Wire level encapsulation framework for increasing FPGA design productivity

Oliver, Timothy Francis

2009

Oliver, T. F. (2009). Wire level encapsulation framework for increasing FPGA design productivity. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/19266>

<https://doi.org/10.32657/10356/19266>

A WIRE LEVEL ENCAPSULATION FRAMEWORK FOR INCREASING FPGA DESIGN PRODUCTIVITY

Timothy Francis Oliver

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

2009

Acknowledgements

I would like to thank my supervisor Dr Douglas Maskell for his support, encouragement, and careful distillation of ideas during the production of this thesis.

I would like to thank Dr. Timo Bretschneider for his suggestions and valuable insights. I thank Dr. Bertil Schmidt for his inspirational drive for accelerated computing. Many thanks to Dr Chris Clarke for the lively discussions and generous hospitality. I thank Mr. Saurav Bhattacharyya for his motivational speeches, Mr. Mohit Sindhwani for a thousand conversations about nothing, Mr. Konstantin Melikhov for his brilliance, and Mr. Tobias Trenchel for plentiful coffee breaks and philosophical discussions. I would like to thank Ms. Nah Kiat Joo and all the people at CHiPES for their excellent help and support.

I would like to thank Dr. Ian McLoughlin, Dr. Lai Ming-Kit (Edmund) and Mr. John Rowe for helping me secure a place at NTU. I would like to thank my family and friends for supporting me from such a distance. I would like to thank Mrs Amy Jesudason for her encouragement and plentiful exam reminders and Mr Jeffrey Jesudason, who I probably owe a pint of pineapple juice.

Finally I would like to thank Miss Joanne Jesudason for her love, support, patience, and undying belief that I can succeed.

Table of Contents

1	Introduction.....	2
1.1	Problem Statement.....	3
1.2	Contributions.....	4
1.3	Journal Publications.....	5
1.4	Conference Publications.....	5
1.5	Organisation.....	6
2	Background.....	7
2.1	FPGA Technology.....	7
2.1.1	History.....	7
2.1.2	Silicon Manufacture.....	8
2.1.3	Computing Performance.....	8
2.1.4	Reconfigurable Computing Architectures.....	9
2.1.5	Discussion.....	12
2.2	FPGA Architecture.....	13
2.2.1	Overview.....	13
2.2.2	Computing Resource.....	13
2.2.3	Interconnect.....	14
2.3	Designer Productivity.....	17
2.3.1	Design Abstraction.....	17
2.3.2	Synthesis and Optimisation.....	18
2.3.3	High Level Language Synthesis.....	19
2.3.4	Packing and Placement.....	20
2.3.5	Routing.....	22
2.3.6	Design Reuse.....	24
2.3.7	Software Design Productivity.....	25
2.3.8	Compile Reuse.....	26
2.3.9	Discussion.....	27
2.4	Pre-Routed Component Encapsulation.....	27
2.4.1	Abstraction of Component Based Systems.....	28
2.4.2	Component Encapsulation.....	29
2.4.3	Communication Layer.....	31
2.4.4	Discussion.....	35
3	Framework.....	37
3.1	Introduction.....	37
3.2	Architectural Model.....	39
3.2.1	Basic Tile Structure.....	39
3.2.2	Tile Resource.....	40
3.2.3	Interconnect	41
3.3	Net-List Mapping	43
3.3.1	Placement	43
3.3.2	Routing	44
3.3.3	Interconnect usage.....	46
3.4	The Limits of Pre-Routing.....	47
3.4.1	Component Encapsulation.....	47
3.4.2	Design Definition Framework.....	49
3.4.3	Wire Identification.....	50
3.4.4	Wire Use Policy.....	50
3.4.5	Interface Definition.....	51
3.4.6	Component Connection Extensions.....	52
3.4.6.1	Interface Extension.....	52
3.4.6.2	Cornering Link Extension.....	53

3.4.6.3	Tunnelling Link Extension.....	55
3.4.7	Floor Planning and Component Shaping.....	56
3.5	Experimental Design Environment.....	58
3.5.1	Design Entry.....	58
3.5.2	Interconnect Graph Generator.....	59
3.5.3	Packer and Placer.....	59
3.5.4	Router.....	60
3.5.5	Discussion.....	60
4	Evaluation.....	61
4.1	Experimental Approach.....	61
4.1.1	Normal Approach.....	61
4.1.2	Pre-Placed Components Approach.....	61
4.1.3	Pre-Routed Components Approach.....	62
4.2	Synthetic Component Generation.....	62
4.3	Evaluation Metrics.....	64
4.4	Target Architecture Parameters.....	66
4.5	Target Architecture Characterisation.....	67
4.6	Interface Bandwidth Study.....	68
4.6.1	Interface Bandwidth Utilisation.....	69
4.6.2	Port Area Shaping.....	76
4.6.3	Component Region Shaping.....	78
4.7	Tunnelling bandwidth Study.....	79
4.7.1	Wire Reservation.....	80
4.7.2	Interfaces and Wire Reservation.....	83
4.7.3	Complementary Policies.....	84
4.8	Discussion.....	85
5	Application.....	88
5.1	Parallel System Design Methodology.....	88
5.2	Biological Sequence Database Scanning.....	90
5.2.1	Motivation for FPGA Acceleration.....	90
5.2.2	Previous Approaches.....	91
5.2.3	Sequence Comparison Algorithm.....	91
5.2.4	Application Scenario.....	93
5.3	Parallel Algorithm Design.....	94
5.3.1	Identifying the Parallelism.....	94
5.3.2	Query Length Scaling.....	97
5.3.3	Mapping to Xilinx Virtex-II Technology.....	99
5.3.4	Precision Scaling.....	101
5.3.5	Dynamic Precision Scaling.....	102
5.4	Mapping to Pre-Routed Components.....	103
5.4.1	Pre-Routing on Virtex-II.....	104
5.4.2	New Pre-Routing Approach.....	106
5.4.3	The Impact of Pre-Routing.....	108
5.4.4	Compiler Effort and Computation Time.....	111
5.5	Summary.....	112
6	Conclusions and Future Work.....	114
6.1	Conclusion.....	114
6.2	Future Work.....	116
7	Terminology.....	118
8	Glossary.....	120
9	Bibliography.....	121

Abstract

This thesis explores the performance impact of optimising the components of a Field Programmable Gate Array (FPGA) system down to the lowest level independently from other parts of the system. The motivation for this is that not only is the design and verification effort put in to a component reused, the optimisation effort expended in mapping, placement and routing is also reused.

FPGA technology has its roots in digital circuit design and, like every silicon technology, it advances every 18 months, doubling the gate capacity available to the designer. The single largest threat to this growth is the gap that is forming between the number of available gates and the ability of designers to use these gates in the time frame of a typical design cycle. The design gap is more acutely felt in the FPGA computing community since the main perceived strength of FPGA technology is its reconfigurability. As an example, High Performance Computing (HPC) on FPGA offers a clear advantage. However, designer productivity issues are a major threat to its wide spread use. HPC is achieved on FPGA by specialising the architecture. Specialisation implies a design process. Thus, designer productivity is the main restricting factor to increasing the computing functionality that FPGA systems can offer.

Reuse is recognised as a powerful approach to improving designer productivity. In the field of software design, reuse of third party software code is possible with both the static linking of pre-compiled libraries, and the dynamic linking of libraries during run-time. Even in the realms of Application Specific Integrated Circuit (ASIC) design, pre-placed and routed “hard” macros are available from third parties. However, currently implemented third party reuse schemes for FPGA design operate at either the source code or the net-list level. The full spectrum of compromises between flexibility and compositional effort have not been explored in previous works. Pre-routed FPGA components represent a reuse strategy that presents very low system composition effort at the cost of very little component flexibility. It is relatively simple to constrain component resource to regions on the surface of an FPGA device. The greater challenge lies in applying constraints on the interconnect usage of each component without adversely affecting system performance. There has been little work done to investigate the proposition of the third party reuse of pre-routed FPGA components.

In order to investigate the feasibility of pre-routed FPGA components, a detailed structural model generator for FPGA architectures has been developed along with a complimentary set of design automation tools. The elements of an FPGA architecture and automated tool behaviour affected by component encapsulation are identified. This leads to a design methodology, including modified tools, facilitating the automated mapping of components to an encapsulated region of FPGA resource and interconnect without interference from any other mapped region. The methodology supports automated construction of structures for communication between independently mapped regions.

A HPC bio-informatics application has been implemented to illustrate the high performance that is achievable using FPGA technology. This bio-informatics application is then used along with synthetic circuits to highlight the strengths and weaknesses of the proposed methodology. This evaluation leads to a set of guidelines for using the proposed design framework for wire level encapsulation.

1 Introduction

High Performance Computing (HPC) on Field Programmable Gate Array (FPGA) is a certainty with many documented cases of FPGA significantly exceeding the computing performance of Instruction Set Architecture (ISA) based systems. FPGA technology provides the fine grain flexibility to specialise the architecture to a given application, in many cases, yielding speed-ups of two orders of magnitude, or more. Furthermore, a single FPGA based system replaces many processors consuming less energy, producing less waste heat, allowing denser packing, and requiring less energy for cooling.

While the advantage of HPC on FPGA is clear, designer productivity issues are a major threat to its wide spread use. FPGA technology has its roots in digital circuit design and, like every silicon technology, it advances every 18 months, doubling the gate capacity available to the designer. The single largest threat to this growth is the gap that is forming between the number of available gates and the ability of designers to use these gates in the time frame of a typical design cycle.

While this design gap is true for Application *Specific* Integrated Circuit (ASIC) technology, it is more acutely felt in the FPGA computing community since the main perceived strength of FPGA technology is its reconfigurability. HPC is achieved on FPGA by specialising the architecture. Specialisation implies a design process. Thus, designer productivity is the main restricting factor to increasing the computing functionality that FPGA systems can offer. On top of this, the fine grain flexibility of FPGA provides a wider design space than a course grain reconfigurable architecture or ISA, requiring extra exploration effort during the design process in order to find an appropriate solution.

There have been many advances in digital system design automation in an attempt to keep designer productivity in step with available gate capacity. Broadly speaking, there are two approaches to improving designer productivity: the first is through the development of design descriptions and their automated translation and optimisation towards a form usable at the physical level; the second is in facilitating the reuse of design descriptions whether original or automatically created and optimised.

The first approach to improving designer productivity through design languages and compilers is based on abstraction. Abstraction is the process of generalisation by reducing information content to capture the detail relevant to an underlying view or model used to describe a digital system. As the size and complexity of digital systems have increased, so has the level of abstraction. A design description at the highest level of abstraction is translated and optimised at each of the lower levels of abstraction. Research into digital system design descriptions, their translation and optimisation is an active field of research and development. While automated tools perform these steps, too time consuming and error-prone for a human designer to attempt, there is still significant human expertise and effort required to iteratively tune each optimisation step to achieve closure on design goals. Translation and optimisation often requires a significant amount of computing effort to achieve a result of acceptable quality.

The second approach of facilitating reuse attempts to capture the effort that has been expended in the detailed design and verification of the design description of a component, so that it will not have to be repeated. Thus, the design cost of a component is amortised across multiple uses. The more a component is reused, the better justified it is to expend extra effort in optimising it. Design reuse is encouraged through two mechanisms: The first is through wider applicability of a design component; The second is through lower integration effort of reusable components. Both require the consideration of reuse during component identification, specification and development.

The effort in system composition can be quantified in terms of the computation required to move from the set of component descriptions to final system representation. Compositional effort is required to support parametrised design descriptions and thus the two key aspects that encourage reuse, component applicability through flexibility and ease of integration, oppose one another. Choosing the design representations, translation and optimisation operations supported within a composition environment provides many a trade-off in terms of flexibility, performance, resource utilisation, and integration effort. The composition of a system is a one-off cost, and is thus under a greater pressure to reduce effort. If optimisation is removed from the process of system composition then the uncertainty in achievable performance is also removed. However, partitioning a design and forcing isolated compilation of its individual parts removes the opportunity to cross-optimize between components. In other words, there will always be a loss in performance when a system is partitioned.

It could be argued that, in the field of ISA based systems, software designer productivity has been the key factor considered over any other, including performance. Certainly the techniques to increase designer productivity are well developed in the field of software based systems. High Level Languages (HLL) are used within a well understood behavioural machine model. Reuse of code is realised through several mechanisms: sub-routine calls, software libraries, and operating system services. Object-oriented concepts are built into the programming languages to make reuse easier. The dynamic allocation and freeing of memory is also built into the language making software systems more flexible. The reuse of third party code is possible with both the static linking of pre-compiled libraries, and the dynamic linking of libraries during run-time.

Even in the realms of ASIC design, pre-placed and routed “hard” macros are available from third parties. Pre-tested, the black box nature of third party components frees the integrator from concerns over their content and correctness. A documented interface is all that is exposed to the designer to facilitate integration into their system.

1.1 Problem Statement

Currently implemented third party reuse schemes for FPGA design operate at either the source code or the net-list level. The full spectrum of compromises between flexibility and compositional effort have not been explored in previous works. Pre-routed FPGA components represent a reuse strategy that presents very low system composition effort at the cost of very little component flexibility. There has been little work done to investigate the proposition of the third party reuse of pre-routed FPGA components. While it is possible to reuse pre-routed portions of a design in FPGA development environments, it is more for the purpose of incremental design. Thus, there is not a well developed facility to define interfaces and component constraints that would make it possible to create

compatible third party components. Without such a design environment it is difficult to study the impact of imposing the constraints necessary for low overhead composition. The few FPGA design environments that do support the composition of pre-routed components have been in the field of dynamically reconfigurable systems. These design environments have limits in both the compositional flexibility they offer and the communication between components. Thus, existing solutions have a significant impact on the performance achievable in a given area of FPGA resource. The restrictions placed on the creation of communication structures by the available FPGA design tools only make systems composed of a handful of pre-routed components feasible. Furthermore, the location and size of these regions have to be fixed. Inflexibility in the size of component regions leads to the fragmentation of the FPGA surface, lowering resource utilisation.

Generally speaking, computing performance is limited by the inherent communication overheads. Inefficiencies in the transport of data limit the number of useful computing circuit-cycles that it is possible to perform. The FPGA compiler tools create a usable abstraction of the architecture. For this reason the tools and architecture of FPGA technology are developed together to achieve the best combination of compile time and mapped circuit performance. The interconnect facilitates all communication in an FPGA based system and so poor abstraction of the interconnect within the compiler tools will result in poor computing performance overall. Thus, there has to be a focus on wire level detail and how it is abstracted throughout the compilation environment to ensure the efficient implementation of pre-routed components.

As it is not possible to freely modify the existing commercial tools, an exploration into the detailed nature of FPGA architecture and associated mapping tools has been conducted, leading to the development of an FPGA model and supporting mapping tools that can be readily modified to explore various component encapsulation techniques. This provides a background into the study of how best to implement a pre-routed component framework at the physical layer. The physical implementation is then abstracted up to a usable representation supported within an automated component compilation environment.

1.2 Contributions

A detailed structural model generator for FPGA architectures has been developed along with a complementary set of design automation tools.

The elements of an FPGA architecture and automated tool behaviour affected by component encapsulation are identified. This leads to a design methodology, including modified automated mapping tools, facilitating the automated mapping of components to an encapsulated region of FPGA resource and interconnect without interference from any other mapped region. The methodology supports automated construction of structures to support communication between independently mapped regions.

A bio-informatics computing application has been implemented to illustrate the high performance that is achievable using FPGA technology. This bio-informatics application is then used along with synthetic circuits to highlight the strengths and weaknesses of the proposed methodology. This evaluation leads to a set of guidelines for using the proposed design framework for wire level encapsulation.

1.3 Journal Publications

- 1) T.F. Oliver, B. Schmidt, J. Yanto and D.L. Maskell, "High-Speed Biological Sequence Analysis with Hidden Markov Models on Reconfigurable Platforms", *IEEE Trans. Information Technology in Biomed.*, doi:10.1109/TITB.2007.904632, 7 pages, 2007.
- 2) T.F. Oliver. and D.L. Maskell, "Pre-Routed FPGA Cores for Rapid System Construction in a Dynamic Reconfigurable System", *EURASIP Journal on Embedded Systems*, doi:10.1155/2007/41867, 2007.
- 3) T.F. Oliver, B. Schmidt, D.L. Maskell, D. Nathan and R. Clemens, "High-speed multiple sequence alignment on a reconfigurable platform", *Int. J. Bioinformatics Research and Applications*, 2006.
- 4) T.F. Oliver, B. Schmidt and D.L. Maskell, "Reconfigurable Architectures for Bio-sequence Database Scanning on FPGAs", *IEEE Trans. Circuits Syst. II*, Vol. 52, pp. 851-855, Dec, 2005.
- 5) T.F. Oliver, B. Schmidt, D. Nathan, R. Clemens and D.L. Maskell, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW", *Bioinformatics*, Vol. 21, pp. 3431 – 3432, Aug., 2005.

1.4 Conference Publications

- 1) T.F. Oliver and D.L. Maskell, "Execution Objects for Dynamically Reconfigurable FPGA Systems", *IEEE Int. Conf. Field Programmable Logic and Applications*, Madrid, Spain, Aug., 2006.
- 2) T.F. Oliver, B. Schmidt, Y. Jakop and D.L. Maskell, "Accelerating the Viterbi Algorithm for Profile Hidden Markov Models Using Reconfigurable Hardware", *Int. Conf. Computational Science*, pp. 522-529, Reading, UK, May, 2006.
- 3) T.F. Oliver, B. Schmidt, J. Yanto and D.L. Maskell, "Accelerating the Viterbi Algorithm for Profile Hidden Markov Models using Reconfigurable Hardware", *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 3991, pp. 522-529, 2006.
- 4) T.F. Oliver and D.L. Maskell, "An FPGA Model for Developing Dynamic Circuit Computing", *IEEE Field-Programmable Technology*, Singapore, Dec., 2005.
- 5) Y.S. Lee, T.F. Oliver and D.L. Maskell, "Reconfigurable Computing: Peripheral Power and Area Optimization Techniques", *IEEE TENCON*, Melbourne, Australia, Nov., 2005.
- 6) J. Yanto, T.F. Oliver, B. Schmidt and D.L. Maskell, "Biological Sequence Analysis with Hidden Markov Models on an FPGA", *Lecture Notes in Computer Science*, Springer-Verlag, Vol 3740, pp. 429-439, Oct., 2005.
- 7) J. Yanto, T.F. Oliver, B. Schmidt and D.L. Maskell, "Biological Sequence Analysis with Hidden Markov Models on an FPGA", *Asia-Pacific Computer Systems Architecture Conference*, Singapore, Oct., 2005.
- 8) T.F. Oliver, B. Schmidt, D. Nathan, R. Clemens and D.L. Maskell, "Multiple Sequence Alignment on an FPGA", *HiPCoMB 2005*, Fukuoka, Japan, July, 2005.
- 9) T.F. Oliver, B. Schmidt, D.L. Maskell and A.P. Vinod, "A Reconfigurable Architecture for Scanning Biosequence Databases", *IEEE Int. Symp. Circuits and Systems*, Kobe, Japan, pp. 4799-4802, May, 2005.
- 10) T.F. Oliver, B. Schmidt and D.L. Maskell, "Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs", *ACM Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, Feb., 2005.

- 11) T.F. Oliver, S. Mohammed, N.M. Krishna and D.L. Maskell, "Accelerating an Embedded RTOS in a SOPC Platform", *IEEE TENCON*, Chiang Mai, Thailand, Nov., 2004.
- 12) M. Sindhvani, T.F. Oliver, D.L. Maskell and T. Srikanthan, "RTOS Acceleration Techniques - Review and Challenges", *Sixth Real-Time Linux Workshop*, Singapore, pp.123-128, Nov., 2004.
- 13) T.F. Oliver and B. Schmidt "High Performance Biosequence Database Scanning on reconfigurable Platforms", *IEEE Int. Parallel and Distributed Processing Symp.*, Santa Fe, NM, 2004, Apr., 2004.
- 14) T.F. Oliver and D.L. Maskell, "Towards run-time re-configuration techniques for real-time embedded applications", *Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, pp. 141-146, Jun., 2003.

1.5 Organisation

Chapter 2 identifies FPGA technology as a suitable platform for high performance computing. Design productivity is identified as a barrier to further adoption leading on to a review of current approaches to increasing productivity. The partitioning and reuse of components within a digital system is identified as worthy of further investigation. A review of existing component encapsulation techniques leads to potential for improvement in the flexibility and usability of encapsulation frameworks.

Chapter 3 starts with a detailed study of FPGA architecture. The architectural detail lays the foundation for the proposal of efficient ways to partition FPGA resources. A framework that supports independent construction of pre-routed components is described in detail. The theoretical limitations of the proposed approach are explored. A modelling environment that is representative of modern FPGA architectures is constructed, along with a complimentary set of design automation tools.

In chapter 4 the experimental design environment is used to compare the quality of compiled designs between the newly proposed approach and existing approaches.

Chapter 5 presents a high-performance computing application from the bio-informatics domain. The algorithm is mapped to FPGA, achieving a significant performance improvement. We then apply our newly developed pre-routed component framework to the bio-informatics application.

Chapter 6 presents the conclusions of this study and explores future research directions based on the work presented here.

2 Background

2.1 FPGA Technology

2.1.1 History

In the spring of 1959 John Pasta, a highly respected applied mathematician and physicist, raised concerns that many vital computational problems were not solvable by existing electronic computers. In his opinion, commercial computer manufacturers had lost interest in exploring risky, innovative computer architectures. Instead, the manufacturers wanted to serve the growing market for conventional computer systems. He expressed this concern to one Gerald Estrin, who had recently come from Von Neumann's Electronic Computer Project at the Institute for Advanced Study in Princeton to start work at UCLA supported by the Department of Mathematics' Numerical Analysis Research Laboratory.

Estrin [Estr05], addressed this concern by proposing the fixed plus variable structure computer architecture. Starting in 1959, his team designed special-purpose subsystems that could run concurrently with programs in a coupled general-purpose computer. They developed circuit modules that had removable, replaceable etched signal harnesses. His team was well aware of the gap between available technology and the need for automatic electronic control. Many computer science and computer engineering issues would have to be dealt with if reconfigurable systems were to be realisable.

The invention of the FPGA, while built to serve the rapid prototyping market, makes automatic electronic control of fixed plus variable computing architectures possible.

In 1986 when FPGA devices first appeared they were able to implement only thousands of system gates [Tess01]. In order to create high performance computing platforms many FPGA devices had to be coupled together. An example of such a system was the Splash-2, which coupled together 272 devices on 16 boards each with 17 XC4010 devices providing a total of around three million system gates [Arno92]. A later example is the TM-2, which had 16 boards each with two Altera 10K100 devices also providing around three million system gates [Lewi98].

Early FPGA based computing did not exhibit a compelling price performance ratio [Ebel97]. The use of FPGA technology for computing applications was not competitive because ISA technology better fitted a restricted silicon area [DeHo00]. However, neither cost nor capacity is a limiting factor for modern FPGA technology. Now FPGA devices use a large silicon area to support tens of millions of system gates [Xilds100], while sequenced instruction architectures are not able to make efficient use of the expansive space available on today's silicon die. Steady improvements in FPGA technology have led to modern, low cost, high-capacity FPGA devices that are able to provide a higher performance per unit cost than available processors.

