, $t_{r,e}$ could increase because shared processing elements can be less area efficient. On the other hand, there may be area overheads associated with a parallel design, and $t_{r,e}$ could be lower than expected when p is reduced. Both scenarios are shown in figure 4.4. Again, we illustrate the case when the parameter improves or deteriorates by 20% for each step serialisation and compare them to a constant $t_{r,e}$. We observe that a continuous reduction in $t_{r,e}$ leads to lower overall processing times mainly for larger values of p. A decrease in $t_{r,e}$ therefore tends to shift p_{opt} to a larger value. If $t_{r,e}$ is increased, p_{opt} tends to shift to a smaller value. However, this effect is not as strong as for changes in processing time. The same rate of variation does not dislocate the optimum from its original value.

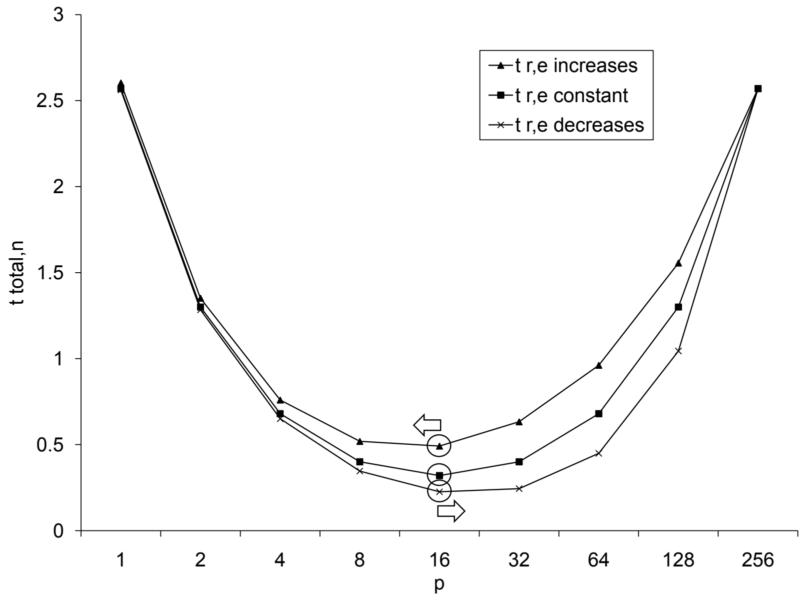


Figure 4.4: Trends for p_{opt} when the elementary reconfiguration time $t_{r,e}$ varies.

The above observations can be used as guidelines to adjust the result of our optimisation for designs that do not scale exactly as assumed. If a design exhibits only one of the two variations, then one can simply explore another value for p according to the trend that is shown above. If both parameters vary in such a way that they lead to the same trend, then the solution is also obvious. For example, if one builds a design for the predicted optimum and observes an increase in $t_{p,e}$ and a reduction in $t_{r,e}$, then both effects work in synergy and suggest that a larger value for p should be tried. If competing effects are observed, however, no simple recommendation can be given. If, for example, both $t_{p,e}$ and $t_{r,e}$ increase or decrease at the same time, then the overall trend is not known and it may be necessary to try both a larger and a smaller value for p. Since the effect of processing time variations is stronger than the one caused by reconfiguration time, it is preferable to explore the trend suggested by $t_{p,e}$ first.

When optimising the size of the design for a given throughput requirement, we face increasingly serialised implementations. For small values of p, variations in processing times have a large influence while reconfiguration time variations are negligible. Hence, the design should be refined according to trends shown in figure 4.3.

4.5.5 Pipelined designs

Signal processing functions are often implemented in a pipelined fashion. Pipelining is a form of parallelism in the sense that multiple processing elements are used to process

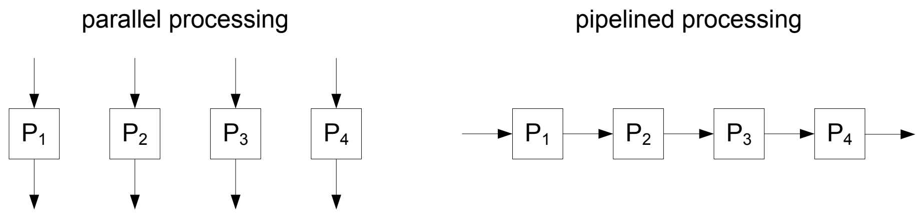


Figure 4.5: Parallel processing and pipelined processing. A pipeline with p processing elements has an additional latency of $p-1$ cycles.

data concurrently. However, parallel execution is commonly interpreted as concurrent and independent processing of data. The key difference in pipelined processing is that data propagates through a series of processing elements. This is illustrated in figure 4.5. Both approaches deliver high data throughput, but pipelined processing leads to an increased latency of $p-1$ cycles for a pipeline with p processing elements.

Optimising pipelined designs with our approach is very similar to optimising designs with parallelism: processing elements can be time-shared to execute multiple processing steps on each data item. This reduces the area and configuration time, while processing time increases. Time-sharing processing elements in a pipeline may not be practical for all applications, but in signal processing, pipeline stages often perform very similar functions. Hence, they may be combined into a shared processing element that is smaller but requires multiple cycles to compute one result.

The principal difference when applying our optimisation to pipelines is the additional latency. If we include the additional latency of $p-1$ processing cycles into equation 4.12, we obtain:

$$t_{total} = n \cdot t_p + t_r = t_{p,e} \cdot \left(\frac{s \cdot n}{p} + (p-1) \right) + t_{r,e} \cdot p \tag{4.21}$$

The optimal degree of parallelism for the above equation is:

$$p_{opt} = \sqrt{\frac{s \cdot n \cdot t_{p,e}}{t_{r,e} + t_{p,e}}} \tag{4.22}$$

In practice, latency does not have a significant effect on our optimisation method because reconfiguration times are much slower than processing times, i.e. $t_{r,e} \gg t_{p,e}$. The difference between equation 4.15 and 4.22 is therefore marginal.

4.5.6 Influence of configuration speed

Reconfiguration speed is an important factor in our approach, and high configuration speeds are necessary to reduce the impact of the configuration time overhead. We now analyse how improvements in configuration speed affect the total processing time and optimal degree of parallelism. Figure 4.6 illustrates the normalised processing time for the same parameters as in figure 4.2 and a workload size of 1000. Also shown is the same function with a configuration speed-up of 10x and 100x. With higher configuration speeds, the optimal implementation is shifted towards higher degrees of parallelism.

If the reconfiguration time in equation 4.15 is reduced by a factor of x then p increases with $\sqrt{x}$, assuming that p stays in the range between 1 and s. Hence, the processing time t_p in equation 4.13 will also be reduced by $\sqrt{x}$. The reconfiguration time t_r will also be reduced by $\sqrt{x}$ because $t_{r,e}$ is reduced by x and p is increased by $\sqrt{x}$. Consequently, the entire processing time t_{total} will decrease with $\sqrt{x}$. This means that the performance Φ_{hw} of a reconfigurable system will increase by $\sqrt{x}$ if the reconfiguration throughput is increased by a factor of x.

Fully serial designs with $p = 1$ are dominated by reconfiguration time. For such a design, a reconfiguration speed-up by a factor of x will also lead to a speed-up of the entire design by approximately x. In fully parallel designs with $p = s$, reconfiguration time does not play a significant role. Thus, speeding up reconfiguration will not lead to significant performance improvements in the design.

A side effect of reducing configuration times is that buffer requirements are also reduced. If the degree of parallelism in the design is not changed, then a reduction of the reconfiguration time by x in equation 4.3 will also lead to a reduction in buffer size by a factor of x. If however the parallelism is increased by $\sqrt{x}$ then the overall buffer size is reduced by $\sqrt{x}$.

4.6 Energy aware optimisation

We now apply our optimisation approach to power and energy. Even though the term “low power” is often used to describe the efficiency of a design it is more appropriate to consider energy per amount of activity. The overall efficiency of a design can be judged by the total amount of energy required for a certain computation including reconfiguration. In the following we develop an energy optimisation technique that shows when a reconfigurable application becomes more energy efficient than a non-reconfigurable one. We then explore

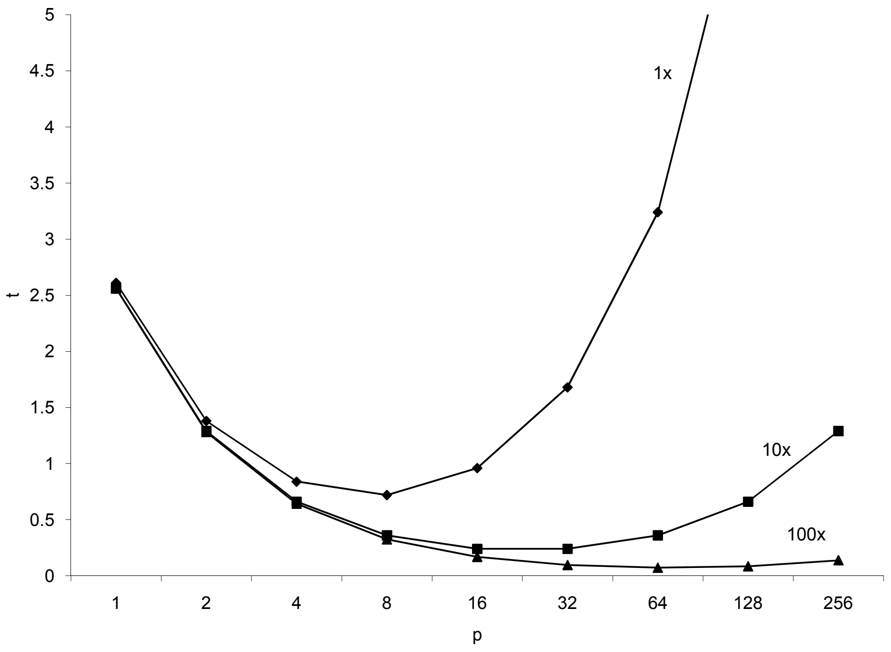


Figure 4.6: Normalised processing times for accelerated configuration speeds. The number of steps s is set to 256 and the workload size n to 1000.

the degree of parallelism for further energy optimisations in the reconfigurable design.

4.6.1 Energy efficiency in reconfigurable designs

Energy is the product of power and time, and it can be lowered by reducing either component, or reducing both at the same time. Module specialisation can improve the energy efficiency of design by reducing both processing time and processing power simultaneously. As explained in section 2.9, a specialised module can be faster than a general purpose one. At the same time, it can also consume less power. A specialised circuit can have less switching activity which reduces dynamic power. It can also require less area, allowing to target a smaller device which uses less static power. Because of these combined reductions, specialisation can yield good improvements in processing energy. This can be exploited in reconfigurable designs, but we also have to consider the energy overhead of reconfiguration.

Figure 4.7 illustrates the energy required by a non-reconfigurable and a reconfigurable design. In the non-reconfigurable design, energy consumption is dominated by the processing energy $E_{p,nrec}$, but it may also include an energy E_{load} for loading new parameters between processing data sets. However, this component can be neglected in many cases as the load time is usually very short compared to the reconfiguration time. The energy

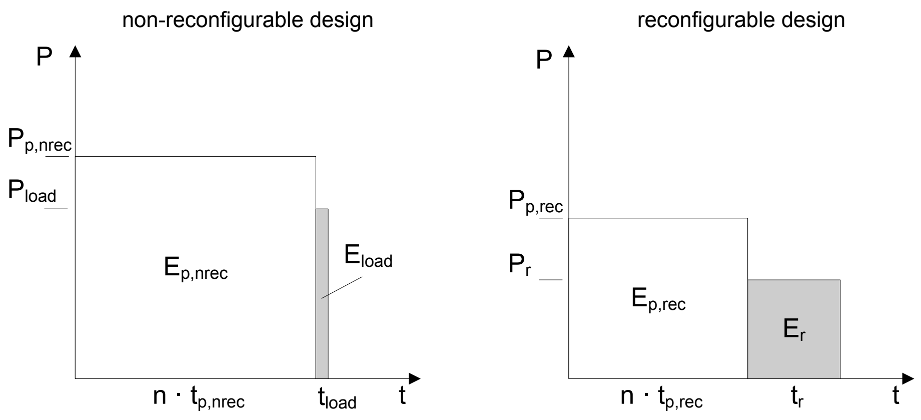


Figure 4.7: Energy components in non-reconfigurable and reconfigurable designs

required to process n data items in a non-reconfigurable implementation is:

$$\begin{aligned} E_{total,nrec} &= E_{p,nrec} + E_{load} \\ &= P_{p,nrec} \cdot t_{p,nrec} \cdot n + P_{load} \cdot t_{load} \\ &\approx P_{p,nrec} \cdot t_{p,nrec} \cdot n \end{aligned} \tag{4.23}$$

The reconfigurable design in figure 4.7 requires less processing energy because it is faster and consumes less processing power. There is however a significant reconfiguration energy E_r that needs to be considered. For a reconfigurable design, the energy to process n data items is given as follows:

$$\begin{aligned} E_{total,rec} &= E_{p,rec} + E_r \\ &= P_{p,rec} \cdot t_{p,rec} \cdot n + P_r \cdot t_r \\ &= P_{p,rec} \cdot t_{p,rec} \cdot n + P_r \cdot \frac{\Theta \cdot A}{\phi_{config}} \end{aligned} \tag{4.24}$$

In order to be more energy efficient than the non-reconfigurable design, the savings in E_p must be larger than the overhead in reconfiguration energy E_r. Like in the performance analysis in section 4.5.1, the reconfiguration overhead is reduced by reconfiguring the smallest possible area A with the highest available speed ϕ_{config}, and by reconfiguring between larger dataset sizes n. To achieve a low E_r, we also require low reconfiguration power P_r. The dataset size n at which the reconfigurable design becomes more energy efficient than the non-reconfigurable one is calculated as follows:

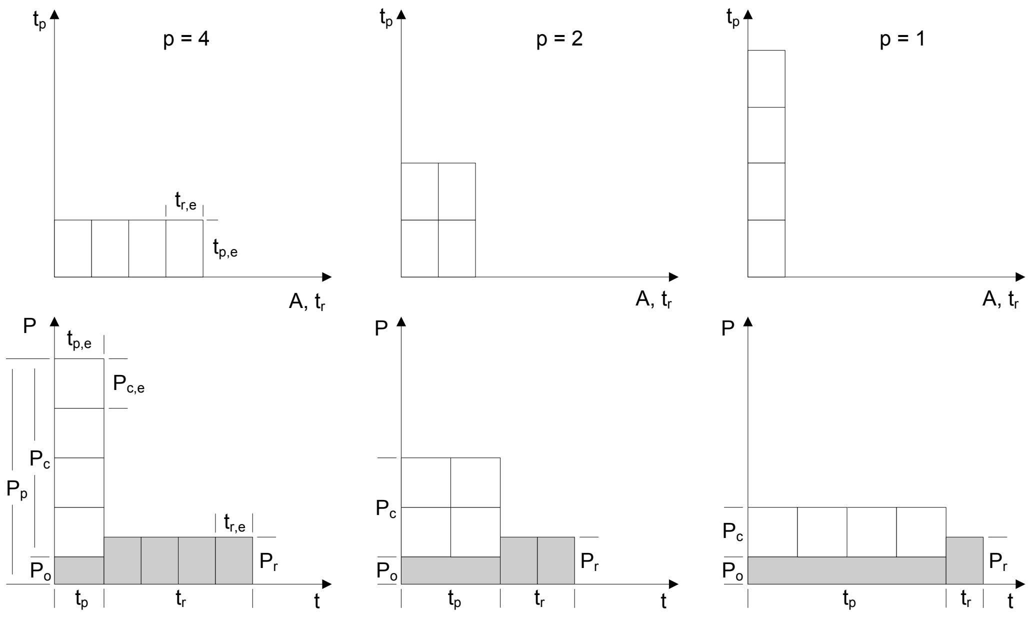


Figure 4.8: Different temporal and spatial mappings of an algorithm with $s = 4$ steps. The upper diagrams show processing time, reconfiguration time and area. The lower diagrams show the influence of p on power for $n = 1$. We have to minimise the energy represented by the grey rectangles.

$$n > \frac{P_r \cdot t_r}{P_{p,nrec} \cdot t_{p,nrec} - P_{p,rec} \cdot t_{p,rec}} = \frac{P_r \cdot A \cdot \Theta}{(P_{p,nrec} \cdot t_{p,nrec} - P_{p,rec} \cdot t_{p,rec}) \cdot \phi_{config}} \tag{4.25}$$

4.6.2 Exploring parallelism for energy

We now consider how parallelism influences the energy efficiency in a reconfigurable design. Figure 4.8 illustrates the implication of parallelism on energy. The upper three diagrams show three different temporal and spatial mappings of an algorithm with four steps, and processing elements are characterised by $t_{p,e}$ and $t_{r,e}$. The lower three diagrams show the implication of p on power and energy, assuming $n = 1$. In our analysis we use four power values: P_p is the total power during processing. It is a combination of computation power P_c and power overhead P_o. P_r is the power during reconfiguration.

Instead of static and dynamic power, we consider power components that scale with p and components that remain constant. We assume that each processing element incurs a certain computation power $P_{c,e}$. Hence, the computation power P_c is directly proportional to p:

$$P_c = P_{c,e} \cdot p \tag{4.26}$$

Using equation 4.10 we obtain the computation energy E_c for processing n data items:

$$E_c = P_{c,e} \cdot p \cdot t_p \cdot n = P_{c,e} \cdot t_{p,e} \cdot s \cdot n \tag{4.27}$$

The energy associated with computation is independent of p and is constant for an algorithm with given s and n.

During processing, we will also find a power overhead P_o that is not directly associated with computation. This overhead may be static power only. However, there can be further elements to this power overhead e.g. the power consumption of auxiliary circuits such as clock managers. This power component is constant and does not scale with p. With equation 4.10 we find that the energy overhead E_o encountered during the processing of n data items is inversely proportional to p:

$$E_o = P_o \cdot t_p \cdot n = P_o \cdot \frac{t_{p,e} \cdot s \cdot n}{p} \tag{4.28}$$

Finally, we have to consider the power and energy consumed during the reconfiguration process. We assume that reconfiguration power P_r is constant. For P_r we do not have to consider the power overhead separately. Reconfiguration energy simply scales with reconfiguration time regardless of the distribution of its components. The reconfiguration energy E_r grows with reconfiguration time t_r and is therefore proportional to p. With equation 4.11 we obtain:

$$E_r = P_r \cdot t_r = P_r \cdot t_{r,e} \cdot p \tag{4.29}$$

The total energy for processing n data items is therefore:

$$\begin{aligned} E_{total} &= E_p + E_r = E_c + E_o + E_r \\ &= P_{c,e} \cdot t_{p,e} \cdot s \cdot n + P_o \cdot \frac{t_{p,e} \cdot s \cdot n}{p} + P_r \cdot t_{r,e} \cdot p \end{aligned} \tag{4.30}$$

The total energy normalised to n is:

$$E_{total,n} = P_{c,e} \cdot t_{p,e} \cdot s + P_o \cdot \frac{t_{p,e} \cdot s}{p} + P_r \cdot \frac{t_{r,e} \cdot p}{n} \tag{4.31}$$

Figure 4.9 shows the normalised energy per datum over p for the parameters $s = 128$, $n = 10,000$, $t_{p,e}/t_{r,e} = 5 \cdot 10^{-4}$ and $P_o/P_r = 0.5$. A variation of either n or $t_{p,e}/t_{r,e}$ with the factor a leads to different optima. We observe the same trends as in our performance optimisation approach: larger datasets or higher processing time to reconfiguration time

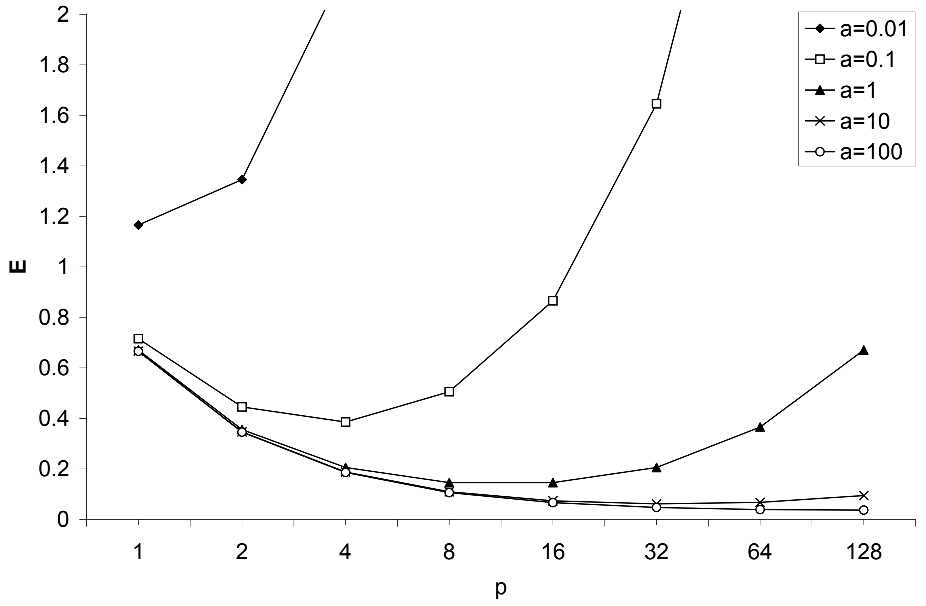


Figure 4.9: Normalised energy per datum for parameter variations of dataset size n, or the processing to configuration time ratio $t_{p,e}/t_{r,e}$.

ratios (faster reconfiguration) push the optimum towards more parallel implementations. Smaller data sets or slower reconfiguration shift the optimum to a more serial solution.

In order to find the optimal degree of parallelism that minimises the energy per datum, we calculate the partial derivative of equation 4.31 with respect to p:

$$\frac{\partial E_{total,n}}{\partial p} = -\frac{P_o \cdot t_{p,e} \cdot s}{p^2} + \frac{P_r \cdot t_{r,e}}{n} \tag{4.32}$$

To find the minimum we set equation 4.32 to 0 and solve for p:

$$p_{opt} = \sqrt{\frac{P_o \cdot t_{p,e} \cdot s \cdot n}{P_r \cdot t_{r,e}}} \tag{4.33}$$

As in our performance optimisation in section 4.5, p_{opt} is usually not a feasible value to specify parallelism. Again, we can determine a feasible value for p by interpreting p_{opt} according to table 4.2.

This optimisation is independent of computation power P_c. It balances the energy associated with the power overhead P_o and the energy associated with reconfiguration power P_r. Graphically this corresponds to reducing the combined area of the grey rectangles in figure 4.8. The area of the white rectangles is constant.

4.6.3 Optimisation steps

Similarly to our performance optimisation in section 4.5, we can explore the design space by building one design, deriving its parameters, calculating p, building the design for the predicted optimum and refining the results if necessary. The following nine steps have to be carried out:

1. Derive s and n from application.
2. Obtain Φ_{config} and Θ for target device.
3. Implement one design and determine t_p, A, P_r and P_0.
4. Calculate t_r, $t_{p,e}$ and $t_{r,e}$ using equations 4.2, 4.10 and 4.11.
5. Find p_{opt} from equation 4.33 and determine a feasible value according to table 4.2.
6. Build design for p.
7. Verify if design matches predicted parameters.
8. If necessary refine result by moving p up or down.
9. Calculate buffer size B using equation 4.3, storage size Ψ using equation 4.1 and check $A \leq A_{max}$.

In step 7 we may find that the design does not scale as predicted because $t_{p,e}$ or $t_{r,e}$ have changed. If this is the case, the design will follow the same trends as shown for performance. Hence, the guidelines presented in section 4.5.4 can also be used for energy optimisation.

4.7 FPGA power measurements

Our optimisations can be performed based on power estimation tools or power measurements. Xilinx currently offers two tools for power estimation. Xilinx Power Estimator provides spreadsheets that allow power estimation based on used resources, clock rates and toggle rates for a particular device. This is useful for fast power estimations early in the design flow but is limited in accuracy. Xilinx XPower is more accurate. It performs power estimation based on a physically implemented design and activity estimates that are derived from design inputs. However, both tools lack support for reconfiguration power analysis. We therefore base our experiments on power measurements. Power measurements can be very accurate but require a target board with power measurement facilities as well as adequate measuring setup and equipment.

The power consumption of a device can be determined by its supply voltage and the current flow through the device:

$$P = V_{CC} \cdot I \tag{4.34}$$

There are two techniques that are commonly used for current sensing: **resistive methods** and **magnetic methods**.

Resistive current sensing involves placing a resistor into the circuit and measuring the voltage drop across the resistor. According to Ohm's Law, the voltage drop across the resistor is proportional to the current flowing through the resistor (equation 4.35). This can be used as a very simple and low-cost measurement setup that works well for very low to medium currents. However, it is an intrusive measurement, as the voltage drop across the resistor influences the circuit. Current flowing through the resistor also leads to power being dissipated by the resistor. In order to reduce both effects, the resistance should be small. For this purpose, special current sense resistors with low resistance and high accuracy are commercially available.

$$V_{sense} = R_{sense} \cdot I \tag{4.35}$$

Magnetic current sensing is based on measuring the magnetic field caused by a current flow. This technique is non-intrusive as it does not cause a voltage drop in the measured circuit. Two principles are available: current transformers and hall sensors. Current transformers only work for alternating currents and are commonly used in high-voltage and high-current situations. Hall sensors produce a voltage difference across a semiconductor that is exposed to a magnetic field and work on both direct currents and alternating currents. Magnetic sensing usually involves using a current clamp. This is a magnetic core that is clamped around a wire carrying a current. This contactless method is often used for electrical safety reasons but the accuracy at very low currents can be limited. Current clamps are also considerably more expensive than current sense resistors.

Because of its simplicity, high accuracy and low cost we use resistive current measurement. When using current sense resistors, two setups are possible: **low-side measurements** where the resistor is placed between the device and ground, and **high-side measurements** where the resistor is placed between the supply voltage and the device.

A **low-side measurement** is the simplest setup as the voltage that is to be measured is ground referenced. However, such a setup will lead to a lifted ground potential on the measured device. This can be undesirable if the device is also connected to other components that operate based on true ground potential. From a practical perspective it

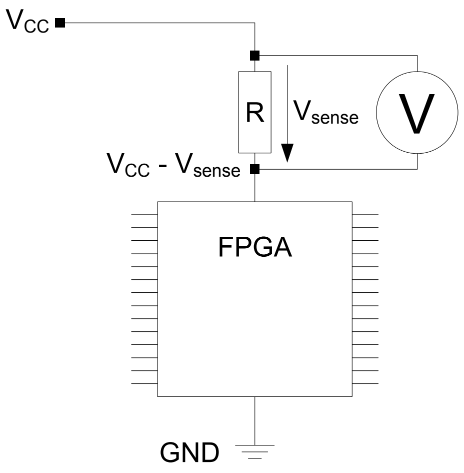


Figure 4.10: Resistive high-side FPGA power measurement.

is also often difficult to access the ground plane on an integrated circuit board to insert a current sense resistor.

High-side measurements measure current on the supply voltage side and do not disturb the ground level, but the voltage drop between supply rail and device still needs to be considered. It is important that the lowered input voltage does not violate the required operating conditions of the device. It can also be more difficult to measure a small signal with a high common mode. This usually requires an operational amplifier or measurement equipment with good common mode rejection.

For FPGA power measurements, low-side measurements are usually ruled out by the fact that most circuit boards use a common ground plane. Splitting or modifying this ground plane is not possible. Some FPGA boards offer current measurement points on the high side in form of jumpers between the voltage regulators and the device. These jumpers can be removed and replaced with current sense resistors. In our experiments we also use a Xilinx ML505 board that does not provide any current measurement facilities. We therefore modify the board by cutting a connection between core supply voltage regulator and the device, and insert a current sense resistor at this point. The device power for resistor-based high-side measurements can be calculated as shown in equation 4.36. Table 4.4 lists supply voltages, and hence common mode voltages, for several Xilinx FPGAs.

$$P = \frac{(V_{CC} - V_{sense}) \cdot V_{sense}}{R_{sense}} \tag{4.36}$$

We can measure V_{sense} with a digital multimeter or a storage oscilloscope. We use a

device	V_{CC}	ΔV_{max}
Virtex-II Pro [130]	$1.5V$	$75mV$
Virtex-4 [133]	$1.2V$	$60mV$
Virtex-5 [134]	$1.0V$	$50mV$
Spartan-3 [132]	$1.2V$	$60mV$

Table 4.4: Core supply voltage and maximal allowed variation of core supply voltage for several Xilinx devices.

Hameg 8012 multimeter that allows measurements of non-ground references signals with a maximum resolution of $10\mu V$. The device has a common mode rejection of $\geq 100dB$. Thus, a common mode voltage of $1.5V$ will result in a maximum error of $15\mu V$. However, the time resolution of the multimeter is limited to one measurement per second. We also use a Tektronix MSO2024 4-channel digital storage oscilloscope that has a maximum sensitivity of $2mV$ per display divide and a maximum time resolution of $1ns$. The oscilloscope can only measure ground referenced signals. It is possible to measure differential signals by using two channels and subtracting one channel from the other but this results in a very low common mode rejection that is not suitable for our measurements. As an alternative, we make non-ground referenced measurements by floating the oscilloscope; i.e. the ground reference of the oscilloscope is removed.

FPGAs usually have separate supply rails for core logic and IOs. This is because higher voltages are needed to support common IO standards. In our experiments we measure FPGA core power only. IO power is highly depended on the loads connected to the IOs. This is to some degree pre-determined by the layout of the development board that is used for the experiment. IO power is therefore not part of the presented optimisations. Core power measurements are sometimes performed by wiring a multimeter directly into the device's supply rails. However, we find this practice unsuitable as this does not offer any control over the current sense resistor. FPGAs are sensitive to core voltage variations and have to stay within a narrow band of the nominal supply voltage. Table 4.4 shows the maximum allowed core voltage variation for several FPGAs. The voltage drop across the internal resistor of a multimeter can exceed this value if high currents are drawn. The FPGA will then cease to operate. Instead, we insert current sense resistors with a resistance small enough so that the maximum allowed voltage drop is not exceeded. Table 4.5 shows the maximum power that can be measured on a device with a given current sense resistor, based on the core supply voltage and maximum allowed core voltage variation as shown in table 4.4. All resistors used in our experiments have

R_{sense}	Virtex-II Pro	Spartan-3	Virtex-5
$5m\Omega$	$21.4W$	$13.7W$	$9.5W$
$50m\Omega$	$2.14W$	$1.37W$	$0.95W$
$470m\Omega$	$227mW$	$146mW$	$101mW$[1]

(1) not practical. Idle power exceeds this value.

Table 4.5: Maximum device power that can be measured for a range of current sense resistors.

1 % tolerance.

On the other hand, the resistance should not be too small as the accuracy would otherwise be limited. This is especially important when using the oscilloscope. The minimum value that can be read from the screen is roughly $0.2mV$ (one tenth of a display divide) while the maximum voltage drop allowed by the FPGA is between $50mV$ and $75mV$. The multimeter allows measurements down to $10\mu V$. We therefore have two setups that allow measurements with either low dynamic range and high time resolution or high dynamic range with low time resolution.

Measuring reconfiguration power P_r for our optimisation method is straightforward as this is simply the device power during reconfiguration. We also require total power during processing P_p to be split into its components P_o and P_c. P_o is the power overhead of a design not related to computation and can be determined by building a design with all the components that are not affected by reconfiguration. This design should include clock managers running at their targeted clock rate. Measuring the power of this design yields P_o. P_c can be calculated as the difference between P_o and P_p.

4.8 Case study

4.8.1 FIR filter

We demonstrate our optimisations on the example of a reconfigurable FIR filter. Such filters are used in many stages of a radio receiver or transmitter. An FIR filter is described by the following equation:

$$y[n] = \sum_{i=0}^{n-1} c_i \cdot x[n-i] \tag{4.37}$$

Each filter output value y is the weighted sum of the current and a finite number of previous input samples. The terms in the above equation are also commonly referred to as taps, and the number of filter coefficients c_i determines how many taps the filter has.

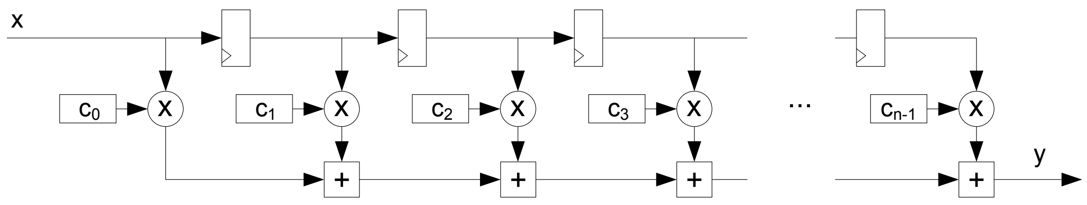


Figure 4.11: Basic structure of an n-tap FIR filter.

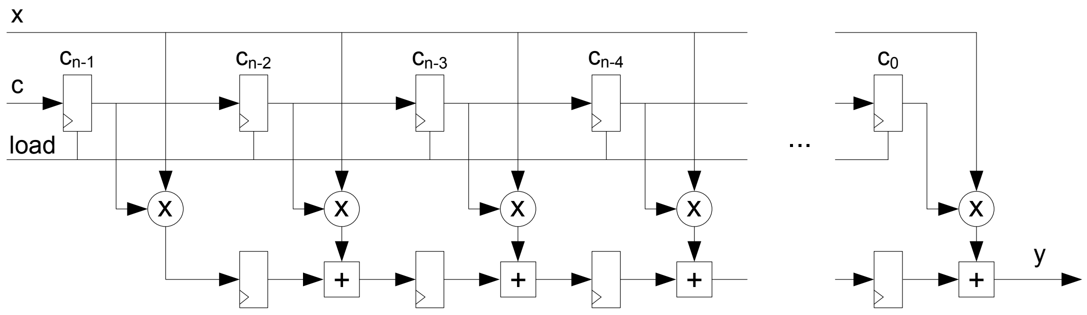


Figure 4.12: Transposed FIR filter with full multipliers and reloadable coefficients.

