Reconfiguration of Field Programmable Logic in Embedded Systems

Irwin O. Kennedy

Doctor of Philosophy

School of Informatics

University of Edinburgh

2005

Abstract

This thesis presents a set of techniques for evaluating and exploiting the programmability of a reconfigurable computing fabric in embedded systems. It concentrates specifically on the Field-Programmable Gate Array.

Reconfigurable logic promises a flexible computing fabric well suited to the low cost, low power, high performance and fast time to market now demanded of today's computing devices. Today, most reconfigurable systems are constructed using ad hoc techniques, making little use of previous designs and intellectual property (IP).

This thesis proposes a systematic approach to reconfigurable real-time system design. It exploits static and medium frequency reconfiguration by exploiting inter-task mutually exclusive resource usage. It exploits high frequency reconfiguration through data-folding specialisation techniques. A procedure for creating parameterised designs that approach minimal coverage of all possible system requirements is described. A runtime framework based upon a regularly occurring system-wide pause of execution is described.

A large case study of the design approach and runtime framework is presented and compared with the static equivalent. The case study system is a commercial Universal Mobile Telecommunications System (UMTS) physical layer processing engine. Equations describing the logic gate and memory requirements of the commercial ASIC design are extracted and used to estimate resource requirements of a low-medium frequency reconfigurable solution. A detailed investigation of very rapid reconfiguration is carried out on a large circuit block. Good logic and memory resource requirement reduction is shown to be possible.

A complementary FPGA reconfiguration architecture is presented. It provides the ability to tradeoff time and space according to the reconfiguration speed requirements of an application domain. A number of configuration compression schemes are investigated. In addition to an excellent compression ratio they are shown to be highly parallelisable and scalable, unlike previous approaches.

Acknowledgements

I would like to thank my supervisor Murray Cole for his support and encouragement. Murray helped turn my work into the thesis presented. I am also very grateful to my previous supervisor Gordon Brebner for his guidance and constructive criticism and to Frank Mullany at Lucent Bell Labs for his excellent technical advice.

Others who were helpful to me include Michael Dales, André DeHon, John Gray, Tom Kean and Steve Trimberger.

This work was supported by an EPSRC CASE award in collaboration with Lucent Bell-Labs.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. Some of the material in this thesis has already been published:

- G. Brebner and I. Kennedy, "Circlets: Circuitry over the Internet", Proceedings of 9th IEEE Symposium on Field-Programmable Custom Computing Machines, April, 2001.

- I. Kennedy and G. Brebner, "Novel Techniques to Enhance the use of Dynamic Reconfiguration in Fine Grain Field Programmable Logic", Proceedings 2nd U.K. ACM SIGDA Workshop on Electronic Design Automation, September, 2002.

- I. Kennedy, "Fast Reconfiguration Through Difference Compression", Proceedings of 11th IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 265-266, April, 2003.

- I. Kennedy, "Exploiting Redundancy to Speedup Reconfiguration of an FPGA", Proceedings of 13th International Conference on Field Programmable Logic and Applications, Springer Lecture Notes in Computer Science 2778, pp. 262-271, September, 2003.

- I. Kennedy and F. Mullany, "Design of a Reconfigurable UMTS Channel Processing Engine", Proceedings IEEE Semiannual Vehicular Technology Conference, May, 2004.

- I. Kennedy, "A dynamically reconfigured UMTS multi-channel complex code matched filter", Proceedings IEEE International Conference on Field-Programmable Technology, December, 2005 (to appear).

(Irwin O. Kennedy)

Contents

List of Figures

List of Tables

Glossary

Chip	A chip is a binary digit at the physical link layer. The redundancy introduced by Forward Error Correction, spreading etc. results in one application layer bit being represented by many chips.
Finger	A RAKE receiver exploits multiple versions of the same transmit stream by descrambling and despreading several delay paths. The different paths are then combined to maximise the received energy. A finger is one correlator unit allocated to a particular delay path which is to be detected and demodulated.
Handover	Handover or Hard Handover refers to the process of switching all UMTS physical links from one NodeB to another NodeB.
Node B	UMTS cellular basestation.
RAKE	RAKE is the name of a receiver architecture for spread spectrum communication systems.
Soft Handover	In Soft Handover, the transfer of radio links is performed gradually ensuring the UE always has at least one active radio link at all times.
Softer Handover	Softer Handover is a special form of soft handover in which radio links are transferred within the same NodeB, for example, between two sectors on the same base station.

Acronyms

3GPP	3rd Generation Partnership Project
3G	3rd Generation Cellular Mobile Telecommunications
ACM	Association for Computing Machinery
ADC	Analogue to Digital Converter
ALAP	As Late As Possible (Scheduling Algorithm)
ALU	Arithmetic Logic Unit
AP	Access Point
API	Application Programming Interface
APS	Application Profile Set
ASAP	As Soon As Possible (Scheduling Algorithm)
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BLER	Block Error Rate
BPSK	Binary Phase Shift Keying
BRAM	Block Random Access Memory
BW	Bandwidth
CAM	Content Addressable Memory
CCMF	Complex Code Matched Filter
CCPCH	Common Control Physical Channel
CDFG	Control Data Flow Graph
CDMA	Code Division Multiple Access
CFG	Control Flow Graph
CF	Code Folded
CFC	Code Folded Control
CLB	Configurable Logic Block
CMF	Code Matched Filter
CPCH	Common Packet Channel
CPICH	Common Pilot Channel
CPU	Central Processing Unit

CRC	Cyclic Redundancy Check
DAC	Digital to Analogue Converter
DC	Direct Current
DCH	Dedicated Channel
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DF	Data Folded
DFC	Data Folded Control
DIMM	Dual In-line Memory Module
DLL	Delay Locked Loop
DPCCH	Dedicated Physical Control Channel
DPCH	Downlink Dedicated Physical Channel
DPDCH	Dedicated Physical Data Channel
DPGA	Dynamically Programmable Gate Array
DPSB	Dot Product Sub Block
DRAM	Dynamic Random Access Memory
DS-CDMA	Direct Sequence Code Division Multiple Access
DSP	Digital Signal Processor
DSSS	Direct Sequence Spread Spectrum
EMI	Electro-Magnetic Interference
ESIP	Extended Soft Information Processor
ETSI	European Telecommunications Standards Institute
FACH	Forward Access Channel
FBI	Feed Back Information
FDD	Frequency Division Duplex
FDMA	Frequency Division Multiple Access
FEC	Forward Error Correction
FF	Flip/Flop
FFT	Fast Fourier Transform
FHT	Fast Hadamard Transform
FIR	Finite Impulse Response

FPFA	Field Programmable Function Array
FPGA	Field Programmable Gate Array
FPL	Field Programmable Logic
FSM	Finite State Machine
GPP	General Purpose Processor
GSM	Global System for Mobile Communications
HC	Hardware Configuration
HCS	Hardware Configuration Set
HDL	Hardware Description Language
HSDPA	High Speed Downlink Packet Access
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response Filter
IOB	Input Output Buffer
IP	Intellectual Property
I/Q	In phase/Quadrature
JEDEC	Joint Electron Device Engineering Council
JIT	Just In Time
LLR	Log Likelihood Ratio
LRU	Least Recently Used
LUT	Look Up Table
LZ	Lempel Ziv
MAC	Medium Access Controller
MAI	Multiple Access Interference
MEMS	Micro Electro-Mechanical Systems
MIMD	Multiple Instruction Multiple Data
MMSE	Minimum Mean Square Error
MPEG	Moving Picture Expert Group
MUX	Multiple xor
NRE	Non Recurring Engineering
OFDM	Orthogonal Frequency Division Modulation

OS	Operating System
OSI	Open Systems Interconnection Reference Model
OVSF	Orthogonal Variable Spreading Factor
PAL	Programmable Array Logic
PCH	Physical Channel
PCPCH	Physical Common Packet Channel
PD	Preamble Detector
PDI	Post Detection Integration
PDSCH	Physical Downlink Shared Channel
PIP	Programmable Interconnect Point
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PRACH	Physical Random Access Channel
QPSK	Quadrature Phase Shift Keying
RACH	Random Access Channel
RTL	Register Transfer Language
SCH	Shared Channel
SDMA	Space Division Multiple Access
SDR	Software Defined Radio
SDRAM	Synchronous Dynamic Random Access Memory
SIA	Semiconductor Industry Association
SIMD	Single Instruction Multiple Data
SLU	Swappable Logic Unit
SNR	Signal to Noise Ratio
SOC	System On Chip
SRB	Sample Rate Buffer
SRL	Shift Register LUT
TDD	Time Division Duplex
TDMA	Time Division Multiple Access
TFCI	Transport Format Combination Indicator
TPC	Transmit Power Control

TTI	Transmission Time Interval
TX	Transmit
UART	Universal Asynchronous Receiver-Transmitter
UE	User Equipment
UL	Uplink
UMTS	Universal Mobile Telecommunications System
UMTSCE	UMTS Channel Element
UMTSSOC	UMTS System On Chip
VHDL	Very High Speed Integrated Circuit HDL
WCDMA	Wideband Code Division Multiple Access

Chapter 1

Introduction

1.1 Introduction

According to Semiconductor Industry Association (SIA) projections[144], the number of transistors per die and the local clock frequencies for high-performance designs will continue to grow exponentially over the next decade. By the year 2014, the SIA predicts that 20 billion transistors will be available per die. As a consequence of such massive transistor integration and the advent of Micro-Electro-Mechanical Systems (MEMS), new application spaces are envisaged as well as the growth of existing ones. For example it is predicted that smart devices woven into our environment will create a world of "ambient intelligence"[13]. An existing application space predicted to benefit from strong growth is wireless communications and the software defined radio (SDR)[140]. The ideal SDR consists of a small analogue stage with all other processing performed in software, although, it may be more appropriate to think of the implementation technology as soft-programmable, rather than a microprocessor. The flexibility enables standards changes to be implemented via a download over the air interface and the support of multiple standards by a single device.

Reconfigurable computing is a term which has emerged to describe a class of computing fabrics which may be reprogrammed electronically post-manufacture. As will be described in Section 1.3 they offer an alternative computing fabric to the system designer, combining the flexibility of the microprocessor and the spatial circuitry im-

plementation of the ASIC. The design of reconfigurable computing systems is the focus of this thesis, with particular attention given to platform Field-Programmable Gate Arrays (FPGAs). A large case study from the domain of wireless communications is used to demonstrate the design techniques proposed.

The introduction and background on reconfigurable computing is presented in Chapter 2. Here we introduce related areas necessary for the understanding of the thesis. The specific area of reconfigurable computing of interest is called Just-In-Time computing (JIT) and is introduced in Section 1.2. Section 1.3 describes the difference between spatial and temporal computing and introduces the main computational fabrics at a system architect's disposal. Section 1.4 provides a short introductory motivation for the choice of wireless communications as the major case study in the thesis. Finally, an overview of each chapter in the thesis is presented in Section 1.5.

1.2 Just In Time Computing

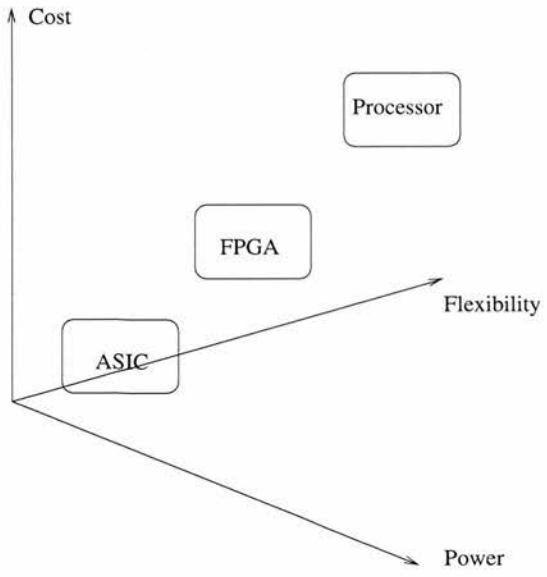

Figure 1.1: Just In Time (JIT) Tradeoff Space

Just In Time Computing refers to the design tradeoff space of embedded systems as described by Rabaey in [131]. As illustrated in Figure 1.1, cost, power and flexibility

are the three axes in the tradeoff space. Notably, performance is missing from the design space. This is a key difference between Just-In-Time computing and traditional, transformational general purpose computing. Rather than attempting to maximise performance, the functionality is known and simply becomes a design constraint. Embedded systems enjoy a well specified set of tasks at design time. This contrasts sharply with general purpose computing where the set of tasks to be performed is not known at design time. The term "Just-In-Time" captures an embedded system's requirement to simply match the system's performance criteria, rather than attempt to perform it as fast as possible.

The tradeoff between the cost, power and flexibility is obviously dependent on the system being designed. For example, power consumption is critical for battery powered devices, but is often only a concern from a heat dissipation perspective in mains powered devices such as television set-top boxes.

The cost tradeoff axis is the complete cost of the solution. For example, an ASIC's cost must include all Non-Recurring Engineering (NRE) charges, such as the mask set used in fabrication, functional verification and physical verification. The microprocessor and FPGA have no NRE costs when bought as a complete, packaged device, however they incur some NRE costs if they are incorporated into an ASIC System On Chip. For example, masks sets must be manufactured to describe the ASIC logic and the intellectual property of the FPGA fabric or processor on the same chip. Some of the verification cost may be reduced, since the intellectual property will be used across many designs, and hence may be partially amortised.

Flexibility is the third tradeoff axis in Figure 1.1. It refers to the level of programmability of the system post-manufacture. With time to market critical for many systems, the ability to apply bug fixes and incorporate changes to evolving standards late in the design process is desirable. The FPGA and the microprocessor are well suited to such requirements. An ASIC designed and optimised to a specific task has very poor flexibility since its functionality is fixed at manufacture. A two stage implementation strategy is adopted for some standards driven products in which an initial solution is rapidly produced using flexible fabrics to establish a market position, and then once the standard has stabilised, cost is reduced by creating an ASIC. Another ex-

ample is a system built for a family of products which may require a level of flexibility for integration purposes. Flexibility is discussed further in Section 2.4.

1.3 Computing Fabrics

1.3.1 Introduction

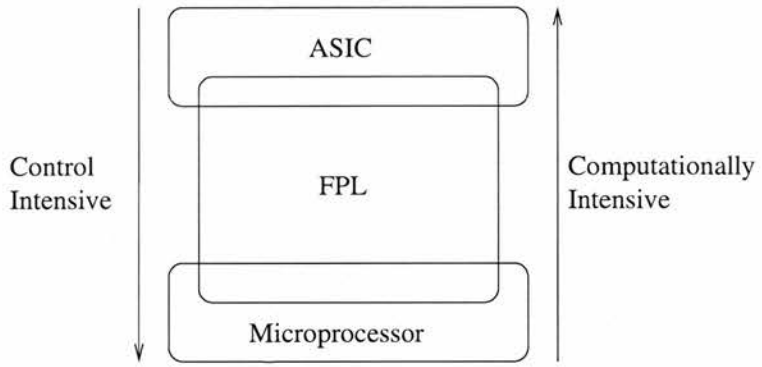

Figure 1.2: Computational Space Occupied by Reconfigurable Logic

There are three broad types of computing fabric - the Application Specific Integrated Circuit (ASIC), Field-Programmable Logic (FPL) and Microprocessor. Note that enhanced forms and hybrids of these basic fabrics also exist, like the digital signal processor (DSP). These are discussed in later sections. Figure 1.2 illustrates the relative suitability of the basic fabrics to control and compute intensive algorithms (arrows indicate increasing suitability). The ASIC is built using libraries of logic cells which define primitives such as logic gates, memories and adders. By connecting the cells circuitry can be described. When an ASIC design is complete, it is fabricated to perform the task exactly as explicitly defined by the cells and wires interconnecting them. FPL, for example the FPGA, is programmed electronically post manufacture by configuring specially designed logic blocks and routing resources. It is described in more detail in Section 2.2.

The Microprocessor and FPGA may be bought as standard devices (pre-manufactured) and hence are programmed to perform their task post-manufacture. This means they

may be used across many different systems, amortising much of the cost and risk associated with an ASIC. The level of abstraction for the processor and the FPGA is higher than that of the ASIC. Neither the FPGA nor ASIC fabric offers the same degree of control as the microprocessor, although the FPGA offers substantially more than the ASIC. As illustrated in Figure 1.2, to be better at control intensive tasks, a fabric trades off performance. This is due to the penalty paid for offering a high level of post-manufacture programmability.

In this section we will define spatial and temporal computation and then base a discussion of the three fabrics around Figure 1.2.

1.3.2 Spatial versus Temporal

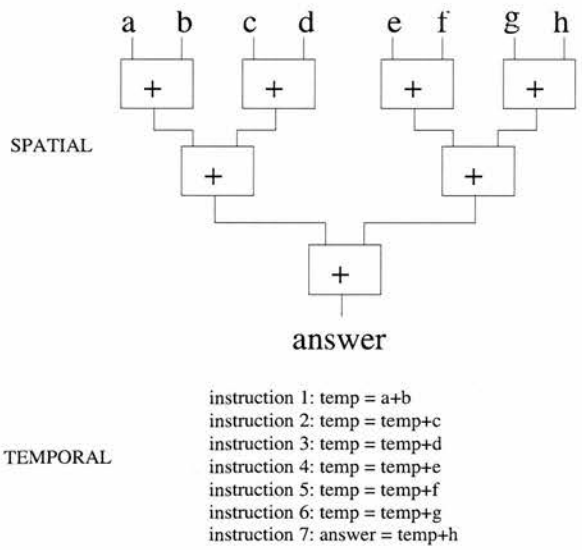

Figure 1.3: Temporal versus Spatial Programming Space

The von Neumann computing paradigm led to the microprocessor architecture. It consists of a fixed datapath and a dynamic set of instructions (program) which configure the datapath to perform a task by way of a number of steps. It provided a common architecture that could be used to perform many different tasks.

Rather than time-multiplexing a fixed data-path, it is possible to implement algorithms in space by constructing circuitry to perform them directly. This may be viewed

as the more natural candidate for implementation, since a microprocessor is effectively emulating hardware. Figure 1.3 illustrates the difference between temporal programming, as is done with a microprocessor and spatial programming, as is done with an ASIC or an FPGA. The temporal example performs the computation by time multiplexing a single adder. The fully spatial implementation uses 7 adders to perform the same computation.

It is possible to use a combination of spatial and temporal styles to tradeoff cost against time. For the sum example in Figure 1.3, the number of adders used is the cost and the time is the number of clock cycles required to perform the computation. An example combination of spatial and temporal techniques would be to use only 3 adders with some registers and control logic to perform the addition in a number of cycles. It should be noted that the brief overview of temporal versus spatial design given here is highly simplified. The ability of the FPGA or an ASIC to use a combination of both design styles means they can implement a microprocessor. However, neither would achieve as good a point in the JIT tradeoff space as a full custom implementation of the microprocessor. In this work we use the term ASIC and FPGA to refer to their classical use as fabrics for the construction of bespoke circuitry.

1.3.3 Computing Fabric Discussion

The ASIC, FPL and microprocessor bring different qualities to the JIT computing tradeoff space. The ASIC may be designed to perform a specific task or set of tasks, hence its silicon area efficiency, speed and power consumption are excellent. It exhibits very little flexibility post-manufacture so if a chip's application space is not well defined at design time, a complete ASIC solution is not wise. FPL is programmed post-manufacture and hence offers flexibility, but this is paid for by being slower and less energy efficient than the ASIC. The FPL fabric design space is varied and the exact tradeoff is dependent on the match between the fabric and the application. Typically, the granularity of the computing elements making up an FPL fabric determines its speed/flexibility/energy tradeoff. This is discussed in Section 2.2.2. The microprocessor offers greatest flexibility, which has in part led to its success. In embedded systems, the high flexibility of the microprocessor is well suited to control flow tasks

such as a user interface.

Flexibility is also leading to the microprocessor's downfall as *the* computational fabric in the new era of computing dominated by just-in-time systems. Increasing clock rates and the exploitation of instruction level parallelism have provided diminishing returns on processor performance[90]. The once impressive scaling of the microprocessor's performance with Moore's law is no longer the case. Today, less than 10% of a modern microprocessor's die actually implements the datapath, with the majority of area taken by the cache to overcome the *memory wall*[174], and circuitry to inject instruction level parallelism. The modern high performance microprocessor's efficiency as measured by the percentage of transistors actively involved with performing computation is extremely low. It is well recognised in the literature that FPL provides a more energy and silicon efficient solution over the microprocessor for computationally intensive tasks such as DSP. For example, in [153] the authors observe energy savings in commercial devices of 89% when using FPL instead of an embedded microprocessor.

Changes have been made to the microprocessor's architecture to make it more suitable to data intensive processing, producing the digital signal processor. These are best characterised by the Harvard architecture, which effectively separates program bus and data bus. Parallelism is enhanced by overlapping data fetch, data operations and address calculations. Instruction set enhancements help minimise loop overheads and simplify the implementation of filters. A thorough summary of DSP architectures can be found in [84]. Despite the introduction of VLIW and multiple function units, the DSP is still fundamentally sequential, being based upon a serial instruction stream. In a move away from the serial instruction stream, the reconfigurable DSP has emerged, which we describe in background section 2.3.3.

1.4 Wireless Communications

The rise of untethered computing devices is a major contributor to massive growth in wireless communications. It has been observed in [131] that the number of worldwide wireless subscribers each year grows as a Fibonacci series, and looks set to continue doing so in the near term. In [20] the authors predict that as the number of wireless

devices operated directly by humans saturates (e.g. cellphones), wirelessly connected machines will continue the growth trend.

Partly as a consequence of this growth in wireless communications, the efficient use of the electromagnetic spectrum enjoys very active research. The sophistication of the algorithms being developed to harness efficiently a given bandwidth is increasing rapidly. In [131] it is pointed out that the computational demands of these algorithms are growing much more quickly than the processor performance as predicted by Moore's law. Therefore, the software defined radio is, at least in the short term, likely to be a reality only in terms of the flexibility to use multiple air-interfaces. High layers in the communication protocol stack may be performed by software on a microprocessor, however lower level layers such as the MAC and Physical layer are too computationally intensive to be satisfied by a microprocessor. Instead, more specialised computational fabrics are required to meet the JIT requirements. These reasons, combined with the evolving nature of new communication standards, suggest that wireless communications is a good application area for reconfigurable computing. Therefore, the 3rd generation cellular standard called the Universal Telecommunications System (UMTS) is chosen for the case study in the thesis. Specifically, the physical layer processing engine of the base station is studied. Appendix A provides some fundamental wireless communications background and references for the interested reader.

1.5 Thesis Overview

This thesis proposes an approach to the design of complex real-time embedded systems on programmable platforms. It offers a realistic approach to harnessing reconfiguration by acknowledging the increased design complexity of runtime reconfiguration. The word "realistic" is used here to differentiate the work from previous work on harnessing reconfiguration. The increased silicon area due to the configuration architecture, the disruption to the traditional design flow and the lack of tool support are considered in addition to the energy and resource savings achievable. It proposes a top-down approach to design, enabling a global optimisation effort rather than the local optimisation effort of the many previously proposed bottom-up approaches.

We use the physical layer processing engine of the Universal Mobile Telecommunication System (UMTS) as a case study throughout the thesis. It is a fully specified design for cellular basestations supporting up to 64 users.

In Chapter 2 we provide the reader with the required background for the understanding of the thesis including a detailed description of the Field Programmable Gate Array, fundamentals of hardware/software co-design, runtime reconfiguration and platform based design.

Chapter 3 explores the configuration architecture of an FPGA and proposes a design tradeoff space. Several techniques are investigated for speeding up reconfiguration and the corresponding required changes to the Xilinx Virtex configuration architecture are investigated.

Chapter 4 describes the proposed design methodology for targeting a reconfigurable fabric. The chapter begins by discussing the difference between transformational and reactive computing systems, and develops this difference to justify the proposed design framework for reconfigurable reactive systems. We then discuss the challenges of runtime reconfiguration and the opportunities of heterogeneous platform based design. The checkpoint based design framework is then proposed, together with its targeting approach.

In Chapter 5, the design of a commercial UMTS channel processing engine is used to apply the methodology in Chapter 4. This is done for all subsystems at a low-medium frequency of reconfiguration and for a single block of circuitry for high frequency reconfiguration.

Chapter 2

Background

2.1 Introduction

This chapter presents essential background material for the understanding of the thesis. Field-Programmable Logic (FPL) is described in Section 2.2 with reconfigurable computing introduced in Section 2.3. Section 2.4 provides an overview of the emerging paradigm of platform based design. Previous approaches to harnessing a runtime reconfigurable Field Programmable Gate Array (FPGA) are outlined in Section 2.5. In Section 2.6 we summarise work in the literature on harnessing run-time reconfiguration. Section 2.7 summarises previous techniques developed for speeding up reconfiguration. Finally, Section 2.8 provides a summary of the chapter. Some fundamental principles of cellular wireless communications are provided in Appendix A.

2.2 FPL

2.2.1 Overview

Field-Programmable Logic, also known as reconfigurable logic, refers to a class of architectures in which the configuration bits are directly involved with controlling hardware and links; otherwise, the architecture is referred to as programmable. Most FPL architectures are a mixture of programmable and reconfigurable.

An FPL fabric performs computation spatially. Most commonly, it is composed

of a two-dimensional grid of computational elements which are connected together using some form of reconfigurable interconnect mesh. We note that other architectural compositions have been proposed, such as the linear array as in RaPiD[61] and PipeRench[138]. These architectures are aimed at highly regular pipelined computation-intensive tasks, and are therefore limited in application. In this section we will discuss the term granularity, describe in detail one particular type of FPL - the Field Programmable Gate Array (FPGA) and outline some previous work on hardware-software co-design.

2.2.2 Granularity

The granularity of an FPL fabric refers to the width of the path between computation elements and the computation performed by an individual element. In [82] the authors look at mapping algorithms to a coarse grain fabric arranged as a two dimensional grid of N-bit ALUs. Another example is the FPGA such as the Xilinx Virtex [177] which uses lookup tables of 4-bit input, 1-bit output to perform computation at a much finer granularity. In general, coarse grain FPL is better suited to word-oriented computation such as multiplication and fine-grain FPL is better suited to bit oriented computation such as the permutation operations typical of encryption algorithms.

2.2.3 FPGA

The Field Programmable Gate Array (FPGA) is a particular type of FPL architecture dating back to around 1989[71]. It consists of a 2 dimensional grid of look up tables and flip/flops connected by a hierarchical interconnect fabric. Through programming the contents of the LUTs and setting multiplexers within the interconnect to connect the LUTs together as required, circuitry is created. Most common is the 4-input, 1-output LUT. It consists of a 16-bit memory and is therefore capable of implementing any function of 4 inputs. Figure 2.1 is a diagram illustrating an FPGA's architecture. It is programmed (post manufacture) by loading a configuration bitstream into the device's highly distributed configuration RAM.

Figure 2.2 shows an example of a commercial architecture's logic block. It is the

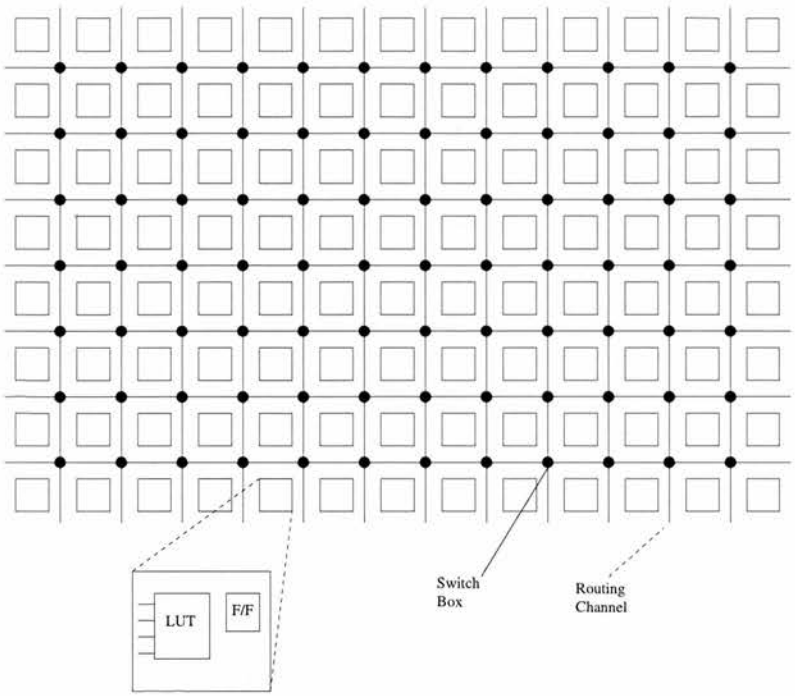

Figure 2.1: Generalised FPGA Architecture

Xilinx Virtex Configurable Logic Block (CLB), containing two identical slices, each slice consisting of two LUT and flip/flop pairs. Rather than the simple single LUT and flip/flop as drawn in the diagram of Figure 2.1, the slices also contain circuitry to help with common arithmetic - for example the provision of a fast carry chain for adders.

2.3 Runtime Reconfigurable FPL

2.3.1 Introduction

Runtime reconfiguration of FPL refers to changing the content of the configuration memory during computation. This enables the algorithm designer to allocate computational resources to match algorithm and data. Such architectures typically couple the FPL with a processor or sequencer to form a hybrid. The sequencer or processor's role is to manage configurations and perform control intensive tasks.

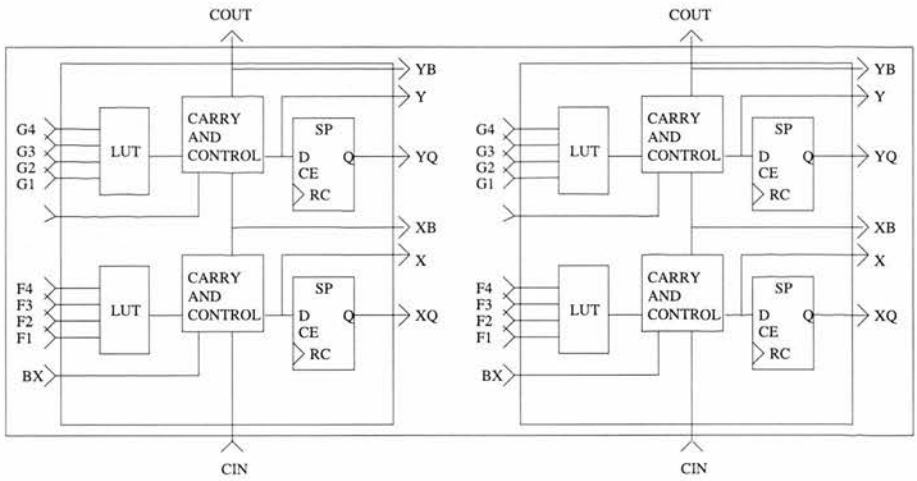

Figure 2.2: Xilinx Virtex CLB

Here we offer a brief overview of the runtime reconfigurable FPGA in section 2.3.2 and the runtime reconfigurable DSP in section 2.3.3. The concept of specialisation is introduced in section 2.3.4 and hardware-software co-design in the context of runtime reconfigurable FPL is described in section 2.3.5.

2.3.2 Runtime Reconfigurable FPGA

Most hybrid processor-FPGA based systems reported in the literature are not runtime reconfigurable, but static. In these systems, the computationally intensive kernels of the application are implemented on the FPGA and the rest on a microprocessor. The FPGA's configuration is downloaded at the start of system execution and does not change during execution. PRISM[11], Splash[10], and the Programmable Active Memory (PAM) work at DEC-Paris labs[17] are frequently referenced examples of this system model.

Examples of runtime reconfigurable FPGA based systems include a reconfigurable compiler and FPGA architecture called WASMII[106], a self reconfiguration processor[64] and work on incremental run-time reconfiguration by Lysaght and Dunlop[109].

2.3.3 DSP and Runtime Reconfigurable DSP

DSP processors are instruction-set programmable architectures specialised to digital signal processing problems[155]. This makes them well suited for repetitive numerical processing tasks such as arithmetic loop kernels, and less suitable for general purpose computing tasks. They benefit from architectural features such as single cycle multiply and accumulate (MAC) units, zero-overhead loop circuitry, address generation units and a throughput oriented memory architecture. An example of a DSP processor is the Texas Instruments TI6416 DSP[158].

The runtime reconfigurable DSP offers an alternative architecture to the traditional DSP. Reconfigurable captures their spatial processing ability and runtime captures its operation. A runtime reconfigurable DSP uses coarse grain processing elements and interconnect path-widths greater than one bit. This is because DSP benefits from modular arithmetic operators and it brings large area savings over the single bit CLB and interconnect offered by the FPGA. However, it should be noted that coarse-grained architectures are poor at bit-level manipulations found, for example, in many communications algorithms.

An example of a runtime reconfigurable DSP is MATRIX[118], based upon a MIMD grid of small 8 bit processors with near-neighbour and length-four connectivity. The REMARC[119] architecture was based upon a grid of 16-bit processors with nearest neighbour and full-length bus based communication, globally controlled with a SIMD-like instruction sequencer.

2.3.4 Specialisation

Specialisation refers to the ability of reconfigurable FPL to be tailored to more closely match its requirements. This is in contrast to an ASIC design which is fixed at manufacture and hence is generalised, satisfying all possible requirements including the "worst-case". Figure 2.3 illustrates this point by showing how a reconfigurable design may exploit better specified design time knowledge such as "initialisation parameters" together with "run-time parameters" to better match the exact requirements. This is often referred to as data-folding, or constant-propagation. The closer an algorithm's

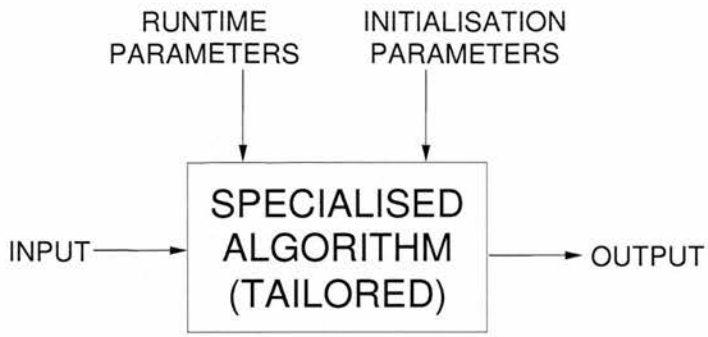

Figure 2.3: Algorithm Implemented on a Reconfigurable FPL Fabric

implementation is to its actual use, the more data is available for specialisation and the less general purpose it becomes. In contrast, the ASIC must implement a "one-size fits all" design. FPL enjoys a continuum of specialisation from static through to very frequent dynamic reconfiguration. The frequency of reconfiguration is a subject of much interest in the literature and is discussed in this thesis. There are numerous examples of specialisation across a wide variety of application domains in the literature, from folding the key into an encryption algorithm[129], to creating a constant coefficient multiplier for a FIR filter[42].

2.3.5 Hardware Software Co-design

2.3.5.1 Overview

A large body of literature exists on the problem of hardware-software co-design. "Hardware" in this work primarily refers to the ASIC and "software" refers to computation performed by the microprocessor. FPL, particularly reconfigurable FPL, blurs the traditional boundary between hardware and software. The most common approach to harnessing FPL in the reconfigurable computing literature has been software centric, i.e. the FPL is treated as a slave to the microprocessor as discussed in Section 2.3.5.2. The other more recent approach is to elevate the FPL's importance from slave to master with logic centric design as discussed in Section 2.3.5.3.

2.3.5.2 Software Centric

There are many projects in the literature which attempt to take a high-level description of a design (e.g. written in C) and automatically map it to a microprocessor and FPL. The architectural solutions differ primarily in the coupling between the microprocessor and the FPL. The coupling varies between including FPL as a reconfigurable function unit on the system bus, to placing it on the peripheral bus as a co-processor.

For example, the Proteus Architecture[48] is an ARM based implementation with a reconfigurable ALU consisting of multiple reconfigurable function units. Applications load custom instructions at run-time to speedup execution. The commercial Triscend A7 device [161] has an ARM processor loosely coupled to a large reconfigurable fabric to provide co-processor functionality. This approach requires a more traditional hardware-software co-design methodology.

The software centric approach usually involves profiling the application and applying the 90:10 rule of thumb, which states that 90% of software execution time is spent in 10% of the code. The kernels representing 10% of execution time are then considered for implementation on the FPL.

2.3.5.3 Logic Centric

The dominance of the software centric approach to reconfigurable system design can in part be explained by the legacy of the general purpose microprocessor. However, it often fails to harness the massive parallelism possible with FPL since it centres around the serial instruction stream. Instead, it would seem more sensible to capture the design in a more logic-centric way, thus preserving its parallelism.

The logic-centric approach to system design elevates the FPL to be the primary vehicle of computation. In [27] Brebner describes a class of systems in which FPL is best suited to performing the computation with the microprocessor relegated to processing only exceptional cases. The illustrative example used is a gigabit IP router design in which line rate processing is performed by the FPL and only exceptional cases are handled by the microprocessor.

2.3.5.4 Summary

It is likely that reconfigurable embedded systems will demand elements of both software and logic centric design approaches. It is important for the designer to appreciate the strengths of each fabric so a good mapping within the JIT trade-off space can be made.

2.4 Platform based design

2.4.1 Introduction

The much heralded System on a Chip (SOC) design paradigm as originally conceived has largely failed due to problems with the integration of IP blocks[9][16]. Platform based design may provide an alternative route to SOC. Here we outline what a platform is and review some of the opportunities and challenges it brings to the embedded system designer. We structure the review using the headings: Flexibility, Heterogeneity, Scalability, Productivity and Abstraction.

2.4.2 What is a platform?

We define the platform here as a standardised programmable architecture and its associated abstraction model. The term standardised describes the ability to target many applications, perhaps from a particular domain, at the architecture. The term programmable emphasises that a level of flexibility exists such that function blocks and the connections between them can be configured post-manufacture. Finally, the abstraction model provides a programmer's view of the architecture without unnecessary detail, but with enough visibility to allow fine-tuning. For example, the use of Matlab and Simulink in the Xilinx System Generator for DSP presents an alternative programming model for FPGAs from Hardware Description Languages (HDLs) such as Verilog. The platform we are interested in is the complete system platform - not simply the architectural design, but a full device manufactured as a standard part, or family of parts.

To further describe what is meant by the platform based approach to SOC design, it is useful to look at the environment in which it has been born. Two powerful forces are shaping the semiconductor industry today - the shrinking size of the time to market window and spiralling non-recurring engineering (NRE) costs. The gap between the level of chip integration possible and the productivity of a designer continues to grow. This results in it becoming increasingly difficult to meet the spike characteristic of market demand. For example in [151], Smith presents evidence that a 1 month lead in time to market can result in a 70% difference in revenue over an ASIC's lifetime. The second problem relates to the cost of manufacturing an ASIC. At each generation of Moore's law, the cost of creating the masks used in the manufacture of a chip grows exponentially. To justify this cost, the chip must be manufactured in large volumes. The way in which platform based design meets this challenge is by removing a lot of choice from the designer. Instead of starting with a blank piece of silicon, the designer is presented with a pre-designed flexible computing fabric to which he maps his system. The fabric is used across several different systems which both increases design reuse and amortises the NRE cost.

Many different platforms exist. Some are highly domain specific such as the Texas Instruments C64x DSP with its turbo decoder co-processor. Others, such as the Xilinx Virtex II FPGA family[180] are targeted more generally at networking and DSP applications. Figure 2.4 illustrates the high level of integration: microprocessors, multipliers, block RAM (BRAM) and giga-bit I/O transceivers within a 2-dimensional grid of LUTs and a rich interconnect fabric.

2.4.3 Flexibility

As discussed in Section 1.1, flexibility is now viewed as a design attribute in computer architecture. It forms one of the three axes in the JIT computing tradeoff space. If a platform provides too much flexibility then it is cost and energy inefficient. If too little flexibility is provided then it is not possible to map the required system functionality.

To explore what the implications of flexibility are, it is useful to view it from two perspectives - design-time and run-time. It is of course true that design-time flexibility must incorporate the required level of runtime flexibility, but examining the two sep-

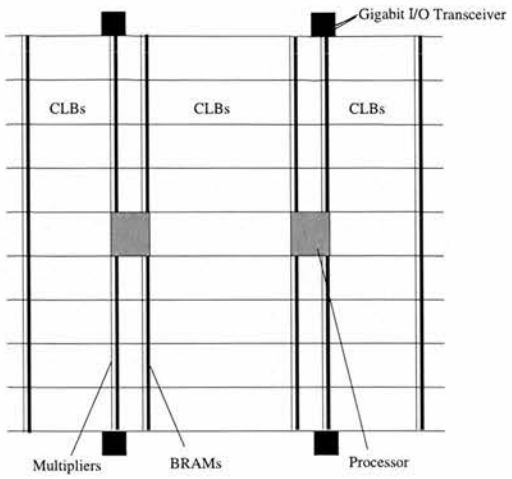

Figure 2.4: Xilinx Virtex II Pro Block Diagram

arately will help to shape this amorphous attribute. (It should be noted that a higher level of flexibility exists - portability between platforms. This is discussed under the heading of abstraction in Section 2.4.7.)

At design time, the prime focus is programming the platform to implement a system. The level of flexibility required is just enough to map the system to the platform, although the availability of extra flexibility eases the mapping as discussed by DeHon in [55]. At runtime, a level of flexibility must exist to satisfy the degree to which the system's functionality is runtime dependent. For example, the requirement to implement arbitrary algorithms demands high flexibility. A (design-time) well-specified data-independent system has a low demand for runtime flexibility.

2.4.4 Heterogeneity

Given some form of application domain, the platform architect's challenge is to create a solution which satisfies the domain's performance requirement, trading off cost, flexibility and energy. This leads to a heterogeneous solution. At the architect's disposal are three broad sets of computational fabric - processor, FPGA and ASIC. Within each of these sets are many further choices. One which is particularly wide is the choice of ASIC elements to include, beyond the mixture of computational fabrics, memory,

interconnect and I/O demand further decisions.

The challenges faced by the system designer targeting a platform are different from those of the SoC designer. The design freedom offered by the SOC created the possibility of a near optimal solution. In terms of value, this often led to a disproportionate amount of effort spent on architecture design over system design. With platform based design, the designer is restricted to the task of mapping rather than creation. One interesting concept this brings to the fore is that of algorithm fluidity, i.e. functionality may be moved between computational fabrics. In [88] the authors coin the term "software decelerators" to refer to functionality implemented in an FPGA logic fabric being shifted to a microprocessor. For example, given a mapping problem where FPGA logic resources are in high demand and microprocessor resources are under utilised then exploitation of such fluidity is useful.

2.4.5 Scalability

Scalability represents a highly prized but elusive goal of computing. The ability to scale a system through the addition of extra silicon resources with little additional design effort is very valuable. Its realisation is likely to run through the centre of a system from design capture right through to the implementation fabric. To be truly scalable (across fabrication process generations) the system description must be implementation independent.

A less demanding form of scalability is to confine its scope to a single process generation, for example, a system which ships in products of different configurations. If the platform targeted is offered as a family of device sizes, then it could be exploited with a scalable design, rather than a single large design which is capable of all product configurations.

2.4.6 Productivity

Productivity is increased right across the design flow when targeting architectural platforms. The productivity gain of reduced designer freedom has already been mentioned. Platform based design increases productivity by allowing the designer to concentrate

on system design rather than the details of the implementation fabric. The use of a common platform brings with it the opportunity for increased use of IP blocks. The central promise of the SOC design failed to reach potential - improved productivity through use of IP blocks. The problem was that design teams had to spend a great deal of time integrating IP blocks together. With a common platform architecture, the potential for straightforward integration exists, bringing with it the true potential of "plug and play" IP based design.

Perhaps the greatest productivity gain is reduced verification and test. With a standard platform architecture, the significant cost of physical test and verification is independent of the design and can therefore be amortised over all designs.

2.4.7 Abstraction

As platform based design envelops more systems, domain specific platforms will be created to better match the requirements of a particular group of systems. Apart from the obvious benefits of increased competition, the availability of many platforms presents the system designer with choice. Traditional design methodologies assume knowledge of the target platform, so the decision must be made early in the design process. Such a methodology ties the system to a particular implementation target, bringing with it several undesirable consequences, two of which are a) the choice of target platform may have been incorrect, leading to a solution with a sub-optimal point in the cost-energy-flexibility tradeoff space and b) platform architectures will evolve to exploit increased chip-level integration, making IP blocks tied to a particular architecture obsolete.

These problems demand design abstraction from the implementation platform. This is an active subject of research in the literature incorporating the large subject area of hardware-software co-design. Platform based design brings its own challenges which are only beginning to be investigated. The level of design abstraction required is likely to be higher than that of hardware-software design due to the heterogeneity of platforms. In [26][31] the authors propose and investigate the idea of Circlets: a circuitry description independent of its implementation, allowing rapid mapping to a concrete fabric. The level of abstraction is low, around the level of the 4-input LUT,

and is therefore limited in scope to homogeneous FPGA fabrics. It is possible to raise the level of abstraction, enabling the use of features such as hardwired multipliers, however the original aim of fast real-time mapping from the description to implementation may be lost. For example, in [74] the authors envisage the requirement for a system's design to remain platform independent during its entire life-cycle. Java byte code provides the processor abstraction and an RTL based byte-code[75] provides the circuitry abstraction. An interpreter at the client maps the RTL description onto the concrete FPGA fabric. The authors do not consider the (substantial) computation required to perform the mapping operation, although they do point out that it would be less than performing the full synthesis from a high-level VHDL description.

Others[79] propose the use of high-level software languages such as System C[123]. The design tools perform the task of mapping the system to an implementation platform. However, it is arguably counter-productive to describe algorithms serially in C and then expect the tools to (re)insert the required level of parallelism.

2.5 Approaches to harnessing a runtime reconfigurable FPGA

2.5.1 Introduction

The ability to reconfigure an FPGA at runtime has stimulated a substantial amount of research into harnessing this feature. In this section we provide a short introduction to methods reported in the literature for harnessing run-time reconfiguration. It provides the background for a major contribution of this thesis in Chapter 4,where we propose a new method for harnessing run-time reconfiguration in highly-concurrent real-time systems.

Section 2.5.2 outlines the principle of using an operating system layer of abstraction to enable multiple, perhaps independent, tasks to share a run-time reconfigurable FPGA. Section 2.5.3 describes the less flexible idea of temporally pipelining execution of a single system. Finally, Section 2.5.4 provides an overview of work reported in the literature on the simplification of circuitry at run-time through data-folding. All

these methods of exploiting run-time reconfiguration are complementary to one another. Their employment is highly system dependent, and this is something we discuss when motivating the framework proposed in Chapter 4.

2.5.2 Multitasking Operating System

Runtime reconfiguration is viewed by many as a technique to elevate the FPGA fabric to a first class computing resource. Work on the "virtualisation" of hardware hypothesises that it can be treated like a microprocessor - a time-multiplexed resource, managed by an operating system. The operating system typically runs on a microprocessor, hence maintaining its traditionally central role in computation. The main purpose of such an operating system is to provide an environment where tasks can execute concurrently with support for inter-task communication. In addition, the operating system is responsible for managing the reconfigurable resources in a consistent, efficient and fair way. In [121], Nollet et al. describe an operating system called OS4RS for reconfigurable resource management. The targeting methodology provides an executable for each computing fabric it may be run on (delaying hardware/software partitioning to runtime) and the operating system dynamically determines which fabric is to be assigned to a task. Since task pre-emption is supported, for example to allow the reassignment of resources, the operating system must instantiate/delete a task, suspend/resume a task, control inter-task communication and handle computing resource exceptions.