2.1.2 Silicon Manufacture

ASIC solutions must overcome the twin hurdles of high Non Recurring Engineering (NRE) costs and long time-to-market. Each new generation of semiconductor technology is increasing the NRE costs and increasing the design and verification effort required to realise an ASIC device. Mask costs alone have exceeded \$1 million in today's technologies, and are expected to double or triple with every new technology generation [Guo04]. Therefore, ASIC implementation is only feasible for high volume applications and products that can tolerate a long time-to-market. Thus, ASIC implementation is becoming prohibitively expensive for the kind of product customisation and differentiation that is required by modern computing systems [Yala02]. In contrast to ASIC devices, which can undoubtedly provide a higher computing performance per unit of silicon area, an FPGA is a truly *generic* platform for high performance computing.

As device capacity increases, single FPGA platforms will see wide spread use in computing systems. The latest FPGA devices provide tens of millions of system gates in one chip [Xilds100]. The move to the 65nm technology generation has increased FPGA performance, reduced power and doubled capacity when compared to 90nm devices [Xilds100], [Altep3s].

2.1.3 Computing Performance

In contrast to the ISA, the primitive level programmability of FPGA technology provides the ability to tailor the hardware to the required algorithm. FPGA devices are reconfigurable at the primitive level. Circuits are built up on FPGA by setting bits that connect wire segments together. This provides a high degree of flexibility so that almost any digital circuit can be implemented.

The peak computational density metric [DeHo00] is a performance indicator measured in bit operations per area-second. The computational density metric is a useful illustration of both FPGA advantage and for deciding on architectural enhancements to FPGA. It was shown that FPGA technology is an order of magnitude above ISA technology in terms of this metric [DeHo00].

Many applications benefit from the superior performance in an FPGA-based computing system over an ISA-based system. The fine grain flexibility of FPGA devices give them a strong advantage when implementing bit level algorithms. For example, FPGA implementations of the data encryption standard (DES) enjoy a speed up of two orders of magnitude over software implementations. The core operations are bit-level substitutions and permutations, which are efficiently implemented on FPGA [Patt00]. Typical video applications implemented on FPGA enjoy a speed up of 20 to 100 times over processor implementations [Guo04]. These impressive improvements in performance are realised in a highly specialised architecture because the precision of each data-path is reduced to the minimum necessary in the application [Oliv05], [Aziz04], [Fend04].

The reconfigurability of FPGA can be used to further improve the computing performance of applications using FPGA. The FPGA is dynamically reconfigured to create a circuit that is highly specialised to the parameters of an application. One example of a clear advantage when using reconfiguration is a DES encryption design that achieves a three times increase in performance over a conventional static FPGA design. The reconfigurable FPGA implementation had a higher performance than ASIC implementations available at that time [Patt00].

FPGA implementations of image processing applications take advantage of the inherent parallelism to achieve large speed-ups over microprocessors. For example, using run-time reconfiguration (RTR) in a binary pattern matching circuit resulted in a 50% area reduction and a 52% performance improvement over a static FPGA version. Only half the number of comparators are required to do any one match and the match pattern storage is not required as it is explicitly embedded in the circuit structure [Shir98]. Another image processing application combining a 1D Sobel image edge detector and a 1D Gaussian image filter using RTR exhibited a 29% area reduction and 22% performance improvement as compared to a static solution that combines the same function using in-circuit path switching. In the static solution there is an increase in path fan-out and an increase in the number of multiplexed paths [Shir98].

A finite impulse response (FIR) filter implementation using a single multiply accumulate (MAC) stage and employing RTR to change tap values showed a 37% improvement in clock speed as compared to a static design [Hero01].

High-performance rule processing systems are needed by network administrators in order to protect Internet systems from attack. Software-based implementations of complex rule processors have been shown to be too slow for processing data in high-speed networks. Hardware parallelism greatly improves performance. Through the use of reconfigurable FPGAs, rule processing systems can be deployed that provide the flexibility to adapt to the persistent changes required of an effective network defence mechanism [Atti05]. Lockwood et al. [Hort02], developed the Field programmable Port eXtender (FPX), a FPGA based network processing system.

Many researchers have examined FPGA implementations of the Smith-Waterman pairwise protein sequence alignment algorithm. A static implementation is 120 times faster than a Pentium-4 implementation [Oliv05]. Changing the precision of the processor from 14-bit to 10-bit increases the performance by 35%. Predicting the maximum precision necessary and dynamically adjusting it while scanning a protein database achieves an additional 6% performance improvement with no loss in accuracy [Oliv05].

Reconfiguration also provides an advantage in irregular architectures such as peripheral interfaces. Previous work has shown that converting multi-function cores to several reconfigurable FPGA cores resulted in core area reduction of around 21% and a simultaneous performance increase of 14% [MacB01].

2.1.4 Reconfigurable Computing Architectures

High performance computing on FPGA is achieved by exploiting the massive inherent parallelism and by using flexibility to specialise the architecture. A significant advantage over ISA devices is achieved because FPGA resources are configurable. Pre-computed lookup tables are used in place of complex mathematical functions. Issues of memory bandwidth are alleviated by having access to multiple memory banks in parallel. The use of custom memory access circuitry maintains close to peak utilisation of the available memory interface bandwidth. Using these techniques, the performance of irregular computing problems experiences a speed-up of up to 30 to 40 times over a high-end workstation [Abra98]

If an application requires a very addition rich mixture of instructions, the FPGA can provide that. In contrast, commodity processors have a fixed allocation of resources that can be successful for a well mixed instruction stream, but has significant limitations in code that is dominated by one instruction [Unde04].

In many software applications only a small portion of an application becomes the performance bottleneck that contributes most to total computation time. The larger part of the code is necessary for completeness, but its execution speed does not limit performance. Processors are designed to handle a rich mix of operations and custom computing circuits are able to accelerate a well defined set of repetitive operations. Consequently, an interesting hybrid approach couples a processor with an FPGA like fabric [DeHo00].

This hybrid approach has seen several RTR systems proposed that augment a general purpose CPU architecture with reconfigurable resources. An early example of this is the dynamic instruction set computer (DISC) [Wirt95]. DISC is composed of a static controller circuit and dynamically placed instruction modules.

The “Garp” architecture [Haus00], is designed to accelerate computationally intensive kernels of a program running on a general purpose MIPS-II architecture using custom hardware instantiated in the reconfigurable space. Operations to copy values between the register file of the CPU and the reconfigurable space are added to the MIPS-II instruction set. This tight integration of a general purpose ISA with reconfigurable space allowed a C compiler to be developed that maps compute intensive kernels to hardware [Call00].

Chimaera [Ye00], places a reconfigurable space within the pipeline of a dynamically-scheduled super scalar processor. MIPS is used as the base ISA and a specialised C compiler packs control and data operations into the reconfigurable space. The C compiler maps, on average, 22% of instructions to the reconfigurable space. A performance increase of 28% over a 4-way out-of-order super scalar architecture was reported .

Triscend [Tris5], [Tris7], developed a family of combined ISA and FPGA on a single chip. The A7 and E5 families combine an ARM processor and 8051 processor respectively with configurable system logic (CSL) that has a density on par with FPGA devices at the low end of the Xilinx Virtex and Virtex-II families.

Quicksilver Technology realised that algorithms are heterogeneous in nature making homogeneous FPGA architectures inappropriate for many algorithmic tasks. They proposed the adaptive computing machine (ACM) composed of five types of nodes: arithmetic, bit manipulation, finite state machine, scalar processor, and configurable input/output. Each node consists of a number of computational units, which can be adapted on the fly. Unlike FPGAs, an ACM has a 128-bit or 256-bit bus dedicated to device configuration. A proprietary design environment was developed including tools to map C programs to the architecture [Plun04].

Stretch introduced the S5 architecture consisting of a conventional 32-bit RISC processor coupled with a programmable instruction set extension fabric (ISEF). Stretch's C compiler is able to compile a C/C++ program to the processor and automatically configure the ISEF with application-specific instructions. To develop an application for the S5 the programmer identifies critical sections to be accelerated, writes one or more extension

instructions as functions in a variant of the C programming language, and accesses those functions from the application program. Performance gains of more than an order of magnitude over the processor alone have been achieved [Stre08].

An increasing number of new chips include a portion of FPGA-like space to add the flexibility to adapt to a number of applications. This trend is due to companies such as M2000 who market FPGA fabric for integration into new ASIC devices. Thus, allowing a chip manufacturer to provide a combination of application specific cores coupled with reconfigurable logic [m2000]. Both the Quicksilver and Stretch devices exhibit a small amount of reconfigurable space, multiple configuration planes, and high bandwidth configuration interfaces. As the use of this technique matures the proportion of silicon area devoted to reconfigurable space is expected to scale up. Therefore low overhead partial reconfiguration is likely to be necessary. The amount of configuration data that is required to set up large parallel structures will become the limiting factor.

While many hybrid architectures have been proposed, most have had little commercial success. As of writing the only commercial hybrid device still existent in the market is the Stretch S5. The mix of paradigms has the effect of mixing the complexities of application design for software and hardware. In order to adopt such a device, a company has to use specialised compiler tools that are as yet unproven. The benefit of using such a device does not outweigh the risk inherent in complex systems that are only supported by a single vendor.

The FPGA vendors have had more success introducing processors embedded inside their FPGA fabrics. Examples of commercially available platforms are the Xilinx Virtex-2 Pro with embedded PowerPC [Xilds083], and the Altera Excalibur with embedded ARM [Altex02]. In contrast to the hybrid architecture vendors mentioned previously the FPGA development environments of Altera and Xilinx are well established. They have built on this advantage and integrated processor cores that have well established programming environments.

FPGA technology also offers soft-core processors that are cheaper and lower risk than hybrid architectures. Soft-core processors use standard FPGA resources to build microprocessor architectures. They are low risk to the FPGA vendor because they do not have to increase their product inventory with special chips. They are low risk to the customer because they do not need to buy a specific part. Furthermore, the customer has a range of soft-core processors to choose from. As well as vendor specific cores such as the Xilinx Microblaze [Xilm08], Altera NIOS-2 [Altni02], there are open source offerings such as the LEON SPARC [Gais08], and Opencores OpenRISC [Open08]. An FPGA vendor's soft-core can be placed on any one of its FPGA devices providing different logic capacities. Third party and open-source soft-core processors have the added flexibility of being portable between FPGA vendors. In addition to being low risk and portable, soft-core processors offer numerous opportunities for performance improvement. NIOS-2 and OpenRISC offer the facility to add custom instructions [Oliv04], with options to choose cache size and bit-width [Altni02], [Open08]. It is possible to instantiate several soft-cores on the same FPGA to create a multiple processor array with custom interfacing logic [Hoar04]. Altera [Altni02], appear to be de-emphasising their embedded ARM core and instead are promoting the NIOS-2 as a more flexible alternative.

As FPGA technology has developed, its use as a computing platform has moved from the academic research space to the commercial domain. Companies such as Starbridge Systems [Star05], and Nallatech [Nall08], offer high performance computing platforms based on FPGA. Further maturity is demonstrated by the fact that FPGA technology is being adopted by high profile super computing companies such as Cray with their XD1 platform [Cray05], and SGI with their RASC platform [SGI08]. All of these offerings are targeted at the high-performance computing domain and thus carry a very high price tag. It has been said that platforms that are affordable to home and small to medium enterprise (SME) users will never appear because they do not provide a large enough profit margin to amortise the cost of development. However with new lower cost parts appearing and the design software maturing this view may change.

To some extent we are seeing this now with AMD's new strategy to improve the potential of their Opteron processor based platforms by opening up the HyperTransport specification [Hype08]. AMD realised that processors will not continue to provide incremental performance improvements achieved over previous years. In opening up HyperTransport they hope application specific accelerators closely coupled with their processors will provide platforms that outperform their competitors. In response to this two companies have created FPGA platforms that slot in to an available Opteron slot on a multiple processor motherboard. DRC offers a re-configurable Xilinx Virtex-4 platform [DRC06], and Xtreme Data offer an Altera Stratix-2 platform [Xtre06]. Placing the FPGA at this point in the system provides a 5.4 GB/s link with up to 4 GB of memory and a 1.6 GB/s link to other AMD Opteron processors on the motherboard [Xtre06]. In response, Intel Corp. has also provided the ability to connect to its Xeon processor front side bus (FSB) and the *QuickAssist* accelerator abstraction layer. Xtreme Data offer an Altera Stratix III FPGA based module which targets the Intel FSB at 1066 MHz. The price of these accelerators is equivalent to a high-end graphics card and will most likely drop in line with the cost of FPGA devices.

2.1.5 Discussion

There are clear examples of the superior performance that is achieved when FPGA technology is used for specialised computing applications. Since it is usual that the small market size for these applications does not provide sufficient revenue to fund the development of a new ASIC for each new technology generation, FPGA technology is an appropriate choice.

Even though early FPGA devices were expensive, they provided a better price-performance ratio than processor-based systems. The increased use of FPGA technology has resulted in the total market expanding. In order for FPGA technology to reach wider markets, vendors are now reducing cost to be competitive with other solutions. This has the effect of reducing the barrier to entry into using reconfigurable FPGA technology and further extends their price-performance advantage.

The inherent compute density of reconfigurable devices has allowed them to emerge as a flexible, high-performance component in computing systems. The fact that they provide a performance increase for a wide range of applications suggests that they are indeed a general-purpose computing platform. The power of reconfigurable systems lies in the immense amount of flexibility that they provide.

The use of embedded hard macros further increases the compute density of FPGA devices and lowers energy consumption. A greater variety of embedded hard macros cores are appearing as both the capacity of FPGA devices and application markets grow. In order to handle a complex mix of operations, FPGA fabrics have been coupled with microprocessors in the same chip. Devices with large and already established FPGA fabrics have had more success in the market than more processor centric offerings or those that use a new proprietary reconfigurable array.

The FPGA compiler tools create a usable abstraction of the architecture. Thus, the tools and architecture of FPGA technology are developed together to achieve the best combination of compile time and mapped circuit performance. The features of FPGA architecture are reviewed in the next section in order to provide a context for the review of design tools in the subsequent section.

2.2 FPGA Architecture

2.2.1 Overview

In order to facilitate scalability in both design and manufacture, FPGA devices are constructed from a set of layout tiles. Using a repeated tile pattern simplifies the design with the tiles reused throughout a family of devices. Furthermore, the tiles create a regular pattern that is easy to fabricate in silicon increasing the yield and reducing cost.

2.2.2 Computing Resource

The classic FPGA architecture is a homogeneous two-dimensional array of logic clusters. Each cluster is composed of several look-up tables (LUT) and configurable flip-flops (FF). The array of logic is surrounded by a ring of input output blocks (IOB) to facilitate the transfer of signals on and off chip.

While almost any logic function can be implemented in such a homogeneous array of logic, it has been shown that the computational density of an FPGA fabric can be increased by including more specialised logic resources in the array.

Two situations undermine the increased computational density provided by a specialised resource unit: Either it is under utilised in a chosen application or its design is too general purpose for a given task. An unnecessary resource unit takes up space without providing any computing capacity; in the extreme case, its inclusion diminishes computational density and effects power dissipation. Attempts to avoid these effects result in conflict. To make sure we can use a specialised resource unit as much as possible, it is generalised. But the more it is generalised, the less suited it is for solving a particular problem, and the less advantage it offers over a configurable solution [DeHo00].

As the use of FPGA technology has spread, analysis of various markets shows that similar operations are desired across many application domains. This allows FPGA vendors to strategically add embedded hard macro units to improve the overall compute density. One of the first hard-core elements to be added was larger memory blocks of 4Kbit capacity [Xilds003], [Altep3a]. The benefit is obvious when one considers that although a Virtex-2 18Kbit RAM uses 28 times the area of a single CLB it has the memory capacity of 144 CLBs [Beau06]. Memory is required in almost every application. Using them for state-

machines, processors [Xilp08], and logic [Wilt97] reclaims the compute density lost to applications that do not use them.

The growing market for FPGA devices in digital signal processing (DSP) applications has prompted vendors to add embedded multiplier hard macros [Xilds031], [Altep1s]. Although the Virtex-2 embedded multiplier consumes 18 times the area of a CLB, it provides an approximate net saving of 82 CLBs and has a higher performance than a multiplier implemented in CLBs [Beau06]. Multipliers have been augmented with extra circuitry to implement multiply accumulate (MAC) functionality as well as various arithmetic and checking operations [Xilds100]. In order to maintain compute density over a wider number of applications the use of multipliers for other functions such as a dynamic shift operation has been proposed [Xilds100].

Now the diversity of FPGA applications has become so wide, no single architecture is able to adequately serve every application domain with equal efficiency. While Xilinx designed the Virtex-II to serve the general market, the Virtex-II Pro was developed for applications that required either embedded 32-bit RISC processor hard macros or Multi-Gigabit Transceiver (MGT) hard macros. This strategy became more pronounced with three members of the Virtex-4 family being developed. The LX for high-density logic applications, the SX for signal processing applications and the FX for embedded processor and high-speed communications applications [Xilds112]. Altera provide a similar range for their Stratix-III family with the L devices focusing on logic-rich applications, the E devices focusing on DSP and memory-rich applications, and the GX devices for applications with MGT requirements [Altep3s].

2.2.3 Interconnect

The computing resources on an FPGA are connected using a programmable interconnect matrix. This generally consists of horizontal and vertical wires gathered into channels between each row and column of resource. The maximum number of signals a given channel can carry is dictated by the number of wires in the channel, W_{FPGA} .

The choice of wire lengths used in the interconnect effects the performance of the FPGA. Complete connectivity is achievable using an interconnect composed entirely of wires that span just two tiles. However, the switches and buffers associated with each wire segment carry both an area overhead and introduce delay to the signal they carry. Thus, both FPGA area and wire delay is reduced using wires that span more than one tile. Contrary to this, a signal traversing a wire that is longer than the distance it needs to travel is inefficient. A typical digital circuit will contain nets that have different fan-outs and travel different distances. By matching the mix of wire lengths in an FPGA interconnect to the distribution of fan-outs found in a digital circuit the area-delay product is minimised. An FPGA architecture is not optimised to one particular circuit, instead a wire mix is chosen that gives a minimum area-delay product over a set of benchmark circuits [Betz99].

The Xilinx XC4000 architecture has a wire length mix that is approximately 25% length 1, 12.5% length 2, 37.5% length 4, and 25% of wires that span one quarter of the device. Betz et al. [Betz99], used a detailed FPGA architecture model and complementary placement and routing tools to investigate both the optimal wire mix and the mix of switch types. They found that an architecture with 83% length 4 wires, 17% length 8 provided a 10% improvement in area-delay product over the approximate XC4000 model.

Rather than using a benchmark set, Singh et al. [Sing02], derived their wire mix from Rent's rule. They found that an architecture with 20% length 2, 67% length 4 and 13% length 8 provided a 29% improvement in area-delay product over the architecture previously proposed by Betz et al.

Internal depopulation of connection points along wires that span more than one tile provides an area and delay saving. Long lines span the entire device and connect to a number of resources along their length. The concept of device spanning long lines gives rise to scalability issues as the gate capacity of FPGA technology increases. Device spanning long lines are no longer used in the Virtex-4 and Virtex-5 architectures.

The Xilinx Virtex architecture uses a wire mix of 22% length 2, 65% length 7 depopulated to 3 connection points and 13% long lines [Xilds003]. The Xilinx Virtex-II architecture has similar wire mix of 5% length 2, 21% length 3, 61% length 7 depopulated to 3 connection points and 13% long lines [Xilds031].

For ease of design, silicon implementation and testing, FPGA interconnect fabrics are constructed from a single layout tile [Lemi04]. Wires that span more than one resource tile must be stepped or twisted to facilitate construction from a single tile [Betz99a], [Lemi04]. An interconnection box is placed at the intersection of each horizontal and vertical wire channel. Previous work has a separate connection box to connect resource pins to channel wires and a switch box to connect between X and Y channel wires [Betz99a]. Xilinx combine these two into a single interconnect box.

Interconnect box *switch flexibility*, F_s , is defined as the number of other wires connecting to any given wire through programmable switches [Rose91]. The lower the value of F_s , the fewer programmable switches that are required which results in less silicon area. The lowest F_s that provides a routable interconnect is widely considered to be $F_s = 3$ for wire end points and $F_s = 1$ for wire midpoints [Betz99a], [Wilt97], [Imra99]. In the classic interconnect box model, each wire connects to three other wires via an optionally buffered pass transistor. This results in six Programmable Interconnect Points (PIPs) for each wire in the channel. *Input connection flexibility*, F_{CI} , is the number of wires to which a resource input can be connected [Betz99a]. Typical values for F_{CI} are between 8 and 16. *Output connection flexibility*, F_{CO} , is the number of wires to which a resource output can be connected [Betz99a]. A typical value for F_{CO} is around 50% of the wires in the channel.

The *diversity* of an FPGA interconnect is defined as the ability of a set of interconnect boxes to route two different signals such that they reach two different wires on the same destination channel, forming two disjoint paths. Lemieux et al. [Lemi02], use the diversity as a metric in an analytical framework for interconnect design that considers each interconnect box as a part of the overall routing fabric. The disjoint interconnect box does not exhibit any diversity. The universal and hyper-universal interconnect boxes were analytically designed to be independently routable for all two-point nets and for multi-point nets respectively [Chan96], [Fan01]. However, in order to provide good diversity they rely on reordering nets at every interconnect box. While it is possible to create a highly diverse and PIP efficient interconnect fabric by defining different switch patterns specified at each interconnect box, a fabric with such variety would be difficult to manufacture and test. The Wilton interconnect box is able to provide diversity by changing the wire set index as a net turns through the box [Wilt97]. The Imran box

provides the same level of routability as the Wilton box with a reduced number of PIPs [Imra99]. The Wilton and Imran interconnect boxes are still able to provide diversity even with the restriction that every interconnect box must be identical. However, F_s must be increased in a single tile architecture to achieve the same level of diversity as a non-uniform interconnect.

A fully buffered interconnect achieves an area-delay saving over an unbuffered interconnect [Lewi03], [Lemi04]. In a fully buffered interconnect, the drive strength of the buffers is adjusted to provide an almost equal delay along each of the different wire lengths present in an architecture. Thus, delay estimation is based on the number and type of wires used rather than fanout and Manhattan distance as used in ASIC and earlier FPGA technology. Wang et al. [Wang03], estimate post routing delay at the placement level using the number and types of routes resulting in an average error in prediction that is within 6%.

Early local interconnects used multi-directional wires. Each wire could be driven from any tile that it passed through and could drive a resource input or another wire in the tiles it passed through [Betz99a]. It has been shown that unidirectional routing fabrics are superior to those that are bi-directional and multi-directional [Lemi04]. Overall, an average area savings of 25% is realized with directional, single-driver wiring. The area-delay product savings is 32% on average, and ranges from 23% to 45%. Circuits with wider interconnect channels obtain better savings. Further, wiring capacitance is reduced by 37% due to reduced switch loading and physical wire length shrinkage [Lemi04]. A fully buffered interconnect allows a simplified timing model where path delay is closely related to the number and types of wires used [Wang03]. Modern commercial architectures use fully buffered unidirectional wires [Betz05], [Xilds031], [Xilds112].