An n-tap filter requires n multiplications and $n-1$ additions for the computation of each output sample. In radio systems, input samples usually arrive in a continuous stream. FIR filters are therefore often implemented in a pipelined architecture that computes one output sample with each arriving input sample. Figure 4.11 illustrates a simple pipelined FIR filter. Input samples pass through a register chain, and are then multiplied with the filter coefficients and summed up.

The disadvantage of the filter structure as shown in figure 4.11 is the long propagation delay through the adder chain. Such a filter would have very low performance. This can be solved by using a transposed filter form as illustrated in figure 4.12. The transposed filter has the order of coefficient reversed and uses a pipelined adder chain which provides much better performance.

An FIR filter may require some form of modification. Examples of filter modifications are coefficient reloading to change the filtered frequency band, or template reloading in a matched filter. Flexible filters can be implemented as non-reconfigurable filters that provide additional circuitry to reload parameters into the design. For example, figure 4.12 shows the structure of a filter with reloadable coefficients that uses a register chain to load and store coefficients. This however makes the circuit more complex and it also requires the use of full multipliers.

As an alternative, filters can be reconfigurable where each instance is specific and optimised to one set of parameters. Figure 4.13 illustrates a fixed coefficient filter that

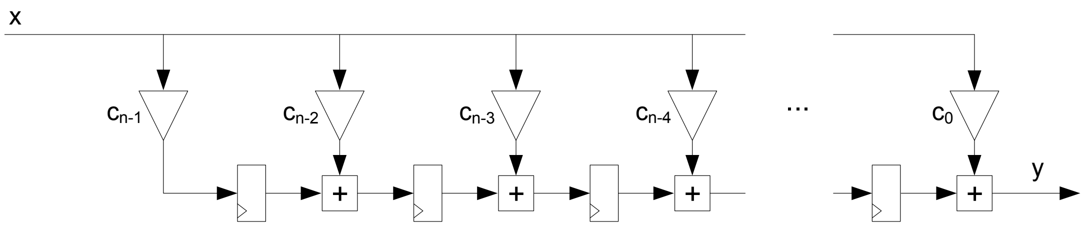


Figure 4.13: Transposed FIR filter with fixed coefficient multipliers.

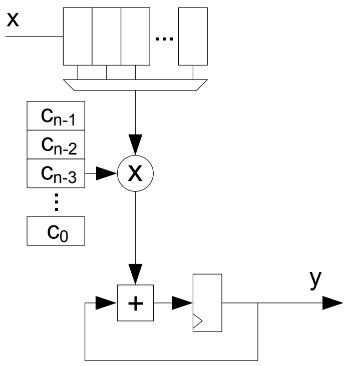


Figure 4.14: Serial FIR implementation. One multiplier and one adder are used to process all filter taps.

does not require storage of coefficients. It also uses multipliers that are specialised to the coefficients. Fixed coefficient multipliers are more area- and power-efficient and higher performing than general purpose ones. If a parameter change becomes necessary, the filter can be reconfigured.

Figure 4.12 and 4.13 show fully parallel filters that use one multiply-accumulate element for each filter tap and compute one result each clock cycle. It is also possible to vary the degree of parallelism and realise filters where multiply-accumulate elements are shared between several filter taps. For example, figure 4.14 shows a fully serial implementation where just one multiplier and adder is used. This filter processes all taps sequentially. It is also possible to create intermediate implementations where each multiplier is shared by several coefficients.

As an example, we now consider a channelisation FIR filter for GSM. The sample rate f_s for this filter is $2.167MHz$ which corresponds to 8 times the baseband bit rate. This is a realistic scenario after a first round of filtering and decimation. The application throughput Φ_{app} is therefore $2.167Msamples/s$. We filter a 200 kHz wide GSM channel and suppress neighbouring channels with an attenuation of at least $-80dB$. The filter coefficients are calculated with the MATLAB Simulink Digital Filter Design blockset.

This results in a filter with 80 coefficients. The number of processing steps s is therefore 80.

We implement a reconfigurable and a non-reconfigurable version of this filter in VHDL. The filters are described in the transposed form, and use 16 bit samples and 16 bit coefficients. The non-reconfigurable filter stores coefficients in a register chain as shown in figure 4.12. A coefficient reload requires 80 clock cycles. The reconfigurable filter contains hard-coded coefficients. The synthesis tools use this to create optimised fixed-coefficient multipliers. Both filters are synthesised with Xilinx XST 11 synthesis tools using IEEE arithmetic libraries. The target device is a Virtex-5 XC5VLX50T FPGA. In Virtex-5 FPGAs, the configuration size per unit of area is $\Theta_{CLB} = 295.2 bytes/CLB$ or $\Theta_{LUT} = 36.9 bytes/LUT$ [118]. The ICAP interface in Virtex-5 FPGAs can support a maximum configuration throughput of $\phi_{config} = 400MB/s$. However, when using the HwICAP core in combination with a MicroBlaze softcore processor we only measure a configuration throughput of $\phi_{config} = 5MB/s$. This relatively low transfer speed can be explained with transfers over the shared OPB bus and the fact that data has to be transferred into the HwICAP BRAM before it can be streamed into ICAP. Claus *et. al.* recently presented an improved ICAP controller that provides a throughput of $\phi_{config} = 300MB/s$ [22]. In the following we analyse how these two different transfer speeds influence the performance and energy efficiency of the reconfigurable design. We also compare two types of configuration controllers. The first type of controller is based on an embedded MicroBlaze processor and the HwICAP core. This system creates an overhead in terms of area and power. We also create a second version of the reconfigurable design based on an external configuration controller. No area and power overhead is added to the design. Even though an external controller will add some general overhead to the system, it could still be a lot more efficient than a MicroBlaze-based processor system. This scenario can be considered the best case in overall efficiency.

All measurements in this case study are performed with a digital storage oscilloscope and current sense resistor as explained in section 4.7. In the following, we first analyse the performance and energy efficiency of a reconfigurable filter and compare it with a non-reconfigurable counterpart. In the second part of our case study, we explore how the degree of parallelism can be used to further improve the reconfigurable design.

Comparing reconfigurable and non-reconfigurable designs

We now compare the performance and energy efficiency of our reconfigurable and non-reconfigurable filter designs. Table 4.7 shows the parameters of our three designs. The MicroBlaze-based reconfigurable design requires less than half of the logic resources than the non-reconfigurable version. The MicroBlaze system also requires BRAM resources which are not shown in this comparison. The reconfigurable design with external reconfiguration requires 64% less logic resources than the non-reconfigurable design. In both reconfigurable designs, the maximum clock frequency increases from 100 MHz to 182 MHz. This results in a 45% reduction in processing time. The processing power is reduced by 60% for the MicroBlaze-based design and by 64% for the design with external reconfiguration. Area and power savings and the performance increase can be explained with the fact that more efficient fixed-coefficient multipliers are used in the reconfigurable design.

However, reconfiguration also incurs a time and energy penalty. Table 4.7 shows reconfiguration time and energy for our different reconfigurable designs. These overheads can be amortised if the dataset size n between reconfigurations is large enough. Figure 4.15 shows the normalised processing times in the non-reconfigurable and reconfigurable designs over a range of dataset sizes. For large data sets, both reconfigurable designs approach total processing times of $5.49 ns/sample$ which is 45% less than the non-reconfigurable design. High configuration speeds are necessary to make the reconfigurable design competitive for intermediate dataset sizes. For small datasets, we can even observe a deterioration in the non-reconfigurable design caused by the parameter reload. Figure 4.16 shows the same comparison for normalised energy consumption. For energy, both high configuration speeds as well as low-power configuration are important to reduce the energy overhead. For large data sets, all reconfigurable designs approach an energy of $6.8 nJ/sample$. This is 5 times more efficient than the non-reconfigurable design with $34.3 nJ/sample$. The dataset sizes n where the reconfigurable designs become beneficial over the non-reconfigurable design is shown in table 4.6.

We observe that reconfiguration can improve both performance and energy efficiency, but the potential for energy reductions is higher than for performance improvements. For energy, reconfiguration becomes beneficial at smaller dataset sizes and the maximum possible improvement is also larger than for performance. This is because energy benefits from both faster processing as well as lower power consumption.

We now consider a scenario where the filter needs to be reconfigured between GSM frames. One GSM frame has a duration of $4.615 ms$ and produces 10,000 samples at the

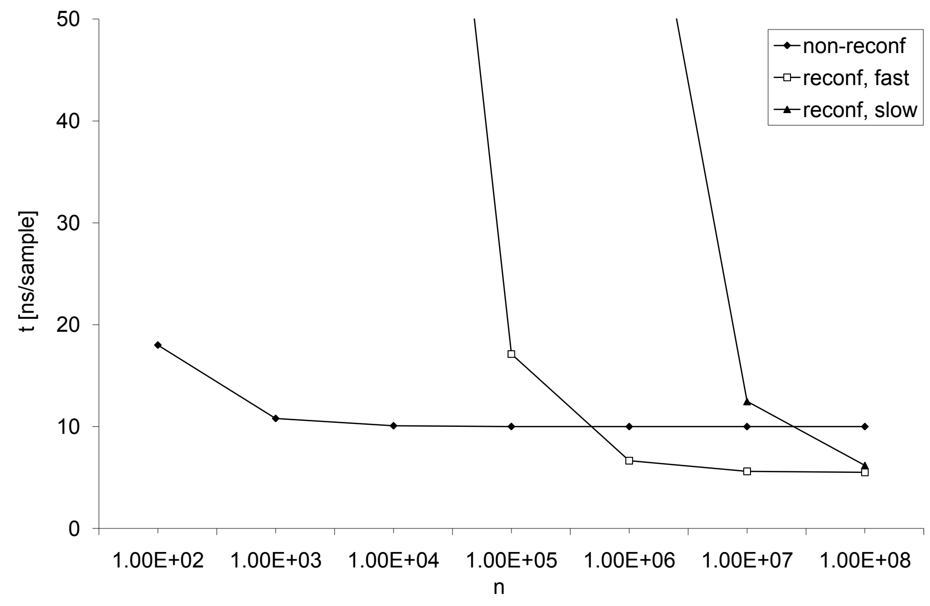


Figure 4.15: Total processing time over dataset size n for slow and fast configuration speeds.

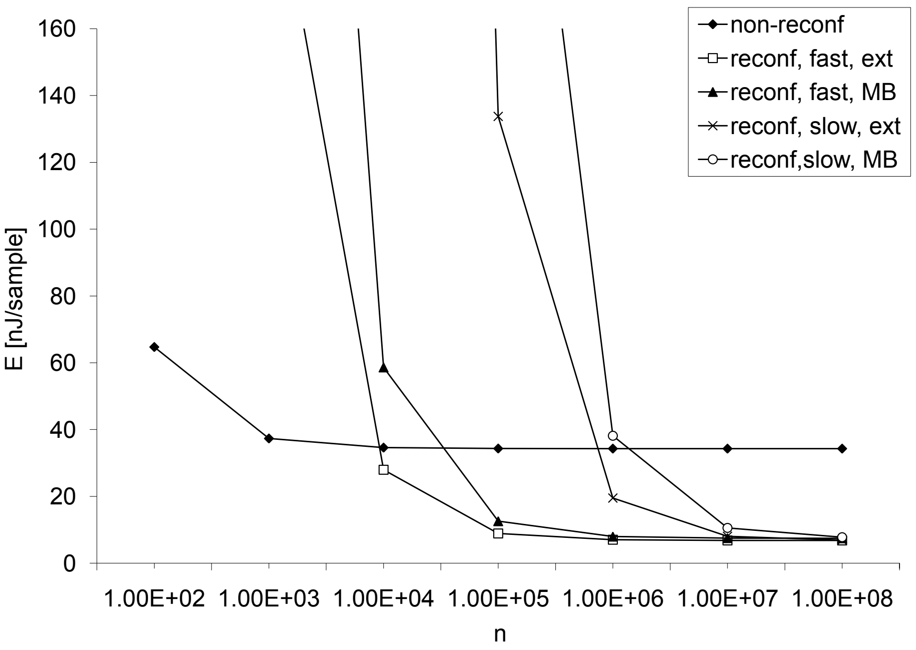


Figure 4.16: Energy consumption over dataset size n for MicroBlaze (MB) and external (ext) reconfiguration and with various configuration speeds.

sample rate of $2.167 Msamples/s$. Hence, the dataset size $n = 10,000$. Table 4.7 also shows the total time and energy to process 10,000 samples in our different filter designs. All reconfigurable designs are slower than the non-reconfigurable design. With fast reconfiguration however, designs have a total processing time of $1.217ms$ which is sufficient for the application requirement of $4.615ms$. Designs with an unoptimised, slow configuration controller are not competitive. Slow reconfiguration also leads to significant energy

config speed	n		
	performance	energy	
		MB config	ext config
$\phi_{config} = 5MB/s$	$1.55 \cdot 10^7$	$1.11 \cdot 10^6$	$4.61 \cdot 10^5$
$\phi_{config} = 300MB/s$	$2.57 \cdot 10^5$	$1.86 \cdot 10^4$	$7.67 \cdot 10^3$

Table 4.6: Dataset size n where the reconfigurable design becomes beneficial over non-reconfigurable design.

penalties. Even with fast reconfiguration, the MicroBlaze system causes an overhead that makes the design less efficient than the non-reconfigurable one. With external reconfiguration however, the reconfigurable system becomes 19% more energy efficient than the non-reconfigurable one.

Exploring parallelism

We now demonstrate how our design exploration method can be used to further optimise the design by changing the degree of parallelism in the implementation. We first consider performance, and carry out the optimisation steps shown in section 4.5.3.

As step 1, we determine the application parameters as $s = 80$ and $n = 10,000$. For Viretx-5 with fast reconfiguration, we obtain $\phi_{config} = 300MB/s$ and $\Theta_{LUT} = 36.9bytes/LUT$ (step 2). We use the reconfigurable design with external configuration as initial implementation. This design is fully parallel, i.e. $p = 80$. As step 3, t_p and A are obtained from table 4.7. The processing time per element is calculated as $t_{p,e} = 5.49ns$ using equation 4.10, the reconfiguration time as $t_r = 1.16ms$ using equation 4.2 and the reconfiguration time per element as $t_{r,e} = 14.5\mu s$ using equation 4.2 and 4.11 (step 4). With equation 4.15 we can calculate $p_{opt} = 17.4$ and choose $p = 20$ as a practical value(step 5). According to equation 4.13, we predict the total processing time for this design to be $50.9ns/sample$ (step 6). As step 7, we build the design for $p = 20$ and obtain its parameters. Table 4.8 shows circuit size, maximum clock rate, time and energy parameters for designs that are built with p ranging from 80 down to 5. This is for illustration purposes; not all of these designs would be built when using our method. A breakdown of the processing time components and circuit size is also illustrated in figure 4.17. For $p = 20$, we find that $t_{total,n} = 95.5ns/sample$ (step 8). This is worse than our calculated value and can be explained by a drop in clock frequency, i.e. the processing time per element $t_{p,e}$ is not constant but increases. The circuit size also does not scale exactly to p, leading to an increase in $t_{r,e}$. Both effects are caused by the fact that serialising requires multipliers

	LUTs	$f_{max}[MHz]$	$t_p[ns]$	$P_p[W]$	$E_p[nJ]$	$t_r[\mu s]$	$P_r[W]$	$E_r[\mu J]$	$t_{total}[\mu s]$	$E_{total}[\mu J]$
reconfigurable, MB, $\phi_{config} = 5MB/s$	12867 [1]	182 [2]	5.49	1.36	7.47	69755	0.44	30692	69810	30767
reconfigurable, MB, $\phi_{config} = 300MB/s$						1162		512	1217	586
reconfigurable, ext, $\phi_{config} = 5MB/s$	9452	182	5.49	1.23	6.79	69755	0.19	12695	69810	12763
reconfigurable, ext, $\phi_{config} = 300MB/s$						1162		211	1217	279
non-reconfigurable, 80 cycle coeff reload	26134	100	10	3.43	34.3	0.8 [3]	3.8 [3]	3 [3]	100	346

note 1: total size of design including MicroBlaze, only 9452 LUTs are reconfigured
note 2: f_{max} applies to FIR, not MicroBlaze
note 3: coefficient reload instead of reconfiguration

Table 4.7: Comparison of the non-reconfigurable and reconfigurable designs with internal MicroBlaze configuration controller (MB) and external configuration controller (ext) and with slow and fast configuration speeds. Total processing time and energy is shown for a dataset size of $n = 10,000$.

	VHDL design								CORE Generator design							
p	LUTs	f_{max} $[MHz]$	P_p $[mW]$	$t_{p,e}$ $[ns]$	$t_{r,e}$ $[\mu s]$	$E_{p,n}$ $[nJ]$	$E_{r,n}$ $[nJ]$	$t_{total,n}$ $[ns]$	$E_{total,n}$ $[nJ]$	LUTs	f_{max} $[MHz]$	P_p $[mW]$	$t_{p,e}$ $[ns]$	$t_{r,e}$ $[\mu s]$	$E_{p,n}$ $[nJ]$	$E_{r,n}$ $[nJ]$	$t_{total,n}$ $[ns]$	$E_{total,n}$ $[nJ]$
80	9452	182	1236	5.49	14.5	6.8	21.1	121.7	27.9	12039	303	1928	3.30	18.5	6.4	27.0	151.4	33.4
40	6278	145	908	6.89	19.3	12.5	14.0	91.0	26.5	6652	302	1276	3.31	20.4	8.5	14.8	88.4	23.3
20	5058	120	778	8.33	31.1	25.9	11.3	95.5	37.2	3321	304	774	3.29	20.4	10.2	7.4	54.0	17.6
10	3129	120	630	8.33	38.4	42.0	7.0	105.2	49.0	1674	306	550	3.27	20.5	14.4	3.7	46.7	18.1
5	1774	110	398	9.09	43.6	57.9	3.9	167.3	61.8	841	307	452	3.26	20.6	23.6	1.9	62.5	25.5

Table 4.8: VHDL and CORE Generator implementations of the reconfigurable FIR filter for various p and a dataset size of $n = 10,000$.

to share coefficients. Thus, there is less potential for optimisation of the circuit resulting in longer processing times and larger area. As step 9, we therefore refine our result. An increase in both $t_{p,e}$ and $t_{r,e}$ causes competing trends for p_{opt}. Since the effect of $t_{p,e}$ is stronger, we try a larger value for p first. This refinement is successful and with $p = 40$ we find the optimum. The processing time is $91.0ns/sample$ which is 25% less than the fully parallel design. The design is also 34% smaller. As step 10, we calculate the buffer size as $B = 26.7kB$ and configuration storage size as $\Psi = 232kB$. The design fits into the target device with 28.800 available LUTs.

We now reimplement the same reconfigurable filter with Xilinx CORE Generator. CORE Generator is a design environment for IP cores and is part of Xilinx ISE software [122]. Filters are implemented using distributed arithmetic and fixed coefficients. Parallelism cannot be directly specified in CORE Generator, but it can be implied by setting clock rates and data rates accordingly. We first build a fully parallel version of the design and perform our optimisation again from step 3. Table 4.8 shows that the fully parallel version requires more area than the comparable VHDL implementation but it is also faster. Again, table 4.8 lists the parameters for designs built with other values of p. We find that the design scales better than the previous VHDL implementation. The clock rate remains nearly constant and the circuit size scales proportionally to p. The details of time and area are illustrated in figure 4.21. From the fully parallel design, we calculate $t_{p,e} = 3.3ns$, $t_r = 1.48ms$ and $t_{r,e} = 18.5\mu s$. All other parameters are identical to the previous case. Using equation 4.15, we calculate $p_{opt} = 11.9$. A practical value for p is 10. For $p = 10$, the predicted processing time is $t_{total,n} = 44.9ns/sample$ and the actual time is $t_{total,n} = 46.7ns/sample$. This is indeed the optimal design, and it requires 69% less processing time and 86% less area than the fully parallel version. The buffer size is $B = 7.1kB$ and the configuration storage size is $\Psi = 62kB$.

Both the optimised VHDL and the CORE Generator design are still slower than the non-reconfigurable design for the given dataset size. They are however sufficient to meet the throughput requirement that is given for our application. In order to find the smallest design that can provide the required throughput of $2.167Msamples/s$ we use equation 4.20. Based on the parameters of the CORE Generator design we obtain $p_{min} = 0.57$ and $p_{max} = 249$. According to table 4.3, the fully serial design with $p = 1$ would be sufficient to provide the required throughput. In our experiment we find that CORE Generator cannot serialise the design further than $p = 5$. Hence, this is the smallest design possible. It has a size of 841 LUTs which is 93% smaller than the fully parallel design.

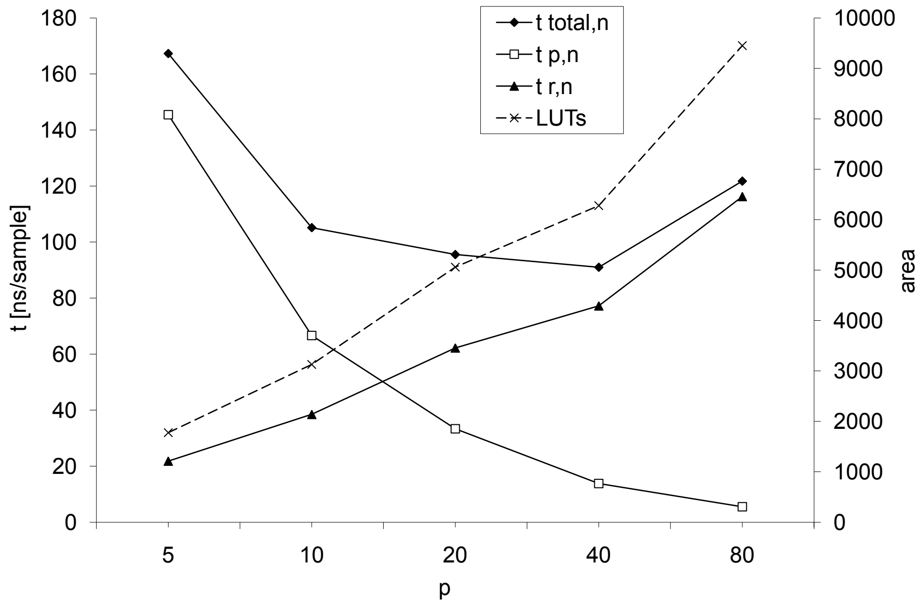


Figure 4.17: Total processing time and area for different levels of parallelism p in VHDL implementation.

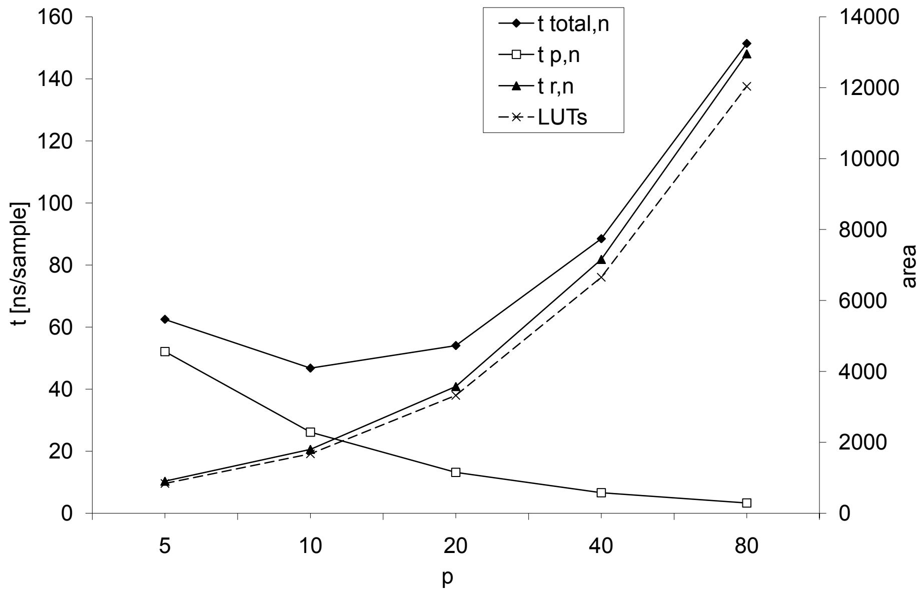


Figure 4.18: Total processing time and area for different levels of parallelism p in CORE Generator implementation.

The above analysis is based on a dataset size of $n = 10,000$. Figure 4.22 shows the processing time of the CORE Generator design when the dataset size varies. For a dataset size of 1000, a fully serial implementation is the best choice for a reconfigurable design. However, in our case this is slower than the non-reconfigurable design. Larger dataset sizes push the optimum to fully parallel implementations which are faster than the non-reconfigurable design.

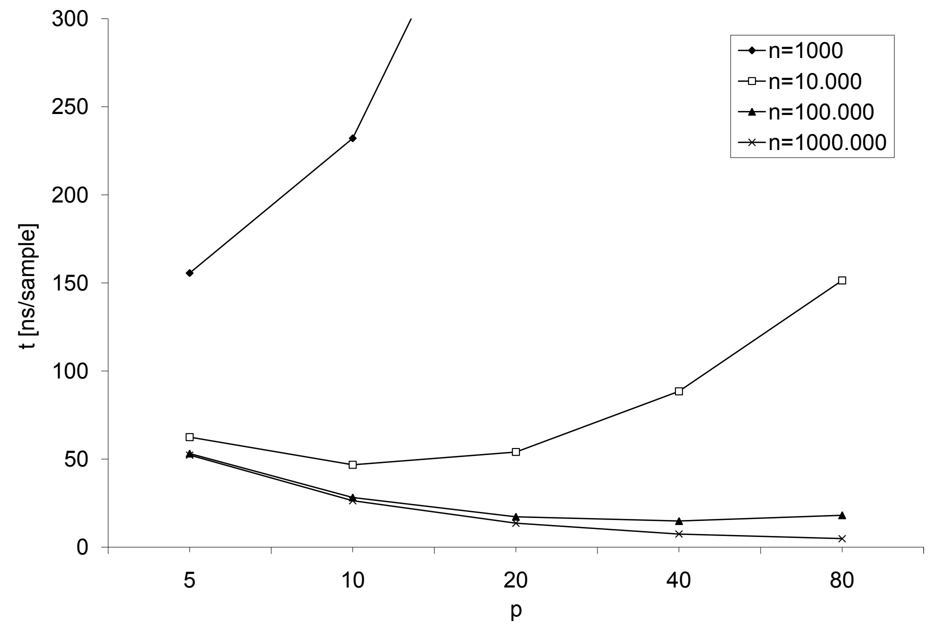


Figure 4.19: Total processing time for different levels of parallelism p and various data set sizes n in CORE Generator implementation.

We now optimise the VHDL design for energy. In section 4.8.1 is shown that the fully parallel reconfigurable version is 19% more energy efficient than the non-reconfigurable design. Now, the degree of parallelism is explored to further improve the design. Step 1 and 2 are the same as for performance optimisation. For step 3, we also need P_r and P_o. Reconfiguration power P_r is obtained from table 4.7 and the power overhead $P_o = 350mW$ is measured on a configured device with clock managers running but without any FIR logic. Step 4 is also identical to the one in performance optimisation. Using equation 4.33 we obtain $p_{opt} = 24.1$ and choose $p = 20$ as a practical value (step 5). A design is built for $p = 20$ and its parameters are measured (step 6). According to equation 4.31, $E_{total,n}$ is expected to be $17.8nJ/sample$ for this design but table 4.8 shows that the real design requires $37.2nJ/sample$. Again, this is worse than expected because processing time and design size do not scale as predicted. As step 8, we perform the same refinement as for performance and increase p. For $p = 40$ we do find an optimum. The design uses an energy of $26.5nJ/sample$ which is 5% less than the fully parallel design and 23% less than the original non-reconfigurable design. The design has the same point of optimality ($p = 40$) for energy as for performance and hence, area improvements, buffer size and storage size are identical to the performance-optimal design (step 9). Reconfiguration and processing energy as well as circuit size for this design is illustrated in figure 4.20.

The energy optimisation is now performed for the CORE Generator design. The power overhead for this design is measured as $P_o = 356mW$. All other parameters are the same

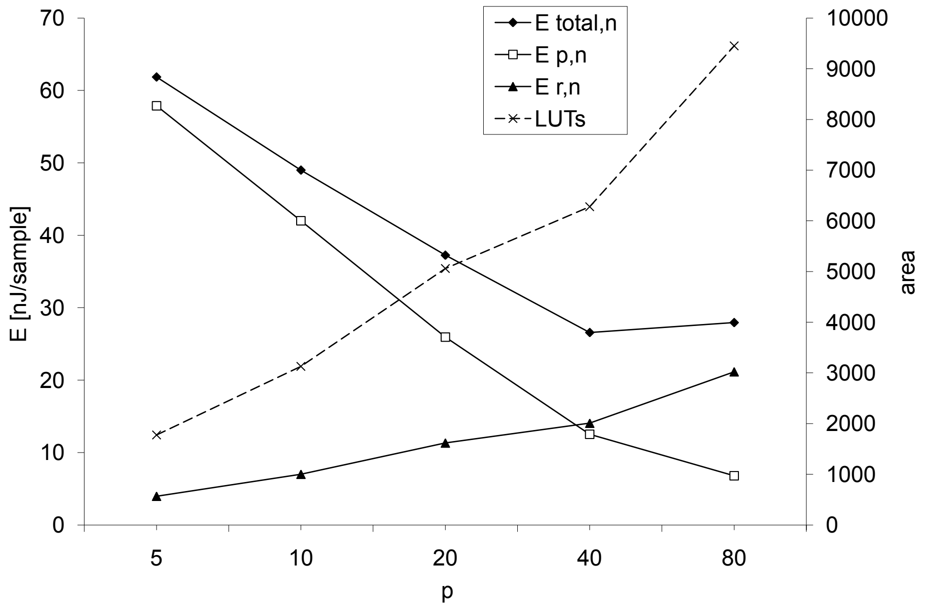


Figure 4.20: Energy efficiency and area for different levels of parallelism p in VHDL implementation.

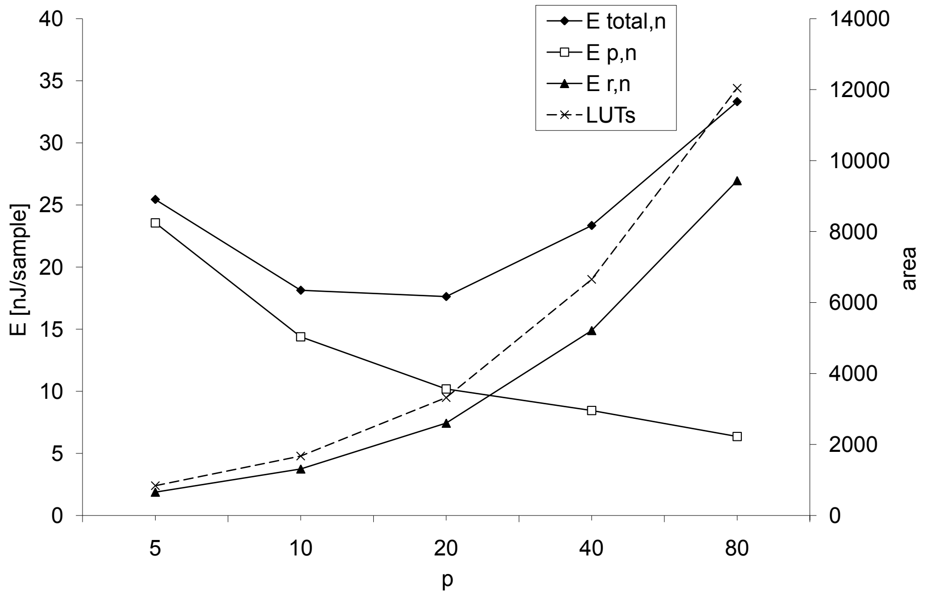


Figure 4.21: Energy efficiency and area for different levels of parallelism p in CORE Generator implementation.

as in the previous performance optimisation. Using equation 4.33, we obtain $p_{opt} = 16.7$. A practical value for p is 16. However, CORE Generator can only efficiently generate designs where parallelism is reduced by powers of 2. We therefore choose the closest practical value $p = 20$. Using equation 4.31 we calculate the energy for this design to be $16.6nJ/sample$. The real energy according to table 4.8 is $17.6nJ/sample$ which is close to what we expect. The design with $p = 20$ is indeed the optimal design and requires 47%

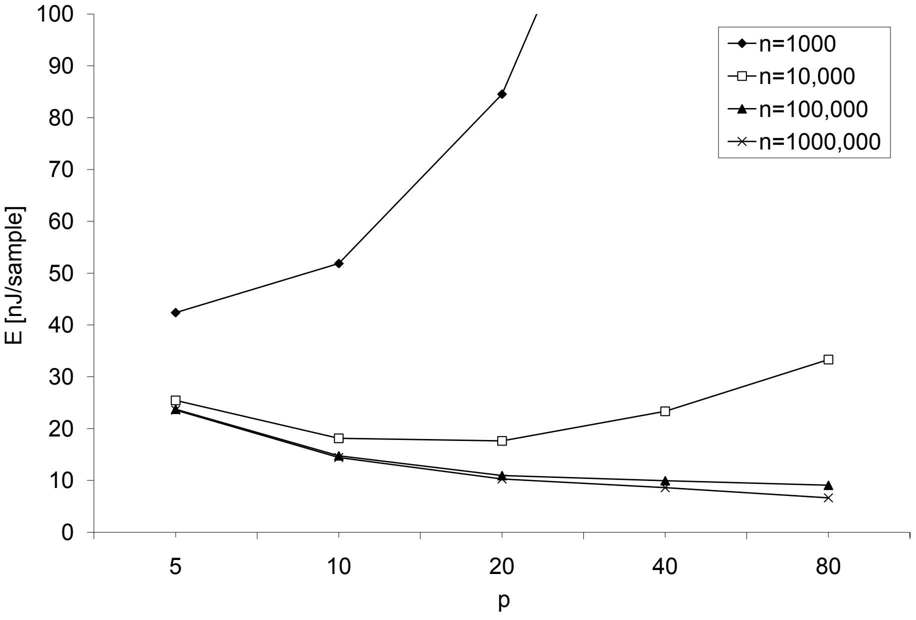


Figure 4.22: Energy efficiency for different levels of parallelism p and various data set sizes n in CORE Generator implementation.

less energy than the fully parallel CORE Generator design and 49% less energy than the original non-reconfigurable design. The design has a size of 3321 LUTs which is 72% less than the fully parallel CORE Generator version, and 87% less than the original design. The energy optimum is different from the performance optimum and for $p = 20$ the buffer size is $B = 14.2kB$ and the configuration storage size is $\Psi = 123kB$. Figure 4.21 illustrates how energy and area scales for the CORE Generator design. In the fully parallel version, the design consumes more energy than the comparable VHDL design, but the CORE Generator design scales better and becomes beneficial as p is reduced.