Task preemption can be costly, particularly in the case where the entire state of an FPGA circuit must be saved. Walder et al. avoid this cost by restricting their operating system model to non-preemptive reconfiguration[167]. This restriction means the tasks are assumed to have no precedence constraints and no real-time constraints to satisfy. In common with the system model of Nollet et al., asynchronous task arrival times and computation times are a-priori unknown, making them both general purpose operating systems. Task switching in a multitasking operating system is discussed by Simmler et al.[145]. They emphasise the importance of critical sections, i.e. time periods when a task must not be preempted. Section 2.6 discusses the major challenges of resource fragmentation management and task interface satisfaction, facing the use of runtime

reconfigurable FPGAs.

2.5.3 Temporal Pipelining

2.5.3.1 Introduction

Traditional circuit design involves making a tradeoff between time and space at design time. Runtime reconfiguration allows the tradeoff to be extended to runtime. Work in the literature often claims that by time multiplexing an FPGA at runtime, systems can be implemented using fewer resources[52][65][86]. It is possible to extract the highest capacity from an FPGA by fully pipelining every operation in the spatial domain. However, the throughput provided by a heavily pipelined implementation is not always required, for example when bottlenecks elsewhere in the system place a limit on the supply of new data. Temporal pipelining is a mechanism which uses available capacity to support more functionality at the expense of throughput.

2.5.3.2 Time Multiplexing

Time multiplexed FPGAs promise to improve logic density by time sharing logic [19] [159] [40][52] [152][85]. These devices allow the reuse of logic blocks and wire segments by having multiple configuration bits controlling them. A change of configuration at runtime can take of the order of a single clock cycle.

A technology mapped netlist contains some number of virtual LUTS which is larger than the number of real LUTs. The netlist is partitioned into sub-circuits, such that the logic in different stages temporally shares the same physical resources. Each stage is referred to as a micro-cycle and one run through all stages is called a user cycle. The outputs produced by a user cycle should be identical to a non time-multiplexed implementation. To store intermediate flip/flop values between micro-cycles, time multiplexed FPGA architectures often include some form of micro-register, for example [160][53]. This is critically important. Although the combinatorial logic can be multiplexed between functions, state cannot. Flip/flop state must be stored for subsequent stages to access the result.

The task of the scheduler is to partition the netlist into micro-cycles ensuring virtual

LUTs are evaluated in the correct order, and no stage has more virtual LUTs than there are real LUTs. A number of algorithms have been proposed for performing this task.

Levelised scheduling[19] orders the LUTs by the number of logic stages from the inputs. A fully levelised arrangement means the number of micro-cycles is equal to the length of the critical path in the netlist. The As Soon as Possible (ASAP) algorithm schedules each LUT as soon as its inputs are ready. As late as possible (ALAP) schedules each LUT in the micro cycle before its output signal is required. In [159], Trimberger notes that although ASAP and ALAP scheduling produce the correct results, they do not make efficient use of the time-multiplexed resources. Importantly, whilst the critical path defines the number of micro-cycles required, most functions are performed using many fewer micro-cycles.

The ideal scheduler would minimise the number of real LUTs required without increasing the number of micro-cycles. The product of such an algorithm would be a near uniform distribution of virtual LUTs across micro-cycles. Many different algorithms have been proposed to tackle this problem, including list-scheduling[159], network flow based multi-way partitioning[107] and enhanced force directed scheduling[39]. Schedule compression is an important part of targeting a time multiplexed FPGA[159]. When the number of logic levels in the critical path is greater than the number of micro-cycles in which it is to be implemented, the scheduler must compress the critical path. This involves merging multiple consecutive stages on the critical path into one stage. In [108] Liu and Wong describe an optimal algorithm for schedule compression.

2.5.3.3 Temporally Systolic Pipelines

Many applications involve applying a number of computational stages to a stream of data. One example is the Moving Picture Experts Group (MPEG) standard for video compression and decompression. The standard encoding sequence consists of motion estimation, motion compensation, discrete cosine transform (DCT), quantisation, inverse quantisation, inverse discrete cosine transform and inverse motion compensation. A conventional FPGA implementation would lay the pipeline out spatially, streaming data through the pipeline. That would be the design of choice if the most heavily pipelined throughput was required. If less throughput is required, the spa-

tially pipelined implementation would require the same number of resources but underutilise them. A temporally pipelined solution could stack the pipeline stages in time. In such a scheme, the lower throughput requirement is met using fewer resources[139].

2.5.3.4 Combined Temporal Pipelining and Resource Sharing

The time-multiplexing techniques listed in Section 2.5.3.2 are based upon a technology mapped netlist. Working on the design at such a low-level of abstraction misses the opportunity to exploit certain design freedoms, such as, the tradeoff between the number of reconfigurations and the resource sharing of higher level function units[37][162][124]. At the architectural synthesis stage, resource sharing within micro-cycles and the cost of communication between stages should be considered in addition to the traditional space/time tradeoffs.

Cardoso observed[37] that the majority of approaches which use a high-level description as their input, perform partitioning and high-level synthesis using two separate algorithms, one after the other, for example [162][124]. Performing synthesis on partitions chosen without formal consideration of resource sharing potential may lose optimum opportunities for sharing. In [37] Cardoso integrates partitioning and resource sharing into one algorithm. It is demonstrated through a number of examples to produce good results. However it is worth noting the principal comparison is made with ASAP levelised scheduling in [66] and not best-in-class previous approaches such as [108].

2.5.4 Data Folding

Circuit specialisation techniques such as constant propagation are commonly used in digital circuit generation to reduce resource count and latency. In a run-time reconfigurable FPGA, these techniques can be extended to run-time[172]. One of the most common methods reported in the literature is the propagation of constants or data folding[63] within arithmetic operators. The circuit specialisation to the algorithm and data-set maximises performance from limited resources. Many digital systems contain operators on which data folding can be applied. For example, digital filters often have constant coefficients which may be folded into the arithmetic

circuitry[87][128][68][117] [70][41]. A dedicated IIR filter was shown by Chou et al. to require half the logic resources of the general purpose equivalent[44]. Other reported examples of circuitry specialisation through data folding include string sequence matching[96][63][73], content addressable memory (CAM)[33] and encryption [129].

2.5.5 Summary

A substantial body of work exists in the literature on harnessing the run-time reconfigurable ability of an FPGA. It is possible to divide the approaches to run-time reconfigurable system design into two sets. One set is software centric (Section 2.3.5.2), and the other set is logic centric (Section 2.3.5.3). The software centric approach has evolved from research on compilers and parallel architectures, and the logic centric approach has evolved from Hardware Description Languages (HDL) and logic synthesis.

In this section we have outlined the major strategies proposed in the literature for harnessing runtime reconfiguration of FPGAs. They all may be exploited by both hardware centric and software centric design approaches.

2.6 Resource Fragmentation and Interface Satisfaction

2.6.1 Introduction

The aim of this section is to survey approaches to tackling the two greatest challenges facing runtime reconfigurable computing on an FPGA: fragmentation and interface satisfaction.

2.6.2 Fragmentation

Early research on harnessing run-time reconfiguration proposed a paradigm analogous to paged virtual memory systems[25][23]. The idea was that circuitry could be swapped on and off the fabric at run-time, creating the concept of a virtual hardware resource. As circuitry tasks arrive and depart, the available resources become fragmented, reducing the ability to place new tasks. However, it quickly became apparent

that the problem of resource fragmentation was a major obstacle to the paradigm's realisation. Here we discuss work in the literature on approaching the problem of resource fragmentation. Consequently, the work surveyed is heavily FPGA based, although the same principles can be applied to other reconfigurable fabrics.

Several researchers have investigated the problem of efficiently managing an FPGA fabric as 2-dimensional circuits are swapped on and off. The proposed techniques usually involve introducing constraints on the task. For example, its bounding box must be rectangular as in [56] by Diessel et al. However, it is NP-complete to decide whether or not a set of rectangular tasks can be placed on a grid without overlap[97]. The techniques therefore seek efficient heuristics to manage fragmentation such as task transformations (Compton et al. [47] and Burns et al. [34]) and local repacking (Diessel et al. [57]). In [14] Bazargan et al. investigate data structures and algorithms for fast runtime placement of tasks. They report on simulation experiments using variants of bottom-left, first-fit and best-fit bin-packing algorithms.

A more radical approach is to superimpose a one-dimensional view of the FPGA. In such a model, tasks occupy a column which stretches the entire height of the FPGA and have variable width. This technique has been applied by several devices, for example PipeWrench[138], GARP[79] and DISC[170]. Brebner and Diessel [29] describe how the one-dimensional view leads to simple allocation and de-fragmentation on an FPGA. Inter-task communication can be provided either by abutting tasks or by routing signals via an interconnecting bus. It should be noted though that there is little supporting work on circuit synthesis when the area occupied must be an integral number of columns. The importance of the solution to this issue becomes particularly apparent when one considers the hierarchical chip-oriented interconnect of modern FPGAs[180].

Another approach is to divide the FPGA fabric into a set of predefined rectangular tiles[115]. An operating system schedules tasks to these tiles, based on a task allocation table that contains information on currently loaded tasks. In a trivial example application [120] Nolett et al. divide the Xilinx Virtex II into 2 tiles. The most obvious disadvantage of using a pre-partitioned FPGA is that it results in a fixed number of fixed sized tiles. In the case of a size mismatch between the size of a task and the

size of a tile, area is wasted. To solve the size mismatch problem at runtime is not appealing. For the case where tasks are significantly smaller than tiles, Nolett et al.[120] propose the use of a multiplexer block. The multiplexer adds an extra abstraction layer that allows the placement of several smaller tasks into a single tile. Its job is to perform port masquerading on the tile communications port. Other examples of tile based architectures include aSoC [104], RAW [166], FPFA [82] and Pleiades [6].

2.6.3 Interface Satisfaction

Routing of configurable systems can take many hours, which is unacceptable given runtime reconfiguration times of sub-second duration. There are however less generalised forms of run-time reconfiguration which make sub-second duration feasible. Here we review the concept of an interface in system design relating it to run-time reconfiguration and outline work in the literature on its satisfaction in run-time reconfigurable systems.

Most system design employs some form of hierarchical abstraction: the top level system is composed of subsystems, which in turn are composed of further subsystems. At each level an interface specifies a well defined method of communication. In a run-time reconfigurable system, interface satisfaction presents a challenge. At the highest level, the system's external interface must always be satisfied despite the fact that it may be undergoing reconfiguration. This means that systems external to those being reconfigured can continue to operate, unaffected and without any knowledge of the reconfiguration. This is simply achieved through the use of a memory buffer. As tasks are moved and swapped on and off the fabric, their interface with other tasks must be satisfied. At the lowest level, within a task, the interface between concrete resources must be satisfied.

It is possible to perform run-time routing by operating on circuitry at a very low level using tools such as Brebner's SPODE [24] and Xilinx's JBits [72]. However, it requires intimate knowledge of the architecture and involves more effort than the use of higher-level descriptions. With the exception of such tools, it is generally deemed impractical to attempt arbitrary routing within an FPGA fabric at runtime. For this reason, the implementation of a task is usually design time fixed [25][120]. While this

has implications for task relocatability and transformability as described by Walder et al. in [167], it removes the problem of run-time interface satisfaction within a task. A number of different ideas have been proposed for satisfying the inter-task interface. Here we briefly describe a representative collection of work from the literature.

Brebner and Donlin[30] present three models of inter task communication, with the network on chip model outlined among others by Marescaux et al. [111] making four in total:

- Fixed wiring
- Reconfigurable switch, e.g. crossbar or bus
- URISC data flow
- Network on chip

The fixed wiring model is suitable for a collection of tiles requiring a fixed, regular interconnection pattern. The reconfigurable switch offers more runtime flexibility, enabling communication either in parallel through a crossbar or in serial through a bus. For example in [62], Eggers et al describe a design containing a 32x32 cross bar which can be reconfigured in 700ns. The URISC data flow model refers to Donlin's work [60] on the flexible Ultimate RISC processor which has a single instruction type - move. The principle is that tiles are connected to the processor's system bus and the processor's sequence of instructions directs inter-task communication. There is no need for a physical realisation of the system bus since an interface such as the XC6200 FastMap interface gives random read/write access to the FPGA configuration memory. The network on chip as proposed by Dally et al. [49] may be used to connect tiles together, with established techniques from network routing used to enable interface reconfiguration.

In summary, there exists a continuum of runtime circuitry interfacing techniques from the fully parallel to serial approaches, trading off flexibility with performance.

2.7 Techniques to Speedup reconfiguration

2.7.1 Introduction

The time taken to perform reconfiguration of an FPGA has a significant impact on the performance of reconfigurable systems. Here we review techniques reported in the literature for reducing the time spent waiting for reconfiguration to complete. This serves as background to the configuration compression technique developed in Chapter 3.

Before we discuss specific architectures and techniques for reconfiguration, it is worthwhile considering reconfiguration at a very high level. Viewed at the system level, a feeling for the effects of specific improvements can be established.

The majority of commercial FPGAs hold their configuration in SRAM. This means that all configuration must be stored off chip in non-volatile memory. The static use of an FPGA involves loading the configuration data from off-chip memory into the device once at system power-on. The run-time reconfigurable use of an FPGA must load configuration data from off-chip storage multiple times during system operation in addition to the initial power-on load. This can significantly affect the system's performance, as time spent reconfiguring is time during which no computation is performed. We acknowledge that it is possible to generate configuration data on-chip, and therefore not require it to be loaded from off-chip memory, but this is a special case, and for this discussion is not considered.

The speed of loading the configuration data depends on the off-chip interface and the length of the bitstream. The interface performance varies according to the number of I/O pins on the chip package dedicated to reconfiguration and the frequency at which the pins may be clocked. The length of the bitstream depends on the size of the chip. Section 3.7.3 discusses the bandwidth of the configuration interface of commercial FPGAs, and the effects of increasing on-chip integration. The time to transfer configuration bits from off-chip to on-chip memory is expressed in equation 2.1. ω is the reconfiguration interface bandwidth in bits/s, L is the number configuration data

bits and K is a constant overhead associated with the architecture.

$$T_{load} = \frac{\omega}{L} + K \tag{2.1}$$

As can be seen from equation 2.1, reducing the amount of data that must be loaded to perform reconfiguration will reduce reconfiguration time. This is an area which has received much attention in the literature. The two major ideas for exploiting the relationship are configuration compression, reviewed in Section 2.7.1.1, and configuration hierarchy, reviewed in Section 2.7.1.2.

As proposed at the start of this section, we shall now look at reconfiguration at the system level rather than the low-level detail of equation 2.1. Instead of demanding the loading of configuration data from off-chip storage as quickly as possible, it is possible to exploit static and dynamic scheduling methods to load configurations in parallel to computation. With the additional on-chip RAM, configuration data can be loaded into a staging area, ready to take advantage of the massive on-chip parallelism to the active configuration memory when reconfiguration is triggered. The major ideas for exploiting additional on-chip memory are the configuration cache, discussed in Section 2.7.1.3, configuration pre-fetch, discussed in Section 2.7.1.4, and configuration cloning, discussed in Section 2.7.1.5.

2.7.1.1 Compression

Several researchers have investigated the compression of FPGA configuration bitstreams [50][98][78]. The idea is to compress the bitstream at design time, so that at run-time it takes less time to load from off-chip RAM through the narrow configuration interface. Once the compressed bitstream is on-chip, it is decompressed in preparation for loading into the active configuration memory.

Compression algorithms can be divided into two categories - lossless compression and lossy compression. In lossless compression, no original information is lost, so a perfect replica of the original data can be produced. An example of loss-less compression is the Lempel-Ziv (LZ) algorithm, used in the Unix gzip utility. Lossy compression results in some original information being discarded. The result of lossy compression is a representation which is similar but not identical to the original. A

good example of lossy compression is the Moving Pictures Expert Group (MPEG) movie compression standard.

In general, lossy compression produces better compression ratios than loss-less compression. Dandalis uses an LZ based approach to compress Xilinx Virtex bitstreams in [50] producing a results which is 59%-89% the size of the original. In [102], Li and Hauck exploit features within the bitstream to propose a technique which achieves a compression result which is on average 25% of the size of the original. In the remainder of this section we will outline some of the techniques proposed in [102].

In [102] the authors review the regularity and suitable symbol length for the Xilinx Virtex bitstream. The main source of regularity was among bitstream sections, or frames[177], which configure similar resources. Regular systolic operations like adders are particularly good for this sort of regularity. The authors best compression technique uses each configuration frame as a fixed size dictionary for the LZ algorithm across all other frames to discover inter-frame regularities. The frames are then optimally ordered for loading into the FPGA. To reduce the size of the dictionary required by the LZ algorithm, the authors use read-back to read frames already present back from the configuration memory.

Another compression technique proposed in [102] uses wild-carding[78]. Wild-carding involves loading only the difference between a frame and an already loaded frame, rather than loading the entire frame. Results for wild card, huffman coding, arithmetic coding and a few variations of read back technique are presented. The read-back algorithm gives the best compression factor of about 4 across a range of circuits.

2.7.1.2 Hierarchy

The concept of a configuration hierarchy has been investigated by a number of researchers [122][120], most notably Schaumont et al. in [137]. The basic idea is that different levels of abstraction beyond the configuration bitstream form a hierarchy of reconfiguration. The authors use the example of a software protocol stack implemented on a processor which is in turn implemented on an FPGA fabric. They point out that the protocol stack is clearly a program as it consists of soft instructions. Then at another level, the FPGA views the program as data to be processed by the reconfigurable

fabric. Schaumont et al. describe how the configuration hierarchy allows control of system complexity, while creating more opportunities for component reuse. In [122] Ogrenci et al. demonstrate that it is possible to reduce the number of configuration bits required for reconfiguration by providing pre-placed coarse grain computation blocks.

The configuration hierarchy is a similar concept to the terms "coarse grain" and "fine grain"[76]. Fine and coarse grain are used to describe features at the architectural level - e.g. 4-LUTs or 4-bit ALUs. To configure a device composed of 4-bit ALUs to perform an 8 bit subtraction is likely to require much less configuration data than the configuration of a 4-LUT based device.

In summary, the reconfiguration hierarchy can speedup reconfiguration through the use of abstraction beyond the underlying architecture. Inserting additional flexibility at the circuit level as described by the bitstream reduces the amount of data that must be loaded to change functionality at run-time.

2.7.1.3 Caching

Configuration caching, is, as its name suggests, very similar to caching instructions on a microprocessor. The idea is that configurations are retained on-chip to reduce the amount of data that must be transferred over the configuration interface. In general, caching takes advantage of two principles - spatial and temporal locality. Spatial locality captures the phenomenon that circuits close together in sequence are typically required close together in time. Temporal locality expresses the principle that circuits accessed recently are likely to be accessed again in only a short while.

In [100] the authors examine the application of configuration caching to coupled processor-FPGA systems. Such systems treat the FPGA fabric as a virtual resource managed by an operating system. During execution, circuits are paged to and from the active configuration memory like pages moved between disk and memory in a microprocessor. These circuits are referred to as swappable logic units (SLU)[25]. The authors observe that circuits have a non-uniform size, unlike memory pages for a processor, making the time to load a circuit variable. Another problem is that the large size of configurations means that only a small number can reside on the chip simultaneously.

Another dimension to the FPGA caching problem is the FPGA's configuration architecture. The best cache solution is different for single context, partially reconfigurable and multiple context FPGAs. For example, to load an SLU onto a single context device involves loading an entire configuration bitstream from external memory. This has a significant latency penalty, during which no computation is performed anywhere on the FPGA. The partially reconfigurable FPGA allows SLUs unaffected by a reconfiguration to continue to operate. Only the area being reconfigured performs no work, and the latency is reduced since only the SLU's configuration is loaded, not the entire device bitstream. The multiple context device is more flexible again, enabling any FPGA resource to be reconfigured in the background to active computation, ready to be switched to the active configuration in the order of a clock cycle.

The authors present a set of configuration cache management algorithms, each tuned to a specific architecture. For the single context device, off-line algorithms such as simulated annealing attempt to extract maximum value from a context change. Maximum value refers to the opportunity to make changes in addition to those requested, since an entire bitstream must be loaded anyway. For example, the required changes together with the predicted future changes can be combined into a single bitstream to reduce the overall reconfiguration overhead.

The multi-context FPGA must also maximise SLU grouping, since its atomic unit of configuration is also an entire context plane. Li et al.[99] recognise that the decision as to which context plane to replace on a context load is very similar to general caching. They use the Belady[15] algorithm from the operating system and architecture fields. Belady operates on the principle that the most likely context to be replaced is that which is least likely to be accessed in the near future. Combining Belady and SLU grouping techniques, a complete, off-line, predictive caching approach is developed.

The partially reconfigurable FPGA presents the most difficult challenge for cache algorithm design. It is directly affected by the different size of SLUs and the different times required to load them. Li et al. present a number of algorithms. The off-line simulated-annealing and Belady based algorithms allow difficult problems to be solved in advance, but lack flexibility. Run-time algorithms must tackle the difficult problems of relocation and de-fragmentation. Again, Li et al. find the operating systems field to

have an algorithm fit for the task - Least Recently Used (LRU). However, they observe that the non-uniform size of SLUs is not considered by the LRU algorithm. To solve this problem, the authors propose a variable credit system, in which both an SLU's size and frequency of invocation are used to determine the evictee.

In [154], Sudhir et al. apply algorithms developed for web caching[36]. The most significant difference between their configuration caching work and that of Li et al. is their eviction policy. Li et al.'s most sophisticated algorithm is based upon a penalty system. It combines the distance of last occurrence, frequency of occurrence and SLU size to determine whether it should be evicted or not. Sudhir et al. consider these factors in addition to other execution history information, specifically, how far into the future the SLU is likely to be used again. In short, the algorithm looks both at an SLU's past and predicted future requirements to make its eviction decision. They demonstrate through simulation that this further reduces configuration overhead across a set of benchmarks.

2.7.1.4 Pre-fetching

Run-time reconfigurable systems execute multiple configurations to perform a task. If the computation performed by a configuration runs to completion before a new configuration is loaded, then a stall occurs while the new configuration is loaded. Configuration pre-fetching complements configuration caching by loading the next configuration before it is actually required. The ability to load a configuration into all or part of the FPGA whilst the rest of the device continues to operate, overlaps reconfiguration latency and useful computation. It is therefore capable of virtually eliminating reconfiguration latency in some systems.

The main challenge for configuration pre-fetching is to predict far enough in advance which configuration will be required next. Many algorithms have complex control flows and data dependencies, which make prediction, and hence pre-fetching, difficult. Pre-fetching is used in many other computing fields. For example, microprocessors pre-fetch data from memory into the processor cache, and virtual memory systems pre-fetch data from disk into the main memory[35]. However, these systems often benefit from regular access patterns, which reconfigurable systems do not enjoy.

In [77], Hauck considers pre-fetching in a tightly coupled FPGA co-processor similar to PRISC [133]. Execution occurs on the microprocessor until an instruction to be performed by an SLU is encountered. At this point, the processor checks whether the SLU is loaded into the FPGA, and if so it executes it. Otherwise, the SLU configuration must be loaded, during which time execution of the serial instruction stream is on hold. To avoid this stall in execution, a program running in the reconfigurable system can insert configuration pre-fetch instructions. These occupy a single slot in the microprocessor pipeline.

Hauck's algorithm for determining where to insert pre-fetch instructions starts with a control flow graph of the algorithm being considered. The graph contains information on the potential of execution paths within the program and thus forms the basis for predicting which SLU will be required next. The principle upon which the algorithm operates is that the SLU required next is the one which can be reached in the least number of cycles. This again exploits the principle of locality, placing high priority on loop kernels.

In [103], Hauck considers the application of pre-fetching to another FPGA configuration architecture. The architecture, proposed by Compton in [45], is the partially reconfigurable FPGA with relocation and de-fragmentation facilities. It was designed to improve hardware utilisation. Final placement of a configuration within the FPGA may be determined at run-time and de-fragmentation provides a method to bring together unused FPGA resources without unloading useful SLUs. Three principal algorithms are investigated: static, dynamic and hybrid form pre-fetching. Static occurs at compile time, similar to the method described above. Dynamic pre-fetching makes the decision as to which context to load at runtime, based upon certain runtime variables. Hauck's algorithm uses a Markov process to determine which SLU to load. Finally, the hybrid solution combines the recent access history provided at run-time with the analysis performed statically. As is usually the case, the hybrid approach combines the best of both worlds to achieve the best results.

2.7.1.5 Cloning and Sharing

In a similar vein to the use of existing configuration memory content as a dictionary for

compression algorithms, configuration cloning and sharing use existing configuration memory content to reduce data transfer over the configuration interface.

Configuration cloning[127] exploits regularity in circuitry by copying configuration regions from one part of the device to another. The main motivation is to exploit temporal and spatial locality of circuits and architectures already present in the FPGA. The area of application proposed by the authors is regular circuit structures such as FIR filters. The idea is that such circuits have dynamically bound iterative constructs - loops in software and regular structures in circuitry. Cloning provides a simple way to increase the length of a circuit, for example, window size, cryptographic key length and pipeline depth. Cloning involves copying the configuration bitstream from one region of the FPGA to one or many other locations.

Minor irregularities exist in what are otherwise highly regular circuit structures. This is acknowledged by the authors, but a convincing solution is not provided. In addition, it is not clear how widely applicable the technique is - particularly to automatically generated circuitry. Control circuitry is unlikely to be amenable to configuration cloning, leaving only highly regular (perhaps hand-placed) datapaths to exploit the technique. Finally, the system-level impact of providing architectural support for cloning is not considered. The silicon area increase required to address memory at the fine level of granularity demanded by cloning may outweigh any resource savings achieved.

Configuration sharing is very similar to cloning. Sezer et al.[141] combine signal flow graphs (SFGs) and configuration data graphs (CFGs). This is designed to produce groups of similar circuits, reducing the amount of configuration data that needs to be loaded to perform reconfigurations.

2.7.2 Summary

Borrowing upon a few decades of research in the fields of computer architecture and operating systems, the reconfigurable computing research community has developed a number of techniques to reduce the overhead of changing configuration at runtime. Many of the techniques may be combined together with architectural changes to speedup execution time.

2.8 Summary

We have provided an overview of the research areas to which this thesis contributes. The FPGA has been introduced in Section 2.2 and its ability to be run-time reconfigured is outlined in Section 2.3. The emerging system-on-chip design paradigm of platform based architectures is outlined in Section 2.4. Section 2.5 introduces some of the major areas of research in the literature into how a reconfigurable FPGA is best harnessed. We then describe the two greatest challenges facing the run-time reconfiguration in Section 2.6, namely interface satisfaction and fragmentation. Finally, Section 2.7 outlines techniques proposed for reducing the overhead imposed on execution time by reconfiguration.

Chapter 3

FPGA Reconfiguration Architecture Design Space

3.1 Introduction to methods for FPGA fast reconfiguration

Modern FPGAs have configuration bit stream sizes in excess of several megabits. When compared to the micro-processor which has a context of only a few registers, the FPGA must move a very large volume of data to switch context. This time to switch context has a direct impact on the level of performance achievable with run-time reconfiguration and is therefore an area of active research. Some demonstrations of run-time reconfigurable logic waste from 25% [171] to 70% [164], even as high as 94% [156] of their computing cycles stalled while the FPGA is reconfigured.

Traditional FPGA architectures have primarily been programmed with the bit-stream loaded serially into a single context configuration memory. This architecture allows only one configuration to be loaded at a time and requires a full reconfiguration for each change. Designers of runtime reconfigurable systems have found the single context device very limiting.

In some applications only part of the FPGA fabric is occupied, or a small part of the fabric needs to be changed, therefore partial reconfiguration of the array is desired. In partially reconfigurable FPGA fabrics, the configuration memory is accessed like a

RAM device. Selective addressing of the memory allows changes to be made as required without the need to load the entire bit stream. It is often possible to perform reconfiguration of a section of the device and to allow another portion to continue operating undisturbed. The Xilinx 6200 [176] configuration memory access is extremely flexible allowing changes to be made at a very fine level of granularity. Partial reconfiguration is also possible in the Xilinx Virtex [177] but at the more coarse grain column addressing level. In contrast to the single context device, the multi-context configuration architecture has multiple memory bits for each configuration bit as in [52][160]. The memory can be visualised as multiple planes of configuration information. One plane can be active at a time, with switching between active context planes performed very quickly - of the order of a single clock cycle. The inclusion of multiple on-chip memory context planes has a significant area cost compared to the single context device.

As discussed in Section 2.7.1.1, a number of other interesting techniques, some adapted from microprocessor architecture design, have been applied in reconfigurable FPGA systems. Configuration caching [99] takes advantage of both temporal and spatial locality like the microprocessor model, but is complicated by the non-uniform latency of configurations and the limited number of configurations which can reside on-chip simultaneously. Configuration pre-fetching [77] is a simple technique where the configuration is done in parallel with useful computation in order to hide its latency. The difficulty with this technique is predicting soon enough in advance what needs to be loaded, particularly in general purpose systems where control flow can be very complex.

Whilst all of the above techniques can be combined to reduce the effect of the off-chip configuration bandwidth problem, it still ultimately has a limiting effect on the application of run-time reconfiguration. Even in the multi-context device with its extremely small context switch time, the number of contexts imposes a limit on how much benefit this brings. The cost of including extra configuration RAM on-chip must also be considered when looking at the multi-context device. The distributed nature of a context memory plane leads to an expensive RAM implementation which can take up 30%[89] of the entire silicon area. More fundamentally, the question of how useful

very rapid reconfiguration is as a system design technique has still to be satisfactorily answered.

Configuration compression has been successfully applied to FPGAs [101][78][102]. This is similar to conventional compression algorithms, reducing the time taken to load a configuration by taking advantage of regularity and repetitions within the data. It has been demonstrated to reduce bitstream size by up to 85% [102] on the Xilinx Virtex architecture.

The compression algorithm designed by Li and Hauck[102] exhibits the best reported performance on modern FPGA devices, but it is designed to operate on entire bitstreams and so is not applicable to partial reconfiguration. In this chapter, new techniques to overcome the off-chip configuration bandwidth bottleneck will be investigated. We start the chapter by providing an overview of the Xilinx Virtex architecture in Section 3.2. In Section 3.3 we approach the problem of speeding up reconfiguration by performing an analysis of what exactly occurs during reconfiguration of a modern fine grain FPGA. This analysis provides the insight used to propose the overlay technique in Section 3.4. We then perform some further analysis based upon these results in Section 3.5 and use this to develop the compression algorithms presented in Section 3.6.

3.2 Xilinx Virtex Architecture

The Xilinx Virtex device family has been chosen for analysis in this chapter. It became established as the leading commercial FPGA architecture and has now been succeeded by the Virtex II family. This section will introduce the architecture in the detail required to understand the analysis performed.

Figure 3.1 shows the designer's high-level view of the Virtex architecture. It should be noted that it is not drawn to scale, nor should the relative locations of architectural features be taken as an accurate representation. It is a fine grained programmable logic architecture composed of 4 input lookup tables (LUTs) clustered into islands called configurable logic blocks (CLBs). The CLBs offer a rich interconnect fabric to route signals between the different architectural features. The connections are a mixture of

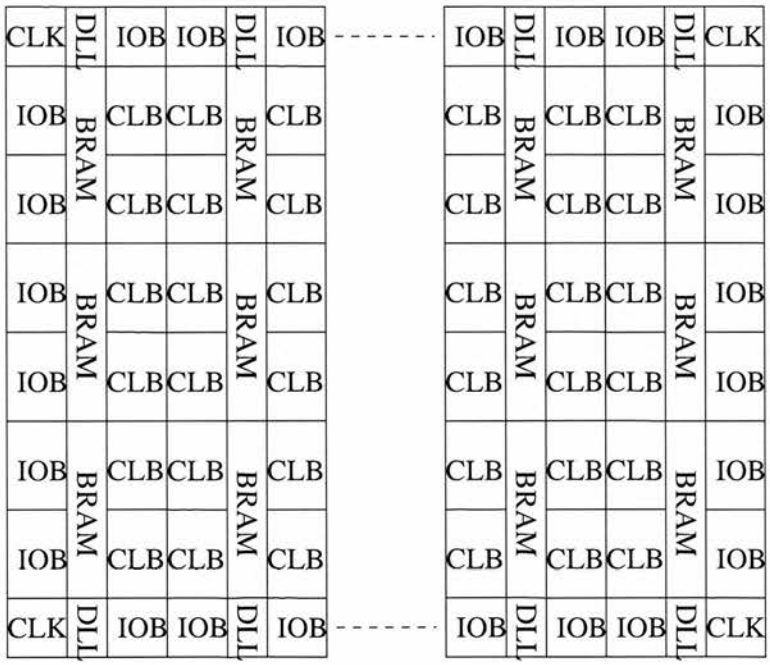

Figure 3.1: Xilinx Virtex Architecture

short nearest neighbour, 16 cells away and chip length wires.

The embedded dual-port block rams (BRAM) each provide 4Kbit of storage. Delay-locked loops (DLL), clock (CLK) and input-output blocks (IOBs) are also marked in the diagram for completeness. More details can be found in [177].

3.3 Reconfiguration Analysis

3.3.1 Introduction

To develop a better understanding of reconfiguration this section sets out to analyse exactly what occurs when an area of fabric is reconfigured. The principal idea driving the analysis is that there is a large amount of redundancy present in modern FPGA architectures to provide their high level of flexibility. This is believed to be an area of potential for exploring new methods to speed up reconfiguration, two of which are bit-stream compression and configuration overlaying. These are explored in later sections.

This section describes the JBits API used to write software for all the analysis work and the circuits used for experiments.

3.3.2 JBits API

To perform a partial reconfiguration of the Virtex, a partial bitstream must be generated. The partial bitstream contains the frames that are different from those already in the configuration memory. Generation of the partial bitstream is not conceptually supported by traditional hardware design environments and so instead is done using other tools. The JBits API is a Java library that provides low level access to the Virtex bitstream at the level of individual LUT contents and multiplexer configuration bits. Relocatable cores, a net router and a net tracer combine to provide a set of high level functions and utilities that make JBits a very powerful development tool. It can be used to edit bitstreams generated by the traditional Xilinx tool flow or it can be used to build designs independently. It was built with the specific aim of giving the hardware designer the low level bitstream access required to perform partial reconfiguration. It provides an easy to use mechanism to keep track of all frames that become "dirty", i.e. modified, as changes are made to the bitstream. Once all the changes have been made the frames marked as dirty can be written out as a partial reconfiguration bitstream.

3.3.3 Circuits Used

The circuits used for experiments were specifically selected to represent the typical types of function for which the Virtex FPGA is used. It is important not to use artificial circuits generated by research tools as is often done in the literature since this may cause uncertainty to be cast over any results. The circuits vary in size from 528 slices to 2320 slices and were placed and routed automatically with their area minimised. They were collected from Xilinx application notes and respected designs in either Verilog or VHDL source form in addition to the Xilinx Core Generator. The Java software written to perform the experimentation using the JBits API operates on the circuits after they have completely passed through the traditional Xilinx tool flow.

3.4 Overlay Technique

3.4.1 Overview

An active area of fabric may be loaded with some of the waiting configuration settings while it is still operating, without interfering with its functionality, reducing the number of changes that must be loaded, and hence reducing the time the area of fabric is off-line, when a reconfiguration is necessary. This is possible because of the high level of redundancy present in modern FPGA routing architectures - many configuration bits in any single mapping will have no effect on the circuits' operation and can therefore be set appropriately for the next circuit. The concept is illustrated in Figure 3.2, with the white square representing the original circuit to be reconfigured. The shaded middle square represents the original circuit still operating as it did but with the fabric it occupies loaded with some of the next circuit's configuration data. The resulting black square represents the second circuit with some of the white circuit's configuration bits still present - whichever do not interfere with the second circuit's operation. The configuration sequence of overlaying those configuration bits which do not affect the operation of the original circuit is called the advance configuration bitstream. The second configuration sequence is referred to as the residual bitstream as it contains the remaining changes required to switch the area of fabric to the second circuit. Since the changes made in the advance stage do not affect the operation of the white circuit it can continue to operate without interruption. The residual changes must be applied with the fabric off-line to avoid unpredictable behaviour. Off-line may mean that either the area of fabric is not clocked or its I/O is removed.

3.4.2 Algorithm and Implementation

Due to the organisation of a column of Virtex CLB's configuration data into frames, a change to a CLB anywhere in the column results in most frames in that column having to be written to the device. This is because a CLB's configuration data is split across all frames in the column. So even if the hypothesis above proved to be correct, it would result in very little change in the size of the residual bitstream when compared to the complete bitstream because, even if only a very small fraction of the bits must be

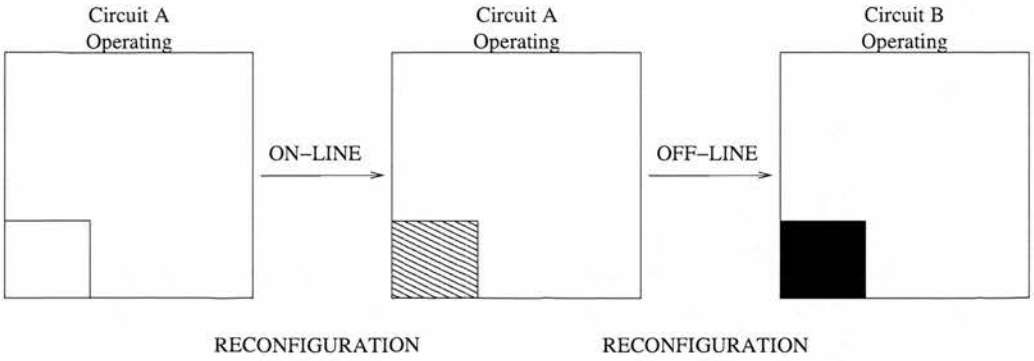

Figure 3.2: The overlay technique configuration sequence.

written, they are likely to dirty many frames. So with the existing frame organisation of the Virtex configuration architecture, the technique will not provide much benefit. With this in mind, a modified implementation of the algorithm was written which would allow analysis to be performed independently of the present configuration architecture.

The modified implementation still makes use of the JBits API, however it is much more involved since some of the more advanced functionality of JBits is tied to the underlying configuration architecture and therefore hides a lot of detail. For this reason JBits could not be used to produce the analysis and results required, so independent methods of tracking the exact position in the bitstream where changes occur had to be developed. This is all additional support software to the top level algorithm of performing the actual overlay algorithm.

To facilitate test and debug, the normal generation of bitstreams by the JBits system is performed with the additional detail captured in parallel. This was achieved through the use of wrapper functions around the JBits API calls. The wrappers provided a mechanism to intercept JBits calls within the java code and to manipulate various user defined data structures before and after the JBits API method is called. Recording the before and after settings of resources has to be translated into location of changes within the bitstream for analysis.

A piece of software was written using the JBits API which takes two bitstreams and the coordinates of the rectangular area of fabric to be reconfigured. The output is the advance and residual bitstreams. The remainder of this algorithm and implementation

section will first describe the platform developed for capturing bitstream changes made by the JBits API and then the overlay algorithm.

The circuits supplied are read in and preprocessed before the overlay algorithm begins. Every possible source in the chip - i.e. output pins in all CLBs and all IOBs are traced using the tracer class. If a netlist is sourced by the pin traced, it is stored for further processing. Each netlist has its source and sink pairs extracted and recorded in addition to every wire segment being extracted for easy lookup during the overlay algorithm stage. Care is taken to record wire names consistently, since a wire may be referred to by two different names depending on the CLB in which it is being referred. For example, a wire NORTH12 will be referred to as SOUTH12 in the CLB above yet they are the same entity. As explained above, to record changes made by the JBits API, wrapper methods record the resources setting before and after the API call. Subsequent changes to the same resource only record the new setting, leaving the 'before' setting as it was before any changes were made. This allows all changes made by the overlay algorithm to be minimally found at the end of the bitstream manipulation by comparing each resource's final setting to its original setting.

The position of a resource's bits in the configuration bitstream is not well documented by Xilinx and the company was not willing to volunteer the mapping translation when approached directly. Instead, using the acquired low level knowledge of the architecture, a process of experimentation and intuition lead to the correct mapping being established. The unusual relationship between configuration bits and their location in the bitstream was very apparent during this investigative work. Eventually two variables were found to combine to produce the bitstream location translation - the tile in which the resource is located (an architecture level abstraction not normally exposed to the FPGA designer) and an undocumented 3D array in the JBits resources class combined with some relative addressing.

The generation of the overlay bitstreams consists of an action phase and a verification phase. The verification phase is necessary to ensure that the resulting circuit operates as it should. The remainder of this section will first describe the generation of the bitstream for the advance configuration stage and then the generation of the residual bitstream.

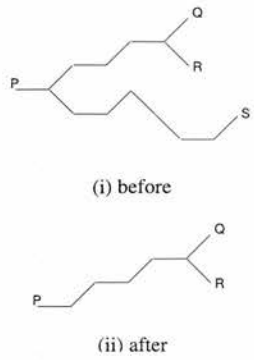

Figure 3.3: A netlist being trimmed

In figure 3.2, wires used by circuit B that are not used by circuit A are configured in the advance bitstream. The verification stage checks each netlist in the resulting bitstream to ensure that it has the same source/sink pairs that it did before modification. If a new sink has been attached inadvertently in the modification phase, it is disconnected and the necessary configuration held back for the residual bitstream. Another necessary check is the timing of the circuit. After the overlay of as many wires as possible, some wires will hang off existing netlists not affecting functionality but will affect timing. The simple adjustment made in this work is to compare the size of each netlist in the modified circuit to the size of the critical path in the original circuit and trim extraneous wires as necessary. Figure 3.3(i) shows an example netlist before it is trimmed and figure 3.3(ii) shows it after it has been trimmed. Again, wires that must be left unconnected during the advance configuration are moved to the residual configuration.

The residual configuration bitstream contains the new CLB configuration data and the remaining routing configuration. A similar verification exercise is performed to ensure the functional and timing accuracy of the resulting circuit - adjusting as appropriate.

3.4.3 Results

The hypothesis proposed for the overlay technique is that the high level of redundancy present in the FPGAs interconnect fabric will translate into a high level of redundancy

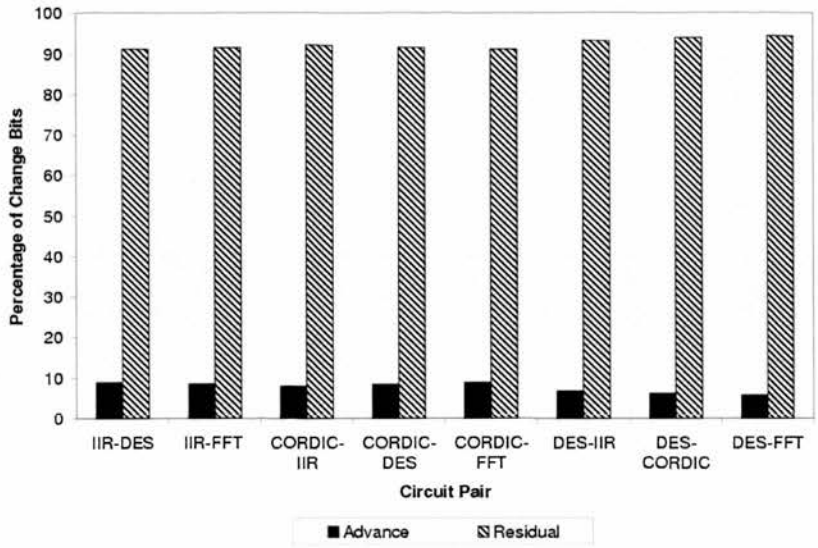

Figure 3.4: The advance and residual changes expressed as a percentage of the total number of changes required to configure the area of fabric occupied

in the configuration bitstream. To test this hypothesis, a straightforward count of the number of frames in the residual and advance bitstreams will not suffice, as explained above. Instead, to test the performance of the overlay algorithm, the number of individual bits that can be set in the advance bitstream is expressed as a percentage of the total number of bits that need to be flipped - i.e. the sum of the advance and residual changes. This was done using the library of routines described above - analysing the changes made during the overlay algorithm through the use of wrapper functions. A number of circuit pairs were overlaid on each other and the result found. Figure 3.4 shows a graph of the circuit pairs tested and the percentage number of bits which could be overlaid.

Consistently around 10% of the configuration data could be written in the advance bitstream without interfering with the operation of the existing circuit. Although it was disappointing that this percentage wasn't larger, the experimentation revealed that the original hypothesis does hold promise, just not so much for the overlay on the Virtex architecture. The reason for the small percentage of bits which could be overlaid is

that many of the routing resources are multiplexer type structures. Multiplexers cannot have bits overlaid without changing the function they perform, but programmable interconnect points (PIPs) can and the 10% overlay figure comes from the PIPs. In fact PIPs were the only resource considered for the overlay technique in the final algorithm in order to reduce the complexity of the implementation. LUT contents and the many input and output multiplexers are not suitable, which leaves only a few small internal CLB multiplexers unconsidered. It is thought that there is some potential for improving the 10% result by exploiting these other resources but the effect is likely to be small. The circuits used for experimentation are very densely placed, so the technique is likely to be more beneficial for less densely packed circuits.

It is interesting to note that the application of changes is easily reversible. This is useful, for example, when only two circuits share the same area of fabric since only one list of changes needs to be loaded.

3.4.4 Summary

The identification of the residual bits also serves as a possible way to generate the inverse configuration change - i.e. invert the changes and return to the original circuit. This is useful in applications where an area of fabric is being time-multiplexed by two circuits. The next section will explore the extent of the redundancy present in the configuration bitstream to find other ways of exploiting it.

3.5 Bit-Stream Redundancy Analysis

3.5.1 Overview

This section investigates the changes made to completely switch from one circuit to another - i.e. not using the overlay technique.

3.5.2 Experiments

The first obvious question which is important to answer is the level of redundancy present in any configuration. This is answered by performing the minimal one-shot re-

configuration and expressing the number of bits that must change as a percentage of the number of bits for the circuit. Again the arrangement of bits in the Virtex architecture means the measure of configuration bits required to load a circuit has to be considered carefully. The straightforward choice is the size of the partial bitstream necessary to configure the fabric, but due to resource bits being scattered across a whole column of frames, many more configuration bits have to be loaded than is necessary. Instead, the actual number of bits necessary to configure the area of fabric occupied is used for the total number of bits. It is calculated as the number of configuration bits for a CLB (864 bits) multiplied by the number of CLBs occupied by the circuit.

The second interesting question considered is the spread of change bits across the different resources. A good understanding of this may help identify an organisation of bits into frames to improve reconfiguration performance. Resources are split into singles wires, F and G input multiplexers, slice internal configuration, look up table contents, output multiplexers and hex and long wire configurations.

Both investigations used the software tools developed on top of JBits for the overlay technique.

3.5.3 Results

3.5.3.1 Change Bits

The number of bits within an area of fabric which must be changed is consistently below 10% as shown in Figure 3.5. This reveals that the original hypothesis of this chapter is true - massive redundancy is present in the configuration bitstream. Observing that only a small number of bits must be changed suggests that compression may be used to express the changes efficiently and by so doing reduce the reconfiguration time. This idea of compressing the configuration changes is developed later in this chapter.