Unidirectional wires also allow an increase in switch flexibility for the same area and delay as a fabric that uses bi-directional wires [Lemi04]. Wire directions must be fixed in the architecture and commercial FPGA devices and academic models use a directional mix that is close to 50%. Unidirectional wires are only driven at an endpoint. Thus, resource outputs and wire midpoints are only able to drive wire endpoints. In previous interconnect models, resource outputs and wire midpoints could drive both wire midpoints and endpoints. It was found that a unidirectional interconnect with $F_s = 8$ has a superior area-delay product when compared to a bidirectional interconnect with $F_s = 4$ [Lemi04].

Altera Stratix devices use direct drive multiplexers [Lewi03], [Betz05]. Resource outputs are connected to MUX inputs rather than directly to wires via pass transistors. It was found that despite the reduced number of locations that can drive a wire using direct drive multiplexers, there is an overall decrease in both area and delay [Lewi03].

Virtex switch boxes do not use multiplexers. Instead they have internal wires that are shared between the switch box IO. This forms a matrix of PIPs. Each PIP is a pass transistor controlled by a configuration SRAM cell. The shared wire architecture results in a higher flexibility per configuration bit than a purely MUX based connection box [Tava97].

Another common feature of all FPGA devices is the global signal distribution network. This *global network*, primarily used for clock distribution, is designed to distribute signals

across the whole device with a minimum of skew. Global networks have a tree like topology with configuration bits at each branch controlling the propagation of a global signal to the sub-branches. Generally, a global network is segregated into a number of regions all linked to a central, vertical spine.

The clock control for the central part of the tree is held in the central configuration frames. In the Virtex-II architecture the bits that control the configuration of the local branches of the global network are clustered at the top and bottom of each configuration frame. In the Virtex-4 and Virtex-5 architectures the bits that control the configuration of the local branches of the global network are embedded in the centre of every configuration frame. Each local branch of a global network terminates at the same point within each switch box. Bits within the switch box region configure the propagation of a global signal from a given network to the nearby logic or interconnect.

2.3 Designer Productivity

FPGA technology has its roots in digital circuit design and, like every silicon technology, it advances every 18 months, doubling the gate capacity available to the designer. The single largest threat to this growth is the gap that is forming between the number of available gates and the ability of designers to use these gates in the time frame of a typical design cycle.

The flexibility of FPGA technology provides the ability to specialise a circuit based on a set of parameters. This results in more optimal, lightweight solutions that both consume less energy and exhibit a higher throughput when compared to a more generic architecture. However, each specialised system must be designed and compiled.

There have been many advances in digital system design automation in an attempt to keep designer productivity in step with available gate capacity. Broadly speaking, there are two approaches to improving designer productivity.

The *first approach* is through the development of design descriptions and automated EDA tools for the translation and optimisation towards a form usable at the physical level. This approach is based on design abstraction, which is the process of generalisation by reducing information content to capture the detail relevant to an underlying view or model used to describe a digital system. The *second approach* is in facilitating the reuse of design descriptions whether original or automatically created and optimised. These two approaches will be discussed in more detail in the following sub-sections.

2.3.1 Design Abstraction

Abstraction is the underlying view or model that a designer uses to create a computing system. Design compilers and execution environments work at different levels of abstraction to manage a system. As the size of computing systems has grown, so has the level of abstraction.

There have been several fundamental levels of abstraction in hardware design. The first digital designs were mapped by hand. As the number of transistors in these circuits increased, low-level standard cells (at the Boolean functional level of complexity) were used as basic building blocks. To improve productivity further, EDA algorithms were developed to automate the placement and routing of a circuit. As the number of transistors on a chip continued to increase, the level of abstraction moved to the logic level (logic

synthesis), micro-architectural level (high-level synthesis) and the architectural level (behavioural synthesis) [Kast02].

Gajski and Kuhn [Gajs83] introduced the Y chart for describing the taxonomy of design automation in electronic systems. Five levels of abstraction have been defined, from the lowest *physical* level, through the *primitive*, *macro*, *component*, to the highest *system* level abstraction. At each level of abstraction a systems' *behaviour*, *structure*, and its *geometry* are represented. The five layers of abstraction combined with the three representations create fifteen *abstraction-representation states* of digital circuit design. Since there is no transition directly from a geometric representation to a behavioural representation, the circular Y chart has been adapted to create a design space grid shown in Figure 2.1.

	Behaviour	Structure	Geometry
System	Components role SW/HW Communicating partners	Component Connectivity	Floor-plan and Memory Allocation
Component	Register Transfer Language	Net-list of operations	Region shape Port locations
Primitive	Logic operators	Technology specific net-list	Placed and routed design
Physical	Set-up and Hold Time	Primitive configurations Wire configurations	Location of bit-fields and their configurations

Figure 2.1 The design space grid adapted for FPGA design

From left to right the representations are behavioural, structural, and geometric. Converting a behavioural representation to a structural representation is referred to as *synthesis*. Converting a structural representation to a geometric representation is referred to as *generation*. The reverse of synthesis is *analysis* and the reverse of generation is *extraction*. From bottom to top are the levels of abstraction: Physical, primitive, component, and system. Moving up the levels of abstraction is referred to as *abstraction* and moving down these levels is referred to as *refinement*. Modifying a design while in the same state is referred to as *optimisation*. Optimisations are done in order to improve some aspect of the circuit. For example, to reduce the power consumption, increase the throughput or minimise the area. Optimisations are at the heart of automated design tools.

2.3.2 Synthesis and Optimisation

As device capacity increases the use of logic synthesis coupled with automated FPGA placement and routing tools have become key in maintaining designer productivity. Every step in the design process attempts to maximise performance while keeping a check on the amount of FPGA resource used. Timing information about every design decision is extracted and used at each optimisation step.

The majority of FPGA design-flows produce a Register Transfer Language (RTL) description for synthesis and logic optimisation. An FPGA compiler tool takes this behavioural RTL representation as input and synthesises a macro-structural

representation, performing optimisations at both the macro and primitive levels before refining the design to a primitive-structure and emitting a translated net-list. Conventional synthesis approaches use iterative optimisation techniques that carry a high processing overhead.

Improved synthesis time is achieved by using programmatic component generation [Moha98], [Altmega], [Menc02], [Hwan98]. Component generators can create any design representation directly from a component-behaviour expressed in terms of a component function and a number of parameters. A conventional hardware component library stores the implementations of a large set of hardware components. A component generation library instead, stores the algorithms that generate a set of hardware components based on input parameters. Refinement is done using a hierarchy of programmatic structural constructors [Menc02]. Generic component generation frameworks produce a macro-structural representation in the form of RTL and thus still require synthesis, optimisation and translation to a primitive-structural representation. Target specific component generators are able to skip the synthesis step accelerating the compile cycle.

Iterative synthesis and logic optimisation together solve a wider set of digital design problems than component generation approaches. Impressive rates of circuit generation have been reported [Menc02]. A well structured hierarchical component library allows designers to capture their expert knowledge for re-use in multiple applications. Designing such a library of components requires good planning and a significant investment of time. Both Xilinx CORE generator and Altera MegaWizard Plug-Ins provide component libraries coupled with a GUI to provide a convenient method of creating often used components that have been tuned for the respective target FPGA devices [Moha98], [Altmega]. It is not possible to generate every kind of circuit using component generation so the approach has become complementary to synthesis, together providing a wide-ranging design productivity improvement.

2.3.3 High Level Language Synthesis

Hardware compilers for high-level languages are increasingly recognised to be the next step to reducing the productivity gap for advanced circuit development in general, and for reconfigurable designs in particular [Todm05]. The Open SystemC Initiative (OSCI) [OSCI08], was created to further the use of SystemC. SystemC is an open-source extension of C++ that supports HW/SW functional modelling. The definition of scheduling and synchronisation of concurrent processes is built into the event-driven simulator. Events are basic dynamic or static process synchronisation objects. SystemC is useful for performing a design space exploration and HW/SW partitioning of a system initially defined in C++. Although SystemC provides decomposition and simulation there is no agreed standard for synthesis or refinement.

Handel-C from Agility Design Solutions provides an environment for cycle-accurate application development using a C-like language in an attempt to further improve designer productivity and open up design to C programmers. The Handel-C language provides directives and constructs to explicitly define parallelism. The compiler analyses the Handel-C code and attempts to optimise and rewrite constructs to increase performance. All operations occur in one deterministic clock cycle, forcing the clock frequency to be reduced to that of the slowest path. Handel-C supports two targets: The first is a simulator target that allows development and testing of code supported by a debugger; The second target is the synthesis engine that creates a net-list for input to

place and route tools. A strong aspect of Handel-C is the availability of hardware abstraction layers (HAL) that make rapid prototyping easy for supported platforms. The Handel-C compiler will output VHDL, Verilog, SystemC or a target specific EDIF net-list [Holl05].

Impulse-C from Impulse Accelerated Technologies is a language and compiler for modelling sequential applications. Impulse-C uses the Streams-C methodology with a focus on compatibility with a standard C development environment. The Streams-C methodology uses a stream-oriented sequential process model where the data elements move through discrete functional blocks [Frig01]. The two fundamental components are independent, potentially concurrent, computing blocks (or processes) and streams that model the communication and synchronisation between processes. The compiler implements each process as a separate state machine. A platform description ties together portions of C code running on a processor and the processes mapped to hardware. The output from the compiler is either generic or FPGA-specific RTL [Pell05]. The Impulse C compiler is able to generate circuits using floating-point circuits inferred by ANSI C code. The compiler outputs RTL requiring further synthesis steps to produce an FPGA executable [Pell05].

Both system level synthesis [Kast02], [Holl05], conventional RTL synthesis, and circuit generation all result in a primitive-structural design representation. In order to map this onto an FPGA device, the primitive-geometric representation has to be generated and optimised. The packing, placement and routing tools handle the optimisation of a geometric-primitive representation.

2.3.4 Packing and Placement

FPGA placement optimisation allocates primitive components described in a net-list to the available FPGA resource while trying to minimise the net delays between connected components [Chan03].

Primitives are first mapped and packed into logic blocks with the objective of minimising the number of inter-block connections. Once the logic blocks have been packed, they have to be allocated to the 2D array of resource available on the FPGA. The primitive-geometric state of a design is initially generated randomly from the primitive-structural representation. This random arrangement is then optimised using placement and routing algorithms. Finding the optimal placement and routing of an arbitrarily generated net-list are NP-complete problems.

Simulated annealing is used to produce a good placement solution [Betz99a]. The base cost function for the simulated annealing algorithm is the overall circuit wire length [Betz99a]. Algorithms have been enhanced to consider the critical path delay and power dissipation of a placement solution [Betz99a], [Lamo03]. The algorithm uses a simulated annealing schedule with time complexity of $O(n^{4/3})$, where n is the number of blocks being placed [Betz99a].

Combining packing and placement provides an improved solution [Chen04]. However, unpacked primitives imply an increase in the number of primitive blocks that require placement, increasing n , resulting in an increase in the placement run-time. Chen et al. [Chen04], employ a quality factor to control the percentage of primitive block moves over packed primitives that are made at each temperature during the simulated annealing

placement. This allows the user to trade quality for improved run-time by reducing the percentage of primitive moves.

Using a combination of bottom-up clustering and hierarchical simulated annealing provides an improved quality for a given run-time over tools that only use primitive level placement. Re-creating cluster information in order to make more abstract geometric optimisations reduces placement optimisation time with good results. The run-time complexity of recreating the clustering is $O(N.K.F)$; where N is the number of primitives being placed; K is the number of nets connected to each primitive; and F is a net fan-out limit. Using three levels of clustering was found to provide good run-time improvements. At each cluster level, an initial constructive placement is created. This initial placement is refined using a simulated annealing placer with an aggressive schedule. The refinement provides cross optimisation between clusters to improve the quality of the solution. Tuning the clustering algorithm provides the facility to select a faster compile time if a high quality result is not required [Sank99].

FPGA placement software accepts placement constraints which allow a previous placement to be re-used or for the designer to specify the location of certain circuit elements [Patt01]. Placement constraints allow expert designers to specify the location of circuit elements in difficult problems that simulated annealing is unable to solve adequately. Placement constraints are necessary to specify the IO pad that an external signal must be attached to. It was found that fixing the IO pad locations increases the number of routing tracks required by an average of 12% [Betz96].

Every element in a circuit is connected to one or more other elements. This creates a layout optimisation problem to determine where best to place each element to minimise the total wire length of the circuit. Circuits are categorised into two types of layout problem: Circuits that have a layout made obvious by their data flow; and circuits that have a complex or non-obvious relationship between data flow and layout. The first types are easy for a designer to predict and specify, the second types will almost always result in a poor quality result when a designer attempts to specify their geometry using placement constraints.

Several component generation frameworks allow the designer to specify flexible layout directives to take advantage of the first type of circuit layout problem. Lava is a hardware description language, which allows the specification of circuit layout and behaviour [Bjes98]. Lava circuit elements are conceptually enclosed in rectangular tiles. Circuit compositors connect circuit elements and define their locations relative to each other. Composed circuits can be combined as sub-circuits using the same set of compositors [Sing04]. A similar approach is used in Self-Implementing Modules (SIMs) which is incorporated into the Xilinx CORE generator framework [Hwan98]. The greatest benefits occur in circuits with high device utilisation, regular structure and regular wire communication between components. For example in an adder tree and in a constant coefficient multiplier there is a simple, and easy to route, wiring relationship between the circuit components that are laid out next to each other. Whereas, slightly more complex circuits, such as a FIR filter or the Fast Fourier Transform (FFT) butterfly network, show that care is required to avoid situations where a good layout has bad consequences for the inter-component wiring. In such cases automatic placement outperforms straightforward hand layout [Sing00].

2.3.5 Routing

After the primitive-geometry of the circuit elements has been optimised the connections between the elements need to be made. This requires allocation of signals to available wire paths in the FPGA and is commonly referred to as routing. A routing algorithm is used to allocate wire resource from the interconnect structure to connect the placed primitives as defined by the circuit net-list [Tsen92], [McMu95]. Routing in FPGA is based upon techniques developed for PCB [Seki91], and standard cell ASIC [Yosh82], systems. However, FPGA routing is more challenging due to the scarcity of routing resource which can result in long wiring paths or even routing failure [McMu95].

The interconnect is represented as a weighted graph $G = (V, E)$, where each vertex, V , is a wire or resource pin, and each edge, E , is a switch that connects two nodes. Routing is the problem of finding the shortest path in the graph while negotiating between nets so that each has exclusive use of the wires that have been allocated to it [McMu95]. A solution to the routing problem for net N_i , is the directed routing tree RT_i , embedded in G and connecting s_i with all its t_{ij} .

PathFinder is a widely used routing algorithm [McMu95] for FPGA. For each net N_i , Dijkstra's algorithm [Dijk59], is used to find the shortest path through the routing graph from the source terminal s_i , to the each sink terminal t_{ij} . The priority queue (PQ) is implemented as a binary heap to facilitate finding the minimum weighted path after each iteration. Performing Dijkstra's algorithm on a graph with E edges and V vertices exhibits a run-time complexity of $O((E+V)\log V)$.

Multiple signals create multiple paths through the routing graph, with no two signals allowed to use the same node in the graph. Routing node overuse or congestion is negotiated between net traces over several iterations. In each iteration every net in the design is revisited, if wires are being overused the net is ripped up and re-routed.

The cost c_n of using a given node n in a routing tree is defined as follows [McMu95]:

$$c_n = (b_n + h_n) \cdot p_n \quad (2.1)$$

where b_n is the base cost of using node n , p_n is a cost based on the number of nets currently using node n , h_n is the representative cost of the historical congestion of node n in previous global routing iterations.

The PathFinder Negotiated Congestion Algorithm is outlined below as described by McMurchie et al [McMu95]:

```
[1] While shared resources exist (global router)
[2]   Loop over all nets i (net router)
[3]     Rip up routing tree RTi
[4]     RTi <- si
[5]     Loop until all sinks tij have been found
[6]       Initialize priority queue PQ to RTi at cost 0
[7]       Loop until new tij is found
[8]         Remove lowest cost node m from PQ
[9]         Loop over fanouts n of node m (expansion)
[10]          Add n to PQ at cost cn + Pim
[11]       End
[12]     End
```

```

[13]      Loop over nodes n in path  $t_{ij}$  to  $s_i$  (backtrace)
[14]      Update  $cn$ 
[15]      Add  $n$  to  $RT_i$ 
[16]      End
[17]      End
[18]      End
[19]      Loop over all nodes  $n$  shared by multiple nets
[20]       $k \leftarrow$  number of nets sharing  $n$ 
[21]       $hn \leftarrow hn + hd(k)$ 
[22]      End
[23] End

```

The signal router loop starts at step 2. The routing tree RT_i from the previous global routing iteration is erased and initialized to the signal source. A loop over all sinks t_{ij} of this signal is begun at step 5. A breadth-first search for the closest sink t_{ij} is performed using the priority queue PQ in steps 7-12. Fanouts n of node m are added to the priority queue at $c_n + P_{im}$, where P_{im} is the cost of the path from s_i to node m .

After a sink is found, all nodes along a back-traced path from the sink to source are added to RT_i (steps 13-16), and this updated RT_i is the source for the search for the next sink (step 6). In this way, all locations on routes to previously found sinks are used as potential sources for routes to subsequent sinks.

At the end of an iteration (steps 19-22) the historical cost of each node shared by multiple resources is incremented by h_d which is a function of k , the number of nets that are sharing the node.

Betz et al. [Betz99a], add two enhancements to the negotiated congestion algorithm. The first enhancement is to only allow a fanout node to be added to the priority queue if it does not take the router beyond some maximum distance outside of the net bounding box. The bounding box test places a condition on the execution of step 10.

The second enhancement is as a result of the observation that the search for sinks is accelerated by initialising the priority queue for each subsequent sink search (step 6) to all the fanout nodes in the previous sink search iteration along with the existing routing tree nodes. Both fanout nodes and routing tree nodes have their cost set to zero. Without this enhancement, the router would have to re-expand all the fanout nodes for each sink search.

A routing difficulty predictor that uses a placed wire length model was developed in [Swar98]. For a placed circuit, W_{MIN} is defined as the minimum wire bandwidth that the router requires in order to successfully route the circuit determined using the following equation:

$$W_{MIN} = \frac{\sum_{n=1}^M q(n)[bb_x(n) + bb_y(n)]}{2NU} \quad (2.2)$$

Where $q(n)$ is an empirically determined net terminal correction factor (from [Chen94]) $bb_x(n)$ and $bb_y(n)$ are the x and y dimensions of the bounding box that contains all pins of net n , U is the fraction of unit length wire segments that the router is able to use before congestion results in un-routable nets, N is the number of tiles, and M is the number of

nets in the circuit. The routing stress predictor for an FPGA device with horizontal and vertical directed channel wire bandwidth of W_{FPGA} is defined as follows:

Predictor	Routing difficulty classification
$W_{FPGA} < W_{MIN}$	Impossible
$W_{MIN} \leq W_{FPGA} < 1.1 \times W_{MIN}$	Difficult
$1.1 \times W_{MIN} \leq W_{FPGA}$	Low stress

Swartz et al. [Swar98], developed a routing algorithm that first sorts nets in a highest fan-out first order and sorts sinks in terms of nearest first order and then uses a depth first search of the routing graph. Swartz et al. demonstrated through experimentation that, for low stress problems, their router achieved a run time nearly linear to the number of blocks in the circuit.

2.3.6 Design Reuse

The second approach to improving designer productivity is through facilitating reuse by capturing the effort that has already been expended in the detailed design and verification of a component, so that this design effort will not have to be repeated. This becomes possible because of the partitioning of a large design into smaller components. These components are then worked on separately making the design process more manageable. While the system itself may have a specific application, with carefully planned partitioning, the components of a system can be made *reusable*. The reuse may occur within the same system or across systems that meet different application needs.

The more reuse of components that is identified and used in system design, the shorter the design time. Similarly, the more applications a component has, the more it will be reused. Widening the applicability of a component is achieved by making it inherently generic and/or making a parametrised description allowing many variants to be described in a single code base. Design description languages support parameters and conditional compilation to facilitate flexibility. Programmatic component generation frameworks provide the ability to create very flexible components.

According to the ITRS roadmap [ITRS08], reuse must increase to account for 94% of the design, as part of the path to achieving a 39.6 times increase in design productivity by 2022.

System decomposition and reuse has matured to the point where many pre-tested *intellectual property* (IP) components are available for integration to accelerate the design cycle. Third party reuse requires pre-defined policies on the description, constraints and usage of design components. In order for a designer to have confidence in a component, there must be some guarantee that it has been properly verified functionally and will integrate smoothly within the chosen development environment both functionally and through the compilation. Supporting documentation and example use cases are necessary to help the designer to choose the right components for their system. The wide use of third party IP components is a testimony to the value of reuse [OCP02].

Packages of reusable third party IP that abstract physical hardware to a common set of interfaces and services are available as *platforms*. A platform presents a restricted model of operation that accelerates development. A platform depends on pre-specified policies

for the structure and operation of components to be compatible with the platform infrastructure. It is common practice to separate the computation from communication in the context of a platform [OCP02], [Sedc04]. A well defined communication infrastructure facilitates its own abstraction for the purposes of simulation and verification of a new system. Platforms foster reuse through several mechanisms: Reuse of the platform services themselves; Since each design component must conform to a standard, their reuse becomes easier; The layer of abstraction facilitates the reuse of applications on different underlying hardware, making it a powerful productivity catalyst.

Design automation systems that provide a platform that directly supports component integration provide an easy to use environment for selecting, configuring and integrating third party components into a system. SOPC builder, by Altera, and System studio, by Xilinx, assist with the integration of components with soft core microprocessors onto a target FPGA device [Altsopc], [Xilps08].

The combination of HLL synthesis, hardware abstraction and automated synthesis of communication structures between software and hardware has been used to create co-development environments such as Agility Design Solutions DK Design Suite [Agil08] and Impulse Accelerated Technologies Impulse-C Co-Developer [Pell05].

FPGA development environments that are based around a HLL language benefit from a predefined platform architecture. A widely used platform template is concurrent processes connected by streams. Streams force the serialisation of blocks of data so they fit their finite bit width. Physically on the FPGA, streams are mapped to either FIFO buffers or a handshaking interface. The physical structures and partitioning, as well as any supporting circuitry is automatically extracted from libraries and built in to a system by the HLL compiler.

With large libraries of reusable components and automated tools that support rapid system integration, less design time is spent on component creation, and more on system integration and testing. However, once integrated, it can take a significant amount of time to compile a system to a binary file for FPGA configuration. HLL co-development environments rely on synthesis of RTL from a HLL description, which in turn rely on platform support packages and the synthesis of RTL to a gate-level net-list representation. Before the net-list is usable on FPGA, it must be placed and routed for the target FPGA.

2.3.7 Software Design Productivity

Using a software programming language does not instantly provide software level productivity. Software development productivity relies on many innovations in compiler tools and techniques. The computing community are well versed with the behavioural programmers' model of an ISA. While a software programmer writing low-level or high performance code may consider the structure of the processor, generally a programmer relies on the compiler to abstract away the details of a computer system. A software programmer is not concerned with the geometry of the computing system.

The software-programming environment abstracts an ISA creating a simple, universal programming model. Register assignments are left to the compiler and only explicitly defined in assembly language. The simplicity of the programming model has allowed developers to focus on improving productivity. As a result the software development process is well matured.