Figure 4.22 illustrates the efficiency of the CORE Generator design for various dataset sizes. Smaller datasets ($n = 1000$) shift the optimum towards a serial implementation, but in our case this version is 15% less energy efficient than the non-reconfigurable design. For larger datasets, the optimum is fully parallel. These designs are all more energy efficient than the non-reconfigurable design.

4.8.2 Cordic

The Coordinate Rotational Digital Computer (Cordic) algorithm is a hardware-efficient method to compute trigonometric functions [110]. In its basic mode, it calculates the rotation of a vector (x, y) by an angle z:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \cdot \begin{bmatrix} cos(z) & -sin(z) \\ cos(z) & sin(z) \end{bmatrix} \tag{4.38}$$

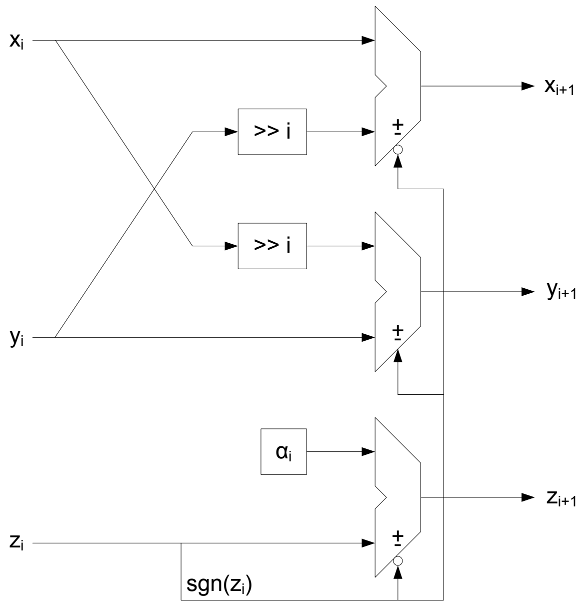


Figure 4.23: Structure of one iteration in the Cordic algorithm.

Cordic is an iterative algorithm that rotates the input vector by certain angles which result in divisions by powers of two. A hardware implementation of this method only requires bit-shifting and addition or subtraction. Figure 4.23 illustrates the basic processing step in Cordic in which the vector is rotated by an angel of α_i. This step is repeated several times with smaller and smaller angles until a sufficiently accurate result is approximated. It generally requires n processing steps to produce an n-bit accurate result. These steps can be computed on a sequential implementation that only employs one processing element as shown in figure 4.23. It is also possible to use a fully unrolled, pipelined design or intermediate versions.

The Cordic algorithm has been extended to compute hyperbolic and square root functions [112] and it can be efficiently implemented in FPGAs [9]. Cordic can also be used as an efficient operator for complex calculations in FFTs and digital filters [45]. The circuit size of a Cordic processor depends on several parameters, and it can be specialised to a certain mode (e.g. rotation only) or to fixed rotation angles [9]. Such a specialised circuit is smaller and more efficient, and it can be reconfigured if necessary.

In the following we consider a reconfigurable Cordic processor and analyse how the degree of parallelism can be used to optimise the design for reconfiguration between various dataset sizes n. Our design is specialised to perform vector rotations as shown in equation 4.38. We use 16-bit samples and the processor requires $s = 16$ processing steps to calculate one result. The design is coded in VHDL and implemented on the same Virtex-5 device that is used in section 4.8.1. Table 4.9 shows the parameters of the reconfigurable

p	LUTs	f_{max} $[MHz]$	P_p $[mW]$	$t_{p,e}$ $[ns]$	t_p $[ns]$	$t_{r,e}$ $[\mu s]$	t_r $[\mu s]$	E_p $[nJ]$	E_r $[uJ]$
16	842	326	540	3.06	3.06	6.47	103.56	1.65	18.85
8	484	324	445	3.08	6.16	7.44	59.53	2.74	10.83
4	280	337	427	2.97	11.86	8.61	34.44	5.07	6.27
2	164	330	390	3.03	24.20	10.09	20.17	9.43	3.67
1	111	295	339	3.38	54.14	13.65	13.65	18.37	2.48

Table 4.9: Parameters of the reconfigurable Cordic processor for various p.

Cordic processor when implemented in several versions ranging from fully pipelined to fully sequential. Reconfiguration times and reconfiguration energy are obtained with fast, external reconfiguration. The processing time per processing element $t_{p,e}$ remains fairly constant throughout all versions of the design, but reconfiguration time per processing element $t_{r,e}$ increases. This increase is caused by the extra area related to additional logic that is needed for sequential use of the processing elements.

We now analyse performance and energy for various dataset sizes n. Instead of performing the optimisation steps that are shown in section 4.5.3 and section 4.6.3, we simply compare real and modelled behaviour. Figure 4.24 illustrates the total processing time per sample over parallelism p. Shown is a comparison between the results of our real designs from table 4.9, and results that are calculated with our model. The modelled results are based on the fully parallel design with $p = 16$. In all cases, the modelled results are fairly close to the ones obtained from our real designs. The increase in reconfiguration time causes the real designs to be somewhat slower than predicted, but the optimal degree of parallelism is not shifted. For a dataset size of $n = 1000$, the optimal degree of parallelism is $p = 2$. For $n = 10,000$ it is $p = 8$, and for $n = 100,000$ the fully parallel design is the optimal solution. Figure 4.24 shows a comparison of energy efficiency in the real and modelled designs. Again, the model is derived from the fully parallel version. Similar to processing time, we see that the actual energy per sample is higher than predicted by the model, but the difference is relatively small. For $n = 1000$, the optimal degree of parallelism is $p = 4$ in the model as well as in the real design. For larger dataset sizes, a fully parallel design is the most energy efficient solution. In these cases, the difference between model and real design is also marginal.

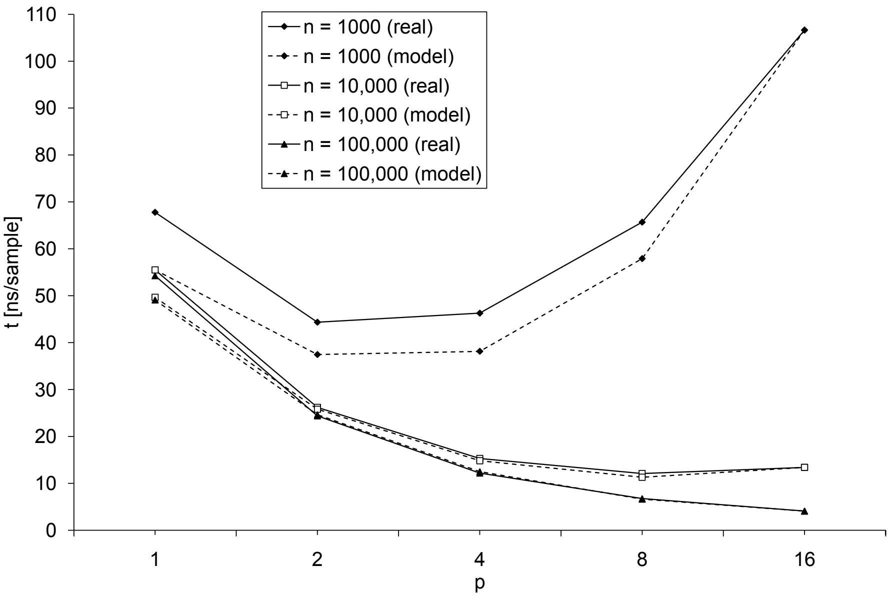


Figure 4.24: Total processing time for different levels of parallelism p and various dataset sizes n. Shown are results for the real designs, and results calculated with our model.

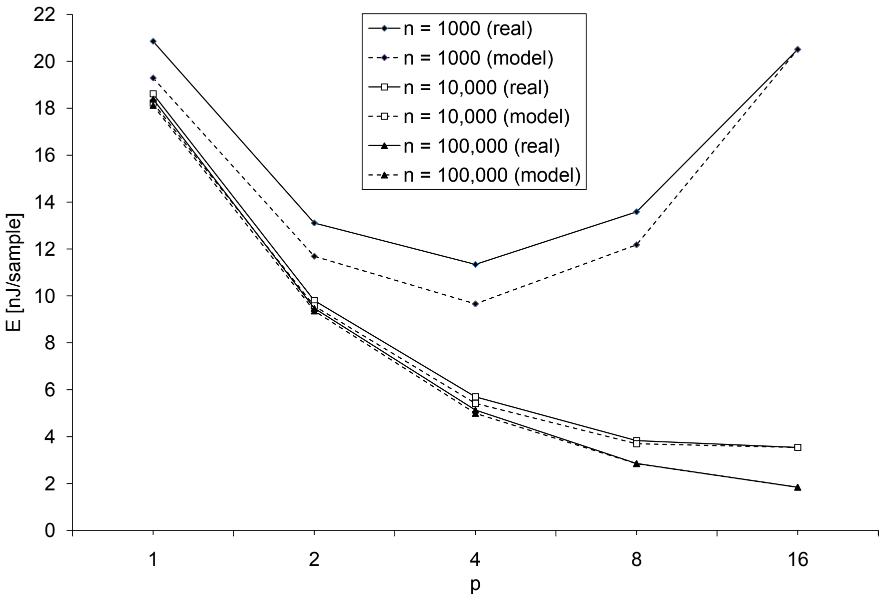


Figure 4.25: Energy efficiency for different levels of parallelism p and various dataset sizes n. Shown are results for the real designs, and results calculated with our model.

4.9 Summary

In this section, a parametric model is introduced that uses application, implementation and device parameters to evaluate and optimise storage, configuration time, buffering, performance and energy efficiency. Reconfiguration time overhead can have a significant impact on the performance of reconfigurable designs, and a traditional approach is to reduce the configuration time or process more data between reconfigurations. We propose a new approach that explores the degree of parallelism in the implementation, and show that this can improve performance and energy efficiency. Parallelism speeds up processing, but it also leads to larger area requirements which causes slower reconfiguration. Likewise, parallelism reduces the impact of constant power overheads but it increases the energy required by reconfiguration. In both cases, we can optimise designs by balancing these competing aspects. We also propose a method for fast design space exploration: one design is built to derive the required implementation parameters and we then use our equations to calculate the optimal degree of parallelism. Some designs might not scale exactly as expected, and the real optimum might be different from the calculated one. For such cases, we analyse the effect of parameter variations on p and give guidelines how these trends can be used to refine the design.

Our case study shows how a reconfigurable FIR filter can be optimised for performance and energy using our method. Comparing non-reconfigurable and reconfigurable versions of the design, we observe that reconfigurable designs are beneficial when processing large datasets between reconfigurations but high configuration speeds are necessary to achieve a benefit for intermediate dataset sizes. The speed of the current standard solution is insufficient, but a recently developed configuration mechanism that is 60 times faster allows to achieve such a benefit. For energy efficiency, it is also important to use a low-power configuration mechanism. In our experiment, we observe that reconfiguration can improve both performance and energy, but the potential for energy improvements is higher because energy benefits from two improvements in a specialised reconfigurable design: faster processing and lower power consumption during processing.

We explore the degree of parallelism to further improve performance and energy in the reconfigurable designs. A VHDL implementation of our reconfigurable FIR filter exhibits parameter variations that influence the optimality of the design. The result needs to be refined, but we are still able to identify the optimal degree of parallelism for performance and energy. A CORE Generator version of the design scales according our model, and our optimisation leads directly to performance- and energy-optimal designs with intermediate degrees of parallelism. In a second case study, we present a reconfigurable Cordic processor that can also be optimised for performance and energy by exploring parallelism. In a comparison of modelled and real behaviour, we find that the parameters of real design are close to the predicted values. The predicted optimal values for p are correct and do not require refinement.

Chapter 5

Bitstream relocation

In a reconfigurable system, modules are represented through partial bitstreams that are used to configure part of the FPGA. Partial bitstreams that are created with Xilinx implementation tools are location specific, i.e. they are targeted to one particular partially reconfigurable region (*PR region*) [131]. However, it is not necessary to restrict a physically implemented module to one location. FPGAs provide a regular fabric, and a circuit could therefore be used in multiple locations.

The advantage of being able to relocate a bitstream is reduced design time as well as reduced memory requirements. Without relocation, for m modules usable in k PR regions, $m \cdot k$ partial bitstreams have to be produced and stored for run-time reconfiguration. With bitstream relocation, one version of the module can be instantiated in all locations, therefore reducing the total number of required bitstreams to m. This significantly reduces the size of memory required to store the partial bitstreams. With full bitstreams containing up to 50 Mbits in Virtex-4 [127] and up to 80 Mbits in Virtex-5 [118] partial bitstreams can have sizes of up to tens of MBits. It is therefore important to keep the number of partial bitstreams as low as possible to reduce cost of the storage, especially since costly fast memories should be used to minimise impact on performance. Another aspect is reduced design time. The generation of bitstreams requires physical implementation including time-consuming placement and routing. Reducing the number of partial bitstreams by a factor of k can considerably shorten the design cycle.

As shown in section 2.8, there is extensive work that either assumes relocatable modules or provides mechanisms for relocation. A fundamental precondition for relocation is that two PR regions have to provide the same underlying fabric in order to allow relocation between the two. Hence, relocation is trivial in fully homogeneous devices as circuits can be allocated to any free part of the logic fabric. Most current FPGAs are heterogeneous and do not allow this simple technique of relocation. As a solution, one

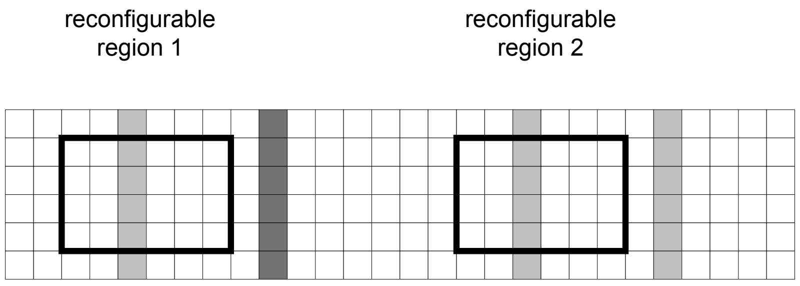


Figure 5.1: Relocation between identical PR regions is limited in heterogeneous architectures.

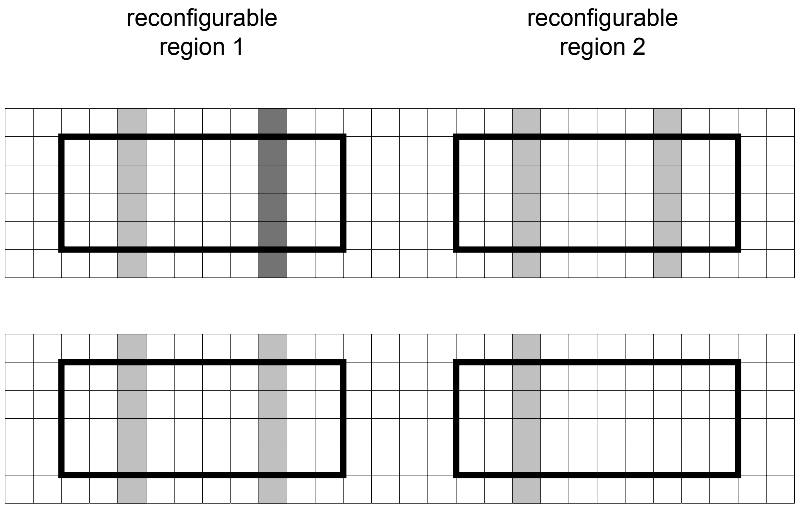


Figure 5.2: Relocation between non-identical PR regions increases flexibility.

can identify PR regions that provide an identical footprint, with heterogeneous resources in the same relative location. This however restricts the placement options for relocatable PR regions as symmetry has to be found. This approach is used in several reconfigurable applications [91, 41, 89]. These applications target devices that have only one type of heterogeneous resource, that is BRAM columns. However, FPGAs are becoming increasingly heterogeneous with multiple types of dedicated resources spread throughout the logic fabric. For example, Virtex-4 and Virtex-5 FPGAs provide BRAM and DSP columns, and they also contain an additional IO and clock column in the middle. This further limits relocatability, as it becomes more difficult to find regions with an identical footprint. This is illustrated in figure 5.1. Irregularities in the pattern of heterogeneous resources limit the number and size of PR regions that can be found. Thus, relocatability in reconfigurable applications is either limited [56, 54] or not used at all [12, 32].

We propose a more flexible approach that allows relocation between regions that are not fully identical as illustrated in figure 5.2. This is motivated by the observation that it is often difficult to identify regions that are fully identical, but it is easy to find regions that are almost identical. Our concept that is presented in section 5.1 is based on identifying regions that are almost identical and using the compatible subset of resources between these regions. Non-matching resources are masked out. We introduce a model with functional layer and configuration memory layer to determine the requirements for relocation and to illustrate our concept.

In section 5.2 we show how our approach can be realised on Virtex-4 FPGAs. We illustrate the compatible subset between three different types of resources and explain the transformations that are necessary for non-matching resources. We also develop a lightweight software implementation that allows to include our approach into a standard configuration controller based on Xilinx embedded processors. The implementation is specific to Virtex-4 but our method can also be adapted to Virtex-5 and Spartan-3 FPGAs and to older architectures such as Virtex-II and Virtex-II Pro.

In the following, we consider the problem of placing PR regions that support our relocation technique. Section 5.3 introduces a formalism and technique to automate the placement of such PR regions. With this method, we aim to reduce the number of non-matching resources when placing a number of PR region on the FPGA. We also consider additional constraints such as communication infrastructures or fragmentation of the residual free space.

Section 5.4 presents a case study that demonstrates and evaluates our flexible bitstream relocation approach in a software-defined radio prototype. We also provide an abstract analysis of relocatability that shows the number of alternative placement options for various regions sizes when allowing for a certain percentage of masked-out resources. Finally, we evaluate placement approach based on various signal processing functions.

In section 5.5 we provide a summary of this chapter and highlight our results.

5.1 Flexible bitstream relocation

In this section, a device independent model is introduced that illustrates the preconditions for relocation in general, as well as the concept of using PR regions that are not fully identical. We also present a design flow for systems that use non-identical regions for relocation, and explain how relocation can be performed by a configuration controller at run time.

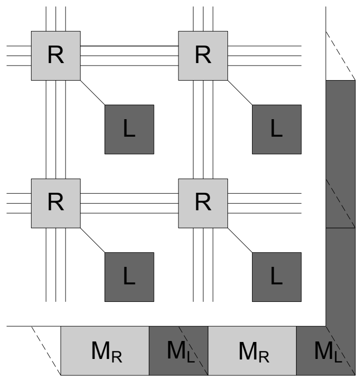


Figure 5.3: FPGA model with functional layer (top) and configuration memory layer (bottom).

5.1.1 Model for relocation

We model the FPGA as an architecture with two layers as illustrated in figure 5.3:

- The functional layer: it contains logic resources L and routing resources R.
- The configuration memory layer: an underlying layer of memory pages M_L and M_R that control the configuration of logic and routing resources of the functional layer.

The configuration memory is characterised by an allocation of configuration data to functional units. Logic resources L and routing resources R are mapped to memory pages M_L and M_R. In a regular FPGA structure, this mapping of resources to memory is uniform for all tiles i.e. configuration data from one specific memory location M_L can be used to configure any resource L on the device. This is the first precondition for bitstream relocation. The second precondition is that the pattern of resources on the functional layer is identical for two PR regions.

First we focus on the functional layer. We consider a fabric of logic resource A which is interrupted by heterogeneous resources B and C. Adjacent to these logic resources are routing blocks R. In FPGAs routing resources are usually uniform i.e. they provide identical connectivity to neighbouring tiles.

Figure 5.4 illustrates an example of two PR regions that are not fully identical, but contain a compatible subset of resources. Region 1 contains heterogeneous resources of type B whereas region 2 provides resources of type C. To enable relocation between these two regions, we identify an identical subset of resources. Identical in both regions are the two outer columns of resource A as well as all routing resources R. Even though logic resources B and C cannot be used, their associated routing resources are still available

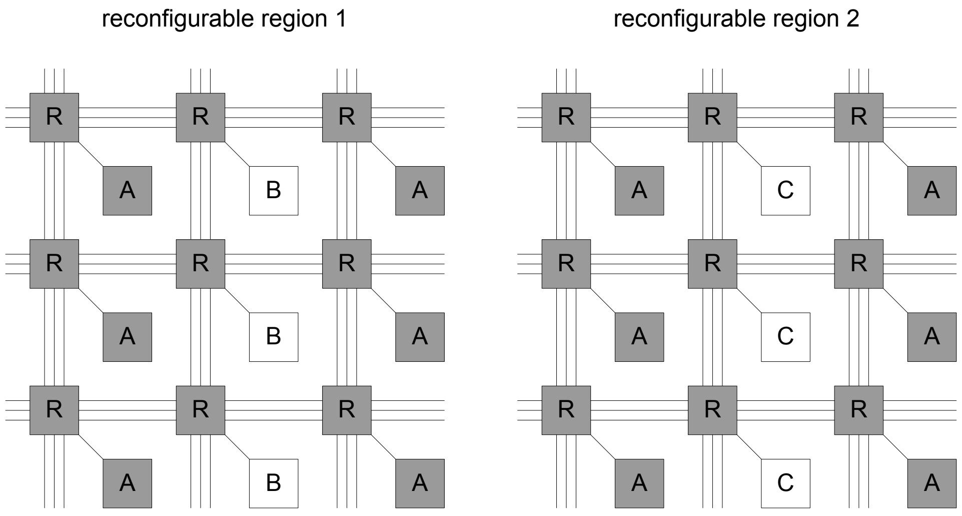


Figure 5.4: Two PR regions with compatible subset of resources in a heterogeneous device.

for connectivity of neighbouring tiles. The compatible subset of resources is highlighted in grey, and it represents an identical functional layer that can be used for relocation between these regions.

Configuration data corresponding to this subset can be considered as relocatable between the two non-identical regions as illustrated in figure 5.5. This configuration can support a functioning circuit in both region 1 and region 2.

5.1.2 Design flow

In order to create relocatable configuration data the designer has to ensure that only the compatible subset of resources is used by the implementation tools. We assume that the design is already partitioned into a static part and a number of reconfigurable modules. The design flow consists of the following steps:

1. Floorplanning: PR regions are placed on the FPGA. If fully identical regions cannot be found the designer tries to minimise the number of mismatching resources. Other floorplanning considerations can include the connectivity of regions, device fragmentation and the location of static logic. This aspect is further explained in section 5.3

2. Bus macro placement: relocatable modules need a unified communication interface for all regions. Hence, bus macros have to be placed in the same location for all PR regions.

3. Selection of the module implementation region: all modules are instantiated in one

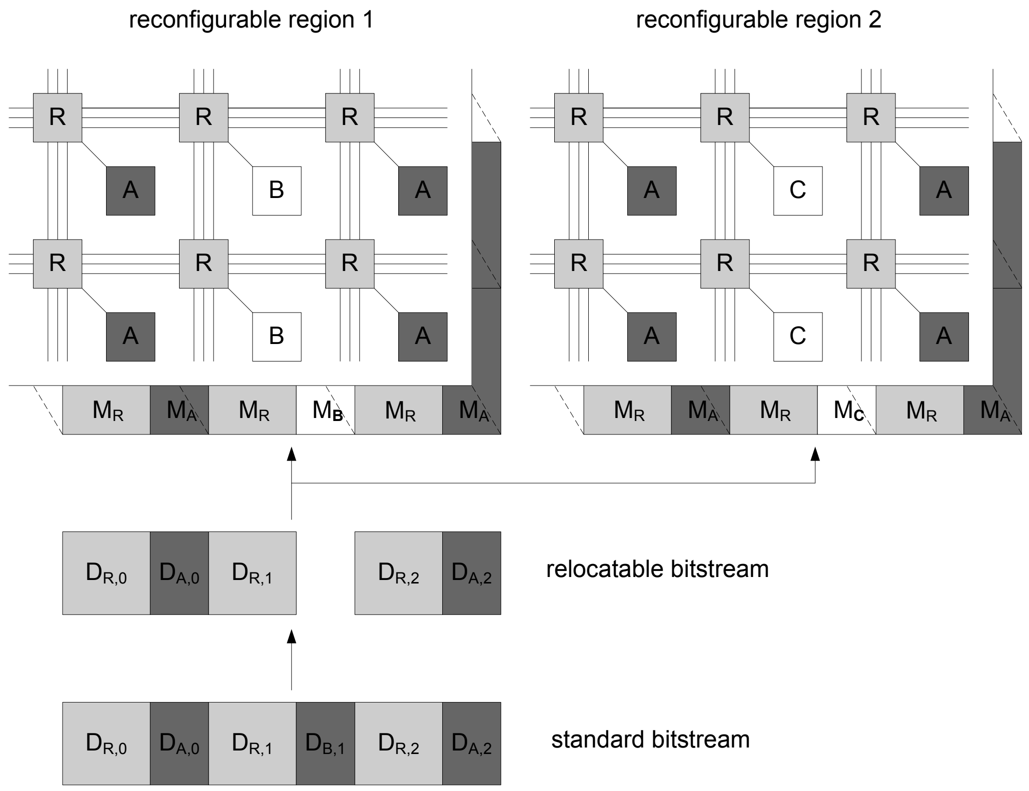


Figure 5.5: Model of a relocatable bitstream: configuration data corresponding to the compatible subset can be loaded in both locations.

PR region only. This region is used for physical implementation.

4. Constraint setup: area constraints are created for all PR regions to keep them free from static logic. Additionally, the designer has to set up implementation constraints that instruct the placement program not to place any logic instances into mismatching logic resources. If, for instance, in figure 5.4 region 1 is chosen as module implementation region, then the placement program is not allowed to place logic into resources B. The router, though, is allowed to use all routing resources. All other constraints such as timing requirements and IO locations are added.

5. Physical implementation: the design tools translate, map, place and route the static design and all reconfigurable modules on the FPGA according to the design constraints.

6. Configuration generation: a full configuration file is generated for initial configuration including the static system. Additionally, one partial configuration file is generated for each module.

As shown in figure 5.5, relocatable configuration data can be generated by creating a circuit in region 1 that does not use resource B. The corresponding standard bitstream will still contain configuration data $D_{B,1}$ for resource B, but it can be transformed into a relocatable bitstream by removing $D_{B,1}$. The resulting configuration data can be used in both regions.

5.1.3 Performing relocation at run time

The relocation of a bitstream can be performed at design time by a tool similar to PARBIT [44]. The disadvantage of this, though, is that separate versions of the bitstream still have to be stored. Hence, relocation is usually performed at run time. In this case, relocation becomes part of the configuration controller's functionality. Relocation may be performed by simply changing the target address but it may also involve the modification of configuration data.

If the memory configuration architecture of all heterogeneous resources is identical, then a relocatable bitstream can be created by inserting a zero-configuration for $D_{B,1}$ that would leave resource B or C in their default configuration. Such a bitstream can be relocated by simply changing the target address.

However, the configuration architecture may differ for different types of heterogeneous resources. For example, resources B and C might require different amounts of configuration data or have different default configurations. In such a case, the bitstream has to be modified at run-time to accommodate these differences. The configuration controller can skip over memory pages of non-matching resources. As an alternative, the controller can dynamically insert the appropriate zero-configuration for the non-matching resource. In order to perform such modifications at run time, the controller needs information about the device architecture. Based on a device map, original address and target address, it is then possible to identify non-matching resources and to perform the necessary modifications.

5.2 Implementation on Virtex-4 FPGAs

In this section we describe how our method can be applied to existing FPGA architectures. We describe the configuration architecture for Virtex-4 FPGAs [127] which support run-time reconfiguration and feature a two-dimensional configuration granularity. We explain how our relocation technique works on Virtex-4 devices and develop a software implementation by extending current HwICAP drivers. However, our method is not limited to Virtex-4 and can also be applied to other devices.

5.2.1 Virtex-4 configuration architecture

Figure 5.6 shows the structure of a Virtex-4 XC4VLX25 FPGA. The device consists of a regular CLB grid which is interrupted by columns of specialised resources. These specialised columns can contain BRAMs, DSPs and IOs. A notable difference to previous Virtex-II and Virtex-II Pro devices, which have combined columns of BRAMs and multipliers, is the separation of BRAMs and DSPs in Virtex-4. Another change in the architecture is the presence of one IO column in the middle of the device. The overall heterogeneity has therefore increased compared to previous Virtex families. Virtex-4 FPGAs also contain a central clock column that controls the functionality of global clock resources such as clock managers and clock buffers. The clock column differs from other heterogeneous resources in that it is present as a column in the configuration memory but it does not interrupt the functional layer.

The configuration memory is organised in 16-CLB high rows that can be reconfigured independently. This differs from previous Virtex families where full columns have to be reconfigured. In Virtex-4, rows can be reconfigured without disturbing any logic above or below, thus enabling much more flexible floorplans than one-dimensional architectures. Figure 5.6 shows a possible floorplan with four reconfigurable modules. Multiple reconfigurable modules as well as static logic can coexist in the same device column. The horizontal granularity is 1 CLB, i.e. each CLB column can be reconfigured independently from its neighbouring columns. Figure 5.6 also illustrates basic aspects of the FPGA's addressing scheme. The device is addressed by top or bottom half, by rows from the centre to the outside and by columns from left to right.

In Virtex-4 FPGAs, the configuration memory is organised in vertical *frames* that have a fixed length of 1312 bits. A frame represents the smallest unit of reconfiguration. Multiple frames are used to configure a column of CLBs, or a column of heterogeneous resource. Figure 5.7 shows the details for a CLB column. Such a column contains logic for local clock distribution in the middle, with each 8 CLBs above and below. This entire column is configured by 22 frames. Each frame consists of 41 32-bit words. The middle word contains configuration data for the local clock logic, and the remaining 40 words (each 20 above and below) configure the CLBs. This organisation is the same for all 22 frames.

Configuration frames are randomly addressable and are addressed by a 32-bit word that is composed of five values. The details of this address field are explained in table 5.1. The top bit is used to indicate whether a frame is located in the top-half or the bottom-

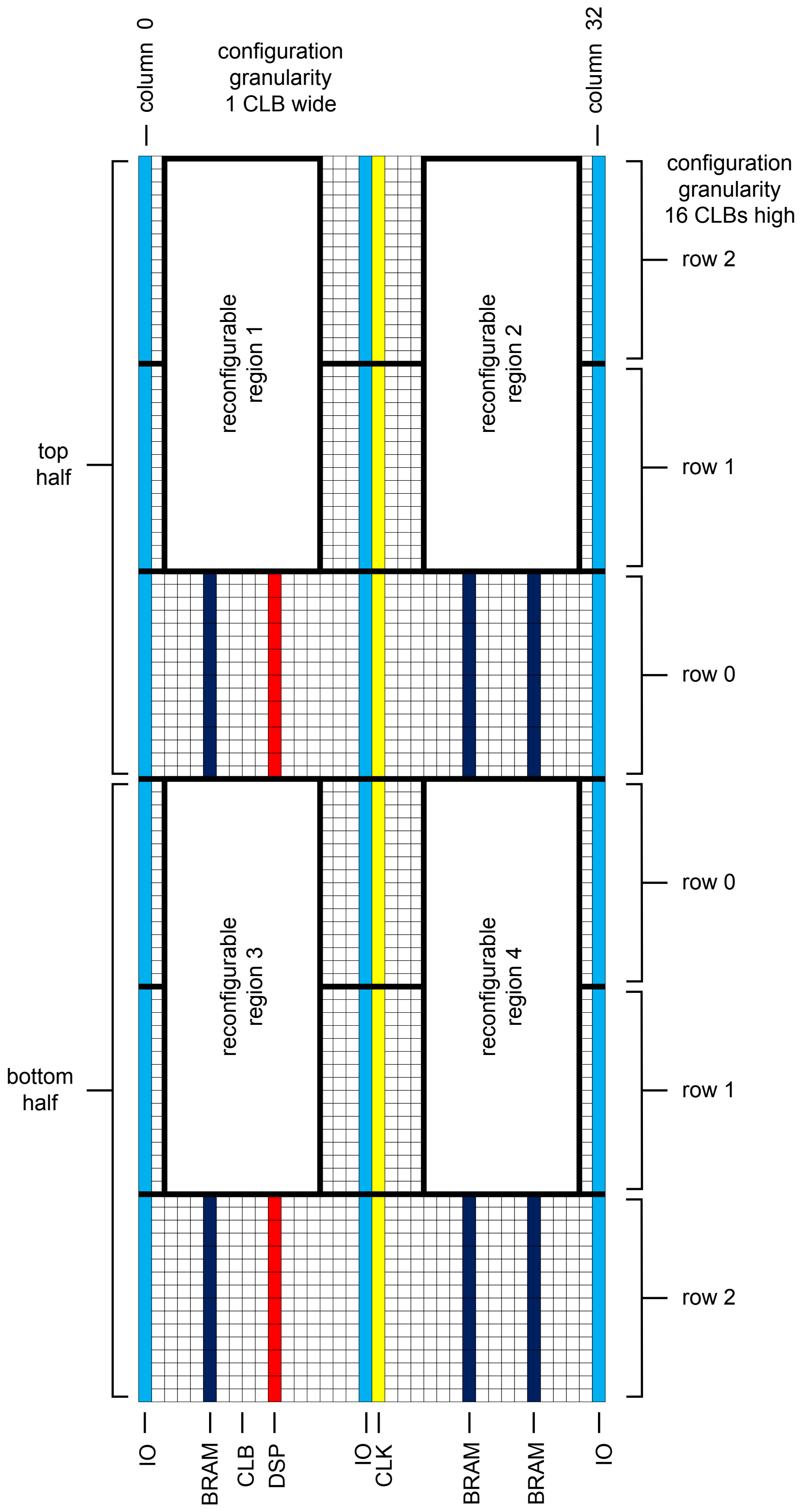


Figure 5.6: Structure of a Virtex4 XC4VLX25 FPGA with an example floorplan of four reconfigurable modules. The addressing scheme with half, row and column is also indicated.