3.5.3.2 Breakdown by resource type

Figure 3.6 shows that the lookup table contents represent the largest proportion of bits which must be changed during reconfiguration. The sum of the slice input multiplexer

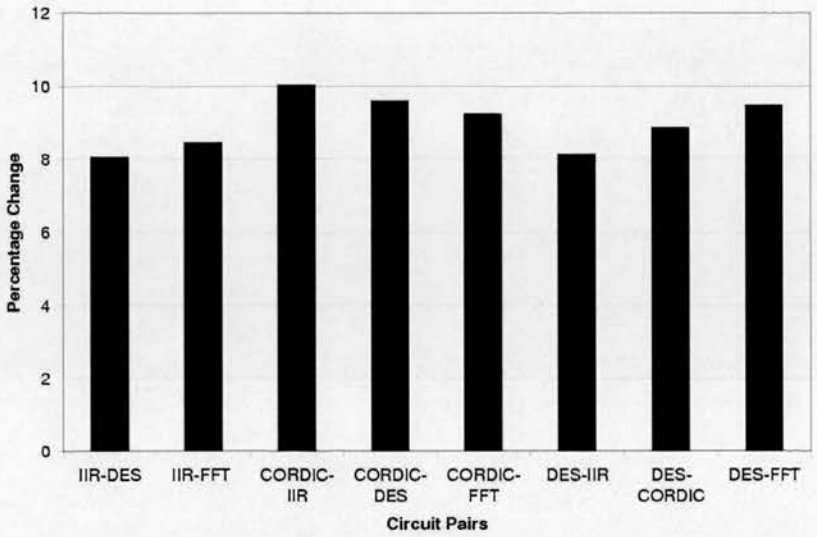

Figure 3.5: The number of changed bits as a percentage of the actual number of bits required to configure the area of fabric occupied

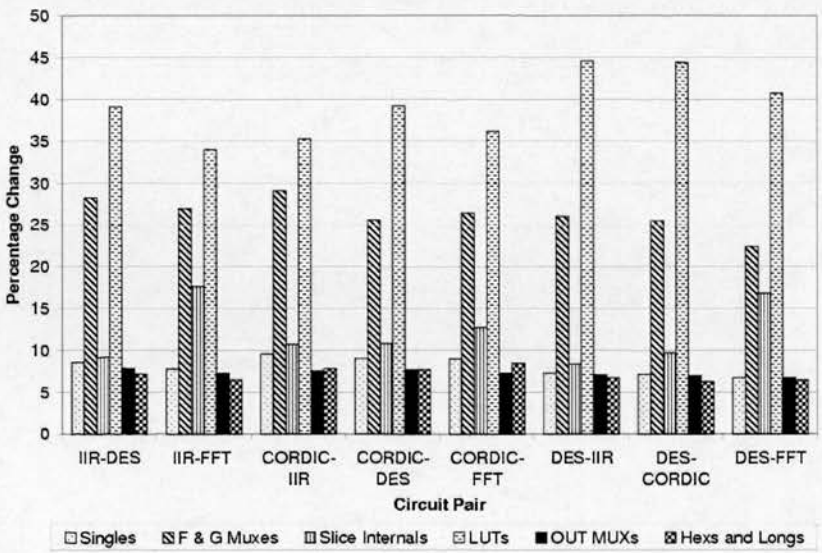

Figure 3.6: The number of changed bits broken down by resource type and expressed as a percentage of the total number of change bits required to configure the area of fabric occupied

changes and the LUT contents represent over 60% of all changes that must be made. When each of the resource type's percentage change is compared to the total number of configuration bits configuring that resource type, the LUT represents a disproportionately large percentage of the change bits. For example, there are 16 bits configuring each LUT in a CLB which is 4% of the total bits configuring a CLB, however the LUT represents between 35% and 45% of all change bits. This suggests that the configuration bits could be arranged differently to take advantage of the disproportionate percentage change.

3.5.4 Summary

The redundancy present in the bitstream has been demonstrated to be significant and strengthens the argument that there must be effective methods to exploit it. The disproportionate representation of change bits across the resource types is also an interesting result which may lead to another technique for configuration compression. The remainder of this chapter investigates configuration change compression as a means of reducing configuration time.

3.6 Changes Compression

3.6.1 Overview

Compression technologies are used extensively across a wide range of application domains to reduce the amount of data required to express information. The high level of redundancy confirmed in the analysis work is an obvious candidate for some form of compression to reduce reconfiguration time. The basic challenge of configuration changes compression is to concisely express a long stream of binary zeroes interrupted occasionally by one or more binary ones (90% binary 0, 10% binary 1). A binary one indicates that the corresponding bit in the configuration memory must be inverted. This section looks at existing non-lossy compression algorithms and investigates their suitability to bitstream changes compression. It then proposes some Virtex specific algorithms and explores a configuration architecture space suitable for leveraging the

compressed changes.

3.6.2 Targeted Compression Algorithms

Generic loss-less compression algorithms do not suit the domain of bitstream decompression on an FPGA. Parallel implementation is necessary to provide a configuration architecture tailorable for the speed of reconfiguration required. Lempel-Ziv 77 performs well on long sequences, but as the sequence length increases, the compression performance degrades. In addition, LZ77 performs well on sequences with repeating sequences like text and other types of data with a small alphabet, but less so on data such as images and the configuration bitstream. If the configuration bitstream were to be decompressed on-chip in parallel, it would have to be split up and each piece processed separately. This requires an independent decompression unit for each piece and for Lempel-Ziv this could quickly become quite a sizable amount of silicon area.

3.6.2.1 Vanilla

The Vanilla algorithm is the most simple of those proposed and tested, hence its name. Its development evolved through a number of guises so this description will describe the various forms it took. First of all the changes bitstream - i.e. the sequence of mainly zeroes with ones in the positions where a bit in the configuration memory is to be flipped, is read in and absolute addressing used. This meant that a large data word had to be used to address any configuration bit in a large device - 16 bits for the largest Virtex part. This performed very badly - in fact using 16 bits to give the position of the 10% of bits produces an expansion rather than a compression! The straightforward alternative to expressing the change bits using absolute addressing is relative addressing. Choosing the size of the data and address parts is key to minimising the expression when using relative addressing. A program was written using JBits to cycle through a range of address/data size pairs to find the optimum arrangement. This was tuned to permit writing zero data points if it meant that the smaller address size used reduced the overall changes expression.

3.6.2.2 Banded

While implementing the Vanilla algorithm, it was observed that configuration bits tend to form clusters in the frame if the circuitry being changed only takes up part of a column. This is because every frame contains configuration bits of every CLB in the column arranged with the bits of the top CLB at the top of the frame down to the column's bottom CLB bits at the bottom of the frame. This was used in the banded compression algorithm to express the configuration changes relative to where the first CLB that is being reconfigured starts in the frame. The implementation uses an offset number of bits to indicate the start point for relative addressing.

3.6.2.3 Partitioned

It was observed above, during the changes analysis work, that the LUT contents, although a very small fraction of the configuration bitstream, represent a very large fraction of the changes bitstream. With this in mind it seemed worth experimenting with expressing the LUT contents explicitly rather than compressing them. The partitioned algorithm expresses the LUT contents for each CLB in the area being reconfigured explicitly and expresses the remaining change bits using the banded technique.

3.6.3 Results

All three compression algorithms were implemented in software using JBits to evaluate their performance. The partitioned algorithm produced the best averaged compression performance of 45% as shown in Figure 3.7. However since the difference between the banded and partitioned technique results is small (5%) and the partitioned algorithm is likely to require much more logic to implement the decompression unit, the banded algorithm is chosen.

The Lempel-Ziv family of algorithms was experimented with to check their performance on the changes dataset and they were found to produce compression ratios around 50%. This is close to the information theoretic limit for the compression of a random binary sequence with 10% of bits equal to binary one. In [102] it was noted that the choice of symbol size for the Lempel-Ziv family of compression algorithms

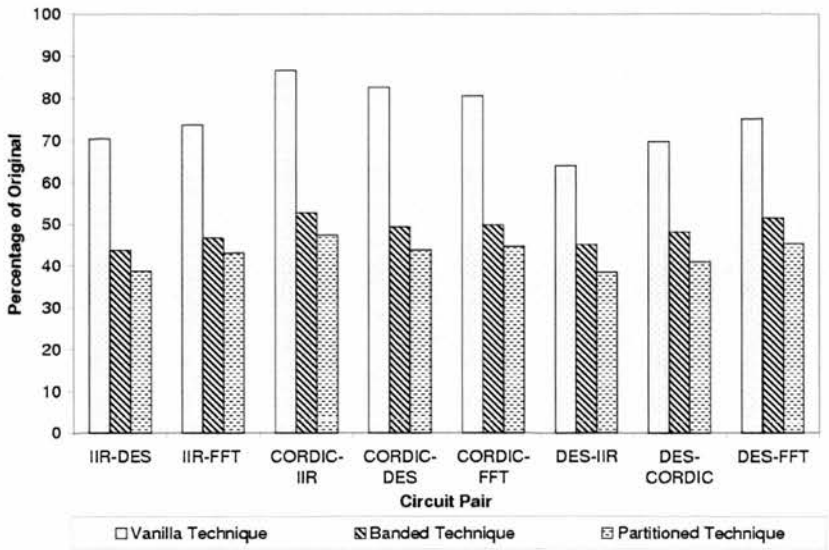

Figure 3.7: The compressed changes dataset size expressed as a percentage of the number of bits required to configure the area of fabric occupied

has a large influence on compression performance, since a poor choice may miss regularities. The LZ family was not experimented with any further because as noted above it was determined that its decompression unit's silicon area cost was large and a parallel decompression architecture would lose many of the regularities exploited in the full serial bitstream[102].

3.6.4 Summary

A reasonably consistent compression algorithm for configuration changes description has been proposed and its performance evaluated. It is highly scalable, can be implemented cheaply in silicon and compares favourably with more complex standard algorithms such as LZ77 used in the Unix gzip utility.

Generation	Device	Slices	Block RAM	Process	Configuration bits
Virtex	XCV1000	12,288	131,072	0.22um	6,127,744
Virtex E	XCV3200E	32,448	851,968	0.18um	16,283,712
Virtex II	XC2V8000	46,592	3,095,576	0.15um	26,194,208
Virtex II Pro	XC2VP100	44,096	8,183,808	0.13um	34,292,832

Table 3.1: Selected data for the largest device in each Virtex series generation

3.7 Virtex II and future platform FPGAs

3.7.1 Introduction

In this chapter so far, we have studied the first generation of Virtex FPGAs. This section examines how the most recent generations in the Virtex series have evolved, in an attempt to establish architectural trends. We will then hypothesise how these trends will affect the feasibility of dynamic reconfiguration.

3.7.2 Architectural Features

Table 3.1 lists the largest device in 5 Virtex FPGA generations. Each generation is fabricated at a different process geometry, providing greater transistor integration per chip. The functionality of a logic slice has remained roughly constant across the generations, which enables inter-generation logic capacity comparisons. In this work we assume that the die area of the largest device is either equal to or larger than the previous generation.

An obvious high-level trend is that the number of different architectural features is changing with each generation (see below). Further, the mixture of architectural features is also changing. This is in stark contrast with the naive belief that the regular, array based architecture of an FPGA would simply scale with Moore's law. There are many reasons why the naive belief is false, spanning from fundamental empirical results such as the relationship between the number of gates in a block and the number of I/Os required (Rent's rule [91]) to advances in the understanding of the

programmable interconnect tradeoff space. Notable work on FPGA interconnect by Betz and Rose[18] shows how a segmented and buffered routing architecture is much superior to nearest neighbour interconnect.

Early FPGA architectures such as the Xilinx 4000 series were simply an array of fine-grain logic blocks connected together by a programmable routing fabric. The Virtex introduced embedded block RAM and logic block functionality to help with arithmetic. The most recent architecture, the Virtex II Pro, includes embedded multiplier units, power PC processors and multi giga-bit input-output transceivers. The amount of embedded memory in the form of block RAM has grown close to 17 times more quickly than the number of logic slices when the largest Virtex and Virtex II Pro devices are compared. On-chip RAM is essential for unpredictable memory access patterns as off-chip latency continues to grow in relation to logic speed. The trend towards more on-chip RAM is also due to off-chip bandwidth not growing at the same rate as transistor integration. Therefore, to satisfy the bandwidth requirements of some applications, RAM must be provided on-chip.

The growth in the die area ratio between "hard" function blocks and the "soft" logic-slice fabric is a significant trend. It is leading to fewer logic-slice configuration bits per transistor, potentially easing the burden on the configuration architecture. Table 3.2 shows what percentage of the configuration bit stream is represented by the logic-slice fabric in the largest member of each Virtex generation. The numbers demonstrate the trend described here, although it should be noted that due to overheads such as frame alignment and padding, the percentage figures may be slightly inaccurate. It should also be noted that an absolute figure for the number of bits per Virtex II CLB is not made available by Xilinx, and instead an estimate was used here (1830 bits/CLB) based upon some experiments using JBits. An accurate value of 864 bits/CLB is known for the Virtex and Virtex E.

As the FPGA is used in more production systems, the established architectural trend is to incorporate more fixed functionality at the expense of flexibility. This reduces both the silicon area and energy gap with the ASIC alternative. The integration of function blocks specialised for particular domains looks set to continue, producing domain specific platform FPGAs[178]. It is clear that the percentage overhead of a

Generation	Device	% Configuration bits for logic-slice fabric
Virtex	XCV1000	86.6
Virtex E	XCV3200E	86.1
Virtex II	XC2V8000	81.4
Virtex II Pro	XC2VP100	58.8

Table 3.2: Percentage of configuration bitstream for logic-slice fabric

given logic-slice reconfiguration architecture is constant. It is not clear if the resource savings provided by reconfiguration of the logic-slice fabric will remain constant as it is replaced by fixed functionality. Part of the reason for this uncertainty is the lack of consensus in the literature over what role the logic-slice fabric is best suited to in a platform SoC. If, as some believe[5], its role will converge on the tightly coupled control (e.g. high speed FSM) of a coarse grain datapath, then the runtime reconfigurability of the logic-slice fabric may have a disproportionately large effect on the overall chip resource usage efficiency.

3.7.3 Off-chip Memory Bandwidth

At present, the speed at which reconfiguration of the FPGA's logic-slice fabric can occur is limited by the configuration interface. As discussed in this chapter, the configuration interface and architecture can be changed to use the maximum off-chip bandwidth. The Virtex II device can implement a 64-bit 200MHz DDR RAM interface to provide 25.6 Gbps. If all user I/O pins are used to connect to DDR RAM, with only one set of control signals provided for all interfaces, an aggregate throughput for the largest Virtex II device of 410 Gbps is possible. It would take $52\mu s$ to load all the XC2V8000 logic-slice configuration bits. Of course, the cost to provide this bandwidth is very high - 16 DDR memory modules are used to produce the throughput, and power consumption during reconfiguration will be considerable, if not unsupportable. Compression techniques as discussed in this chapter would reduce the time required to $25\ \mu s$.

Generation	Device	User I/O Pins	Slices per pin
Virtex	XCV1000	512	24
Virtex E	XCV3200E	804	40
Virtex II	XC2V8000	1,108	42
Virtex II Pro	XC2VP100	1,164	39

Table 3.3: User I/O pins for the largest device in each Virtex generation

In the context of an application, $25\mu s$ may be considerable if latency is critical. Network routers must minimise packet handling latency, as they are only part of the cumulative end-to-end latency experienced by an application. For example, the average number of hops made by Internet traffic today is 15[134]. When one considers other delays such as buffering in host systems and application processing delays, it becomes clear that latency introduced at routers must be minimised. An 8Gbps IPV6 forwarding implementation by Intel achieves a maximum latency of 60us for an Internet traffic distribution of packet sizes[114]. To compete with this implementation, the computational delay of a runtime reconfigurable solution must be less than or equal to the difference ($35\mu s$).

The network router serves as an example to show that systems with tight latency requirements place restrictions on the application of runtime reconfiguration using off-chip RAM. Table 3.3 shows that the number of I/O pins in each Virtex generation's largest device is not growing at the same rate as the number of slices available. The Virtex II Pro shows a drop in the slices/pin count, probably due to the displacement of logic-slices by hard cores. So, despite the pedestrian improvement in the number of user I/O pins and RAM interface standards compared to on-chip transistor integration, maximum available off-chip memory bandwidth may manage to keep up with the growth in logic-slices.

3.7.4 Summary

The number of logic-slices is growing at a reduced rate compared to on-chip transistor integration. The result is that as a percentage of overall cost, the silicon area required to facilitate fast run-time reconfiguration using on-chip RAM is reducing. There exists an open question as to whether run-time reconfiguration's requirements change as the ratio of logic-slices to transistor integration reduces. We illustrate by example the critical importance of reconfiguration latency to communication systems.

3.8 Summary

In this chapter we have pointed to the large design space for reconfiguration architectures. In Section 3.2 we introduced a major commercial FPGA architecture - the Xilinx Virtex. We then performed an analysis of what exactly occurs during reconfiguration of the Virtex in Section 3.3. The overlay technique is the first result of the analysis, as presented in Section 3.4. Further experimentation in Section 3.5 reveals the extent and characteristic of the redundancy in the Virtex configuration bitstream. This is taken advantage of in Section 3.6 to develop a highly scalable and low-cost compression algorithm. Finally, Section 3.7 considers the demand that will be placed on configuration architectures by new platform FPGAs.

Chapter 4

A Design Methodology for the Reconfigurable Platform FPGA

4.1 Introduction

For reconfiguration to be widely accepted by embedded system designers, it must provide significant resource savings in return for little additional design effort. This demands integration with established design flows, the use of familiar tools and languages and the ability to make use of existing intellectual property. In short, reconfiguration must be an evolution, not a revolution.

Section 4.2 characterises the application domain the methodology and its associated architecture should target. Section 4.3 gives an overview of our new design methodology, describing the main principles upon which the approach is built. We split the detailed description of the methodology over three different sections. Section 4.4 describes a simple runtime framework, referred to as the Checkpoint Framework, onto which the implementation is mapped. Section 4.5 describes the procedure for efficiently targeting the system at the framework. Finally, the algorithms and techniques used in the targeting procedure for achieving the improved resource usage are given in section 4.5.4. Section 4.6 lists the main contributions the methodology makes and compares it to previous work. The chapter ends with a summary in section 4.7.

4.2 Application Domain

4.2.1 Introduction

Over the past decade, the dominant form of computing has shifted from general purpose to embedded realtime[131]. Correspondingly, reconfigurable research activity has shifted from general purpose, highly flexible architectures such as PRISC[133] and Garp[79] to less flexible, domain specific architectures such as RaPiD[61], Pleiades[5] and PipeRench[138].

Whilst the factors driving research and development have changed, the fundamental promise offered by dynamic reconfiguration remains unchanged. Where an architecture offers flexibility that may be exploited at run-time, there exists the potential for resource savings, as shown in a theoretic result by Brebner in [32].

FPGA fabrics have also evolved, from the regular array of LUTs and RAM such as the Xilinx Virtex[177] to the Xilinx Virtex-4[179] with its heterogeneous organisation of LUTs, RAM, embedded multipliers and DSP blocks.

This section describes the application domain targeted by the methodology (section 4.2.2) and critiques existing implementation candidates (section 4.2.3).

4.2.2 Static Application Domain

4.2.2.1 Introduction

To help form a methodology for the reconfigurable platform FPGA, it is worth characterising the static FPGA's application space and features. We consider the reconfigurable FPGA to simply be an enhanced version of the static FPGA. All functionality remains the same, including the LUT fabric, ratio of RAM to logic and the DSP core functionality; only the speed of reconfiguration changes. Restricting architectural change to this single variable lends confidence to the assertion that the domain of application will stay the same.

4.2.2.2 Application Characteristics

When we compare the platform FPGA to other reconfigurable architectures we note

the following:

1. High level of concurrency.
2. Saturated flexibility.
3. Real-time system focused.
4. Heterogeneous architecture.

These characteristics carve out the application space occupied by platform FPGAs. The generous provision of LUT-fabric and programmable DSP blocks offers greater application concurrency than a non-specialised CPU of equivalent silicon area. This is in part due to the large fraction of silicon area in a CPU devoted to extracting the limits of parallelism within a serial instruction stream. DSPs with specialised co-processors are able to compete with the platform FPGA's concurrency in certain niche applications. For example, Texas Instrument's C6416T DSP[158] includes Viterbi and Turbo co-processors for Forward Error Correction in communication systems. However, servicing processor interrupts can pose problems in a real-time system and esoteric architectures demand specialist knowledge to program effectively. For example, network processors are architecturally diverse and are often programmed by hand using assembly language[142][112].

The saturated flexibility characteristic is unique to an FPGA fabric. The 4-bit LUT offers a very general-purpose computational element and the routing fabric offers high redundancy for unrivalled control. The degree of flexibility is illustrated by the fact that less than 10% of configuration bits are important to a typical circuit implemented on the fabric. Other, less general purpose reconfigurable architectures, incrementally add flexibility to meet a well specified requirement. For example, the Totem Project[46] produces a coarse grain architecture with just enough flexibility to implement any one of a set of netlists. The undesirable increase in area and the timing effects of extraneous routing is minimised. Similarly, in[83] the authors automatically generate a reconfigurable architecture, but with PLA/PAL arrays which are much better at implementing random logic.

Characteristic	Augmented Processor	Virtual H/W (1)	Virtual H/W (2)	Mult. Context FPGA
High Concurrency	✗	✓	✓	✓
Saturated Flexibility	✓	✓	✗	✓
Real-Time Focused	✗	✗	✓	✗
Heterogeneous Architecture	✗	✗	✓	✗
Example	Proteus[48]	SLU[25]	Pleiades[6]	DPGA[52]

Table 4.1: Suitability of selected reconfigurable FPGA models to the characteristics of the static FPGA's application domains.

The real-time system focus of platform FPGAs is obvious when one considers the primary application domains: telecommunications and DSP. Algorithm implementation in circuitry explicitly deals with worst-case possibilities - well suited to real-time systems. By claiming the spatial resources required for worst-case operation at the global clock frequency, the designer guarantees meeting real-time requirements.

4.2.3 Model of computation

4.2.3.1 Introduction

The design process for static, synchronous RTL is well established and understood. Reconfiguration extends the temporal dimension in the space-time tradeoff space from design time into run-time. The challenge for the research community is how to exploit the additional design freedom provided by reconfiguration. Selected previous approaches to exploiting reconfiguration in an FPGA are characterised in table 4.1. None fully match the characteristics of the application domain satisfied by the platform FPGA. In this section we explain and discuss this assertion.

4.2.3.2 Processor

Several of the computation models listed in section 4.2.3.1 are microprocessor-centric, identifying the most computationally intensive software routines and transferring them to an FPGA fabric. These techniques assume the microprocessor should maintain its central role in computation, an assumption which is criticised by other SoC researchers[28]. A continuation of the stored program model of computation is likened to shrinking the motherboard and placing it on a chip. In such a model, the microprocessor coordinates activity, with programmable logic implementing function units attached to the system bus, or co-processors on a peripheral bus. The combination of a central controller (microprocessor) and communication via buses leaves concurrency at the mercy of two bottle necks: the serial instruction stream and shared buses. Further, the microprocessor's central arbiter role can make real-time systems difficult to implement, such as when processor cycles are consumed by unforeseen sequences of interrupt requests.

Commercial examples of processors augmented with an FPGA fabric do exist, such as the Triscend A7[161], however they tend to be used as micro-controllers, with the FPGA fabric implementing peripheral functions and interfaces. Conversely, in platform FPGA architectures, the microprocessor is included as a complement to the dominant programmable logic fabric. In this role of diminished responsibility, a proposed function for the microprocessor is as a handler of exceptional control flows[27].

4.2.3.3 Virtual Hardware

The SLU model of computing, as described in sections 2.6.2 and 2.7.1.3, promises a solution to the problems of the processor centric model. Constructing a system from SLUs enables the concurrency of freely formed logic to be fully harnessed. Functionality is no longer partitioned or constrained by the pre-divided regions of fabric in the function unit model. Communication is no longer throttled by the use of shared buses; instead the abundant FPGA routing resources provide dedicated, high-bandwidth communication links between SLUs.

The unconstrained shape and size of the SLU footprint is both a strength and a weakness. Loading and unloading SLUs raises the complex tasks of fragmentation management and interface satisfaction. Heuristic methods to tackle these NP-complete

problems involve constraining the degrees of freedom, but they do not provide a satisfactory solution for real-time applications. Real-time solutions are deterministic and involve a highly constrained version of the problem. These solutions come with little supporting evidence of application, in effect producing an architecture with unknown application. The same criticism does not apply to the pure, unconstrained problem, because it can claim the existing wealth of static systems (which are valid in the virtual hardware model), as evidence that dynamic reconfiguration can only provide increased application potential.

The heterogeneity of platform FPGAs poses further problems for the virtual hardware model. Memory blocks and DSP functionality embedded in the programmable logic fabric destroy the luxury of homogeneity enjoyed by most virtual hardware work in the literature. Homogeneity allows an SLU to be placed anywhere on the grid of LUTs providing its interface is satisfied. On a heterogeneous fabric, the interface with fixed embedded functionality must also be satisfied; limiting the freedom of placement, therefore reducing the flexibility of the model. This also affects placement in the static model of an FPGA, but the fluidity of placement at the granularity of a LUT, rather than the SLU, makes it less acute.

A common and simple virtual hardware approach to reconfiguration, is to temporally partition a design into several pages which are loaded into the FPGA in succession. For example, in [165], a single-FPGA video coder, which is reconfigured dynamically between Motion Estimation, DCT and Quantisation is described. The suitability of such systems to implementation in an FPGA can be challenged in terms of the characteristics in table 4.1. If it is possible to serialise the algorithm and still satisfy requirements, then perhaps the concurrency offered by an FPGA is not necessary, and instead a high-end DSP would be suitable. The authors do not offer consideration of this point or a technology comparison.

4.2.3.4 Synthesis

The logic-centric approach to exploiting dynamic reconfiguration extends high-level synthesis through temporal partitioning. As described in section 2.5.3, models exist for performing the combined problem of resource allocation, scheduling and temporal

partitioning. However, we assert that the solutions based upon these models are not feasible in an FPGA. We present three reasons to back-up this assertion:

1. The first reason is cost. The models employ control data flow graphs (CDFG), working at the level of individual operations such as add and multiply. The low-level of abstraction requires reconfiguration at speeds on the order of operator latency to be competitive with the static alternative. The word-oriented interconnect and compute elements of a coarse grain fabric may be enhanced to meet this reconfiguration requirement for a reasonable increase in silicon area. By comparison, to extend the already saturated flexibility of an FPGA to perform reconfiguration at the speed of an operator requires much more silicon area. This stems from the fact that in a coarse grain fabric, the configuration bit area can be amortised across several wires when the wires are data buses [43]. In an FPGA, every wire is individually configurable. Also, implementations by DeHon and Trimberger (see Appendix D) show the need for registers to re-time signals. In the general-purpose architecture of an FPGA, all the re-timing registers are of equal size, capable of catering for some "worst-case" requirement, maximising their implementation cost.

2. The second reason is off-chip bandwidth. As described in Appendix D, the provision of multiple configuration RAM bits per resource makes it possible to both switch rapidly between configurations and to load new configurations while performing computation. The parallel load is subject to the same off-chip bandwidth bottle-neck as the single-context FPGA (chapter 3). With a limited number of contexts, it would be necessary to load a new configuration from off-chip RAM in the order of the time taken to perform an operation. This is not feasible.

3. The third reason is power consumption. To switch all configuration bits on the order of the time taken to perform a single operation would create thermal problems[160].

Table 4.2: Three strand approach to exploiting reconfiguration

Description	Frequency	Configuration Access
1. Static	STATIC	COMPLETE
2. Inter-subsystem	DYNAMIC	COMPLETE
3. Intra-subsystem	DYNAMIC	RESTRICTED

4.2.4 Summary

The platform FPGA meets the demands of applications which are predominantly real-time, have high-levels of concurrency for which there is no specialised architecture available and are produced in low-medium volume. When considering enhancements to the configuration architecture, it is important to note that they will effect its suitability for the established areas of application. Any significant changes to the power-flexibility-cost tradeoff must produce either reasonable benefits across a wide range of existing applications or large benefits to a small set of applications. Otherwise, there exists a danger of creating a new architecture with no domain of application. In essence, as mentioned in the introduction to this chapter, reconfiguration has the potential to enhance the implementation of existing applications, not to define new applications.

Our analysis of existing reconfigurable models concludes that none are suitable for the existing application domain satisfied by the static FPGA.

4.3 Methodology Overview

Our methodology adopts a robust, pragmatic and conservative approach to harnessing reconfiguration. It recognises the dominance of the established approaches to static synchronous RTL design, and unlike previous work in the literature[72][92], it aims to enhance, instead of replace, these proven design practices. The application area we use to illustrate our approach is the real-time latency-sensitive System on Chip (RLSoCs).

Our methodology harnesses reconfiguration in a three strand approach, listed in table 4.2. The strands differ in terms of their access to the configuration memory

(complete or restricted), and when the access is performed (static or dynamic). Both static and inter-subsystem strands have the freedom to change any part of the configuration. Intra-subsystem is restricted to those configuration changes that can be performed in-situ, that is, it uses local hooks to the configuration memory as opposed to the general-purpose configuration interface. The general-purpose configuration interface is a shared resource, so its use must be arbitrated. The local configuration memory hook is not shared, so its use is autonomous. For example, in the Xilinx Virtex architecture, the Select-MAP interface provides complete, general-purpose access to the configuration memory, and the LUT 'D' pin on the SRL16 set of primitives is a restricted, local configuration memory hook.

The three strand approach provides the conceptually simple and fabric independent foundation upon which the methodology is built. The high-level problem tackled by the methodology is how to cost-effectively harness reconfiguration in an RLSoC, with cost measured as the bill of materials.

Low-medium frequency reconfiguration (inter-subsystem) is an under-explored area in the the literature, hence its exploitation forms the main thrust of the methodology's presentation. An RLSoC consists of subsystems cooperating to perform a shared task. As the system's task changes with time, so does each subsystem's task; therefore it follows that a subsystem's resource requirement also changes with time. *Our key insight is that these temporal resource requirement fluctuations are not necessarily correlated across subsystems, producing the potential for inter-subsystem resource sharing.*

To aid understanding of this insight, it is helpful to consider its effect pictorially. Figure 4.1 depicts two possible implementations of the same hypothetical system. On the left hand-side is an ASIC implementation, and on the right hand-side is a reconfigurable implementation. The five shaded boxes represent different subsystems, and their area is proportional to their resource requirements. The figure clearly illustrates how the system-level insight may be exploited. The ASIC flow produces a single, combined worst-case instance. The reconfigurable flow produces multiple, isolated, worst-case instances. The sub-systems vary in size between system instances because they are tailored to the particular subset of functionality required. The functionality varies due to temporal changes in the system environment. Only one system instance

may be resident on the fabric. The loading of instances is presided over by a permanently resident control unit which is described in section 4.4. A number of points can be made upon consideration of the reconfigurable solution. First, the largest instance requires less resources than the ASIC instance. Second, subsystems are not always required simultaneously. Third, the largest instance of each subsystem must still be implemented by the reconfigurable solution. Fourth, each system instance satisfies an entire subset of system-level requirements. The lines between the two design flows link the five ASIC subsystems to the reconfigurable instance in which they are at their maximum size.

These observations apply to the application of inter-subsystem reconfiguration in table 4.2. It should be noted that static and intra-subsystem reconfiguration are complementary to the savings achievable with inter-subsystem reconfiguration. Static exploits the fact a reconfigurable fabric may be offered in a range of sizes. This enables a more efficient (and cost-effective) mapping between individual product requirements and their implementation. The ASIC design flow misses this opportunity. The application of intra-subsystem reconfiguration will always require fewer resources than the monolithic implementation. The third observation of inter-subsystem reconfiguration above is improved upon by the application of intra-subsystem reconfiguration.

4.4 Checkpoint Framework

The Checkpoint Framework defines the reconfigurable behaviour of the device at run-time. Its purpose is to present a simple, synchronous and design-time fixed target for the system at run-time.

The runtime framework consists of inserting a periodic sequence of special time-points into system execution, called the checkpoint heartbeat. On the rising edge of the heartbeat, a permanently resident control unit calculates whether a reconfiguration should occur. If the checkpoint enable signal is set to high, computation in a sub-area of the fabric is suspended, configuration changes are carried out in the suspended area, and then computation is resumed. The checkpoint heartbeat is periodic, of constant duration and provides an opportunity for a statically determined area of

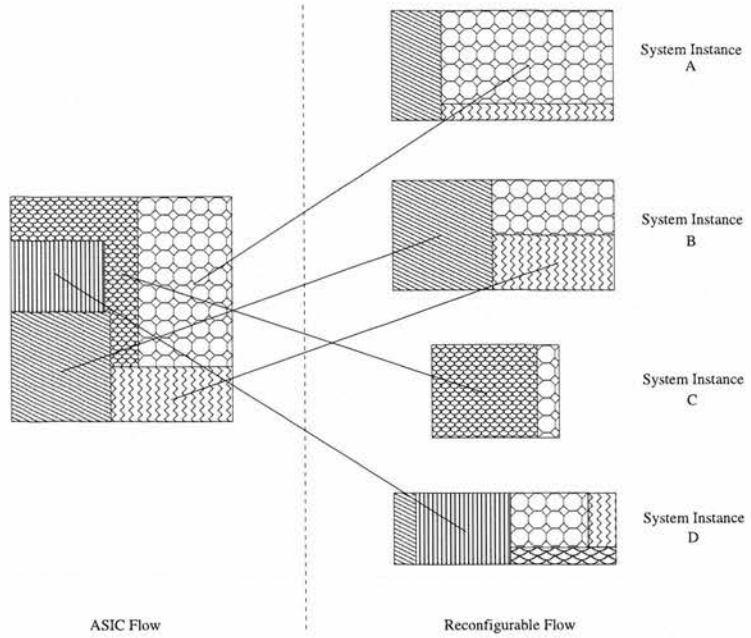

Figure 4.1: Traditional Implementation versus Reconfigurable Implementation Exploiting Inter-Subsystem Resource Sharing

fabric to be reconfigured. The ratio between the time spent reconfiguring and the time spent performing computation is important. Together with the reconfiguration speed of the target device, the ratio determines whether inter-subsystem reconfiguration is worth performing. Figure 4.2 illustrates the relationship between checkpoints, inter-subsystem reconfiguration and the time spent performing computation. On every rising edge of the checkpoint heartbeat, computation always halts in the statically determined reconfigurable area of fabric, providing an opportunity to make configuration changes. As can be seen in the picture, not every opportunity to perform reconfiguration is taken.

The configuration remains fixed between two checkpoints, meaning it must be capable of satisfying all possible requirements during that time. When a checkpoint occurs, the area of fabric being reconfigured must present a consistent interface to the rest of the system. That is, the interface must continue to consume and produce data. This effect is achieved by using RAM to buffer the input and output. Although not exclusively the case, buffering is commonly performed at the chip I/O interface to make

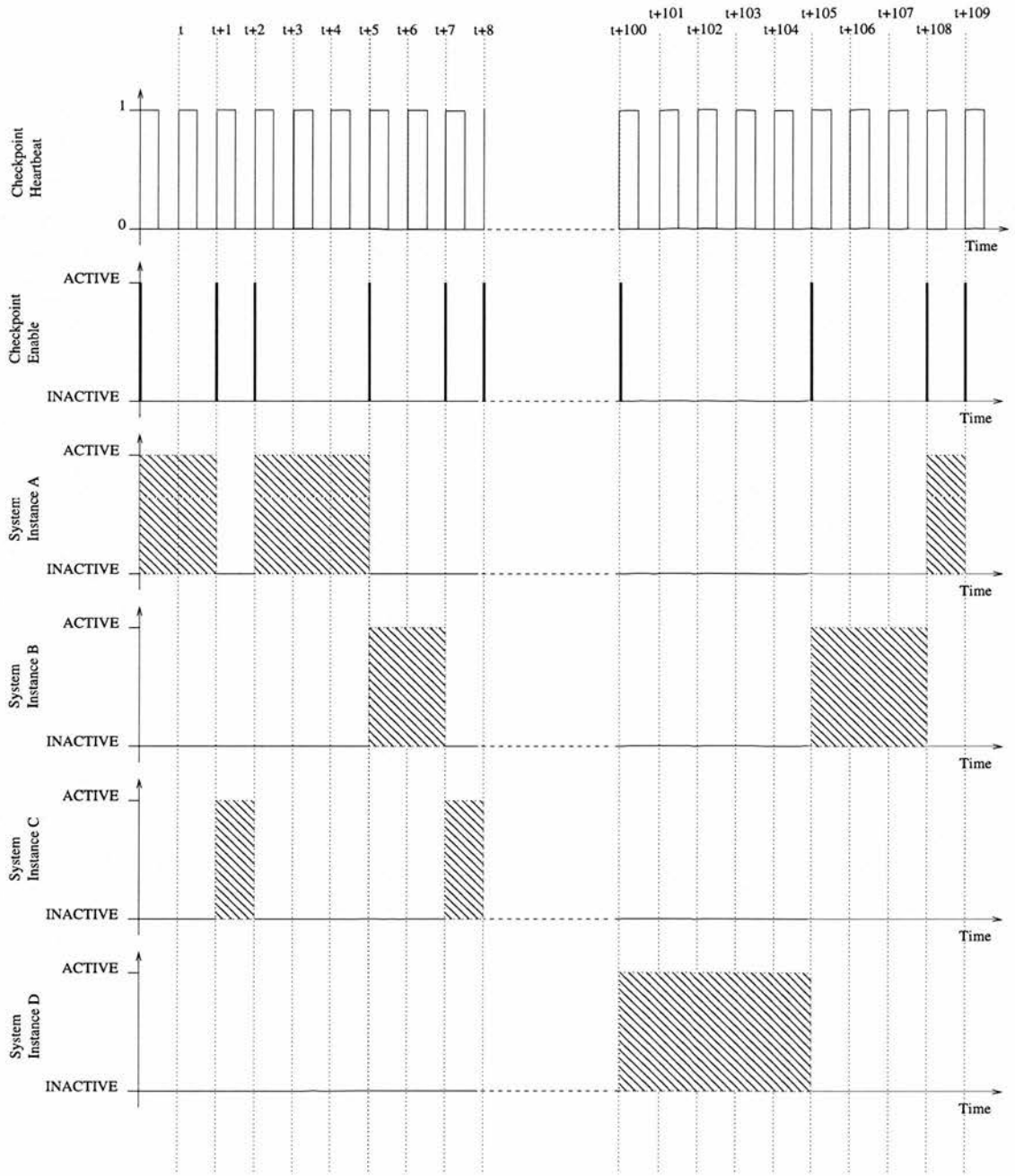

Figure 4.2: Illustration of timing relationships between checkpoints, computation and reconfiguration.

use of cheap off-chip RAM.

4.5 Targeting

4.5.1 Introduction

The process of selecting a suitable checkpoint frequency, optimally mapping a design to the run-time framework and configuration organisation are discussed in this section. These ideas are then combined in Section 4.5.5 to produce a step-by-step procedure for targeting the checkpoint framework.

4.5.2 Reconfiguration Frequency

The checkpoint methodology's place in the continuum of reconfiguration frequencies is much closer to static design (reconfiguration frequency of zero), than it is to changing state on each and every FSM transition, as proposed in [54]. The choice of reconfiguration frequency is based upon the observations of others, and estimates of silicon area required to facilitate different frequencies of reconfiguration.

The guiding principle of the checkpoint methodology is the maintenance of a hierarchical view of system construction. Traditional, worst case design splits the system into blocks in a hierarchical manner. With the interfaces between blocks well defined, each block can be implemented with little consideration of the other blocks. In performing the hierarchical division of the entire system into blocks, certain characteristics of the system as a whole are lost. For example, how the processing load across blocks varies with time. Blocks are implemented to cope with their worst case processing load, producing a solution constructed out of worst-case blocks, which will never be simultaneously required at full load.

By continuously evaluating the system requirements at runtime, a more suitable hierarchical breakdown is possible. Blocks can satisfy present case system requirements, not overall worst-case requirements.

4.5.3 Reconfiguration Organisation

The procedure for identifying the checkpoint configurations is based upon two steps. The steps act as filters, reducing the set of all system configurations to the minimal set of configurations required to implement the system. This section illustrates the principle contribution made by the checkpoint approach to reconfiguration. It places the key optimisation step of the targeting procedure in context.

Today's system development is progressing towards connecting together pre-designed and tested cores to build a complete system. Our targeting procedure takes advantage of this trend by recognising that selecting different core parameterisation values requires very little effort. A new system configuration is easily produced, since it simply involves applying a new set parameters across all the cores, and instructing the tools to produce another configuration bitstream. The additional design effort for a reconfigurable implementation is performed by the existing design tools, not the system designer.

The process of selecting core parameters corresponding to minimal FPGA resource usage is illustrated in Figure 4.3. The bubble at the top of the picture represents all valid instances of the system. This spans from the instance where no work is being performed to the instances where there is maximum resource usage. The first operation, is the productisation or static filter stage. This filter's effect is a combination of both business and engineering decisions. It uses knowledge of the products the system is going to form a part of, to tailor their implementation to more closely match their requirements. For example, consider the number of antenna, cell size and air interface protocols supported in a wireless base station. With the static filter applied, several versions of the same system, differing by their fixed parameterisation, proceed to the second stage independently.

The dynamic filter takes the set of possible instances for the statically separated system and produces the set of instances which minimally implement the system. As illustrated in Figure 4.3, the implementation instances specify the parameters for each core in the design. It is important to note that an implementation instance may not have an exact dynamic instance counterpart. This is because, to produce the system with minimal overall cost function, two or more dynamic instances may be combined

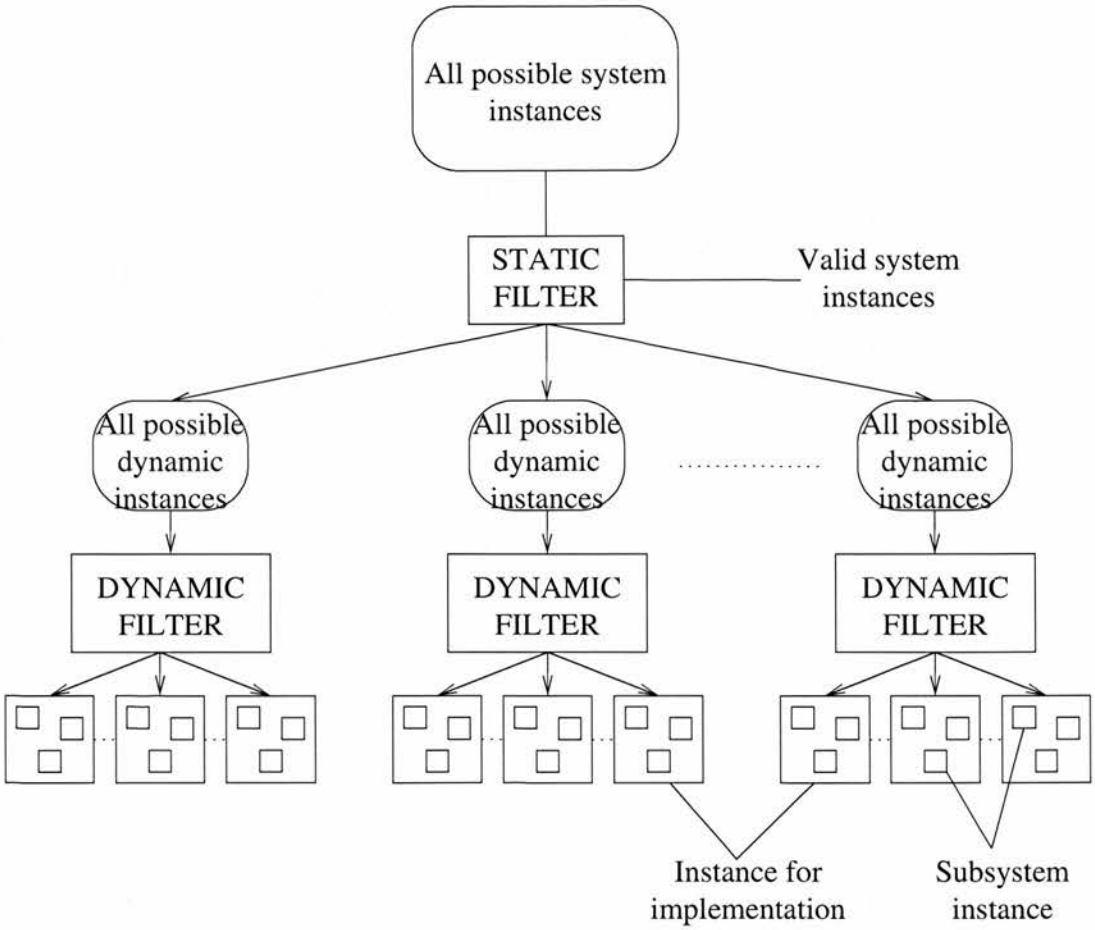

Figure 4.3: Configuration instance specification process

to form an implementation instance. As discussed in the next section, the cost function for the reconfigurable system is a combination of the FPGA size required and the number of configurations.

4.5.4 Optimisation

The key insight noted on page 70 results in the definition of a combinatorial optimisation problem, relating the coverage of specific system instances to hardware configurations and the cost of these configurations.

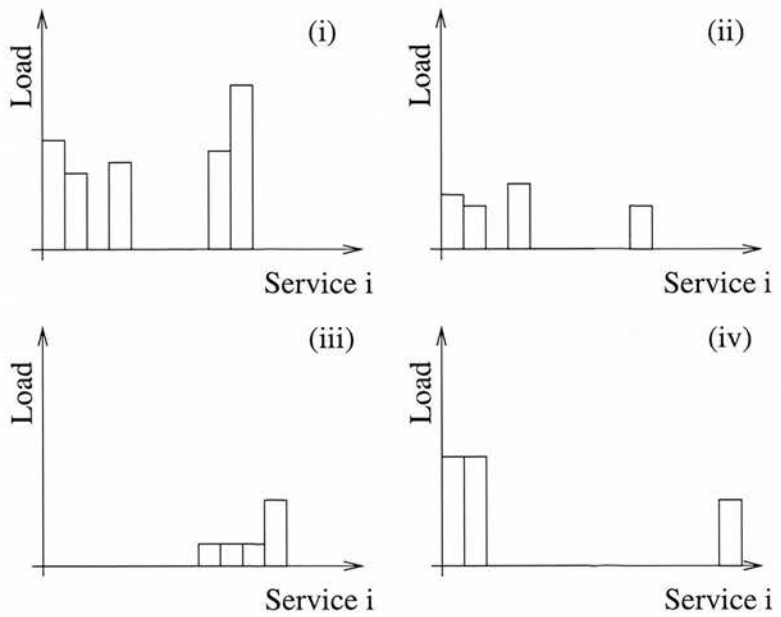

Figure 4.4: Four examples of load distribution across the different service types.