A software designer will almost always rely on an operating system to provide a supporting platform of commonly used services. The facility to parameterise library objects and data structures are readily available, increasing their reuse. A software system is broken into several objects. Each object can be compiled independently so that, if an object is changed at source, only that object needs to be recompiled. A software system is composed by linking objects together to create an executable. Libraries of useful routines may be called in, either statically at compile time or during execution through dynamic linking. A software linker allows one pre-compiled piece of object code to reference routines and constants within another piece of object code by looking up the address offset from the start address in a table in a known location within the object. Standardised object formats afford the composition of reusable components from independent parties.

2.3.8 Compile Reuse

System integration tools make the rapid composition of a system from pre-designed components a reality. However, it still takes a significant amount of computational effort to map the system to a target architecture. The amount of computational effort to complete, roughly measured in computing cycles, depends on the state of the components provided.

In order to reduce the compile time, components can be pre-compiled and the result captured for reuse, just like any other design description. Each component is constructed by an independent process. Independence implies that there is no communication between the two processes during construction, so information to be shared between processes must be pre-defined and remains constant while the component construction is in progress. Once the interfaces of a component are well defined the internal design is performed independently from the rest of the system. The independence afforded by encapsulation makes well-defined components ideal for re-use by third parties.

Consider that in a system of components, the total number of primitives N is divided between K components, thus on average, each component contains N/K primitives. The primitives inside the component are *hidden* from the compiler. Since they are hidden, we must assume that primitives internal to a component already have a good quality arrangement and so do not need any attention from the compiler. The configuration of each primitive is *stored* within the component. Applying primitive constraints to build a component instantiation has a linear time complexity. This makes system composition complexity per component, rather than per primitive, so run-time is a function of K rather than N , reducing complexity by a factor of 100 to 1000.

When the components of a system are optimised in isolation the opportunity to cross-optimize between components is removed. Partitioning a circuit and optimising each partition separately will lead to a poorer performance (a lower quality result) than a design that is first “flattened” and then optimised. Conversely the partitioned design will compile more quickly than the flattened design. In many cases, the loss of cross-optimisation potential between components can be minimised through careful design to create near-optimal partitions that need no further effort from automated tools.

Since components that are compiled independently do not allow optimisation across their boundaries until they are brought together into a system, FPGA compiler tools that accept component structural-primitive representations build a flat system level structural-

primitive representation to optimise it. An example of a potential optimisation is where there are equivalent registers on both sides of an interface that two components connect over. Building a complete net-list allows the compiler to identify such optimisations and improve the performance of the system.

Capturing different stages in the compilation provides different compromises in terms of flexibility and compile time. The higher levels of abstraction tend to offer more flexibility and powerful design options to the integrator. A disadvantage of components provided as structural-primitive representations is that they are not as flexible as an RTL representation. Often structural-primitive components are made available as a set of net-lists, each with a different parameter combination. Such a solution is only feasible for a small number of parameter combinations. Structural-primitive components only hide information from the synthesis step. Subsequent steps in the compilation flow do not benefit from information hiding.

2.3.9 Discussion

A system will always be specific to an application, whereas a number of its components will have some reuse potential. The cost of creating a component is amortised across a number of uses. The more that a component will be reused, the better justified it is to expend extra effort in optimising it.

Primitive placement optimisation using simulated annealing has a computational complexity of $O(n^{4/3})$. Placement requires memory for the location occupancy, net bounding boxes, and cost tables. Routing has a computational complexity that is linear for low stress problems and in the worst case a complexity of $O((E+V)\log V)$. Routing requires large amounts of memory to accommodate the search heap, net trace information, net bounding boxes and the routing graph which typically has over a thousand nodes per tile.

Capturing the placement and routing of a component would eliminate these time and memory consuming processes and make system composition both easier and quicker. This would provide the opportunity to amortise the effort over multiple uses. Contrary to this, expending effort for each unique component use affords a higher level of flexibility. While a description at a higher level of abstraction provides more flexibility, it requires more compiler effort to reach a usable representation. The composition of a system is a one-off cost, and is thus under a greater pressure to reduce cost. Reusing a component at the source level allows for maximum flexibility at the expense of compiler effort.

This thesis explores the performance impact of optimising the components of a system down to the lowest level completely independently from other parts of the system. The motivation for this is that not only is the design and verification effort put into a component reused, the optimisation effort expended in mapping, placement and routing is also reused.

The following section reviews previous work on pre-routing FPGA components.

2.4 Pre-Routed Component Encapsulation

The concept of pre-compiled components, while being well developed and accepted in both the ASIC and software communities, is underdeveloped in the realm of EDA for FPGA. Compared to pre-compiled software components, the geometry of objects within

an FPGA are complex. An FPGA component occupies a two dimensional space and links between objects are made through net traces composed from several wires. The length of these net traces are constrained by the maximum required clock frequency.

Isolating component geometry is key to creating pre-compiled components that can be connected together to create a system on FPGA. Currently available commercial packing, placement and routing tools do not provide a framework to specify the necessary constraints. Several previous works, reviewed in the next sub-sections, have explored the composition of a system on FPGA from pre-compiled components.

2.4.1 Abstraction of Component Based Systems

Brebner [Breb97], introduced the *sea of accelerators* system model within the swappable logic unit (SLU) paradigm. Each accelerator is a task that is an independent operator, only having access to the host operating system so that no communication is performed across task boundaries. This system model draws strong parallels with the process model of software operating systems, where each process communicates through operating system services and never directly to another process. One advantage of such a set-up is that irregular communication patterns are handled centrally by a software process. Another advantage is that the supposedly software centric host, manages the sequences of input data, output reads, and access to memory. These simplifications create an intuitive environment for RTR development. The throughput of the central process will be the bottleneck in this system model.

Each task uses an SLU or component, which is abstracted to the amount of resource it requires to be implemented on the target FPGA device. For simplicity, each component is allocated a rectangular region of FPGA resource.

The issues of finite resource allocation and fragmentation have been identified in proposed RTR computing systems that support the dynamic instantiation and deletion of tasks running on an FPGA [Baza99]. Fragmentation results in task instantiations being rejected when the FPGA resource utilisation is still below 100%. A *fitter* attempts to manage the locations of each task instantiation to minimise fragmentation and therefore minimise task rejection. Bazargan et al. [Baza99], developed fitting strategies based on Brebner's sea-of-accelerators system model. Communication between tasks does not occur in the system model and communication structures between a task and the central process are not considered.

FPGA devices exhibit massive wire parallelism allowing for distributed communication. In order to take advantage of this inherent parallelism, task inter-communication requires wires to form connections between task areas. If tasks need to communicate it is clear that this must be considered by the task placement approach.

Ahmadinia et al. [Ahma04], presented a fitting approach that considers inter-task communication on a homogeneous FPGA device. Since many tasks will potentially have off-chip communication both connections between tasks and connections between tasks and the device IO are considered. The system is presented to the execution environment as a net-list of components. The set of nets that connect two components is abstracted to a single link weighted by the number of connections between the two components. The communication cost of a system is measured in terms of the overall sum of the Manhattan distance between the centre of connected tasks weighted by bus width. The

communication cost was reduced by one order of magnitude when compared with the KAMER fitting strategy [Baza99]. Furthermore, the communication aware fitter exhibited a comparable task rejection rate even though this was not an optimisation criterion [Ahma04b].

While placement decisions are made per component the routing still needs to be performed per wire. To be able to perform routing at run-time requires a significant amount of computer memory to store the routing resource graph. For low congestion problems, routing run-time is roughly linear in relation to the number of elements being connected [Swar98]. However, a significant number of CPU cycles are required to perform routing, even for routing a circuit that exhibits low congestion. Approaches such as specialised stitching routers [Gucc99], template routing [Kell00], and partial pre-routing [Blod00] have been proposed to reduce the computational overhead of routing, however, none of these have been investigated for their impact on overall system performance. If both components and the communication links between components can be completely pre-routed then the overhead of routing will be eliminated from the execution environment.

2.4.2 Component Encapsulation

Component encapsulation was considered to be restricted by the flexibility of the configuration transfer logic of the target FPGA. Since components are pre-routed, the inter-component communication mechanism is another significant limiting factor.

Two methods of pre-routing modules to accelerate the FPGA routing step using a theoretical FPGA architecture were described [Tess99].

Planar isolation was considered for rapid system construction from pre-routed components. In planar isolation all intra-component routing must only use wire resource within its boundary and all inter-component routing must only use routing resources that are between components. This approach was identified as requiring extra CLB space just for inter-component routing. Furthermore, it was shown that planar isolation is non-scalable for circuits with Rent exponents greater than 0.5 and that the rate of CLB loss scales exponentially with design size.

A *routing domain* is a set of wires in the FPGA that are only reachable through connection blocks. Disjoint switch boxes partition channels very strictly into routing domains. *Domain-based isolation* segments every FPGA channel between inter and intra-component routing. An experimental approach suggested that the channel width had to be increased by 50% to support routing domain isolation.

These routing domains were created by early switch box designs. However, disjoint switch-boxes are no longer considered useful in FPGA fabrics so clear domains do not exist. Modern FPGA switch blocks eliminate domains by permuting switch box connections between track groups [Wilt97], [Imra99], [Lemi02]. The optimal switch-box flexibility has been experimentally determined to be a minimum of 3 [Lemi02]. The typical switch-box flexibility (F_s) of a commercial architecture is around 5 or 6.

In an FPGA routing fabric, a wire can be driven by one of many sources. In design flows for static systems, the routing tools ensure a wire is only driven by one source. Early FPGA architectures [Xilds003] used bi-directional wires that could be driven by one of

several tri-state buffers along its length. The presence of bi-directional wires presents a problem for dynamically reconfigurable FPGA systems. If components are constructed without knowledge of how an adjacent component was constructed there is a high probability that a boundary-crossing wire could be driven from both sides of a region border and cause a destructive current to flow. The impact of managing the potential contention increases with the wire lengths used in the FPGA interconnects [Char03]. However, it is impossible to know in advance, at the time of component construction, the arrangement of signal-conducting elements in every other component it may be placed adjacent to.

Patterson [Patt03], presented a method of component construction that involves locking routes to logic resources in a column and then overlapping these columns to form connections between the components. The solution does not provide any mechanism to guarantee that contention in the communication column does not occur. It was suggested that contention be resolved at run time, a proposal that would exhibit both a highly non-deterministic response time and no guarantee of success.

Dyer et al. [Dyer02], describe a method of constructing swappable components in the Xilinx Virtex FPGA architecture. A large amount of user intervention was required for the prototype system design. A single reconfigurable region was described. Two methods of building communication structures between the static and reconfigurable region were explored. The first approach used JBits [Kell00] to create connections between the circuit placed in the reconfigurable region and the static part. However, this approach was dropped after issues were encountered. Thus a second approach was adopted, where the signals between the static and dynamic regions passed through logic slices. JBits was still used to perform partial reconfiguration of the dynamic region. The logic pass-through slices are contained within macros created by hand in the Xilinx graphical FPGA editor and then instantiated in the VHDL design. Single slice macros were used to direct static signals around the reconfigurable region. Double slice macros, with fixed routing in between the two slices, were created to constrain signals between the static and dynamic regions. The placement of the macro slices was fixed to ensure that signals in the static part of the system were not affected by the reconfiguration. It was found that a routing exclusion zone of around 3 to 6 CLBs was required between components to avoid destructive contention, resulting in a significant waste of FPGA area. Further resource is wasted in the use of slices to constrain signal paths. Since the signals have to pass through extra levels of logic the delay is increased.

When creating components, Horta et al. [Hort02] used routing constraints to control the signal paths between the static and dynamic portions of the system. The routing constraints were satisfied using a modified version of the router in which the use of individual PIPs could be disabled and signals could be assigned to use specific wire segments. This ability was used to provide constraints to ensure: The reconfigurable component nets did not pass outside of the dynamic region; that the static portion of the system did not use wires passing in to the dynamic region; and that signals passing from the static to the dynamic region used the same wires in every reconfigurable component. A *gasket* area around each dynamic region was defined to maintain isolation between the static and dynamic portions of the system. Systems with only a small number of components on each device were reported.

The facility to constrain circuits to an area of FPGA resource has been incorporated into commercial tools to enable swapping at run-time for the Xilinx Virtex and Virtex-II architectures [Xilap290]. The circuit structure within this area is fixed by the standard synthesis, placement and routing tools. This method provides for communication between neighbouring modules through tri-state buses known as bus macros [Cami02]. This allows modules to be compiled right up to the completion of the generation of the configuration file. Furthermore, it allows regions on an FPGA to execute while other regions have their components swapped.

The RAPTOR2000 platform developed by Kalte et al. [Kalt04], provides more flexibility with regard to component width and location than the previously mentioned slot-based approaches. The pattern of tri-state buses on the Virtex architecture repeats every four columns. Utilizing this, they implemented a system that supports components of any width that can be placed anywhere along a single contiguous reconfigurable region. Both a component's placement and its width are flexible along one dimension (components have to span the entire height of the device) and both width and position increment by four columns. The platform infrastructure is placed at either end of the reconfigurable region. The communication between components is unique in that it does not use inter-region communication macros that require a fixed placement. Instead the tri-state wires create a flexible bus architecture that reaches horizontally across the device. In the Virtex-II architecture, the minimum horizontal placement flexibility is four. It was suggested that four versions of a core be created for increased placement flexibility [Koes05].

Subsequent dynamic component methodologies use the mainstream design tools to create the logic structure and a fixed set of macros are used to communicate between dynamic regions [Beck07], [Hubn06], [Sedc06]. The placement of these macros is locked, forming fixed reconfigurable regions or slots. The communication efficiency and flexibility of the system is dictated by the efficiency and flexibility of the communication macros and the communication infrastructure between slots.

Computing performance is limited by inherent communication overheads. Inefficiencies in the transport of data limit the number of useful computing circuit-cycles that it is possible to perform in today's silicon technology. One of the key features of FPGA technology is that it provides massive wire parallelism on chip and good I/O parallelism off chip. Typically 80% of die area on a commercial FPGA is devoted to interconnect [Sing02], with 70% of the configuration bits being associated with the control of interconnect [Breb03]. Thus, interconnect is a major factor in the functional density metric, as it affects the area, execution time and configuration time of a system. Poor abstraction of the communication within computing applications on FPGA is dangerous and will result in extremely poor performance.

The limitations of the communication structures between reconfigurable regions have strongly influenced the flexibility and performance of these RTR platforms.

2.4.3 Communication Layer

The communication macros used in pre-routing are a key consideration in the system level communication strategy employed by the platforms that support dynamic composition of pre-routed components.

Walder [Wald04], Marescaux [Mare02] and Bobda [Bobd05] use tri-state bus macros [Xilap290]. While others [Dyer02], [Hort02] noted that a significant design effort was required to realize the communication structures.

Walder et al. [Wald03], worked on a practical system implementation based around a slot architecture. The FPGA resource is split into regions of a fixed width where each region occupies the full height of the device. Components are pre-compiled and characterised by their width and execution time. The scheduling of components onto suitably sized areas is reduced to a simple queue model. The platform requires that each component has a standard interface which maintains a link to its neighbouring regions to ensure all components can communicate with the host system. The communication infrastructure forms a 16-bit bus running at 20MHz shared between all the components. Components have a separate interface to each of their neighbours. The platform manages the external and internal memory simplifying access for the reconfigurable components. The bandwidth of data flow in the platform is limited by the single bus architecture. Memory access is also performed across this single bus. The decision to put internal memory under the control of the platform infrastructure was due to the floor-plan of the Xilinx Virtex architecture where memory is at the left and right edges of the die.

Taylor et al. [Tayl02], developed the FPX platform to process network traffic at high data rates. The FPX platform prototype supported up to four reconfigurable regions on a Virtex-E 2000 device. A ring network was used as it has more parallelism and uses simple point to point links that are able to run at a higher clock frequency. The network used a 32-bit bus running at 200MHz. The Virtex-E device has internal memory blocks distributed more evenly across the die than the Virtex architecture, thus each component has access to dedicated internal memory. Each component has an interface to the FPX memory arbiter which marshals access to external memory.

Partition fragmentation occurs in cases where the FPGA area is divided into a finite number of predefined *slots* that may only contain one component. If the component size is not exactly the same size as the slot, then the remaining empty space in the slot is wasted, resulting in partition fragmentation [Hand04]. Furthermore, if a component is too big for any one slot on the platform, then it must be partitioned across several slots. Wigley [Wigl05], explored the automation of component splitting and generally found that splitting results in a loss in application performance. Hubner et al. [Hubn06], created a slot interface structure that supports multiple components in the same slot. The components sharing the slot also share the bandwidth of the interface, limiting the data transfer bandwidth to each component.

In the platform proposed by Kalte et al. [Kalt04], external IO is restricted to either end of the reconfigurable region. A single bus that spans the reconfigurable region provides the communication infrastructure. In their prototype platform the bus has two 32-bit data channels, one in each direction, running at 18MHz. Since every component in the reconfigurable region shares this bus it will be the communication bottleneck. In order to combat this, segmentation of the bus was proposed so that groups of components share their own private portion of the bus.

One early implementation of a network on chip (NOC) on FPGA was presented by Marescaux et al. [Mare02]. A 2D torus topology was proposed using *wormhole routing* with two time multiplexed *virtual channels* to provide deadlock free operation. Wormhole

routing is a blocking, hop-based, deterministic routing algorithm. It uses relative addresses and thus does not require a global knowledge of the network. In wormhole switching, message packets are pipelined through the network. This technique relieves routers from buffering complete messages, thus making them relatively small and fast. The router configuration is set up through reconfiguration. The fully-pipelined hardware achieves a network bandwidth of 77.6MBytes/s at 40MHz. The use of the NOC was demonstrated in combination with dynamic partial reconfiguration on a Xilinx Virtex FPGA. Due to restrictions in the Xilinx modular design flow, the 2D network was folded into a 1D-structure fitting the column-based architecture. The prototype system had four component slots that were able to communicate with every other slot. In this platform, the signals pass through the component regions and have to be maintained by a component. The flexibility that an NOC brings is paid for in terms of the resource consumed by the routers and network interfaces. In the above arrangement each router required a separate BRAM column. A router had a resource overhead of 1 BRAM and 223 slices and the network interface required by each task had a resource overhead of 259 slices and optionally 2 BRAM for buffering if required.

The static NOC infrastructure effectively limits components to fixed sized slots. Bobda et al. [Bobd04], proposed the dynamic NOC that removes this restriction. As long as a strongly connected network is maintained, the dynamic NOC is able to support arbitrarily sized component sites. Initially a mesh network is placed on the FPGA device. When a component is placed on the device it hides part of the network, which is restored when the component is removed. The routers are connected by a 32-bit wide bus and contain six 32-bit wide FIFO buffers. The network was able to run at 75MHz providing a bandwidth of 2.4Gbps. Each router resource requirement is significant at approximately 1689 logic slices and 6 BRAM. Several packet routing schemes were investigated for their ability to support a dynamically changing environment. While, in theory, this approach is able to support full 2D component flexibility, there was no evidence of a prototype system that actually exhibited this ability.

Instead of solving the issues with a dynamic NOC, the authors went on to develop the Erlangen Slot Machine (ESM) that incorporates several innovations to circumvent the difficulties that RTR computing on FPGA presents [Bobd05]. The ESM provides component slots that span the height of a Xilinx Virtex-II 6000 FPGA device. A number of slots may be combined to accommodate larger components. The ESM architecture template defines four methods which components may use to communicate:

- Direct neighbour connection
- Non-neighbour connection through a Reconfigurable Multiple Bus (RMB)
- Through external SRAM shared between slots
- Through a cross bar switch that also connects external peripherals

Each component must incorporate RMB logic that consumes around 338 logic slices. The RMB provides four separate buses so that several components may communicate in parallel. Each external SRAM is connected to a memory interface that provides access to three adjacent slots. The issue of fixed IO locations is addressed with a cross bar switch between the reconfigurable component region and external peripherals. In order to cope with the restrictions of the Xilinx MAP directive file (MDF), which forces reconfigurable regions to span the height of the device, the switch was placed in a separate FPGA device. This forces all IO streams to pass through two extra sets of IO pads between the two FPGA devices.

The constraint that reconfigurable regions must span the height of the device is only imposed by the bit-stream transfer tools. The standard bit-stream generator will generate partial bit-streams for a number of contiguous CLB columns. Re-location of this region is not supported by the tools. Sedcole et al. [Sedc05], use *merge dynamic reconfiguration* to eliminate this restriction to provide 2D flexibility in the SONIC-on-a-Chip platform. Thus, communication along both axes becomes useful. Using this new degree of freedom in static wiring and component region layout, several layouts of contiguous PE regions connected by the two bus structures were suggested.

The SONIC-on-a-chip platform template is based on a linear array of processing elements (PEs) that are each connected to their nearest neighbour via a unidirectional 32-bit *chain bus* running at 25MHz. In addition to this, every PE is connected to the central 32-bit *sonic bus* running at 50MHz which provides both communication between PEs and a link to the central processor sub-system [Sedc06]. Each PE region has an equal number of logic, RAM and multiplier resources. The pattern of resource is identical for each region so that a PE may be placed in any one slot. Partition fragmentation is alleviated to a certain extent by allowing PEs to occupy more than one contiguous slot. While two dimensional layouts for both the Virtex-II and Virtex-4 FPGAs were suggested, the systems that were actually implemented only had two slots.

Within a particular domain of computing, an algorithm will typically exhibit a specific communication pattern that in turn benefits from a particular communication infrastructure. Thus, platforms with a single communication infrastructure tend to be suitable for a specific application domain. SONIC-on-a-chip was designed specifically for video processing applications. The FPX platform was designed for network processing. Platforms by Kalte et al. [Kalt04], and Walder et al. [Wald04], both use a traditional bus architecture. The ESM proposed by Bobda et al. [Bobd05], is unique in that it provides several modes of communication all at once.

In the reconfigurable processor arena, NIOS and MicroBlaze processor based systems are naturally centred around memory mapped bus architectures [Altni02], [Xilm08]. Custom instructions provide tightly coupled communication within an ISA architecture [Altni02]. Simple streaming interfaces mapped into a processor's memory space provide a low overhead link between ISA and custom logic [Xilfs07]. Vendors of high level language tools provide their own platforms, optionally on top of FPGA vendors soft processor platforms. Both Handel-C and Impulse-C promote a streaming communication architecture and have support library packages that afford some portability through abstraction of the underlying hardware [Agil08], [Pell05].

However, a designer can only pre-plan the communication structure of a system using pre-compiled components to a certain extent. A common solution to providing communication between all components is a shared bus. However, the bandwidth of the bus will limit the bandwidth of data between every component in the system. The bandwidth issues are partly solved by using multiple buses or a single bus that can be segmented between groups of communicating components. Network on chip structures provide a convenient method of sharing a more parallel interconnect between multiple components. An advantage of network technology is that it has been proven to handle dynamic introduction and removal of nodes. However, the buffering and routing circuits present a significant overhead. While dynamically constructed NOCs have been proposed,

the low level mapping tools to support the construction of these structures hamper their development.

The extra logic required to connect to a shared interface structure contributes to the overhead that a platform imposes on the application it is supporting. It is often the case that the communication infrastructure becomes the bottleneck unless it is adapted to the model of computing used in an application. In contrast to this, the low level wire structures will always be employed to convey data between components. Thus, a detailed study of how the mapping tools create these structures is key to improving their performance.