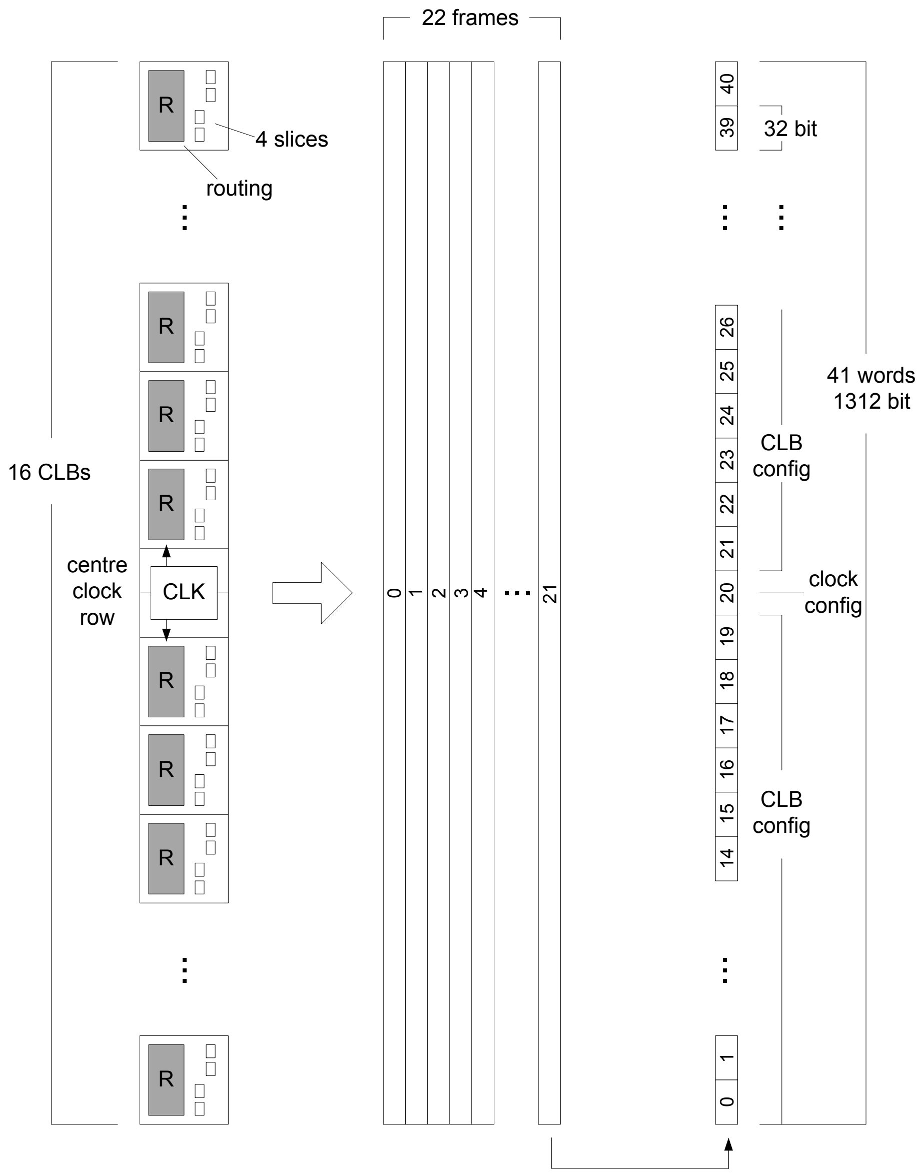


Figure 5.7: Organisation of configuration memory in Virtex-4 FPGAs for one CLB column.

half of the device. The block type address field can have three different values: Block type 0 contains all frames for CLB, DSP, IO and clock configuration. Block type 1 contains BRAM interconnect configuration frames. Block type 2 specifies frames with BRAM data content. The row address specifies the row within one half of the FPGA and is incremented from the centre of the device. Within each row resources are organised in columns. A column has a height of 16 CLBs, 8 DSPs or 4 BRAMs. The column address is incremented from the left side of the FPGA. Since BRAM interconnect and data frames have a different block type, they are addressed independently from all other resources. Finally, the minor address is used to address frames within one column.

address bits	type	description
31:23	unused	
22	top bit	selects between top-half and bottom half of device, top = 0, bottom = 1
21:19	block type	selects block type, CLB, DSP, IO, CLK = 000 BRAM interconnect = 001, BRAM data = 010
18:14	row address	selects row of frames, rows are 16 CLBs high, address increases from centre of device
16:6	column address	selects device column, columns are counted from the left and separately for each block type
5:0	minor address	selects frame number within one column

Table 5.1: Frame addressing in Virtex-4 FPGAs.

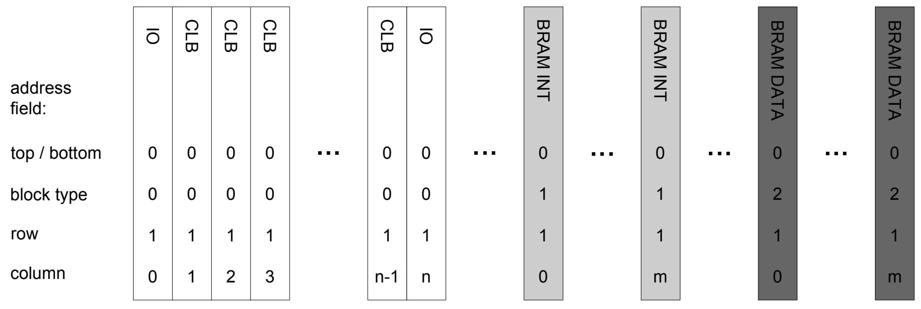


Figure 5.8: Configuration memory map of Virtex-4 FPGAs.

Figure 5.8 illustrates the memory map of the FPGA for one full row, not showing minor frame addresses. Resources that are located next to each other on the functional layer are also adjacent in the memory space with the exception of BRAMs. BRAM interconnect and data have their own block types and are therefore located at the end of the column.

Each column in the configuration memory is configured by multiple frames as illustrated in figure 5.9. CLB columns are configured by 22 frames where the first 20 frames control routing resources and the last two frames configure the logic resources of the CLB column. A DSP column is configured by 21 frames. Again the routing resources are configured by the first 20 frames. The last frame contains the configuration of the DSP. BRAM configuration data consist of two separate block types. The first block contains 20 frames of routing information. A second block contains 64 frames of BRAM data. CLB, DSP and BRAM columns contain an equal number of routing frames, and routing frames can be exchanged between these types of resources. Hence, the first 20 frames of each column correspond to routing configuration memory M_R that is part of the compatible

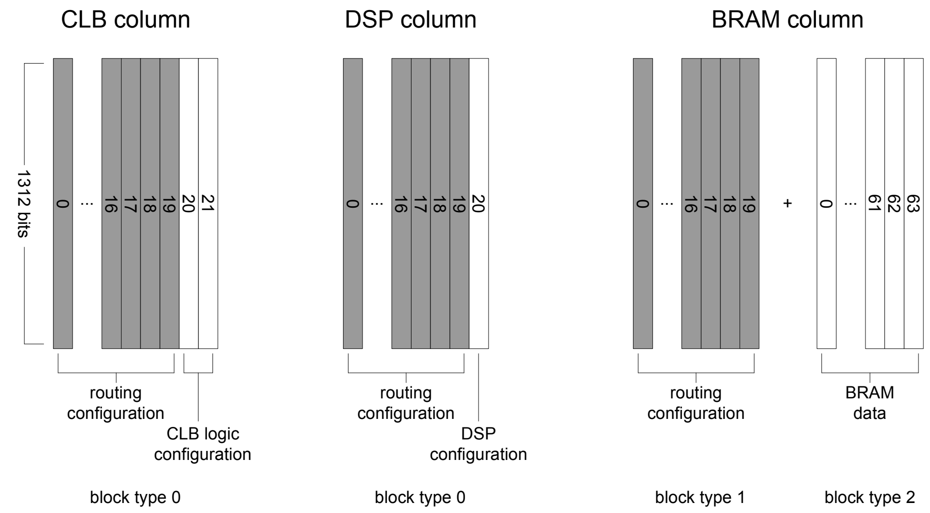


Figure 5.9: Configuration memory of CLB, DSP and BRAM resources.

subset.

Frames in the bottom half of the device are bit reversed compared to frames in the top half, with exception of the middle word of the frame. Thus, the mapping of memory to resources is not uniform as required in section 5.1.1. However, this can be easily addressed by reversing the bits when relocating between top and bottom half.

Configuration frames are randomly addressable but during a normal configuration process, frames are usually auto-incremented throughout the FPGA's address space. A bitstream contains a start address and frames are sequentially written throughout all frames, columns, rows, block types and both halves. A partial bitstream, that does not cover the full device, usually has multiple start addresses that are auto-incremented from to cover the according PR regions. When performing relocation over changing block types, the irregularities in the configuration memory map (see figure 5.8) mean that sections of auto-incremented frames may have to be split and addressed separately. We will explain this in detail in the following section.

5.2.2 Bitstream relocation in Virtex-4

We now describe how our method of configuration relocation can be applied to Virtex-4 devices. However, it is also possible to use the same method on other heterogeneous FPGAs. An adaptation to Virtex-5 is straightforward as it provides a very similar configuration architecture. Frames cover rows that are 20 CLBs high and a different number frames is used by each resource type, but the principle is otherwise the same. It is also possible to apply the technique to Spartan-3 or older Virtex-II and Virtex-II Pro FPGA families. As explained in section 2.3, these devices have configuration frames that cover the full height of the device, resulting in one-dimensional layouts. This has implications on floorplanning but our concept of using configuration frames that represent a compatible subset of resources can also be used.

Because of the columnar structure of the configuration memory, we have to consider vertical and horizontal relocation separately. In a system as illustrated in figure 5.6, no relocation would be possible with prior-art techniques that solely rely on changing target addresses. A vertical relocation from region 1 to 3 provides identical resources on the functional layer. Thus, the same circuit can be used in both regions without restrictions. The mapping of memory to functional layer, though, is not identical because of the bit-reversal between top and bottom half. For a horizontal relocation between regions 1 and 2, one encounters non-matching resources on the functional layer.

The solution for vertical relocation between top and bottom half is simple: the frame is bit reversed with the exception of the middle word for local clock configuration. This section of the frame is in the same order on both halves. Vertical relocation has the advantage of providing identical resources. On the other hand, relocatability is limited to steps of 16 CLBs.

For horizontal relocation we have to consider the heterogeneity of the device. Depending on the size of the module and other floorplanning constraints it can be impossible to find completely identical PR regions. Regions 1 and 2 in figure 5.6 exhibit a mismatch between a DSP column in region 1 and a BRAM column in region 2. However, both columns provide the same routing resources to their neighbouring CLB columns. These routing resources are configured by the same 20 frames of configuration data. Therefore, the configuration can be relocated by copying the first 20 frames of the DSP column into the memory location of the according BRAM interconnect columns. The last frame, containing DSP configuration data, is discarded. All other columns can be copied by address offsetting. For the reverse transformation from region 2 to 1, 20 frames of BRAM interconnect data are copied into the according memory location of the DSP column. The last frame (number 21) can be omitted or filled with a pad frame if address auto-increment is used. A pad-frame is a zero-configuration that is valid for all resources. The same operations apply to transformations between CLB and DSP and between CLB and BRAM. In any case, the first 20 frames of routing configuration are copied; DSP or CLB logic configuration frames can be filled with one or two pad frames. In comparison to vertical relocation, horizontal relocation allows much finer placement steps of 1 CLB. However, it is preferable to choose a placement that reduces the number of non-matching columns. Note that transfers from or to non-matching BRAM do not have to include configuration data of block type 2. The BRAM is unused and its data content irrelevant.

In order to enable the horizontal relocation between two or more not fully identical PR regions, all mismatching resources have to be prohibited from being used by the implementation tools. The Xilinx place and route tool can be directed by PROHIBIT constraints to not place logic instances into certain resources. The router, though, is still allowed to use the corresponding routing resources. If in our current example a module is physically implemented in region 1 of figure 5.6 then PROHIBIT constraints have to be applied to all DSPs in this region. The resulting module can be used in all four PR regions. For relocation from region 1 to 4, both frame reversal and the DSP to BRAM frame transformation are applied. It should be noted that we do not need to prohibit all heterogeneous resources in relocatable modules. In our example a module can make use of the left BRAM column if required. Only non-matching resources are prohibited.

5.2.3 Software implementation

This section presents a software implementation of our method for FPGA self-reconfiguration with an internal processor. However, self-reconfiguration is not strictly necessary and our software solution could be adapted to run on an external processor. Instead of generating pre-parsed bitstreams which lack configuration data for mismatching resources as indicated in figure 5.5, we choose to transform regular partial bitstreams produced by the implementation tools. A bitstream is parsed and modified at run time according to the requirements of the target region. We choose a software implementation because configuration controllers are mainly realised as processor-based systems.

We have implemented our technique as a software driver for the OPB HwICAP core [119]. As explained in section 2.4, HwICAP is an IP core provided by Xilinx as part of the Embedded Development Kit, and it allows the fast and easy creation of a processor-based system for FPGA self-reconfiguration. The hardware- and software-structure of a HwICAP-based reconfigurable system is illustrated in figure 2.8. Our software implementation extends the functionality of existing HwICAP device drivers as shown in figure 5.10. The standard high-level driver provides a function `SetConfiguration()` to load partial bitstreams into ICAP. The high-level driver invokes the low-level functions `StorageBufferWrite()` and `DeviceWrite()` to transfer configuration data to the local HwICAP buffer, and to write the data to the device. Our device driver builds on the same low-level functions and pro-

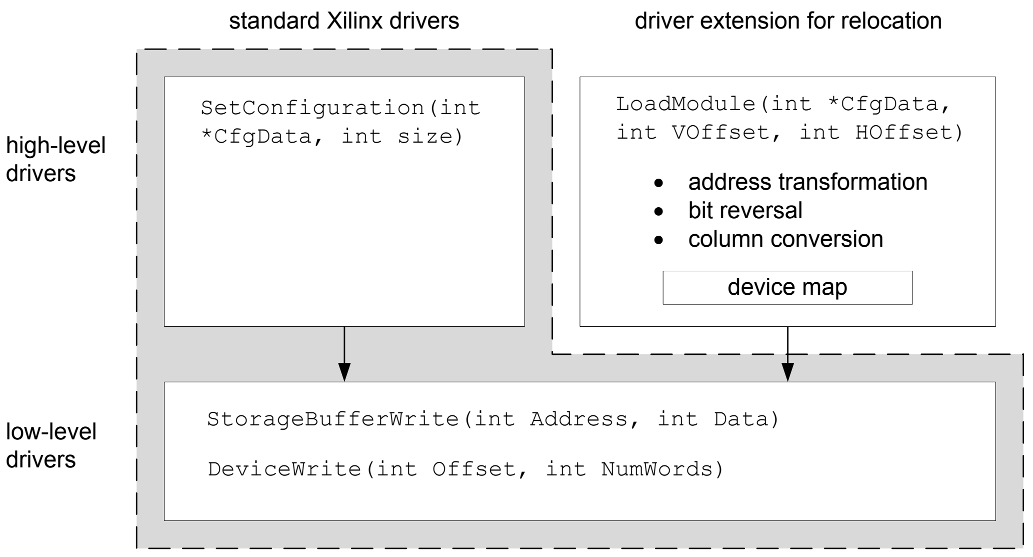

standard Xilinx drivers
driver extension for relocation
high-level drivers
SetConfiguration(int *CfgData, int size)
LoadModule(int *CfgData, int VOffset, int HOffset)
address transformation
bit reversal
column conversion
device map
low-level drivers
StorageBufferWrite(int Address, int Data)
DeviceWrite(int Offset, int NumWords)

Figure 5.10: Driver structure of HwICAP. Our high-level driver adds relocation functionality based on existing low-level drivers.

vides a new high-level function called `LoadModule()` which allows us to load any partial bitstream with a vertical and horizontal offset from its original location. The vertical offset has to be a multiple of 16 CLBs since this is the basic vertical unit of configuration. The horizontal offset can be any whole number of CLBs. The function returns an error if one tries to relocate a module outside the bounds of the device. Our driver also contains device maps specifying the location of heterogeneous resources for all Virtex-4 devices. To relocate the bitstream, our driver performs three tasks:

1. Address transformation: the memory addresses for configuration data have to be calculated based on a location offset applied to the bitstream.

2. Bit reversal: if configuration frames are relocated into the other half of the FPGA the order of bits has to be reversed (except the clock configuration).

3. Column conversion: if a column is relocated to a non-identical location the routing frames are copied and pad frames are added if necessary.

Our driver function first extracts the original module location based on the address given in the bitstream. The bitstream is then parsed into blocks of configuration data which correspond to one column of resources, e.g. one column of CLB, DSP or BRAM configuration data. As a next step, the new target address is determined for each column. The new column address cannot be calculated based on simple equations because of the irregular structure of the configuration memory map show in figure 5.8. Configuration data can be exchanged between block type 0 and block type 1 in case of a resource mismatch involving a BRAM. BRAMs follow a different address scheme than CLBs, DSPs, IOs and clocks and it is not possible to correlate both address spaces without further device information. Therefore we read the IDCODE which is part of the bitstream header and specifies the device type. We then retrieve the device map for this device and use this to transform both block type 0 and type 1 address spaces into a unified linear address space. This enables calculation of the target address and checking for mismatching resources.

The next step is to relocate the configuration data. In the case of horizontal relocation, the function checks if the target location provides an identical resource. If the target column provides the same resource the configuration data block is written out without modification. In case of a resource mismatch the function copies the first 20 frames. CLB target columns are filled with two pad frames and DSP target columns with one pad frame. In the case of vertical relocation, the function checks if the target region is located in the opposite half of the FPGA. Is this case the frame is bit reversed. For combined horizontal and vertical relocation, both steps are executed sequentially.

After our driver has performed all the above steps, blocks are written to the device by using the low-level drivers as shown in figure 5.10. Each block of configuration data is combined with the new target address and directly written to the device before the next block is parsed. This keeps the memory requirements during the bitstream parsing low, because data do not have to be copied between block types and the driver only has to keep track of one target address at the time.

Figure 5.11 illustrates an example for a horizontal relocation that contains one non-identical column. The original partial bitstream generated by the Bitgen tool contains a start sequence, start address, configuration data and end sequence. For relocation, the bitstream is parsed into columns which are written out individually. The first column is retargeted to an identical resource, and configuration data in simply copied with a new address. For the second column, a mismatch is detected. The function DSP2BRAM copies the first 20 frames containing routing information and discards the last frame. The last column is relocated to an identical resource without modification. Each column is transferred with a start and end sequence to avoid synchronisation loss with ICAP between transfers. ICAP expects constant streaming of configuration data during an ongoing configuration process and interruptions caused by the processing of the bitstream can result in synchronisation problems. The average increase in overall data volume is only 4% compared to the original bitstream.

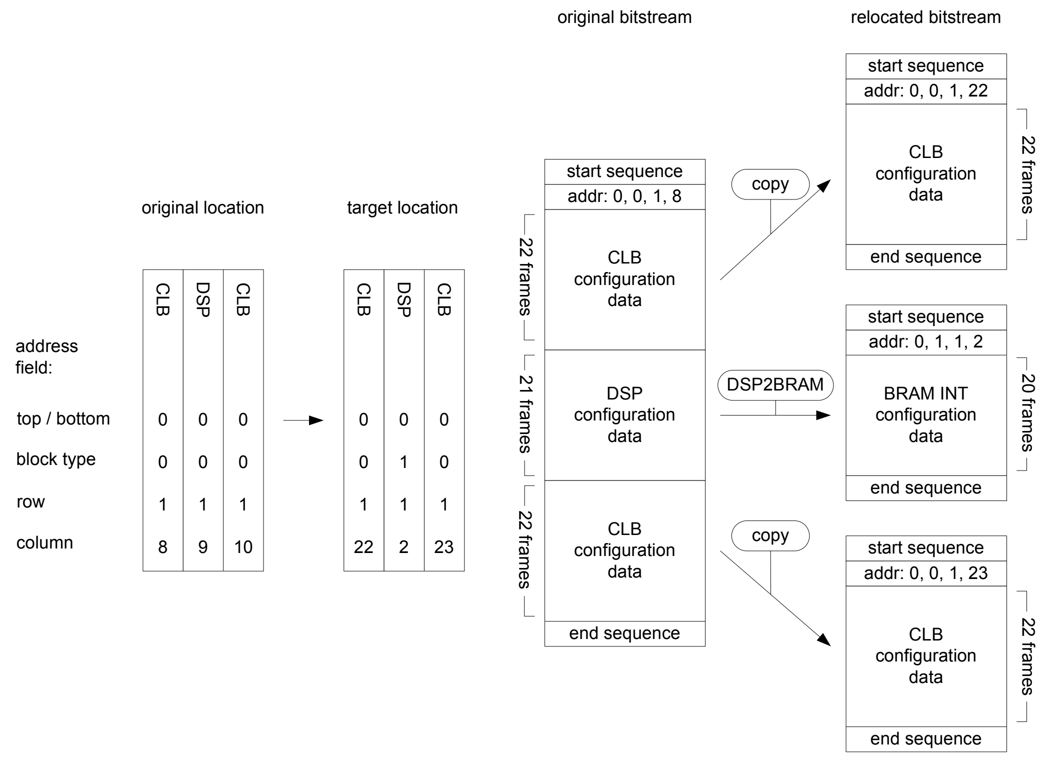


Figure 5.11: Example of a horizontal bitstream relocation from DSP to BRAM.

5.3 Placement of relocatable PR regions

Using relocatable modules in heterogeneous devices introduces additional placement constraints for the PR regions compared to non-relocatable modules. This applies to both identical and non-identical target regions. As explained in section 5.1, systems that use the prior-art approach of identical regions need an identical footprint of resources on the functional layer. Our approach does not require an identical footprint, but the number of non-matching resources should be minimised as they can only be used for routing.

In the following we present a floorplanning technique that aims to place relocatable PR regions on the device while minimising the amount of non-matching resources. We overlay regions to evaluate what resources they have in common. This is illustrated in figure 5.12. The mask contains a map of identical resources that can be used, and non-matching resources are marked as PROHIBIT. This information can later be passed on to the implementation tools so that no logic is placed in masked-out resources.

PR regions have to satisfy the resource requirements of computational tasks that have to be implemented in these regions. Other constraints such as the static system and communication infrastructures have to be considered as well. Our floorplanning technique can be used to find placements for a fixed number of PR regions, or to find the maximum

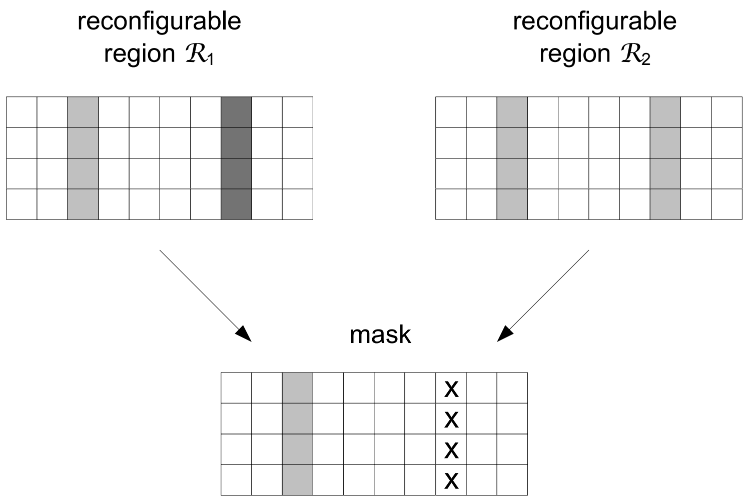

reconfigurable region $\mathcal{R}_1$
reconfigurable region $\mathcal{R}_2$
mask
X
X
X
X

Figure 5.12: Mask creation for two non-identical regions

number of PR regions that can be placed on the device.

5.3.1 FPGA model for region placement

We consider an application with m computational tasks $\mathcal{T}$ that have to be implemented as reconfigurable modules. This application is mapped onto an architecture with k PR regions. A PR region is a dedicated region in the FPGA that can hold one reconfigurable module at a time. Reconfigurable modules are dynamically allocated to PR regions depending on application requirements: if task $\mathcal{T}_i$ is required, the according module is loaded into a free PR region.

The aim is to find a feasible floorplan for the reconfigurable system that consists of a static part $\mathcal{S}$ and k PR regions $\mathcal{R}_i$. A feasible floorplan requires PR regions to be non-overlapping and large enough to accommodate all tasks. Communication infrastructures have to be considered as well. Regions may require free space between them to accommodate the necessary infrastructure.

We model the architecture and application based on the resources provided and required. The architecture can be a two-dimensional configuration architecture such as Virtex-4 or Virtex-5 or a one-dimensional architecture such as Virtex-2, Virtex-2 Pro or Spartan-3. In the following, we describe our approach based on the Virtex-4 configuration architecture described in section 5.2.

In order to describe resources required by a module or provided by a region, we introduce the n-tuple u. Each tuple component represents the number of a particular type of resource. For the remainder of this paper, we use a 3-tuple that represents the resources

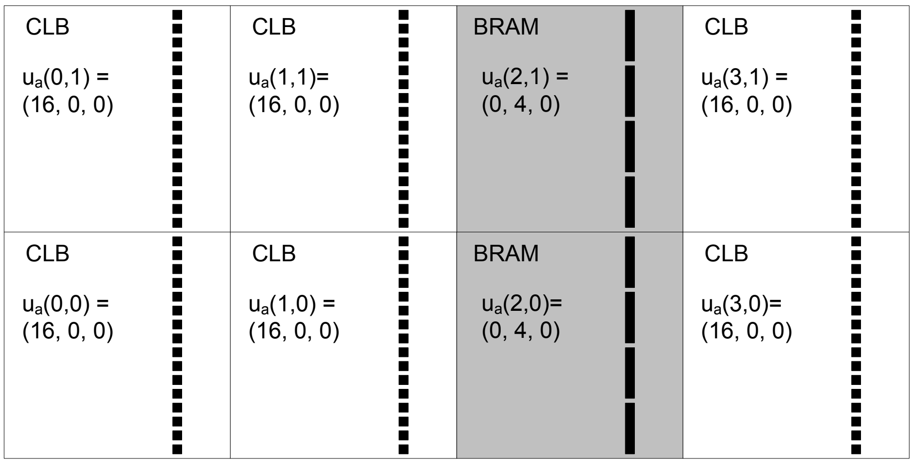


Figure 5.13: Modelling of the FPGA architecture. Shown are atomic PR units of Virtex-4 FPGAs.

CLB, BRAM and DSP: $u = (u_{clb}, u_{bram}, u_{dsp})$. However, the tuple can be easily extended for additional resources if necessary. Resources that are made available by a region are denoted u_a and resources that are required by a module are denoted u_r.

We model the FPGA architecture as a two-dimensional array, where each array element represents an atomic partially reconfigurable unit (PR unit). A PR unit is the smallest unit of reconfiguration from the perspective of module-based reconfiguration. As shown in section 5.2, Virtex-4 FPGAs have PR units that have a size of 16x1 CLBs, 4x1 BRAMs or 8x1 DSPs. In our model, each array element contains a tuple u_a that represents the available resources in this PR unit. For instance, a PR unit in Virtex-4 FPGAs with the x,y coordinates (3,1) is of the type CLB and has $u_a(3, 1) = (16, 0, 0)$ as illustrated in figure 5.13. When describing one-dimensional architectures such as Virtex-2, PR units span the height of the entire device and the array therefore contains only one row of elements. The placement problem is then simplified to analysing one row. The resources of the device are represented by u_{device} which is the sum of all resources in the device. A PR region $\mathcal{R}_i$ is a rectangular sub-array of the device and its available resources are represented by the tuple $u_a(\mathcal{R}_i)$.

We can model the application as a set of permanent functions or tasks corresponding to the static system part $\mathcal{S}$ and a set of m reconfigurable tasks $\mathcal{T}_1$ to $\mathcal{T}_m$. Each task $\mathcal{T}_i$ is characterised by its resource requirements $u_r(\mathcal{T}_i)$. The resource requirements of the static part is $u_r(\mathcal{S})$. The reconfigurable system provides k reconfigurable regions $\mathcal{R}_1$ to $\mathcal{R}_k$ that hardware implementations of the tasks can be allocated to. The number of regions depends on how many tasks need to run concurrently. The resource requirement $u_r(\mathcal{R}_i)$

of a region is defined by the combination of resource requirements of all tasks that can be allocated to this region. More specifically, it is the elementwise maximum of all $u(\mathcal{T}_i)$ for all tasks $\mathcal{T}$.

$$u_r(\mathcal{R}_i) = \begin{pmatrix} \max(u_{clb}(\mathcal{T}_1), \cdots u_{clb}(\mathcal{T}_m)) \\ \max(u_{bram}(\mathcal{T}_1), \cdots u_{bram}(\mathcal{T}_m)) \\ \max(u_{dsp}(\mathcal{T}_1), \cdots u_{dsp}(\mathcal{T}_m)) \end{pmatrix} \tag{5.1}$$

A feasible PR region must satisfy this resource requirement, that is $u_a(\mathcal{R}_i) \geq u_r(\mathcal{R}_i)$. In practice, a PR region will always contain more resources than the minimum requirement. Due to the heterogeneous nature of the fabric, regions may contain resources that are not needed or masked out. Feasible regions usually also provide more resources than required because of their rectangular shape. If the resources provided by a region are very close to the minimum requirement, the placement and routing tool may have difficulties to successfully implement a task in this region. This can be solved by slightly increasing the region size.

In our case all $u_r(\mathcal{R}_i)$ are equal since the same set of tasks can be used in all regions. The resource requirement for the entire system is the sum of the static part and all PR regions:

$$u_{r,system} = u_r(\mathcal{S}) + \sum_{i=1}^{k} u_r(\mathcal{R}_i) = u_r(\mathcal{S}) + k \cdot u_r(\mathcal{R}) \tag{5.2}$$

The first step in implementing a system is choosing a target device that provides enough resources:

$$u_{device} \geq u_{r,system} \tag{5.3}$$

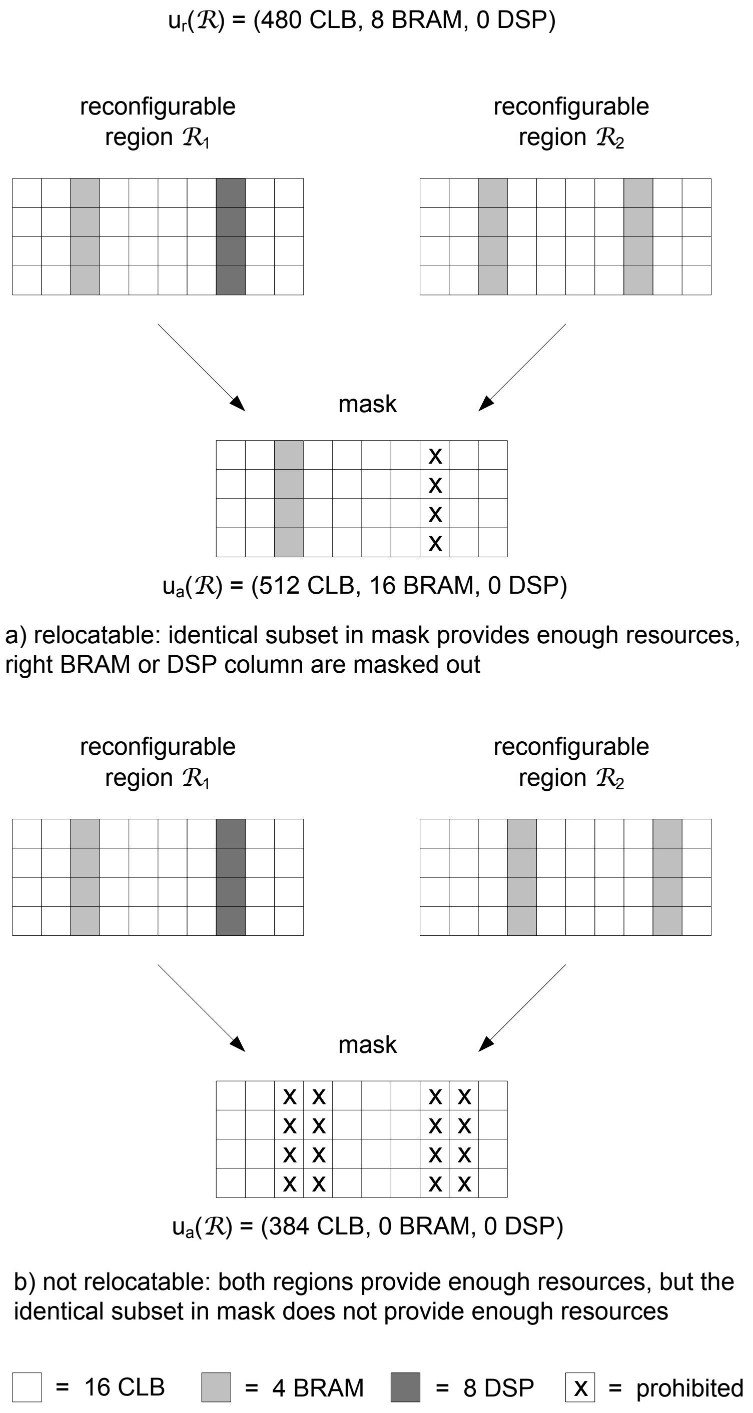
This, however, is only an initial estimate and does not account for masked out resources or area required for communication channels. If $u_{r,system}$ is close to u_{device} it may be necessary to use the next larger device for the final implementation.