Figure 4.4 shows four examples of how processing load may be distributed across different service types offered by a system. We refer to these patterns as application profiles. It is important to note, for example, that a hardware configuration, h, which satisfies application profile Figure 4.4(i), can also satisfy the application profile in Figure 4.4(ii).

- We define an application profile, a, to be a vector in $\mathbb{Z}^I$, where I is the number of different services offered.
- We define an application profile set to be a set of application profiles.
- We define a hardware configuration to be a vector in $\mathbb{Z}^N$, where N is the number of hardware blocks in the system.
- We define a hardware configuration set to be a set of hardware configurat ions.
- We define a hardware configuration cost function $C :: (\mathbb{Z}^N) -> area$.

- We define an objective function, $O :: (int, area) -> combinedcostunits$. The integer is the number of configurations, and the area is that of a hardware configuration.

- We define an application profile to block cost mapping function $PB :: (vector \in \mathbb{Z}^I, int) -> cost$, which takes an application profile vector and a block type, and returns the block's cost, for that particular application profile.

Given an application profile set APS, a profile-to-block function PB, an objective function O and a hardware configuration cost function C, the overall goal is to find a hardware configuration set HCS, such that:

$$\begin{aligned} &\forall a \in \text{ APS } \exists\, h \in HCS \text{ such that} \\ &\forall j \in \{1 \ldots N\}\ PB(a, j) \leq h_j \end{aligned} \tag{4.1}$$

and

$$\begin{aligned} &O(k, max(C(h) \mid h \in HCS)) \text{ is minimised} \\ &\text{where } k = |HCS| \end{aligned} \tag{4.2}$$

For every application profile in the AP set, there is a configuration in the HC set which at least satisfies its requirements for every type of block. The terms *area* and *cost* are deliberately undefined, because they depend on the target technology. It is likely that they will be the same, but we reserve the ability for the cost function to consider composite global effects on the area required.

4.5.5 Procedure

In this section we combine the novel ideas presented here with the standard static design flow to produce a reconfigurable design procedure. Together, the procedure and framework produce the reconfigurable design methodology.

The targeting procedure is described in pseudo code as follows:

1. Capture system requirements
2. Perform preliminary system design
3. Extract decision variables
4. Separate system into static products
5. Capture system constraints (inequalities)
6. Extract system cost function
7. Minimise system cost function
8. Direct intra-subsystem reconfiguration effort
9. Advance implementation
10. Generate checkpoint control unit

Step 1 is identical to traditional design flows: system requirements are established and refined to produce a well defined set of tasks the system is to perform. Step 2 involves making traditional high-level design decisions, such as the partitioning of the system into subsystems and algorithm selection. With the system's high-level design established and well understood, step 3 uses this knowledge to list the variables upon which its resource requirements depend. We use the nomenclature of Operations Research to refer to these variables as decision variables. Along with the decision variable's name, description and the set of values it can be assigned, we also record the maximum frequency at which it changes. The extraction of decision variables is best done at the subsystem level. With the local algorithmic detail available, decision variables that would otherwise not be identified are made obvious. Those decision variables with zero frequency of change are referred to as static. The product requirements are examined in step 4 to determine how the static variables may be usefully folded into the design. Folding here may involve reducing the range of values a variable may take, so it cannot actually be folded at this stage. The result of folding is a set of system instances, each of which satisfies a subset of the total system requirements. One of the subsets may be the full set of requirements, meaning one system instance is capable of performing the task of any other system instance.

Step 5 involves expressing all system constraints as inequalities in terms of the decision variables. These inequalities require in depth system knowledge to establish,

and may be the product of detailed algorithmic analysis and simulation, as is often performed when designing complex systems.

The system instances are constructed from the same subsystems. They will implement similar (if not identical) algorithms, so subsystem implementation follows the established parameterised IP block design principles. In the reconfigurable solution, the parameters are decision variables. Steps six and seven determine the set of decision variable instances necessary to minimally implement each system instance. The minimisation function models the resource requirements on the target reconfigurable device. We refer to this function as the objective function. Step 6 in the targeting procedure is the derivation of the objective function for each system instance. It is a composite function capturing the total requirement of all resource types in the target implementation fabric. A resource requirement is expressed in terms of the decision variables. Step 7 uses the objective function (step 6), system constraints (step 4) and decision variable details (step 3) to find a more efficient implementation. The details of step 7, the most significant step in the procedure, are given in section 4.5.4. Step 8 directs the application of intra sub-system reconfiguration. This is done by examining the resource requirements of the largest system instance, and identifying where further specialisation effort would produce most resource savings benefit.

A traditional iterative implementation strategy will pass over steps 4-9 many times. Initially it will use low accuracy estimates for the objective function. With each iteration, the implementation advance in step 9 will improve the estimate's accuracy, culminating in it no longer being an estimate, but instead being extracted from the actual implementation.

The separation of the system into system instances will be continuously evaluated. As the objective function accuracy improves, it may no longer be possible to justify the original choice of system instances. For example, the difference in resource requirements between two instances may not be enough to merit the required additional subsystem design effort.

The checkpoint enable signal is produced by the permanently resident control unit. It is an FSM, fed by the decision variables and produces the checkpoint enable signal, and the system configuration to load. The creation of the control unit is the tenth and

final step of the targeting procedure.

4.6 Discussion

4.6.1 Introduction

The combination of the design framework and targeting procedure produces a new methodology with a number of distinguishing characteristics. In this section we list the characteristics and discuss them in terms of their contribution.

4.6.2 Contributions

The primary strength of our methodology is its non-obtrusive system-level approach. Reconfiguration is treated as an enhancement to existing design techniques, providing a mechanism to create more silicon efficient FPGA based SoCs. It is non-obtrusive because it simply directs the familiar development of parameterised subsystems; the direction being the specification of parameter ranges for the different system instances. The system-level view considers the cost-speed tradeoff of the configuration architecture, leading to the checkpoint run-time framework. It also directs intra-subsystem and inter-subsystem design effort to where it is of most benefit.

Unlike any other approach, our methodology exploits knowledge of the system in operation, and hence its effect on system resource requirements, at design-time. Conventional reconfigurable wisdom says this operational knowledge can only be discovered at run-time. To take advantage of such knowledge, the system must allocate resources fluidly at run-time, reacting to changes in the operational requirements; however this is not feasible due to the complexity of place and route. Our methodology avoids this conundrum by acquiring certain operational knowledge at design time. This is done by constructing a set of equations (describing the objective function) from what is normally implicit system knowledge, such as the algebraic relationship between two decision variables. Such a relationship has little bearing on conventional design, where systems are a compounded worst-case, having been constructed from the worst-case design of all sub-systems; however a reconfigurable design can use algebraic relation-

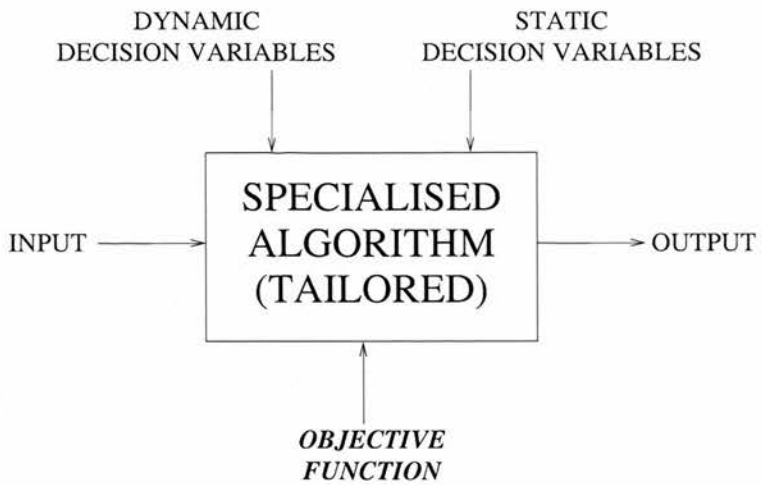

Figure 4.5: Implementation of a Reconfigurable Algorithm Enhanced by Knowledge of the Global Objective Function

ships between decision variables to reduce the resources required to satisfy system demands.

We capture this unique feature of our methodology in Figure 4.5. It differs from the conventional reconfigurable diagram (Figure 2.3) through the additional source of information feeding the custom algorithm design - the objective function.

Our methodology's distinguishing features are listed below. They may be grouped as follows: 1-4 concern Restricted Design Freedom, because their existence arises from design and architectural constraints, while 5-6 concern abstraction, because their strength derives from implementation independence. We explore these distinguishing features under their respective group names in the rest of this section.

1. Real-time system applicable.
2. Three fixed reconfiguration frequencies.
3. Automatic extraction of inter-subsystem reconfiguration potential.
4. Directs intra-subsystem reconfiguration with global objective.
5. Independent of tools and languages.

6. Independent of FPGA fabric.

4.6.3 Restricted Design Freedom

As we approach the multi-billion transistor SoC, an emerging rule of thumb is that designers need freedom from choice, giving impetus to platform based design. The principle behind a platform architecture is to concentrate design effort where it offers most value, and remove it from tasks common to all SoCs, such as physical layout and verification. Removing these freedoms in effect gives more time to concentrate on the remaining, more valuable, design freedoms. Our methodology adopts a similar approach to reconfiguration with its use of the run-time framework. The restrictions are as follows:

1. Full configuration memory accesses are specified at design-time.
2. Full configuration memory access occurs only at checkpoints.
3. The checkpoint sequence (number, duration and frequency) is determined at design-time.
4. In-situ reconfiguration is limited to local configuration memory hooks provided by the architecture.

The justification for the particular restrictions imposed by the methodology is now given: A tradeoff exists between reconfiguration speed and silicon area. As reconfiguration speed increases, the required silicon area increases, as discussed in chapter 3. All resource savings made with reconfiguration come from data folding. If all decision variables are assumed to have the same resource saving potential if folded, it follows that the best resource-savings to reconfiguration-cost-ratio will come from variables which change infrequently.

These constraints on design freedom reduce the reconfigurable tradeoff space. The temporal dimension introduced into the design space by reconfiguration changes from being continuous to being discrete, synchronous and design-time fixed. This injects immediate structure into what is often an ad hoc situation. No longer is the designer

free to consider the merits of ad hoc access to the global configuration interface and the potential for generating circuitry at run-time is limited to what is possible with local in-situ configuration memory hooks. The intention of these restrictions is to maximise the resource savings to effort ratio, whilst maintaining enough flexibility to be applicable across a range of RLSoCs.

There is a clear link between the constraints listed above and the restricted design freedom characteristics listed in the contributions subsection. Restricting both checkpoint sequence generation and full configuration memory access to design time, ensures no complex design decisions or circuit generation is attempted at run-time. This makes real-time systems feasible.

The objective function directs design effort from a system-level - avoiding ad-hoc local optimisation as performed in other approaches[135][164]. A global view is essential when trying to reach near minimum system resource usage. Our methodology considers the total cost of implementation, including the reconfiguration architecture silicon area and off-chip I/O memory buffer.

4.6.4 Abstraction

4.6.4.1 Languages and Tools

Our methodology's independence from design language, tools and implementation forms a high-level of abstraction. In an era of platform based design, abstraction is at the heart of all systems, creating the need for a methodology capable of guiding high-level design decisions.

We believe that achieving resource savings through reconfiguration is not tied to the particular language used, but to the methodology applied. Other work emphasises the importance of language expressiveness to reconfigurable system design, which may improve productivity, but does not provide an insight on how to harness reconfiguration. For example, the Lava[22] language from Bjesse et al. allows circuitry to be described using higher-order functions and polymorphism and allows layout to be captured without the use of layout combinators[146]. Therefore, more elegant, readable descriptions of circuitry are possible, analogous to the purported advantages of soft-

ware functional programming languages such as Haskell when compared to C.

Most previous work on run-time reconfiguration, for example JBits, takes a bottom-up approach to design by tailoring logic slices and constructing cores by progressive composition. The most abstract realisation in JBits, the run-time parameterisable core (RTPCore), does not provide any methodology for the designer other than composition. Another approach to design of systems which require reconfiguration is to use a top-down methodology, constructing systems from plug-compatible components, such as programmable multi-function cores[110]. Our methodology combines both design approaches: intra-subsystem reconfiguration is bottom-up design and inter-subsystem reconfiguration is top-down design. It extends the MacBeth and Lysaght top-down concept of plug-compatible components, to plug-compatible systems. The UART component used as a demonstrator in their paper[110] is trivially small when considered in terms of today's multi-million gate SoC designs. The authors offer no methodology to scale the approach beyond a single component to multiple inter-operating components.

4.6.4.2 Architecture

Today's platform FPGAs and SOCs are constructed from a heterogeneous mixture of computational fabrics. The new methodology presented here recognises this, and by reducing the conceptual complexity of reconfiguration, heterogeneous architectures become as straightforward to target as the homogeneous fabrics focused on in previous approaches (section 4.2.3).

The checkpoint run-time framework forms a layer of abstraction, separating the reconfiguration details of a particular architecture from the design targeting and optimisation procedures.

4.7 Summary

Reconfigurable RLSoC demands a different approach to those ideas in the literature proposed for general purpose systems. In this chapter we have motivated, described and discussed a run-time framework and design methodology for targeting RLSoCs at reconfigurable computing fabrics. Our approach aims to limit design freedom and

hence reduce the complexity of reconfigurable design, whilst simultaneously keeping most gains through reconfiguration attainable.

Chapter 5

Case Study: UMTS Physical Layer Processing

5.1 Introduction

The Checkpoint design methodology described in Chapter 4 is now applied in a large case-study, the physical layer processing performed by a UMTS base station. For an overview of UMTS, please refer to appendix B. This chapter begins by introducing the commercial ASIC design the case-study is based upon in Section 5.2. Then the particular sub-blocks focused on in the case-study are described in Section 5.3. Section 5.4 describes how the checkpoint methodology is applied with Section 5.5 exploring the changes required to the Xilinx Virtex architecture to implement the Checkpoint Runtime Framework. Section 5.6 discusses the assumptions that had to be made to perform the case-study. The resulting logic and memory resource savings are presented in Section 5.7 and the outcome of the case-study is summarised in Section 5.8.

5.2 UMTSSOC Overview

The UMTSSOC processor is a sophisticated SOC ASIC design. It performs the physical layer digital signal processing (DSP) for the 3GPP UMTS basestation in both the direction of the User Equipment (UE), referred to as the downlink, and the direction of

the basestation, referred to as the uplink. It is designed to fully comply with Release 4 of the 3GPP specification. This section outlines where UMTSSOC fits into the overall basestation solution.

The UMTSSOC functionality covers the top part of what is classically referred to as the physical layer in the OSI model, and the lower part of what is referred to as the medium access controller. The rest of the physical layer is the ADC/DACs, analogue stages and antenna. Above, the UMTSSOC interfaces with the rest of the medium access controller, which in the case of the basestation, among other things, assigns which calls are processed by the channel element.

The UMTSSOC is capable of supporting up to 64 calls simultaneously. A basestation is configured for a particular cell site by adding the required number of UMTSSOC equipped cards. Each card increases the maximum number of calls that can be processed by 64 for each chip on the card. The basestation may have up to six sectors, each sector with 2 antennas and 3 carrier frequencies. This produces a total of 36 different radio sources in a maximum capacity basestation. Of those 36 potential radio sources, each UMTSSOC is capable of both transmission and receipt of calls on 12 of them - equivalent to 3 sectors and 2 carrier frequencies or any combination thereof. The term 'call' is used here to refer to a dedicated connection between the basestation and the UE. The nomenclature arises from the telecommunication industry's historical focus on voice traffic, but here it refers to both voice dedicated channels and data dedicated channels.

UMTSSOC has two uplink feeds, one on-time and the other delayed by 1.026 frames. The reason for having a delayed feed is explained in Section B.2.3. Each feed has 36 radio sources, sampled at twice the chip rate, producing a "full" and "half-chip" component for each radio source. A chip refers to the time taken to transmit a single bit in the spread and scrambled transmit stream. The main clock frequency of UMTSSOC was chosen to be 184.32MHz since many processing steps must be performed per chip received. With the UMTS chip rate at 3.84Mcps, there are 48 clocks per chip. Such a high ratio of clocks to chips is indicative of the heavy signal processing involved to achieve satisfactory performance. The demand for a high clock rate can partially be illustrated by the searcher's need to correlate 2 antenna sources, both full and half chip

Partition	Logic Gates	Memory Bits	% UMTSCE Silicon Area
Preamble Detector	117,000	526,080	12.1
Searcher	407,613	2,235,104	15.5
RAKE Receiver	961,973	4,056,750	28.5
Power Controller	117,000	1,660,224	7.9
Transmitter	301,480	543,776	6.9
Extended Soft Information Processor	525,078	5,560,848	29.1

Table 5.1: UMTSCE ASIC Resource Requirements

components (4 streams) for up to 64 users.

5.3 UMTSCE Outline

The ASIC implementation of UMTSSOC has 8 partitions which together perform all the physical layer digital signal processing. The 6 partitions directly involved in channel processing are considered here for parameterisation and will be referred to as the UMTSCE. Table 5.1 lists the partitions and their ASIC resource requirements.

The rest of this section gives a high-level overview of how the UMTSCE partitions interact. Section B.2 gives a short description of the function performed by each block, with simplified block diagrams of the partitions used throughout the descriptions to aid understanding.

A central challenge of spread spectrum CDMA is the receiver architecture for finding the strongest multi-path component signals from the UE. As stated in Section A.6.2, a CDMA system is interference limited, so the aim of a good receiver architecture is to extract a low power signal and so minimise the required transmit power. Movement of both the UE and objects in its vicinity result in a rapidly changing radio propagation channel between the UE and the basestation. The single transmitted signal may be reflected off various objects, and hence follow many different paths to the base station antennas. The different paths arrive at the antenna at different times, so

the receiver must search continuously across a range of delay spreads for the strongest multi-path signal component.

When the UE is first switched on, it initiates contact with the basestation by transmitting a RACH preamble. The basestation must listen continuously for this RACH preamble, which is demanding since the UE may be anywhere within the cell. The basestation must therefore correlate continuously the sampled antenna stream with expected signatures in the lookout for UEs wishing to establish a connection. This continuous scanning of the antenna sample stream for initial UE RACH preambles is performed by the Preamble Detector (PD) partition.

On establishing contact with the basestation through the RACH preamble, the UE may request a dedicated channel. A dedicated channel is used for voice or data at different speeds depending on the spreading factor used. It is closely monitored in an attempt to ensure that the strongest multi-path components are used for reception as described above. It is not necessary to search the entire sample stream continuously for communication from the UE on the dedicated channel since, relative to the basestation the time of arrival of the UE transmission is known, initially advised by the preamble detector on channel setup. As radio channel changes occur, for example as the UE moves from outside to inside a building, the strongest multi-paths will arrive at different times. Due to the continuous and rapid reassessment of the channel at the receiver, these changes may be tracked fairly smoothly. The search space is restricted to a window of 192 chips in the UMTSCE implementation, equivalent to 7.5Km, and is searched every 0.6ms. The reason why a 7.5Km space is searched every 0.6ms is not that the UE may have travelled that distance, but instead that a small change in the channel conditions may mean that a reflection off a distant object has become a strong multi-path.

The searcher outputs the best set of delay offsets for each channel in terms of the number of half-chips from "base-station time". These are passed to the RAKE receiver which interpolates the half-chip sample resolution to further fine tune the multi-path components. The RAKE outputs the UE data bits as estimates or "soft bits" after de-spreading and de-scrambling. The estimates are fed into the Extended Soft Information Processor (ESIP) for channel decoding to finally produce the transmitted "hard bits".

One method of minimising transmit power as described in Section B.1.2.2 is by way of power control bits interleaved with the data providing a feedback path. The RAKE provides this feedback mechanism to the power controller partition.

The Preamble Detector, Searcher and RAKE all form the uplink chain. The ESIP and Power Controller are shared between the uplink and the downlink. The downlink chain is simpler and has only one dedicated partition - the transmitter. The transmitter spreads and scrambles data destined for the UE, and may output more than one transmit stream per UE for soft handover - i.e. moving between sectors on the same basestation.

5.4 Methodology Application

5.4.1 Introduction

In this section we describe how the targeting procedure is applied to the signal processing performed in a UMTS base station (a so-called "Node B")[2]. To avoid the large implementation effort, all results are derived from the commercial UMTSSOC implementation. The UMTSSOC is an existing ASIC design and we work backwards to determine the size of an equivalent reconfigurable implementation instead of designing a completely new reconfigurable solution. This has the advantage of both providing an assurance that the system engineering is correct and makes a study of such a large complex system feasible. It has the disadvantage that key system engineering decisions that are likely to be made differently for a reconfigurable system solution are artificially imposed. The ASIC is designed for a single sector carrier so that it provides a scalable solution. For example, economies of scale like designing a subsystem to satisfy the requirements of several sector carriers are not realised.

It is still possible for the reconfigurable system solution to exploit different product configurations and dynamic resource requirements. For example, the size of the CMF in the searcher is dependent on the size of the search window. If, instead of targeting an ASIC, the UMTSCE was to be targeted at a family of reconfigurable fabrics, then the implementation could be tailored to perform exactly what is required and therefore minimise the fabric size required. For instance, a basestation configuration for an urban environment may have a much smaller search window than that of a basestation

configuration for a rural environment.

The design of such a UMTS processing engine is driven, at the top level, by the Node B configurations. This set of configurations, in turn, depends on the overall system requirements, i.e. (a) static, product deployment parameters such as cell size and environment (e.g. indoors, urban and rural) and (b) more dynamic parameters such as the traffic mix at any given point in time (e.g. voice, high-speed and data).

The parameterisation of the different subsystems within the processing engine depends on different aspects of the various system configurations supported:

- For subsystems supporting transport channel encoding (channel encoding, rate-matching, interleaving, etc.) for the downlink and decoding for the uplink, the key factors are primarily the choice of channel coding technique (convolutional or turbo) and the total aggregate engine data throughput.

- The number of channels processed controls the size of the rake receiver and the transmitter (modulation and spreading functions). The rake receiver also depends on the number of "fingers"[1] per channel received.

- The size of the multi-path searcher for detecting new multi-path components depends heavily on the number of channels to be searched and the search window size. The latter depends upon the possible spread of multi-path delays which in turn is dependent on the type of environment and cell size.

- The RACH preamble detector[2] capabilities will be set by the level of expected usage of the RACH in the cell's uplink.

Thus different subsystems will have their most demanding instance in different product configurations. For example, the transmitter is most heavily loaded when there are a large number of low data rate (e.g. voice) users, while the channel decoders might be most heavily taxed with a low number of high data rate channels employing the more intensive (in terms of decoding) turbo codes. These two situations, due to

[1] A finger is one correlator unit allocated to one multi-path component to be detected and demodulated.

[2] The preamble detector hunts for short signature sequences sent by the mobile terminals requesting permission to send a short burst of data using the RACH (random access channel).

air interface constraints, are mutually exclusive at any given point in time in a given Node B. Hence, a channel processing engine designed to cover all requirements simultaneously will have significant portions of its functionality unused at any given point in time. Runtime reconfiguration can take advantage of these time varying demands, by only requiring enough resources to satisfy the largest single instance of operational demand.

5.4.2 Step 1: Capture system requirements

Like all engineering design, the procedure begins with some form of hand-off from the customer. Its purpose is to unambiguously specify the high-level system requirements to ensure the best implementation.

The requirements for the UMTS Node B are as follows:

1. Node B physical layer processing for UMTS release 4
2. Interfaces
   (a) Antenna samples interface
   (b) User data interface
   (c) Control interface
3. Cell radius (Km)
   (a) Pico: 0-0.2
   (b) Micro: 0.2-1.0
   (c) Macro: 1.0-30.0
4. Antenna: Minimum 2, Maximum 12
5. Sectors: Minimum 1, Maximum 6

These requirements will have been shaped by the initial (entire product) specification work. This will include considerations as diverse as the Node B's overall cost,

physical footprint and reliability. For example, a cost assessment of the analog stages may have shaped the upper limit on the number of antennae.

It is worth noting that the UMTSSOC solution is designed to meet the demands of a single sector carrier. When a Node B is configured, UMTSSOC cards are inserted to match the number of sector-carriers to be serviced. Additional opportunities for reconfiguration, which may exist as the size of the problem scales, cannot be exploited when basing the reconfigurable solution upon the UMTSSOC.

5.4.3 Step 2: Perform preliminary system design

With the interface to the rest of the system defined, and the functional requirements stated, system design can begin. The specification documents created by 3GPP provide a clear definition of what the system must perform. The second step in the targeting procedure involves decomposing the system into its subsystems and defining their interfaces.

We require the major subsystems described in section B.2, namely, searcher, preamble detector, forward error correction, RAKE receiver, power controller and transmitters. Where there is a choice of which algorithm to use for a particular subsystem, simulation work may be performed to help select one. For example, innovation within the RAKE receiver is likely to have an effect on the overall Node B performance.

A block diagram and short summary of the function performed by each subsystem is given in section B.2. The classic systematic procedure of dividing the system into subsystems is followed, allowing implementation to progress independently across subsystems once the interfaces are defined.

System simulation and analysis work (along with fundamental results in information theory) will help to define many of the system parameters. An example result is that the antenna outputs are sampled at twice the chip rate, and use 4 bits for both the I and Q components. Another interesting high-level design decision is how to perform despreading of radio frames. The control bits specifying the data spreading rate are distributed across the UMTS radio frame, meaning that the entire frame must be received before the spreading rate is known. There are several approaches to dealing with this

issue, but the one chosen here is to have two antenna data buses, one delayed in time by a radio frame using off-chip RAM and the other arriving on time. The control data is extracted using the on-time data bus, just in time to enable informed despreading on the delayed data bus.

In this case study, to obtain an accurate model of the computational requirements, it is insufficient to consider in isolation the UMTS specification or the information theoretic understanding of CDMA communications. Instead, it is necessary to model how the disparate factors combine to engineer the solution. In essence, the best model is the system.

The system may be viewed as a hierarchical collection of function blocks. At the highest level, the input is antenna samples and the output is OSI layer 1 bits. This level of abstraction offers little insight into the computation performed. At the next level down in the system decomposition tree, we get closer to where the computation is performed, but it is not until we drill down to the leaf function blocks, that the actual function being performed becomes apparent.

Upon inspection of the block diagrams and the accompanying algorithm descriptions, it is possible to list the variables influencing the system's resource requirements. The definition of what constitutes a variable needs to be clarified. We define it as a parameter which is dependent on the implementation; so the number of chips in a control symbol is not a variable, but the number of antenna on the base station is a variable.

We define a user to be a four tuple of connection parameters:

$$user = \{type, rate, TTI, code\} \tag{5.1}$$

This tuple represents the high-level characteristics of the required receiver architecture. Type indicates whether the connection is dedicated voice, dedicated data, or if the connection is over the shared channel (SCH). Rate is the speed of the connection (Kbps), TTI is the transmission time interval and code is the type of forward error correction. We list the values these parameters can take in table 5.2, together with the maximum frequency at which they change.

The maximum frequency of change is derived from the UMTS radio frame interval of 10ms. On every radio frame there is the option to change an existing connection's

Connection Parameter Name	Possible Values	Change Frequency
Rate (Data)	16,32,128,384,768 Kbps	100Hz
Rate (Voice)	4.75,5.15,5.9,6.7,7.4,7.95,10.2,12.2 Kbps	100Hz
Rate (SCH)	16,32,64	100Hz
TTI (data)	10,20,40,80ms	100Hz
TTI (voice)	20ms	100Hz
TTI (SCH)	20ms	100Hz
Code (data)	Turbo	100Hz
Code (voice)	Convolutional	100Hz
Code (SCH)	Convolutional	100Hz

Table 5.2: Connection parameters value ranges

Variable Name	Possible Values
Cell radius	0-16Km
Number of sectors	1-3
Antennas per sector	1-4
Searcher window size(chips)	50-400
Fingers per user (urban, suburban)	8
Fingers per user (rural)	16

Table 5.3: Static system variables

settings. It is also a suitable frequency to perform any waiting connection setups or tear-downs. *Therefore, the frequency of 100Hz forms the checkpoint rate in this case-study.*

5.4.4 Step 3: Extract decision variables

In the UMTSSOC there are 20 logic blocks and 13 memory blocks to be considered for parameterisation. Together, these 33 blocks construct the 5 partitions of UMTSSOC, therefore they form the hardware configuration vector. The parameterised cost equations describing the hardware blocks are extracted in section 5.4.7. Details of the 33

hardware blocks are given in Appendix C.1.

The decision variables are:

- The integer c, which is the number of hardware configurations in the optimal set.
- For each of the c hardware configurations, a tuple of 33 values describing it.

Both the size of the set c, i.e. the number of configurations, and the values of each vector in the set, i.e. the resource demands of the hardware configuration, are important. The decision variable c may be defined implicitly by the size of the set of hardware configurations. The goal of the targeting procedure's optimisation step is to find the covering set of hardware configuration vectors which optimise the combined cost of the FPGA and configuration storage cost, as we will see in 5.4.8.

5.4.5 Step 4: Separate system into static products

Business decisions drive step 4 in the targeting procedure. The location of a Node B characterises its operation and specification. Step 4 takes advantage of the different deployment scenarios by folding this knowledge into the design. The knowledge is in the form of values for the static variables, rather than the generalised value ranges.

In cellular network planning, there exists a tradeoff between a cell's capacity and its coverage. As the cell's coverage increases, its capacity decreases. It follows that to satisfy the call densities in built up urban areas, cell sizes are smaller than those in rural areas. Based upon marketing and system engineering knowledge within Lucent, we split the system into three typical static products: rural, urban and suburban. The particular instances chosen are listed in table 5.4. They do not cover all required products, since cell characteristics vary according to the building materials, and terrain in the environment served, but they provide a good example for the purposes of this case study. The primary difference between the three products is the cell size. The larger a cell is, the less capacity it has. The Rural basestation may have large geographical features such as hills providing radio wave reflections, so its search window is the largest. The environment of an Urban or Suburban basestation is comprised of buildings, traffic and general street furniture providing many radio reflections in close proximity to the

Table 5.4: List of Basestation Product Configurations

Product Configuration	*S*	*R*	*U*
Environment	Suburban	Rural	Urban
Approx. max. cell radius [km]	4	16	2
Traffic levels	High	Low	High
No. of sectors	3	3	1
Antennas per sector	2	4	2
Searcher window size [chips]	100	400	50
Max. no. of fingers per user†	8	16	8

†Dedicated channels only: lower figures for RACH

transmitted signal. Correspondingly, the searcher window size is considerably smaller than the Rural basestation product. Due to the size of the cell, the signal to noise ratio at the basestation antenna will generally be worst in the rural basestation, so there is a significant gain for investing more resources in RAKE receiver fingers.

5.4.6 Step 5: Capture system constraints (inequalities)

The ultimate objective of this methodology is to produce an optimal set of hardware configurations to implement the system. This requires the input of application specific knowledge to bound the required functionality. However, to avoid ad hoc implementations it is important to ensure that reconfigurable design flow and expert knowledge are kept orthogonal. Step 5 captures the expert problem specific knowledge, providing a well defined interface between the rest of the reconfigurable targeting procedure and specific details of the application. In this step we illustrate how expert UMTS system engineering knowledge shapes the reconfigurable design space.

If we examine the combinations of the different connection types in table 5.2, we see there are 31 different connections which a user may request. For example, {data,32Kbps,20ms,8} is a valid 4-tuple in the set of 31 possible 4-tuples. This forms a more elegant representation of the problem for the optimisation step. When modelling the state of the system inside any radio frame (checkpoint) period, we will have 0-

64 users, each using one of the 31 connection variants. This application profile can be represented with a vector $\in \mathbb{Z}^{31}$, giving the number of users for every connection variant:

- We define an application profile set (APS) to be a set of application profiles.
- We define an application profile, a, to be a vector $\in \mathbb{Z}^{31}$

The APS contains all valid instances the system is required to perform. It is not arbitrary in $\mathbb{Z}^{31}$, but is limited by a couple of factors. The first factor is the available capacity in the cell. In W-CDMA UMTS, all users share a 5MHz frequency bandwidth on the uplink and downlink, placing a fundamental interference limit on the capacity of the system. The transmit power of each user (similarly for downlink) is reduced to limit interference, however, the power must keep the channel above the minimum signal-to-noise ratio for satisfactory quality. A cell is at maximum capacity when every user in the cell is at the minimum signal-to-noise ratio required to satisfy call quality. Modelling capacity and coverage of a UMTS cell is a complex and important part of network planning. Such a model maps between signal-to-noise levels within the cell and whether a call can take place or not.

We draw an illustrative picture in Figure 5.1, with the number of users against service type, to form a histogram. Only a small number of 384Kbps data transfers can be in progress before the air interface is saturated, forming one application profile. A radically different profile would be 60 low rate voice calls, spread across many slightly different service types.

Unfortunately, the expertise to analyse UMTS capacity models and hence the constraints they place on the optimisation problem is not available for this project. Such expertise would consider inter-cell interference, cell shrinkage, capacity versus coverage and the different quality of service demanded by different traffic types. However, we believe that the definition of application profiles by a UMTS capacity expert is feasible. Such knowledge must be available to define any UMTS system - it is implementation independent. Instead of defining the entire application profile space, system engineering expertise within the industrial partner on this project (Lucent Technology) provided corner point traffic profiles which maximise a UMTS cell's capacity. It is

reasonable to assume that these profiles will either match or at least come close to the maximum processing demands placed upon the system. The Lucent knowledge is based upon a commercial basestation solution.

The corner points for the UMTS cell capacity as defined by Lucent experts are as follows:

1. Fully loaded by voice users using dedicated channels (DCH).
2. Fully loaded by medium-rate data users using DCH with a long TTI (transmission time interval, i.e. packet duration).
3. Fully loaded by high-rate data users using DCH.
4. Fully loaded by a large number of low-rate data users using the FACH/RACH mode of communication [3].

For the different product configurations, "fully loaded" corresponds to different levels of peak traffic demand—as shown in Tables 5.5 and 5.6. It should be noted that an actual implementation may require a few more intermediate dynamic instances—however the above represent the corner-points for the purposes of estimating resource requirements. Table 5.5 lists the dynamic instances for the suburban product. Table 5.6 lists the dynamic instances for the rural product.

The second constraint upon the APS is a result of this study being based upon an ASIC design of UMTSSOC. An upper bound of 64 is placed upon the number of users which may be supported. This figure is capable of satisfying most operating demands, but it will unnecessarily constrain the system (from an air-interface capacity perspective) in some operating scenarios, such as, exclusively very low rate voice calls.

$$\sum_{i=1}^{31} a_i \leq 64 \; \forall a \in APS \tag{5.2}$$

5.4.7 Step 6: Extract system cost function

With the high-level architecture defined, the teams responsible for each of the top-level blocks begin preliminary design. In the beginning, before design entry has started,

Table 5.5: List of Dynamic Instances for Suburban Product Configuration.

Dynamic Instance	*S*1	*S*2	*S*3	*S*4
No. of users	60	12	3	8
Peak data rate per user [kb/s]	12.2	64	384	64
Uplink spreading factor	64	16	4	16
TTI length [ms]	20	80	10	20
Peak total throughput [kb/s]	732	768	1152	512
Uplink channels	DCH	DCH	DCH	RACH
Coding†	C	T	T	C
No. of fingers per user	8	8	8	4
Av. no. of links per DL channel	1.5	1.5	1.5	1
Total no. of common channels	7	7	7	14
No. of preamble detectors	1	1	1	3

†'T' implies turbo encoding and 'C', convolutional encoding.

Table 5.6: List of Dynamic Instances for Rural Product Configuration.

Dynamic Instance	*R*1	*R*2	*R*3	*R*4
No. of users	20	4	1	8
Peak data rate per user [kb/s]	12.2	64	384	64
Uplink spreading factor	64	16	4	16
TTI length [ms]	20	80	10	20
Peak total throughput [kb/s]	244	256	384	512
Uplink channels	DCH	DCH	DCH	RACH
Coding†	C	T	T	C
No. of fingers per user	16	16	16	8
Av. no. of links per DL channel	1.5	1.5	1.5	1
Total no. of common channels	7	7	7	14
No. of preamble detectors	1	1	1	3

†'T' implies turbo encoding and 'C', convolutional encoding.

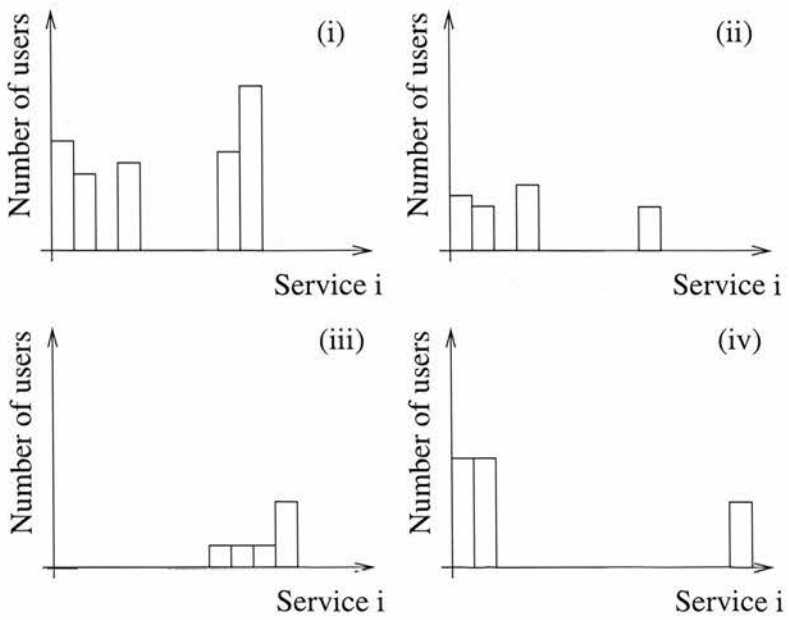

Figure 5.1: Four examples of user distribution across the different service types.

only rough estimates of the resource requirements can be made. As the design progresses, estimates turn into actual resource requirements, converging on the actual requirements.

In this case study, the complete UMTSSOC solution already exists, and we are working backwards towards a reconfigurable solution. Final equations expressing the cost of each hardware block can be extracted, although there are several assumptions that must be made, which are discussed in section 5.6.

To keep the case study clear and concise, we summarise the cost function here, placing the details of extraction in appendix C.1. Extraction involved studying algorithm design documents and hardware implementation details. During this process, each block's dependency on the application profile was identified. The dependency variables have a simple link to the application profile vector. For example, some hardware blocks depend on the total throughput, which can be easily calculated by multiplying the total number of users of each service by the service rate and summing.

Table 5.7 lists all the parameters used in the cost functions, together with how they may be calculated. "Static" indicates that the parameter doesn't change once the

Parameter	Calculation and Dependency
window	static
sources	static
max.rate	static
users	function(profile)
common	function(profile)
rach	function(profile)
fingers	4-tuple
rate	4-tuple
TTI	4-tuple
coding	4-tuple
s.f.	function(4-tuple)

Table 5.7: Cost function parameters and their calculation dependencies.

product is deployed. "function(profile)" indicates that the parameter is dependent on the complete profile, for example, the amount of RACH preamble detection that needs to be performed is dependent on how loaded the air interface is. "4-tuple" refers to the 4-tuple defined earlier, i.e. each service is a 4-tuple, and the parameters listed as dependent on the 4-tuple are found by performing a simple (static) table lookup. "function(4-tuple)" indicates the parameter is calculated from the 4-tuple. For example, the spreading factor is dependent on the rate of the particular service. We distinguish the (static) service 4-tuple values from the static basestation setup parameters listed in table 5.3. This is done because they are strongly linked to the dynamic service profiles.

We present the summary of the cost functions in two tables: logic gates (table 5.8) and memory bits (table 5.9). The ceiling function subscript denotes the granularity of the ceiling function, i.e. the size of the resource requirement steps between different implementations. Each hardware block (logic and memory) is assigned a unique name and the appendix equation from which it is derived is given.

Block	Cost Function	Eqn.
L1	$26{,}666 \cdot \text{rach}$	C.1
L2	$11{,}733 \cdot \text{rach}$	C.2
L3	$14{,}000 \cdot \text{rach}$	C.3
L4	$5{,}333 \cdot \text{rach}$	C.4
L5	$\left\lceil \frac{4 \cdot \text{users} \cdot \text{window}}{12{,}288} \right\rceil_1 \cdot 58{,}192$	C.6
L6	$\left(\frac{window+256}{448}\right) \cdot 29{,}562$	C.6
L7	$\left\lceil \frac{4 \cdot window \cdot users}{12{,}288} \right\rceil_1 \cdot 13{,}395$	C.7
L8	$sources \cdot 2{,}732$	C.8
L9	$\lceil \sum_1^{users} fingers \cdot 2{,}013 \rceil_{128{,}873}$	C.10
L10	$\left\lceil \left(\frac{users+common}{192}\right) \cdot 301{,}480 \right\rceil_{37{,}685}$	C.11
L11	$\lceil users \cdot 1{,}828 \rceil_{14{,}625}$	C.12
L12	$\lceil users \cdot 2{,}430 \rceil_{19{,}445}$	C.13
L13	$\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{25{,}400}{max.rate}\right) \right\rceil_{3{,}175}$	C.14
L14	$\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{13{,}700}{max.rate}\right) \right\rceil_{1{,}712}$	C.15
L15	if 'c' $\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{107{,}500}{max.rate}\right) \right\rceil_{1{,}344}$ + if 't' $\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{322{,}500}{max.rate}\right) \right\rceil_{4{,}031}$	C.16
L16	$\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{20{,}200}{max.rate}\right) \right\rceil_{2{,}525}$	C.17
L17	$\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{13{,}750}{max.rate}\right) \right\rceil_{1{,}718}$	C.18
L18	if 'c' $\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{40{,}000}{max.rate}\right) \right\rceil_{5{,}000}$ + if 't' $\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{160{,}000}{max.rate}\right) \right\rceil_{20{,}000}$	C.19
L19	$\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{11{,}520}{max.rate}\right) \right\rceil_{1{,}440}$ if TTI>10ms	C.20
L20	$\left\lceil \left(\sum_1^{users} rate\right) \cdot \left(\frac{17{,}352}{max.rate}\right) \right\rceil_{2{,}170}$ if TTI>10ms	C.21

Table 5.8: Cost functions for logic blocks.

Block	Cost Function	Eqn.
M1	$(\sum_1^{users} fingers) \cdot 4000$	C.23
M2	$\sum_1^{users} TTI \cdot rate \cdot 4.5$ if TTI>10ms	C.26
M3	$window \cdot users \cdot 76$	C.25
M4	$((\sum_1^{users} 3 \cdot rate \cdot (TTI + 30\text{ms})) + 714) \cdot 5$ for TTI>10ms	C.24
M5	$windows \cdot users \cdot 88$	C.27
M6	$\sum_1^{users} fingers \cdot 1,664$	C.28
M7	$users \cdot 10,240$	C.29
M8	$sources \cdot 40,960$	C.30
M9	$480,000 \cdot (\sum_1^{users} \frac{1}{s.f.})$	C.31
M10	$rach \cdot 98,304$	C.32
M11	$users \cdot 4,096$	C.33
M12	$(1.5 \text{ users} + common) \cdot 2,730$	C.34
M13	$rach \cdot 57,344$	C.35

Table 5.9: Cost functions for memory blocks.

5.4.8 Step 7: Minimise system cost function

Before we review the minimisation problem, we list the key factors shaping the problem of implementing the reconfigurable UMTSSOC:

1. 0-64 users.
2. A user may request 1 of 31 different services.
3. Each service has different resource and processing requirements.
4. Each service has different air interface requirements.
5. An upper limit on the air interface capacity exists.
6. The number of each type of service may change at 10ms intervals.

In step 6 we defined the cost function for 33 hardware blocks and in step 5 we found the number of services to be 31. Using these numbers for N and I respectively, the optimisation step may be written as follows:

- We define an application profile set to be a set of application profiles.
  - The application specific expert knowledge defines the combinations of services to be simultaneously supported by the system. For example, the number of each type of voice call and data call.
- We define an application profile, a, to be a vector in $\mathbb{Z}^{31}$, where 31 is the number of different services offered.
  - The system must support up to 64 users spread across the 31 different service types. The valid service profiles are determined by the expert knowledge provided in step 4 of the methodology.
- We define a hardware configuration set to be a set of hardware configurations.
  - A hardware configuration corresponds to the bitstream for specifying the system implemented by the FPGA.
- We define a hardware configuration to be a vector in $\mathbb{Z}^{33}$, where 33 is the number of hardware blocks in the system.
  - The architecture of UMTSSOC can be described in terms of the size of its 33 constituent building blocks. The hardware configuration vector contains this information, for example the number of gates (logic) or bits (memory) required to implement the block. Similarly, for a heterogeneous fabric, such as a platform FPGA, we define a hardware configuration to be a vector in $(\mathbb{Z}^4)^{33}$. This allows each block to record the number of each resource type (slice, multiplier, DSP block, processor) present in the target device.
- We define a hardware configuration cost function $C :: (\mathbb{Z}^{33}) -> area$.
  - To optimise the total system cost (configuration storage and FPGA size), there must be a common cost metric. The hardware configuration cost function translates a hardware configuration into the cost metric. This function is system specific, since factors such as volume pricing and the choice of configuration storage will affect it.

- We define an objective function, $O :: (int, area) -> combinedcostunits$. The integer is the number of configurations, and the area is that of a hardware configuration.
  - The objective function finds the minimum combined cost of the configuration store and FPGA size. The *int* input to the objective function is the number of hardware configurations of size *area* required to implement the system. The *area* input is the size of the largest hardware configuration.
  - To avoid a multi-objective optimisation problem, the objective function, O, combines the cost of the FPGA and the cost of the configuration storage cost. This combination function will be dependent on a number of factors, such as the volume in which the system will be manufactured. For example, Urban cells are most densely packed in the Urban environment where UMTS coverage is initially being provided, so the production volume of Urban basestations is going to far exceed that of rural cells. Therefore, a rural basestation product may not be able to get bulk purchase discounts on flash memory as an Urban basestation. Another cost combination factor is the configuration storage medium used (remote network, hard-disk or flash). For example, extreme weather conditions might be outside the operating range of a hard-disk drive, or if the back-haul bandwidth is at a premium it may be costly to transfer configurations over the network.

    The multiple configuration storage cost forms an upper limit on the number of configurations permitted for a particular FPGA size. Beyond this limit it is more cost effective to move to the next FPGA size.
  - The optimisation function is supplied with the number of resources available across all members of the FPGA family, such as the number of memory blocks and LUT resources. With this knowledge, it is able to check if a hardware configuration will fit a particular device. In addition, a hardware block may be supplied with more than one profile-to-block function, providing alternative implementations. These alternatives can be used in an attempt to soak up any uncommitted resources before moving to a larger device. For example, some of the ESIP partition's control functions are

performed in software and they will have an equivalent logic fabric implementation. The logic fabric implementation may require many LUT resources, but if the resources are unused and there are not enough processor cycles available for the software implementation, it is important to be flexible and use the logic to avoid moving to a larger FPGA. The extent to which hardware blocks are supplied with alternative implementations is at the discretion of the design team, and is guided by the output of the optimisation step.

- We define an application profile to block cost mapping function $PB :: (vector \in \mathbb{Z}^{31}, int) -> cost$, which takes an application profile vector and a block type, and returns the block's cost, for that particular application profile.