2.4.4 Discussion

The ability to encapsulate components at the wire level and provide communication between regions has facilitated more practical research using binary components in prototype FPGA systems. In order to pre-route a component there must first be isolation between components. A study on the impact of partitioning and cluster placement concluded that the partitioning quality is critically important for producing a high quality final placement. It was shown that the divide-and-conquer scheme was likely to have a quality loss of 10-20% when compared with an ideal flat placement [Wang03]. In order to achieve a good quality partition a designer requires accurate feedback on the effect of each partitioning decision. Once components have been isolated, communication channels need to be created between components.

It is designer productivity that is holding back FPGA computing from wide spread use, not FPGA performance. Thus, one could say there is a margin of performance that could be sacrificed to improve productivity. Increased productivity will increase the number of architectures created, reclaiming lost performance through specialisation. However, the performance penalty must be carefully measured.

Generally speaking, computing performance is limited by the inherent communication overheads. Inefficiencies in the transport of data limit the number of useful computing circuit-cycles that it is possible to perform. One of the key features of FPGA technology is that it provides massive wire parallelism on chip and good IO parallelism off chip, with much of the FPGA real estate devoted to interconnect. Thus, how the interconnect is used by a system on FPGA will strongly effect circuit delay, area and power. Poor abstraction of the communication within computing applications on FPGA will result in poor performance. Thus, for efficient isolation and communication, there has to be a focus on wire level detail. This dictates a focus on the automated tools that map connections to the available interconnect wires. If component encapsulation is going to effectively improve designer productivity, the task of specifying a component's encapsulation and interfaces must have a convenience that is equal to the existing static design flows. Past encapsulation environments have required a large amount of effort on the part of the developer to specify the necessary wire constraints without a robust means to automate the process. This has resulted in little exploration into multiple interface types and component based system topologies.

More detail about the interconnect architecture is required to develop automated tools that support wire level component encapsulation. The structure and operation of state of the art FPGA technology has become significantly complex. While commercial FPGA vendors are very active in the research community, there is still a lack in the availability

and support for using this kind of information. There is no significant reason for the vendors to freely release and support the information required for third parties to build their own design tools.

There is, however, a large body of information on what constitutes a good FPGA architecture and mapping tools in general. Imposing the restrictions of the commercial tools on an investigation into the improvement of component mapping techniques will provide little advancement. A more productive strategy, employed in the research to support this thesis, is to build an adequately detailed model of both FPGA technology and mapping tools that are both useful for exploring dynamic reconfiguration techniques and broadly applicable to state of the art commercial FPGA technology.

3 Framework

“To permit computations which are beyond the capabilities of present systems by providing an inventory of high speed substructures and rules for interconnecting them such that the entire system may be temporarily distorted into a problem oriented special purpose computer.” [Estr60]

3.1 Introduction

Estrin's vision of a fixed plus variable structure computer architecture became a reality with reconfigurable computing on FPGA. However, the time it takes to create a problem oriented special purpose computer is significant. Put another way: The number of computing cycles it requires to convert a system description to an FPGA configuration is significant. Each time an FPGA is configured there is a time penalty. This *compiler effort* is weighed against the number of computing cycles that will be saved when using a special purpose computing circuit, over a general purpose architecture. If the special purpose computer is to operate continuously, or a large number of units will be in operation, then the aggregate number of computations saved will be higher than the number involved in compilation.

Previous measures, such as the functional density metric [Wirt97], shown in equation 3.1, have attempted to provide a relative measure of performance which is then used to compare different implementations of a given task.

$$D_n = \frac{1}{A_E \left(t_{En} + \frac{t_T}{n} \right)} \quad (3.1)$$

where D_n is a function of: the silicon area (A_E); the time taken to perform a computing step (t_{En}); the configuration transfer time (t_T); and number of compute steps between reconfigurations (n).

The configuration transfer time (t_T) is a fundamental overhead of an RTR system. A region of FPGA cannot perform useful computation while it is being reconfigured. There are several approaches used to reduce the impact of configuration time on the functional density of a system [Bitt97], [Durb01], [Naka99], [Mali05].

Unfortunately, these previous measures assume that every bit-stream has already been compiled and do not consider compilation time as a cost. By extending the functional density metric to include these overheads, it is possible to assess the benefit and cost of different dynamic reconfiguration options such as: choosing which variable parameters to support through reconfiguration; the impact of adding flexibility; and how much flexibility to incorporate into component descriptions.

To provide a more accurate performance metric, we have extended the functional density metric of [Wirt97] to consider the run-time reconfigurable system as a whole, by additionally considering the time and silicon area attributed to the *preparation* of a configuration. This new metric, called the holistic functional density can be expressed as:

$$D_n = \frac{1}{(A_E + A_S + A_P + A_M) \times \left(t_{En} + \frac{t_P + t_T}{n} \right)} \quad (3.2)$$

where the holistic functional density (D_n) is a function of: the silicon area of the executing FPGA (A_E); the silicon area attributed to the storage of the design representation (A_S); the silicon area to support the reconfiguration process (A_P); the silicon area for the process memory (A_M); the time taken to process a representation into a configuration (t_P); the time taken to perform a computing step (t_{En}); the configuration transfer time (t_T); and number of compute steps that will be performed by this configuration (n).

Mainstream FPGA EDA tools are designed to achieve a good trade-off between FPGA area (A_E) and execution time (t_{En}). They will typically require a circuit preparation time (t_P) that is in the order of tens of minutes to hours (depending on the size and complexity of the design). While a long preparation time is acceptable in a statically reconfigurable environment where the result of a compilation is used in many units deployed to market, it is not acceptable in niche applications such as FPGA-based HPC. While much of the current research is aimed at achieving performance gains through increases in the execution speed of the application when mapped to FPGA, equation (3.2) shows that even greater productivity gains can be achieved by reducing the preparation (compilation) time (t_P). The more cycles performed by the execution object produced by the compiler, the more worthwhile is the time spent on compilation, similarly, the more instances of the execution object that are reused the more time that is saved on compilation. However, to achieve reuse, components would need to be individually optimised, rather than the current practice of global optimisation.

At the behavioural level, a system can be represented by a number of components. Each component will exchange information with a number of other components within communicating groups. Currently, when a system of components is mapped to FPGA, it is first “flattened” and then optimised, thus losing the component detail. To achieve component reuse, the component detail would need to be preserved. However, it is a non-trivial problem to maintain isolation between components throughout the compilation process to the final geometrically optimised representation of the design. Thus, we propose that, in order to create compatible communicating components, each must adhere to a common *policy* on the use of any shared resource. A policy ensures that each component is protected from interference by any other component that is resident on a shared medium. Each component is given exclusive rights to a sub-set of the fine grain resource on the FPGA device. *Interface definitions* are used by the mapping tools to create compatible communication structures between components.

It is relatively simple to constrain component resource to regions on the surface of an FPGA device. The greater challenge lies in applying constraints on the interconnect usage of each component without adversely affecting system performance. Thus the focus of this study is on creating pre-routed components. Then, assuming that it is possible to produce pre-routed components, we must determine if it is worth the effort. That is, we must substantiate the following hypothesis:

“The productivity benefit of using pre-routed components outweighs any performance impact that may arise”

This raises a number of other questions. How do we quantify and equate performance and productivity? If we relate performance to execution time and productivity to compile time, then we need to determine how pre-routing effects the system compile time and how pre-routing affects the execution cycle time and device utilisation.

In order to address these issues, a detailed architectural model is first constructed (defined in section 3.2, specifics in section 4.4, characterised in 4.5) to better understand how the mapping tools should partition FPGA resource between components and construct the interfaces on a target architecture. To produce compatible pre-routed components, the tools must impose geometric constraints on the placement and routing. Therefore, the resultant mapping produced by placement and routing tools in a “natural” unconstrained scenario is investigated in section 3.3. This leads to the development of a component constraint policy and interface definition framework in section 3.4 that provides a similar result when components are pre-placed and routed independently. The intention is that techniques developed in this study are applicable to commercial FPGA compiler software without changes to the target FPGA architecture. Then, and only then, can we address the productivity versus performance issue of the above hypothesis. We achieve this (in subsection 5.4.4) by comparing the proposed pre-routing technique with an approach that only isolates components up to the end of the placement phase, and with an approach that flattens the entire design before placement and routing.

3.2 Architectural Model

In this section we present an architectural model that is based on the underlying principles of FPGA design where the architecture is a trade-off between ease of manufacture, silicon area, circuit performance, and the run-time of automated mapping tools. While the model presented here is representative of modern, commercially available, FPGA devices, it has been intentionally simplified to maintain focus on the issues surrounding pre-routing.

3.2.1 Basic Tile Structure

The architecture model follows the convention that the positive X axis is along the horizontal axis and increases in value towards the right of the page. The positive Y axis is along the vertical axis and increases in value towards the bottom of the page. A two-dimensional tile space is defined where each tile uses the structural template shown in Figure 3.1.

Every *tile* has a connection to a computing *resource*, such as a cluster of *logic elements* (LE), high density RAM, IO interface, multiplier or processor. Every tile has two wire channels that span the tile, one channel propagates signals along the X axis and the other along the Y axis. Every tile has a signal *interconnect box* which facilitates the connections between the resource interface and the two wire channels. A tile resource may optionally be able to drive a *global* signal network. It is assumed that the interconnect box provides read access to each global signal network.

Figure 3.1 Basic FPGA tile

The resource interface, interconnect box, and wire channels are the same on every tile. Although the interconnect is uniform, each tile may be connected to a unique type of resource.

3.2.2 Tile Resource

The architecture model follows the convention that a device will have a width of W tiles in the X dimension and a height of H tiles in the Y dimension. Its tiles are given X, Y coordinates. The top left tile is at $1, 1$ and the bottom right tile is at W, H . The location $0, 0$ is used to indicate that an element is unallocated.

Each tile has a resource box. A device map defines the computing resource available at any given site. A set of resource types is defined. A resource type can optionally add a number of device IO pads. The device resource map is built up by defining arrays of resource types with an origin coordinate and the number of resources in the X and Y dimension.

We define pre-routed component *placement flexibility* as the number of legal locations that a component can be placed on a given FPGA device. The higher the placement flexibility, the more potential there is to reuse the pre-routed component mapping.

This study focuses on wire level encapsulation, therefore the architecture generator has not been complicated with providing support for all the various resource types. In general it can be observed that an FPGA has several resource types that are *abundant* (taking up 80% or more of the area), and several resource types that are sparsely placed around the die. A typical *abundant resource type* is logic and would be found in contiguous tile arrays of two or more tiles across. A typical *sparse resource type* is I/O and would be found in a ring around the edge of the die, in strips or as single tiles in isolation. A single *sparse resource type* and a single *abundant resource type* suffice to create the component placement flexibility problem that heterogeneous FPGA technology presents. Therefore, the resources used in this study have intentionally been restricted to logic resource and I/O resource types that only cover one tile. IO is considered to be the *sparse resource* and the logic resource is the *abundant resource*.

Placing IO resource in a ring around a sea of logic resources allows the modelling of the classical FPGA layout. Placing the IO in vertical columns throughout the device allows the modelling of recent Xilinx FPGAs such as the Virtex-4 and Virtex-5 devices. Placing IO resource in small clusters throughout an array of logic resource creates a similar placement problem to that posed by more coarse grain blocks such as DSP, memory or processor resource blocks.

3.2.3 Interconnect

In this subsection we present the details of an interconnect model that is based on *wire sets*.

FPGA interconnect technology has evolved to be highly regular, unidirectional and fully buffered, with connections being made either using a shared wire PIP structure [Tava97] or using direct drive multiplexers [Betz05]. Since an FPGA interconnect is constructed from a repeated set of tiles, wires that span more than one tile are stepped as shown, for the X-direction only, in Figure 3.2.

Figure 3.2 A diagram of track stepping

Wires that span W tiles produce exactly W wiring tracks on each tile. A group of W such wires is termed a *wire set*. In each tile, one wire in a set will begin and one will end. Each wire in the set has a length of $L=W-1$ tiles unless it is truncated by the edge of the device. In the case of a unidirectional interconnect, all the wires in a set are driven in the same direction. This is similar to the *track group* model used by Lemieux et al. [Lemi02].

A wire has a pin in every tile it passes through, numbered 1 to W . Each wire pin is optionally connected to a sink or source signal from the interconnect box on the tile. A wire set is specified in terms of its size W and an *IN* and *OUT* vector as follows:

- W : Wire set size, pins numbered 1 to W , W must be greater than 1
- IN : The input selection binary vector of length W , Elements numbered 1 to W
- OUT : The output selection binary vector of length W , Elements numbered 1 to W

The *IN* vector indicates the pins that have an input to the wire from the tile interconnect box. The *OUT* vector indicates the pins that have an output from the wire to the tile interconnect box. When the sum of all elements of the *IN* vector is greater than one, the wire set is bidirectional. When a wire is bidirectional, each input driver to the wire requires a tri-state control..

The X and Y channels are built from an integer number of wire sets, where each set may have a different value of W . The elements of the *IN* and *OUT* vectors are ordered from 1

to W in a positive direction along the X or Y axis. Thus, the directionality of the wire is dictated by the values of the elements in both vectors. For example, a wire set with $IN[I]=1$ and $OUT[W]=1$ describes a wire set carrying signals in a positive direction along an axis.

To be able to uniquely identify a wire it is given an origin tile and wire index. The wire will span a number of tiles. We use the convention that the tile with the highest value coordinates is the wire origin. As an example, the wire highlighted in Figure 3.2 has its origin tile at $X=3$.

The interconnect box on a tile facilitates the connection between the X and Y channels and the tile resource IO. The switch box and connection boxes are merged so that all the connection options may be defined in the same way. The resource connection provides the same number of inputs and outputs on every tile. The resource IO is assigned to different functions for each resource type. The interconnect box also provides the option to drive certain resource and wire inputs from a global signal network. Figure 3.3 illustrates how the same resource connections to the interconnect box are used for both an IO pad resource tile and a logic resource tile. Note the grey parts in Figure 3.3 are unused by a particular type of tile.

Figure 3.3 Pin mapping of logic resource and pad resource

Each wire input or resource input has a multiplexer. Multiplexers with up to 8-inputs are used to drive unidirectional wires. The increased *switch flexibility*, F_s , provides a higher level of diversity in an architecture where every interconnect box is identical. Multiplexers with up to 16-inputs are used to drive resource inputs. Multiplexer inputs may be driven from wire endpoints, wire midpoints, resource outputs, and global signals. An interconnect box specification provides the detail of what resource or wire outputs can connect to other resource or wire inputs. A set of multiplexer patterns is first defined. Each pattern is made up of a number of turns from the set *straight*, *from left*, *from right*, *u-turn*, *from resource* and *from global*. The architecture generator attempts to find channel outputs to populate the multiplexer inputs using the specified patterns.

The detailed routing of the global signal network is not modelled, although a finite number of global signal lines are defined. The interconnect box does not provide connections to drive a global signal network, instead the resource type can optionally

drive global signals. Only one tile can drive any given global signal. Every global signal has an output pin on every tile. In the architectures used for the study only IO resources drive global signals.

3.3 Net-List Mapping

In this section we investigate the resultant mapping produced by placement and routing tools in a “natural” unconstrained scenario.

The goal is to capture the result of a component placement and routing for reuse. This implies that the primitive geometry of a component must be optimised in isolation from any other component. The nature of automated placement, routing and how the interconnect is used is investigated in the following sub-sections. This leads to a strategy that tries to minimise the impact of compiling components in isolation, thus minimising the integration effort.

3.3.1 Placement

The input to the placement process is a component described as a synthesised net-list of primitives and a set of interface definitions, where each interface has a set of connections, and whole interfaces are assigned to an edge offset. The component is given exclusive use of a rectangle of FPGA resource that has a *component region* of a fixed area that adequately contain the set of primitives. The result of the algorithm is an optimised placement for every primitive in the component.

The placement algorithm that will be investigated is based on simulated annealing (SA) using overall circuit wire length as a base cost function. Figure 3.4 illustrates several approaches to the placement optimisation of primitives within two components whose resource tiles are coloured either light or dark grey.

Figure 3.4 Region constrained placement issues

In the conventional *unconstrained* SA placement algorithm (Figure 3.4 (a)), if two components have resource elements which communicate with each other, the placement algorithm moves the components closer together, blurring the boundaries between the two components, making it difficult to isolate each component. However, it is different for

the case where the geometry of these components are optimised independently: The communicating elements need to be placed within a component region such that, when placed next to a communicating component, the total wire length used in connections between the components is minimised.

Without knowing where a communicating component will be placed relative to the component being optimised makes it impossible to minimise the wire length (Figure 3.4 (c)). Placing communicating elements at the centre of a component minimises the worst case wire length for all possible positions of the communicating component. However, the best case worsens with increasing distance from the border to the centre of the component. If the edge of the component region over which communication will occur is pre-defined as an *interface position* then there is a better potential to minimise wire length to the equivalent wire length in the unconstrained case (Figure 3.4 (d)). It is important to note that specifying one interface position, fixes the relative placement of the two components.

When the SA algorithm is applied to *region constrained* components (Figure 3.4 (b)), the communicating elements in different regions migrate to the border between the two components. Signal congestion will build up in the area either side of the boundary when there are a large number of elements communicating across that boundary. Without region constraints, the communicating elements are free to share the entire bandwidth present in the combined regions of both components. In the region constrained case, the effective bandwidth *between* the components is reduced to the total wire bandwidth across the border. Gayasen et al. [Gaya04], found that region placement constraints result in a reduction of the maximum achievable clock speed by up to 8%.

3.3.2 Routing

The connections of a placed design are optimised using a router. As stated in subsection 2.3.5, *PathFinder* is a widely used routing algorithm [McMu95]. Betz et al. [Betz99a], enhanced *PathFinder* to restrict the routers search space to use only the routing resource within a rectangular *bounding box* that encompasses all the pins of the net. Figure 3.5 illustrates some of the issues surrounding net bounding boxes.

Figure 3.5 Net traces are confined to a bounding box

The smaller the bounding box, the less flexibility there is in the routing. With less flexibility, it is more difficult to handle congestion since there are less possible paths. The potential for congestion is reduced by *expanding* the bounding box of all nets by a number of tiles to provide extra paths in the search space.

Applying *region constraints* to a component means that the bounding box cannot be expanded past the region boundary. The nets that have their bounding boxes further from a component border have more flexibility in their choice of route.

The *PathFinder* algorithm negotiates routing node overuse or *congestion* between net traces over several iterations [McMu95]. In each iteration every net in the design is revisited, if wires are being overused the net is ripped up and re-routed. A router is not able to use the entire wire bandwidth available because congestion effects will result in un-routable nets.

Because congestion is negotiated between every net in a given area of the interconnect fabric, adding net traces to an area that already has routed nets is prone to failure unless the router is allowed to rip up and re-route the existing traces. Xu et al. [Xu03], implemented a system that re-routed nets when a fault occurred on the interconnect fabric. They found it necessary to allow the system to rip up and re-route other nets to cope with congestion. Blodget et al. [Blod00], proposed pre-routing everything in a component except for interface nets. To cope with congested cases, this approach would require the ability to rip up and re-routed existing net traces, negating the advantage of capturing the component's routing. An alternative approach was to reserve an area of the interconnect fabric just for connecting components, however, this was shown to be non scalable [Tess99]. An alternative approach is required that is both scalable and that does not require routing at component connection time.

3.3.3 Interconnect usage

A simple connectivity model, of closer is better, is often assumed by a placement algorithm. However, because the interconnect fabric is composed of discrete wires that are horizontally and vertically directed, the distance between two connected components is a *Manhattan distance*. It has been shown that, in a fully buffered interconnect fabric, the delay is a function of the number of *wire hops* between a source pin and sink pin [Wang03]. This is because the buffers are tuned to provide the same delay independent of the wire length. Given this fact and the fact that passing through switches costs time, a router will attempt to create a trace with the minimum number of wire hops.

The router will generally use the longest wires it can to connect two pins, finishing off the trace with shorter wires when necessary. Wang et al. developed the following path delay predictor based on the location of two pins:

```

driver_to_driven_delay( clbDriver, clbDriven ) {
    if ( distance_between( clbDriver, clbDriven ) == 0 ) {
        fast = 1;
    } else {
        hori = horizontal_distance( clbDriver, clbDriven );
        vert = vertical_distance( clbDriver, clbDriven );
        get_num_lines( hori, long, hex, double, direct );
        get_num_lines( vert, long, hex, double, direct );
    }
    delay = long*long;
    delay += hex*hex;
    delay += double*double;
    delay += direct*direct;
    delay += fast*fast;
}

get_num_lines( dist, long, hex, double, direct ) {
    long += dist/length_of_long;
    dist = dist%length_of_long;
    hex += dist/length_of_hex;
    dist = dist%length_of_hex;
    double += dist/length_of_double;
    dist = dist%length_of_double;
    direct = dist;
}

```

The above predictor was found to be reasonably accurate when applied to designs mapped to the Xilinx Virtex-II architecture [Wang03]. This suggests that traces were using the most direct route to connect pins. Generally a router uses the most direct paths and does not overshoot unless there is a lack of direct paths either due to congestion or due to lack of switch paths. An important aspect to consider is the available switch options in the target interconnect fabric. Typically, there are nearly always switch paths that connect two wires of the same length and direction at their end points. Generally there are more switch options at a wire's end-points than at intermediate connection points. It is less probable to be able to turn a corner or to be able to connect to a different length of wire through a switch box. It is highly improbable that a switch allows a trace to turn back on itself or that a switch allows the overlapping of wires of the same length.

3.4 The Limits of Pre-Routing

Based on the investigation of circuit mapping in the previous section we now propose a method of component encapsulation that enables the pre-routing of components and connection by abutment. We then go on to explore the limitations of constructing a system from a set of pre-routed components. This leads to ideas about floor planning that provide guidelines on how to shape and encapsulate pre-routed components to make for failure free system composition.

3.4.1 Component Encapsulation

Consider two connected components that have their resource constrained to abutting regions. The four boundaries of a component region are named Positive X, Negative X, Positive Y, and Negative Y (PX, NX, PY, and NY).

The border line between the two component regions will bisect a number of interconnect wires. If intra-component net bounding boxes are constrained to their own region then no intra-component traces will be bisected by the region border. If traces always take the shortest route then each inter-component trace will contain exactly one wire that is bisected by the border between the two regions. The wires bisected by the component region boundary are highlighted in bold in Figure 3.6.

Figure 3.6 Wires bisected by a region boundary

In order to minimise the impact of a component encapsulation technique on the performance of a system it should produce a similar trace mapping to an unconstrained routing. Since bisected wires are used in traces between components in an unconstrained system it is proposed that component interface signals are mapped to bisected wires. Interface signals are locked to the same wire on both sides of the region boundary. Abutting the region boundaries of two components co-locates a set of bisected wires from both components, forming a connection between them. This connection or *link* is point-to-point, connecting two matching mapped interfaces or *ports*. The ports must be exactly aligned in order for a correct connection to be made.

There are many cases where a net will have a sink in a number of components. In a system that has not had its routing constrained: the multiple sink nets will be created using multiple pin wire segments in the interconnect fabric. Figure 3.7 shows two ways in which a net with multiple sinks may be split up to be supported by the point-to-point link paradigm. The $W=2$ wires used in the net trace are highlighted in black, the lighter wires are unused. The wire segments bisected by component region boundaries are shown in

heavy bold. The source in the white component region has four sinks, two in the dark grey component region and two in the light grey component region. The first approach connects this net serially between components. The second approach fans out in the source component to each sink component.