5.3.2 Placement algorithm

We now want to find a feasible placement for all PR regions. For a placement to be feasible, regions have to be non-overlapping and large enough according to equation 5.1. Ideally we try to find PR regions that are fully identical, but this can be difficult to achieve in heterogeneous FPGAs. If it is not possible to find identical regions, we make use of the technique presented in section 5.1.

In the following we present a simple heuristic to find a feasible placement of regions with a compatible subset of resources. The technique is similar to template matching in that it searches for matching regions across the device. The difference to regular template matching is that matching resources are counted separately for each resource type, and the template (called *mask*) is updated throughout the search with non-matching resources. The mask is used to check whether the compatible subset of resources is sufficient, and it keeps track of positions that are masked out. This is illustrated in figure 5.14. A resource requirement of $u_r(\mathcal{R}) = (480, 8, 0)$ is given. Comparing the two regions shown in figure 5.14a results in a mask with $u_a = (512, 16, 0)$. This is sufficient for the given requirement, and relocation between these two regions is possible. In figure 5.14b, the mask only shows available resources of $u_a = (384, 0, 0)$. Hence, regions cannot be used for relocation even though each PR region by itself provides enough resources. In this case the search continues without recording a match. If a match as illustrated in figure 5.14a is found, then the mask is updated with the new PROHIBIT values, and locations marked as such will be blocked from being used when searching for further regions.

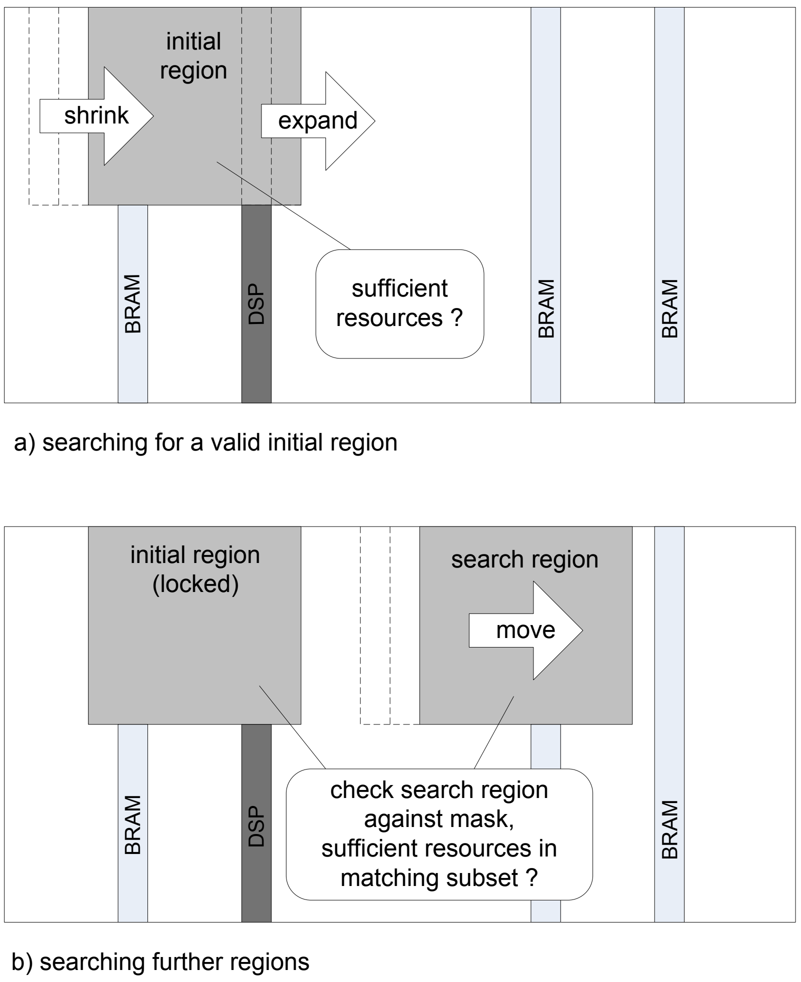
The key part of our algorithm involves finding one initial regions that is suitable, and searching for further compatible regions across the device. This is illustrated in figure 5.15. The search for an initial region begins by expanding the right boundary of the region until the resource requirements are met, that is all values in $u_a(\mathcal{R})$ have to be larger or equal to the ones specified in $u_r(\mathcal{R})$. Next, the region is shrunk by moving the left boundary up to the point where further shrinking would violate the resource requirements. This shrinking is done because a region might have to expand far to the right to gain enough of one type of resource. At the same time, it might also gain excessive amounts of other resources. These can be reduced by shrinking the left side as indicated in figure 5.15a. The region now has minimal size and is locked down as a seed for all other possible regions. A mask containing all the resources of initial region is created. In the following we try to find all possible additional regions that provide a sufficiently large identical subset of resources with the initial region. This is done by sliding a search window with the same size of the initial region along the device as illustrated in figure 5.15b. In each location we use the mask to check if the underlying fabric provides a compatible subset with sufficient resources. If a match is found, the region is locked down and added to a region list. If there are non-matching resources in the matching area then the mask is updated with PROHIBIT values as illustrated in figure 5.14a. The search continues with the updated mask and we proceed in the same manner to find all possible further regions within the current row.



$u_r(\mathcal{R})$ = (480 CLB, 8 BRAM, 0 DSP)
reconfigurable region $\mathcal{R}_1$
reconfigurable region $\mathcal{R}_2$
mask
$u_a(\mathcal{R})$ = (512 CLB, 16 BRAM, 0 DSP)
a) relocatable: identical subset in mask provides enough resources, right BRAM or DSP column are masked out
reconfigurable region $\mathcal{R}_1$
reconfigurable region $\mathcal{R}_2$
mask
$u_a(\mathcal{R})$ = (384 CLB, 0 BRAM, 0 DSP)
b) not relocatable: both regions provide enough resources, but the identical subset in mask does not provide enough resources
= 16 CLB
= 4 BRAM
= 8 DSP
x = prohibited

Figure 5.14: Examples for relocatable and non-relocatable scenarios for PR regions with a resource requirement of 480 CLBs and 8 BRAMs

Since devices exhibit vertical symmetry, we can replicate a solution for one row vertically across the device until the full height is covered. However, devices with PowerPC cores are not fully symmetrical vertically and regions overlapping with PowerPCs have to be eliminated. While populating the device with regions, the algorithm keeps track of the


initial region
shrink
expand
sufficient resources ?
BRAM
DSP
BRAM
BRAM
a) searching for a valid initial region
initial region (locked)
search region
move
check search region against mask, sufficient resources in matching subset ?
BRAM
DSP
BRAM
b) searching further regions
Figure 5.15: The key steps of our algorithm involve finding an initial region and searching for further regions with a sufficient number of matching resources.

resources used by the current region list.

The above procedure will fill the entire device with compatible PR regions based on the current initial region. The algorithm considers this region list as valid if at least k regions can be found. Surplus regions will later be eliminated based on additional design constraints such as the location of the static system. If, as an alternative, the number of regions k is not fixed, then the algorithm can be used to maximise the number of PR regions and hence, the number of tasks that can be executed in parallel. Valid region lists are added to a placement list, which represents competing valid design choices.

The search for compatible regions as illustrated in figure 5.15b is repeated in a nested loop for various initial regions and aspect ratios in order to populate the placement list with various placement choices. The inner loop repeats the search for alternative initial regions that are obtained with the expand-and-shrink mechanism along the top row of the device as shown in figure 5.15a. The outer loop changes the aspect ratio of the initial region. In practice, extreme region aspect ratios can be omitted because placement and routing in very narrow regions is often difficult, and it can have negative effects on performance. The search space has typically a size of several hundred PR units, with the largest Virtex-4 device containing 1248 PR units in a 104x12 array.

The next step is to choose a suitable region list from the placement list generated by the algorithm. We can use three metrics that help selecting a region list: *number of PR regions*, *fragmentation* and *masked-out resources.*

Number of PR regions: If the algorithm is used to find the maximum number of possible PR regions k, then the first criterion is to select a region list with maximum k.

Fragmentation: While our method does not introduce any problems with run-time fragmentation, the PR regions can fragment the residual free space of the device. This can cause problems with implementing the static part of the system. We can use a method presented by Walder *et. al.* that calculates the fragmentation of the free space by using maximum free rectangles [111]. A maximum free rectangle is a rectangle that does not intersect with any PR region and that is not contained by any larger free rectangles. This is illustrated in figure 5.16. Maximum free rectangles are (a, b), (c, d), (e, f) and (g, d). The fragmentation F can be calculated based on the area A_i of all maximum free rectangles:

$$F = 1 - \frac{\sqrt{\sum_{i=1}^{j} A_i^2}}{\sum_{i=1}^{j} A_i} \tag{5.4}$$

F is a value between 0 and 1 with lower values denoting less fragmentation. Choosing a region list with lower F simplifies implementing the static system. For this analysis, PR units are counted as one unit of area, regardless of their resource type.

Masked-out resources: Our method can mask out non-matching resources in PR regions in order to allow module relocation. Masked-out resources cannot be used when implementing tasks in a region and thus, may increase the size of the PR region. An increased area leads to larger configuration bitstreams and slower reconfiguration time. As a third criterion we therefore choose regions with the smallest percentage of masked-out resources. However, the existence of masked-out resources does not necessarily mean that a relocatable region is larger than a non-relocatable one. The example in figure 5.14a shows that relocatable regions can contain masked-out resources that are simply not needed by $u_r(R)$.

We can add support for communication structures to our algorithm. As explained in section 2.5, reconfigurable designs often use regular communication infrastructures that

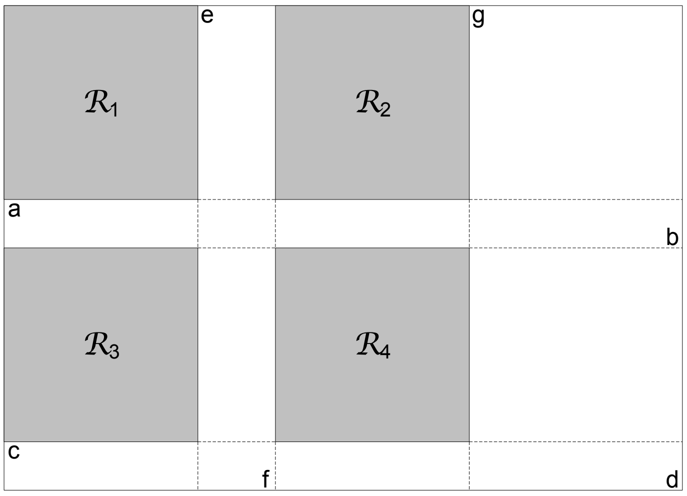


Figure 5.16: Maximum free rectangles of the fragmented residual free area.

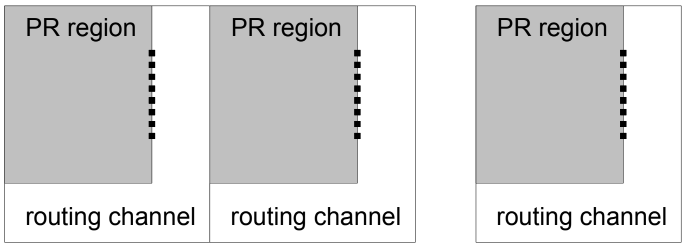


Figure 5.17: Supporting communication infrastructures in the placement algorithm.

require free channels between the PR regions. This can be supported in our algorithm by extending the region's resource requirements with dedicated routing channels. This is shown in figure 5.17. After finding an initial region and generating the region mask, a previously specified amount of extra rows or columns is added to the mask. These additional PR units are marked as PROHIBIT since they cannot be used by the module. The PROHIBIT marking also means that any type FPGA resource can be used since all of them provide routing. PR units that are kept free for routing channels are not part of a region in the resulting floorplan and hence, can also be used by the static part of the system. Routing channels are not considered to contribute to device fragmentation, because they are intentionally created. Routing channels can be added to either side of a region, and can constitute of a single row or column or an L-shape.

5.4 Case study

In this section we present several experiments and analyses to study our relocation technique. We develop a reconfigurable system with two non-identical PR regions to test and evaluate our software driver implementation. We also perform an abstract analysis of relocation that shows the number of relocation options if regions have to be fully identical, and if some percentage of resources can be masked out. Finally, we study our placement approach for relocatable regions based on a several signal processing cores.

5.4.1 Bitstream relocation in software-defined radio prototype

We develop a software-defined radio prototype with two PR regions that can each host one of four reconfigurable filter modules. The filter modules are relocatable between these two regions and the following types are available: low pass, band pass, high pass and all pass. The two PR regions are connected to two audio channels. We can apply a filter to an audio channel, and successful reconfiguration and relocation will lead to audible changes in the signal. The purpose of this design is to test and evaluate our software implementation with simple infrastructure and a typical development board. However, our prototype could also be adapted to use more complex reconfigurable waveforms such as the ones used in [89] and [108].

The design is implemented on a Xilinx ML401 board that contains a Virtex-4 XC4VLX25 FPGA. Figure 5.18 illustrates the structure of our system. It contains a MicroBlaze-based configuration controller together with the two reconfigurable regions. The MicroBlaze processor runs at 100 MHz and uses 64 kB of on-chip memory. The system also contains the HwICAP core with our extended drivers. A flash controller is used to retrieve the partial bitstreams from external flash memory during system boot-up. Bitstreams are not read from flash memory for reconfiguration because of the very slow transfer speeds. Instead they are transferred into DDR RAM memory which allows fast transfers. The system is controlled through a UART core that is connected to a terminal program.

In this example, relocatability reduces the number of partial bitstreams by a factor of $k = 2$ since one implementation of each filter can be used in both PR regions. The two regions could theoretically be placed above each other, making use of identical functional layers. However, due to the required access to DDR RAM and flash IO pins this is not possible. Thus, regions are located next to each other, covering functional layers that are not fully identical. The left region contains one BRAM and one DSP column and the right region provides two BRAM columns.

Figure 5.19 shows the placed and routed design with the high pass filter present in the left region. The right region is empty. All filters are implemented in the left region with a PROHIBIT constraint on the non-matching DSP column. In this particular design

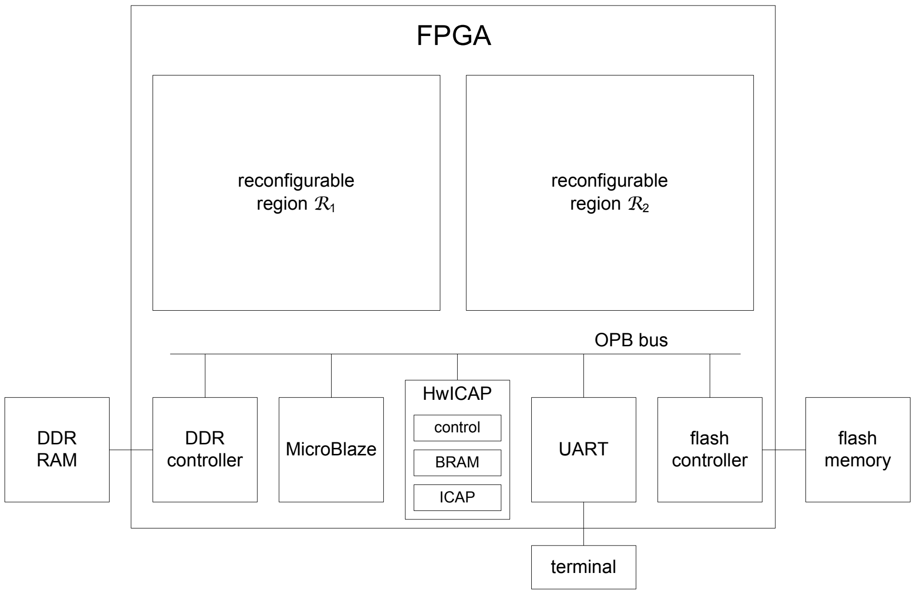


Figure 5.18: Structure of the system: We use two reconfigurable regions and a configuration controller based on MicroBlaze and HwICAP.

however, this would not be necessary as filters are implemented as distributed arithmetic and do not use DSPs. The design is implemented with the Xilinx early access partial reconfiguration toolflow based on ISE 9.2. A PC with a 3.2 GHz Pentium 4 CPU and 1 GB of RAM is used. Since the number of partial bitstreams that need to be produced is cut from 8 to 4, the design compile time is also reduced from 54 min to 31 min, a reduction of 43%.

If a filter is configured into the left region, the unmodified bitstream is loaded with the `SetConfiguration()` function of the original device driver. In order to load a filter to the right region the bitstream is relocated with the `LoadModule()` function of our device driver. The function transforms the bitstream accordingly. In our design, filters function properly in both regions. Hence, the design could prove that a bitstream can be successfully loaded into a non-identical region using our bitstream transformations. Furthermore, we evaluate the configuration speed in our system. The time required to load a bitstream is measured using a bus timer module. The partial bitstreams have a size of $\Psi = 127\ kB$ and can be loaded to the original location in $t_r = 25.4\ ms$. This is equivalent to a transfer rate of $\Phi_{config} = 5\ MB/s$. Loading the bitstream to the right location with our device driver takes $t_r = 27\ ms$ which represents a transfer rate of $\Phi_{config} = 4.7\ MB/s$. This represents performance penalty of 6% which can be explained with the increased data volume and the additional processing. These results could vary with system frequency, system load or

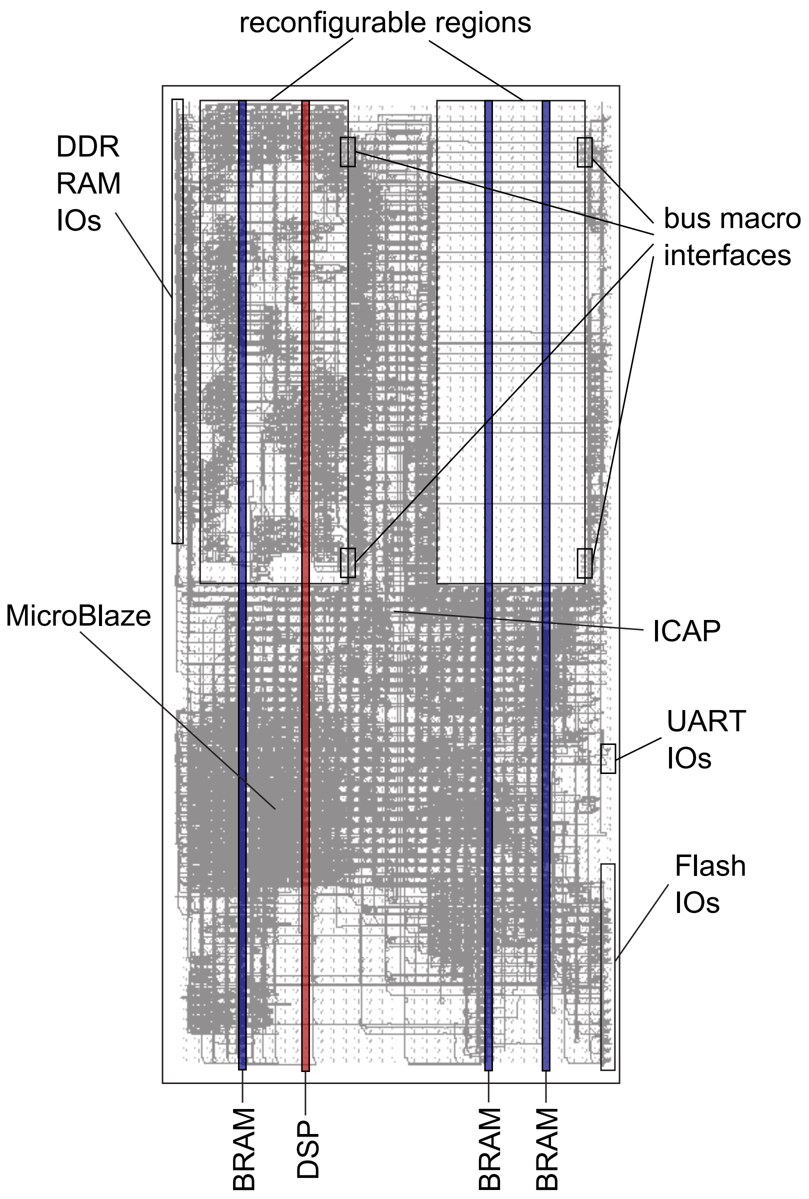


Figure 5.19: Reconfigurable and relocatable software-defined radio prototype: filters for two channels on a Virtex-4 XC4VLX25 FPGA. A filter is implemented only in the left region and one bitstream is stored. The configuration can be relocated dynamically.

when using the PowerPC. We could not measure our driver with the improved ICAP core presented in [22] as it does not use the HwICAP drivers. However, our method could be adapted as a stand-alone functionality that can work with other types of ICAP cores as well.

5.4.2 Analysis of relocatability

In this section we analyse how the relocatability is enhanced by our method. The analysis is performed on a Virtex-4 XC4VFX140 FPGA. We choose this device because it is the largest in the Virtex-4 FX series and therefore a good potential candidate for the implementation of a complex reconfigurable software-defined radio system. The FPGA has a height of 192 CLBs and a width of 84 CLBs. A total of 12 BRAM, 2 DSP and 3 IO columns interrupt the CLB fabric. These resources are arranged in a pattern where four CLB columns are interleaved with one heterogeneous column. The device also contains two PowerPC processors which are excluded from our method. Although it might be possible to identify a compatible subset between regions with PowerPCs we consider this as impractical. Rows above or below the PowerPC core, though, can be used without restriction.

To put module sizes in perspective, we try to obtain a size estimate of potential reconfigurable modules. The filters from our previous example require about 2000 LUTs which is equivalent to 250 CLBs. Thus, a filter could be constrained to a region of 16x16 CLBs. The radio waveforms in [89] and the cores in section 5.4.3 are up to four times larger and could cover a region of 32x32 CLBs. We therefore consider a width of 16 CLBs as narrow reference region, and a width of 32 CLBs as wide reference region. These numbers correspond to a relative width of 19% and 38% of the device.

In the following analysis we vary the width of PR regions and measure the number of alternative placement options for horizontal relocation. The width of a region is measured relative to the width of the device. Figure 5.20 illustrates the relocatability for three different scenarios. Traces in the diagrams correspond to regions with growing width, with the abscissa showing the relative width and the ordinate showing how many alternative placements for a region of this width can be found. The upper diagram illustrates alternative placements without our method; the target region has to be fully identical. Small regions with a width up to 10% have up to 15 alterative placement options. Regions with a width between 10% and 15% can have up to five further placement options and all other regions with a width up 30% have at most one further placement option. All regions wider than 30% cannot support relocation. For the narrow reference region, there is only one fully identical alternative region. However, regions identified as valid options do not necessarily have to be feasible in a real floorplan. The placement can be too close to a PowerPC, or can cause undesirable fragmentation of the device. The wide reference region cannot support relocation.

With our method it is in principle possible to relocate a module to any other region. However, it is preferable to maximise the percentage of matching resources since all non-matching resources have to be masked out. The middle diagram illustrates alternative placement options if we require at least 90% identical resources between two regions. Overall, the relocatability is significantly increased and there are 11 alternative placements for the narrow region and two for the wide region. Regions with a width up to 46% can be used for relocation. Traces do not decrease monotonically because of the relative nature of our analysis. Including a non-identical resource can cause a region to drop below the required 90% threshold and further expansion over identical resources will increase this ratio again. If we expect at least 80% matching resources, regions with a width up to 99% can support relocation as illustrated in the lower diagram. However, this is only of theoretical relevance as any region wider than 50% of the device would automatically overlap with its alternative placement. The narrow region can have 34 alternative placements and the wide region 11. For our reference regions a parameter of 90% match seems to be appropriate as it provides sufficient alternative placements. It has to be noted that non-matching resources might not be needed by the application. For example a module solely implemented in CLB logic uses BRAM columns only for routing. If this module is relocated over a DSP column no additional resources are wasted by our method.

Future FPGA architectures can be optimised for bitstream relocation without having to relinquish heterogeneity. If specialised columns are arranged in a regular interleaved pattern similar to Virtex-4 FX devices, relocation is significantly simplified. An FPGA designer can try to create a high level of symmetry in the device. Potentially columns can be rearranged without affecting the overall performance. This could provide enough identical regions from smaller modules. Larger modules might be hindered by one or two non-identical columns. Our technique will also enable the relocation of those modules.

5.4.3 Placement case study

We now evaluate the region placement method presented in section 5.3. This is done based on a number of signal processing cores used in software-defined radio. Table 5.2 shows a range of signal processing cores and their resource requirements with some parameter variations. All cores are generated with Xilinx CORE Generator and synthesised for Xilinx Virtex-4 FPGAs. We cluster cores of similar size into various task groups and calculate the region resource requirement $u_r(\mathcal{R})$ for these groups based on equation 5.1. Our algorithm is then used to find regions that satisfy this requirement and allow relocation.

We also add a resource requirement for the static part of the system, including a reconfiguration manager based on a MicroBlaze processor and the internal configuration access port (ICAP). The resource requirement of this static system is $u_r(\mathcal{S}) = (515, 33, 3)$.

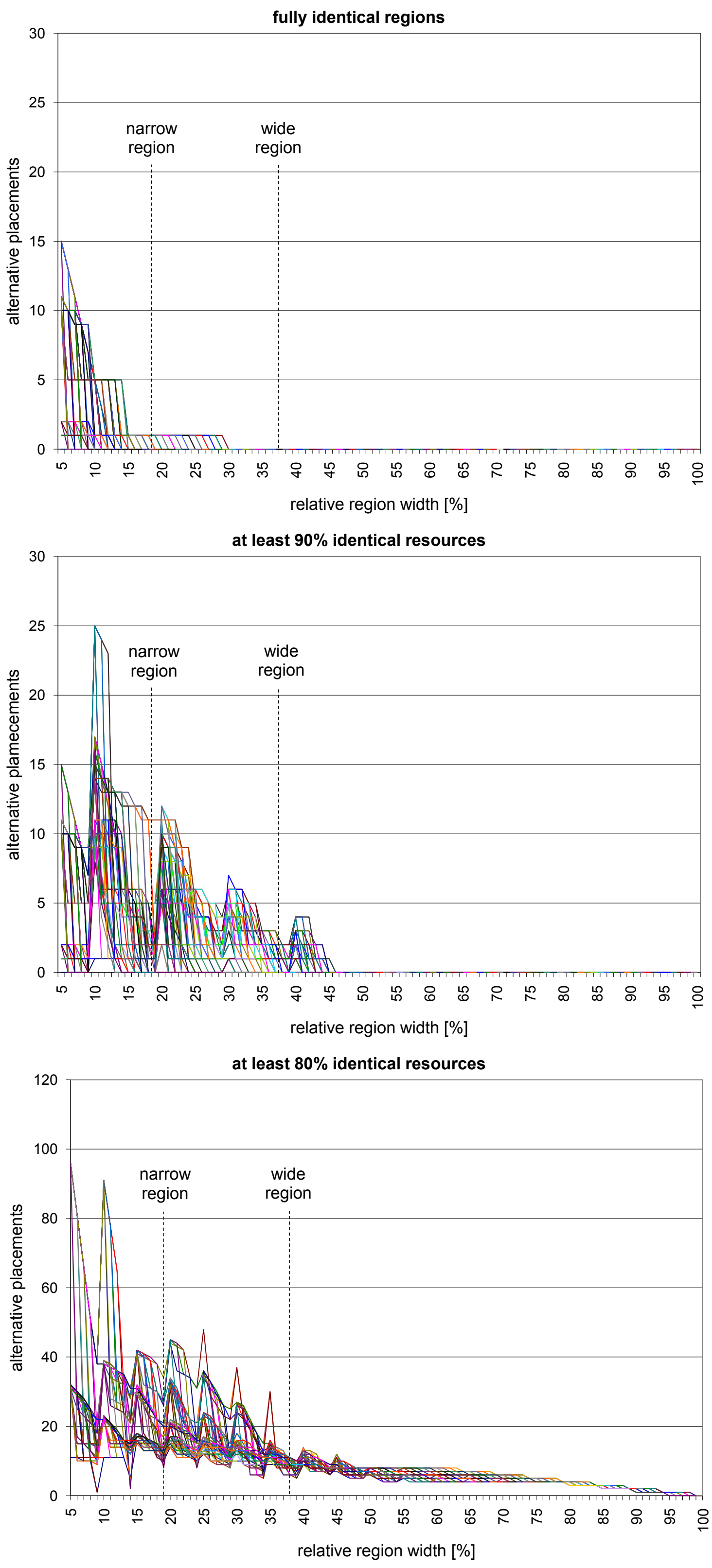


Figure 5.20: Alternative placement options for software-defined radio application on a Virtex-4 XC4VFX140 FPGA for different relative module widths and for fully identical, at least 90% identical and at least 80% identical target regions.

core	number	version	$u_r(\mathcal{T})$
FIR	1	64 tap dist arithm	(961,0,0)
	2	32 tap speed	(235,0,16)
	3	64 tap speed	(474,0,32)
	4	32 tap 3-chan	(870,0,48)
Direct Down Conversion	5	dyn range 108dB	(488,16,0)
	6	dyn range 60dB	(437,0,0)
Direct Digital Synthesis	7	small	(88,1,0)
	8	medium	(105,1,2)
	9	large	(133,8,3)
Cordic	10	Sin/Cos	(154,0,0)
	11	Rotate	(170,0,0)
FFT	12	32pt complex	(869,16,0)
	13	256pt radix	(200,0,3)
	14	256pt pipeline	(281,0,0)
	15	1024pt rad. 8-chan lite	(604,7,16)
	16	1024pt rad. 8-chan	(1004,17,24)
Forward Error Correction	17	speed opt.	(301,13,0)
Viterbi Decoder	18	parallel	(495,2,0)
	19	serial	(272,2,0)
Turbo Decoder	20	core 1	(611,17,0)
	21	core 2	(315,7,0)

Table 5.2: Signal processing cores and their resource requirements.

As a communication constraint we require a free routing channel with a height of 16 CLBs below each region. Given $u_r(\mathcal{R})$ and $u_r(\mathcal{S})$, we use our algorithm to find the maximum possible number of PR regions on the target device. We compare the results to relocatable regions requiring an identical footprint as well as non-relocatable regions.