  - The PB function can be used to determine the hardware configuration vector required to satisfy an application profile. The hardware configuration cost function, C, is used to translate between a hardware configuration vector and the FPGA implementation cost.

    The profile-to-block function, PB, translates between the entire service demand placed upon the system at any instant and the requirements of a particular hardware block. For example, memory block M5 in table 5.9 is dependent on two parameters: window (static) and users (dynamic). The number of users is a function of the entire service profile, and is therefore the sum of all user counts across all services. Equation 5.3 describes the PB function for M5, with the function *count* taking the set of application profiles and the profile number of interest as input and returns the number of users of that particular profile in the APS: $count :: (APS, int) -> int$.

$$PB_{M5} = \sum_{i=1}^{31} count(APS, i) \cdot windows \cdot 88 \tag{5.3}$$

We will now describe the optimisation step formally. Given an application profile set APS, a profile-to-block function PB, an objective function O and a hardware

configuration cost function C, find a hardware configuration set HCS, such that:

$$\begin{aligned} &\forall a \in \text{ APS } \exists\, h \in HCS \text{ such that} \\ &\forall j \in \{1 \ldots 33\}\ PB(a,j) \leq h_j \end{aligned} \tag{5.4}$$

and

$$\begin{aligned} &O(k, max(C(h) \mid h \in HCS)) \text{ is minimised} \\ &\text{where } k = |HCS| \end{aligned} \tag{5.5}$$

As described in section 4.5.3, the static parameters form the first filter in the optimised implementation step. The second stage, or dynamic filter stage, is performed on the APS constrained by the expert knowledge. As explained in step 4, the expert input for step 4 was not made available to this project, so we were not able to perform the optimisation step. Instead we use high-level expert system-engineering knowledge within the project sponsor, Lucent Technology, to supply corner-points which saturate the UMTS air interface. These corner points capture very different application service profiles, and as will be demonstrated by the results, give a clear demonstration of the resource savings possible with the checkpoint methodology. It is noted that some service profiles within the APS may fall between the corner-points, so a small number of additional hardware configurations would be necessary to provide complete coverage.

5.4.9 Step 8: Direct intra-subsystem reconfiguration effort

5.4.9.1 Introduction

Rapid intra-subsystem reconfiguration applies to fabric elements configurable at frequencies of the order of megahertz. Examples include microprocessor instruction streams, DSP blocks and the SRL16 functionality of LUTs.

The Complex Code Match Filter (CCMF) sub-block within the searcher partition is selected here as an illustrative example of intra-subsystem reconfiguration. It represents 67% of the logic in the searcher partition so any resource usage reductions through increased SRL16 usage will have a significant impact at the system level.

First, the UMTSCE ASIC implementation is described. Then the equivalent static design implemented on the Xilinx Virtex FPGA family is outlined and its resource usage is estimated for use as a benchmark with which to compare the runtime reconfigurable designs. The CCMF algorithm involves two variables - the code and data samples. Both variables are considered as candidates for folding into the circuitry. This involves creating dynamically reconfigurable designs and estimating their resource usage. A detailed implementation study is performed for the data-folded design. The results are presented and compared with the static equivalent design, comparing latency, frequency of operation and resource requirement. The section finishes with a discussion of some alternative high-speed reconfigurable techniques not applied in the study.

5.4.10 Algorithm Description

The CCMF performs the correlation (see A.6.10) between the expected code sequence and the incoming data streams of two sources, one 256-chip symbol at a time. The expected code sequence is locally generated and is the result of multiplying the spreading code, scrambling code and known pilot bits used at the UE.

The CCMF top level view is illustrated in Figure 5.2. The code is stored in a code register as shown in the diagram, which is simply two 256-bit registers. The data is stored in the two registers at the top of the diagram, one for the real part (I) and one for the imaginary part (Q). Each data register is a 448-word shift register, where each word is a 4-bit sample of a chip. Only the rightmost 400 words of the data register are output to the DPSBs as a 400 word-vector. The remaining 48 words will be shifted over until they reach the output. Dot Product Sub-block zero (DPSB0) takes words 0-255 as its inputs and computes the results for hypotheses 0-47, one per clock cycle. DPSB1 takes words 48-303 as its inputs and computes hypothesis 48-95. DPSB2 takes words 96-351 as its inputs and computes the results for hypotheses 96-143. DPSB3 takes words 144-399 as its inputs and computes the results for hypotheses 144-191. Thus, inside 48 clock cycles the 192 hypotheses for a source component are computed. Inside (4×48) 192 clock cycles, all hypotheses for two sources both full-chip and half-chip components are computed. The terms full-chip and half-chip are derived from the fact

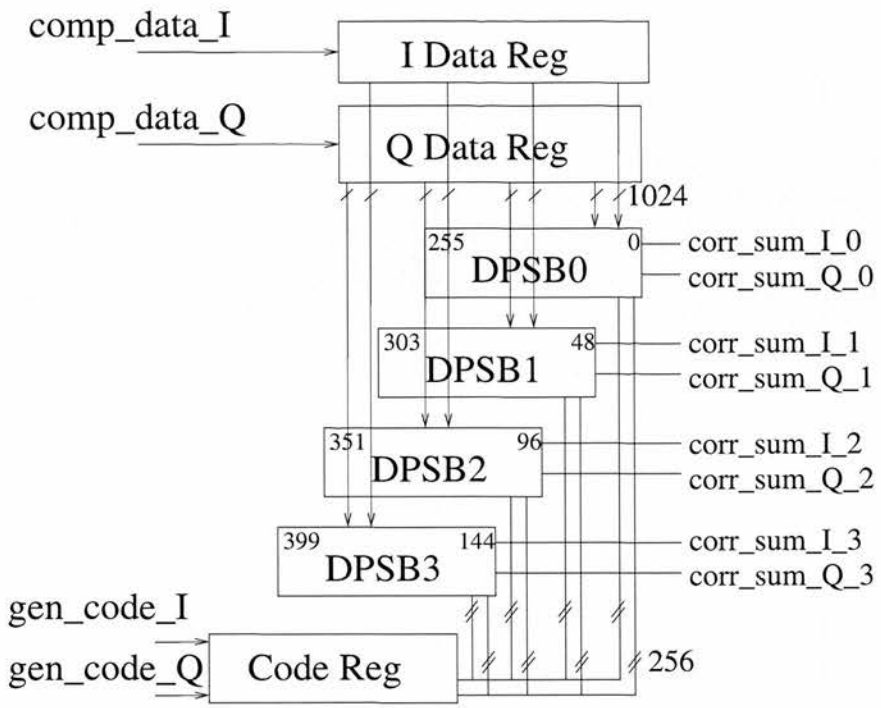

Figure 5.2: CCMF Top Level Structure

that the sampling frequency is twice the chip-rate.

The correlation computation carried out at each of the 256 positions in the DPSB is shown in Figure 5.3 - a 10 bit input produces a 10 bit output. The computation performed is to multiply the complex conjugate of the locally generated code by the complex data sample. Code value 0 maps to -1 and code value 1 maps to 1, so the (I,Q) code may take one of four values (1+j), (1-j), (-1+j) or (-1-j). The result is that the four multiply blocks in Figure 5.3 output either the input or the negative of the input. The output of each cell is fed into a pair of adder trees and coherently summed across a DPSB - one tree is for the real part and one tree is for the imaginary part.

5.4.11 Static Design

To benchmark the reconfigurable CCMF designs proposed later in this section, an estimate of the resource requirements for the static implementation on an FPGA is

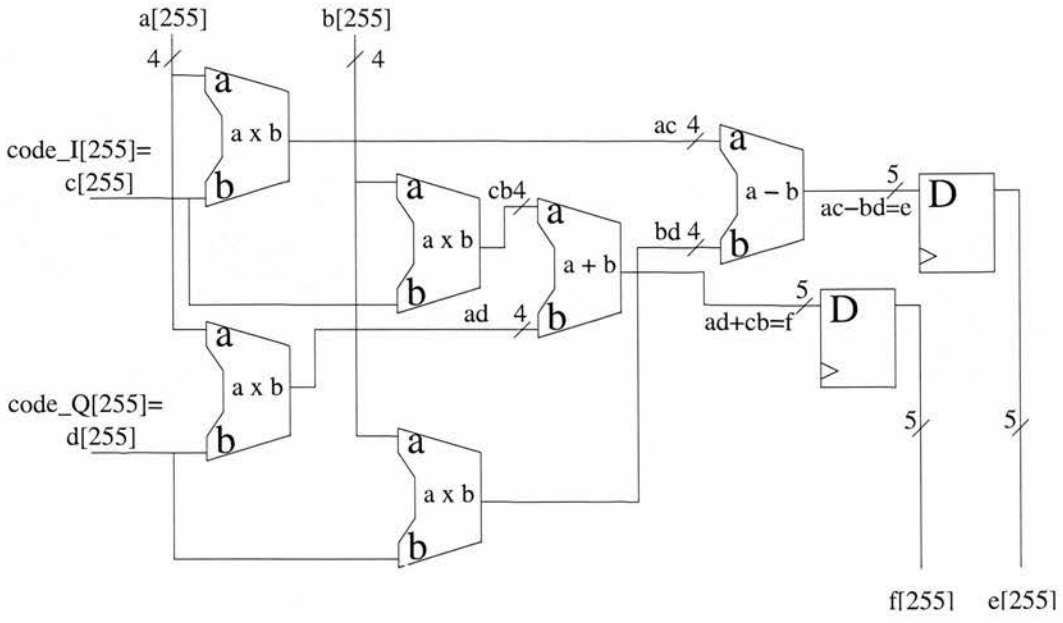

Figure 5.3: CCMF Dot Product Sub Block Cell

made here.

Equation 5.6 shows the cost of the CCMF. Each DPSB is constructed from 256 identical cells drawn in Figure 5.3. There are four 4×1 bit multipliers, a 4+4 bit adder and a 4-4 bit subtracter. The multipliers output either the input or the negative of the input. To compute the twos complement (negative) of the four bit I or Q part requires 4 LUTs. With the negative of a and b available, the four multipliers simply become multiplexers selecting between the respective input unchanged or the negative of the respective input. Figure 5.4 illustrates the CCMF cell with the multipliers replaced by 2's complement units and multiplexers. For each bit the multiplexer must select between two values, so a LUT is required per bit - 4 LUTs per multiplexer. The adder and subtracter units may be implemented with 5 LUTs. Therefore, the qualitatively described DPSB cell requires a total of 32 LUTs, with the synthesised VHDL description requiring 30 LUTs and 40 FFs (flip/flops).

$$C_S = C_{DPSB} \times 4 \tag{5.6}$$

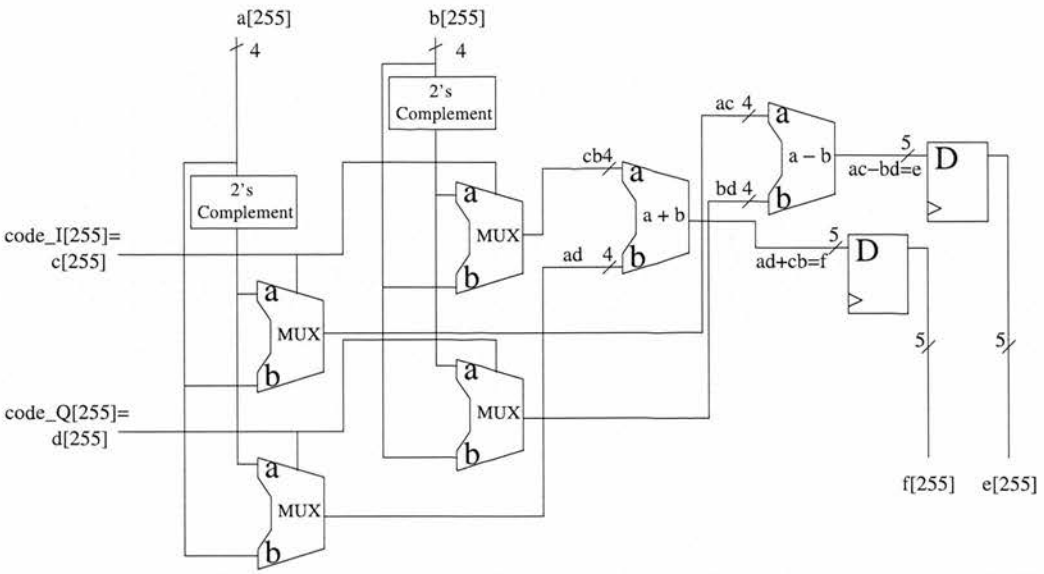

Figure 5.4: CCMF Cell with multipliers replaced with negation units and multiplexers

$$C_S = (256 \times (30,40)) \times 4 = (30720LUT, 40960FF) \tag{5.7}$$

5.4.11.1 Dynamic Design: Code Fold

The CCMF is constructed from many identical cells, tiled with their interfaces connected together to form the complete filter. If it is possible to simplify the implementation of the cell, such that it requires fewer resource requirements, then the saving is multiplied across the complete design.

The VHDL interface for the tiled generator unit is shown in Figure 5.5. 8 data bits and 2 code bits feed the generator every clock cycle, producing 10 result bits, split equally between the I and Q parts. Figure 5.6 shows the top-level structural description of the generator. As described in Section 5.4.10, the code remains constant for 192 cycles. Therefore, the opportunity exists to replace the generator unit with a cell specialised to a particular code. When the code changes, the cell must be changed by populating the SRL16s with the correct value. A black box diagram of the code "folded" unit is shown in Figure 5.7.

```
entity generator is
port(
Clock   : in  std_logic;
InDataI : in signed (3 downto 0);
InDataQ : in signed (3 downto 0);
InCodeI : in std_logic;
InCodeQ : in std_logic;
OutResultI : out generatorcelloutput; -- 5 bits
OutResultQ : out generatorcelloutput  -- 5 bits
);
end generator;
```

Figure 5.5: The VHDL interface description for the generator unit

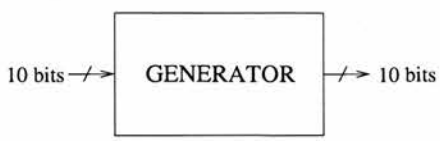

Figure 5.6: Generator Unit Black Box

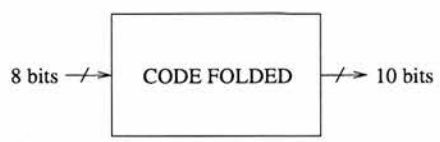

Figure 5.7: CF Unit Black Box

Code	LUT2	LUT3	LUT4	Total
00	12	2	4	18
01	12	2	4	18
10	11	1	2	14
11	12	2	4	18

Table 5.10: LUT requirement breakdown for the CF cell

The LUT requirement of the code folded (CF) unit is shown in Table 5.10. LUT2, LUT3 and LUT4 refer to the number of inputs used on a LUT, so for example, if only 3 LUT inputs are connected to signals, the LUT is referred to as a LUT3. We observe that three of the code values require the same mixture of 18 LUTs with one requiring 14. Before proposing a CF cell to replace the generator unit, we note the following:

1. Since a LUT2 uses 8-bits of the 16-bit LUT, only LUT2s may have new content shifted in while they are in operation (input adjusted accordingly). LUT3s and LUT4s would overflow.

2. The LUT3s and LUT4s must be replicated to allow reconfiguration to occur in parallel to operation. Otherwise cycles would have to be allocated just for reconfiguration. Replication requires an additional 6 LUTs.

3. Although no code folded unit requires more than 18 LUTs, the routing between the LUTs may be different. Routing cannot be changed in-situ (see Section 4.3), so the most flexible LUT arrangement to produce 10 outputs from 8 inputs may be required.

4. Configuration selection logic and counters/adders to adjust the LUT input as a new configuration is shifted in require additional resources.

If we ignore point 3 above, the code folded cell requires 18+6=24 LUTs. Equation 5.8 describes the total resource requirement for the CF design. The total LUT requirement is 24,576 plus the resources required to implement the cold-folded control (CFC) circuitry outlined in point 4 above.

$$C_{CF} = (24 + C_{CFC}) \cdot 4 \cdot 256 \tag{5.8}$$

5.4.11.2 Dynamic Design: Data Fold

The other candidate for folding into the generator unit as described in Figure 5.5 is the data. The black box diagram of the resulting cell is drawn in Figure 5.8. With the 8 data inputs folded in at design-time, the functionality of the cell is substantially reduced.

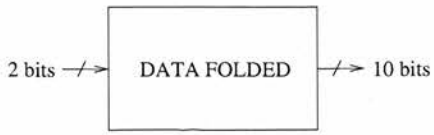

Figure 5.8: DF Unit Black Box

We note the following before describing the data-folded (DF) design's resource requirements:

1. With only 2-inputs per cell, the 10 outputs can be produced using 10 LUT2s.
2. A LUT2 may have its new configuration loaded in parallel to operation.
3. The LUT2 input must be adjusted as the new configuration is shifted in, but unlike the CF design it is the same stimulus for every LUT in a cell.
4. Folding the 8-bit data input into the unit produces a maximum of 256 different cell configurations. These configurations are best produced on demand by a full generator unit shared by multiple cells.
5. The data input changes every 48 clock cycles, so reconfiguration must occur inside 48 cycles.

The basic cell size for the DF design is only 10 LUTs. The new configuration data must be generated, which requires some detailed design work to determine how many generators are required. With 48 cycles to reconfigure and 4 cycles to compute the result of the four code values on each data item, a generator unit can be shared by 12 data items. So to calculate the new configuration for all 447 data items requires a minimum of 38 generator units.

Equation 5.9 describes the estimated cost of the DF design. 11,380 LUTs is the basic cost, with the data-folded control (DFC) circuitry (point 3) in addition.

$$C_{DF} = (10 + C_{DFC}) \cdot 4 \cdot 256 + 38 \cdot 30 \tag{5.9}$$

5.4.11.3 Data Fold Implementation and Result

We select the DF design for implementation since it better simplifies the basic cell. In this section we present a detailed description of the DF design's implementation. Timing diagrams for the cell, cell clustering around the generator unit to form larger components, and implementation results are presented.

The data-folded cell primarily consists of the 10 SRL16 enabled LUTs and the adder to adjust the code input. A little extra control circuitry produces the chip enable (CE) signal for shifting-in the new configuration at the correct 4-cycle window. In a cluster of cells which share the same generator unit, the CE signal can be produced using a single SRL16 LUT acting as a 4 bit shift-register. When initialised correctly, the output of the first shift register in the chain is binary '1' for the first four clock cycles and binary '0' for the remaining 44 cycles. The second cell's CE shift register outputs binary '1' for cycles 3-7 and '0' otherwise, and so on. Therefore, when the generator is producing the configuration for a cell, the cell's CE signal is high to shift in the new configuration.

Figure 5.9 shows the reconfiguration sequence for a data-folded cell. The contents of the 10 SRL16 instantiated LUTs are shown at 5 consecutive clock cycles. At CLK_{i-1} only the bottom four locations of the SRL16s are occupied. This is a state unique to power-on-reset. At all other-times, old configuration data that has been shifted out of the bottom row occupies the upper locations. At CLK_{i-1}, the cell's chip-enable signal goes high. This results in the SRL16s loading whatever is on their shift-register input pin into the first memory-bit and shifting the existing content further into the LUT by 1-bit. The input pin of the 10 SRL16s is driven by the 10-bit output of the associated generator unit. CLK_i loads the configuration for code "11", CLK_{i+1} loads the configuration for code "10" and so on. At CLK_{i+3}, the CE signal goes low as reconfiguration is complete. The 4-bit LUT input is adjusted to deal with the shifting memory content. Since the code provides the input to all LUTs in the cell, the small cost of the adjustment circuitry is amortised across the whole cell.

The DF design's high-level structure with the details of cell clustering around generator units is drawn in Figure 5.10. The diagram indicates logical structure rather than physical placement. The principal observations to make about the design are as

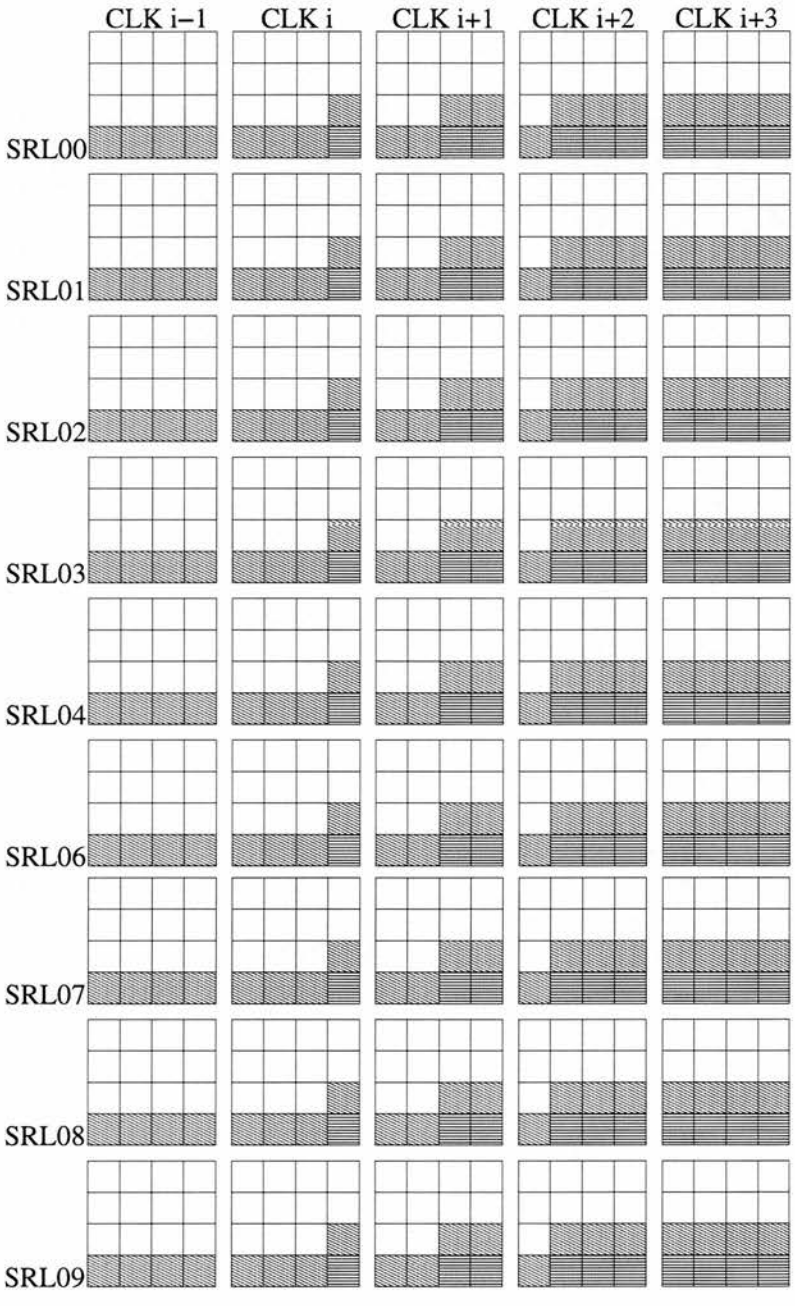

Figure 5.9: Cell Reconfiguration Sequence

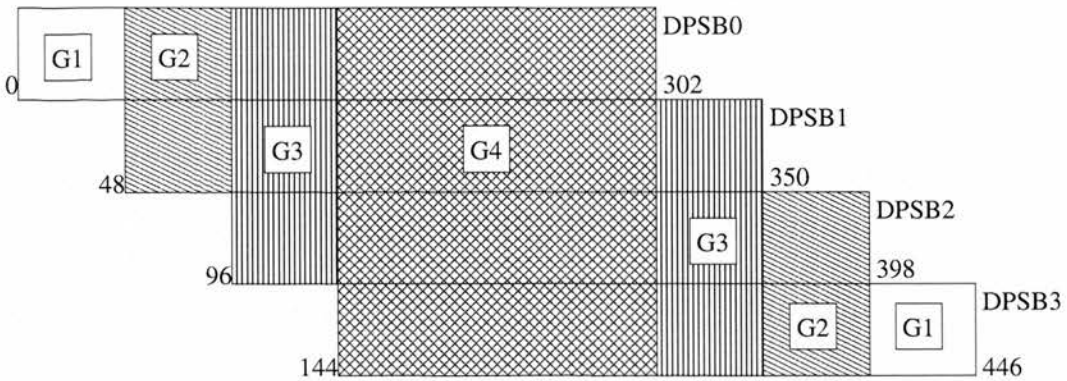

Figure 5.10: Data-folded Implementation

follows:

- The DPSB length is extended from 256 to 303 cells to contain all data items simultaneously.
- The DPSBs are aligned so that the same data items form a column.

With the DPSBs aligned by common data items, the generator units need only perform the minimum number of evaluations. For example, the configuration for data item 144 must be produced for all four DPSBs. A single generator connected to all four cells can supply the configuration. The shaded areas in the picture indicate the cell clustering around a single generator unit. G1 indicates the generator feeds one cell per data item, so inside 48 cycles at 4 cycles per data item, a G1 unit configures 12 cells. A G4 unit configures 4 cells per data item, producing configuration for a total of 48 cells inside the 48 cycle reconfiguration period.

The shaded regions indicate that G4 units construct most of the data-folded CCMF design. The generator cost is maximally amortised in a G4. Table 5.11 lists the resource requirement of each generator cluster, $Gn \mid n \in \{1..4\}$. The SRL16 column shows the number of LUTs instantiated as SRL16 registers. These results, and all other implementation results presented, are produced by the Xilinx Synthesis Tool version 7.1.02i.

Figure 5.11 shows a structural representation of the G4 unit. All 48 cells are fed by a single generator. The cells are arranged into columns of four corresponding to

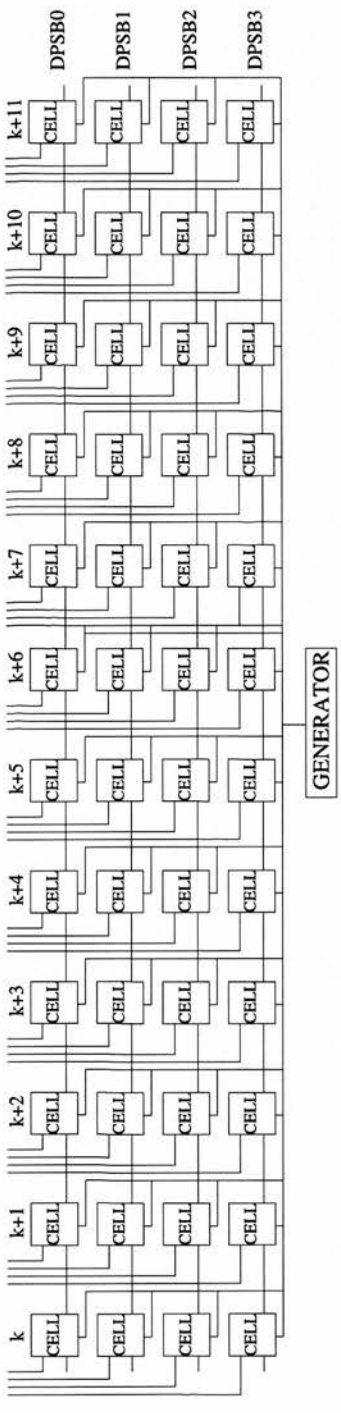

Figure 5.11: Structure of G4 unit

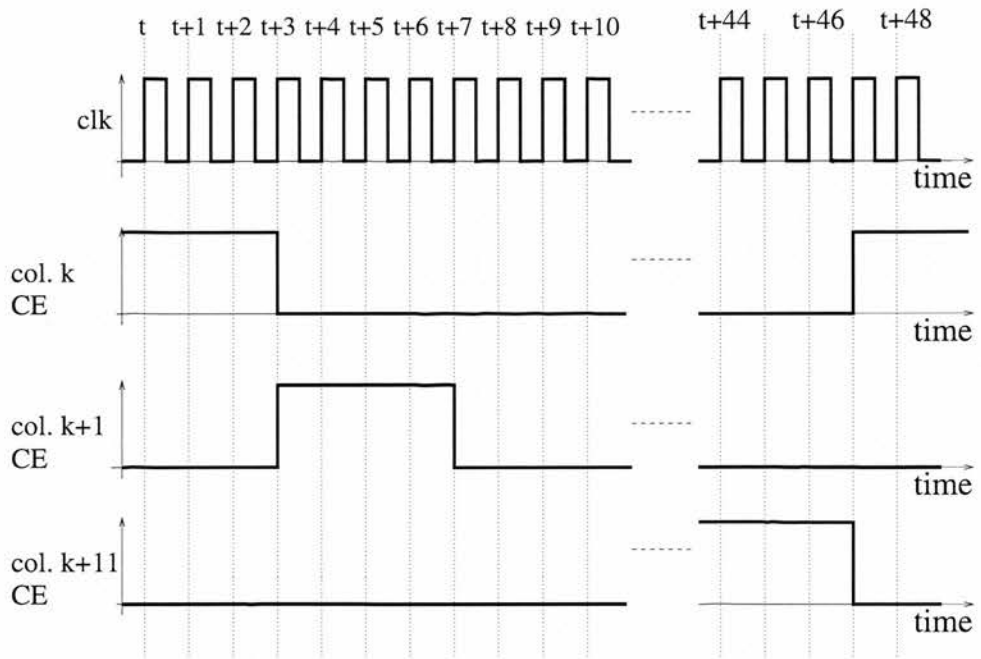

Figure 5.12: G4 unit CE timing diagram

the same data item, with a row forming a section of one of the four DPSBs. There is one SRL16 CE signal for each column of cells (not drawn). In 48 cycles the generator produces a new configuration for all 12 columns. The code input to each cell is shifted along the row, each shift corresponding to one hypothesis. As described in Section 5.4.10, every 192 clock cycles a new code is loaded. This must be done in a single clock cycle, so each cell also has a parallel-code input (not drawn). The output of a hypothesis is drawn feeding towards the top of the diagram in Figure 5.11.

The timing of the G4 unit is described pictorially in Figure 5.12. The column SRL16 chip enable signal is focused on since it is the most interesting signal. The top graph is the system clock, the rising edge of which is numbered. The other three graphs show the CE signal for columns k, k+1 and k+11 in the G4 unit. Note that a columns CE signal is high for four cycles and low for the remaining cycles in the 48-cycle reconfiguration period. The CE signal ripples through the columns consecutively. After 48 clocks, all 12 rows are reconfigured and the process repeats, as can be seen on the right of the graph for column k.

Gn Unit	SRL16	LUT	FF
G1	132	217	252
G2	264	397	457
G3	396	577	660
G4	528	757	864

Table 5.11: Generator Unit Costs: *Gn* is 1 Generator with 12n cells

Data Item Range	Construction	LUT	FF
0-47	4G1	868	1008
48-95	4G2	1588	1828
96-143	4G3	2308	2640
144-302	13G4+3G1	9931	11352
303-350	4G3	2308	2640
351-398	4G2	1588	1828
399-446	4G1	868	1008

Table 5.12: Construction of CCMF from *Gn* units

Table 5.12 gives the details of the shaded regions in Figure 5.10. Note that data-item range 144-302 cannot be served by an integer number of G4 units. The remainder of 3 columns would be served by a special unit of one generator and 12 cells, which we (very conservatively) approximate here as 3G1 units. It can be observed the greatest number of resources go to the G4 constructed section of the design.

The total LUT and FF requirements for the static design and the DF dynamically reconfigurable design are shown in Table 5.13. The last two columns show the implementation speed of the slowest building blocks of the respective designs on the highest speed grade Xilinx Virtex4 and Virtex2. The DF design represents a 37% and 56% saving in LUTs and FFs respectively over the static equivalent.

It is worth noting that the DF implementation introduces an additional latency of 48 cycles when compared to the static design. This is necessary to populate the cells with their first configuration after a power-on reset. We do not believe this poses a problem

Design	LUT	FF	XC4vlx25-12	XC2v250-6
Static	30720	40960	421MHz	323MHz
Data-Folded	19459	22304	421MHz	323MHz

Table 5.13: Results of reconfigurable CCMF implementation

as the next stage in the Searcher (post-detection integration) must wait 192 cycles for all 4-source components anyway. The DF implementation therefore increases latency by at most 25%. This could be reduced by introducing more generator units. For example, doubling the number of generators at a cost of 1140 LUTs would half the additional latency.

The FPGA implementations easily meet the 184.32MHz requirement of the UMTSSOC system clock. In fact, it may be possible to operate the CCMF implementation at twice the system clock frequency and use only half the resources.

5.4.11.4 Discussion

In this section we consider a couple of design possibilities not discussed above. First we consider the CF design if the assumption that 18 LUTs can implement all cells does not hold. Then we consider using a statistical reconfigurable design technique.

If we assume the four CF cell configurations require routing changes then the CF design presented in Section 5.4.11.1 is invalid. Instead, to exploit holding the code constant would involve replacing the CCMF cell illustrated in Figure 5.4 with a set of cells, one of which is selected according to the code to be implemented at each location of the DPSB. As described in Section 5.4.11.1 and table 5.10, the code may assume one of four values (1,1), (1,-1), (-1,1) or (-1,-1). The minimal implementation of these functions has different LUT requirements. For example, since a code component value '-1' demands a two's complement unit whereas a code component value '1' corresponds to no function (pass through).

There are two problems with the practical realisation of these logic resource gains. First, the construction of a DPSB involves selecting the correct cell at each position and hooking its interface up to neighbouring cells, so a limited amount of runtime place-

ment and routing is required. This is non-trivial and requires much more silicon area to perform than can be saved. The problem is compounded by the fact that the changes require access to the fabric's configuration memory and the change is dependent on the local circuit's state. Configuration caching techniques have limited applicability due to the dependence on local state, so massive configuration memory bandwidth is required to perform the reconfiguration fast enough. The second problem with a cell's configuration being dependent on the code is that the footprint of the DPSB will change according to the code it implements. This is a problem because the resource requirements of the CCMF must be bounded in a system context to determine how many resources must be reserved. If the code can assume any value then the worst-case cell size must be considered possible at all 256-locations of a DPSB. To attempt to match the varying resource requirements of the CCMF with another circuit in an "hour glass" type sharing arrangement such that the sum of their resource requirements is a constant would seem both very difficult and highly unlikely to succeed. Therefore, it is the upper bound (or worst case) resource requirements of the CCMF that determine the benefit of reconfiguration. If the code generation function were such that it could be guaranteed to only assume a subset of all possible values then it may be possible to attain greater resource savings with reconfiguration.

Statistical techniques could provide a solution to both the configuration memory access problem and the circuit footprint bounding problem. Given a statistical performance specification for the CCMF, it would be possible to implement the DPSBs with a fixed construction by splitting the cell type configuration evenly between the four versions. A code which does not split evenly between the four cell types could have its adder tree outputs adjusted (perhaps scaled) appropriately. Data values would then be ordered so that they are presented to the correct code unit at each correlation step. Consideration of a statistical implementation would involve careful analysis of the consequences on performance at an algorithmic level. For the CCMF, dangerous possibilities that must be considered are the performance penalty at the RAKE for the occasional poor search. Poor searches could result from erroneous adjustments of the adder tree outputs or a code that demands an implementation very unevenly split between the four cell types.

5.4.11.5 Summary

The reconfigurable CCMF design offers substantial resource savings over the static equivalent. The in-situ reconfiguration means the design can slot in as a direct replacement, requiring no special treatment at the system-level.

5.4.12 Step 9: Advance implementation

With the primary instances defined, and candidates for intra-subsystem reconfiguration identified, implementation can progress.

5.4.13 Step 10: Generate checkpoint control unit

The FSM controlling which configuration should be resident is straightforward. The decision variables are its inputs and it produces the checkpoint enable signal and the hardware configuration number to load. An example of its operation is a highly loaded low-rate voice profile. Several voice calls complete simultaneously (handover of a passing vehicle to another basestation). A medium data-rate call is requested, and since there is enough air-interface capacity, the call can be supported. This is evaluated by the FSM by checking which hardware configuration is able to support the desired service profile, and if it is different to the resident configuration, the match is loaded at the next checkpoint.

5.5 Xilinx Virtex Configuration architecture to support UMTSCE implementation using Checkpoint Framework

5.5.1 Overview

This section presents a proposed new configuration architecture for the Xilinx Virtex FPGA using the banded compression algorithm described in Chapter 3. The architecture is one example in a family of architectures trading off time and space. They are

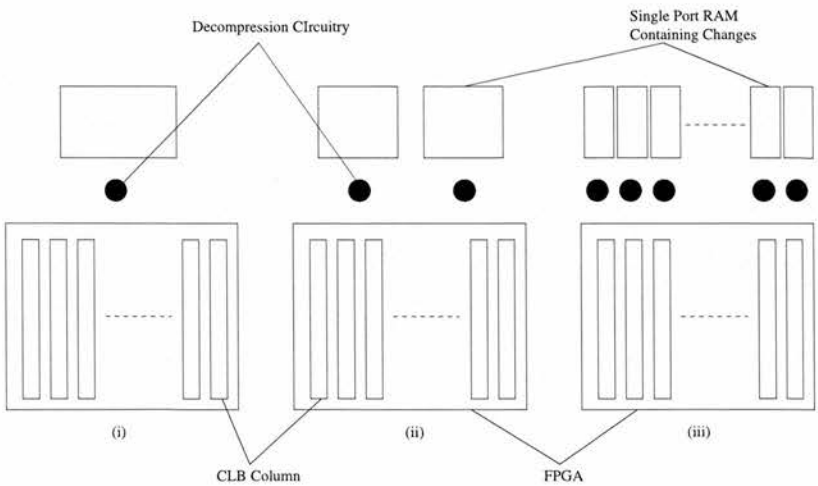

Figure 5.13: Configuration changes on-chip RAM arrangements

composed of on-chip RAM to store configuration changes and decompression control unit(s) to read-back the existing configuration memory contents and decompress and apply them. The specific architecture explored in this section is aimed at providing the speed of reconfiguration necessary for the reconfigurable UMTSCE. The various configuration memory arrangements are then discussed and a decompression unit design given. The silicon area requirements of the configuration architecture are then estimated.

5.5.2 Requirements

The UMTS channel element is chosen as a case study in this thesis and it is used here to examine the implementation of a changes decompression based reconfiguration architecture. It is established in section 5.4.3 that the radio frame time of 10ms in UMTS is a good candidate period for reconfiguration to occur. It is thought to be acceptable to spend 1% of time reconfiguring, so 100μs is the required reconfiguration time.

5.5.3 Change Memory Arrangements

It is well understood that a large memory is more efficient than several smaller memories that provide the same storage capacity. So, a system able to tolerate a long reconfiguration period can use a single large configuration changes memory, but a system that demands a very short reconfiguration period requires the parallelism offered by multiple small configuration memories. Various on-chip memory arrangements are possible to provide the required reconfiguration speed with minimal area as illustrated in Figure 5.13. Each of the sub-figures shows an on-chip changes RAM arrangement, the black circles represent the control and decompression circuitry and the tall rectangles represent a column of CLBs. In Figure 5.13(i) the configuration changes subsystem requires the smallest amount of silicon area but is capable only of performing reconfiguration serially. Figure 5.13(ii) has doubled the reconfiguration bandwidth by splitting the changes RAM into two separate RAMs each with an independent configuration controller. Depending on the application domain, it may be satisfactory to split the columns into two groups of equal size and assign one memory/controller to one group and the second memory/controller to the other group of columns. If it is required to concentrate the available configuration bandwidth to any sub-set of columns in the device, a cross-bar switch access interface would be necessary. Figure 5.13(iii) presents the arrangement where there is a memory/controller combination for every CLB column.

5.5.4 Selected Memory Arrangement

To provide the reconfiguration time of 100μs necessary for the UMTS channel processing engine, a parallel implementation is required. For the Xilinx Virtex 1000 device with 64 CLB rows with 864 bits per CLB, around 87 changes must be made per CLB which sums to 5568 changes in a column. If all changes must be applied within 100us then a cycle time of 17ns is required - or a clock of 58.8MHz. This clock rate should be achievable, which suggests that a full-column parallel configuration changes RAM architecture is able to provide the required speed.

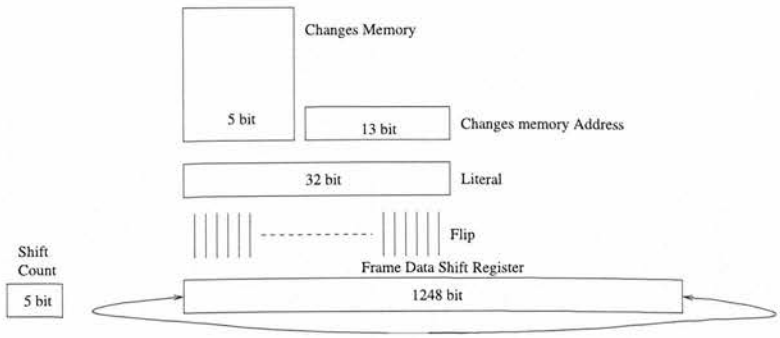

Figure 5.14: Block diagram of circuitry for decompression

5.5.5 Decompression unit design

The decompression unit must make a change to its column's configuration memory every clock cycle. The changes RAM must provide a change bit address and the control circuitry must apply it every clock cycle. A block diagram depicting an example of such a sub-system is shown in Figure 5.14. With an average of 86 changes per CLB and 64 CLBs in a column, 5504 memory locations are required. Using the banded compression algorithm, 5 bits was found to be the optimum word size. During decompression, when the 5-bit relative offset is read from the changes RAM, it is expanded into a 32-bit literal containing 31 zeroes and a single binary 1 where the change is to be made. The literal is then combined with the existing frame contents to flip the appropriate bit, indicated by the lines marked with the word Flip. Control circuitry keeps track of the 32-bit frame content currently being modified and performs a circular shift as appropriate to ensure that the correct part of the frame is exposed to the literal. When all changes have been applied - i.e. the next change bit is in the next frame, the present frame data register is written back to the configuration memory and the next frame read in.

5.5.6 Implementation

The 1.8V VirtexE manufactured in 180nm technology has a CLB area of 34,692μm^2. For example, the XCV1000 has 64 CLBs per column, which equates to 2.22mm^2 per column. An SRAM cell in 180nm is about 4μm^2, so the additional configuration

memory per column of 27,520 bits is about $0.11mm^2$. This equates to under 5% of a column's area. The decompression circuitry is estimated to have negligible size in comparison to the configuration changes memory size. As an alternative to having fixed blocks, the CLB fabric itself could be programmed to perform the decompression. Further, it may be feasible to reclaim this fabric area by loading uncompressed configuration data.

With enough off-chip bandwidth it would be possible to store changes using commodity RAM. Off-chip RAM normally forms part of an embedded system, so an increase in its size is all that is required. It is desirable to have only the decompression unit on-chip as it minimises silicon area which may not be used by all applications. This is particularly true for a product such as the Xilinx Virtex device since its domain of application is very wide. As stated earlier, a feature included on-chip must either be critical for an important application area or useful across a large number of applications. The critical problem of placing the RAM off-chip is bandwidth. Additional I/O purely for reconfiguration is not considered for the reasons just listed. Dedicating I/O to the configuration RAM is undesirable because:

- The number of I/O pins has a strong correlation with the cost of chip-packaging and packaging becomes the dominant cost component of small die size chips.
- I/O bandwidth is a limiting resource in some applications making them unlikely to use reconfiguration if it ties up I/O pins.

The checkpoint framework's bandwidth usage is extremely bursty in nature and, in addition, there is a drastic reduction in system I/O requirements (approaching zero) during reconfiguration since most (if not all) subsystems go off-line. Therefore the opportunity exists to share I/O pins between the system and the configuration architecture. This is the method we favour for implementation.

I/O multiplexing is an established technique that can be implemented with negligible impact on performance or increase in silicon area. It has an impact on high-speed printed-circuit board design since traces potentially have increased capacitance and inductance. Probably most troublesome of these parasitics is the increased capacitance caused by a more dense routing in the PCB. Traces routed side by side can form the two

plates of a capacitor and hence a signal from one trace can be coupled onto the other causing interference. This is particularly dangerous in mixed signal (analogue and digital) designs. Another problem is that the additional capacitance increases the time the chip output buffer requires to drive the trace to the correct voltage - though transistor sizing can help mitigate this effect. Adhering to design rules that avoid the capacitive coupling problem becomes more difficult as the density of traces increases. The increased wiring density also increases the opportunity for inductive coupling. Rules for minimising EMI may be more difficult to satisfy with an increase in the number of traces. One notable project looking at I/O multiplexing is the Virtual Wires project at MIT[12].

5.6 Assumptions

Due to the complexity of the system being analysed and the limited resources available to assist perform the case-study, certain assumptions had to be made. This is particularly the case when performing the translation between the ASIC and reconfigurable implementation. In this section, we discuss the assumptions and their likely effect on the results.

A small number of logic blocks in UMTSSOC implement algorithms iteratively. We assume that reconfigurable implementations exist which can reduce the resource requirement of these blocks. In most cases where this assumption applies it seems reasonable, except, perhaps, for the ARM processor in the ESIP. However, a reconfigurable implementation may have a hard wired processor at its disposal, in which case it is not necessary to seek soft-logic alternatives.

The other major assumption is the accuracy of the dynamic instance corner-points selected. We believe they represent dynamic instances which saturate the air interface bandwidth. Other combinations of services will also saturate the bandwidth, but we have confidence in the system engineering knowledge within Lucent to provide the most computationally intensive corner-points. As mentioned earlier, the corner-points selection was based upon extensive system engineering knowledge within the industrial partner. This knowledge provided an alternative route to estimate the resource

savings possible, since the method prescribed by the methodology in step 4 was not possible in this project due to a lack of domain specific expertise. However, the expert knowledge for constraining the application profile set exists within any UMTS algorithms team.

5.7 Results

The cost functions expressed in step 6 describe over 85% of each subsystem in UMTSCE. Using these and the dynamic configuration instances found in step 7, we are able to calculate the cost of each instance for the three static products. Since we are working from an ASIC design, the result will be expressed in terms of logic gates and memory bits. A wholly FPGA based case-study would express the result in logic slices and block RAM.

We give the results in two separate tables, expressing the number of logic gates and memory bits as a percentage of the ASIC solution. Table 5.14 contains the memory requirements and table 5.15 contains the logic requirements. Since the power control memory requirements are very small they are omitted. In the instance column, S, U and R stand for Suburban, Urban and Rural respectively. The number (1-4) concatenated to the letter indicates the corner-point scenario as described in step 5. The dynamic instance requiring the greatest number of resources from each static configuration represents the saving of the checkpoint methodology. Therefore, although Suburban instance S4 achieves excellent resource savings over the ASIC solution, it is instance S1 which determines the size of device required to implement the Suburban solution. It can be observed that corner-point 1 demands the most resources across all basestation products.

Note that some of the ASIC design figures are greater than the maximum of the column, such as the rake memory and encoder/decoder logic. The reason this occurs is that the corner-point service profiles place a unique set of demands upon the partitions. The reconfigurable implementation's hardware configurations are tailored for the unique service profile they satisfy. Since there is only one hardware configuration for the ASIC implementation, it must provide a superset of functionality and cover all

the service corner-points. Therefore, despite the fact that all features of all hardware blocks are never required simultaneously, the ASIC hardware configuration must still implement them simultaneously. For example, in S1, the ESIP encoder/decoder must provide maximum throughput convolutional coding, with no turbo coding. In S3, the ESIP encoder/decoder must provide maximum throughput turbo coding, with no convolutional coding. The reconfigurable implementation of these corner points produces a hardware configuration only capable of performing the coding required. The ASIC implementation must produce a single hardware configuration which can simultaneously perform both convolutional coding and turbo coding at maximum throughput, despite this never being required. Some sharing of resources does exist in the ASIC implementation. For example, the interleavers feeding the coding stage provide data to both the turbo and convolutional coder.