Figure 3.7 Multiple sink nets in a region constrained system

High fan-out nets, such as clocks, are routed on a global network which is orthogonal to component regions. A key concern when routing multiple sink nets is that differences in signal delay, or *skew*, from the source to each sink is within the sampling window of the destination register. As long as this concern is met, it does not matter whether the fan-out occurs in the source component region (illustrated on the right in Figure 3.7) or across component regions (illustrated on the left in Figure 3.7). However, for the proposed encapsulation method, the fan-out must be constrained to the source component region. A multiple sink net must be replicated on each port that connects to a destination component. Fan-in is naturally handled within the component that accepts multiple links.

It is important to note that there is only a finite number of wires along any given border line drawn across the surface of an FPGA. The *wire bandwidth* that is achievable across a border will vary depending on the depth that the wires reach either side of the border. For example, in an interconnect architecture with 20% of wires of length 2, an interface restricted to one tile either side of the border will only be able to use 20% of the total wire bandwidth in each channel. Thus, available wire bandwidth is a function of the area that the interface is restricted to. An *interface area* (of $2D \times E$) is defined as the region formed by D tiles on either side of the E tiles along a border. Figure 3.8 shows the interface area in each region of two connecting components. The bold wires are bisected by the region boundary.

Figure 3.8 Interface area bandwidth

The maximum wire bandwidth, W_{IFmax} , across the border, in an FPGA interconnect fabric made up of K wire-sets each of size W_i , is defined as follows:

$$W_{IFmax} = E \cdot \sum_{i=1}^K \min(W_i - 1, D) \quad (3.3)$$

An important advantage of the isolation approach proposed here is that an interface can potentially use all of the wire bandwidth that the FPGA architecture offers to carry signals across a border. However, there are a number of factors that may restrict the use of the maximum interface bandwidth. Figure 3.9 shows the bisected wires available in an interconnect channel. The bold wire is internal to the grey component region. Two wires are only useful for interfacing externally and two wires could be useful for internal connections or for interfacing.

Figure 3.9 Bisected wire availability

Firstly, it is not generally possible to use the full wire bandwidth of an interface area due to congestion effects. Secondly, bisected wires that have a driving pin and at least one other pin within a region are useful for internal connections. Thirdly, wires with a single pin in the region can only be used for carrying signals across a region boundary. Therefore, in order to provide a consistent number of available paths within a component region, we restrict interfaces from using those wires in a set that offer a path internal to the component region.

In addition, when the wires in a wire set are longer than the component dimension, a number of wires will neither start nor end in the component region. These wires cannot be used by the component and so should be automatically reserved. Note also that equation (3.3) implies that only D wires at most may be used from a wire set of any size.

3.4.2 Design Definition Framework

The component encapsulation framework presented here provides for the capture and reuse of several different design artefacts. Together they ensure contention free interoperability of independently constructed components on the same FPGA device.

Contractually, components are confined to a rectangular region. The four borders that make up the region boundary bisect wires in channels that run perpendicular to them. The bisected wires made available for interfaces must be the same on both sides of a border, and indeed across the whole system, therefore this information is captured in a *wire use policy* (see subsection 3.4.4). Such a policy is potentially useful for every system mapped to a given target FPGA architecture.

The components at either end of a link must use the same signal to wire mapping, captured in an *interface definition* (see subsection 3.4.5). The interface definition must be published before the connecting components are created. Once published, an interface definition may be used on any number of compatible components. The interface definition takes wires made available by the wire use policy and assigns signals to them. Mapping an interface resolves the two dimensions of a component *port area*.

The set of port areas, their locations and the resource requirements of a component are used in deciding the shape of a component. The rectangle that encapsulates a component and the port edge locations are captured in a *component template*.

A system is described as a set of component instance locations and a list of links. Each link is described as a pair of instance-port specifiers. Component shaping and system floor planning are discussed in a later subsection.

3.4.3 Wire Identification

Wires along a region boundary are identified by a unique set of three values: The tile offset from the most negative point along the border; the wire set within the bisected wire channel on that tile; and the wire within the wire set.

Wires in a set are stepped therefore, on each tile one wire in a set begins and one wire ends. This fact is used to formulate a scheme that identifies wires relative to a border line. Each wire is identified by its furthest positive reach from a given region boundary as shown in Figure 3.10.

Figure 3.10 Bisected wire identification

The example interconnect channel shown has a wire set of size four, that provides three wires across any given border. The three wires are indexed 1, 2, and 3. Wires of index 1, 2 and 3 would have a positive reach of 1, 2 and 3 from the border, respectively. Thus, it is possible to identify a specific bisected wire that exists in both regions.

3.4.4 Wire Use Policy

The *wire use policy* defines how every wire that crosses the borders of a component may be used, enabling sharing of the total wire bandwidth in a channel between internal nets, interfaces to nearest neighbour components and tunnelling interfaces. The intention is that these policies would be developed and standardised by an FPGA architecture expert. To connect neighbouring components, the policy identifies the bisected wires along the

border that are available for carrying signals to and from neighbouring components. The policy allows specification of this down to the individual wire level.

Wire sets are reserved by the policy to provide connectivity through a component. If partial wire sets are reserved for tunnelling then, placement flexibility of components connected across the region is reduced. Therefore, complete wire sets are reserved to provide connectivity through a component. The reservation is uniform across every wire channel in the component. The router will consider all wires belonging to a reserved set as external to a component.

In summary, the wire use policy specifies:

- The direction of each wire set as dictated by the architecture
- The wires in a set that are available to carry interface signals
- Whether a wire set is *reserved* for connecting through a component

It is important to note that in a unidirectional interconnect, all wires within a wire set drive signals in the same direction.

Wire set direction is dictated by the architecture in a unidirectional interconnect fabric. The combination of wire set direction and region boundary crossed determines a wire's function. For example: The wires in a set that carry signals in a positive direction will be outputs on a positive boundary (PX and PY), and inputs on the negative boundaries (NX and NY).

The policy is applied to all channels in a given direction uniformly to facilitate the relocation of interfaces along a region boundary. The X and Y directed channels can be considered as independent. Therefore, one policy is defined for the X directed wire channels and one policy is defined for the Y directed channels.

3.4.5 Interface Definition

This subsection outlines how component interfaces are specified to map to pre-defined interface definitions. Each signal present on a component boundary should be present in the RTL description and subsequently at the top-level of the component net-list. Each signal is identified by a unique name. The name is a combination of interface type name ("gals2" in the example given below), port area location (here there are two: nx1 and px1), the signal name within the interface (dat[1:0], req, and ack). The following example shows the module interface defined in part of the RTL description conforming to the IEEE Verilog 1364-2001 Standard:

```
module data_forward (
    input  wire      global_clk;      /* Global signals present in all interfaces */
    input  wire      global_reset;
    input  wire [1:0] gals2_nx1_dat; /* Port 1 (inverted) */
    input  wire      gals2_nx1_req;
    output reg       gals2_nx1_ack;
    output reg [1:0] gals2_px1_dat; /* Port 2 (original) */
    output reg       gals2_px1_req;
    input  wire      gals2_px1_ack;
);
```

The example interface has been instantiated as two port areas. The first port area has its signal directions inverted with respect to the second port area. The first port area is at

location 1 on the NX edge of the module. The second port area is at location 1 on the PX edge of the module.

The example interfaces are placed on opposite P and N borders. Data would flow into the first instance and out of the second instance in a NX to PX direction. These interfaces are compatible because the same interface definition is used in the signal allocation of both. Abutting the P edge of one instance of this component with the N border of another instance will create a link between their two ports.

The signals pre-pended with *global* are not mapped to interface wires, instead they are expected on the global signal network. The global signals are referenced in the interface simply to indicate the global signals that are necessary to support the interface.

3.4.6 Component Connection Extensions

The placement flexibility of pre-routed components presents a number of complex issues and restrictions. Component connection by abutting interface surfaces requires that components must be placed next to one another. However, it would be desirable to have some degree of flexibility in a component's placement relative to components to which it is connected. Being able to shift the interface surface would provide that flexibility. Some options to provide this flexibility are discussed below.

3.4.6.1 Interface Extension

Consider pre-routed *interface extensions* that connect to an interface by abutment and effectively shift the connecting surface along an axis perpendicular to its original region border. Because all components are pre-routed, the shifted interface surface must present the same wires to the connecting component as the original interface surface. If an interface uses wires of a single wire length (L) then the interface surface may be shifted in steps of L tiles along an axis perpendicular to the original region border.

Shifting the surface adds wires into each signal path, increasing the delay. This increased delay is proportional to the number of additional wires that have been appended to shift the interface surface. However, to make full use of the wire bandwidth offered by a contemporary interconnect fabric requires interfaces to use wires of different lengths, making the interface extension more complicated.

For example, consider an interface surface that uses wire sets of size 2, 3, and 4, and which needs to be extended by some integer number of tiles (N). Given a typical wire mix [Sing02] of,

- 5% wire set $W = 2$, wire length is 1,
- 21% wire set $W = 3$, wire length is 2,
- 61% wire set $W = 4$, wire length is 3,

and given an interface which uses 2% of length 1 wires, 10% of length 2 wires, and 10% of length 3 wires, this gives a total wire bandwidth of 22%. If the shift is $N=1$, then the length 1 wires are the most optimal (no partial usage or overshoot) to shift the interface surface. However, length 1 wires only provide 5% of the total bandwidth, while 22% is required to shift the interface surface. In order to make interface surface shifting by $N=1$ practicable, the interface bandwidth must be restricted to 5% for this architecture. This approach will still be restricted by the fact that an interconnect box will not usually provide the switch paths that would be required to connect wires of length 2, and length 3 to all of the length 1 wires and back.

For the above example, the granularity with which N can be varied is 6, that is an integer multiple of all wire lengths as shown in Figure 3.11.

Figure 3.11 Interface extension example

This still presents a problem when we consider interconnect delay. For a shift of $N=6$ the signals assigned to wires of length 1 will have traversed 6 extra wire hops, those assigned to wires of length 2 will have traversed 3 extra wire hops, and those assigned to wires of length 3 will have traversed 2 extra wire hops. Thus, a worst case delay associated with 6 hops would have to be factored into the delay budget for a component's interface.

This illustrates two important points: firstly, in order to provide scalable bandwidth, interface signals must be extended using the same wire type they were assigned; and secondly, for an interface that uses a mix of wire lengths, the interface shift must be an integer multiple of every wire length in the interface.

3.4.6.2 Cornering Link Extension

The previous discussion only considers connecting interfaces that are aligned and on the same axis. Connecting interfaces on different axes implies that the signal paths must turn a corner. Consider an interface that is X directed of width EY that is to be connected to a Y directed interface of width EX. The connection requires a minimum routing only region of EX by EY tiles as shown in Figure 3.12. Since a corner connection uses both X and Y directed interconnect channels, only one connection can occupy a corner routing region.

Wires within a wire set are staggered along the axis of propagation. Thus, when turning a corner a set of signals assigned to one wire set in the original axis can naturally spread to wires across the channel in the destination axis. Furthermore, signals assigned to different channels can map to staggered wires within a set in the same channel. The wire assignment of the X and Y directed interfaces can be specified together to take advantage of this fact.

For example, consider an interface that only uses wire sets of $W=3$. Each wire set provides two wires of length $L=2$. For the sake of keeping this scenario simple, X and Y channels have the same number of $W=3$ wire sets and the interconnect box provides the necessary switch paths. If the interface has eight signals, labelled alphabetically A to H, then as there are only two signals per channel, an interface width of 4 channels is required. Consider a link that connects a PX port to a PY port in a routing only region of 4 by 4 tiles. We denote signal to wire assignments in the form: $signal \Rightarrow \{channel, wire\}$. In this example the wire set index is the same in both source and destination channels.

Figure 3.12 Corner connection routing region

Thus if the interface assignment used by the PX port is:

$$A \Rightarrow \{1, 1\}, B \Rightarrow \{1, 2\}, C \Rightarrow \{2, 1\}, D \Rightarrow \{2, 2\}$$

$$E \Rightarrow \{3, 1\}, F \Rightarrow \{3, 2\}, G \Rightarrow \{4, 1\}, H \Rightarrow \{4, 2\}$$

Then the interface assignment for the PY port would be:

$$A \Rightarrow \{1, 1\}, B \Rightarrow \{2, 1\}, C \Rightarrow \{1, 2\}, D \Rightarrow \{2, 2\}$$

$$E \Rightarrow \{3, 1\}, F \Rightarrow \{4, 1\}, G \Rightarrow \{3, 2\}, H \Rightarrow \{4, 2\}$$

This is illustrated in Figure 3.12.

In this example the signal assignments from PX to PY channels equalise the number of hops for each signal. It is important to note that if PY and NY interface assignments remain compatible, then one corner direction (in this example PX to NY) will have unbalanced hop lengths across signal paths.

An arbitrary shift of the PY interface surface is handled by modifying the PY port interface assignment to account for the different distance. For the destination interface assignment to be the same for both left and right corners requires that the distance from left destination border to right destination border must be an integer multiple of all the wire lengths used. The dimensional restrictions on a corner turning link are shown in Figure 3.12.

Both straight and corner turning interface extensions require a routing only region, which wastes all the logic resource in those regions. In the next subsection, interface extension through other components is considered as an alternative means to connect non-neighbouring components.

3.4.6.3 Tunnelling Link Extension

Using bisected wires for nearest neighbour communication by abutment is generally not sufficient for system composition. Interface extension, both *straight* and *cornering*, offers increased placement flexibility, but at the expense of routing only regions.

Typically, FPGA vendors implement a channel architecture that has sufficient wire bandwidth to route their most complex benchmark circuits. Furthermore, the wire bandwidth will be increased by some fraction above what was required to route the most complex benchmark. In the past this increase has been as high as 20% [Lewi03]. Thus, for a typical component circuit (that will generally not be anywhere near as large as the most complex benchmarks used) there will be a significant amount of spare routing resource in the component region. In addition to this, the number of longer wires used will vary with the size of a circuit, with fewer long wires being used when an FPGA is broken into component regions. Wires that are longer than the width of a region will be left unused because they pass straight through the region.

Therefore, we propose that a percentage of longer wires are reserved and are used for connecting across a component to provide connectivity between non-neighbouring components. The ports on the two components that connect through a neighbour component will have to use an interface assignment to wire sets, called tunnelling wires, reserved from the neighbour component. In order to provide placement flexibility of components linked through another component by tunnelling, we propose that whole wire sets are reserved. The reserved tunnelling wire sets would be the same in every channel. Figure 3.13 shows both a structural and mapped representation of three components A, B, C all connected to one another.

Figure 3.13 Components connected through tunneling links

The wire detail of the nearest neighbour links and the tunnelling link are shown in the mapped representation in Figure 3.13. One $W = 2$ wire set is used for the nearest neighbour links and one $W = 2$ wire set is used for a tunnelling link.

Using a tunnelling link as an interface extension consisting of only reserved wire sets does present a number of conflicting criteria. Firstly, as interfaces can only be extended in steps, the region size of a component can only be varied in steps. The step size is dictated by the wire lengths used to connect across the component. A large step size will result in resource wastage due to region size quantisation. Secondly, the number of steps an

interface can be extended is limited by the maximum wire delay allowed across a connection. This limits the maximum width of the component region to be crossed. Thus, longer wires are better for crossing large distances, while shorter wires are better for flexibility of component size. Additionally, reserving tunnelling bandwidth when routing a component will increase the potential for routing congestion as we are effectively sharing the wire bandwidth between component connection time and system connection time. How reservation affects routing congestion will be investigated more in section 4.7.

3.4.7 Floor Planning and Component Shaping

We have previously discussed component reuse, so now let us outline a general model of an application specific system on FPGA that exploits reusable components. It is usual that the FPGA device will be provided on a general purpose board that creates a platform coupling the FPGA to several hardware interfaces and some on board memory. The platform provider will supply a set of IP components that assist in connecting to the memory and interfaces specific to the board. In many cases the platform will come with a software API and a corresponding set of FPGA components that facilitate control and data transfer between the FPGA and a host processor. On the other hand, the FPGA provider will supply general purpose (generic) components such as arithmetic circuits, FIFO buffers, etc. In addition to these, an application specific system will introduce its own set of components and tie these together with both the platform components and the generic components.

An encapsulation method based on pre-routed components introduces a number of restrictions, and thus we must formulate guidelines for mapping an application specific system under these constraints.

A system is first partitioned into components that are suitable for pre-routing. The partitioning must consider two important aspects: Firstly, all links are point-to-point, therefore components must be created with this in mind. While the source components may not fit this model, they can be incorporated into components that do. Secondly, the granularity at which a system is partitioned into isolated components must be considered. The overhead of pre-routing is greater for small components.

After the system has been partitioned into components connected by point-to-point links, interface definitions are created and are assigned to wires using the wire use policy. Assigned interfaces create a port area, which must be accommodated within a component region. The port presents a connecting surface along one edge of a component, and is not able to be split across the edges of a component. Thus, the system floor plan must provide every component with an adequate edge region co-located with each component that it is connected to. The wire bandwidth required across a given region boundary may dictate the minimum length of that edge. The required depth of a port area dictates the minimum component region depth. Thus, component shape is influenced by the amount of wire bandwidth required along a given region border.

An ideal floor plan provides each component with a region of the device that is just sufficient to accommodate the resource it requires. Thus, it is desirable to avoid inflating a component region to an area that is larger than is required to contain the number of primitives in the component. However, it has been found that, since the interconnect dominates the silicon real estate, 100% logic utilisation in an FPGA fabric adversely impacts on the overall performance due to routing congestion effects [DeHo99].

Therefore, it is advisable to expand a component region by a certain percentage from the absolute minimum necessary in order to yield a good use of interconnect. In addition, Wang et al. [Wang03], argue that since there are generally more intra-component nets than inter-component nets an aspect ratio of 1.0 (i.e. square) will yield the best routing performance.

Traditionally, floor planning occurs before placement and routing and thus has freedom in component shaping. In contrast to this, our components are created and pre-compiled independently before the floor planning process. To ensure a feasible floor plan is possible, an outline floor plan is created before components are pre-compiled. This guides the component shaping process.

We first build a platform specific floor-plan that mostly consists of interface components that contain IO blocks. Generally, the platform dictates the IO block placement and components that use IO blocks will be constrained to a specific location on the device. Within this platform outline, there will be regions of FPGA that are available for an application. The platform components present a number of interfaces to the application region. The next layer may be a set of components that facilitate access to these interfaces from more than one location in the application region. The application region is further broken into a number of sub-regions to be used by application components and generic components. Each sub-region may be unique in the type of resource it has and the interfaces it has access to. Application components are mapped to fit into these sub-regions. The sub-regions may be merged and changed to fit the application. The proposed encapsulation framework enforces only one restriction on the placement of unconnected components relative to one another: Their regions must not overlap. Since the interconnect is the same on every tile, components can be re-located by any integer number of tiles. Thus, once a set of interface signals are mapped to wires this *interface definition* may be relocated by any integer number of tiles. Furthermore, pre-routed components may be relocated on the surface of the FPGA by any integer number of tiles. The interconnect does not affect the placement flexibility of abutting unconnected components along the axis parallel to their touching surfaces or along the axis of a wire channel. However, connected components must be placed to precisely align their port areas and create a link. While link extension provides some flexibility as outlined in the previous subsection, routing only areas are preferably avoided.

While the interconnect is highly regular, the patterns of resource types on a heterogeneous FPGA present restrictions to pre-routed component placement flexibility. For example, consider two regions (A and B) of an FPGA that each have the same dimensions. If region A and B only contain logic resource then a component created in region A can be relocated to B. However, if region A has some tiles that are IO resource while region B only has logic resource, then components created in one region cannot be relocated to the other region. If the two regions have the same amount of logic resource and the same amount of IO resource, the locations of these resources in a region create a specific *resource pattern*. In order to be able to relocate components between the two regions the resource patterns of region A and B must match.

While it is important to consider the effect of resource patterns on placement flexibility in a compositional framework, the focus in this work is on the interconnect. Therefore, the placement flexibility issues surrounding resource patterns are not considered in any great depth.

3.5 Experimental Design Environment

This section reports on the specific details of the experimental design mapping tools that have been created to study wire component encapsulation within the FPGA design flow.

The proposed component encapsulation environment dictates a number of constraints that must be represented in the compilation environment. A component is constrained to an area that contains enough of each of the types of resource that it requires. Interface regions can be defined at any point along the boundary line. Interface descriptions, as noted previously, must be predefined. The description is interpreted by the automated tools to constrain the IO signals of a component to wires along the boundary.

Currently there are no open source tools along with an open FPGA architecture available for exploring the complex interaction of architecture, compiler and the component composition environment in enough detail to investigate the proposed pre-routing techniques. Thus, we have developed an architectural language and framework that describes the structure, function and configuration of a two-dimensional array of resources that can be connected using a uniform interconnect. We then developed a set of packing, placement and routing tools for exploring the effects of component encapsulation and pre-routing.

The architectural concepts outlined in section 3.2 are the basis of a framework that is able to read an XML description of an architecture and generate a graphical representation, interconnection graph for connection allocation, and an outline configuration architecture. The framework is able to describe a class of devices that include commercially available island style FPGA devices and more general programmable routing structures that are constructed from identical interconnecting tiles. We then go on to describe the packing, placement and routing tools and some of the issues relating to their usage in our encapsulation framework.

3.5.1 Design Entry

We use Verilog HDL to describe a component's function. As well as practical example systems, we use GNL [Stro00] to create a wide number of synthetically generated circuits with pre-defined routing complexity. In order to integrate GNL into our design flow, we have added Verilog HDL code generation to the existing GNL program.

The functional Verilog module is placed in a wrapper module that includes the interface type and port location in each signal name. The Verilog code is compiled into an EDIF net-list using an open source simulation and synthesis engine [Will06]. The EDIF is annotated with a reference to the policy and the component dimensions.

The synthesis target used is the Xilinx Virtex architecture. This produces an EDIF net-list of FPGA primitives, a set of interface signals, and a set of assigned interface definitions. The EDIF net-list is read in by our experimental compiler tool. The logic resource tiles in our target architecture contain a number of *logic elements* containing a 4-input LUT, MUX, FF and Carry logic. The logic elements are similar to half a Virtex CLB slice which facilitates simple mapping of a synthesised EDIF to our architectural model.

3.5.2 Interconnect Graph Generator

A router requires all the wires and switches in the programmable interconnect to be represented in a routing resource graph. Each wire is represented as a node and each switch is represented as an edge between the two wire nodes that it connects.

A wire spans multiple tiles but only has one graph node as it can only be used exactly once in a connection allocation. Each path through a wire input multiplexer is a directed edge between two wire nodes.

In previous work, a fully expanded interconnect graph was used [Betz99a]. A full interconnect graph requires one node for each connection resource in the device. Since every tile is identical in the model we can reduce the full interconnect graph by splitting it into two data structures: the tile interconnect graph and node state array. Only the tile interconnect graph has node edge information. For a device of width W and height H tiles with e edges in a tile, this represents a memory saving proportional to $((W \times H) - 1) \times e$ over a flat routing graph approach.