Table 5.3 shows the number of PR regions that can be found for various $u_r(\mathcal{R})$ with and without relocation on a range of Virtex-4 devices. Column two shows the total number of resources provide by the device. In column three, we show which tasks from table 5.2 are clustered into a group, and we list the combined resource requirements for this group in column four. Column five shows the actual resources provided by the PR regions that are found by our algorithm. As a comparison, column six lists the number of possible regions with relocation when requiring fully identical regions. This scenario represents prior art in relocation. Column seven shows the number of possible relocatable regions found by our algorithm that is enabled by our method of masking our non-matching resources. We see that in 9 out of 15 cases, our algorithm finds more regions than would be possible with identical regions. In the other six cases, fully identical regions can be found which is illustrated by a value of 0% masked-out PR regions. Column eight lists

1	2	3	4	5	6	7	8	9	10
target device	u_{device}	task group (core number)	$u_r(\mathcal{R}) = \sum u_r(\mathcal{T})$	$u_a(\mathcal{R})$	$k^{(1)}$	$k^{(2)}$	$k^{(3)}$	masked-out PR units per region	F
XC4VFX100	(10544,376,160)	1, 12, 20	(961,17,0)	(1024,32,0)	2	4	6	10%	0
		5, 6, 18, 20	(611,17,0)	(640,32,0)	5	6	6	7.7%	0.38
		17, 19, 21	(315,13,0)	(320,16,0)	6	9	9	7.7%	0.46
		7, 14, 19, 21	(315,7,0)	(320,8,0)	6	12	12	21.4%	0.35
XC4VSX55	(6144,320,512)	1, 4, 12, 16	(1004,17,48)	(1056,60,96)	3	3	3	0%	0.13
		3, 5, 15, 20	(611,17,48)	(640,32,64)	5	5	5	0%	0.35
		2, 8, 9, 13	(235,8,16)	(256,16,32)	11	11	11	0%	0.47
XC4VLX40	(1608,96,64)	6, 18, 21	(495,7,0)	(512,8,0)	3	3	4	0%	0.17
		7, 10, 11, 19, 21	(315,7,0)	(320,8,0)	3	5	5	8.3%	0.34
XC4VLX80	(8960,200,80)	1, 12, 20	(961,17,0)	(1024,32,0)	2	4	4	5.3%	0
		5, 6, 18, 20	(611,17,0)	(640,32,0)	2	4	4	7.7%	0.16
		6, 14	(437,0,0)	(448,0,0)	6	12	12	12.5%	0.32
		17, 19, 21	(315,13,0)	(320,16,0)	6	6	6	0%	0.17
		7, 14, 19, 21	(315,7,0)	(320,8,0)	6	6	9	0%	0.15
		10, 11, 14	(281,0,0)	(288,0,0)	6	12	12	18.2%	0.54

(1) identical, relocatable regions
(2) non-identical, relocatable regions using masking out of non-matching resources as shown in figure 5.14
(3) non-relocatable regions

Table 5.3: Number of possible PR regions k on various Virtex-4 FPGAs for relocatable and non-relocatable scenarios.

the number of PR regions that can be found if modules do not have to be relocatable. Comparing columns seven and eight, we see that in 12 out of 15 cases, our algorithm finds as many relocatable regions as regions without relocation, therefore reducing the configuration storage requirement in these cases without impacting the number of regions that can be found. The percentage of resources that have to be masked out when using our method is shown in column nine, and the fragmentation F of the resulting floorplan based on equation 5.4 is given in column ten. When searching for the maximum number of PR regions, the algorithm only finds a limited number of solutions that often bear resemblance to each other e.g. showing symmetry or minor variations. The optimisation potential for reducing fragmentation of the residual free space or the percentage of masked-out resources is therefore limited. The presented solutions are, if possible, optimised for masked-out resources first and secondly for fragmentation. All designs provide sufficient unfragmented space to implement the static reconfiguration manager.

The resulting floorplans for first, fifth and last case in table 5.3 are shown in figure 5.21 5.22 and 5.23. Figure 5.21 illustrates the first floorplan for the XC4VFX100 device. Relocation between identical regions is only possible for vertical relocation, e.g. between region 1 and 4. Relocatability is improved by our method as it is now possible to identify four regions (2, 3, 5 and 6). This requires masking out 10% of all PR units in each region. However, when giving up relocation it is possible to identify six feasible PR regions. Relocatability is hampered by the somewhat irregular structure in the device. BRAMs and DSPs are placed in a regular pattern towards the left and the right side of the device, but there is a larger CLB island towards the centre. On the left side, PowerPC cores stop our algorithm from finding suitable regions. The fragmentation of the floorplan with the four relocatable regions has a value of 0. Regions are concentrated on one side without gaps, leaving one large free rectangle. Routing channels below the regions are intentionally created and do not count for fragmentation.

Figure 5.22 shows the floorplan with three regions on a XC4VSX55 device. The SX55 device is particularly relocation-friendly because all its heterogeneous resources are placed in a regular pattern with equal distances. Hence, no resources have to be masked out for relocation and it is possible to find as many relocatable regions as it would be without relocation. The floorplan has a moderate fragmentation of 0.13. The lower left corner is kept free for the static system, but it would be possible to use any of the other quadrants as well.

Figure 5.23 shows a floorplan with twelve small regions on a XV4VLX80 device. This

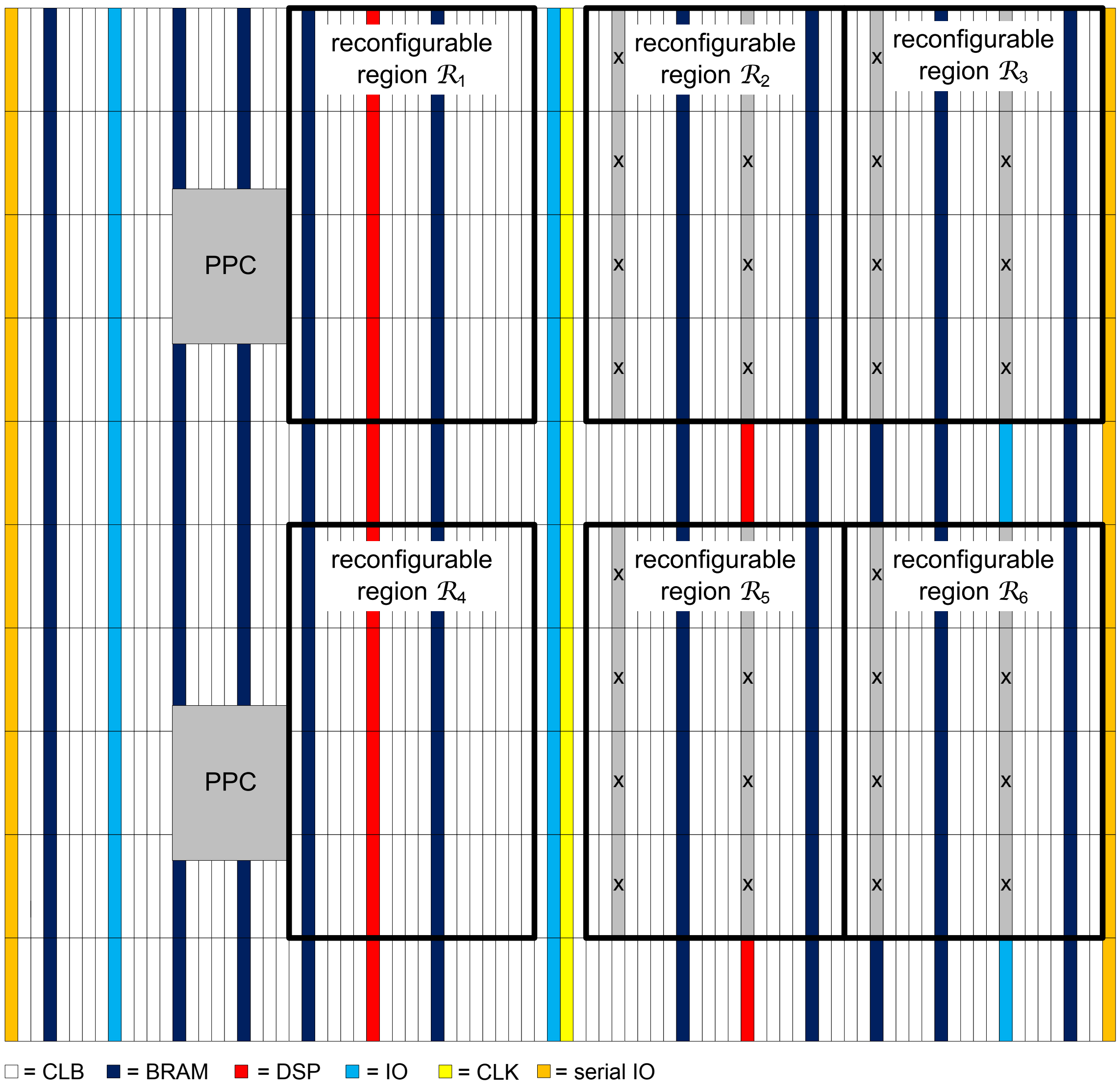

reconfigurable region $\mathcal{R}_1$
reconfigurable region $\mathcal{R}_2$
reconfigurable region $\mathcal{R}_3$
reconfigurable region $\mathcal{R}_4$
reconfigurable region $\mathcal{R}_5$
reconfigurable region $\mathcal{R}_6$
PPC
PPC
x
□ = CLB ■ = BRAM ■ = DSP ■ = IO ■ = CLK ■ = serial IO

Figure 5.21: Floorplan on a Virtex-4 XC4VFX100 with four relocatable regions and six non-relocatable regions with $u_a(\mathcal{R}) = (1024, 32, 0)$.

device also contains a large CLB island in the centre but in this case, it does not hinder relocation as heterogeneous PR units are not needed in the PR region. Our method doubles the number of relocatable regions that can be found. This requires masking out 18.2% of the PR units in each region. However, not all masked-out resources are wasted. BRAMs in the outer regions are not wasted as they would not be used by the tasks anyway. Only masked-out CLBs in the centre regions are an actual overhead. The fragmentation of the floorplan is high due to the large number of small regions interrupted by free space.

We now implement and compare designs for the 12 cases where an equal amount of relocatable and non-relocatable regions can be found. Again, designs are built with Xilinx ISE 9.2 tools. Table 5.4 shows a comparison of configuration storage sizes Ψ for each

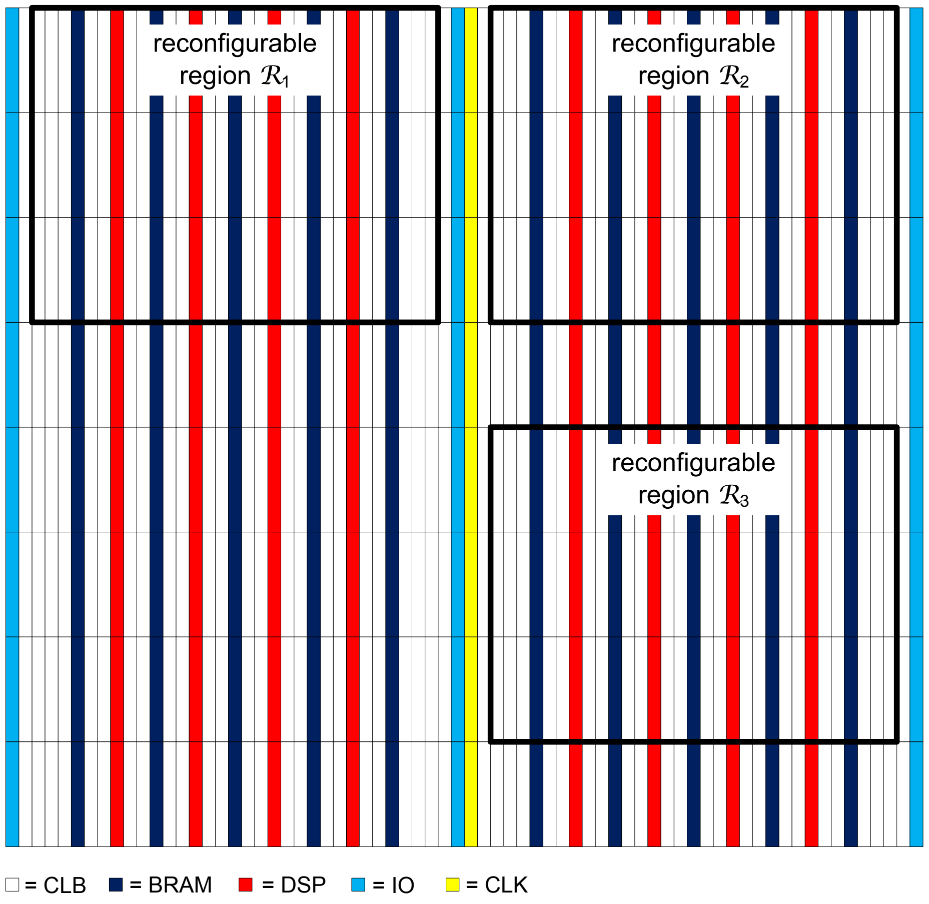

reconfigurable region $\mathcal{R}_1$
reconfigurable region $\mathcal{R}_2$
reconfigurable region $\mathcal{R}_3$
□ = CLB ■ = BRAM ■ = DSP ■ = IO ■ = CLK

Figure 5.22: Floorplan on a Virtex-4 XC4VSX55 with three fully identical relocatable regions with $u_a(\mathcal{R}) = (1056, 60, 96)$.

module with and without relocatability. For the relocatable case, the size of a single bitstream which can be used in all PR regions is shown. Without relocation, we list the combined size of all bitstreams that are needed to configure all k regions. On average we find that the configuration storage requirement per module is reduced from 1220.6 kB to 206 kB which represents a factor of 5.9. The maximum is a reduction from 1434.6 kB to 119.6 kb in the third case. This represents a factor of 12.

5.5 Summary

Bitstream relocation can significantly reduce configuration storage requirements and design compile time since only one version of each module has to be produced and stored. However, relocatability is hampered by heterogeneous resources and previous work has focussed on finding identical regions. This becomes more challenging as heterogeneity increases.

We have introduced a bitstream relocation technique that does not require fully identi-

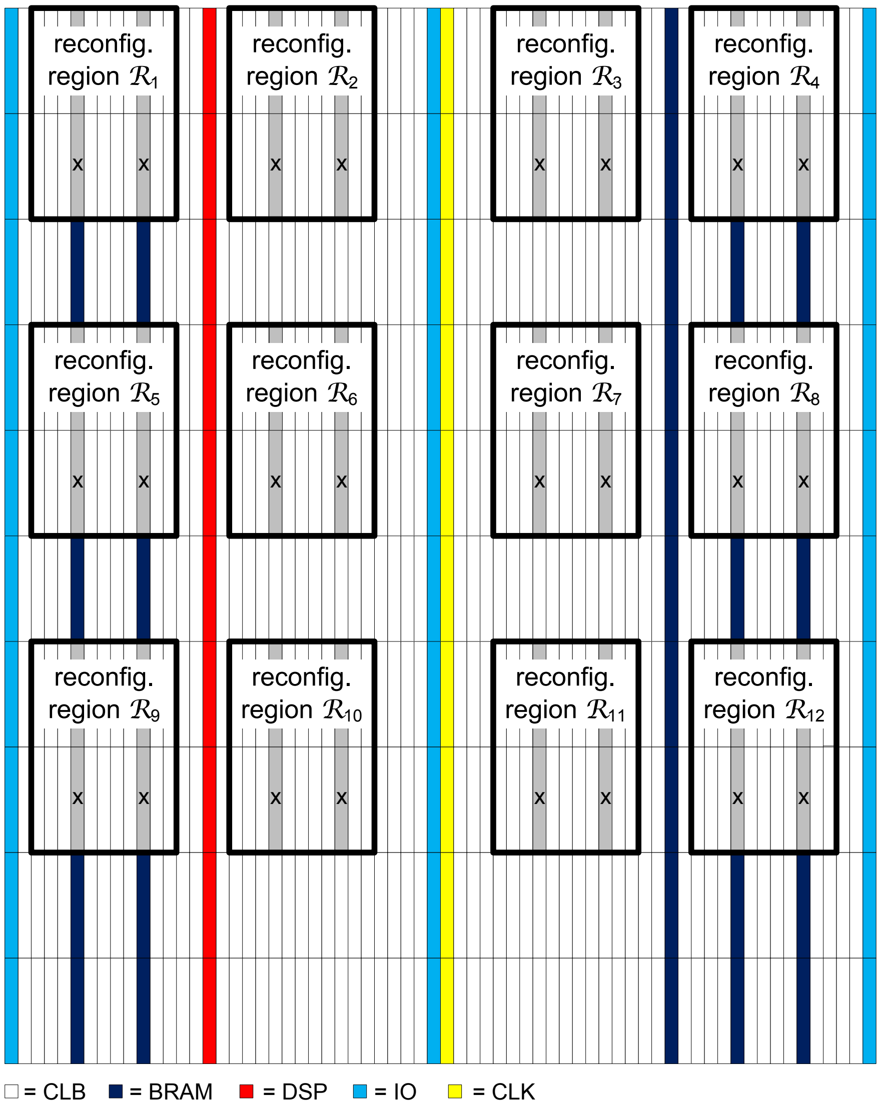

reconfig. region $\mathcal{R}_1$
reconfig. region $\mathcal{R}_2$
reconfig. region $\mathcal{R}_3$
reconfig. region $\mathcal{R}_4$
reconfig. region $\mathcal{R}_5$
reconfig. region $\mathcal{R}_6$
reconfig. region $\mathcal{R}_7$
reconfig. region $\mathcal{R}_8$
reconfig. region $\mathcal{R}_9$
reconfig. region $\mathcal{R}_{10}$
reconfig. region $\mathcal{R}_{11}$
reconfig. region $\mathcal{R}_{12}$
x
□ = CLB ■ = BRAM ■ = DSP ■ = IO ■ = CLK

Figure 5.23: Floorplan on a Virtex-4 XC4VLX80 with twelve relocatable regions with $u_a(\mathcal{R}) = (288, 0, 0)$.

cal regions. Instead we use the compatible subset of resources between PR regions. We use a formalism based on functional layer and configuration layer as an abstract model. If two tiles provide identical resources on the functional layer and have an identical mapping to the configuration memory layer, then they are considered compatible. Non-matching resources are masked out and cannot be used. Routing resources adjacent to non-matching logic resources are usually compatible and can be used for connectivity between neighbouring resources. In the following we demonstrate how this technique can be adapted to Virtex-4 FPGAs. We explain the details of the configuration memory architecture and show how configuration data representing the compatible subset can be relocated. We also implement our method as a software driver for HwICAP. This integrates our relocation

target device	$u_a(\mathcal{R})$	k	Ψ [kB] (with relocation)	Ψ [kB] (w/o relocation)
XC4VFX100	(640,32,0)	6	267.8	1606.8
	(320,16,0)	9	134.0	1205.9
	(320,8,0)	12	119.6	1434.6
XC4VSX55	(1056,60,96)	3	486.3	1458.8
	(640,32,64)	5	282.2	1411.2
	(256,16,32)	11	126.8	1394.4
XC4VLX40	(320,8,0)	5	106.4	532.0
XC4VLX80	(1024,32,0)	4	354.4	1417.6
	(640,32,0)	4	267.8	1071.2
	(448,0,0)	12	114.3	1371.6
	(320,16,0)	6	134.0	803.9
	(288,0,0)	12	78.2	938.7
average			206.0	1220.6

Table 5.4: Configuration sizes per module Ψ, with and without relocation.

technique into the existing self-reconfiguration framework provided by Xilinx EDK tools.

Furthermore we have presented an approach to place PR regions for relocatable modules on the device, taking into account heterogeneity and masked-out resources. We also consider communication infrastructures and fragmentation of the residual free space for the static system.

In our case studies we show a software-defined radio prototype with two reconfigurable regions, where the number of partial bitstreams is reduced by a factor of 2 and the design time shortened by 43%. We also present an abstract analysis of relocatability based on region width relative to the device size. It is shown that our technique greatly enhances the relocatability, especially for larger modules.

Finally, we evaluate our placement technique and show that we are often able to find more regions than it would be possible with fully identical regions. In most cases, it is possible to find as many relocatable regions as without relocation. In a case study with signal processing cores we show that PR regions allowing relocation reduce the storage size by a factor of 5.9 on average over a range of device sizes. We also find that equidistant placement of heterogeneous resources simplifies the search for regions allowing relocation. Devices without regular placement of heterogeneous resources hamper relocation or increase the number of masked out resources. Future devices could be designed to help relocatability if an equidistant arrangement of heterogeneous resources is used and symmetry between the heterogeneous resources is maximised.

Chapter 6

Power evaluation

In this chapter we present a technique to evaluate the power efficiency of reconfigurable devices. Power and energy can be reduced with energy-aware optimisation techniques such as the one presented in section 4.6 but the device architecture also has to be power efficient. We therefore introduce an approach to evaluate the power efficiency in fine-grain reconfigurable devices. The approach also takes into account device temperature as well as dedicated low-power modes.

During active processing, current FPGAs consume around one order of magnitude more power than ASICs [57]. The gap during inactive periods is even larger. Mobile applications often have active processing interleaved with long periods of inactivity. Hence, dedicated low-power modes are equally important as efficiency during processing. FPGAs without dedicated low-power modes can consume tens to hundreds of milliwatts when they are idle, while standby power requirements for mobile devices are often in the microwatt range. Recently, several devices with dedicated low-power modes have become available (see table 2.2) but these modes vary in their characteristics such as retention of state or wake-up times.

As explained in section 2.10, previous work on low-power FPGAs often focusses on the operational power encountered during active processing but does not include the evaluation of low-power modes. Other aspects that are currently not addressed are the realistic modelling of applications representative of mobile devices, the consideration of peak, average and standby power, as well as thermal aspects in these scenarios. These issues are addressed in the GroundHog 2009 benchmark suite [38]. GroundHog 2009 is a benchmark suite to evaluate the power efficiency of reconfigurable devices for mobile applications. The power evaluation method presented in this chapter is part of GroundHog 2009.

In section 6.1 we give an overview over GroundHog 2009. We outline the basic challenges when benchmarking reconfigurable devices for power and explain how these challenges are addressed. We briefly introduce the seven benchmarks that are part of the benchmark suite. The key contribution in this chapter is a power evaluation method that is also used as one of the seven benchmarks in GroundHog 2009. The other six benchmarks are not original contributions to this thesis. GroundHog 2009 represents work that was conducted together with Peter Jamieson[1] and with additional contributions from Tero Rissa[2]. We also thank David Thomas[3] for supplying the random number generator that is used in our benchmark circuit. The basic considerations of FPGA power benchmarking that are covered in section 6.1 are based on joint work on GroundHog 2009. The material in the rest of this chapter represents work that was primarily developed and implemented by the author.

Section 6.3 outlines the details of this benchmark. The key concept of this benchmark is to evaluate the power efficiency of a reconfigurable fabric in different activity modes. Activity modes include active processing, inactivity as well as dedicated low-power modes. Furthermore, we cover thermal aspects. Heat has to be considered in mobile devices even though they are generally thought of as "low power". Because of their dense packing and limited air circulation or thermal conductivity of their cases, they too can encounter thermal problems. Hence, we characterise how the device heats up for a given amount of activity, and we analyse the feedback effect of temperature on power.

In section 6.4, we explain how our benchmark can be implemented and highlight practical aspects of FPGA temperature measurements. We perform the benchmark on five commercial FPGAs. This is shown in section 6.5. The tested FPGAs cover a range of process technologies. One FPGA features a dedicated-low power mode and another one is optimised for low static power. Section 6.6 provides a summary of this chapter.

6.1 Low-power benchmarking for FPGAs

GroundHog 2009 is a low-power benchmarking suite for reconfigurable devices and specifically addresses the requirements of mobile applications. The motivation for this is twofold: we want to provide means to evaluate current devices in terms of their power efficiency for mobile scenarios. Furthermore, we want to provide a basis to conceive and develop improvements in future devices.

FPGA benchmarking in general is more challenging than a processor benchmark. Processor benchmarks often provide an executable that performs a number of operations and

[1]Imperial College London, London, UK; *now at:* Miami University, Oxford, OH, USA
[2]Nokia R&D, Tampere, Finland
[3]Imperial College London, London, UK

reports results and statistics to the user. The equivalent of an executable for an FPGA would be a configuration bitstream, but FPGA configurations are specific to each device type. Configurations for all possible devices would have to be made available. Another issue is the non-uniform IO infrastructure on various boards: the same type of FPGA can have different interfaces on different boards and systems. Hence, we cannot provide such a simple equivalent to an executable. Providing a benchmark in form of synthesisable HDL code is also difficult. Some features may not be synthesisable across various FPGA architectures or they may synthesise with varying efficiency. Thus, such a benchmark would, at least to some degree, measure how well the HDL description fits the targeted architecture rather than how good the FPGA is. Another problem is that FPGAs range widely in their size and cannot be compared fairly with a single design. FPGA benchmarks will always require some form of designer interaction since no simple executable or synthesisable description can be used. The challenge is to specify the benchmark in such a way that the effort or skill of the designer will have no major impact on the result.

Another challenge is to specify a meaningful test case. Previous work has often used the MCNC benchmark suite [138] for FPGA power evaluation. MCNC provides a range of simple circuits such as state machines and combinatorial logic specified in RTL. There are several reasons why such a collection of circuits is not adequate for power benchmarking. First, it is unclear how the results based on these simple circuits are relevant to any particular application. Secondly, RTL-level descriptions may map to different architectures with varying efficiency. And finally, these circuits are not combined with input stimuli and results may change depending on what inputs are used.

Finally, one faces the question of how specific a benchmark and its environment should be. For example, one could strictly define a test design and also require a particular environment in which this test should be performed. Examples of environmental parameters are clock speeds, supply voltages, IO interfaces or ambient temperature. A strictly defined environment has the advantage that results of different benchmark users can easily be compared. The downside, though, is that this particular environment might not be a meaningful scenario for the benchmark user. A flexible environment on the other hand allows users to adapt the benchmark to their requirements, but this may mean that it is more difficult to compare results.

These issues are addressed in GroundHog 2009. The goals are to eliminate the influence of the designer as much as possible, to avoid design descriptions that map to some architectures more efficiently than others, and provide a general methodology on how to setup and measure the device, and how to report results. Since meaningful evaluation of devices is more important than global comparison and ranking, we allow flexible environments.

The GroundHog 2009 benchmark suite consists of seven benchmarks. The designs are:

- GH09.B0 - Fabric analysis (described in this chapter)
- GH09.B1 - Port expander and keypad controller
- GH09.B2 - Glue logic
- GH09.B3 - AES encryption cipher
- GH09.B4 - Lempel-Ziv data compression
- GH09.B5 - Bridge chip
- GH09.B6 - 2D convolution

GH09.B1 to GH09.B6 are application specific benchmarks and represent functionalities commonly found in mobile devices. They also cover a range of characteristics such as arithmetic complexity, performance requirements, bit-level or byte-level operations and data-flow oriented or control-flow oriented computation. Designs are specified on a functional level in order to allow them to be targeted to wide range of devices and to avoid the issues with synthesisable HDL code described above. The benchmarks also contain several input stimuli with varying parameters: amount and rate of activity and length of inactive periods. This variation is important because the same amount of activity spread over time, or bundled into short bursts with long breaks in-between can point out different power characteristics.

These benchmarks can be used in user-defined environments since we want to allow a broad range of evaluations. For example, in a comparison of two devices, one might be the better choice for low-activity peripheral tasks while the other can be more efficient for computationally complex baseband or application processing with high clock frequencies. Hence, the benchmark does not produce an absolute ranking of devices. It is intended as a tool to evaluate devices according to the requirements of a particular scenario. The benchmark user is free to use any of the benchmarks and define an environment such that it creates an appropriate test case for their purposes. The benchmarks GH09.B1 to GH09.B6 are not further covered in this chapter but they are explained in detail in [48].

GH09.B0 differs from the other benchmarks in that it is a synthetic, application-independent test case that can be used for fast and simple evaluation of the device's power characteristics including dedicated low-power modes. We use a dense random number generator circuit with no potential for logic optimisation, or optimised placement and routing. This circuit is characterised by its very high activity and does not require any other input stimuli than the clock. The motivation for this is to provide a simple, initial classification of the power efficiency of the device. The classification can reflect realistic high-power or low-power scenarios a device could be operated in. Another important aspect is thermal characteristics such as the temperature dependency of power and heating up of the device under different processing scenarios. The proposed methodology should also be able to capture the improvements in future devices with new low-power techniques.

Even though it is not the primary purpose of the GroundHog 2009 benchmark suite to perform general device comparisons, GH09.B0 can also be used in such a way to evaluate the power efficiency of current commercial FPGAs, the effectiveness of low-power modes as well as general trends in technology. In order to perform such a comparison, some benchmark parameters have to be defined. GH09.B0 has two parameters that can be modified: *clock frequency* and *logic utilisation*. In the following we describe an implementation of GH09.B0 with clock frequency and logic utilisation chosen such that they represent a reasonably challenging test case. Benchmark users are encouraged to use the same parameters, but parameters can be changed if a user feels that they are inappropriate for their purposes. Comparisons are then still possible by scaling and normalising the results.

6.2 Power consumption in CMOS devices

The total power consumption in CMOS devices is a combination of static and dynamic power. Static power is caused by leakage currents inside the device while dynamic power is caused by switching activity.

Static power has two main components: gate leakage and sub-threshold leakage, with the latter being the most significant component [80]. Gate leakage is a leakage current from the transistor gate through the gate oxide into the substrate. Sub-threshold leakage or source-to-drain leakage is a leakage current from the transistor source to the drain when the transistor is turned off and the gate voltage is below the threshold voltage. Lowering the threshold voltage increases performance but causes sub-threshold leakage to grow exponentially. The sub-threshold leakage also increases exponentially with the junction temperature. Static power used to be a minor component of the total power consumption but lower threshold voltages in modern CMOS devices have led to a growth of this component. Because of its temperature dependency static power can also increase through feedback effects from the overall power dissipation. In a worst case scenario it can lead to a thermal runaway. Static power can be addressed by new techniques in process technology such as high-k gate oxides that allow thicker gate oxides for equal performance. Static power during inactive periods can be addressed by power gating that can turn an inactive circuit off. FPGA-specific techniques include using thicker gate oxides for non-performance-critical transistors such as configuration memory cells or using programmable back-biasing voltages so that non-critical path logic can operate slower and with less power. Static power has a strong dependency on the variation of process parameters [15]. Hence, the static power profile of a device can vary from die-to-die, wafer-to-wafer and lot-to-lot.

Dynamic power on the other hand is caused by a combination of charging and discharging load capacitances as well as short-circuit currents when transistors switch. Charging and discharging capacitances is usually the dominant effect [80]. Dynamic power is given by equation 6.1. It has linear dependency on the clock frequency f and the capacitance C, and a quadratic dependency on the supply voltage V. In general, the capacitance is defined by all transistors in the device. In an FPGA however, it depends on the number of logic and routing elements used in a particular configuration. The factor α is the activity or toggle rate and depends on the design and its input stimuli. Dynamic power is usually independent of temperature and also does not depend as strongly on process variation as static power.

$$P_{dynamic} = \alpha \cdot C \cdot V^2 \cdot f \tag{6.1}$$

Dynamic power is improved by reducing the transistor capacitances through feature scaling and reducing the voltage through voltage scaling. Additional improvements can be made by reducing the switching activity in the device. For example, the clock frequency can be scaled according to processing requirements or the clock can be stopped completely during inactive periods.

6.3 Fabric characterisation method

In section 6.1 we showed that FPGA benchmarking is fundamentally different and more challenging than a processor benchmark. For this particular benchmark we find the following challenges. The method should:

- Be applicable to a wide range of devices.

- Be a fair comparison and results free from implementation tool influences or hand-optimisations.
- Cover different power modes and possible future techniques that are currently not available.

The basic concept of our method is to implement a highly active circuit on the FPGA and scale the circuit size with device size such that high logic utilisation is achieved. The power is then measured in several activity modes. Rather than measuring static and dynamic power directly, we characterise the device based on the power that it consumes in a certain mode. The activity modes represent active processing, inactivity and dedicated low-power states. We also want to characterise how the device heats up under active processing. This methodology represents worst-case and best-case scenarios and characterises the power and temperature envelope of the device. More application specific-measurements can be achieved with benchmarks GH09.B1 to GH09.B6.

The benefit of this characterisation is that it allows us to assess the general suitability of a device for a low-power design. Moreover, it can be used to compare and optimise devices for lower power. The key aspects of our method can be summarised as follows:

- Use random number generators (RNGs) as test circuit with high activity.
- Scale the size of the test circuit to the device size. Use 90% of the logic resources in the device.
- Run the test circuit at a fixed clock rate of 100MHz when active.
- Specify the behaviour of activity modes and measure power in these modes.
- Measure temperature in active mode.
- If device heats up 35°C, perform additional measurements based on various duty cycles of activity.

To create a worst case active processing scenario, we use pseudo random number generators as test circuits [102]. These random number generators are based on binary linear recurrences where each bit of the next state is generated based on a linear combination of the current state. Compared to linear feedback shift registers (LFSR), the most common type of random number generators, this improves quality of the random numbers. For our purposes, the main advantage is the lack of optimisation potential in the circuit. This will minimise the influence of the implementation tools on the result of our characterisation. Using binary linear recurrence yields a very dense circuit where each RNG state flip-flop is fed by a LUT which in turn has its inputs connected to some other state flip-flops. An n-bit RNG maps to exactly n LUTs and n flip-flops. This circuit does not provide any potential for logic optimisation and thus, eliminates the influence of the synthesis tools. The circuit is also characterised by LUTs that are heavily interconnected to seemingly random points. It does not provide any opportunity for optimised placement and routing other than concentrating the RNG circuit to the smallest possible area. It also results in an implementation that will exercise all different kinds of short and long wires of the routing fabric. The random number generator circuit is also characterised by a high and uniformly distributed toggle rate, and therefore suitable to act as worst-case scenario of maximum activity in the FPGA fabric. The statistical chance of toggling on the rising clock edge is 50% for all flip-flops in the circuit. Hence, the total toggle rate of the circuit is 50%.

We use a 512-bit random number generator core that maps to exactly 512 LUTs and 512 flip-flops. Since current FPGAs provide tens to hundreds of thousands of LUTs and flip-flops, we scale the size of the test circuit to the size of the device. To achieve high logic utilisation while still allowing routability, we implement multiple instances of the random number generators so that 90% of all logic resources are used. The resulting power consumption is normalised to the number of LUTs in order to allow a comparison of differently sized chips.

The cores are driven by a 100 MHz clock when the circuit is active. This represents a intermediate clock frequency that is commonly found in FPGA designs and is therefore useful as a reference point for basic evaluation. Some benchmark users may want to use other clock frequencies as high-performance designs can run in excess of 300 MHz while 32 MHz is a typical frequency in mobile applications. Comparisons to measurements based on other clock frequencies are still possible by normalising the results to clock frequency.