Primary instance $R3$ has very low resource requirements. However, as described in the methodology, it is the instance with maximum resource requirements that determines the implementation size. It may be possible to exploit the low resource requirements of $R3$. For example, the unused reconfigurable fabric may be powered down to reduce energy consumption or be exploited by more complex receiver algorithms with higher performance.

We observe that the dynamic savings are due to the mutually exclusive maximum resource usage within the ESIP unit and RACH preamble detector. Within the ESIP unit, the encoding/decoding of a call is either convolutional or turbo, never both, providing a relationship which can be exploited by the checkpoint methodology. For example, when the air interface is saturated with voice calls, convolutional coding is all that is required - turbo coding logic and memory resources can be freed up. The opposite is true if the air interface is saturated with data traffic. The RACH preamble unit's resource requirement varies with the air-interface load. As it approaches fully loaded, it will be possible to support fewer RACH connections, relieving the requirement for 3 detectors.

Table 5.14: Memory Requirements (in Bits) Normalised with respect to the ASIC Design Total.

Instance	Searcher	RAKE	ESIP	P.D.	TX	Total
$S1$	12.1	36.4	26.2	1.9	3.2	79.8
$S2$	2.4	9.7	26.0	1.9	0.8	40.8
$S3$	0.6	4.7	13.6	1.9	0.4	21.1
$S4$	1.6	5.2	9.8	5.7	0.7	23.1
$U1$	6.0	34.4	26.0	1.9	2.2	70.5
$U2$	1.2	7.7	24.9	1.9	0.6	36.3
$U3$	0.3	2.7	13.6	1.9	0.3	18.8
$U4$	0.8	3.2	9.8	5.7	0.7	20.3
$R1$	16.1	28.3	8.7	1.9	1.2	56.3
$R2$	3.2	10.5	8.7	1.9	0.4	24.7
$R3$	0.8	7.1	4.5	1.9	0.3	14.7
$R4$	6.4	10.5	9.8	5.7	0.7	33.2
ASIC design	16.1	39.4	35.5	5.7	3.2	100.0

5.8 Outcome

Table 5.16 shows the percentage savings of a reconfigurable design over the ASIC design. The total is broken down into the saving due to static reconfiguration and dynamic reconfiguration. We list the logic and memory savings for all three basestation scenarios (Suburban, Urban and Rural). The saving is the difference between the largest hardware configuration in each scenario and the ASIC solution.

These results can be considered in a number of contexts. For example, a single sector carrier solution which covers both suburban and rural configurations would have an estimated logic saving of 20.4% and a memory saving of 20.2%. Similarly, a single sector-carrier solution for just a rural configuration would have an estimated logic saving of 41.3% and memory saving of 43.7%.

From tables 5.14 and 5.15 we see the other configuration instances only require

Table 5.15: Logic Requirements (in Gates) Normalised with respect to the ASIC Design Total.

Instance	Searcher	RAKE	Enc./ Dec.	P.D.	TX Power	Cont.	Total
$S1$	9.0	43.1	11.8	2.9	7.9	4.9	79.6
$S2$	5.5	10.8	11.3	2.9	3.2	1.2	34.9
$S3$	5.5	5.4	12.4	2.9	1.6	0.6	28.3
$S4$	5.5	5.4	4.2	8.8	1.6	0.6	26.0
$U1$	4.8	43.1	11.8	2.9	4.7	4.9	72.2
$U2$	4.8	10.8	11.3	2.9	1.6	1.2	32.6
$U3$	4.8	5.4	12.4	2.9	1.6	0.7	27.6
$U4$	4.8	5.4	4.2	8.8	1.6	0.6	25.3
$R1$	14.2	32.3	4.2	2.9	3.2	1.8	58.7
$R2$	7.2	10.8	4.1	2.9	1.6	0.6	27.2
$R3$	7.2	5.4	5.1	2.9	1.6	0.6	22.9
$R4$	10.7	10.8	4.2	8.8	1.6	0.6	36.7
ASIC design	14.2	43.1	21.1	8.8	7.9	4.9	100.0

between 26% and 62% of their instance 1 implementation resource requirements. This shows that the largest configuration instance dominates resource requirements. Potential therefore exists for saving substantial amounts of static power by switching off unused resources when hardware configurations 2-4 are resident. Alternatively, the spare resources may be used to implement more advanced algorithms, for example, enhanced power control algorithms to increase cell capacity.

Table 5.16: Percentage Logic and Memory Savings Over the ASIC Design

Product Configuration	Static	Dynamic	Total
S (Logic)	5.3	15.2	20.4
U (Logic)	12.6	15.2	27.8
R (Logic)	30.6	10.7	41.3
S (Memory)	7.0	13.1	20.2
U (Memory)	17.2	12.3	29.5
R (Memory)	35.7	8.0	43.7

Chapter 6

Conclusions

6.1 Summary

Chapter 2 argues for the need to approach reconfiguration in real-time systems in a different way from reconfiguration in the acceleration of general purpose systems. Embedded systems are recognised as the dominant application space of computing for the foreseeable future. In this era of computer architecture, it is argued that the Just-In-Time tradeoff space replaces the single goal of maximising general purpose computing performance. We predict that domain specific platforms composed of heterogeneous computing elements will win an increasing proportion of embedded system designs.

In Chapter 3 an FPGA configuration architecture design space is proposed which spans the single shift-register to the multiple memory plane architecture. The architecture is highly parallelisable and scalable. A compression scheme is developed for the column parallel configuration architecture. The compression scheme compares favourably with previous compression based configuration architectures which are not parallelisable.

In Chapter 4 we propose a new framework and approach to real-time reconfigurable design. We motivate the framework's principles by examining previous work in the literature on reconfiguration. Different reconfiguration speeds are compared and contrasted, and equations used to develop an idea of how reconfiguration can be applied most profitably within real-time SoCs. The result is the definition of a soft-

architectural framework to simplify the targeting of heterogeneous platform FPGAs. A system-level view of design is combined with bottom-up circuit creation to exploit static, medium and high frequency reconfiguration. The approach allows the integration of existing intellectual property by being language independent and making use of existing design flows.

We demonstrate the the soft-architectural framework and targeting approach through a UMTS case study. Equations describing parameterised subsystems are derived in Appendix C and used to demonstrate the harnessing of static and medium frequency reconfiguration in Chapter 5. The case study shows that the most easily attained resource savings are at a low reconfiguration frequency. Further data-folding should be applied only with a system-level view to guide the minimisation of resource usage.

6.2 Conclusions

Most ideas in the thesis relate to the design of reconfigurable systems. Whilst the proposed simple conceptual design approach is important and arguably absent from the literature, perhaps one of the greatest design problems to be faced in the era of platform based design is abstraction from the underlying architectures to facilitate portability. This problem is not investigated in the thesis, but is closely related to reconfiguration. As well as helping with design reuse, to design a system which is independent of the underlying architecture would provide the opportunity to leave the selection of device used to implement the system until late in the design. This would allow a closer match to be made between actual requirements and the device chosen.

Instead of an abstraction model, work focused on design for resource saving purposes. However, a saving of an order of magnitude, as may naively be expected given reported applications in the literature, was found to be difficult to achieve, if not unrealistic in typical systems. In an attempt to extract better savings, attention was given to exploiting different versions of a product or system. This proves to be a promising approach, but the need to capture system requirement bounds in terms of equations could present problems. The bounding requirement equations that the subsystems must satisfy may be difficult to extract and their accuracy is important for the correct minimal

implementation of the system. The number of primary instances required for the implementation of a system may in some cases be quite large, perhaps negating the cost saving benefit of reconfiguration. Another possible problem with the checkpoint approach concerns how applicable it is to latency sensitive applications, or systems with latency sensitive subsystems. The 10-100 microsecond checkpoint duration may be critical and insurmountable in some systems, while other systems may be able to exploit the proposed techniques for hiding and dealing with it. These issues need further investigation.

The decision to concentrate the thesis on the reconfigurable platform FPGA was made due to the ready availability of experimental resources and expertise. It may have been more systematic to have performed an initial evaluation of different reconfigurable fabrics and chosen one based upon some criteria. For example, intuitively, coarse grain fabrics are likely to exploit high frequency reconfiguration more profitably. This may mean they offer wider research opportunities, although we believe the principles of the checkpoint methodology are also applicable to coarse grain fabrics. We offer an intuitive explanation for not pursing high frequency reconfiguration of an FPGA but a theoretical explanation for why it is not feasible would be highly desirable in the field of reconfigurable computing. Such an explanation would provide the link between the frequency of reconfiguration and the benefit achievable.

6.3 Future Work

6.3.1 Overview

As with any research, the ideas and results presented here give rise to many more questions. In this section, some fruitful areas for future research are described which relate closely to the work presented. In Section 6.3.2 we suggest ways to improve upon the configuration architecture in this thesis. In Section 6.3.4 we outline some future work to exploit reconfiguration more fully.

6.3.2 Architectural Features to aid Reconfiguration

The basic analysis of what occurs during reconfiguration of an FPGA provides a platform for investigating new configuration architectures. In the thesis we use it to develop a simple but highly parallelisable difference compression based configuration architecture. Here we present a number of research directions for building upon the basic analysis.

The checkpoint soft-architectural framework uses a small number of configurations to implement a system. There is a well defined sequence of instance usage available at design time. It is likely that information can be exploited higher in the synthesis tool chain than at the bitstream generation level. Take for example two primary instances which are known to follow one another during operation. The first instance is completely routed, then the router is asked to route the second circuit with both timing and changes minimisation as constraints. This would also involve more investigation of circuit level effects such as the effects of antennae on routes.

A major result of the basic changes analysis is that over 70% of changes occur in the LUT contents and MUXs feeding the LUT contents. Solutions to efficiently providing high bandwidth connections to these resources and maximising the overlay technique as described in the previous paragraph are likely to produce excellent results.

6.3.3 Off-chip Memory Feasibility Study

Modern platform FPGAs do not satisfy the large memory requirement of the UMTSCE algorithms. Over 80% of die area is devoted to RAM in the UMTSCE. It is likely that future domain specific [46][5] reconfigurable logic will offer a ratio of RAM to logic more suited to that demanded by DS-CDMA baseband processing. Alternatively, new technologies and hardware techniques to improve memory bandwidth are emerging such as on-chip optical interconnect [181][116] and selective exposure fabrication [178].

In this section, the mismatch between the UMTSCE RAM to logic ratio and the ratio offered by platform FPGAs is explored. The largest RAM blocks have their access pattern and peak throughput demands profiled in order to determine whether they

can be moved off-chip. If most of the large RAM blocks can be placed off-chip then it would be reasonable to assume that an FPGA targeted design could be implemented efficiently. The short study presented here is a preliminary feasibility assessment for future work. Future work would involve using the checkpoint methodology to implement the UMTSCE on an FPGA supplemented by off-chip RAM.

6.3.3.1 High Bandwidth Memory Interfaces

Today's commodity Dynamic RAM (DRAM) requirements are driven primarily by the demands of microprocessor architectures. Improved capacity, throughput, latency and cost are all important. The domain of network processing has a different set of requirements and often uses Static RAM (SRAM) for its improved latency over DRAM. In this section DRAM is considered for off-chip storage as it is significantly cheaper than SRAM and offers high performance with careful design. Also, as will be discussed later, latency hiding techniques can be employed for many algorithms, so latency is not of prime importance.

Double Data Rate (DDR) SDRAM is currently the mainstream commodity memory standard, with (evolutionary) DDR2 poised to take over in the near future. The DDR standard was approved by the Joint Electron Device Engineering Council (JEDEC) in 1998. It is an enhancement to the previous PC-100 standard RAM, supporting data transfers on both edges of the clock, effectively doubling throughput.

Read and write accesses to DDR RAM are burst oriented with accesses starting at a specified location and continuing for a programmed number of locations. The "ACTIVE" command begins an access, followed by a "READ" or "WRITE" command. The address bits registered with the ACTIVE command are used to select the memory bank and row to be used. The address bits registered with the "READ" or "WRITE" command are used to select the bank and the starting column location for the burst access. Burst lengths are 2, 4 or 8 locations. An "AUTO PRECHARGE" function may be enabled to provide a self-timed row pre-charge that is initiated at the end of the burst access. The pipelined, multi-banked architecture of DDR SDRAM enables concurrent operation, therefore hiding row pre-charge and activation time.

The Dual In-line Memory Module (DIMM) packaging of DDR SDRAM is offered

with clock speeds of 100MHz through to 200MHz with a 64-bit data bus enabling peak throughputs of up to 23.8 Gbps. Achieving peak throughput is very much dependent on how memory is partitioned across the banks and the design of the memory interface.

The resource requirements of a DDR interface are low. A Virtex 2 100MHz interface requires only 645 slices and 110 Input Output Blocks [67].

6.3.3.2 Preliminary Bandwidth and Access Pattern Investigation

It is only possible to pre-fetch successfully if the address to be read from is known in advance, hence the importance of determining the memory access pattern of the algorithm using the memory. For example, if an algorithm has a regular, perhaps even statically defined access pattern, then pre-fetching data is straightforward. With a less regular but still predictable access pattern, pre-fetching is possible but the address generation complexity increases. It is not possible to perform pre-fetching if the access pattern is run-time dependent and addresses are generated as they are required. The final practical consideration is the match between the memory word-size and the data word size. If many more bits are read than are required then bandwidth is wasted unless the adjoining bits can be efficiently stored on-chip and used later. For on-chip storage of additional bits to be efficient they must be processed quickly, otherwise the advantage of using off-chip storage reduces. With DDR memory, the minimum burst size of 2 words inherently affects the match between what must be read from memory and what is required.

Table 6.1 lists the largest memory blocks in the UMTSCE with their total memory throughput (read and write throughput summed). The blank entries are due to the data being difficult to extract from the design documents made available. The memory blocks listed in the table represent 70% of the total UMTSCE memory usage. The throughput figures are the raw requirements calculated by studying how the memory content is used by the logic. They do not consider inefficiencies introduced by the memory interface word width. The column labelled sequential indicates whether the memory access pattern is sequential or not.

About half the memory is sequentially accessed, meaning that very efficient, long burst length accesses with the potential to reach peak DDR throughput are possible.

Partition	Memory	Size	BW (Mbit/s)	Sequential	On/Off Chip
RAKE	Channel Est. Circ. Buff.	2,048,800	42	Y	Off
ESIP	1st De-Interleaver	1,966,080	57	N	Off
Searcher	Coherent PDI	1,081,344	32,440	Y	On
ESIP	1st Interleaver	1,048,576	23	N	Off
Searcher	Non-Coherent PDI	933,888	4,790	Y	Off
RAKE	Decimated FFT	851,968	130	Y	Off
ESIP	Config/Result	655,360	1,228	N	Off
ESIP	Instruction TCM	524,288	983	N	Off
RAKE	Sample Rate Buffer	491,520	1,843	Y	Off
ESIP	2nd De-Interleaver	491,520	1,843	N	Off
PD	Shift Register	294,912	553	Y	Off
ESIP	Data TCM	262,144	NA	NA	NA
TX	Input Buffer	262,144	NA	NA	NA
PD	Deskew	172,032	11,800	N	Off

Table 6.1: Largest UMTSCE memory block throughputs

The total throughput demands of the memories suitable for sequential access are reasonable, with the exception of the Searcher Coherent PDI memory. Due to its extremely high throughput demand, the searcher memory must remain on-chip.

With the exception of the PD Deskew memory and possibly the ESIP instruction TCM (tightly coupled memory), all non-sequentially accessed memories can also be placed off-chip. All access, although not sequential, is pseudo random, so latency can be hidden by pre-fetching data. With a memory word length of 64 bits and actual word lengths as low as 16 bits, the available memory throughput is not used efficiently, but since the raw throughput is low, the inefficiency can be accommodated.

The PD de-skew memory throughput cannot be accommodated with off-chip RAM because, due to massive bandwidth inefficiencies, the required bandwidth exceeds that of DDR at its peak. The read efficiency is very poor because random 8-bit data words are required. The effective throughput to get the required 8 words per clock cycle with

64-bit word-length DDR RAM is $8 \cdot 64 \cdot$ Clock Rate $= 94,372$Mbits/s. This, combined with further inefficiencies such as the minimum burst length of 2 words, mean the Preamble Detector Deskew memory must stay on-chip.

6.3.3.3 Summary

Carefully planned use of off-chip RAM can achieve reasonable operational throughput. This study estimates that all but 2 of the largest RAMs in the UMTSCE can be placed off chip without saturating the bandwidth available with commodity DDR RAM. This result suggests the on-chip memory requirement for the CE can be reduced to a level at which the ratio of logic to memory is similar to that offered by present platform FPGA architectures.

6.3.4 Design with Reconfiguration

The checkpoint framework provides a generic way to maximise a platform's programmability. In the larger picture of design automation, it would be useful to investigate ways of integrating it into hardware-software co-design. The guiding idea of primary instances of functionality may be extended to hardware-software partitioning. As the functionality required of the system changes with time, it is probable that the partitioning across heterogeneous fabrics will change - for example to minimise power consumption.

Perhaps the most obvious area for future work is to look at other applications and application domains in an attempt to characterise better those systems where the approach is of most benefit. A method for automatic examination of static netlists and determination of their potential to exploit reconfiguration would be extremely valuable. The combination of read-back on every clock cycle to measure circuit activity together with higher-level observations would be an interesting line of research. The insertion of additional monitoring circuitry could help reduce the amount of read-back necessary.

Another area not investigated here concerns the automated solving of the resource requirements equations. There are likely to be techniques in the area of mathematical

optimisation which could be applied here. It is noted that step functions may exist, so the volume of optimisation may not be continuous.

Finally, rapid reconfiguration represents only a small proportion of the improvements reported here. This is likely to be due to the difficulty in identifying good candidates. It would be extremely useful to develop techniques for the automatic identification of candidates for high frequency reconfiguration (LUT contents only). A quantitative approach would be to construct test circuitry to monitor state changes during a circuit's operation. Those areas where state change occurs slowly may provide opportunity for datafolding.

Appendix A

Digital Communications

A.1 Introduction

This appendix presents some of the fundamentals of digital mobile communications. As there are many textbooks which cover these topics in detail, a very condensed presentation of the material will be made here - only going into detail on certain topics that are vital to the understanding of work presented in the thesis.

A.2 Digital Modulation

Today's mobile radio systems solely use digital modulation techniques. In communication engineering, passband signals are usually dealt with. The most basic definition of a passband signal is that the Fourier transform S(f) of the signal s(t) satisfies the condition:

$$s(0) = 0. \tag{A.1}$$

which implies that the signal contains no DC component. Although this is sufficient for equivalent baseband representation, in real systems - especially in mobile radio, the signals dealt with are often centred around a 'carrier' frequency which is much higher than the bandwidth of the signal itself.

The passband signal may be represented as follows

$$s(t) = s_I(t)cos(\omega_0 t) - s_Q(t)sin(\omega_0 t). \tag{A.2}$$

The form of A.2 is referred to as the quadrature component or I/Q representation, where $s_I(t)$ and $s_Q(t)$ are the in-phase and quadrature phase component, respectively of $s_L(t)$, the complex valued *equivalent baseband signal.* The complex valued equivalent baseband signal $s_L(t)$ can be derived from the real base-band signal $s(t)$ by introducing the *analytic signal* $s_+(t)$ which contains only the positive frequencies of $s(t)$.

$$S_+(f) = (1 + sign(f))S(f). \tag{A.3}$$

Shifting the spectrum $S_+(f)$ of the analytical signal $s_+(t)$ down to the baseband, we find find the spectrum $S_L(f)$ of the equivalent baseband signal $S_L(t)$.

$$S_L(f) = S_+(f + f_0). \tag{A.4}$$

$$s_L(t) = s_+(t)e^{-j\omega_0 t} \tag{A.5}$$

Signals and their mathematical representation have been extensively studied [157] [38] [81] [130] [125] [69] [126].

A.3 Power Spectral Density

Random signals are often used in communications engineering to mitigate against interference so that the information gets transmitted. Therefore, it is necessary to model the signals through their statistical properties. The usual signal description in the time and frequency domains cannot be applied. Instead, the autocorrelation function $\phi_s s(\tau)$ of the random signal $s(t)$ is used,

$$\phi(t) = E\{s(t)s(t+\tau)\} \tag{A.6}$$

with $E\{.\}$ denoting the expectation value. We find the representation in the frequency domain if we take the Fourier transform of the autocorrelation function.

$$\Phi(f) = \int_{-\infty}^{\infty} \phi(\tau) e^{\omega\tau} d\tau. \tag{A.7}$$

$\Phi(f)$ is the *power spectral density* or PSD representing the distribution of power as a function of frequency for the stochastic signal $s(t)$. The inverse Fourier transform at $\tau = 0$ is,

$$\phi(0) = E\{s^2(t)\} \tag{A.8}$$

which is the average power of the stochastic signal. A detailed treatment of random signals and stochastic processes is given in [157] [38] [130] [69] [173] [113] [51].

A.4 Constellation Diagram

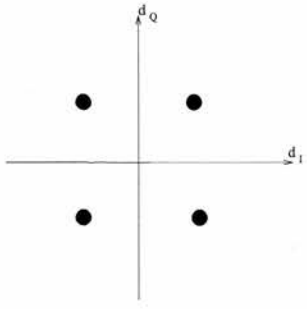

Figure A.1: Constellation for QPSK modulation.

In general, linear modulated communication signals can be expressed with:

$$s_L(t) = \sum_{n=-\infty}^{\infty} d(n) g(t - nT). \tag{A.9}$$

The equivalent baseband signal $s_L(t)$ is the sum of time-shifted impulse responses g(t) of the transmit pulse shaping filter where each is weighted with the corresponding complex data symbol $d(n)$. While $g(t)$ is a deterministic signal (since we know the

impulse response of the transmit filter), the complex data symbols $d(n)$ carry the information and are the outcome of a stochastic process. The constellation diagram is the representation of the data symbols in the complex plane. Figure A.1 shows a simple example where the constellation diagram for quadrature phase shift keying (QPSK) is depicted. All the four possible data symbols have the same amplitude (vector length), but have their value determined by their phase. After demodulation and detection at the receiver, a decision must be made as to which data symbol was transmitted. A diagram where the constellation of the actual signal is drawn, including distortions, gives valuable information about the signal integrity in the various stages of the transmitter and receiver. The constellation diagram thus serves as a qualitative measure of the modulation accuracy of the signal.

Equation A.9 shows how a linear modulated signal may be represented, and a pass-band signal can be represented by

$$s(t) = (\sum_{n=-\infty}^{\infty} d_I(n)g(t-nT))cos(\omega_0 t) - (\sum_{n=-\infty}^{\infty} d_Q(n)g(t-nT))sin(\omega_0 t). \quad \text{(A.10)}$$

The equivalent base-band representation is the sum of the time-shifted impulse responses $g(t)$ of the transmit pulse shaping filter which are weighted with the complex data symbols $d(n) = d_I(n) + d_Q(n)j$. For linear modulation techniques, the amplitude of the transmitted signal varies linearly with the data symbols $d(n)$ [132] and, therefore, the superposition principle with regard to the mapping of the data symbols onto the transmit signal is valid [130]. In general, linear modulation techniques are bandwidth efficient but they suffer from strong amplitude variations in the transmitted signal and hence require highly linear implementations in both the transmitter and receiver.

While $g(t)$ is deterministic (since the impulse response of the transmit filter is known) the complex symbols d(n) which carry the information are the result of a stochastic process. A widely used transmit filter in mobile communications is the *raised cosine* type filter, which has the important properties that it satisfies Nyquist's first criterion and it can be split between the transmitter and receiver to create a matched filter. The resulting widely used filter is the *root raised cosine* filter.

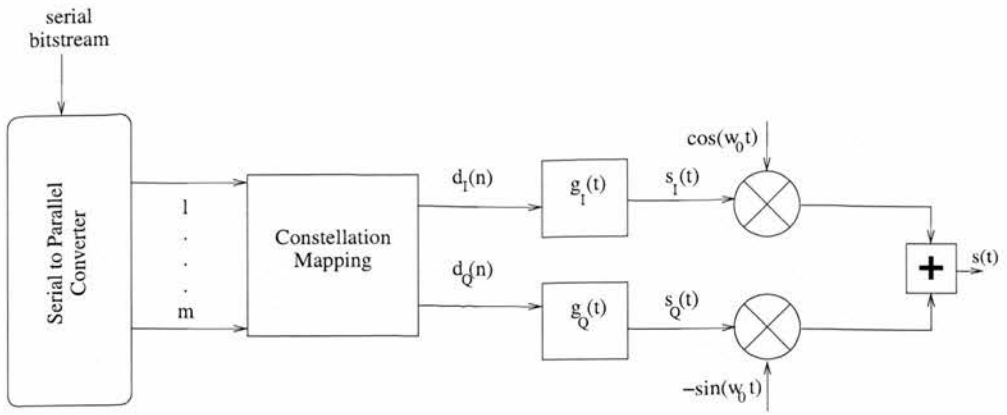

Figure A.2: Generic modulation architecture for digital signals

Depending on the actual modulation format, the serial stream of bits is converted to a stream of m parallel bits. Each m -bit-wide data word is then mapped onto the constellation diagram, giving the real $(d_I(n))$ and imaginary $(d_Q(n))$ part of the data symbol $(d(n))$. The pulse shaping filter sits in both the I and Q paths. The filter outputs are combined with the carrier frequency $(cos(\omega_0 t)$ in the I path and $-sin(-\omega_0 t)$ in the Q path) to translate the baseband signal to the carrier frequency f_0. Finally, the sum of both paths gives the real-valued band-pass signal for transmission. This architecture can be applied to any linear modulation technique and is illustrated in Figure A.2.

A.5 Bandwidth Efficiency

The various modulation techniques have different bandwidth efficiency. A common way of expressing bandwidth efficiency (η_s) which gives the achievable bit rate per Hz is

$$\eta_s = \left(\frac{\text{transmission rate}}{\text{required bandwidth}} \right) \tag{A.11}$$

The units for η_s are bits/s/Hz. The theoretical values for the bandwidth efficiency of different modulation techniques are listed in Table A.5. These values do not include the effect of transmit pulse shaping. The reader is referred to [130] [125] for further treatment of this topic.

Modulation technique	η_s $(bit/s/Hz)$
BPSK	1
QPSK	2
16QPSK	4

Table A.1: Theoretical Bandwidth Efficiency of Several Modulation Techniques

A.6 Cellular Radio Terminology

A.6.1 Duplex Techniques

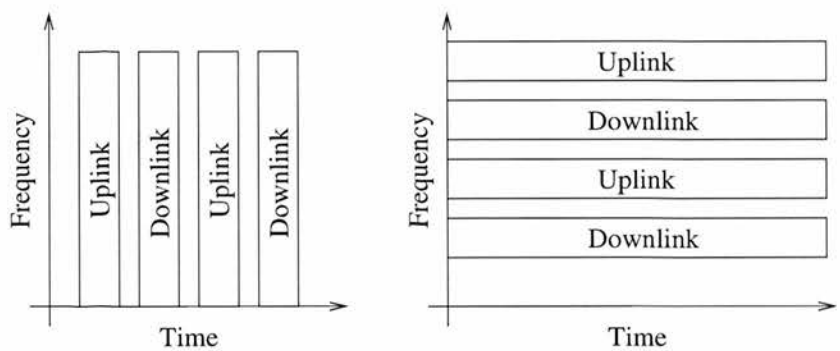

Figure A.3: TDD and FDD techniques for separating the signals of transmit and receive

A duplex technique is necessary if a terminal must both receive and transmit simultaneously. To perform this, the transmit and receive signals must be separated. Frequency Division Duplex (FDD) and Time Division Duplex (TDD) are used and depicted in Figure A.3. With FDD, transmit and receive are separated by frequency, however this is paid for by the requirement of a costly duplexer that ensures the high power transmit signal is radiated only to the antenna and not into the receive path. In TDD, a simple switch is sufficient but accurate timing is required between the terminals and the base station. The second generation standard, Global System Mobile (GSM) was FDD, the new 3G standard, UMTS, includes both TDD and FDD.

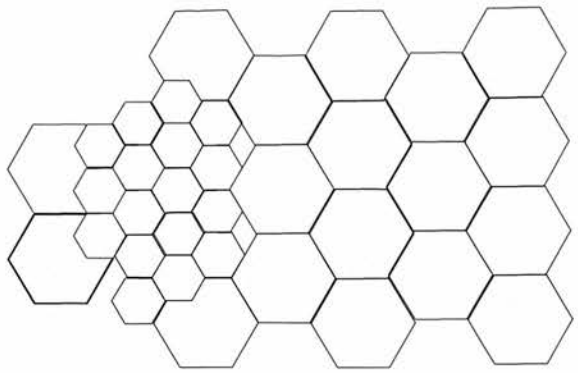

Figure A.4: Hierarchical Cellular Plan

A.6.2 Hierarchical Cell Structure and Frequency Reuse

Cellular systems made the coverage of large geographical areas possible. The subdivision of an area into a sufficient number of cells helps to solve two problems. First, limiting the cell size reduces the transmit power requirements of the terminal and hence increases battery life. Secondly, the size of the cell determines the maximum number of users per unit area. Cells that are too large can result in too low a signal to noise ratio for satisfactory operation. Each cell has a basestation and is assigned a certain frequency band. Usually the frequency band owned by an operator is divided into a set of sub-bands and these sub-bands are assigned to cells. A sub-band used by one cell can be used by another if it is far enough away to limit the interference to tolerable levels. In TDMA based GSM, a frequency reuse factor of 3 or greater is used, but in CDMA based UMTS a frequency reuse factor of just one may be used. Figure A.4 shows an example network with the commonly used hexagonal representation of a cell.

The size of cells in a CDMA based cellular network varies depending on their load. This is because CDMA is interference limited, and as more users enter a cell, the interference level increases, until it is not possible for users at the cell boundary to communicate. This effect of the cell growing and contracting in coverage with the level of interference is referred to as "cell breathing". It is imperative that the network planners take this effect into account.

A.6.3 Multiple Access Schemes

The radio spectrum is a limited resource, so much effort has been spent to find methods to used it most effectively. This includes finding efficient ways of sharing a given bandwidth amongst many different signals for different users - multiple access.

It is possible to separate a given bandwidth into the time domain, frequency domain, code domain or spatial domain. It is also possible to combine two or more of these mechanisms to create hybrid access schemes.

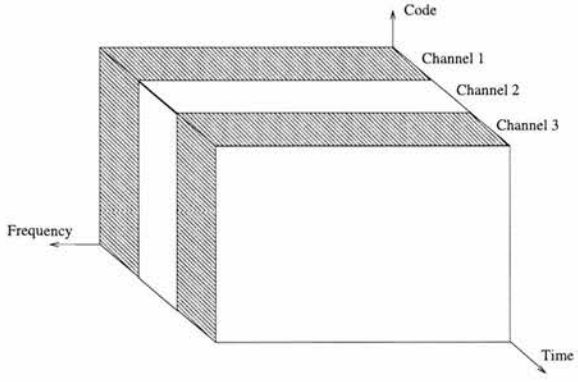

Figure A.5: Diagram illustrating Time Division Multiple Access (TDMA)

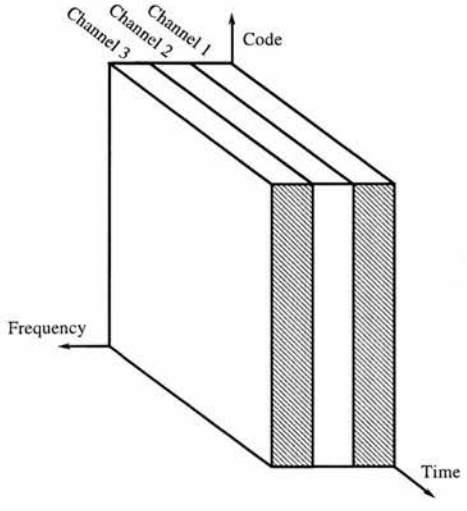

Figure A.6: Diagram illustrating Frequency Division Multiple Access (FDMA)

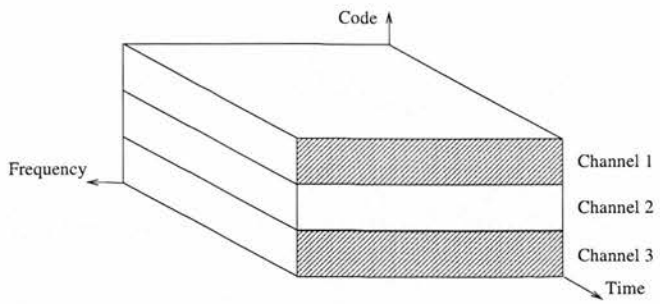

Figure A.7: Diagram illustrating Code Division Multiple Access (CDMA)

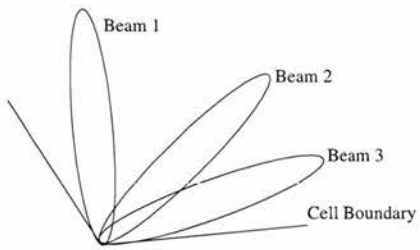

Figure A.8: Diagram illustrating Space Division Multiple Access (SDMA)

- Time Division Multiple Access (TDMA) enables several users to use a single bandwidth by assigning a unique time slot to each user. During the time slot the user has the complete bandwidth. (See Figure A.5)

- Frequency Division Multiple Access (FDMA) shares the available bandwidth by splitting it into sub-bands and assigning those sub-bands to individual users. (See Figure A.6)

- Code Division Multiple Access (CDMA) shares the available bandwidth by assigning each user a code which is used to spread their signal over a wide band carrier signal. (See Figure A.7)

- Space Division Multiple Access (SDMA) can be used if all users within a cell are located at positions with different azimuth angles. The most straight forward approach is to use sectorised antennae, which divide the cell into sectors where different users can only be distinguished in the spatial domain if they are in different sectors. More sophisticated forms of SDMA involve steered antenna

beams. It is widely anticipated that SDMA will be combined with a more basic multiple access system to enhance system capacity. Figure A.8 depicts the use of sectorised antennas.

In theory it does not matter whether the spectrum is divided into frequencies or time slots - the capacity provided from these multiple access schemes is the same. However due to typical radio environment conditions, and the changing requirements of the different traffic flows, CDMA has emerged as the scheme of choice for 3rd generation cellular networks. Refer to [93] for an overview of Cellular CDMA and [8] [105] [150] [175] for early 1970's information-theoretic investigations into the multiple-access communication model.

A.6.4 The Mobile Radio Channel

The characteristics of the environment in which radio communication takes place between the transmitter and the receiver greatly affect the performance of the system. A thorough understanding of the mechanisms in the radio channel, their mathematical model and the received signal characteristics is of major importance to mobile radio communication design. This section outlines the topics important for understanding the material presented in this thesis. For a more detailed description of this large topic please refer to [132] [94] [59] [147] [148] [149].

There are four distinct features of the mobile radio channel which have significant impact on the signal reaching the receiver antenna. The first feature is *multi-path propagation* . The electromagnetic wave originating from the transmitter usually follows several different paths to the receiver antenna. This is due to scattering, reflection and diffraction mechanisms, which create paths other than the direct line of sight path (which in many cases does not exist). Multi-path propagation causes fading, i.e., spatial variation of the received signal power due to constructive and destructive summing of the signal arriving via different paths. A second feature is *time variance* . The propagation channel can vary with time because the mobile terminal can move around or the objects causing scattering, reflection and diffraction move around. The third feature is *Doppler spread* . This captures the spreading of the transmitted signal in the

frequency domain due to the Doppler effect. The fourth feature is the *spatial dispersion* of signals. Since the signals arrive at the receiver from different directions, the radio channel varies with the azimuth angle, i.e., if a directional antenna is used at the receiver, the radio channel exhibits different characteristics as the antenna is pointed in different directions.

A.6.5 Diversity Techniques

A common means to combat signal loss due to fading is through the use of diversity techniques. Basically, if a signal arrives at the receiver via different paths, with each path experiencing independent fading, then carefully combining the received signals will produce a strong signal. This is due to the fact that for many independent paths there is only a small probability that all paths will be in a deep fade at the same time. In fact, if the probability, p, that the SNR of one path is below a certain threshold, then p^N is the probability that the SNR values on all N paths are below the threshold at the same time. Some of the most important diversity techniques (all of which, except frequency diversity, can be used in UMTS) follow:

- Space Diversity: Two antennae are used at the receiver to feed independent signals to the receiver input.
- Frequency Diversity: Frequency diversity is obtained by using several channels at different frequencies. A common type of frequency diversity is Orthogonal Frequency Division Multiplexing (OFDM). Here, several data symbols are transmitted in parallel on many closely spaced carriers. Therefore, the frequency selective fading will degrade some of the carriers but not all of them. Combined with powerful error correcting codes, lost data can be recovered to reduce the bandwidth consumed by retransmissions.
- Time Diversity: Time diversity uses a similar principle to frequency diversity. Data symbols are interleaved, i.e. data is distributed over several time slots. If one slot is lost due to fading, error-coding can recover the data, because only a small fraction of the data block is lost due to interleaving.

- Multi-path Diversity: Different signal components can sometimes be resolved through signal processing - this is called multi-path diversity. This is possible with direct sequence spread spectrum systems, together with a RAKE receiver.

A.6.6 Maximum Ratio Combining

There are several techniques for combining the different signal branches from the diversity techniques above. The most effective method is called Maximum Ratio Combining. Each branch is weighted with its complex conjugate path weight before summing up. Thus it is necessary to know or estimate these path weights and this involves highly complex computations.

A.6.7 Spread Spectrum Techniques

A spread spectrum system is that in which the bandwidth consumed by the signal transmitted is much larger than the baseband signal. This is achieved through the use of a signal independent of the information signal.

Shannon expressed the basis of the spread spectrum technique with the channel capacity formula in [143]:

$$C = B \cdot log_2\left(1 + \frac{S}{N}\right) \tag{A.12}$$

The channel capacity C is the maximum data rate which can in principle be transmitted over an Additive White Gaussian Noise (AWGN) channel without any error, with bandwidth B and signal-to-noise ration S/N. The past 50 years of information theory has managed to get today's technology within less than 1dB of Shannon's theoretical limit [163].

There exists a linear dependency between channel capacity C, and bandwidth B, whereas the capacity only increases logarithmically with the SNR. Further, it is clear that bandwidth can be traded off against SNR.

The most widely used spread spectrum technique, and the method used by UMTS, and hence of importance to this work, is called *Direct Sequence Spread Spectrum*

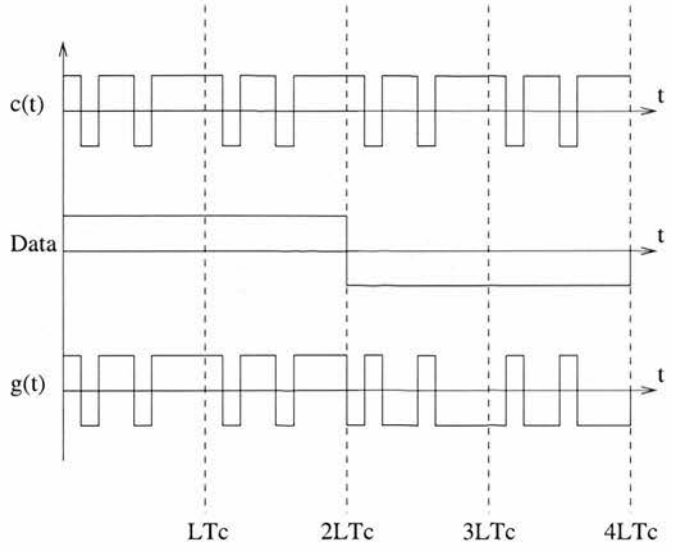

Figure A.9: Direct Sequence Spread Spectrum System signals in time domain

(DSSS) . In DSSS, the signal to be transmitted is modulated with a digital code sequence, which is independent of the data and has a much higher clock rate than the data signal bandwidth.

A.6.8 Direct Sequence Spread Spectrum

In DSSS, the digital data signal is modulated with a code sequence $c(t)$. Figure A.9 illustrates this. The individual bits of the code are referred to as chips to differentiate them from the data signal. After transmission at the carrier frequency, reception and down-conversion to baseband, the spreading must be reversed in the receiver to recover the original baseband data signal. Multiplying the received signal by the same code $c(t)$ achieves this (must be done synchronously though).

Figure A.10 shows the effect of spreading in the frequency domain. The multiplication with the high rate spreading code sequence $c(t)$ broadens the transmit bandwidth and at the same time lowers the power spectral density.

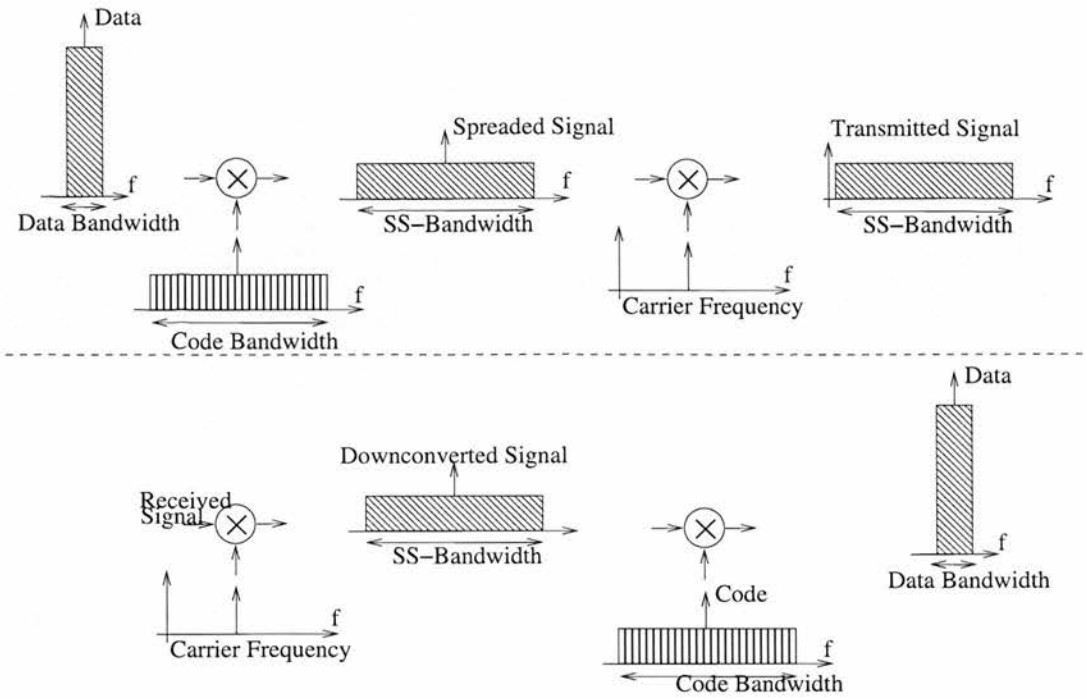

Figure A.10: Direct Sequence Spread Spectrum System signals in frequency domain

A.6.9 DS-CDMA

Direct Sequence Spread Spectrum can be used to provide *Code Division Multiple Access (CDMA)*. This is accomplished by assigning each user a different code for the spectrum spreading process. Under ideal conditions and if these codes are chosen properly, there will be negligible interference between the users despite them transmitting in the same frequency band at the same time. Under real conditions there is some residual multiple access interference.

A.6.10 Correlation

All Spread Spectrum CDMA receivers rely on correlation. This section introduces the concept of correlation.

For a discrete time signal, if $c_x(i)$ is the ith chip in a code x with length L, the aperiodic discrete time *autocorrelation function* is defined in equation A.13. This

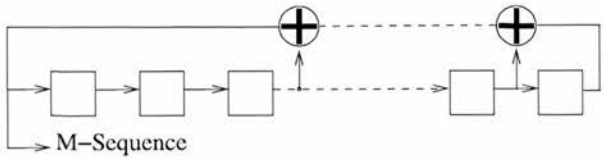

Figure A.11: Drawing of a code sequence generator

equation is correct if all users are synchronised and codes are mutually exclusive.

$$\Phi_{xx}(l) = \sum_{i=0}^{L-1} (c_x(i)c_x(i+l)) \text{ for } 0 \leq 1 \leq \text{L-1} \tag{A.13}$$

The discrete aperiodic cross-correlation function is defined as

$$\Phi_{xy}(l) = \sum_{i=0}^{L-1} (c_x(i)c_y(i+l)) \text{ for } 0 \leq 1 \leq \text{L-1} \tag{A.14}$$

For a CDMA system, the codes should have a high autocorrelation peak at $l = 0$. For all other values of l, $\Phi_{xx}(l)$ should be as close to zero as possible. This aids in signal acquisition and tracking and gives reliable symbol decisions. For a high user capacity, the codes must be mutually orthogonal, resulting in the condition for the cross-correlation function

$$\Phi_{xy}(l = 0) = 0 \tag{A.15}$$

A.6.11 Code generation

The spreading codes are generated using a shift register, with the appropriate XOR logic for feedback to the input. An example is illustrated in Figure A.11. A family of codes that can be generated using the shift register described are M-Sequences. These are generated by irreducible polynomials[58]. For different starting sequences the resulting M-sequence generator is always the same - it just has a different time offset, or *code phase* .

For cross-correlation purposes, it has been found [136] that only a small number of M-sequences have mutually *good* cross-correlation properties. This prevents their use for distinguishing between CDMA users.

A.6.12 Gold Sequences

Gold sequences are generated by the modulo 2 addition of two "good" M-sequences of equal length. This produces a large number of gold sequences because two M-sequences can have $2^N - 1$ different relative offsets between them, with each offset resulting in a different Gold Sequence. Gold sequences are used for scrambling in UMTS-FDD, separating terminals on the uplink and basestations (or sectors) on the downlink.

A.6.13 Walsh Sequences

Walsh sequences are orthogonal functions [168] and are used in UMTS as channelisation codes in both the uplink and downlink. On the uplink they separate data and control data, on the downlink they separate terminal channels. They can be computed by reading a row from the matrix produced by the following recursion rule

$$H_{2i} = \begin{pmatrix} H_i & H_i \\ H_i & -H_i \end{pmatrix} \tag{A.16}$$

This matrix is called the Hadamard matrix. As an example, the Hadamard matrix for order 4 in shown in equation A.17. It can be observed that the rows are mutually orthogonal.

$$H_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{A.17}$$

A.6.14 Receiver

The optimum receiver structure uses a matched filter. The receive signal is correlated with all possible transmit signals and the one with the strongest correlation result is chosen.

In a real system, there is interference caused by other users which are active in the same frequency band and transmitting at the same time - Multiple Access Interference

(MAI). In UMTS (FDD mode), the downlink signals are time aligned, however in the uplink this is not the case. Power control is employed to try to ensure all signals are received with the same power. In the ideal case, i.e. the signals have codes that are perfectly orthogonal, the interference user signals are time aligned. All the interferers vanish and the receiver has the same performance as in the single user case. However, the codes are not perfectly orthogonal, the signals are not time aligned and the signals do not have the same power. (In addition to the non-ideal characteristics of the transmitter and receiver).