The node state array holds wire node specific information such as occupancy, usage cost and the selected edge. Previous work suggests that, as the routing resource utilisation is only around 40%, even for complex designs that exhibit 99% resource utilisation, a 50% storage saving can be achieved by only storing those parts of the routing graph that are visited by the router [Kell03]. Thus, the interconnect state array is allocated dynamically as nodes are visited during connection allocation.

In order to handle the single tile graph and dynamically allocated node state array, the router keeps track of the X and Y tile coordinates at each iteration. Following an edge in the tile graph implies a change in both the tile coordinates and in the state array index. To facilitate rapid calculation of the next tile coordinate, each edge is annotated with pre-calculated tile x,y offsets. To facilitate rapid calculation of the next array index, each edge is annotated with the pre-calculated index offset.

3.5.3 Packer and Placer

The wire end-point tile positions of interface signals are identified to the placer so that it can find the optimal location for primitives that connect to interfaces.

Normally the process of packing logic clusters only considers connectivity, leaving the placement of the clusters to the placement algorithm. Considering that two logic elements could be optimally placed close to interfaces on opposite borders, unless the packer considered this constraint it may pack both of these elements into the same cluster, forcing one interface connection to be sub-optimal.

The simulated annealing placement algorithm from VPR [Betz99a] has been adapted here. In order to take into account the interface positions, the packing is done within the same algorithm as the placement. It was found that combining packing and placement provides a higher quality placement solution in general [Chen04].

For each *iteration* in the algorithm an unlocked element is picked at random, then a destination location suitable for the element is also selected at random. The element is moved to this location and then accepted or rejected based on the change in cost. The placer must ensure that the randomly chosen destination is the same element-type.

Furthermore, the placer must ensure that a destination that has another circuit element locked to that location is not chosen. These two considerations make it difficult to use the dynamic range limit implemented by previous approaches. It has been reported that removing the range limit reduces the performance by around 10% [Egur05]. Therefore the range limit has been removed from our placement algorithm to cope with a complex set of suitable destination locations. The adaptive probability based range limiting suggested by Eguro et al. [Egur05], could be added to improve performance.

3.5.4 Router

Each net in the net-list has one source and one or more sinks. The placer has provided a location for each sink and source in the net-list. The task of the router is to map the net-list signals to the routing graph using each node no more than once.

The router developed for use in this thesis is based on the breadth first negotiating congestion driven routing algorithm from VPR [Betz99a]. The VPR router is based on the *PathFinder* algorithm by McMurchie et al. [McMu95]. The original *PathFinder* routing algorithm is outlined in subsection 2.3.5. The modifications made to the original algorithms are outlined in this section.

As mentioned in subsection 3.3.3 the delay in a fully buffered interconnect fabric is a function of the number of *wire hops*. Therefore, the base cost of all wire nodes, b_n , (defined in subsection 2.3.5), is set to 1.

In order to support the wire policy constraints, each node in the routing resource graph is marked as internal, reserved, input or output. Before routing begins the wire use policy and the assigned interface definitions are used to identify and mark reserved, input and output nodes. The router has been modified so that both a resource pin and a wire node can be marked as a sink or source. During routing no reserved nodes are used. Input and output nodes are only used if they are specified as sources or sinks respectively.

The router already checks that a fanout node meets the net bounding box conditions before adding it to the priority queue. In the same way, the router was modified to check whether a node meets the wire policy constraints before adding it to the priority queue.

3.5.5 Discussion

In the previous sections, we have outlined the development of a framework and supporting tools for a general model of an application specific system on FPGA that exploits reusable components. However, a number of issues were raised, including the complex relationship between the interface bandwidth used, congestion and the resulting performance of the mapped circuit. The implementation of the new framework within an FPGA modelling environment, along with the supporting tool-set will facilitate experiments that will answer these questions. These experiments comprise the remainder of this thesis.

4 Evaluation

In this chapter we explore several approaches to mapping a system of components to an FPGA fabric. Every component in the system is synthesised into a net-list. Each component is annotated with its dimensions and the locations of its ports. Two directives are used to build a system: The “New” directive places components; and the “Link” directive specifies a pair of component ports that are to be connected.

The system compiler then performs the following steps to *allocate* circuit elements to the FPGA architecture: The *primitive allocation* packs *logic elements* with look-up tables and flip-flops; the *resource allocation* tries to optimise the position of these elements within the two-dimensional *resource* array so that the total estimated net length is minimised; and finally *connection allocation* attempts to find optimal paths through the interconnect for each connection described in the net-list.

The system is presented to the compiler as a set of components and instructions on how to connect them. Each component is a separate net-list and separate resource footprint. The system is *merged* by combining all component instances into a single net-list and creating a single resource footprint. A system can be *merged* after any one of the above allocation stages.

4.1 Experimental Approach

In order to judge the impact of pre-routing components we compare three approaches to system construction:

4.1.1 Normal Approach

The first, the normal approach, merges the system after *primitive allocation* and then performs *resource allocation* and *connection allocation*. The normal approach corresponds to the conventional FPGA design flow where a net-list of primitives is packed, placed and routed globally without any hard partitioning during allocation. The normal approach is used as a baseline for comparison as it is considered to achieve the highest quality result.

The normal approach, where the system is *merged after primitive allocation*, is referred to as the **MPA** approach in this chapter.

4.1.2 Pre-Placed Components Approach

The second, the pre-placed components approach, performs *primitive allocation* and *resource allocation* on each component in isolation, then merges the system, and finally performs *connection allocation*. The pre-placed components approach is similar to the conventional region constrained design flow. The main difference that our pre-placed components approach has over the conventional region constrained design flow is that placement guides are provided for resources that have external component connections. The placement guides, which are derived from the interface definition, allow a component resource allocation to be performed in complete isolation from any other component. Isolated optimisation without the guides would result in the two resources that connect to

an inter-component net being far from each other. In a conventional region constrained design flow, the resource allocation would be done globally to avoid this situation.

Comparing the normal approach with the pre-placed components approach allows us to observe the impact of component region constraints and the impact of providing placement guides for ports.

The pre-placed components approach, where the system is *merged after resource allocation*, is referred to as the **MRA** approach in this chapter.

4.1.3 Pre-Routed Components Approach

The third, the pre-routed components approach, performs *primitive allocation*, *resource allocation* and *connection allocation* on each component in isolation before merging components into a system. Our pre-routed components approach is similar to region constrained partial reconfiguration design flows. The unique aspect of our approach is that inter-component wires are used as the isolation point rather than previous approaches that have used resource either side of an interface as the isolation point.

Comparing the pre-routed components approach with the normal approach will highlight the impact that routing components in isolation has on the quality of a compilation result. Comparing the pre-routed and pre-placed approaches provides us with a view of the additive impact on mapping quality that the pre-routing approach has over the pre-placed approach.

The pre-routed components approach, where the system is *merged after connection allocation*, is referred to as the **MCA** approach in this chapter.

4.2 Synthetic Component Generation

This section introduces the synthetic circuit generator, called GNL [Stro00], and introduces Rent's rule [Land71], that is used both to classify circuit interconnect complexity and to guide the synthetic circuit generation process.

To evaluate the different system compilation approaches, synthetic circuits are created and mapped into components. Using synthetic circuits allows us to control the routing complexity of a system. The synthetic circuits are packaged as components and presented to the system compiler for mapping to the target FPGA architecture.

A circuit is represented by a set of interconnected *blocks*. Each block has a number of *terminals*. A connection between blocks is called a *net*. A *net* has one *source pin* that connects to one block output *terminal* and one or more *sink pins* that connect to block input terminals. A net that connects to more than two blocks is called *multi-pin net*. Nets that connect to blocks inside the circuit are called *internal nets*. Net pins that connect outside the circuit are called *external pins*. In order to correctly represent these external pins they are represented as a type of virtual block. At the next level of hierarchy up, a circuit is viewed as a block with the external pins mapping to terminals. Thus, a hierarchical view of a complete design may be represented.

The *interconnect complexity* of a circuit strongly affects the effort required to find a reasonable connection allocation. Differences in interconnect complexity across different circuits were observed by Rent and his observations have been captured by Rent's rule

[Land71]. This rule dictates the relationship between the number of blocks in a circuit and the number of circuit terminals.

$$T = T_b B^p \quad (4.1)$$

Equation (4.1) defines the number of external circuit terminals, T , as a function of the average number of terminals per block, T_b , the total number of circuit blocks, B , and the Rent exponent p . The Rent exponent is a measure of interconnect complexity of a circuit and is bounded between 0 and 1, with increasing values for increasing interconnect complexity. The typical range of the Rent exponent is between 0.47 and 0.75 [Russ72].

Dividing a circuit into disjoint sub-circuits, or partitions, is called *partitioning*. Each partition contains a subset of the circuit blocks. Partitions are created in such a way as to minimise the number of nets crossing partition boundaries.

Rent's rule predicts that, as a circuit is successively partitioned into smaller and smaller partitions, the interconnect complexity, represented by the Rent exponent p , will remain constant across all levels of the hierarchy. The validity of Rent's rule is a result of the fact that designers tend to build their designs hierarchically and impose the same complexity at each level of hierarchy. Having said this, it has been observed that the value of p can vary as the hierarchy of the design is traversed. One particular case is that values of T are less than that predicted by Rent's rule because of the practical limit on the number of package pads. This limitation tends to reduce the value of p above a certain value of B , known as the *second region* of Rent's rule.

Another important aspect of a circuit's interconnect is its *net degree distribution*. A net's *degree* is defined as the number of pins that the net has. The net degree distribution is the set of numbers that lists the quantity of each net degree in a circuit. The net degree distribution has been observed to follow a power law in real circuits [Stro98].

Partitioning is performed from the highest (top) level of the hierarchy towards the lowest (bottom) level of hierarchy, top-down. Clustering is the reverse of partitioning, starting from the lowest level of hierarchy and completing at the highest level of hierarchy, bottom-up. It was found that creating synthetic circuits by clustering a set of blocks made it easy to control the interconnect complexity and net degree distribution. Thus, the synthetic circuit generator, called GNL, generates circuits by clustering a predefined set of blocks to create nets that satisfy both Rent's rule and the power law for net degree distribution [Stro00].

The first step in creating a synthetic circuit is to define a set of primitive blocks. The following is defined for each block: a name, the number of input terminals, the number of output terminals and whether it is combinatorial or sequential. Next, the library distribution is defined as the number of each of these block types that are to be used in the synthetic circuit. From this information the average number of terminals per block, T_b , is calculated.

We define the *clustering level*, b , as the upper bound on the number of blocks in each cluster. The clustering level identifies a specific level in the circuit hierarchy. For a circuit containing B blocks, at the lowest clustering level, $b = 1$, there are B clusters of 1 block, at a clustering level $b = i$ there are B/i clusters of i blocks, and at the highest clustering level, $b = B$, there is a single cluster of B blocks.

Equation (4.2) defines the average number of terminals on a circuit at clustering level i , given the previous clustering level $i-1$, where B_i and B_{i-1} are the number of blocks per cluster at cluster level i and $i-1$ respectively. At the lowest clustering level, $b = 1$, where each cluster has exactly one block, T_1 is the average number of terminals per block across the set of blocks that are given to build the circuit. Changing the Rent exponent at a clustering level b creates a boundary between Rent regions. GNL allows the Rent exponent to be defined at any number of clustering levels.

$$T_i = T_{(i-1)} \left(\frac{B_i}{B_{(i-1)}} \right)^{(p_i)} \quad (4.2)$$

GNL is used to create synthetic circuits with specific properties that are then encapsulated as *components* for presentation to the system compiler. The blocks used at the lowest clustering level are logic elements with between 1 and 4 input pins and 1 output pin. The distribution of block pin combinations reported in [Betz05] was used, as shown in Table 4.1, producing an average $T_1 = 4.27$.

Table 4.1 Terminal distribution of basic library components

Block terminals	Library distribution
2	8%
3	12%
4	25%
5	55%

The circuits are packaged with a component template that adds information about the dimensions of the region that the circuit will be mapped to, the wire use policy to employ, and the location of each port and its interface type as outlined in section 3.4. The top level terminals on a synthetic circuit are mapped to the interface signals within each port on the component template. The information in the component template is used by the system compiler to correctly map the circuit within a system.

The dimensions of a component region are selected by: firstly accommodating the resource the component requires while attempting to maximise resource utilisation; and secondly accommodating the port areas in their required locations. Components generally have more internal connections than external connections. Therefore, the closer the aspect ratio of a component region is to being a square, the shorter the average distance the internal nets will have to reach. Thus, an attempt is made to keep the aspect ratio of components as near to square as possible.

4.3 Evaluation Metrics

A number of metrics are defined in this section and are used to indicate the performance of each approach to system compilation.

In order to provide an indication of a component's routing complexity for the synthetic circuits used, the Rent exponent is reported for each clustering level specified. Figure 4.1 shows the visualisation output from our experimental tool set. The blue cells are occupied resources, the grey grid is the interconnect, where the individual tile switch boxes are in white. The green lines are the unrouted nets. Two designs are shown, illustrating the

difference in routing complexity for a system with a low routing complexity, where $p = 0.1$, and a system with a high routing complexity, where $p = 0.9$.

Figure 4.1 Post placement view of un-routed signals for systems with (a) $p = 0.1$, (b) $p = 0.9$

The size and shape of a component will effect the quality of a compilation result. Therefore, dimensions of each component as well as the number of pads and logic elements used in a component is recorded.

The routing difficulty predictor (W_{MIN} , equation (2.2)) is calculated after resource allocation. W_{MIN} provides an estimate of the minimum wire channel bandwidth requirement. Observing an increase in W_{MIN} from one system compilation approach to another indicates an increase in routing congestion.

An important aspect of the pre-routing technique proposed in this thesis is the potential to reduce the compile time through partitioning of the placement and routing optimisation problem. Therefore, we record the placement and routing effort required for each of the three system compilation approaches.

A single resource allocation iteration is defined as one swap attempt by the placer (see subsection 3.5.3 for further details). We record the total number of resource allocation iterations, I_{RA} , required to optimise the placement of a system. Note that, for the MRA and MCA approaches, I_{RA} is the sum of resource allocation iterations required to optimise the placement of each unique component in a system, and for the MPA approach, I_{RA} is the total resource allocation iterations required to optimise the placement of the merged system.

A single connection allocation iteration is defined as one router node expansion step (each iteration of loop steps 7 to 12 of the negotiating congestion algorithm outlined in subsection 2.3.5). We record the total number of connection allocation iterations, I_{CA} , required to route a system. Note that, for the MCA approach, I_{CA} is the sum of connection allocation iterations required to route each unique component in a system, and for the MPA and MRA approaches, I_{CA} is the total connection allocation iterations required to optimise the routing of the merged system.

The compile time is weighed against the impact on the performance of the resultant system. We measure the critical path length, W_{LP} , as the number of wires used in the longest sink to source path. Observing an increase in W_{LP} from one system compilation approach to another indicates a reduction in system performance.

The total number of wires used in the system, W_U , provides an indication of interconnect utilisation. This should correspond with the routing complexity of the circuits that have been mapped. Observing an increase in W_U from one system compilation approach to another indicates an overall increase in path length.

4.4 Target Architecture Parameters

This section reports on the specific parameters of the target FPGA architecture used in the following studies.

The resource architecture used in this evaluation has logic resource *tiles* that contain four *logic elements* and IO resource tiles that contain four *IO pads*. Each logic element has a 4-input LUT, a flip-flop, and carry chain logic. The IO resource tiles are in vertical columns reaching from the top to the bottom of a device. Both logic and IO resource tiles have the same interconnect infrastructure with an X and Y wire channel and a interconnect box which facilitates connection between wires on the tile and the local tile resource.

The interconnect architecture used in this evaluation has a channel width of $W_{FPGA} = 40$. The channel is composed of 20 unidirectional wire sets of $W = 2$, 10 in each direction, and 10 bi-directional $W = 3$ wire sets. A maximum switch box flexibility of $F_S = 8$ was allowed. The actual switch box flexibility is between $F_S = 6$ and $F_S = 7$ wires. The resource output connection flexibility is $F_{CO} = 12, 14$, and 16 wires. The resource input connection flexibility is $F_{CI} = 8$ wires. For more background and definitions of the terms for the interconnect see subsection 2.2.3. For more information on how the interconnect is defined see subsection 3.2.3.

The Rent exponent of an island style FPGA is fixed at $p = 0.5$. The number of interconnect wires crossing a single tile is $2 \times W_{FPGA}$ (comprising the X and Y channel). This value is substituted in as T_I for the target FPGA architecture. For an island-style FPGA interconnect architecture, Rent's rule predicts the rate of increase in supported terminals, T_{FPGA} , as a function of B tiles as:

$$T_{FPGA} = (2 \times W_{FPGA}) B^{0.5} \quad (4.3)$$

When the Rent exponent of a circuit is higher than $p = 0.5$, the rate at which the number of circuit terminals increases with respect to the number of circuit elements will be higher than that of the FPGA architecture. Thus a fixed interconnect architecture can support a circuit with a high degree of connection complexity, but only up to a finite number of elements.

It has been shown that $3b+2$ logic resource terminals is sufficient to achieve full connectivity to a cluster of b logic elements [Betz99], where each logic element is a four 4-input LUT/register combination. It was found that the maximum number of terminals required on a cluster of four logic elements was 13 [Betz99].

Figure 4.2 illustrates the limit that an island style FPGA architecture places on the interconnect complexity (p) of a circuit. Rent's rule (Equation (4.1)) is used to predict the number of circuit terminals (T) required for circuits packed into B logic resource tiles.

Figure 4.2 The predicted limit on circuit terminals (T) that an example island style FPGA architecture places on circuits of varying interconnect complexity (p)

The number of terminals required is plotted for interconnect complexities (p) between 0.1 and 0.9. In all cases the average number of terminals per tile (T_l) is 13. The number of terminals supported by an FPGA (T_{FPGA}) with B tiles and $W_{FPGA} = 40$ is plotted using Equation (4.3).

T_{FPGA} must be higher than the total number of circuit terminals to achieve a successful mapping. At 100 tiles, the number of terminals of the $p = 0.9$ curve exceeds that supported by the FPGA architecture. For the $p = 0.8$ case, this occurs at 441 tiles. Thus, Rent's rule sets upper bounds for circuit size and complexity that an interconnect architecture can support. When congestion effects are considered, the circuit size will be somewhat smaller than this theoretical maximum.

4.5 Target Architecture Characterisation

Given that we have created a new architectural model it is important to try and characterise its performance. In this section we produce 9 benchmark circuits (created by specifying a particular Rent exponent $p = 0.1$ to $p = 0.9$) and map them to the target architecture specified in 4.4.

Rent's rule predicts an exponential relationship between the number of circuit elements and the number of terminals. The array size used in this study is kept at a size that, for the given wire channel width, can support a reasonable range of routing complexity. In this

architectural characterisation we have used an array that is 28 tiles by 24 tiles. It has four IOB columns providing 384 IO pads and 576 logic tiles, each with four logic elements, and a $W_{FPGA} = 40$.

Nine synthetic circuits were created, each with 2000 logic elements (85% logic resource utilisation) and increasing levels of routing complexity between $p = 0.1$ and $p = 0.9$. We saw in the previous section that such an architecture can only support a circuit with a Rent exponent of $p = 0.9$ up to a 100 logic tiles (400 logic elements). Because we require a set of synthetic benchmarks with varying Rent exponent up to $p = 0.9$ we specify the Rent exponent at a clustering level of $b = 100$ logic tiles to fit the predicted maximum number of tiles supported by an architecture with $W_{FPGA} = 40$.

Figure 4.3 shows the component mapping difficulty versus it's Rent exponent. From Figure 4.3 it can be observed that the routing difficulty predictor (W_{MIN} , equation (2.2)) indicates an increased difficulty with an increased Rent exponent. The number of routing iterations, to completely route the design, closely tracks the difficulty predictor as does the total number of wire segments used to connect the design.

The mapping tools successfully mapped the $p = 0.9$ case which had a $W_{MIN} = 44.6$. This represents a circuit that required 111.5% of W_{FPGA} to achieve a successful connection allocation. While the predictor suggests an impossible routing problem the compiler succeeded in connection allocation. Being able to successfully map a circuit where $p = 0.9$ at a cluster level 400 elements (100 logic resource tiles) to an interconnect architecture with $W_{FPGA} = 40$ is in accordance with the predictions using Rent's rule in the previous section. Thus, the combination of mapping algorithms and FPGA architecture is able to perform at least as well as the theory predicts.

Figure 4.3 Mapping difficulty with increased Rent exponent

4.6 Interface Bandwidth Study

In this section we will investigate the impact on the quality of compilation when a system is split into components that are compiled in complete isolation. Five sets of benchmark circuits are created ($B = 16, 36, 64, 100$, and 256). The circuits in a specific set fill a specific number of tiles. Each set contains four circuits each built to use a specific wire interface bandwidth ($W_{IF} = 10, 20, 30$, and 40%). In total, twenty

benchmark circuits are used in this section. Each circuit is mapped to two complimentary components that make up a simple system.

The pre-routing technique proposed in this thesis, unlike existing techniques [Sedc04] [Kalt04], [Xilap290], does not, in theory, require any extra resource area to facilitate pre-routing. However, both the previous approaches to pre-routing and the technique proposed in this thesis will suffer from more subtle effects such as congestion and loss of cross-optimisation potential. In much the same way as locking IO signals to pad locations increases routing congestion, locking inter-component signals to bisected wires also has the potential to increase congestion. Splitting a system into components which are compiled in complete isolation prevents the compiler from performing optimisation of resource allocation or connection allocation across component boundaries. This loss of *component cross-optimisation potential* will affect the quality of the compilation results. The rest of this chapter investigates the impact that the proposed pre-routing technique has on compilation quality. In chapter 5, a real world application is used to investigate the practicalities of the proposed pre-routing technique. In subsection 5.4.1, enough evidence has been gathered to compare the proposed technique with previous pre-routing approaches at a conceptual level.

4.6.1 Interface Bandwidth Utilisation

We used a simple system template to evaluate the impact of pre-routing. The system template has two components, each with one port, connected by a single link. The system is described to the compiler in XML as illustrated in Figure 4.4. In effect it is a system that has been partitioned once. The two halves are optimised independently and then merged in accordance with either the MPA, MRA, or MCA system construction approach.

```
<mad>
  <new module="b16_px1" instance="u1" x="2" y="1"/>
  <new module="b16_nx1" instance="u2" x="6" y="1"/>
  <link instancea="u1" porta="px1" instanceb="u2" portb="nx1"/>
</mad>
```

(b)

Figure 4.4 Simple two component single link system (a) Schematic and (b) XML description of an example system

The two components are created from the same base circuit. A set of benchmark circuits are created by varying two parameters: the number of circuit elements; and, the interface bandwidth used to connect another component.

Increasing the number of circuit elements implies that the area of a component region must expand. As the region area grows so does the number of path options for internal signal routing, and thus the routing flexibility increases with region size. Furthermore, the more elements in a components' circuit, the more capacity it has for producing interface signals. Countering this is the fact that the placer could place elements that are connected to an interface further away from the interface surface. This behaviour would result in larger increases in critical path length.