To enable a comparison of devices with different low-power capabilities, we define the behaviour of activity modes. These activity modes specify how the device behaves in a certain mode rather than the means by which this mode is implemented. The two basic modes that are applicable to all devices are *active* and *inactive*. In *active* mode, the test circuit continuously generates random numbers at 100 MHz clock frequency. The power consumption in this mode is a combination of static and high dynamic power. In *inactive* mode, the circuit does not generate random numbers, but its state is preserved and it

	activity mode			
	standard		device-specific	
	active	inactive	sleep	hibernate
generate output	yes	no	no	no
retain state	-	yes	yes	no
wakeup time	-	instant	$500\mu s$	$50ms$

Table 6.1: Examples of activity modes. The first two modes are fixed, further modes can be defined based on the device capabilities

can be instantly brought back into active mode. These are the most basic activity modes and a transition between these two modes can usually be implemented with a simple clock gating approach as illustrated in figure 6.1. However, we are only concerned about the power profile of the inactive device with preservation of state and instant wake-up capability, and not the details of its technical implementation. If clock gating is chosen to implement this mode, it will show static power only. In other cases, there might be supporting circuitry such as clock managers that are still operating and drawing dynamic power. It is important to point out that in this context, we are not interested in measuring pure static power but rather the minimal power to implement the *inactive* mode.

In order to evaluate devices with advanced low-power modes such as the ones in table 2.2, we characterise the behaviour of the low-power mode and compare the power consumption with the two basic modes. Table 6.1 illustrates an example with our two basic activity modes *active* and *inactive*, and two hypothetical advanced modes *sleep* and *hibernate*. The behaviour of the basic modes is fixed, while the behaviour of the advanced modes depends on the device capabilities.

We are also interested in the temperature that the device can reach in active and inactive mode, as well as the temperature influence on power. Thermal characterisation of a device usually involves determining the thermal resistance of the device between die (also called junction) and case or junction and ambient air. For this purpose, a special test die is usually mounted in a case and heated up to a defined junction temperature T_j by applying power P. After measuring the case temperature T_c the junction-to-case thermal resistance ϑ_{jc} can then be calculated as follows:

$$\vartheta_{jc} = \frac{T_j - T_c}{P} \tag{6.2}$$

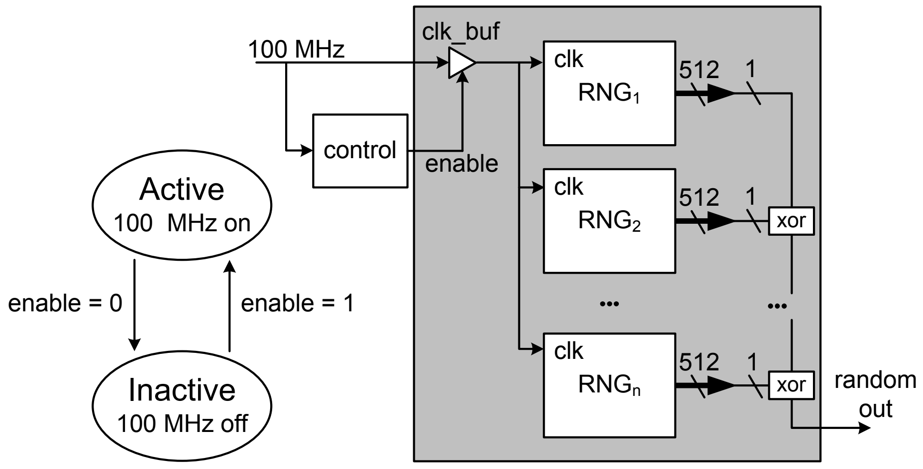


Figure 6.1: Fabric characterisation circuit with random number generators.

It is often difficult to measure the junction temperature directly. We therefore use case temperature measurements which can be taken easily. Case temperature is also more representative of how the chip affects its environment. In order to reduce external influences on the measurement we use the following environment: a chip is mounted on a PC board and surrounded by ambient air with a temperature of 25℃. The board is placed in a large open cardboard enclosure which is supposed to reduce airflow to natural convection and reflect infrared radiation. No heatsinks or active cooling systems are to be used. We can also calculate the junction temperature based on the measured case temperature using equation 6.2.

In the following, we distinguish between *cold* and *hot* devices in our characterisation. Devices which do not exceed a surface temperature of 35℃ when being in active mode are characterised as *cold*. In this case, the influence of temperature is small. We record the device's low temperature T_{min}, when the device is constantly in inactive mode, and the device's high temperature T_{max} when it is constantly in active mode. We measure active power when the device has reached T_{max} and we measure inactive power at T_{min}. Dedicated low-power modes are also measured at T_{min}. Power results are reported as absolute values and normalised to LUTs. Inactive power is normalised to total LUTs in the device, and active power to used LUTs in the test circuit. Activity related power can also be reported as normalised to clock frequency if one wants to compare different clock frequencies. This is the difference between active and inactive power divided by number of used LUTs and clock frequency:

$$P_{f,nrom} = \frac{P_{active} - P_{inactive}}{LUTs \cdot f} \tag{6.3}$$

For *hot* devices which exceed a surface temperature of 35°C, we measure active power at T_{max}, and we measure inactive power for T_{min} and T_{max}. Inactive power at T_{min} represents the power that the device consumes when it is inactive mode for a long time and hence, has cooled down. Inactive power at T_{max} is the power that the device consumes when it just entered the inactive mode after active processing. Additionally, we use a more detailed analysis that is based on adjusting the amount of activity in the device. We want to analyse how a device heats up for a certain amount of activity, and measure how this influences power. The activity in the device is adjusted by periodically switching the device between active and inactive state with various duty cycles. For each duty cycle we measure instantaneous active power, inactive power and temperature. To take these measurements, we first wait until the temperature has converged to its final value for this particular duty cycle as illustrated in figure 6.2. We record this temperature and then take a reading of the instantaneous active power and inactive power. As mentioned earlier, inactive power does not necessarily have to be equivalent to static power and can have dynamic components as well. Nonetheless, we expect a strong temperature dependency on inactive power because of its close relation to static power. Likewise, we record the instantaneous active power that we expect to be less temperature dependent, since it is largely based on dynamic power. These measurements are taken for duty cycles from 0% to 100% with 5% increments. Results for temperature and normalised active and inactive power are reported over duty cycle. For devices with additional low-power modes, the power in these modes is measured in ambient temperature without any duty cycle.

6.4 Implementation of the fabric characterisation method

We implement our fabric characterisation in Xilinx Virtex and Spartan series FPGAs and one Silicon Blue iCE65 FPGA. For each device, we obtain the number of available LUTs. Then, multiple RNG cores are instantiated such that 90 % of all LUTs are used. This is illustrated in 6.1. The RNG cores only have a clock input and a 512-bit wide output. Each core is initialised with a different seed so that logic optimisation tools cannot combine several cores into one. One output pin of each core is connected via an XOR chain so that the cores cannot be removed by logic optimisation. The generated random numbers are not used otherwise. The following synthesis and physical implementation of the random number generator circuit is straightforward. After placement and routing, the reported

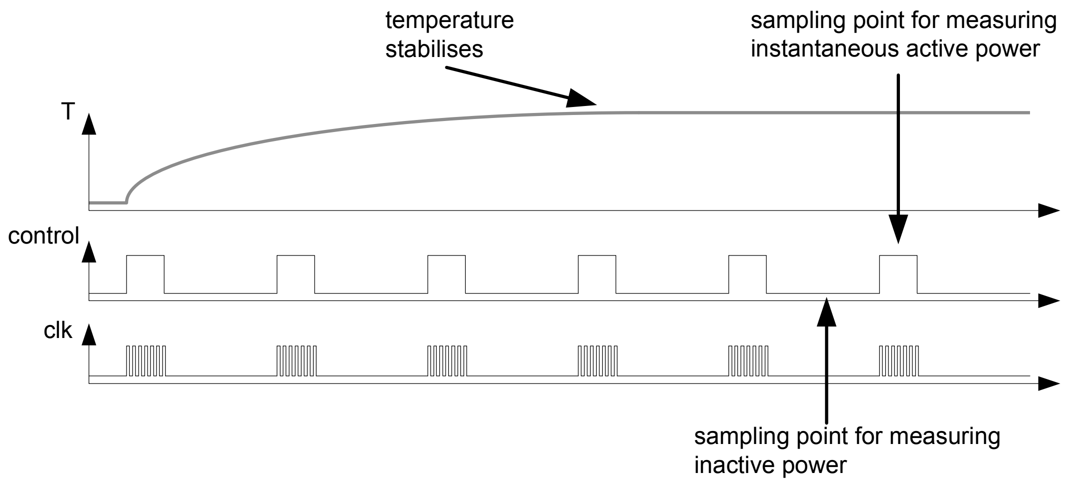

temperature stabilises
sampling point for measuring instantaneous active power
T
control
clk
sampling point for measuring inactive power

Figure 6.2: Depiction of power measurements on *hot* devices.

LUT usage of the implementation tools should match the expected value of $512 \cdot n$, where n is the number of instantiated RNGs. The timing report should also be checked if the required clock frequency of 100 MHz has been achieved.

Xilinx Virtex series and Spartan-3E FPGAs do not support any low-power modes other than simply stopping the clock. The most efficient way of setting the device into inactive mode as specified in section 6.3 is to disable the clock tree using an internal clock buffer. This buffer is located at the root of the clock tree and therefore reduces the dynamic component of inactive power to a minimum. The enable signal of the clock buffer is connected to an external pulse generator with variable duty cycle. Spartan-3A and Spartan-3AN FPGAs also feature an advanced low-power mode called *suspend* [126]. This mode is controlled by an external signal pin and does not require modifications to the logic design itself. Silicon Blue iCE65 FPGAs provide an *iCEgate* latch which will stop input signals from propagating into the device. In our case we use the iCEgate as a clock buffer, which is again controlled by an external pulse generator.

For power measurements we use the technique explained in section 4.7. Power measurements are based on core power and do not include IO banks. All boards except the Xilinx ML505 board provide jumpers in the FPGA's supply rails that can be used for power measurements. For measurements on the Xilinx ML505 board, we again use our board that was modified to allow core power measurements.

We use an Optris MiniSight Plus infrared thermometer to determine the surface temperature of the device. This contactless method minimises the influence of the measurement on the system. Infrared thermometers measure the thermal radiation of an object and work very well on matt, black cases. However, they are inaccurate on shiny metal surfaces since these emit less infrared radiation. We therefore apply a thin blackened aluminium plate to devices with a metallic surface.

6.5 Results and measurements

We perform a fabric characterisation as described in section 6.3 on five commercially available FPGAs. Our measurements cover four Xilinx devices and one Silicon Blue device as listed in table 6.2. The tested devices vary notably in their process technology and core voltage and thus have significantly different power profiles. Table 6.2 also shows the logic capacity of each device, the number of RNG cores used and the resulting device usage which should be as close as possible to 90%. Both Spartan devices have less than half of the logic capacity of the Virtex devices. The Silicon Blue device is the smallest device with significantly less logic capacity than all the others. The Spartan devices have plastic packages which have higher thermal resistances. The thermal resistance for the Silicon Blue device is not specified in the data sheet. The designs targeting Xilinx device are implemented with Xilinx ISE 9.2. The design for the Silicon Blue FPGA is implemented with Silicon Blue iCEcube 2008 development software.

Table 6.3 shows minimum and maximum temperature as well as inactive and active power for all devices. The table lists the minimum case temperature the device reaches when constantly being inactive and the maximum temperature the device reaches with full duty cycle of active processing. Also shown is the maximum junction temperature that is calculated based on equation 6.2. Inactive power is shown at minimum temperature for a device that is held in an inactive state for a longer period of time and hence, is cooled down. Inactive power at maximum temperature represents the case where the device is just switched off from active processing. The table also lists total and normalised active power for maximum temperature and the clock-normalised activity related power that can be used for comparisons with other clock frequencies.

Virtex-II Pro is the only device that cannot be run at full duty cycle without overheating. At 60% duty cycle, the device reaches a surface temperature of 74℃. Based on the thermal resistance and the power consumption, we calculate that this surface temperature corresponds to a junction temperature 81℃. This is close to the maximum allowed junction temperature of 85℃ and the device cannot be run at higher duty cycles without the risk of being damaged. Even though the device has a ceramic package with low thermal resistance, the overall high core power leads to a significant difference between case and junction temperature. All other devices can be run at full duty cycles without overheat-

FPGA family	device	board	process technology	core voltage	thermal resistance ϑ_{JC}	number of LUTs/FFs	number of RNGs	logic utilisation
Xilinx Virtex-II Pro [130]	XC2VP30	XUP	130nm	1.5 V	0.6 °C/W	27,392	48	89.7%
Xilinx Spartan-3E [125]	XC3S500E	Spartan-3E Starter Kit	90nm	1.2 V	9.8 °C/W	9,312	16	88%
Xilinx Spartan-3AN [125]	XC3S700AN	Spartan-3AN Starter Kit	90nm	1.2 V	5.3 °C/W	11,776	21	91.3%
Xilinx Virtex-5 [128]	XC5VLX50T	ML505	65nm	1.0 V	0.2 °C/W	28,800	50	88.9%
Silicon Blue iCE65 [96]	iCE65L04	iCEman Eval Kit	65nm	1.2 V	n/a	3,520	6	87.3%

Table 6.2: Details of FPGAs used for our fabric characterisation experiment.

device	temperature			power						
	case		junction	inactive mode				active mode		
	T_{min}	T_{max}	T_{max}	$P_{total}(T_{min})$ $[mW]$	$P_{norm}(T_{min})$ $[\mu W/LUT]$	$P_{total}(T_{max})$ $[mW]$	$P_{norm}(T_{max})$ $[\mu W/LUT]$	$P_{total}(T_{max})$ $[W]$	$P_{norm}(T_{max})$ $[\mu W/LUT]$	$P_{f,norm}(T_{max})$ $[\mu W/LUT \cdot MHz]$
XC2VP30	26°C	74°C	81°C	24.0	0.88	216.0	7.89	11.190	455.3	4.54
XC3S500E	26°C	46°C	56°C	19.2	2.06	26.2	2.81	1.022	124.8	1.22
XC3S700AN	26°C	47°C	54°C	18.7	1.59	28.8	2.45	1.363	126.8	1.25
XC5VLX50T	30°C	54°C	55°C	349.0	12.12	412.1	14.31	2.545	99.4	0.86
iCE65L04	25°C	29°C	29°C^1	0.13	0.037	0.13	0.037	0.324	105.5	1.05

(1) estimated, case thermal resistance not reported

Table 6.3: Temperature and power measurement results. Shown are total and normalised inactive and active power as well as frequency normalised activity related power.

ing. The Spartan devices have a plastic case with high thermal resistance and therefore also show a significant difference between case and junction temperature. Despite the high thermal resistance, Spartan devices do not overheat because the overall power is low enough. The Virtex-5 FPGA has a metallic package with very low thermal resistance. Hence, there is almost no difference between case and junction temperature despite moderate core power. The Silicon Blue iCE65 FPGA does not exceed a temperature of 29°C. The thermal resistance of the package is not specified but because of the very low core power we estimate that the junction temperature is roughly equal to the case temperature. There is no measurable feedback of temperature on inactive power and the device is treated as a cold device. For all Xilinx devices, we can observe a temperature feedback of the active processing on inactive power. For Virtex-II Pro, we find an ninefold increase of inactive power between minimum and maximum temperature.

The normalised active power of the iCE65 device is similar to Virtex-5 which is manufactured in the same process technology. Normalised inactive power is significantly lower in the iCE65 FPGA. The inactive mode in the iCE65 device is implemented using iCEgate latch feature which can freeze the state of an IO bank. Outputs are kept at their current state and input signals do not propagate into the device. Internal switching activity is therefore stopped completely. This is not a dedicated low-power state by itself, but contributes to good power efficiency when being inactive. The inactive power of the iCE65 FPGA is significantly better than all other devices which can be explained by the prevention of internal switching activity by the iCEgate and the fact that the device is specifically optimised for low leakage. Optimising for low leakage however comes at the expense of performance. It is challenging to achieve timing closure for our test design on the iCE65 FPGA while all other Xilinx devices provided good performance headroom.

In Xilinx devices, we can also observe an improvement of normalised active power with modern process technology while at the same time, inactive power deteriorates. Since there is a temperature feedback on power in Xilinx device we further analyse these devices using our duty cycle measurement method.

Figure 6.3 shows the surface temperature of the Xilinx devices over the duty cycle. The temperature is measured with an infrared thermometer as explained in section 6.4. For all devices the temperature increases almost linearly. Virtex-II Pro has the steepest slope and reaches a surface temperature of 74°C at 60% duty cycle. Spartan-3E, Spartan-3AN and Virtex-5 reach a surface temperature of 46°C, 47°C and 54°C respectively when on full duty cycle.

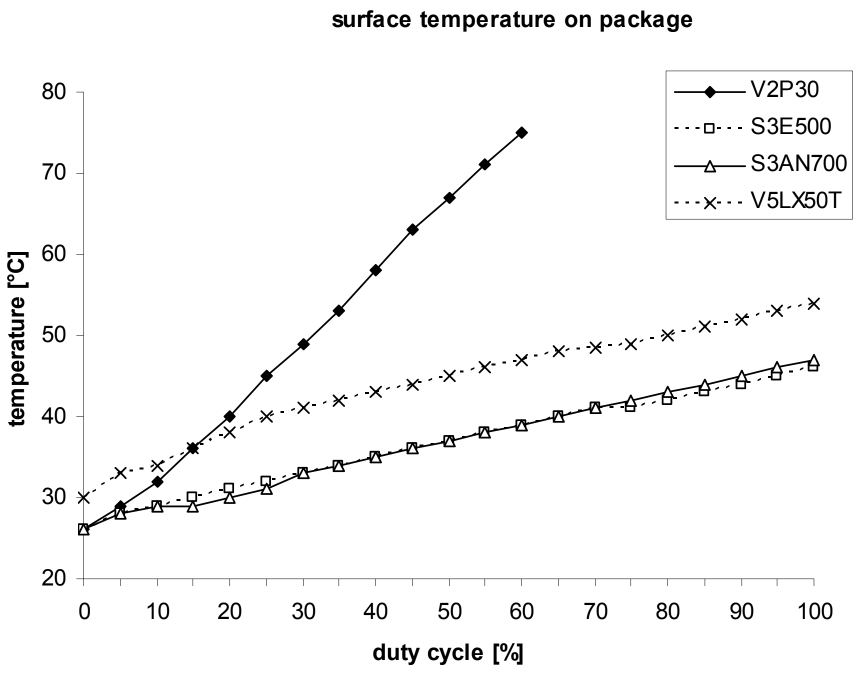


Figure 6.3: Variation of device surface temperature with duty cycle.

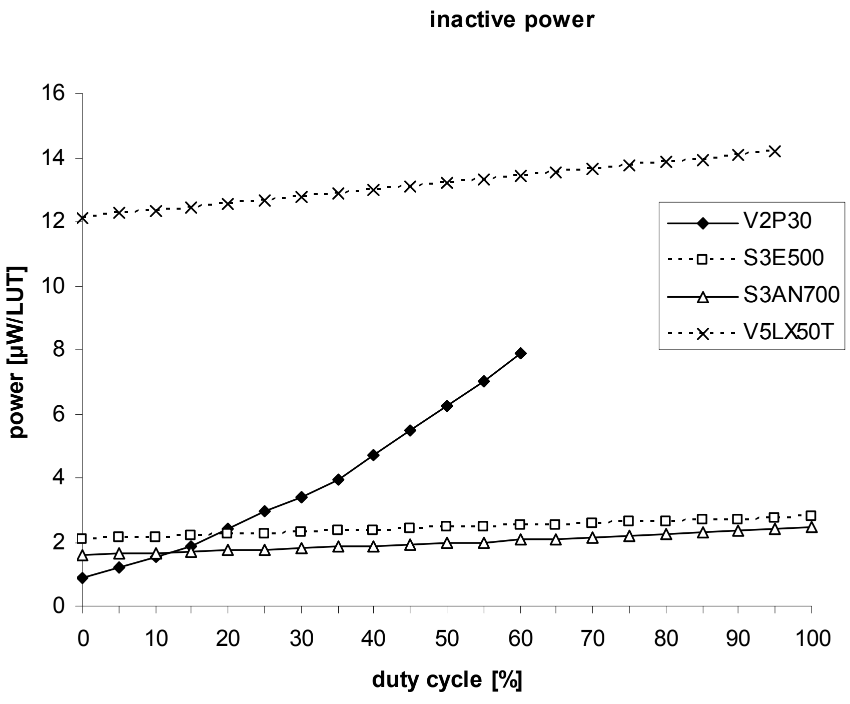


Figure 6.4: Variation of inactive power consumption with duty cycle.

Figure 6.4 illustrates the inactive power consumption normalised per LUT over duty cycle. Each point on the line is measured when the temperature of the FPGA has stabilized for a given duty cycle as illustrated in Figure 6.2. We observe that inactive power

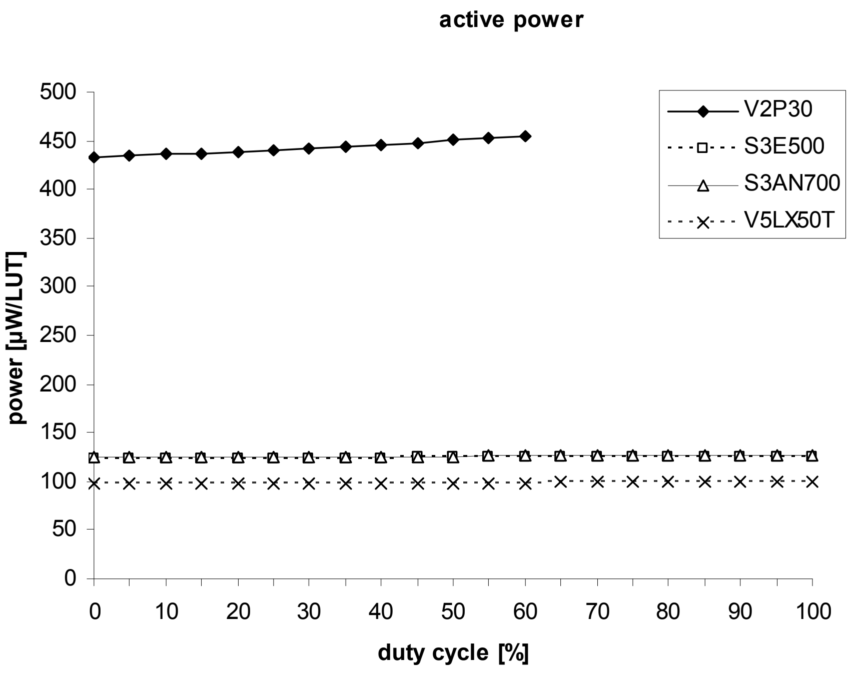


Figure 6.5: Variation of instantaneous active power consumption with duty cycle.

consumption, which in our implementation is almost equivalent to static power, increases with the duty cycle. This is due to the rising temperature of the device caused by heat dissipated during the active part of the duty cycle. The Virtex-5 device, which is manufactured in 65 nm process, has a 12 times higher inactive power consumption under cold conditions (0% duty cycle) than the 130 nm Virtex-II Pro. This can be explained with worsened static power in the smaller process technology but unused hard IP blocks may also have an influence. The inactive power in Virtex-II Pro deteriorates quickly with higher duty cycles because of the high device temperature as illustrated in figure 6.3. Virtex-5 and the Spartan-3 devices develop less heat which leads to a less progressive increase in inactive power. The Spartan devices, which are manufactured in a newer process technology than Virtex-II Pro, show a higher static power consumption for low duty cycles which corresponds to low temperatures. At higher duty cycles, however, the lower device temperature (see figure 6.3) leads to lower inactive power compared to the older process technology. Of all four Xilinx FPGAs, Spartan-3 devices show the overall best efficiency during inactive phases.

Figure 6.5 shows the normalised active power consumption for all Xilinx devices. This is the instantaneous active power during on-phase of the duty cycle as illustrated in figure 6.2. We can observe a notable improvement in active power for newer devices which is due to feature and voltage scaling in the process technology. The improvement from Virtex-II Pro (130 nm) to Spartan-3 (90 nm) is especially noteworthy. Compared to Virtex-5, the active power is reduced by more than a factor of 4. The active power consumption is relatively independent of duty cycle and temperature, although this could change if static power becomes a more dominant component in active power. In current devices, we find that inactive power is considerably less than active power although the ratio increases from 0.2% to 14% between Virtex-II Pro and Virtex-5 in cold conditions, and from 1.5% to 16% in hot conditions. On the other hand, devices can be optimised for low leakage by trading off performance. The Silicon Blue iCE65 device for example has an inactive to active power ratio of 0.04%.

power	active mode	inactive mode	suspend mode
$P_{int}\,[mW]$	1349	18.7	18.1
$P_{aux}\,[mW]$	44	43.6	5.8
$P_{total}\,[mW]$	1393	62.3	23.9

Table 6.4: Total core power in a Xilinx Spartan-3AN 700 FPGA for active, inactive and suspend modes. All values are measured at 25°C.

The iCE65 FPGA provides a means of implementing a very efficient inactive mode, while the Spartan-3AN is the only device that features an additional dedicated low-power mode. This mode is called a suspend mode and it reduces the power consumption of all auxiliary circuits powered on the VCC_{aux} rail [126]. The logic state is preserved during suspend mode and the wake-up time ranges between 100 μs and 500 μs. Table 6.4 illustrates the core power consumption in all modes. Compared to the inactive mode, the suspend mode reduces the power consumption by a factor of 3.

In an overall comparison, the Silicon Blue iCE65 FPGA is the most power efficient device. It consumes slightly more active power than Virtex-5 but significantly less inactive power than all other devices. However, it has the lowest logic capacity and was specifically optimised for low leakage, which has a negative impact on performance. The device performance, which is application specific, is not addressed in this fabric characterisation method but can be evaluated using the additional power benchmarks presented in [48]. From Xilinx devices, we can also observe a general trend of reduction of active power with advancing process technology while inactive power is becoming more challenging. Dedicated low-power modes are needed to address the issue of power consumption during periods of inactivity. Spartan-3AN is the only device in our experiment that features a low-power mode. The inactive power reduction by a factor of 3 represents a good improvement but a power consumption of $23.9mW$ in suspend mode is too high for most power budgets in mobile applications.

6.6 Summary

In this chapter we consider the power efficiency of FPGAs. FPGAs are compelling devices with regard to their performance and flexibility, but their current power demands bar them for being used in most mobile applications. Mobile applications are characterised by periods of inactivity, and target architectures for such applications require not only low operational power, but also dedicated low-power modes.

To address this challenge, the GroundHog 2009 low-power benchmark suite is developed. We illustrate the principal challenges when benchmarking FPGAs and explain how these are addressed in GroundHog 2009. Six of the seven benchmarks in GroundHog 2009 are application specific and representative of functionalities commonly found in mobile devices. In this chapter, we focus on a synthetic, application-independent benchmark for characterising the fabric of FPGAs in several activity modes. This benchmark is intended for a fast and simple initial evaluation of the device. A random number generator is used as a highly active test circuit, and devices are evaluated based on active and inactive modes as well as dedicated low-power modes when available. We also measure the temperature that the device reaches in these modes, and analyse the effect of temperature on power. This methodology represents best-case and worst-case scenarios that characterise the power and temperature envelope of the device. This technique is useful for evaluating the suitability of a device for low-power design as well as for comparing and improving existing devices. We provide guidelines for implementing the benchmark and describe procedures for measuring active and inactive power and temperature on FPGAs using a simple experimental setup.

The proposed fabric characterisation is performed for four Xilinx FPGAs and one Silicon Blue FPGA. Our measurements show that the 65 nm Virtex-5 consumes four times less active power than the 130 nm Virtex-II Pro. On the other hand, we observe that in Virtex-5 inactive power is increased by one order of magnitude. The Virtex-II Pro device heats up to the point where it could potentially be damaged and the temperature also has a strong feedback effect on inactive power. The more modern Xilinx devices generate less heat per activity and suffer less from temperature-based deterioration of inactive power. In a comparison between Xilinx Virtex-5 and Silicon Blue iCE65 devices which are manufactured in the same technology node, we find that active power is similar but inactive power is significantly improved in the case of Silicon Blue. This is because the Silicon Blue device is specifically optimised for low leakage, trading off device performance. Another optimisation is the reduction of internal switching activity through a feature called iCEgate. This gate can be switched on and off instantaneously. Another example of architectural low-power improvements can be found in Xilinx Spartan-3AN FPGAs. These devices feature a suspend mode which allows to power down auxiliary clock circuits. This mode reduces power by a factor of three compared to the normal inactive mode. Leaving the suspend mode requires a wake-up time of hundreds of microseconds.

Chapter 7

Conclusions and future work

7.1 Conclusions

Reconfigurable hardware is an emerging technology that can provide appealing benefits in performance, productivity and flexibility. Performance is delivered through dedicated circuits tailored to the demands of the application, similar to in an ASIC. Productivity is increased by moving from a risky and time-consuming chip design process to designing for a pre-fabricated, programmable device which is important since the cost of chip manufacturing is rising exponentially. Since most FPGAs are reprogrammable, it is also possible to exploit this as an advantage over traditional fixed hardware: devices can be reprogrammed after they are deployed, and it is also possible to reconfigure them during the run time of an application.

Software-defined radio is an area that demands flexibility which cannot be delivered through traditional approaches based on fixed hardware. On the other hand, programmable instruction set architectures are usually not powerful and efficient enough for radio applications. FPGAs are an appealing technology for this field because they deliver performance through dedicated circuits while also being flexible. However, to become a viable solution for a wide range of mobile and other consumer applications, performance and power efficiency have to be further improved. Reconfiguration can become a competitive advantage for FPGAs, and especially the use of run-time reconfiguration for achieving improvements in radio systems has not yet been fully explored.

We identify four different opportunities of employing reconfiguration effectively in software-defined radios. Reconfiguration provides compelling benefits, but the design flow is usually complex and partial bitstreams have to be produced and stored. If reconfiguration is used as a frequent dynamic feature of the application, e.g. reconfiguration during a call or data session, then it is also important to consider and balance configuration overheads. Handling these aspects and choosing the correct design approach may be difficult even for an experienced designer. The material presented in this thesis addresses these issues, and we develop techniques to improve design efficiency and productivity.

In chapter 4 we have presented a parametric model that allows us to systematically optimise reconfigurable designs for performance, area and energy efficiency. In our model we use application parameters, target device parameters as well as implementation parameters that result from a particular mapping of the application onto the target device. Based on this model we perform several optimisations. A key aspect is the exploration of parallelism to improve the design. Parallelism leads to fast processing, but a parallel design is also large and requires a long time to reconfigure. Depending on how frequently the design is reconfigured, a fully parallel implementation may therefore not be the optimal solution. Our aim is to provide an approach for exploring the effect of parallelism on performance. However, building a range of designs in order to find the fastest option would be a time-consuming process. Instead, we propose simple optimisation steps that predict the optimal degree of parallelism based on the parameters of one initial implementation. In practice, some designs may not scale exactly as assumed in our model. For such cases, we analyse how parameter variations influence the design, and give simple guidelines about how the results of the optimisation can be refined. We can also explore the degree of parallelism to reduce the energy per computation. On the one hand, parallelism reduces the energy consumed during processing by minimising the impact of power overhead that is not directly related to computation. On the other hand, the large area that needs to be reconfigured also incurs a high configuration energy overhead. This leads to a similar optimisation problem to performance optimisation. We adapt our optimisation steps for energy efficiency, and optimise the design based on performance and power parameters of one initial implementation.

We perform an extensive case study of this method on a reconfigurable FIR filter and a reconfigurable Cordic processor. We find that a reconfigurable FIR filter is 82% faster and 64% smaller than a non-reconfigurable version, but the configuration time overhead needs to be considered if the filter is reconfigured frequently. For frequent reconfiguration, high reconfiguration speeds are essential to make the reconfigurable design competitive. To improve energy, reconfiguration needs to be fast, but the reconfiguration mechanism also needs to have low power consumption. In our case study we find that reconfiguration can improve both performance and energy efficiency, but the potential for energy improvements is higher because energy efficiency benefits from both faster processing as well as lower power consumption during processing. The reconfigurable version of the filter can be up to five times more energy efficient than the non-reconfigurable version. We also demonstrate how performance, area and energy can be improved by exploring the degree of parallelism. For example, a CORE Generator implementation of our reconfigurable filter can be optimised by choosing an intermediate degree of parallelism. This design uses 47% less energy than the fully parallel reconfigurable design, and 49% less energy than non-reconfigurable design. Our technique is also useful for designs that do not scale exactly according to our model and we show how the results can be refined. We also examine a reconfigurable Cordic processor and find that it mostly scales according to our model.

The influence of parallelism on the optimality of the design provides an exciting new insight into the fundamental aspects of reconfigurable designs. Our analytical model is a valuable addition to the existing models, and is straightforward to use for iterative signal processing applications. The same principle may be used in other application area as well, and provides many opportunities for future work.

In chapter 5 we have presented a method that increases the relocatability of partial bitstreams. Relocatable bitstreams have the advantage of reducing compilation times as well as configuration storage requirements. Instead of having to produce and store a bitstream for every possible target location, only one bitstream is necessary with relocation. However, current standard design tools do not support relocatable bitstreams. Relocation is also hampered by heterogeneity in the device architecture and previous work has solely focussed on enabling relocation between identical PR regions. As heterogeneity increases, it becomes increasingly difficult to identify such identical regions. Our approach is not limited to such identical regions. Instead, we identify the compatible subset between two or more regions. Resources are considered to be compatible if they provide the same function and if their mappings to the configuration memory are identical. Non-matching resources are masked-out: they are prohibited from being used and their configuration data are omitted. We demonstrate how this technique can be realised on Virtex-4 FPGAs and present a lightweight software implementation. The functionality of our method is added to the existing self-reconfiguration framework based on Xilinx embedded processors by extending the drivers of the HwICAP IP core. We evaluate our method in a software-defined radio prototype with two PR regions for reconfigurable filters. In this design, our method cuts the storage requirement in half, and reduces the design compile time by 43%. The configuration speed is slowed down by 6% when relocation is performed with our extended software driver. In an abstract analysis of relocatability, we show that the number of relocatable PR regions that can be found on a heterogeneous device is greatly enhanced by our method.