As well as MAI and AWGN the received signal is strongly affected by multi-path fading. It is possible to design systems to take into account MAI by removing known interferers, but this is not done in practice due to the high amount of processing required. A less computationally intensive receiver which is used in UMTS implementations is the RAKE Receiver. There are two variants, the single user RAKE and the joint detection RAKE receiver. For practical reasons of cost, the single user RAKE is used in a mobile terminal and the joint detection receiver in a base station.

A.6.15 Single User RAKE

CDMA receivers are actually able to exploit the multi-path effects to improve their performance by using a good autocorrelation function (i.e. $\Phi_{xx}(\tau = 0) = L$, $\Phi_{xx}(\tau! = 0) = \varepsilon$), with ε very small compared to L. Due to the multi-path, the receiver's antenna has a superposition of the transmit signal, each with a different path delay. If the paths differ by one or more chip periods, then a correlation receiver synchronised to one of the paths can partially suppress all other paths. This helps reduce fading due to the multi-path effect, but relies on near-perfect time synchronisation with the desired signal.

The RAKE receiver makes use of several correlators operating in parallel, each with a different time offset, hence synchronised to a different path. Only the strongest paths are selected to be correlated and the output of each correlator is fed into a decision unit to produce the final output of the RAKE. This design results in a better receiver because it does not rely on just one path.

The name RAKE receiver come from its similarity to a garden rake. Figure A.12

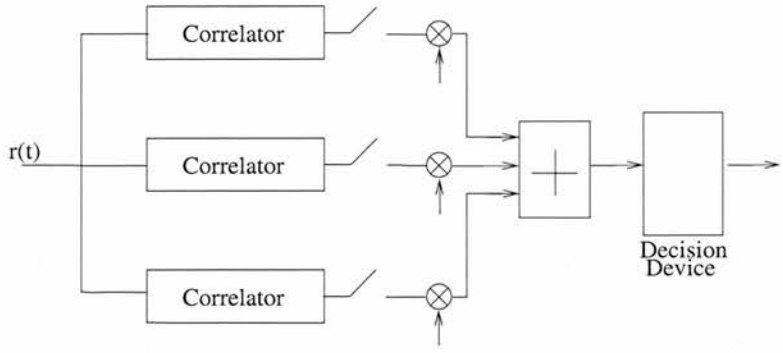

Figure A.12: Maximum ratio combining RAKE receiver with 3 fingers

shows a block diagram of the RAKE. The correlators are referred to as fingers. Each of the fingers is collecting the signal energy over the path to which it is synchronised. Combining the outputs of each finger produces an SNR better than a single finger could achieve. There are several methods of implementing the combination stage, but maximum ratio combining gives the highest SNR for the decision variable [94]. It is applied by weighting the outputs of the fingers with the complex conjugate coefficients of the path as shown in Figure A.12. Multiple antennae can also be used to further exploit spatial diversity, although the fingers must only track the paths of the radio channel associated with that antenna.

A.6.16 Multi User Detection

Combating the interference caused by other users in the system, rather than treating it as AWGN is referred to as multi-user-detection. This is split into two techniques, *Interference Cancellation (IC)* and *Joint Detection*. In this work we are concerned only with Joint Detection.

A.6.17 Joint Detection: Maximum Likelihood Sequence

Several types of joint detection techniques exist, each consisting of a correlator bank followed by some form of transformation. The Maximum Likelihood Sequence Estimator gives the best performance, but suffers from the computational complexity

growing at an exponential rate with the number of users. The technique used in the receiver looked at in this work is called the *Minimum Mean Square Error (MMSE) Detector* . Its complexity increases linearly with the number of users.

Appendix B

UMTS

B.1 Overview of UMTS physical Layer

B.1.1 UMTS Introduction

In 1991, ETSI created a technical sub-committee (now part of 3GPP[4]), to develop a third generation mobile system called Universal Mobile Telecommunication System (UMTS). Two air-interfaces were selected : FDD W-CDMA and TDD TD-CDMA. FDD W-CDMA which is better for large cell sizes and has had most operator interest is the air interface described here.

FDD mode is based on a DS-CDMA air-interface providing single frequency reuse, soft hand-off and the use of sectorised antennas. It is designed to be deployed in a cellular network with cells of different sizes, ranging from macro-cells (0.5Km to 10Km), micro-cells (50m to 500m) and pico-cells (5 to 50m). Handover between cells is designed to be transparent to the user. Further, coexistence with GSM, including handover between UMTS and GSM is emphasised.

One of the key advantages of UMTS over GSM is its spectral efficiency - estimated to be in the order of 2 or better[7]. UMTS uses two 5 MHz bands to provide data speeds of up to 2Mbps. With the introduction of High Speed Downlink Packet Access (HSDPA), peak rates of 14Mbps will be available.

The physical layer and its relationship with the two layers above it is illustrated in Figure B.1. A logical channel is characterised by the type of information transferred.

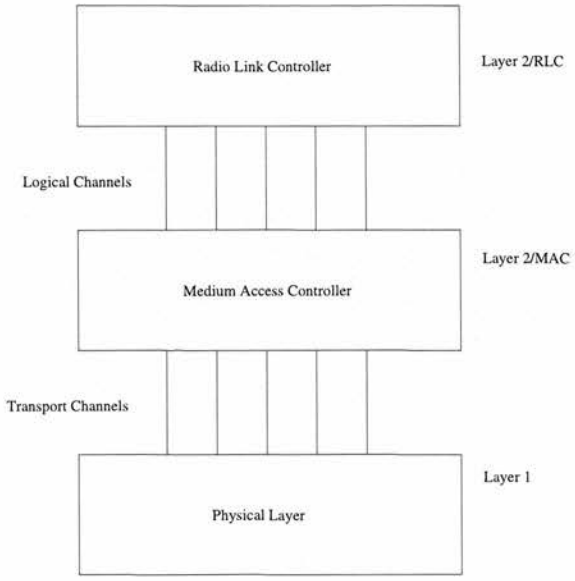

Figure B.1: Physical Layer Relation to Upper Layers

A transport channel is characterised by how the information is transported over the air-interface. A physical channel consists of radio frames and time slots. Once transport channels are channel coded and matched to the data rate offered by the physical channels, they are mapped to the physical channels.

B.1.2 Air Interface Principles

The chip rate of the system is 3.84Mcps. The frame length is 10ms which is divided into 15 slots. Spreading factors range from 4 to 256 in the uplink and 4 to 512 in the downlink. Thus on the uplink, the symbol rates vary from 960 Ksymbols/s to 7.5 Ksymbols/s. To separate channels from the same source Orthogonal Variable Spreading Factor (OVSF) codes are used. To separate sectors on the downlink, long scrambling codes constructed from Gold codes are used. Similar codes are used to separate users on the uplink and downlink. The processing of multiplying a vector by its code is often referred to as spreading and scrambling.

B.1.2.1 Handover

To enable terminal mobility with a continuous connection, UMTS has extensive support referred to as "Handover". There are several different types of handover:

- Inter-frequency handover
- Soft, softer and hard handover
- Handover between GSM and WCDMA
- Handover between TDD and FDD modes

The algorithm for making the decision to perform a handover can be based upon a number of different types of information. Examples are a physical channel's Bit Error Rate (BER), and transport channel's block error rate (BLER).

B.1.2.2 Power Control

In WCDMA all users share the same radio bandwidth, separated by codes. The result is that other users appear as random noise, so capacity is limited by interference. This makes good power control very important. FDD UMTS has fast closed loop and slower open loop power control mechanisms.

Fast power control operates at 1.5KHz (per slot) with a step size of 1dB, however larger step sizes are also permitted. It operates on both the Uplink and Downlink at a rate faster than any path loss change ($<$ 300Km/hr).

Outer loop power control operates at much lower frequency range - 10 to 100 Hz. Its purpose is to define the convergence point for which fast power control should be aiming. This is determined by higher layers which are based upon the required quality of service required by the data traffic. Measurements such as block error rate inform the decision. The basic aim is to minimise the power margin needed, therefore saving capacity.

B.1.3 Uplink Physical Channels

There are two dedicated uplink physical channels and two shared physical channels.

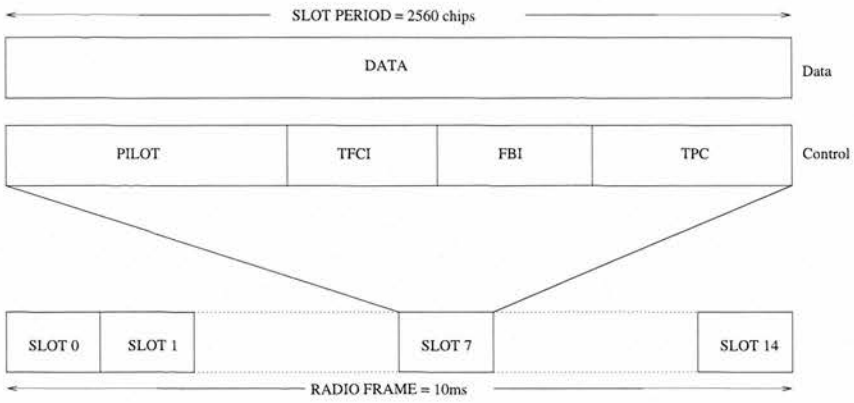

Figure B.2: Radio Frame Structure UL DPDCH and UL DPCCH

- The Uplink Dedicated Physical Data Channel (UL DPDCH) and the Uplink Dedicated Physical Control Channel (UL DPCCH).
- The Physical Random Access Channel (PRACH) and Physical Common Packet Channel (PCPCH)

The uplink DPDCH is used to carry data from the higher layers. The uplink DPCCH is used to carry control information generated by physical layer. Control information consists of known pilot bits to help with channel estimation, feed back information (FBI), transmit power control (TPC) commands and an optional transport-format combination indicator (TFCI). Figure B.2 shows the frame structure for the UL DPDCH and DPCCH channels. The message data content is determined by the spreading factor, though the chip rate stays constant. Many variable rate services can be multiplexed within each DPDCH frame but the spreading factor used may only vary between frames. Note that the complete radio frame must be received before the spreading factor is known since the FBI bits are split up across the 15 slots. This has major design consequences in the RAKE receiver discussed in Section B.2.3.

The PRACH is used to carry the RACH. The random-access transmission is based upon a slotted ALOHA approach with fast acquisition indication. The mobile can start a PRACH at certain well defined times called access slots. There are 15 access slots per two frames, spaced 5120 chips apart. The RACH transmission consists of a preamble of length 4096 chips, repeated with increasing transmit power until acknowledged by

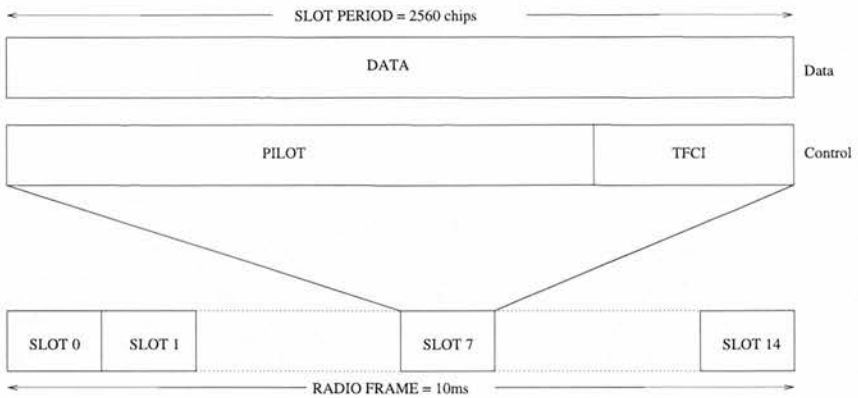

Figure B.3: Radio Frame Structure RACH message

the basestation, then a message part. The preamble part consists of a 16-bit signature repeated 256 times. In total there are 16 signatures based on a Hadamard code set of length 16. The message part is either 10ms or 20ms in duration. Figure B.3 shows the structure of the random-access message part radio frame. Each slot is split into two parts, a data part that carries layer 2 data and a control part that carries layer 1 control information.

The PCPCH is used to carry the Common Packet Channel (CPCH) transport channel. The CPCH is a contention based, random access channel used for bursty data traffic. It is very similar to the PRACH and is therefore not described further here.

B.1.4 Downlink Physical Channels

There is one Dedicated Physical Channel, one shared and five common control channels on the downlink:

- Downlink Dedicated Physical Channel (DPCH)
- Physical Downlink Shared Channel (PDSCH)
- Primary and Secondary Common Control Physical Channels (CCPCH)
- Primary and Secondary Common Pilot Channels (CPICH)
- Synchronisation Channel (SCH)

The Physical Downlink Shared Channel (PDSCH) is a variable-rate channel shared through the use of code multiplexing.

The primary CCPCH is a fixed-rate physical downlink channel used to carry a broadcast channel containing cell specific information. The secondary CCPCH is a variable rate physical downlink channel used to carry the Forward Access Channel (FACH) and Paging Channel (PCH). The PCH is used to support efficient sleep mode operation of mobiles. The FACH is similar in purpose to the RACH - used for short data bursts.

The Common Pilot Channel (CPICH) is a fixed-rate physical downlink channel that carries a predefined bit sequence. It is the default phase reference for all downlink physical channels.

The Synchronisation Channel (SCH) is used for cell search. It consists of two sub channels. The primary channel is the same across all channels so the mobile terminal can easily find it, the secondary channel (synchronised with the primary) then gives the UE details such as the code used by the cell.

The details of the downlink frame structure are not important for the understanding of material presented here. The reader is referred to the standard documents for details [2].

B.2 UMTS Partition Overview

B.2.1 Preamble Detector

The primary purpose of the preamble detector is to sense attempts by a UE to access the basestation. Three different preamble detectors are implemented using a 4-segment cross-correlation architecture. Most of the following describes the algorithm of a single preamble detector.

The UE transmits a known signature of 16 bits repeated 256 times which results in a 4096 chip preamble. Since the scrambling code used by the UE can be generated locally, cross-correlation is possible to detect the preamble. The results of the Code Matched Filters are fed into the post-detection integration (PDI) block where they are coherently accumulated. The PDI block also performs diversity combining on the two

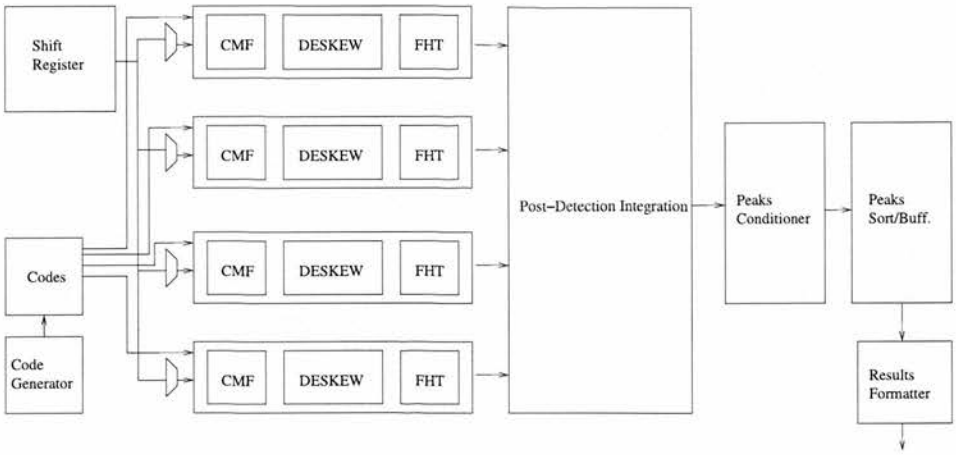

Figure B.4: UMTSSOC: Preamble Detector Partition

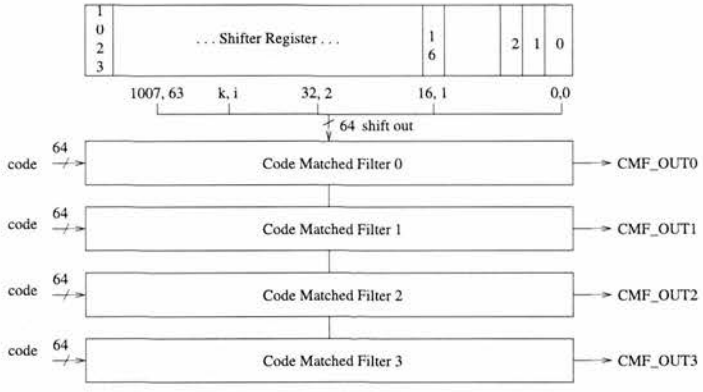

Figure B.5: UMTSSOC: Preamble Detector CMF Block

different antenna sources after they are accumulated. The output of the PDI block feeds the peak conditioning block which applies a threshold which acts like a filter. Finally the peak sorter finds the best set of peaks for each of the 3 preamble detectors.

As shown in Figure B.4, the correlation is implemented by splitting the task into four separate FIR filters. Inside each CMF block are four 64-chip wide matched filters, so in total there are sixteen 64-chip matched filters for both full and half-chip sample streams. Each segment works on 1024 of the 4096 chips in the preamble in a heavily time-multiplexed design. The 1024 chips are fed to the four correlators 64 chips at a time as shown in Figure B.5. With the 64-chip sample held constant, it is correlated

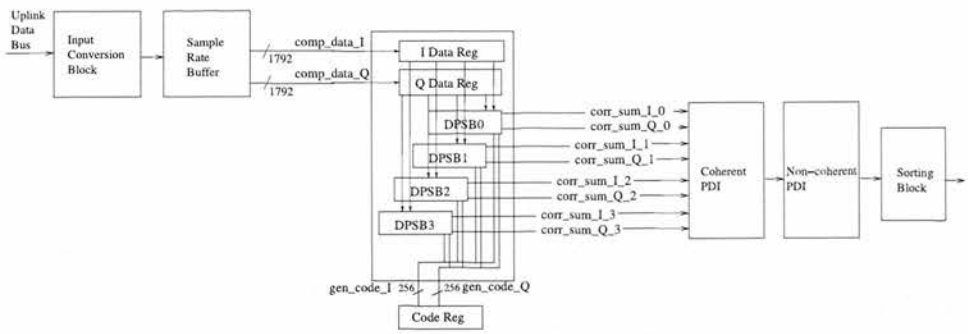

Figure B.6: UMTSSOC: Searcher Partition

against the code (broken into 64 bit chunks) and this is done for the three preamble detector codes. Therefore, the number of cycles required to correlate against a single hypothesis is $16 \times 3 = 48$ cycles which corresponds to the number of cycles available per chip period.

The Fast Hadamard Transform (FHT) function takes 16 dot-products produced by CMF block corresponding to a signature hypothesis and attempts to uncover the orthogonal Hadamard preamble code. Since the FHT demands the CMF sums in a different order from that in which they are produced, a de-skew stage is inserted to present them in the correct order. The output of the two Hadamard transform blocks (one for full and half-chip streams) is a set of 16 complex likelihood values per chip, one for each of the 16 possible preamble signatures.

The post-detection integration stage takes the likelihood values produced by the FHT and performs coherent and non-coherent accumulations. After the accumulation stage the diversity pair may be combined before the result is passed to the sorting block. Finally the sorting block produces the best set of peaks, which are formatted and stored in RAM.

B.2.2 Searcher

The searcher maintains a list of the strongest multi-path delay spreads for each of up to 64 dedicated channels. Figure B.6 shows a block diagram of the searcher.

Data is received from the Up-Link Data bus and is converted into a suitable format

by the Input Conversion Block, which also pre-selects the 12 sources to be available to the Searcher. The data is then stored in the Sample Rate Buffer.

To search for 64 users and 2 sources, the Searcher uses 4 Dot Product Sub-blocks (DPSB) in the Complex Code Matched Filter (CCMF). A delay spread of 7.5Km or 192 chip periods is determined to be a good search window. The length of each DPSB is 256 chips (one symbol), and they are spaced at 48 chip intervals with respect to the data. The CCMF Convolution Operation of the data source and the locally generated scrambling code is completed in 48 clocks per source component. With a clock frequency of 184.32MHz, a filter length of 256, and 192 convolutions required per chip (48 clocks), we see that we need 192/48 = 4 DPSB.

The correlated hypotheses generated by the CCMF pass through the Coherent and Non-Coherent Post Detection Integration Blocks to calculate the ultimate overall energy metric. Since the pilot values are known in advance, a coherent accumulation over the pilot bits is performed. All energy calculated in a time slot is non-coherently combined over multiple time slots.

The final results are sorted by the Sorter Block, which provides a table of the highest correlation peaks (per user) to the Post-Processing Block. Post-Processing is implemented by comparing the results to a pre-defined threshold. This block also contains an energy conversion function that scales the magnitude of the results.

B.2.3 RAKE Receiver

The RAKE partition demodulates and de-spreads the uplink channels. It may be viewed in terms of three separate functions - the Front End, Control Section and Back End. The RAKE-Front End operates at chip rate and delivers measurements of delay and magnitude of "fingers"[1] of a particular user. The RAKE Back End operates at symbol rate and interfaces to the Power Controller block and the ESIP. The RAKE internal Control coordinates the sub-blocks of the RAKE and performs communication with the UMTSCE Controller. Figure B.7 shows the control and data flow between the RAKE sub-blocks and other partitions.

[1] A finger is one correlator unit allocated to one multi-path component to be detected and demodulated.

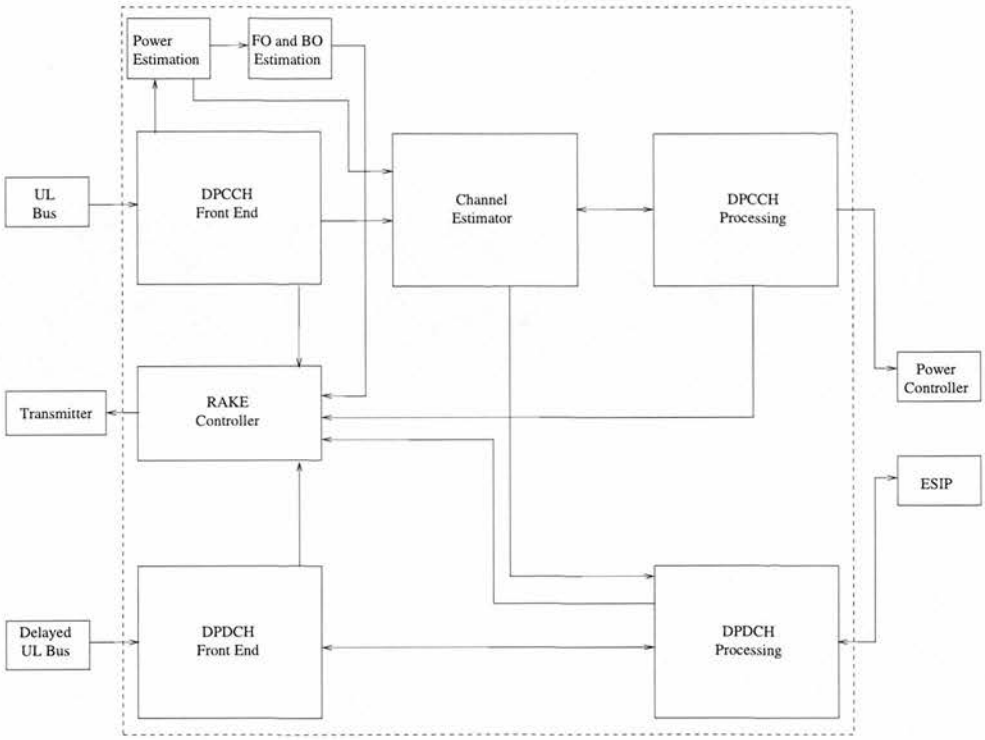

Figure B.7: UMTSSOC: RAKE Partition

The RAKE front-end receives sample data streams from the uplink for control part processing. The samples are also buffered off-chip to provide a second data stream that it is delayed in time by 1.026 frames relative to the control processing path. This is necessary because the control bits are spread across the entire radio frame and they must be retrieved first to find the data spreading factor used. For both control and data branches, 12 out of 36 input sample streams are selected by the Antenna Select block (not drawn in diagram). The input data streams are stored separately within a Sample Rate Buffer that is required to adjust receive timing in multiples of sub-chips for every RAKE finger of every user. These functions are common to all RAKE channels. The subsequent re-sample block extracts the exact source of samples and provides time synchronisation of input samples in fractional units of a chip for each RAKE finger of all users. The scrambling code and channelisation code are generated for de-spreading separately the control channel (DPCCH) and the data channel (DPDCH). The resulting

de-spread symbols of DPCCH early and late samples are fed into the tracker, which monitors the delay of each finger.

Power estimation produces commands for the fast uplink power control. Channel estimation occurs on the on-time (i.e. not buffered) DPCCH symbols to aid processing of the (delayed by one frame) DPDCH symbols. The processing of the DPCCH stream extracts the fields shown in Figure B.2, and uses the pilot bits to estimate uplink quality etc. The DPDCH processing uses the channel estimates to calculate log-likelihood ratios (LLR) for the DPDCH symbols. These are used for FEC decoding in the ESIP.

The granularity in time is one DPCCH-symbol, or 12288 clock cycles. Within this time period the operational blocks of the RAKE Receiver Partition process all 64 RAKE channels in a fixed time slice scheme. For easier representation, the time slice to handle one RAKE channel is called a user slice and is separated into 8 slices to perform 8 fingers, each slice is called a finger slice.

B.2.4 Power Controller

Good power control is essential for maximising the air interface as described in Section B.1.2.2. The RAKE performs demodulation and de-spreading of uplink channels and the transmitter modulates and spreads downlink channels. On the downlink, the RAKE supplies the PC with control information received in the uplink channel and the PC converts this into power multipliers used by the transmitter block. On the uplink, the RAKE generates power control information based on the uplink channels which are passed to the PC. In addition, when transmit power control bits are required, the power controller generates these commands for the transmitter.

B.2.5 Transmitter

The transmitter is the final partition in the downlink data path. It receives data from the Extended Soft Information Processor and power control weights from the PC and outputs transmit streams for digital to analog conversion. The transmitter performs the following functions:

- Channelisation with OVSF code

- Channel Construction
- Scrambling
- Multiplication of data with linear power values
- Pilot field generation, and TPC bit repetition
- Transmit diversity on all channels
- Creation of gaps in transmission due to compressed mode
- Multi-code transmission
- Softer handover

Transmit diversity and soft handover result in the transmission of downlink physical channels on a number of antennae. Transmit streams carrying the same data but destined for different antennae have different power control applied. Transmitter capability is described in terms of transmit resources, of which there are 192 in total. For the 64 user channels there are 128 transmit resources, 129-160 are for down-link shared channels and control channels. The remaining 161-192 transmit resources are for pilot channels etc.

B.2.6 Extended Soft Information Processor (ESIP)

The ESIP partition performs the majority of channel coding and multiplexing functions described in [1]. It operates on both uplink data and downlink data. As shown in Figure B.2.6, on the uplink it takes data from the RAKE receiver and performs the processing from 2nd de-interleaver to CRC computation. On the downlink, the ESIP takes logical transport channels, performs CRC attachment through to 2nd interleaving before passing the data on to the transmitter partition.

In normal operation, the downlink is capable of processing the 64 user channels in addition to the 32 shared channels. In the downlink there are two stages to the processing. First a complete TTI worth of data is stored in the 1st interleaver before it can be interleaved and supplied to the second stage which outputs the fully interleaved

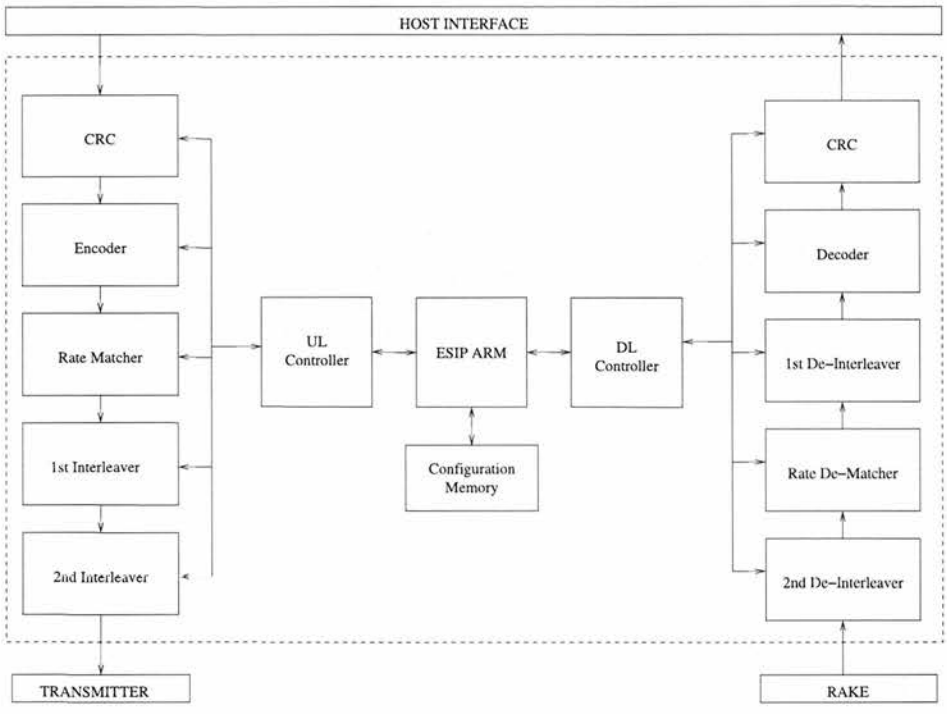

Figure B.8: UMTSSOC: ESIP Partition

and encoded data to the transmitter partition. In normal operation the uplink is capable of processing 64 user channels. Like the downlink, the uplink is also split into two stages. Physical channel processing from the RAKE receiver to the 1st-de-interleaver operates on radio frames and the 1st-de-interleaver operates on complete TTIs of data.

Each of the main function blocks is briefly described here:

- Interleavers: 1st interleaver operates on complete TTIs of data and the 2nd interleaver operates on radio frames of data. Interleaving involves reordering the data pseudo-randomly to improve frame error rates over the air interface.

- Channel Coding: The channel decoding architecture employs a unified approach, sharing functionality between Convolutional and Turbo codes as described in [21]. Channel encoding is much simpler than decoding, requiring significantly less hardware resources.

- Rate Matching: Applies puncturing and repetition to the data output by the encoder.
- CRC: A CRC is attached to each transport block within a user channel.

Appendix C

UMTSCE Parameterisation

C.1 Parameterisation

C.1.1 Introduction

The checkpoint framework for the design of reconfigurable computing systems, described in Chapter 4, involves expressing how each subsystem's resource demands vary with system parameters. These equations are then used to calculate what system configurations (primary instances) are needed to minimally span the entire set of system requirements. The parameterised development of subsystems may then proceed accordingly.

This section will examine both the logic and memory within the UMTSCE partitions to determine how their size depends on the parameters listed in Table C.1. The first three listed parameters are static, configured at installation of the base station. The parameters in the lower part of the table are dynamic, changing value at runtime depending on the usage pattern. The result will be a set of equations describing how logic and memory requirements of each partition vary with the parameters in Table C.1.

In Chapter 5 the equations are used to investigate the difference in resource requirements of a number of basestation configurations.

Parameter Name	Value
Number of radio sources	1-12
Searcher window size [chips]	50-400
Max. Number of fingers per user‡	4-16
No. of users	0-64
Peak data rate per user [kb/s]	0-384
Uplink spreading factor	4-256
TTI length [ms]	10-80
Peak total throughput [kb/s]	0-1152
Uplink channels	DCH/RACH
Coding†	C/T
Av. no. of links per DL channel‡	0-2
Total no. of common channels	0-32
No. of preamble detectors	0-3

†'T' implies turbo encoding and 'C', convolutional encoding.

‡Dedicated channels only: lower figures for RACH.

Table C.1: List of Parameters for UMTSCE Parameterised Design.

C.1.2 Logic

Detailed specification documents including the design of individual pieces of circuitry were available for determining the logic gate requirements of each partition. This involved a systematic procedure of becoming familiar with the functionality of the partition, then identifying its main subsystems. This led to a decision on how it should be parametrically described. As per the checkpoint framework, the subset of variables in Table C.1 which had an effect on the partition's logic gate requirements was identified. Then, the largest subsystem designs were carefully studied in detail to extract how their gate requirements varied with the parameter set, and then equations were written to capture the relationship.

As briefly mentioned in the introduction to this section, the UMTSCE was designed to perform the baseband processing of a single sector/carrier so that it could be treated as a discrete building block for constructing basestations of different sizes. Therefore,

a tradeoff was made between complexity, flexibility and performance. For example, the number of radio sources which the searcher partition can choose from is restricted to 12 out of a possible total of 36, and only 2 of the 12 available may be used during a search. In a large basestation with 36 radio sources available, it may be desirable to have a few more than 12 available to the searcher. Conversely, in a small basestation with only 6 radio sources, having the flexibility to provide 12 is wasteful. In this work, the tradeoffs made for the UMTSCE ASIC implementation are preserved from an algorithmic perspective, but the exact UMTSCE requirements are used to minimise unnecessary functionality.

Some of the partitions exhibit a high level of datapath time-multiplexing which makes it difficult to extract the way in which the implementation would scale if parameterised. Time-multiplexing is a technique which an ASIC implementation can employ to offer more flexibility, but comes at the cost of extra complexity due to additional state storage and control. The conservative approach taken here is to assume the gate count of a *lean* parameterised implementation is approximately equal to the gate count of a linearly scaled time-multiplexed implementation. It may be necessary to introduce ceiling functions to recognise that, rather than being continuous, there may be discrete steps in the parameterised implementation size. This is considered in Chapter 5 where the equations are put to use. Here we are concerned only with deriving the equations.

Table C.2 lists the percentage of each subsystem in the UMTSCE. Those partitions listed as 100% exhibited such heavy time-multiplexing that it was deemed reasonable to simply scale the entire partition. The partitions for which it was feasible to scale sub-blocks individually will have their equations normalised to 100% when they are used.

C.1.2.1 Preamble Detector

Although the preamble detector's operation is dependent on both the cell size and the number of preambles to be searched for, the cell size has little impact on the datapath complexity. This is because the radio sources will be continually screened by the datapath for evidence of a preamble request and it is only the accumulation algorithms

Partition Name	Percentage
Preamble Detector	82
Searcher	86
RAKE	100
Transmitter	100
Power Controller	100
ESIP	95

Table C.2: Percentage of UMTSCE logic described parametrically

which depend on the cell size. So cell size is more of a control factor rather than a complexity dimension. Up to three preamble detectors are available for searching any combination of the 3 possible preambles in order to provide flexibility.

The preamble detector's heart is the Code Matched Filter (CMF). As explained in the overview of the preamble detector, the CMF outputs correlation results to the FHT, which in turn outputs to the post-detection integration stage. Finally, the peaks sorter block produces the results.

These are the four largest logic blocks and their gate requirement simply depends on the number of preamble detectors. Complexity is not affected by which code is being detected nor, as previously discussed, the cell size. The logic gate requirement of each block is listed here:

$$CMF = 26,666 \cdot rach \tag{C.1}$$

$$FHT = 11,733 \cdot rach \tag{C.2}$$

$$PDI = 14,000 \cdot rach \tag{C.3}$$

$$Peaks = 5,333 \cdot rach \tag{C.4}$$

C.1.2.2 Searcher

In common with the Preamble Detector, the Searcher's logic gate count is dominated by a CMF engine. The Sorter and SRB are the next two largest blocks and together all three represent over 80% of the Searcher's gate count. The Searcher's CMF implementation is much more demanding than the Preamble Detector CMF since it must find the strongest multi-path components for up to 64 users inside every access slot. The search is conducted within a window, the size of which also has a direct impact on the Searcher CMF gate requirement. Since the Searcher algorithm operates on 256-chip DPSBs, it is the number of DPSBs required to satisfy the search requirements that determines the CCMF size. Equation C.5 describes the number of DPSBs required for a given Window Size and Number of Users. The ceiling function is used to avoid fractional numbers of DPSB.

$$\text{Number DPSB} = \left\lceil \frac{\text{Window Size} \cdot \text{Number Users}}{3,072} \right\rceil \tag{C.5}$$

The size of a DPSB is easily calculated by dividing the DPSB block in the UMTSCE by four. The registers feeding the CMF (one for I and Q parts) hold the data necessary to feed the DPSBs during a single user/source component search. Its gate count is therefore directly proportional to the Symbol Size + Window Size. Equation C.6 describes the total CMF size.

$$\begin{aligned} \text{Searcher CMF} =& (\text{Number DPSB} \cdot \text{DPSB Size}) + \text{Feed Registers} \\ =& \text{Number DPSB} \left(\frac{232,768}{4} \right) + 29,562 \left(\frac{\text{Window Size} + 256}{192 + 256} \right) \end{aligned} \tag{C.6}$$

The Searcher datapath flow from the CMF block through to the Sorter block is split into four streams originating from the four DPSB blocks. Since the streams emerging from the non-coherent PDI block must be sorted in parallel it is appropriate to parameterise the Sorter implementation by the number of DPSBs as described in equation

C.7.

$$\begin{aligned} \text{Searcher Sorter} &= \text{Number DPSB} \cdot \frac{\text{Sorter Size}}{4} \\ &= \text{Number DPSB} \cdot 13,395 \end{aligned} \tag{C.7}$$

The Searcher's SRB gate count is proportional to the number of sources that are to be made available to the CMF block. A maximum of 12 sources is made available in the UMTSCE design. Equation C.8 describes the SRB gate requirement.

$$\begin{aligned} \text{Searcher SRB} &= \left(\frac{\text{CE SRB Size}}{\text{Maximum UMTSCE Sources}} \right) \text{Number of Sources} \\ &= \left(\frac{32,784}{12} \right) \text{Number of Sources} \end{aligned} \tag{C.8}$$

The total logic gate count for the Searcher adjusted for the percentage of the total parameterised is given in equation C.9.

$$\begin{aligned} \text{Searcher Gates} &= 1.169\,(\text{CMF} + \text{SRB} + \text{Sorter}) \\ \text{Searcher Gates} &= 1.169 \left(\left(\left\lceil \frac{\text{Window Size} \cdot \text{Number Users}}{3,072} \right\rceil \right) 71,587 \right. \\ &\quad \left. + (\text{Window Size} + 256)66 + (\text{Sources})2,732 \right) \end{aligned} \tag{C.9}$$

C.1.2.3 RAKE

Most of the RAKE partition is highly time-multiplexed. For example, it uses a single finger to perform up to $64 \cdot 8 = 512$ fingers per slot and has a pipelined control path running alongside the datapath. It is appropriate to estimate that its resource usage scales linearly with the product of the (average) number of fingers per user and the number of users. Equation C.10 describes the gate requirements of the RAKE partition.

$$\begin{aligned} \text{RAKE Gates} &= \left(\frac{\text{Number Users} \cdot \text{Number Fingers}}{64 \cdot 8} \right) \cdot 1,030,989 \\ &= \left(\frac{\text{Number Users} \cdot \text{Number Fingers}}{512} \right) \cdot 1,030,989 \end{aligned} \tag{C.10}$$

C.1.2.4 Transmitter

The transmitter partition again uses heavy time multiplexing to provide the flexibility to facilitate transmission of 192 separate streams. As explained earlier in this chapter, a separate transmit path is required to prepare the same data on different sectors. Shared channels, soft handover and pilot channels all contribute to the total transmit resources being significantly greater than the maximum dedicated downlink channels (64). It is appropriate to estimate that the gate count of the transmitter partition scales linearly with the number of transmit resources required, as described in equation C.11.

$$\begin{aligned} \text{Transmitter Gates} &= \left(\frac{\text{Number Downlinks} + \text{Shared Channels}}{128+32+32}\right) \cdot 301{,}480 \\ &= \left(\frac{\text{Number Downlinks} + \text{Shared Channels}}{192}\right) \cdot 301{,}480 \end{aligned} \tag{C.11}$$

C.1.2.5 Power Controller

The power controller also exhibits a high level of time-multiplexing so its gate requirement must be estimated globally. Power control is only applied to the uplink dedicated channels so the power controller is estimated to scale linearly with the number of users as described in equation C.12.

$$\text{Power Controller Gates} = \left(\frac{\text{Number Users}}{64}\right) \cdot 117{,}000 \tag{C.12}$$

C.1.2.6 ESIP

The ESIP partition provides the interface for the UMTSCE datapath with the MAC layer. It has an ARM processor controlling operation of two sub-controllers, one for the uplink datapath and another for the downlink datapath. The demand placed upon all the controller units can be estimated to grow linearly with the number of users. The two direct datapath control blocks also scale with the aggregate user throughput. For the purposes of this investigation, the ARM gate count will be used as a measure of the demand placed upon it and it will be assumed that an alternative implementation exists which scales linearly with the number of users as described in equation C.13.

It is also estimated that the uplink and downlink controller implementations could be scaled according to the demand placed upon them as shown in equations C.14 and C.15.

$$\text{ARM} = \frac{\text{Number Users}}{64} \cdot 155,560 \tag{C.13}$$

$$\text{Downlink Controller} = \frac{\text{Number Users} \cdot \text{Av. DL Throughput}}{\text{Maximum DL Throughput}} \cdot 25,400 \tag{C.14}$$

$$\text{Uplink Controller} = \frac{\text{Number Users} \cdot \text{Av. UL Throughput}}{\text{Maximum UL Throughput}} \cdot 13,700 \tag{C.15}$$

Within the downlink datapath, the encoder and interleavers dominate gate count. The computational demand placed upon the encoder depends on two things: the aggregate throughput and whether it is turbo encoding or convolutional encoding. It is estimated that turbo encoding is three times more computationally expensive relative to convolutional, due to the second recursive convolutional encoder and its associated interleaver. Equation C.16 describes the encoder.

$$\begin{aligned}\text{DL Encoder} = \Big(& \sum_{user=1}^{user=64} \frac{\text{DL Convolutional Throughput(user)}}{\text{Max. DL Throughput} \cdot 4} \\ & + \sum_{user=1}^{user=64} \frac{\text{DL Turbo Throughput(user)} \cdot 3}{\text{Max. DL Throughput} \cdot 4} \Big) \cdot 43,000\end{aligned} \tag{C.16}$$

The first interleaver stores a TTI worth of transport blocks for each user and then interleaves them. However, dedicated channels with $TTI = 10ms$ are not interleaved. The demand placed upon the 1st Interleaver is therefore dependent on the TTI length of the dedicated channel. The demand is also dependent on the throughput as described in equation C.17.

$$\begin{aligned}\text{1st Interleaver} = & \sum_{user} \frac{\text{DL Throughput(user)}}{\text{Max. DL Throughput}} \cdot 20,200 \\ & \textit{for all user where TTI(user)} > \textit{10ms}\end{aligned} \tag{C.17}$$

The second interleaver is applied to all downlink data and is described in equation C.18.

$$\text{2nd Interleaver} = \sum_{user=1}^{64} \frac{\text{DL Throughput(user)}}{\text{Max. DL Throughput}} \cdot 13{,}750 \tag{C.18}$$

Uplink FEC is much more computationally demanding than FEC on the downlink since it is on the UL that the more demanding iterative algorithms are used to recover the data from the noisy air interface channel and reduce the BER. The uplink processing chain is basically the reverse of the DL chain. Equations C.19, C.20 and C.21 describe the decoder, 1st de-interleaver and 2nd de-interleaver respectively.

$$\begin{aligned}\text{UL Decoder} = (&\sum_{user=1}^{user=64} \frac{\text{UL Convolutional Throughput(user)}}{\text{Max. UL Throughput} \cdot 5} \\ &+ \sum_{user=1}^{user=64} \frac{\text{UL Turbo Throughput(user)} \cdot 4}{\text{Max. UL Throughput} \cdot 5}) \cdot 200{,}000\end{aligned} \tag{C.19}$$

$$\begin{aligned}\text{1st De-Interleaver} = &\sum_{user} \frac{\text{UL Throughput(user)}}{\text{Max. UL Throughput}} \cdot 11{,}520 \\ &\textit{for all user where TTI(user)} > \textit{10ms}\end{aligned} \tag{C.20}$$

$$\text{2nd De-Interleaver} = \sum_{user=1}^{64} \frac{\text{UL Throughput(user)}}{\text{Max. UL Throughput}} \cdot 17{,}352 \tag{C.21}$$

The total ESIP gate requirement is given in Equation C.22.

$$\begin{aligned}\text{ESIP Gates} = &1.05 \cdot (\text{ARM} + \text{DL Controller} \\ &+ \text{UL Controller} + \text{DL Encoder} + \text{1st Interleaver} \\ &+ \text{2nd Interleaver} + \text{UL Decoder})\end{aligned} \tag{C.22}$$

C.1.3 Memory

The UMTSCE memory was parameterised differently from the logic. Instead of parameterisation by partition, the largest memory blocks across all CE partitions were

Partition	Name	Size (Mbits)	Percentage of UMTSCE total
RAKE	Circular Buffer	2,048,000	15
ESIP	1st De-Interleaver	1,081,344	8
Searcher	Non-Coherent PDI	1,081,344	8
ESIP	1st Interleaver	921,600	7
Searcher	Coherent PDI	1,081,344	8
RAKE	Decimated FFT	851,968	6
ESIP	Config. Result	655,360	5
RAKE	Sample Rate Buffer	491,520	4
ESIP	2nd De-Interleaver	480,000	4
Preamble Detector	Shift Register	294,912	2
ESIP	Data TCM	262,144	2
Transmitter	Input Buffer	262,144	2
Preamble Detector	Deskew	172,032	1
	TOTAL	*9,683,712*	*72*

Table C.3: CE Memories Parameterised

selected to be described parametrically. They total 72% of the memory requirement and are listed in Table C.3 along with the percentage of the total UMTSCE memory they represent.

The rest of this subsection will give a brief outline of each of the memories and an equation describing their size.

C.1.3.1 RAKE Circular Buffer

The RAKE circular buffer holds 20 slots of data for feeding FIR channel estimation filters. These produce channel estimates using the DPCCH pilot channels for use when processing the DPCCH non-pilot symbols and the DPDCH data. Equation C.23 de-

scribes the parameterised RAKE Circular Buffer (CB).

$$\begin{aligned} \text{RAKE CB} &= \text{slots} \cdot \text{half-slots} \cdot \text{symbols/half-slot} \cdot \text{bits/symbol} \cdot \text{users} \cdot \text{fingers} \\ &= 20 \cdot 2 \cdot 5 \cdot 20 \cdot \text{users} \cdot \text{fingers} \\ &= 4,000 \cdot \text{users} \cdot \text{fingers} \end{aligned} \tag{C.23}$$

C.1.3.2 ESIP First De-Interleaver

For every transport channel, the first De-Interleaver memory must be able to store a complete TTIs worth of data plus an additional 30ms of data to allow previous stage (rate de-matcher) to continue streaming as the TTI of data is processed. The symbols are represented by 5-bit log-likelihood ratios (LLR) and the transport channels are typically coded at 1/3 rate.

It has been calculated that the worst case memory requirement of the De-Interleaver is equivalent to 16 64Kbps (TTI=80ms) DCH channels and their associated signalling channels of 3.4Kbps (TTI=40ms). Equation C.24a describes the ESIP de-interleaver worst-case size and C.24 parameterises the de-interleaver. The actual requirement (1,848,960 bits) is a little greater than that given in equation C.24a since the equation is an approximation which does not include such things as tail bits and CRCs.

$$\begin{aligned} \text{ESIP De-Interleaver} &= \left(\frac{16 \text{ users} \cdot 64 \text{ Kbps} \cdot (80\text{ms}+30\text{ms})}{1/3 \text{ coding rate}} \right. \\ &\quad \left. + \frac{16 \text{ users} \cdot 3.4 \text{ Kbps} \cdot (40\text{m}+30\text{ms})}{1/3 \text{ coding rate}} \right) \cdot 5bits \\ &= 349,344\text{symbols} \cdot 5 \text{ bits} \\ &= 1,746,720\text{bits} \end{aligned} \tag{C.24a}$$

$$\text{ESIP De-Interleaver} = \sum_{user=1}^{64} \left(\frac{\text{Throughput(user)} \cdot (\text{TTI(user)}+0.03)}{1/3} + \frac{3{,}400 \cdot (0.04+0.03)}{1/3} \right) \cdot 5$$

for all user where TTI(user) > 10ms

$$= \sum_{user=1}^{64} \left(\frac{\text{Throughput(user)} \cdot (\text{TTI(user)}+0.03)}{1/3} + 714 \right) \cdot 5$$

for all user where TTI(user) > 10ms

(C.24b)

C.1.3.3 Searcher Non-Coherent PDI

The Non-Coherent Post-Detection Integration (PDI) block takes the I/Q component results calculated by the Coherent PDI block and combines them into a magnitude, accumulating them over an entire frame. The magnitude (energy metric) may require 19 bits to store. Equation C.25 describes the Non-Coherent PDI memory.

$$\begin{aligned}\text{Searcher Non-Coherent PDI} &= \text{Users} \cdot \text{Window Length} \cdot \text{Samples/Chip} \cdot \text{Sources} \cdot \text{Energy Metric} \\ &= 64 \cdot 192 \cdot 2 \cdot 2 \cdot 19 \\ &= 933{,}888 \text{ bits}\end{aligned}$$

(C.25a)

$$\text{Searcher Non-Coherent PDI} = \text{Users} \cdot \text{Window Length} \cdot 76\text{bits} \qquad \text{(C.25b)}$$

C.1.3.4 ESIP 1st Interleaver

The ESIP Interleaver memory has a different worst case requirement from the 1st de-interleaver. In common with the De-Interleaver, only TTIs greater than 10ms are interleaved. The worst case service combination for the 1st Interleaver is 96 users at

40Kbps and a TTI of 80ms. Equation C.26 describes the 1st Interleaver memory size.

$$\begin{aligned}\text{ESIP 1st Interleaver} &= \sum_{user=1}^{64} \text{Throughput(user)} \cdot 1/\text{Coding Rate} \cdot \text{TTI(user)} \\ &\quad \textit{for all user where TTI(user)} > \textit{10ms} \\ &= \sum_{user=1}^{64} \text{Throughput(user)} \cdot 4.5 \cdot \text{TTI(user)} \\ &\quad \textit{for all user where TTI(user)} > \textit{10ms}\end{aligned} \tag{C.26a}$$

$$\begin{aligned}\text{ESIP 1st Interleaver} &= \sum_{user=1}^{64} \text{Throughput(user)} \cdot 4.5 \cdot \text{TTI(user)} \\ &\quad \textit{for all user where TTI(user)} > \textit{10ms}\end{aligned} \tag{C.26b}$$

C.1.3.5 Searcher Coherent PDI

The Coherent Post-Detection Integration block accumulates correlation results from the CMF block across multiple symbols. Most searches involve coherently accumulating symbol-level results from the Pilot field and the TPC field. It is described in equation C.27.

$$\begin{aligned}\text{Searcher Coherent PDI} &= \text{Window Length} \cdot \text{Samples per Chip} \cdot \text{I/Q} \cdot \text{Users} \cdot \text{sources/user} \cdot \text{result size} \\ &= 192 \cdot 2 \cdot 2 \cdot 64 \cdot 2 \cdot 11 \\ &= 1,081,344 \text{ bits}\end{aligned} \tag{C.27a}$$

$$\text{Searcher Coherent PDI} = \text{Window Length} \cdot \text{Users} \cdot 88 \text{ bits} \tag{C.27b}$$

C.1.3.6 RAKE Decimated FFT

The decimated FFT is used in frequency offset estimation. It performs 8 FFT64 (one per finger) and accumulates the power spectral values. It therefore scales with the number of users and the number of fingers per user. It stores up to 64 symbols per symbol,

with 26 bits per symbol. The size of the Decimated FFT buffer in the UMTSCE is described in equation C.28a, with the parameterised size given in C.28b.

$$\begin{aligned}\text{RAKE Decimated FFT} &= \text{Users} \cdot \text{Fingers per User} \cdot \text{Symbols per Finger} \cdot \text{Symbol Size} \\ &= 64 \cdot 8 \cdot 64 \cdot 26 \\ &= 851,968 \text{ bits}\end{aligned} \tag{C.28a}$$

$$\begin{aligned}\text{RAKE Decimated FFT} &= \text{Users} \cdot \text{Fingers per User} \cdot \text{Symbols per Finger} \cdot \text{Symbol Size} \\ &= \text{Users} \cdot \text{Fingers per User} \cdot 1,664 \text{ bits}\end{aligned} \tag{C.28b}$$

C.1.3.7 ESIP Configuration Result

The ESIP configuration and result RAM is used by the ARM processor in the ESIP. It has two purposes: a) to store the boot code for the processor and b) to hold the transport channel configuration information. When being used for b) the information in a) is overwritten. The memory's size is determined by its use as a store for the transport channel configuration information. It therefore scales according to the number of users as shown in equation C.29.

$$\text{ESIP Configuration and Result RAM} = \text{Users} \cdot 10,240 \text{ bits} \tag{C.29}$$

C.1.3.8 RAKE Sample Rate Buffer

The RAKE sample rate buffer (SRB) is a ring-buffer organised to hold 4 symbols for 12 antenna sources. The SRB is split into two parts, one part for DPCCH and one for DPDCH. The purpose of the SRB is to adjust receive timing in multiples of samples for the RAKE fingers. The size of the UMTSCE SRB RAM is given in equation C.30a

and is described parametrically in equation C.30b.

$$\begin{aligned}\text{SRB RAM} &= \text{Sources} \cdot \text{Symbols} \cdot \text{Sample Rate} \cdot \text{Chips/Symbol} \cdot \text{Sample Size} \cdot \text{DPDCH/DPCCH} \\ &= 12 \cdot 4 \cdot 2 \cdot 256 \cdot 10 \cdot 2 \\ &= 491,520 \text{ bits}\end{aligned} \tag{C.30a}$$

$$\begin{aligned}\text{SRB RAM} &= \text{Sources} \cdot \text{Symbols} \cdot \text{Sample Rate} \cdot \text{Chips/Symbol} \cdot \text{Sample Size} \cdot \text{DPDCH/DPCCH} \\ &= \text{Sources} \cdot 40,960\end{aligned} \tag{C.30b}$$

C.1.3.9 ESIP 2nd De-Interleaver

The 2nd De-interleaver operates on data on the uplink coming from the RAKE. It must buffer 2 data frames of LLR data for up to 64 users. It acts as a double buffer, one frame for processing by ESIP and one frame for receiving data from the RAKE unit.

The worst case memory required is 64 users with a spreading factor of 64, resulting in 600 LLRs per user frame, and 38,400 LLRs total per radio frame. Considering compressed mode operation results in an increase to 48,000. Double buffering and 5 bits per LLR make the total memory 480,000 bits as described in equation C.31a and parametrically described in C.31b.

$$\begin{aligned}\text{2nd De-interleaver} &= \text{2 Radio Frames} \cdot \text{Time per Radio Frame} \cdot \text{Compressed Mode Factor} \\ &\quad \cdot \text{LLR Size} \cdot \sum_{user=1}^{64} \text{Throughput(user)} \\ &= 2 \cdot 10ms \cdot 1.25 \cdot 5 \cdot 3,840,000 \\ &= 480,000 \text{ bits}\end{aligned} \tag{C.31a}$$

$$\text{2nd De-interleaver} = 480,000 \cdot \sum_{1}^{64} \frac{1}{s.f.} \tag{C.31b}$$

C.1.3.10 Preamble Detector Shift Register

The purpose of the Preamble Detector Shift Registers is to provide the data samples to the code matched filters as required. There are three shift registers, one for each preamble detector. They can contain any two radio sources, both full and half-chip samples. Each shift register is required to store 4,096 chips, outputting $4 \cdot 64 = 256$ chips at a time to the code matched filter. The total memory is shown in equation C.32a and parametrically described in C.32b.

$$\begin{aligned}\text{Preamble Detector Shift Registers} &= \text{Preamble Detectors} \cdot \text{Source per Detector} \cdot \text{Preamble Length} \\ &\quad \cdot \text{I/Q} \cdot \text{Sample Size} \cdot \text{Full and Half Chip} \\ &= 3 \cdot 2 \cdot 4,096 \cdot 2 \cdot 3 \cdot 2 \\ &= 294,912 \text{ bits}\end{aligned} \tag{C.32a}$$

$$\begin{aligned}\text{Preamble Detector Shift Registers} &= \text{Preamble Detectors} \cdot \text{Source per Detector} \cdot \text{Preamble Length} \\ &\quad \cdot \text{I/Q} \cdot \text{Sample Size} \cdot \text{Full and Half Chip} \\ &= \text{Preamble Detectors} \cdot 98,304 \text{ bits}\end{aligned} \tag{C.32b}$$

C.1.3.11 ESIP Data TCM

The ESIP Data Tightly Coupled Memory (TCM) is used by the ESIP ARM control channel and transport channel processing. It is estimated to scale with the number of control and transport channels, however as a first approximation this is proportional to the number of users. Equation C.33 shows this parameterised description.

$$\text{ESIP Data TCM} = \text{Users} \cdot 4,096 \text{ bits} \tag{C.33}$$

C.1.3.12 Transmitter Input Buffer

The Transmitter Input Buffer must be able to buffer the data destined for the 160 transmit resources. At first approximation this is proportional to the number of downlink

and common channels as shown in equation C.34.

$$\text{Transmitter Input Buffer} = \left(\frac{\text{Dedicated Channels} + \text{Common Channels}}{96}\right) \cdot 262,144 \text{ bits} \tag{C.34}$$

C.1.3.13 Preamble Detector Deskew

The output from the CMF block is not in the order required by the FHT block. It is the job of the de-skew block to reorder (de-skew) the data appropriately. The size of the memory is proportional to the number of Preamble Detectors as described in equation C.35.

$$\text{Preamble Detector Deskew} = \text{Preamble Detectors} \cdot 57,344 \text{ bits} \tag{C.35}$$

Appendix D

Review of the multiple-context FPGA

D.1 Introduction

In this review we focus on the multiple context FPGA work of Andre DeHon [52][53][54], one of the most referenced pieces of work in the area. In [52], DeHon argues the case for dynamically programmable FPGAs to be part of general purpose computing architectures. We discuss this and consider its suitability for the SoCs implemented by today's statically programmed FPGAs.

D.2 Modes of operations

A dynamically programmable gate array (DPGA) is an FPGA with multiple configuration memory cells per configuration bit. The configuration bits are arranged into planes, corresponding to an entire configuration for the FPGA. One configuration plane is active at any one time, i.e. the contents of its memory cell control the device's configuration. The DPGA was envisioned to operate in several modes: logic engine mode, time share mode, static mode and mixed mode[160]. Logic engine mode refers to switching context at the maximum frequency possible, where the reconfiguration frequency is less than the task frequency. Time share mode refers to switching context more slowly than the task frequency, perhaps in response to an interrupt or a data dependency. In static mode, an area of the device does not appear to be reconfiguring.

This is achieved by having the same configuration data in all contexts (or planes). Mixed mode refers to a single application making use of all modes in different areas of the device.

D.3 Critique

A concept used to reason about run-time reconfiguration is functional capacity[52][172]. In [52], DeHon reasons that an FPGA is operating at top functional capacity when all combinatorial logic elements are operating at maximum frequency. To achieve this operational capacity, every LUT output is registered and there is at most one stage of logic between flip/flops. DeHon notes that if a design has deeper logic paths, or goes unused for periods time, the FPGA logic capacity is under utilised. The goal of maximising operational capacity is set by DeHon in his Multiple Context FPGA work. This goal presents a number of practical obstacles which are not acknowledged in DeHon's work. We review these problems in this section.

D.3.1 Latency

The first problem concerns latency. If heavy pipelining is employed to achieve maximum capacity utilisation, circuit latency is also maximised. Many communication and DSP systems are latency sensitive so cannot exploit temporal pipelining to reduce resource requirements. These application domains together represent a very large fraction of those in which today's platform FPGAs are used.

D.3.2 Re-timing

The second problem is the additional area required to facilitate re-timing. If a circuit's logic depth is restricted to a single level, as is the case in a DPGA, then functions which cannot be performed by one LUT will take multiple clock cycles to compute. Signals therefore arrive at LUT inputs at different clock cycles and must be re-timed. A major caveat[160] of DeHon's original DPGA architecture[52] is the lack of architectural features to aid signal re-timing.

To perform signal re-timing[95][169] , either special purpose registers must be added to the FPGA architecture, or existing resources (flip/flops or RAM) used to construct the registers. DeHon recognises the re-timing problem in a followup publication [53], and selects the solution of special purpose registers, noting that the use of flip/flops for re-timing would dramatically reduce capacity. The cost of the re-timing registers on each of the LUT inputs increases the area required by DeHon's 8 context DPGA architecture by 46%[53]. This increase in area is in addition to the 24% increase for the configuration context RAM.

D.3.3 Parasitic Effects of Multiple Contexts

The third problem is closely related to the second problem. DeHon does not factor in all the effects of the run-time reconfiguration overhead. The equation derived for the area of the time-multiplexed FPGA LUT with separate input latches, is shown in equation D.1.

$$\text{Area Latched DPGA LUT} = 500\text{K}\lambda^2 + c \cdot 130\text{K}\lambda^2 \tag{D.1}$$

The lambda symbol represents half the minimum feature size and 'c' is the number of contexts. The single context FPGA LUT area is given in equation D.2.

$$\text{Area FPGA LUT} = 580\text{K}\lambda^2 \tag{D.2}$$

DeHon states the minimum period of the FPGA LUT as 7ns and the latched DPGA LUT as 9.5ns. The latched DPGA LUT period is treated as a constant - independent of the number of contexts. This cannot be the case, since the increased area of the LUT due to contexts affects the distance signals must propagate. This effect can be mitigated by transistor sizing, but it would increase the LUT area and this is not included in the equations. For example, according to equation D.1, the 32 context DPGA has 6x the area of the 2 context FPGA. A LUT area increase of 6x would increase wire length by sqrt(6). This translates directly into an increased delay of 2.5x or 17.5ns which is not discussed in the paper.

To illustrate the concept of an FPGA's maximum capacity, DeHon uses the throughput ratio, R, shown in equation D.3. This ratio provides a measure of how quickly the task (or circuit) operates compared with the maximum frequency the device can be clocked at. In essence, it captures the minimum LUT period (which is constant) to critical path period ratio.

$$R = \frac{\text{LUT period}}{\text{Task period}} \tag{D.3}$$

The graph in Figure D.1 shows the efficiency of running a task of throughput ratio R on a device with c contexts. The efficiency measure does not take into account the speed decrease introduced by the circuitry for multiple contexts. Figure D.2 draws the same graph with the efficiency values adjusted to reflect the reduced speed at which the multiple context device LUTs operate at. In the second graph, an efficiency value of one corresponds to all LUTs operating at the speed of the single context FPGA. The efficiency metric adjustment reflects the speed $\times$ area product without any transistor resizing to optimally adjust for the increase in area. At this design point for the DPGA, we note that the 4 context device is more efficient at a throughput ratio between two and three. The 8 context device is only more efficient than the single context FPGA above a throughput ratio of 4.

D.3.4 Finite State Machine

DeHon[53][54] proposes finite state machines (FSMs) as good candidate circuitry for implementation on a multi-context device. It is noted in particular that one never needs the full FSM logic at any point in time. During any cycle, the logic from only one state is active. It is proposed that in a multiple context FPGA, each context contains only a portion of the state graph. This technique is used by the multi-context programmable array (PLD) from National Semiconductor, MAPL[80]. If such a technique were used to implement FSMs on a multiple context FPGA, it would be extremely wasteful of resources. Each FSM in the system will potentially change state at different times,

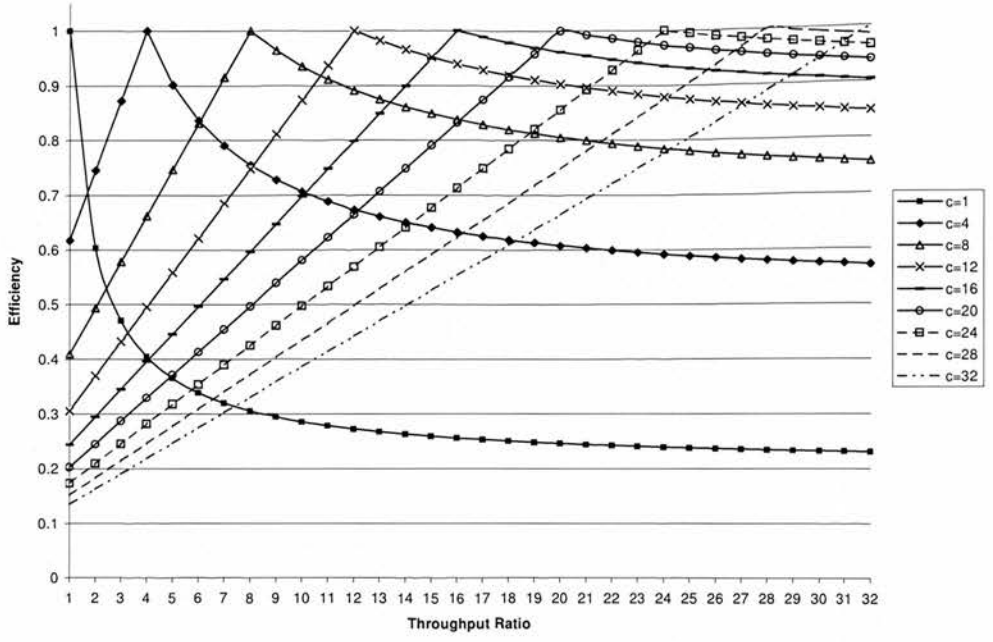

Figure D.1: DPGA utilisation efficiency versus throughput ratio for various context counts

therefore every partition of every FSM graph requires its own configuration memory plane. Assignment of an entire configuration memory plane to only a small area of the chip is extremely wasteful.

D.3.5 Power Consumption

Switching context at high frequency, as is done in logic emulation mode and perhaps time share mode, is reported by Trimberger et al.[160] to consume excessive levels of power. They note that although each configuration bit line has a small capacitance, there are 100,000 bit-lines in a 20x20 array. The time-multiplexed XC4000E architecture is reported to consume tens of watts when operating at 40MHz. This suggests that with current fabrication technology, reconfiguration on the order of a single cycle is not feasible due to thermal energy dissipation problems.

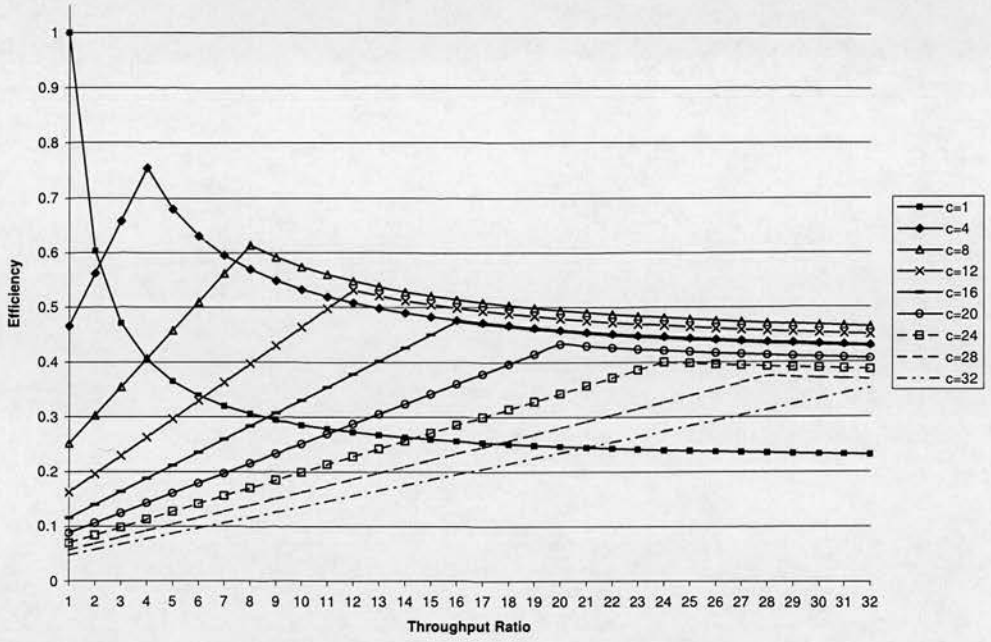

Figure D.2: DPGA utilisation efficiency versus throughput ratio for various context counts. Efficiency is adjusted for the worst case speed penalty caused by the addition of multiple context functionality.

Bibliography

[1] 3GPP. 3GPP RAN Multiplexing and channel coding (FDD) Tech. Spec 25.212. *3GPP Specification*, 1999.

[2] 3GPP. 3GPP RAN Physical Channels and Mapping of Transport Channels onto Physical Channels (FDD) Tech. Spec. 25.211. *3GPP Specification*, 1999.

[3] 3GPP. 3GPP RAN Radio Resource Control (RRC) Protocol Specification Tech. Spec. 25.331. *3GPP Specification*, 1999.

[4] 3GPP. http://www.3gpp.org/. URL, 2004.

[5] A. Abnous and J. Rabaey. Ultra Low Power Domain Specifc Multimedia Processors. *Proceedings IEEE VLSI Signal Processing Workshop*, 1996.

[6] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, and J. Rabaey. Evaluation of a Low-Power Reconfigurable DSP Archiecture. *Proceedings of the Reconfigurable Architectures Workshop*, March 1998.

[7] A.G. Acx and P. Mendribil. Capacity evaluation of the UTRA FDD and TDD modes. *49th Vehicular Technology Conference*, pages 1999–2003, 1999.

[8] R. Ahlswede. Multi-way communication channels. *2nd Int. Symp. on Information Transmission*, 1971.

[9] B. Altizer, L. Cooke, and G. Martin. Platform-Based Design: What is Required for Viable SoC Design? *Internet Website: Design-Reuse*, http://www.us.design-reuse.com/articles/article3340.html, 2004.

[10] J.M. Arnold. The Splash 2 Software Environment. *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 88–93, April 1993.

[11] P.M. Athanas and H.F. Silverman. Processor Reconfiguration Through Instruction Set Metamorphis. *IEEE Computer*, pages 11–18, March 1993.

[12] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. *IEEE Workshop on FPGAs for Custom Computing Machines*.

[13] T. Basten, L. Benini, A. Chandrakasan, M. Lindwer, J. Liu, R. Min, and F. Zhao. Scaling into Ambient Intelligence. *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, 2003.

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

[15] L.A. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal 5(2)*, pages 78–101, 1966.

[16] L. Benini and G. De Micheli. Networks on chips: A New SoC Paradigm. *IEEE Computer*, pages 70–78, 2002.

[17] P. Bertin, D. Roncin, and J. Vuillemin. Programmable Active Memories: A performance Assessment. *Proceedings of Symposium on Research on Integrated Systems*, pages 88–102, 1993.

[18] V. Betz and J. Rose. FPGA routing architecture: segmentation and buffering to optimize speed and density. *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 59–68, 1999.

[19] N.B. Bhat. Novel Techniques for High Performance Field Programmable Logic Devices. *UCB/ERL M93/80 University of California Berkeley*, 1993.

[20] Q. Bi, G. Zysman, and H. Menkes. Wireless mobile communications at the start of the 21st century. *IEEE Communications Magazine*, 39:110–116, 2001.

[21] M. Bickerstaff and D. Garret et. al. A unified Turbo/Viterbi Channel Decoder for 3GPP Mobile Wireless in 0.18um CMOS. *IEEE ISSCC*, pages 124–125, February 2002.

[22] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. *International Conference on Lisp and Functional Programming 98*, 1998.

[23] G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. *6th International Workshop on Field-Programmable Logic and Applications*, pages 327–336, September 1996.

[24] G. Brebner. CHASTE: a Hardware-Software Codesign Testbed for the Xilinx XC6200. *Proceedings of the 4th IPPD/SPDP Reconfigurable Architectures Workshop*, pages 16–23, 1997.

[25] G. Brebner. The Swappable Logic Unit. *Proc. 5th Annual IEEE Symposium on Custom Computing Machines*, pages 77–86, 1997.

[26] G. Brebner. Circlets: Circuits as Applets. *IEEE Field Programmable Custom Computing Machines (FCCM)*, pages 300–301, 1998.

[27] G. Brebner. Multithreading for Logic-Centric Systems. *Field-Programmable Logic and Applications*, pages 5–14, September 2002.

[28] G. Brebner. Eccentric SoC Architectures as the future norm. *Proceedings Euromicro Symposium on Digital System Design*, pages 2–9, September 2003.

[29] G. Brebner and O. Diessel. Chip-Based Reconfigurable Task Management. *Field-Programmable Logic and Applications*, pages 182–191, 2001.

[30] G. Brebner and A. Donlin. Runtime Reconfigurable Routing. *Proceedings of the 5th IPPD/SPDP Reconfigurable Architectures Workshop*, pages 25–30, 1998.

[31] G. Brebner and I. Kennedy. Circlets: Circuitry over the Internet. *IEEE Field Programmable Custom Computing Machines (FCCM)*, 2001.

[32] G.J. Brebner. Programmable Logic Has More Computational Power than Fixed Logic. *Proceedings 14th International Conference on Field-Programmable Logic and Applications*, pages 404–413, 2004.

[33] J. Brelet and B. New. Designing Flexible, Fast CAMS with Virtex Family FPGAs. *Xilinx Literature XAPP203*, September 1999.

[34] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit. A Dynamic Reconfiguration Run-Time System. *IEEE Proceedings of the International Symposium on FPGAs for Custom Computing Machines*, pages 66–75, 1997.

[35] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[36] P. Cao and S. Irani. Cost-aware www proxy caching. *In Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[37] J.M.P. Cardoso. A Novel Algorithm Combining Temporal Partitioning and Sharing of Functional Units. *IEEE 9th Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, May 2001.

[38] A.B. Carlson. *Communication Systems - An Introduction to Signals and Noise in Electrical Communication*. McGraw-Hill, 3rd edition, 1986.

[39] D. Chang and M. Marek-Sadowska. Partitioning Sequential Circuits on dynamically reconfigurable FPGAs. *International Symposium on Field Programmable Gate Arrays*, February 1998.

[40] D. Chang and M. Marek-Saowska. Buffer Minimisation and Time-Multiplexed I/O on Dynamically Reconfigurable FPGAs. *International Symposium on Field Programmbale Gate Arrays (FPGA)*, 1997.

[41] K. Chapman. Constant Coefficient Multipliers for the XC4000E. *Xilinx Application Literature XAPP054*, December 1996.

[42] K. Chapman, P. Hardy, A. Miller, and M. George. CDMA Matched Filter Implementation in Virtex Devices. *Xilinx Application Notes: XAPP212*, http://direct.xilinx.com/bvdocs/appnotes/xapp212.pdf, January 2001.

[43] D. Cherepacha and D. Lewis. DP-FPGA: an FPGA architecture optimised for datapaths. *Proceedings of 9th International Conference on VLSI Design*, pages 329–343, 1996.

[44] C. Chou, S. Mohanakrishnan, and J.B. Evans. FPGA implementation of digital filters. *Proceedings of the Fourth International Conference on Signal Processing and Applications and Technology*, pages 80–88, 1993.

[45] K. Compton, J. Cooley, S. Knol, and S. Hauck. Abstract: Configuration Relocation and Defragmentation for FPGAs. *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[46] K. Compton and S. Hauck. Totem: Custom Reconfigurable Array Generation. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[47] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck. Configuration Relocation and Defragmentation for FPGAs. *IEEE Transactions on VLSI*, pages 209–220, 2002.

[48] M. Dales. Initial Analysis of the Proteus Architecture. *11th International Conference on Field Programmable Logic and Applications*, pages 623–627, August 2001.

[49] W.J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. *Design Automation Conference*, pages 684–689, 2001.

[50] A. Dandalis and V. Prasanna. Configuration Compression for FPGA-based Embedded Systems. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[51] W.B. Davenport. *Probability and Random Processes*. McGraw-Hill, 1970.

[52] A. DeHon. DPGA Utilization and Application. *ACM/SIGDA International Symposium on FPGAs*, 1996.

[53] A. DeHon. Dynamically Programmable Gate Arrays: A Step Toward Increased Computational Density. *Fourth Canadian Workshop of Field Programmable Devices*, May 1996.

[54] A. DeHon. Reconfigurable Architectures for General-Purpose Computing. *Ph.D. Thesis, AI Technical Report 1586*, September 1996.

[55] A. DeHon. Balancing Interconnect and Computation in a Reconfigurable Computing Array. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February 1999.

[56] O. Diessel and H. ElGindy. Run-Time Compaction of FPGA Designs. *7th International Workshop on Field-Programmable Logic and Applications*, pages 131–140, September 1997.

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

[58] R.C. Dixon. *Spread Spectrum Systems with Commercial Applications*. Wiley, 3rd edition, 1994.

[59] J. Doble. *Introduction to Radio Propagation for Fixed and Mobile Communications*. Artech House, 1996.

[60] A. Donlin. Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry. *Proceedings 7th International Workshop on Field-Programmable Logic and its Applications*, pages 200–208, 1998.

[61] C. Ebeling, D.C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. *Proceedings of 6th International Workshop on Field-Programmable and its Applications*, pages 126–135, September 1996.

[62] H. Eggers, P. Lysaght, H. Dick, and G. McGregor. Fast Reconfigurable Crossbar Switching in FPGAs. *Proceedings 6th International Workshop on Field-Programmable Logic and its Applications*, pages 297–306, 1996.

[63] P.W. Foulk. Data-Folding in SRAM Configurable FPGAs. *Proceedings of the IEEE International Symposium on Field Configurable Custom Computing Machines (FCCM)*, pages 163–171, 1993.

[64] P.C. French and R.W. Taylor. A self-reconfiguring processor. *IEEE Workshop on FPGAs for Custom Computing (FCCM)*, pages 50–59, April 1993.

[65] K.M. GajjalaPurna and D. Bhatia. Paritioning in Time: A Paradigm for Reconfigurable Computing. *IEEE ICCD*, pages 340–345, October 1998.

[66] K.M. GajjalaPurna and D. Bhatia. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *IEEE Transactions on Computers*, 48:579–591, 1999.

[67] M. George. *DDR SDRAM DIMM Interface for Virtex-II Devices*, January 2003.

[68] A. Giri, V. Visvanathan, S.K. Nandy, and S.K. Ghoshal. High Speed Filtering on SRAM based FPGAs. *Proceedings of 7th International Conference on VLSI Design*, pages 229–232, 1994.

[69] I. Glover and P. Grant. *Digital Communications*. Prentice Hall, 1998.

[70] G.R. Goslin. Using Xilinx FPGAs to design custom digital signal processing devices. *Proceedins of DSPX*, pages 565–604, January 1995.

[71] J.P. Gray and T.A. Kean. Configurable Hardware: New Paradigm for Computation. *The 10th CalTech Conference on VLSI*, pages 277–293, March 1989.

[72] S.A. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based Interface for Reconfigurable Computing. *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[73] B. Gunther, G. Milne, and L. Narasimhan. Assessing Document Relevance with Run-Time Reconfigurable Machines. *Proceedings IEEE Workshop on Field Programmable Gate Arrays for Custom Computing Machines*, 1996.

[74] Y. Ha, B. Mei, P. Schaumont, S. Vernalde, R. Laurwereins, and H. DeMan. Development of a Design Framework for Platform-Independent Networked Reconfiguration of Software and Hardware. *Proceedings 11th International Workshop on Field-Programmable Logic and its Applications*, pages 264–274, August 2001.

[75] Y. Ha, P. Schaumont, M. Engels, S. Vernalde, F. Potargent, L. Rijnders, and H. DeMan. A Hardware Virtual Machine to Support Networked Reconfiguration. *Proceedings 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, pages 194–199, June 2000.

[76] R. Hartenstein. A Decade of Reconfigurable Computing: A Visionary Perspective. *Proceedings of the Conference on Design Automation and Test Europe (DATE)*, pages 13–16, March 2001.

[77] S. Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, 1998.

[78] S. Hauck, Z. Li, and E. Schwabe. Configuration Compression for Xilinx 6200 FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1107–1113, 1999.

[79] J.R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.

[80] D. Hawley. Advanced PLD Architectures. *Proceedings of the Conference on Field Programmable Gate Arrays*, pages 11–23, 1991.

[81] S. Haykin. *Communication Systems*. Wiley, 3rd edition, 1994.

[82] P.M. Heysters, J. Smit, G. Smit, and P. Havinga. Mapping of DSP Algorithms on Field Programmable Function Arrays. *Proceedings of the 10th Workshop on Field-Programmable Logic and Applications*, pages 400–411, 2000.

[83] M. Holland and S. Hauch. Automatic Creation of Reconfigurable PALs/PLAs for SoC. *Proceedings International Conference on Field Programmable Logic and Applications*, pages 536–545, 2004.

[84] Y.H. Hu. *Programmable Digital Signal Processors*. Marcel Dekker, New York, 2000.

[85] D. Jones and D.M. Lewis. A time-multiplexed FPGA architecture for logic emulation. *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1995.

[86] M. Kaul and R. Vemuri. Temporal Partitioning combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs. *Design and Test in Europe (DATE)*, pages 389–396, 1999.

[87] T. Kean, B. New, and B. Slous. A Fast Constant Coefficient Multiplier for the XC6200. *Proceedings of the 6th International Workshop on Field Programmable Logic and Applications*, 1996.

[88] E. Keller, G. Brebner, and P. James-Roxby. Software Decelerators. *Field-Programmable Logic and Applications*, pages 385–395, September 2003.

[89] I. Kennedy. Exploiting Redundancy to Speedup Reconfiguration of an FPGA. *International Conference on Field-Programmable Logic and Applications*, 2003.

[90] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion=Transistor Era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[91] B.S. Landman and R.L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Trans. on Computers*, C-20:1469–1479, 1971.

[92] G. Lee and G. Milne. A methodology for design of run-time reconfigurable systems. *Proceedings of the IEEE International Conference on Field-Programmable Technology,(FPT).*, pages 60–67, 2002.

[93] W. Lee. Overview of Cellular CDMA. *IEEE Trans Vechicular Technology*, 1991.

[94] W.C.Y. Lee. *Mobile Communications Design Fundamentals*. Wiley, 2nd edition, 1993.

[95] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. *Proceedings Third Caltech Conference on VLSI*, March 1993.

[96] E. Lemoine and D. Merceron. Run Time Reconfiguration of FPGA for Scanning Genomic Databases. *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 90–98, 1995.

[97] K. Li and K.H. Cheng. Complexity of resource allocation and job scheduling problems on partitionable mesh connected systems. *Proceedings 1st IEEE Symposium on Parallel and Distributed Processing*, pages 358–365, 1989.

[98] Z. Li. *Configuration Management Techniques for Reconfigurable Computing*. PhD thesis, Northwestern University, 2002.

[99] Z. Li, K. Compton, and S. Hauck. Configuration Caching Management Techniques for Reconfigurable Computing. *IEEE Symposium for Custom Computing Machines*, pages 87–96, 2000.

[100] Z. Li, K. Compton, and S. Hauck. Configuration Caching Techniques for FPGA. *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2000.

[101] Z. Li and S. Hauck. Don't Care Discovery for FPGA Configuration Compression. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 91–100, 1999.

[102] Z. Li and S. Hauck. Configuration Compression for Virtex FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[103] Z. Li and S. Hauck. Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation. *Proceedings IEEE International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 187–195, 2002.

[104] J. Liang, S. Swaminathan, and R. Tessier. aSoC: A scalable, Single-Chip Communications Architecture. *IEEE Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[105] H. Liao. *Multiple Access Channels*. PhD thesis, Dept. of Electrical Enginerring, U. of Hawaii, 1972.

[106] X.P. Ling and H. Amano. WASMII: A data driven computer on virtual hardware. *IEEE Workshop on FPGAS for Custom Computing Machines (FCCM)*, pages 33–42, 1993.

[107] H. Liu and D.F. Wong. Network Flow Based Circuit Partitioning for Time-Multiplexed FPGAs. *International Conference on Computer Aided Design*, November 1998.

[108] H. Liu and D.F. Wong. A graph theoretic optimal algorithm for schedule compression in time-multiplexed FPGA partitioning. *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design (ICAD)*, pages 400–405, 1999.

[109] P. Lysaght and J. Dunlop. Dynamic reconfiguration of FPGAs. *Proceedings of the International Workshop on Field Programmable Logic and applications (FPL)*, pages 33–42, 1993.

[110] J. MacBeth and P. Lysaght. Dynamically Reconfigurable Cores. *Field Programmable Logic and Applications*, pages 462–472, August 2001.

[111] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs. *Field-Programmable Logic and Applications*, pages 795–805, 2002.

[112] C. Matsumoto. Net processors face programming trade-offs. *EE Times*, http://www.eetimes.com/story/OEG20020830S0061, November 2002.

[113] R.N. McDonough and A.D. Whalen. *Detection of Signals in Noise*. Academic Press, 2nd edition, 1995.

[114] D. Meng, R. Gunturi, and M. Castelino. IXP2400 Intel Network Processor IPv6 Forwarding Benchmark Full Disclosure Report for Gigabit Ethernet. *Network Processing Forum Benchmarking Working Group*, 2003.

[115] P. Merino, M. Jacome, and J.C. Lopez. A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems. *Proc. IEEE Symposium on FPGA's for Custom Computing Machines (FCCM)*, pages 324–325, 1998.

[116] D.A.B. Miller. Optical Interconnects to Silicon. *IEEE Journal Selected Topics in Quantum Electronics*, pages 1312–1317, 2000.

[117] L. Mintzer. FIR Filters with Field-Programmable Gate Arrays. *Journal of VLSI Signal Processing*, pages 119–127, 1993.

[118] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. *Proceedings IEEE Field Programmable Custom Computing Machines*, April 1996.

[119] T. Miyamori and K. Olukotun. REMARC: Reconfigurable Multimedia Array Coprocessor. *Proceedings ACM Conference on Field Programmable Gate Array (FPGA)*, February 1998.

[120] V. Nollet, J-Y, Mignolet, T.A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems. *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2003.

[121] V. Nollett, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfiguable SoC. *Reconfigurable Architectures Workshop 2003*, 2003.

[122] S. Ogrenci and M. Sarrafzadeh. Stategically Reconfigurable Systems. *Reconfigurable Architectures Workshop 2001*, 2001.

[123] OSCI. Open System C Initiative. http://www.systemc.org/.

[124] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures. *Reconfigurable Architectures Workshop (RAW)*, pages 31–36, 1998.

[125] A. Papoulis. *The Fourier Integral and its Applications*. McGraw-Hill, 1962.

[126] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, 3rd edition, 1991.

[127] S.R. Park and W. Burleson. Configuration Cloning: Exploiting Regularity in Dynamic DSP Architectures. *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 81–89, 1999.

[128] V. Pasham, A. Miller, and K. Chapman. Transposed Form FIR Filters. *Xilinx Application Literature XAPP219*, October 2001.

[129] C. Patterson. High Performance DES Encyrption in Virtex FPGAs using JBits. *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 113–121, April 2000.

[130] J.G. Proakis. *Digital Communications*. McGraw-Hill, 3rd edition, 1995.

[131] J. Rabaey. Silicon Architectures for Wireless Systems. *Tutorial at Hot Chips*, http://www.hotchips.org/archive/hc13/hc13pres_pdf/t1a-rabaey.pdf, 2001.

[132] T.S. Rappaport. *Wirelss Communications*. Prentice Hall, 1996.

[133] R. Razdan and M.D. Smith. High-Performance Microarchitectures with Hardware-Programmable Functional Units. *Proc. 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–180, November 1994.

[134] L.G. Roberts. Beyond Moore's Law: Internet Growth Trends. *IEEE Computer*, 33:117–119, 2000.

[135] S. Singh and P. James-Roxby. Rapid Construction of Partial Configuration Datastreams from High-Level Constructs Using JBits. *Proceedings 11th International Conference on Field-Programmable Logic and Applications*, pages 346–356, August 2001.

[136] D.V. Sarwate and M.P. Pursley. Crosscorrelation Properties of Pseudorandom and Related Sequences. *Proc. IEEE*, 68:593–619, May 1980.

[137] P. Schaumont, I. Verbauwhede, M. Sarrafzadeh, and K. Keutzer. A Quick Safari Through the Reconfiguration Jungle. *38th Conference on Design Automation (DAC)*, June 2001.

[138] H. Schmitt. Incremental reconfiguration for pipelined applications. *IEEE Proceedings 5th International Symposium on FPGAs for Custom Computing Machines*, pages 47–45, 1997.

[139] B. Schnoer, C. Jones, and J. Villasenor. Issues in Wireless Video Coding Using Run-time-reconfigurable FPGAs. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 85–89, 1995.

[140] SDR. Software defined radio forum. URL.

[141] S. Sezer, R. Woods, J. Heron, and A. Marshall. Fast Partial Reconfiguration for FCCMs. *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 318–319, 1998.

[142] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. *Proceedings 2nd Workshop on Network Processors (NP-2), 9th International Symposium on High Performance Computer Architectures*, February 2003.

[143] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technology Journal*, 27:379–423, July 1948.

[144] SIA. Semiconductor Industry Association: The National Technology Roadmap for Semiconductors. *Technical Report*, 2003.

[145] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA Coprocessors. *Proceedings of the 10th International Workshop on Field Programmable Logic (FPL)*, pages 121–130, 2000.

[146] S. Singh. Death of the RLOC? *IEEE International Symposium on Field Configurable Custom Computing Machines*, 2000.

[147] K. Siwiak. *Radiowave Propagation and Antennas for Personal Communications*. Artech House, 1998.

[148] B. Sklar. Rayleigh Fading Channels in Mobile Digital Communication Systems Part I: Characterisation. *IEEE Communications Magazine*, 35:90–101, July 1997.

[149] B. Sklar. Rayleigh Fading Channels in Mobile Digital Communication Systems Part I: Mitigation. *IEEE Communications Magazine*, 35:102–109, July 1997.

[150] D. Slepian and J.K. Wolf. A coding theorem for multiple access channels with correlated sources. *Bell System Technology Journal*, September 1973.

[151] M.J. Smith. *Application-specific Integrated Circuits*. Addison Wesley Longman, 1995.

[152] J. Spillane and H. Owen. Temporal Partitioning for Partially Reconfigurable Field Programmable Gate Arrays. *4th Reconfigurable Architectures Workshop*

(RAW), part of the Proceedings of the 13th International Parallel and Distributed Processing Symposium (IPDPS) Workshops 1998, March 1998.

[153] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. Using On-Chip Configurable Logic to Reduce Embedded System Software Energy. *Proceedings IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2002.

[154] S. Sudhir, S. Nath, and S.C. Goldstein. Configuration Caching and Swapping. *Proceedings 11th International Conference on Field Programmable Logic and Applications*, pages 192–201, August 2001.

[155] E.J. Tan and W.B. Heinzelman. DSP architectures: past, present and futures. *ACM SIGARCH Computer Architecture News*, pages 6–19, June 2003.

[156] C. Tanougast, Y. Berviller, and S. Weber. Optimization of Motion Estimator for Run Time Reconfiguration Implementation. *7th Reconfigurable Architectures Workshop (RAW), part of the Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS) Workshops 2000*, May 2000.

[157] H. Taub and D.L. Schiling. *Principles of Communications Systems*. McGraw-Hill, 1986.

[158] TI. Texas Instruments: TMS320C6416T-1000 Fixed Point DSP. October 2004.

[159] S. Trimberger. Scheduling Designs into a Time-Multiplexed FPGA. *International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 153–160, 1998.

[160] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Mulitplexed FPGA. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

[161] Triscend. Triscend Corporation: A7S 32-bit Customizable System-on-Chip Platform. *Product Datasheet*, http://www.triscend.com/products/Documentation/dsa7csoc.pdf, August 2002.

[162] M. Vasilko and D. Ait-Boudaoud. Architectural Synthesis Techniques for Dynamically Reconfigurable Logic. *Field-Programmable Logic and Applications (FPL)*, pages 290–296, 1996.

[163] S. Verdu and S.W. McLaughlin, editors. *Information Theory: 50 Years of Discovery*. IEEE Press, 2000.

[164] J. Villasenor, B. Schoner, K.N. Chia, C. Zapata, H.J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith. Configurable Computing Solutions for Automatic Target Recognition. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 70–79, 1996.

[165] J. Villasenor, B. Schoner, and C. Jones. Video Communications Using Rapidly Reconfigurable Hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 565–567, 1995.

[166] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, September 1997.

[167] H. Walder and M. Platzner. Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform. *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 24–30, June 2002.

[168] J.L. Walsh. A Closed Set of Normal Orthogonal Functions. *American J. Mathematics*, 45:5–24, 1923.

[169] N. Weaver, J. Hauser, and J. Wawrzynek. The SFRA: A Corner-Turn FPGA Architecture. *Proceedings of IEEE Conference on Field Programmable Gate Arrays (FPGA)*, February 2004.

[170] M.J. Wirthlin and B.L. Hutchings. Sequencing run-time reconfigured hardware with software. *ACM Proceedings Fourth International Symposium on Field Programmable Gate Arrays*, pages 122–128, 1996.

[171] M.J. Wirthlin and B.L. Hutchings. Sequencying Run-Time Reconfigured Hardware with Software. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 122–128, 1996.

[172] M.J. Wirthlin and B.L. Hutchings. Improving Functional Density Through Run-Time Constant Propagation. *IEEE Symposium on Field Programable Gate Arrays (FPGA)*, pages 86–92, 1997.

[173] J.M. Wozencraft and I.M. Jacobs. *Principles of Communication Engineering*. Wiley, 1967.

[174] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM Computer Architecture News*, 13:20–24, March 1995.

[175] A.D. Wyner. Recent results in Shannon Theory. *IEEE Transactions on Information Theory*, 20:2–10, January 1974.

[176] Xilinx. *XC6200, Advance Product Specification*. San Jose, CA, 1996.

[177] Xilinx. *Virtex TM Configuration Architecture Advanced Users Guide*. San Jose, CA, 1999.

[178] Xilinx. Xilinx Unveils Revolutional FPGA Architecture, Enables Next-Generation Platform FPGAs. URL, December 2003.

[179] Xilinx. Virtex-4 Family Overview, DS112. 2004.

[180] Xilinx. *Virtex II Platform FPGA*, June 2004.

[181] G.I. Yayla, P.J. Marchand, and S.C. Esener. Speed and Energy Analysis of Digital Interconnections: Comparison of On-Chip, Off-Chip and Free-Space Technologies. *Applied Optics*, 37:205–227, January 1998.