When floorplanning a system with relocatable modules, one has to identify regions that have as few non-matching resources as possible while also including other constraints. If only a few regions have to be placed, a designer may find a viable solution intuitively, but for larger number of regions, such a manual approach may be inefficient. We propose a method to automate the region placement. The key aspect is to identify a set of viable floorplans with regions that satisfy the requirements of the modules and that reduce the number of non-matching resources. We can also factor in other constraints, such as the location of communication infrastructures and the fragmentation of the residual free space. We perform a case study with a range of signal processing cores often used in software-defined radio. When placing these cores on a range of target devices, we find that in most cases relocatability is increased by our method. Only for small modules or very regular target devices we can find an equal number of PR regions where no resources have to be masked out. We also observe that heterogeneous device architectures can be optimised for relocation by placing heterogeneous resources in a regular pattern. This minimises the number of resources that have to be masked out for relocation.

Bitstream relocation is a popular subject in research on run-time reconfiguration, and our relocation technique is an immediate contribution to the field. It can be easily incorporated into other frameworks or adapted for alternative target devices.

Reconfigurability is an appealing aspect of FPGAs that can be used to improve several design aspects, including power and energy. However, mobile applications have strict power constraints that also require architectural innovations. Operational power needs to be low, and during periods of inactivity power needs to be further reduced with dedicated low-power modes. This is addressed in chapter 6. We present a methodology to evaluate the power efficiency of fine-grain reconfigurable devices. This methodology is part of the GroundHog 2009 low-power benchmarking suite and it is intended as a fast, initial and application independent evaluation. We use a random number generator as a highly active test circuit without optimisation potential. This test circuit is used to evaluate the power in active and inactive mode, as well as dedicated low-power modes if available. We measure the temperature that the device reaches in these modes, and we also measure the feedback effect of temperature on power consumption.

We perform our power benchmark on four Xilinx FPGAs and one Silicon Blue FPGA.

These devices cover a range of process technologies, sizes and package types. We observe that the Virtex-5 device consumes four times less active power and twelve times more inactive power than an older Virtex-II Pro device. This seems to be mainly related to the developments in process technology, but other factors such as architectural changes may also play a role. However, advances through process technology alone are not enough to meet the strict power constraints in mobile devices. Furthermore, this will not close the power gap between reconfigurable and fixed hardware, as all architectures can benefit from advances in process technology. We also find that high active power in Virtex-II Pro leads to heating of the device which causes inactive power to deteriorate significantly. Newer Xilinx devices generate less heat per activity, and in the Silicon Blue FPGA, temperature effects are marginal. Furthermore, we compare Xilinx Virtex-5 and Silicon Blue FPGAs which are manufactured in the same process technology node. The Silicon Blue device consumes similar active power but shows significantly reduced inactive power compared to Virtex-5. This optimisation is based on power-performance trade off: using slower and less leaky transistors can substantially reduce inactive power. We also evaluate a dedicated low-power mode in Spartan-3AN FPGAs which allows to switch off auxiliary clock circuits. In our experiment, using this feature reduces the power by a factor of three compared to normal inactive power. However, the power is still in the range of tens of milliwatts which is too high for many mobile applications. To meet mobile power requirements, future devices need further improvements especially with regard to their inactive behaviour.

7.2 Future work

The work presented in this thesis could be extended in several ways, as discussed below.

7.2.1 Design tool development

Our parametric optimisation presented in chapter 4 currently requires manual intervention to derive implementation and device parameters, to perform the subsequent calculations and to change the degree of parallelism in the design. This process could be automated. It is possible to develop a software tool that automatically extracts area and performance parameters from the implementation tool reports. With a database of device parameters, the tool can then perform all calculations and report the optimal degree of parallelism to the designer. Optimising for energy would also require an automated power measurement mechanism. Automating design iterations with changing degrees of parallelism is more difficult. Changing parallelism in the implementation may require manual changes in the hardware design description. However, if parallelism can be specified as a simple parameter, it is possible to create a fully automated tool that reiterates design cycles until the optimal design is found.

7.2.2 Widening scope of applications

We can apply our optimisations to a wide range of components for software-defined radio and other applications, and study the extend that they can be optimised for performance and energy. These case studies could also include a more detailed analysis of the scalability of various component models. Currently we assume that processing time and reconfiguration time scale with parallelism. If this is not the case, we use the trend of the parameter variation as a guideline to refine the optimisation. A deeper insight into the effects that cause components not to scale according to our model could be used to create more detailed models which would capture these effects. On the other hand, we should be careful not to overcomplicate the model. Creating a complex and accurate model and deriving the necessary parameters may be less productive than using a simple model with a few reiterations for refinements.

Our parametric optimisation approach targets signal processing applications which have fixed application parameters, and the design is optimised according to these parameters at design time. The same principles could also be used to optimise a design at run time. For example, a system could be dynamically optimised for changing dataset sizes. To support dynamic optimisation, an automated version of our technique needs to be developed and integrated into the configuration controller. It would also require pre-compiling and storing several design versions. As an alternative, new design versions may be generated dynamically at the cost of extra time and energy overhead.

7.2.3 Extending range of devices

Our relocation technique in chapter 5 is demonstrated and implemented for Virtex-4 FPGAs. This method could be adapted to support a wide range of devices families. This requires us to study the details of the configuration memory architecture, and to adapt the presented mechanisms accordingly. The adapted functionality can then be added to our device driver. One issue that is currently not addressed is relocation to regions that contain IO columns. In principle, our method allows relocation to such a region, but the IO resources in this region would become unusable. It is desirable to develop mechanisms that preserve the functionality of IOs when they are located in a relocatable region.

Our device driver is currently implemented based on Xilinx embedded processors and the HwICAP IP core. However, the configuration speed of the HwICAP core is relatively low. In chapter 4 we show that configuration speed is often crucial for the overall efficiency of a reconfigurable application. Since the speed of HwICAP is not sufficient for our performance and energy optimisations, we consider a recently presented IP core that provides 60 times faster configuration speed [22]. However, our current device driver would not be compatible with this improved core. In order to combine relocation with fast reconfiguration rates, we can adapt our method to work with existing faster configuration cores, or we can develop a custom configuration core that is tailored to deliver the required transformations as efficiently as possible.

It would also be interesting to explore how this technique and device model can be adapted to support coarse-grain reconfigurable devices [42].

7.2.4 Floorplanning

The placement method for relocatable regions presented in section 5.3 could be integrated into a graphical floorplanning tool. It is usually not possible to fully automate floorplanning as some decisions are best made by a designer. For example, choosing a region list and placing the static design requires manual interaction. The productivity of this process can be increased by displaying the results in graphical user interface and allowing the designer to quickly compare or even alter viable solutions. Another useful feature would be allowing the designer to specify constraints in the graphical format. This can be achieved by developing a graphical front end to our algorithm.

7.2.5 Power characterisation

The power characterisation benchmark in chapter 6 currently only measures activity in the fine-grain reconfigurable fabric. The method could be extended to cover coarse-grain reconfigurable devices, heterogeneous resources in fine-grain FPGAs, or the effects of run-time reconfiguration. However, it would be more challenging to find simple functions that provide little optimisation potential for the design tools, while applicable to a range of heterogeneous blocks with varying features.

The measurement techniques can be extended to cover application-specific benchmarks. This capability would enable us to study processing speed per unit of power or per unit of energy consumed.

We could also use our benchmark to measure device variation. The results that are presented in section 6.5 are based on a single device. However, devices are known to vary, and especially static power is strongly influenced by parameter variations. If a larger number of devices are available, then the benchmark can be used to study the variation in these devices. Inactive power and low-power modes are usually dominated by static power and are therefore expected to vary the most. There is also the possibility of using our benchmark to characterise parameter variations within a single device, to explore trends of systematic and random variations.

Glossary

List of abbreviations used in this thesis.

ADC - analogue-to-digital converter

AES - advanced encryption standard

ASIC - application-specific integrated circuit

API - application programming interface

BRAM - block RAM

CLB - configurable logic block

CMOS - complementary metal oxide semiconductor

CORDIC - coordinate rotation digital computer

CPU - central processing unit

DDR - double data rate

DES - data encryption standard

DSP - digital signal processor

FF - flip flop

FFT - fast Fourier transform

FIR - finite impulse response

FPGA - field-programmable gate array

GPP - general purpose processor

HDL - hardware description language

HwICAP	- hardware ICAP
ICAP	- internal configuration access port
IF	- intermediate frequency
IIR	- infinite impulse response
IO	- input/output
IP	- intellectual property
MUX	- multiplexer
NCO	- numerically-controlled oscillator
NRE	- non-recurring engineering
LNA	- low-noise amplifier
LUT	- look-up table
OPB	- on-chip peripheral bus
PLB	- processor local bus
PLD	- programmable logic device
PR	- partially reconfigurable
RAM	- random-access memory
RF	- radio frequency
RNG	- random number generator
ROM	- read-only memory
RTL	- register transfer level
SRAM	- static RAM
VHDL	- very-high-speed integrated circuit hardware description language
VCO	- voltage-controlled oscillator

List of symbols used in this thesis.

A - area

B - buffer size

C - capacitance

D - configuration data

d - functional density

E - energy

F - fragmentation

f - frequency

h - header size

I - electrical current

k - number of regions in a reconfigurable system

n - number of data items that need to be processed between reconfigurations

M - configuration memory page

m - number of reconfigurable modules in a reconfigurable system

P - power

p - degree of parallelism

R - electrical resistance

$\mathcal{R}$ - reconfigurable region

r - configuration ratio

S - speed-up

$\mathcal{S}$ - static region

s - number of processing steps per data item in a algorithm

T - temperature

$\mathcal{T}$ - task

t - time

u - resource tuple

V - voltage

Φ - throughput

Θ - configuration size per unit of area

ϑ - thermal resistance

Ψ - configuration storage size

Bibliography

[1] Actel Corporation. *Igloo Handbook*, April 2009.

[2] "Advanced Encryption Standard (AES)". Federal Information Processing Standards Publication 197, National Institute of Standards and Technology, November 2001.

[3] E. Ahmed and J. Rose. "The effect of LUT and cluster size on deep-submicron FPGA performance and density". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, March 2004.

[4] Altera Corporation. *Stratix-II Device Data Sheet*, 2004.

[5] Altera Corporation. *40-nm FPGA Power Management and Advantages*, 2008. White Paper.

[6] Altera Corporation. *Stratix-III Device Handbook*, 2010.

[7] Altera Corporation. *Stratix-IV Device Handbook*, 2010.

[8] C. Amicucci, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. "SyCERS: a SystemC Design Exploration Framework for SoC Reconfigurable Architecture". In *International Conference on Engineering of Reconfigurable Systems*, pp. 63–69. CSREA Press, 2006.

[9] R. Andraka. "A survey of CORDIC algorithms for FPGA based computers". In *ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 191–200. ACM, 1998.

[10] S. Banerjee, E. Bozorgzadeh, and N. Dutt. "Integrating Physical Constraints in HW-SW Partitioning for Architectures With Partial Dynamic Reconfiguration". *IEEE Transactions on VLSI Systems*, 14(11):1189–1202, 2006.

[11] K. Bazargan, R. Kastner, and M. Sarrafzadeh. "Fast Template Placement for Reconfigurable Computing Systems". *IEEE Design and Test of Computers*, 17(1):68–83, 2000.

[12] J. Becker, M. Hübner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. "Dynamic and Partial FPGA Exploitation". In *Proceedings of the IEEE*, volume 95, pp. 438–452. IEEE, 2007.

[13] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan. “A Self-Reconfiguring Platform”. In *Field-Programmable Logic and Applications*, LNCS 2778, pp. 565–574. Springer, 2003.

[14] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. P. Fekete, and J. van der Veen. “DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices”. In *Field Programmable Logic and Applications*, pp. 153–158. IEEE, 2005.

[15] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. “Parameter variations and impact on circuits and microarchitecture”. In *Design Automation Conference*, pp. 338 – 342. ACM, 2003.

[16] G. Brebner. “A Virtual Hardware Operating System for the Xilinx XC6200”. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL’96)*, pp. 327–336. Springer, 1996.

[17] G. Brebner and O. Diessel. “Chip-Based Reconfigurable Task Management”. In *FPL ’01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pp. 182–191. Springer, 2001.

[18] K. Bruneel, F. Abouelella, and D. Stroobandt. “Automatically mapping applications to a self-reconfiguring platform”. In *Design, Automation and Test in Europe*, pp. 964–969. IEEE, 2009.

[19] K. Bruneel, P. Bertels, and D. Stroobandt. “A method for fast hardware specialization at run-time”. In *Field Programmable Logic and Applications*, pp. 35–40. IEEE, 2007.

[20] Celoxica Limited. *Handel-C Language Reference Manual*, 2003.

[21] C. Claus, F. Müller, J. Zeppenfeld, and W. Stechele. “A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration”. In *IPDPS*, pp. 1–7. IEEE, 2007.

[22] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hübner, and J. Becker. “A Multi-Platform Controller Allowing for Maximum Dynamic Partial Reconfiguration Throughput”. In *Field Programmable Logic and Applications*, pp. 535–538. IEEE, 2008.

[23] J. W. Cooley and J. W. Tukey. “An Algorithm for the Machine Calculation of Complex Fourier Series”. *Mathematics of Computation*, 19(90):297–301, 1965.

[24] R. Cordone, F. Redaelli, M. Redaelli, M. D. Santambrogio, and D. Sciuto. “Partitioning and Scheduling of Task Graphs on Partially Dynamically Reconfigurable FPGAs”. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(5):662–675, 2009.

[25] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing.* Prentice Hall, 1983.

[26] A. DeHon and J. Wawrzynek. "Reconfigurable computing: what, why, and implications for design automation". In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pp. 610–615. ACM Press, 1999.

[27] A. Derbyshire, T. Becker, and W. Luk. "Incremental elaboration for run-time reconfigurable hardware designs". In *International conference on compilers, architecture and synthesis for embedded systems*, pp. 93–102. ACM Press, 2006.

[28] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. "Dynamic scheduling of tasks on partially reconfigurable FPGAs". In *IEE Proceedings: Computers and Digital Techniques*, pp. 181–188, 2000.

[29] R. Dimond, O. Mencer, and W. Luk. "CUSTARD - A Customisable Threaded FPGA Soft Processor and Tools". In *Field Programmable Logic and Applications*, pp. 1–6. IEEE, 2005.

[30] A. Donlin. "Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry". In *Field-Programmable Logic and Applications*, LNCS 1482, pp. 199–208. Springer, 1998.

[31] A. Donlin, P. Lysaght, B. Blodget, and G. Troeger. "A Virtual File System for Dynamically Reconfigurable FPGAs". In *Field-Programmable Logic and Applications*, pp. 1127–1129. Springer, 2004.

[32] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. "Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing". *ACM Transactions Reconfigurable Technology and Systems*, 1(4):1–23, 2009.

[33] F. Ferrandi, M. D. Santambrogio, and DSciuto. "A Design Methodology for Dynamic Reconfiguration: The Caronte Architecture". In *International Parallel and Distributed Processing Symposium*, page 163.2. IEEE Computer Society, 2005.

[34] A. Gayasen, K. Lee, V. Narayanan, M. Kandemir, M. J. Irwin, and T. Tuan. "A Dual-Vdd Low Power FPGA Architecture". In *Field-Programmable Logic and its applications*, pp. 145–157. Springer, 2004.

[35] V. George, H. Zhang, and J. Rabaey. "The design of a low energy FPGA". In *Intl. symposium on Low power electronics and design*, pp. 188–193, 1999.

[36] A. H. Gholamipour, H. Eslami, A. Eltawil, and F. Kurdahi. "Size-Reconfiguration Delay Tradeoffs for a Class of DSP Blocks in Multi-mode Communication Systems". In *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, pp. 71–78. IEEE Computer Society, 2009.

[37] "GNU Radio". `http://gnuradio.org/`.

[38] "GroundHog Benchmark Suite". Technical report, Imperial College London, 2009. `http://cc.doc.ic.ac.uk/projects/GROUNDHOG/`.

[39] S. Guccione and D. Levi. "Run-Time Parameterizable Cores". In *Field-Programmable Logic and Applications*, LNCS 1673, pp. 215–222. Springer, 1999.

[40] S. Guccione, D. Levi, and P. Sundararajan. "JBits: Java Based Interface for Reconfigurable Computing". In *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*. The John Hopkins University, 1999.

[41] J. Hagemeyer, B. Keltelhoit, M. Koester, and M. Porrmann. "A Design Methodology for Communication Infrastructures on Partially Reconfigurable FPGAs". In *Field Programmable Logic and Applications*, pp. 331–338. IEEE, 2007.

[42] W. Han, Y. Yi, M. Muir, I. Nousias, T. Arslan, and A. T. Erdogan. "Multicore architectures with dynamically reconfigurable array processors for wireless broadband technologies". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1830–1843, December 2009.

[43] S. Hauck. "Configuration prefetch for single context reconfigurable coprocessors". In *ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 65–74. ACM Press, 1998.

[44] E. Horta and J. W. Lockwood. "PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)". Technical Report WUCS-01-13, Washington University, Department of Computer Science, 2001.

[45] Y. H. Hu. "CORDIC-based VLSI architectures for digital signal processing". *Signal Processing Magazine*, 9(3):16 –35, 1992.

[46] M. Huebner, T. Becker, and J. Becker. "Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration". In *Proceedings of the 17th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pp. 28–32. ACM Press, 2004.

[47] M. Huebner, M. Ullmann, F. Weissel, and J. Becker. "Real-Time Configuration Code Decompression for Dynamic FPGA Self-Reconfiguration". In *International Parallel and Distributed Processing Symposium*, page 138b. IEEE Computer Society, 2004.

[48] P. Jamieson, T. Becker, P. Y. K. Cheung, W. Luk, T. Rissa, and T. Pitkänen. "Benchmarking and evaluating reconfigurable architectures targeting the mobile do-

main". *ACM Transactions on Design Automation of Electronic Systems*, 15(2):1–24, 2010.

[49] H. Kalte, G. Lee, M. Porrmann, and U. Rückert. "REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems". In *19th International Parallel and Distributed Processing Symposium*, page 151b. IEEE Computer Society, 2005.

[50] E. Keller. "Dynamic Circuit Specialization of a CORDIC Processor". In *Reconfigurable Technology: FPGAs for Computing and Applications II*, SPIE Vol. 4212, pp. 134–141. The International Society for Optical Engineering, 2000.

[51] E. Keller. "JRoute: A Run-Time Routing API for FPGA Hardware". In *International Parallel and Distributed Processing Symposium*, LNCS 1800, pp. 874–881. Springer, 2000. Reconfigurable Architectures Workshop.

[52] I. Kennedy. "A Dynamically Reconfigured UMTS Multi-Channel Complex Code Matched Filter". In *Field-Programmable Technology*, pp. 199–206. IEEE, 2005.

[53] D. Koch, C. Beckhoff, and J. Teich. "A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs". In *Field Programmable Logic and Applications*, pp. 119 – 124. IEEE, 2008.

[54] M. Koester, W. Luk, J. Hagemeyer, and M. Porrmann. "Design Optimizations to Improve Placeability of Partial Reconfiguration Modules". In *Design, Automation and Test in Europe*, pp. 976–981. IEEE, 2009.

[55] M. Koester, M. Porrmann, and H. Kalte. "Task placement for heterogeneous reconfigurable architectures". In *Field-Programmable Technology*, pp. 43–50. IEEE, 2005.

[56] S. Koh and O. Diessel. "Configuration Merging in Point-to-Point Networks for Module-Based FPGA Reconfiguration". *ACM Transactions Reconfigurable Technology and Systems*, 3(1):1–36, 2010.

[57] I. Kuon and J. Rose. "Measuring the Gap Between FPGAs and ASICs". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb. 2007.

[58] I. Kuon, R. Tessier, and J. Rose. "FPGA Architecture: Survey and Challenges". *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[59] T. T.-O. Kwok and Y.-K. Kwok. "On the Design of a Self-Reconfigurable SoPC Based Cryptographic Engine". In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pp. 876–881. IEEE Computer Society, 2004.

[60] J. Lamoureux and S. J. E. Wilton. "On the trade-off between power and flexibility of FPGA clock networks". *ACM Transactions Reconfigurable Technology and Systems*, 1(3):1–33, 2008.

[61] Lattice Semiconductor Corporation. *MachXO Family Data Sheet*, June 2009.

[62] T. K. Lee, A. Derbyshire, W. Luk, and P. Y. K. Cheung. "High-level Language Extensions for Run-time Reconfigurable Systems". In *Proc. IEEE International Conference on Field-Programmable Technology*, pp. 144–151. IEEE Computer Society Press, 2003.

[63] Z. Li, K. Compton, and S. Hauck. "Configuration Caching Management Techniques for Reconfigurable Computing". In *Field-Programmable Custom Computing Machines*, pp. 22–38. IEEE Computer Society, 2000.

[64] Y. Lu, T. Marconi, G. Gaydadjiev, K. Bertels, and R. Meeuws. "A self-adaptive online task placement algorithm for partially reconfigurable systems". In *International Symposium on Parallel and Distributed Processing*, pp. 1–8. IEEE, 2008.

[65] W. Luk, N. Shirazi, and P. Y. K. Cheung. "Modelling and Optimising Run-Time Reconfigurable Systems". In *Field-Programmable Custom Computing Machines*, pp. 167–176. IEEE Computer Society Press, 1996.

[66] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. "Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration on Xilinx FPGAs". In *Field Programmable Logic and Applications*, pp. 1–6. IEEE, 2006.

[67] P. Lysaght and D. Levi. "Of Gates and Wires". In *18th International Parallel and Distributed Processing Symposium*, pp. 132–137. IEEE Computer Society Press, 2004.

[68] P. Lysaght, G. McGregor, and J. Stockwood. "Configuration Controller Synthesis for Dynamically Reconfigurable Systems". In *IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*, pp. 9/1–9/9. IEEE Computer Society Press, 1996.

[69] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. "The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer". *Journal of VLSI Signal Processing*, 47(1):15–31, 2007.

[70] T. S. T. Mak, P. Sedcole, P. Y. K. Cheung, and W. Luk. "On-FPGA Communication Architectures and Design Factors". In *Field Programmable Logic and Applications*, pp. 1–8. IEEE, 2006.

[71] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. "Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs". In *Field-Programmable Logic and Applications*, pp. 795–805. Springer, 2002.

[72] G. McGregor and P. Lysaght. "Self Controlling Dynamic Reconfiguration: A Case Study". In *Field-Programmable Logic and Applications*, pp. 144–154. Springer, 1999.

[73] S. McMillan and C. Patterson. "JBits Implementations of the Advanced Encrpytion Standard (Rijndael)". In *Field-Programmable Logic and Applications*, LNCS 2147, pp. 162–171. Springer, 2001.

[74] Mentor Graphics, Inc. *Catapult C Synthesis Datasheet*, 2010.

[75] J. Mitola. "The Software Radio Architecture". *Communications Magazine*, 33(5):26–38, 1995.

[76] J. Noguera and I. Kennedy. "Power Reduction in Network Equipment Through Adaptive Partial Reconfiguration". In *Field Programmable Logic and Application*, pp. 240–245. IEEE, 2007.

[77] PACT XPP Technologies AG. *XPP-III Processor Overview*, 2006. White Paper.

[78] C. Patterson. "High Performance DES Encryption in Virtex FPGAs using JBits". In *Field-Programmable Custom Computing Machines*, pp. 113–121. IEEE Computer Society Press, 2000.

[79] K. Paulsson, M. Hübner, G. Auer, M. Dreschmann, L. Chen, and J. Becker. "Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self-Reconfiguration on Xilinx Spartan III FPGAs". In *Field Programmable Logic and Applications*, pp. 351–356. IEEE, 2007.

[80] C. Piguet. *Low-power CMOS circuits.* CRC, 2005.

[81] T. Pionteck, R. Koch, and C. Albrecht. "Applying Partial Reconfiguration to Networks-On-Chips". In *Field Programmable Logic and Applications*, pp. 1–6. IEEE, 2006.

[82] P. G. Potter, W. Luk, and P. Y. K. Cheung. "Partition-based exploration for reconfigurable JPEG designs". In *Design, Automation and Test in Europe*, pp. 886–889. IEEE, 2009.

[83] J. G. Proakis and D. K. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications.* Prentice Hall, 1995.

[84] T. Rissa, R. Uusikartano, and J. Niittylahti. "Adaptive FIR filter architectures for run-time reconfigurable FPGAs". In *International Conference on Field-Programmable Technology*, pp. 52–59. IEEE, 2002.

[85] D. Robinson and P. Lysaght. "Modelling and Synthesis of Configuration Controllers for Dynamically Reconfigurable Logic Systems using the DCS CAD Framework". In *Field-Programmable Logic and Applications*, pp. 41–50. Springer, 1999.

[86] D. Robinson, G. McGregor, and P. Lysaght. "New CAD Framework Extends Simulation of Dynamically Reconfigurable Logic". In *Field-Programmable Logic and Applications*, pp. 1–8. Springer, 1998.

[87] A. Schallenberg, W. Nebel, A. Herrholz, P. A. Hartmann, and F. Oppenheimer. "OSSS+R: A framework for application level modelling and synthesis of reconfigurable systems". In *Design, Automation and Test in Europe*, pp. 970–975. IEEE, 2009.

[88] P. Schumacher and P. Jha. "Fast and accurate resource estimation of RTL-based designs targeting FPGAs". In *Field Programmable Logic and Applications*, pp. 59–64. IEEE, 2008.

[89] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. "Modular dynamic reconfiguration in Virtex FPGAs". *IEE Proceedings Computers and Digital Techniques*, 153(3):157–164, 2006.

[90] P. Sedcole and P. Y. K. Cheung. "Parametric yield in FPGAs due to within-die delay variations: a quantitative analysis". In *International Symposium on Field Programmable Gate Arrays*, pp. 178–187. ACM, 2007.

[91] P. Sedcole, P. Y. K. Cheung, G. A. Constantinides, and W. Luk. "A Structured Methodology for System-on-an-FPGA Design". In *Field Programmable Logic and Application*, pp. 1047–1051. Springer, 2004.

[92] S. Seng, W. Luk, and P. Y. K. Cheung. "Run-Time Adaptive Flexible Instruction Processors". In *Field-Programmable Logic and Applications*, pp. 545–555. Springer, 2002.

[93] L. Shang, A. S. Kaviani, and K. Bathala. "Dynamic power consumption in Virtex-II FPGA family". In *ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pp. 157–164. ACM, 2002.

[94] N. Shirazi, W. Luk, and P. Y. K. Cheung. "Run-Time Management of Dynamically Reconfigurable Designs". In *Field-Programmable Logic and Applications*, LNCS 1482, pp. 59–68. Springer, 1998.

[95] Silicon Hive. *HiveFlex CSP2500 Series Digital RF Processor*, 2009.

[96] SiliconBlue Technologies Corporation. *iCE65 Ultra Low-Power Programmable Logic Family Data Sheet*, June 2009.

[97] S. Singh, J. Hogg, and D. McAuley. “Expressing Dynamic Reconfiguration By Partial Evaluation”. In *Field-Programmable Custom Computing Machines*, pp. 188–194. IEEE Computer Society Press, 1996.

[98] L. Singhal and E. Bozorgzadeh. “Multi-layer Floorplanning on a Sequence of Reconfigurable Designs”. In *Field Programmable Logic and Applications*, pp. 1–8. IEEE, 2006.

[99] C. Steiger, H. Walder, and M. Platzner. “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks”. *IEEE Transactions on Computers*, 53(11):1393–1407, 2004.

[100] R. Tessier, S. Swaminathan, R. Ramaswamy, D. Goeckel, and W. Burleson. “A reconfigurable, power-efficient adaptive Viterbi decoder”. *IEEE Transactions on VLSI Systems*, 13(4):484–488, 2005.

[101] D. B. Thomas and W. Luk. “A Domain Specific Language for Reconfigurable Path-based Monte Carlo Simulations”. In *Field-Programmable Technology*, pp. 97–104. IEEE, 2007.

[102] D. B. Thomas and W. Luk. “High Quality Uniform Random Number Generation Using LUT Optimised State-transition Matrices”. *VLSI Signal Processing*, 47(1):77–92, 2007.

[103] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. “A Time-Multiplexed FPGA”. In *Field-Programmable Custom Computing Machines*, pp. 22–28. IEEE Computer Society Press, 1997.

[104] A. H. T. Tse, D. B. Thomas, and W. Luk. “Accelerating Quadrature Methods for Option Valuation”. In *Field Programmable Custom Computing Machines*, pp. 29–36. IEEE Computer Society, 2009.

[105] T. Tuan, A. Rahman, S. Das, S. Trimberger, and S. Kao. “A 90-nm Low-Power FPGA for Battery-Powered Applications”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):296–300, Feb. 2007.

[106] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. “A Self-Reconfigurable Implementation of the JPEG Encoder”. In *ASAP’07 - 18th International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2007.

[107] W. Tuttlebee. *Software Defined Radio: Enabling Technologies*. Wiley, 2002.

[108] M. Uhm. “Software-Defined Radio: The New Architectural Paradigm”. *DSP Magazin*, pp. 40–42, October 2005.

[109] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. K. Kuzmanov, and E. M. Panainte. “The Molen Polymorphic Processor”. *IEEE Transactions on Computers*, 53(11):1363–1375, November 2004.

[110] J. Volder. “The CORDIC Trigonometric Computing Technique”. *IRE Transactions on Electronic Computing*, EC-8(3):330–334, 1959.

[111] H. Walder and M. Platzner. “Non-preemptive multitasking on FPGA: Task placement and footprint transform”. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 24–30. CSREA Press, 2002.

[112] J. S. Walther. “A Unified Algorithm for Elementary Functions”. In *Spring Joint Computer Conference*, pp. 379–385, 1971.

[113] A. Wiesler and F. K. Jondral. “A software radio for second- and third-generation mobile systems”. *Vehicular Technology, IEEE Transactions on*, 51(4):738–748, July 2002.

[114] J. Williams and N. Bergmann. “Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip”. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 163–169. CSREA Press, 2004.

[115] M. Wirthlin and B. Hutchings. “Improving Functional Density Using Run-Time Circuit Reconfiguration”. *IEEE Transactions on VLSI Systems*, 6(2):247–256, 1998.

[116] Xilinx, Inc. *Architecting Systems for Upgradability with IRL (Internet Reconfigurable Logic)*, 2001. Application Note.

[117] Xilinx, Inc. *Power vs. Performance: The 90 nm Inflection Point*, 2006. White Paper.

[118] Xilinx, Inc. *Virtex-5 Configuration Guide v2.1*, October 2006.

[119] Xilinx, Inc. *Xilinx Logic Core: OPB HWICAP*, July 2006.

[120] Xilinx, Inc. *Correcting Single-Event Upsets in Virtex-II Platform FPGA Configuration Memory*, 2007. Application Note.

[121] Xilinx, Inc. *Embedded Systems Tools Reference Manual, EDK 9.2i*, September 2007.

[122] Xilinx, Inc. *ISE 9.2i Software Manuals and Help*, October 2007.

[123] Xilinx, Inc. *MicroBlaze Processor Reference Guide v8.0*, June 2007.

[124] Xilinx, Inc. *PlanAhead User Guide v9.2*, July 2007.

[125] Xilinx, Inc. *Spartan-3 Generation FPGA User Guide v.1.2*, April 2007.

[126] Xilinx, Inc. *Using Suspend Mode in Spartan-3 Generation FPGA*, May 2007.

[127] Xilinx, Inc. *Virtex-4 Configuration Guide v1.5*, January 2007.

[128] Xilinx, Inc. *Virtex-5 Family Platfrom Overview LX, LXT, and SXT Platforms*, September 2007.

[129] Xilinx, Inc. *Virtex-5 User Guide*, September 2007.

[130] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, May 2007.

[131] Xilinx, Inc. *Early Access Partial Reconfiguration User Guide*, 2008. UG208 v1.2.

[132] Xilinx, Inc. *Spartan-3E FPGA Family: Complete Data Sheet*, April 2008.

[133] Xilinx, Inc. *Virtex-4 FPGA Data Sheet: DC and Switching Characteristics*, September 2009.

[134] Xilinx, Inc. *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics*, June 2009.

[135] Xilinx, Inc. *PowerPC 405 Processor Block Reference Guide v2.4*, January 2010.

[136] Xilinx, Inc. *Spartan-6 Family Overview v.1.4*, March 2010.

[137] Xilinx, Inc. *Virtex-6 Family Overview*, January 2010.

[138] S. Yang. "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0". Technical report, Microelectronics Center of North Carolina, 1991.

[139] S. Young, P. Alfke, C. Fewer, S. McMillan, B. Blodget, and D. Levi. "A High I/O Reconfigurable Crossbar Switch". In *Field-Programmable Custom Computing Machines*, pp. 3–10. IEEE Computer Society Press, 2003.

[140] P. Zipf, M. Glesner, C. Bauer, and H. Wojtkowiak. "Handling FPGA Faults and Configuration Sequencing Using a Hardware Extension". In *Field-Programmable Logic and Applications*, pp. 71–80. Springer, 2002.