We define the interface surface length, E , as the number of tiles along an interface surface. The interface bandwidth, W_{IF} , is the percentage of W_{FPGA} used across the interface surface and is calculated as:

$$W_{IF} = \frac{T_{IF}}{E \times W_{FPGA}} \times 100 \quad (4.4)$$

Where T_{IF} is the number of component terminals mapped to signals in the interface. Note that, because we are mapping all of a components terminals to one interface, T_{IF} is equal to T as dictated by equation (4.1). As the interface bandwidth utilisation increases, the number of signals crossing the interface surface increases. Thus, more signals are constrained in the port area of each component, increasing the potential congestion.

A set of benchmark components have been created for various combinations of region size and interface bandwidth. Table 4.2 shows the interface bandwidth, W_{IF} (as a percentage of W_{FPGA}), region area in logic tiles, B , region dimensions, X , Y , percentage tile occupancy, B_{OC} , total interface terminals, T , and Rent exponent, p , of the benchmark components used in this study. Synthetic circuits are then produced that yield near 100% resource utilisation for the various region areas.

Table 4.2 Key benchmark component characteristics

B	X, Y	B _{OC}	W _{IF} = 10%		W _{IF} = 20%		W _{IF} = 30%		W _{IF} = 40%	
			T	p	T	p	T	p	T	p
16	4,4	98.44%	16	0.32	32	0.49	48	0.58	64	0.65
36	6,6	99.31%	24	0.35	48	0.49	72	0.57	96	0.63
64	8,8	99.61%	32	0.36	64	0.49	96	0.56	128	0.61
100	10,10	99.75%	40	0.37	80	0.49	120	0.56	160	0.61
256	16,16	99.61%	64	0.39	128	0.49	192	0.55	256	0.59

Before creating the interface definitions, we first explored several heuristics for interface wire allocation. Four heuristics were created:

W2 uses $W = 2$ wire sets;

W2W3.1 uses $W = 2$ wire sets and wire 1 from $W = 3$ wire sets;

W2W3.2 uses $W = 2$ wire sets and wire 2 from $W = 3$ wire sets; and

W3 uses both wires 1 and 2 from $W = 3$ wire sets.

The $B = 36$, $T = 24$ and $B = 36$, $T = 48$ benchmark systems were then mapped using interface allocations that use the four different heuristics. Table 4.3 shows the heuristic details and the resultant critical path lengths for both the MPA and MCA approaches when using the different interface allocations. The “*” in Table 4.3 indicates that the wire set indicated is used in the heuristic, while the “-” indicates that the wire set indicated is not used.

Table 4.3 Interface wire allocation heuristic and resultant critical path lengths

	Allocation (Set size W , Wire)			$B = 36, T = 24$		$B = 36, T = 48$	
Heuristic	2, 1	3, 1	3, 2	$W_{LP(MPA)}$	$W_{LP(MCA)}$	$W_{LP(MPA)}$	$W_{LP(MCA)}$
W2	*	-	-	9	12	11	12
W2W3.1	*	*	-	9	13	11	12
W2W3.2	*	-	*	9	13	11	16
W3	-	*	*	9	11	11	11

The quality of the results for the MPA approach are not affected by the interface allocation but they do provide a basis for comparison with the MCA approach. The results indicate that the W3 heuristic provides the lowest increase in critical path for the MCA approach compared to the MPA approach. Therefore, we will use the W3 heuristic for the rest of this study.

Interface definition templates that use 4, 8, 12, and 16 signals per tile were created. This translates to a W_{IF} equal to 10, 20, 30 and 40% of W_{FPGA} respectively. From these templates, interface definitions were then created for combinations of W_{IF} and T . The synthetic circuits have exactly the right number of terminals to meet the interface bandwidth, with half the signals defined as inputs and half as outputs.

The maximum depth a bisected wire reaches into a component region is $W-1$ because it must reach at least one tile into another component region. Thus, as the interconnect architecture used only has wires of length $W = 3$ and $W = 2$, the port area can only be a maximum of 2 tiles deep. Therefore, the dimensions of the port areas used in this benchmark set are $2 \times E$ tiles. This places a limit on the number of elements that may be placed in a port area. While the limit does not prevent the port placement guides from functioning it does add to potential congestion in a component when the limit is exceeded. Table 4.4 records the percentage of a component's resources that are connected to an interface, B_{IF} , and the percentage of tile occupancy within the port area, B_{PA} , for each benchmark in the set.

Table 4.4 Benchmark characteristics: Interface connected elements and element locations in port area

B	X, Y	B_{OC}	$W_{IF} = 10\%$		$W_{IF} = 20\%$		$W_{IF} = 30\%$		$W_{IF} = 40\%$	
			B_{IF}	B_{PA}	B_{IF}	B_{PA}	B_{IF}	B_{PA}	B_{IF}	B_{PA}
16	4,4	98.44%	25	50	51	50	76	50*	102**	50*
36	6,6	99.31%	17	33	34	33	50	33*	67	33*
64	8,8	99.61%	13	25	25	25	38	25*	50	25*
100	10,10	99.75%	10	20	20	20	30	20*	40	20*
256	16,16	99.61%	6	12.5	13	12.5	19	12.5*	25	12.5*

* Port area limit exceeded

** More than 100% indicates resources connect to more than one interface signal

When the port area limit is exceeded elements have more chance of moving outside of the port area, increasing the impact on the critical path length. The port area limit is exceeded for $W_{IF} > 20\%$ of W_{FPGA} .

The components with a small region size are pushed to produce the number of terminals to achieve a W_{IF} of 40%. For the smallest component, $B = 16$, with the highest W_{IF} every element in the component is connecting to the interface at least once.

For the system template used, the MRA and MCA approaches effectively partition the placement problem into two isolated component regions. Comparing the routing difficulty predictor, W_{MIN} , reported for MPA and MRA indicates the effect this has on the difficulty of the routing problem produced. The results in Table 4.5 show that the routing difficulty increases both as the number of elements in a component increases and as the interface bandwidth increases.

Table 4.5 Benchmark component routing difficulty predictor values reported for the MPA and MRA compiler approaches

B	16		36		64		100		256	
W_{IF}	MPA	MRA	MPA	MRA	MPA	MRA	MPA	MRA	MPA	MRA
10%	17.5	18.3	19.1	19.3	20.3	20.7	21.7	21.5*	24	23.5*
20%	19.4	21.1	22.1	23.1	23.7	24.6	25.2	26.1	28.5	28.9
30%	20.8	24.2	24.2	25.8	26.3	27.5	28	29.1	31.6	33
40%	21.6	26.9	26	29.5	28.8	30.9	30.7	32.3	34.7	37

* Cases where MRA reduces the predicted routing difficulty

Using the MRA approach increases W_{MIN} on average 0.23, 1.0, 1.9, and 3.3 for a W_{IF} of 10, 20, 30, and 40% of W_{FPGA} respectively. All the benchmarks have their routing difficulty classified as low stress (see subsection 2.3.5) when using the MPA approach. Only the $B = 256$, $W_{IF} = 40$, changes classification from low-stress when using the MPA approach to difficult when using the MRA approach.

Note that the resource allocation result from the MRA approach is re-used in the MCA approach. Therefore, a loss of placement quality in the MRA approach will have an effect on the overall quality of the MCA approach.

With regards to compiler effort, the MRA approach sees an average reduction of 23.13% in resource allocation iterations (I_{RA}). The MRA approach results in an average increase of 2.4% in connection allocation iterations (I_{CA}) when compared to the MPA approach. The increase in routing effort required is explained by an average increase in W_{MIN} of 6%.

The MCA approach provides an average reduction of 21.73% in connection allocation iterations when compared to the MPA approach. The MCA approach also benefits from the 23.13% reduction in resource allocation iterations that the MRA approach yields.

Table 4.6 shows the compiler effort, I_{CA} and I_{RA} , averaged over benchmark cases with the same value of B tiles. It can be seen that, as component size (B) increases so do both I_{CA} and I_{RA} .

Table 4.6 Compiler effort averaged over cases with the same value of B

B	$I_{CA(MPA)}$	$I_{CA(MCA)}$	$I_{RA(MPA)}$	$I_{RA(MCA)}$	$(I_{CA(MCA)} - I_{CA(MPA)})/I_{CA(MPA)}$	$(I_{RA(MCA)} - I_{RA(MPA)})/I_{RA(MPA)}$
16	553388	347691	435735	346455	-0.37	-0.20
36	2053149	1516757	1359576	1050977	-0.26	-0.23
64	4732502	3753520	2963690	2263380	-0.21	-0.24
100	12934467	10950750	5457697	4038076	-0.15	-0.26
256	32040961	29615932	19334835	14884395	-0.08	-0.23

The fractional difference in the I_{RA} of the MPA approach and the I_{RA} of the MCA approach $((I_{RA(MCA)} - I_{RA(MPA)})/I_{RA(MPA)})$ shows that the reduction in placement iterations stays roughly constant as the size of component (B) varies. The magnitude of the fractional difference in I_{CA} reduces with increased components size. We expect this as when components increase in size, the number of internal nets increases at a higher rate than the number of external nets. Thus, the benefit of the MCA approach, that is partitioning the routing problem, diminishes with increased component size.

Table 4.7 shows I_{CA} and I_{RA} averaged over benchmark cases with the same value of interface bandwidth utilisation (W_{IF}). It can be seen that, increasing interface bandwidth utilisation does not have a strong effect on I_{RA} , for both the MPA and MCA approaches.

Table 4.7 Compiler effort averaged over cases with the same value of W_{IF}

W_{IF}	$I_{CA(MPA)}$	$I_{CA(MCA)}$	$I_{RA(MPA)}$	$I_{RA(MCA)}$	$(I_{CA(MCA)} - I_{CA(MPA)})/I_{CA(MPA)}$	$(I_{RA(MCA)} - I_{RA(MPA)})/I_{RA(MPA)}$
10%	5896601	4965032	5997735	4501744	-0.16	-0.25
20%	7589586	6404628	5842159	4557584	-0.16	-0.22
30%	8749414	7771941	5992036	4526255	-0.11	-0.24
40%	9899748	9282593	5809296	4481042	-0.06	-0.23

The increase in I_{CA} for the MPA approach increases with increasing W_{IF} is explained by the fact that interconnect complexity (p) increases with W_{IF} . The MCA approach sees a slightly higher rate of increase in I_{CA} , probably because of congestion caused by an increasing number of signals constrained to the port area. Because of this added congestion the fractional difference in the I_{CA} of the MPA approach and the I_{CA} of the MCA approach $((I_{CA(MCA)} - I_{CA(MPA)})/I_{CA(MPA)})$ diminishes with increased W_{IF} .

Figure 4.5 shows how the MRA approach impacts on the critical path length, W_{LP} , in the system when compared to the MPA approach. The absolute increase in the longest path length, $W_{LP(MRA)} - W_{LP(MPA)}$, is plotted against interface bandwidth utilization, W_{IF} , represented as a fraction of W_{FPGA} .

For three of the $B = 36$ cases and two of the $B = 256$ cases there is a reduction (improvement) in W_{LP} . For three of the $B = 16$ cases, one of the $B = 64$ cases, and two of the $B = 100$ cases there is no change in W_{LP} .

Figure 4.5 Absolute increase in the longest path length, W_{LP} , of the MRA approach compared to the MPA approach

The variation in the absolute increase in longest path length was 1, 1, 3, 3, and 7 for the $B = 16, 36, 64, 100$, and 256 cases respectively. The average increase in longest path length was 0.25, -0.75, 1.5, 1, and -0.75 for the $B = 16, 36, 64, 100$, and 256 cases respectively. The average increase had no discernible relation to component size. However, the variance increases significantly with component size. As component size increases, so does its interface surface. Spreading the port wires across a wider interface surface appears to increase the variability in the result of the MRA approach.

The graphs in Figure 4.6 show the impact of the MCA approach on the critical path length of the system, W_{LP} , when compared to the MPA approach. The absolute increase in the longest path length, $W_{LP(MCA)} - W_{LP(MPA)}$, is plotted against interface bandwidth utilization, W_{IF} , represented as a fraction of W_{FPGA} . The average increase in W_{LP} is 4.0 wire hops with a maximum increase of 10 wire hops. For the $W_{IF} = 40\%$ cases the absolute increase varies between 3 and 10 hops.

Figure 4.6 absolute increase in the longest path length, W_{LP} , of the MCA approach compared to the MPA approach.

Table 4.8 shows the critical path length (W_{LP}) averaged over benchmark cases with the same value of component size (B). The critical path length increases with B for both the MCA and MPA approaches. It appears that the fractional increase in critical path length for the MCA approach over the MPA approach ($(W_{LP(MCA)} - W_{LP(MPA)})/W_{LP(MPA)}$) reduces with component size. However, the absolute difference remains roughly constant across the range of component size. The apparent reduction in fractional difference is explained by the fact that the overall critical path lengths are increasing for both approaches, while the absolute difference remains roughly constant. In fact the absolute difference in W_{LP} is highest for the $B = 16$ and $B = 256$ cases. This suggests that the small component size, with less interconnect area, and therefore less interconnect flexibility is not so able to cope with congestion. As B increases, interconnect flexibility increases, coping better with the congestion that the MCA approach introduces. Again, as a component increases in size so does its interface surface. Nets are spread across the interface surface, causing the critical path length produced by the MCA approach to increase further.

Table 4.8 Critical path length averaged over cases with the same value of B

B	$W_{LP(MPA)}$	$W_{LP(MCA)}$	$W_{LP(MCA)} - W_{LP(MPA)}$	$(W_{LP(MCA)} - W_{LP(MPA)})/W_{LP(MPA)}$
16	7	12	5	0.59
36	11	13	2	0.23
64	13	16	3	0.26
100	15	21	6	0.37
256	25	30	5	0.18

Table 4.9 shows critical path length (W_{LP}) averaged over benchmark cases with the same value of interface bandwidth utilisation (W_{IF}). The critical path lengths produced by the MPA approach show a small increase with increased interface bandwidth utilisation.

Table 4.9 Critical path length averaged over cases with the same value of W_{IF}

W_{IF}	$W_{LP(MPA)}$	$W_{LP(MCA)}$	$W_{LP(MCA)} - W_{LP(MPA)}$	$(W_{LP(MCA)} - W_{LP(MPA)})/W_{LP(MPA)}$
10%	13	16	3	0.26
20%	13	16	3	0.24
30%	16	19	3	0.22
40%	15	21	6	0.41

However, the critical path lengths produced by the MCA approach show a larger increase with increased interface bandwidth utilisation. Therefore, port area congestion is probably playing a larger part in affecting the critical path length in the MCA approach. The absolute increase that the MCA approach introduces into the critical path length is constant except for the $W_{IF} = 40\%$ cases where it increases.

4.6.2 Port Area Shaping

In this subsection we take 6 benchmarks from subsection 4.6.1 (the $W_{IF} = 10\%$ and $W_{IF} = 20\%$ from the $B = 16$, $B = 64$ and $B = 256$ sets). The port areas of each are first compressed and then positioned at different points to investigate how the shape and position of a port affects compilation quality.

Given a number of signals that is less than the total capacity of the interfacing edge, $W_{IF} \times E$, is it better to spread signals across the surface or to increase the tile bandwidth in a smaller *compressed* port region? If the port stretches across the entire interface surface then it only has one possible position. When a port length is less than the edge length, where is the best position to place it? To a certain extent the answer to these questions will be specific to the interfacing circuits. However, we will investigate the general effect of position to provide some general guidelines.

As we have seen so far, interface wire constraints to facilitate systems built from pre-routed components has had an adverse impact on the performance of the resultant system (when compared to a system built using a conventional, MPA, approach). Thus, it is interesting to investigate whether changing the geometry of the port areas will improve this. Figure 4.7 (a) shows the approach used so far where a port area spans the entire interface surface. Intuition suggests that this will “stretch” some nets out towards the corners of the components' region, increasing path lengths.

Figure 4.7 (a) Port area uses entire edge length (b) Port area compressed by a factor of 2 (c) Compressed port area positioning

Figure 4.7 (b) shows the same port area compressed by a factor of two which does not stretch nets as far as the uncompressed port area. Figure 4.7 (c) shows the possible positions of the compressed port area. A port area that is in the corner of a device (b) presents a higher potential for congestion than a centrally located port area (c) which has a higher potential to be closer to the natural centre of a circuit.

Table 4.10 shows the number of tiles, B , dimensions, X,Y , percentage occupancy of resource tiles, B_{OC} , circuit interface terminals, T , bandwidth utilisation of the uncompressed port, W_{IF} , edge length, E , bandwidth utilisation within the compressed port region, $W_{IF(C)}$, the edge length of the compressed port region, $E_{(C)}$, and the port positions used in each of the benchmark systems used in this study.

Table 4.10 Port compression and positioning benchmarks

B	X, Y	B_{OC}(%)	T	W_{IF}	E	W_{IF(C)}	E_(C)	Port Pos.
16	4,4	98.44	16	10%	4	20%	2	1,2,3
16	4,4	98.44	32	20%	4	40%	2	1,2,3
64	8,8	99.61	32	10%	8	20%	4	1,2,3,4,5
64	8,8	99.61	64	20%	8	40%	4	1,2,3,4,5
256	16,16	99.61	64	10%	16	20%	8	1,2,3,4,6,8
256	16,16	99.61	128	20%	16	40%	8	1,2,3,4,6,8

A system of two similar components was created for each of the benchmarks listed above. They were mapped to the target architecture using the three approaches as before. Figure 4.8 (a) shows the increase in critical path length of systems mapped using the MCA approach as compared to the MPA approach, $W_{LP(MCA)} - W_{LP(MPA)}$. Figure 4.8 (b) shows the average increase in critical path length. Note that the left-most points on the graph, plotted at a negative normalised position, are actually the results for the systems that use an uncompressed port area.

It can be seen from Figure 4.8 (b) that that performance is reduced when a port area is compressed towards one corner of a component. However, there is a compressed port area position that produces an MCA result that is either the same as or better than the uncompressed port area. Generally we see that a centralised port location reduces the impact of the MCA approach on the critical path length.

From this we can conclude that compressing a port area and correctly positioning it will improve the MCA approach. The results concur with our intuition, that a centrally located port area is better than one placed in a corner. We find from this study that the adverse effect on critical path length that pre-routing suffers is reduced from a 40% increase to a 10% increase by using compressed and positioned ports.

(a)

(b)

Figure 4.8 (a) Increase in critical path length $W_{LP(MCA)} - W_{LP(MPA)}$ and (b) Average increase in critical path length over all benchmarks plotted against normalised port position

4.6.3 Component Region Shaping

In this subsection we take the $B = 64$, $W_{IF} = 20\%$ benchmark system from subsection 4.6.1 and adjust the shape of the region to investigate the effects.

As the aspect ratio of a component, with an area of B tiles, tends from a square towards a single line of tiles, the perimeter increases from $4 \times \sqrt{B}$ to $2B+2$. Consider that instead of increasing interface bandwidth utilisation, the interface surface is “stretched” to

accommodate an increased number of signals. It is important to note that, while two sides become long, the other two sides become shorter. Thus, only two sides are able to support interface surfaces with a larger number of signals.

However, it is usual that a component has more internal connections than external connections. Therefore, the closer the aspect ratio of a component region is to square, the shorter average distance the internal nets will have to reach. Conversely, stretching a component will increase internal net lengths and therefore increase the critical path length.

The $B = 64$, $T = 64$ benchmark system has an interface surface length $E = 8$, and uses an interface bandwidth of 20% of W_{FPGA} . We have re-mapped it with region dimensions where the interface surface length is $E = 4$. This forces the interface bandwidth utilisation to double to 40% of W_{FPGA} in order to support the same number of signals. We have then re-mapped it again with region dimensions where the interface surface length is $E = 16$. This allows us to reduce the interface bandwidth utilisation by half, making it 10% of W_{FPGA} , while still being able to support the required number of interface signals. Table 4.11 shows the characteristics of these benchmarks, the total wires used, W_U , and the critical path length, W_{LP} , for both the MPA approach and the MCA approach.

Table 4.11 Aspect ratio benchmark characteristics and results

E	X, Y	B	B _{CO}	T	W _{IF}	W _{U(MPA)}	W _{U(MCA)}	W _{LP(MPA)}	W _{LP(MCA)}
4	16,4	64	99.61%	64	40%	2408	2221	19	23
8	8,8	64	99.61%	64	20%	2025	2016	13	15
16	4,16	64	99.61%	64	10%	2026	2154	11	21

For the $E = 4$ case, the critical path of the MPA approach was increased by 46%. This is expected because the region of the entire system was stretched to 4 by 32 tiles. For the $E = 16$ case we see the result of the MPA approach has improved by 15% because the system region was 16 by 8 tiles. Therefore, we expected the performance of the MCA to improve. However, changing the aspect ratio from square increased the critical path length when using the MCA approach. Furthermore, the impact of the MCA approach gets worse, it rises from an increase of 15% over the MPA approach for the $E = 8$ case to an increase of 21% and 90% for the $E = 4$ and $E = 16$ cases respectively. The MCA approach adds congestion to the $E = 4$ case. The increase in critical path length seen in the $E = 16$ case is because nets are spread across a wider interface surface effectively stretching them out.

Results from the port area shaping study suggest that it is better to have a centralised port area that is not stretched across the interface surface. We mapped the $E = 16$ case with a port that had a compressed length of 8 tiles requiring a $W_{IF} = 20\%$ of W_{FPGA} placed centrally along the interface surface. This reduced the impact of the MCA approach on the critical path from an increase of 90% to an increase of 15%.

4.7 Tunnelling bandwidth Study

In the previous section we have explored varying the interface bandwidth. In this section we will explore the effect of reserving wires for tunnelling links through components as outlined in subsection 3.4.6.3.

The model of interface extension we have presented so far is similar to planar isolation (see subsection 2.4.2 for more detail). The computing resource in the area that the interface extension occupies is wasted. Planar isolation has been shown to be non-scalable (see subsection 2.4.2 for more detail). While the new approach presented in this thesis has some similarities with planar isolation presented by Tessier [Tess99], it has the advantage that neighbouring components can be connected directly without the need for a routing-only area.

Instead of isolating component connections at the boundary of each component, Domain-based isolation ([Tess99] see subsection 2.4.2 for more detail) only allowed 30% of W_{FPGA} for internal components. The remaining 70% of W_{FPGA} was reserved from internal component routing to create connections between components. Tessier found that W_{FPGA} had to be increased by 50% to route a benchmark system using domain-based isolation.

Rather than putting connections into only two categories: internal and external, we have effectively split the external category into non-neighbour and neighbour connections. By restricting the placement flexibility of neighbouring components we can make more effective use of internal routing resource. Furthermore, we move the overhead of routing from system composition time to component compile time.

We use the wire use policy to split the channel wire bandwidth W_{FPGA} into tunnelling bandwidth W_T , internal bandwidth W_{IN} and interface bandwidth W_{IF} , as shown in Figure 4.9.

Figure 4.9 Region wire bandwidth usage split between W_T , W_{IF} , and W_{IN}

Reservation completely removes wire sets from use before a component is routed, reducing the effective W_{FPGA} . Therefore, we expect that the interconnect complexity, p , supported will reduce as more wires are reserved for tunnelling. The channel wire bandwidth must be shared between tunnelling and interface usage. Thus, the theoretical limits are that both $W_T + W_{IF} \leq W_{FPGA}$ and $W_T + W_{IN} \leq W_{FPGA}$ must hold true. In practice, congestion effects will reduce the amount of W_{FPGA} that can be used for tunnelling before connection allocation fails.

4.7.1 Wire Reservation

In this subsection we will investigate the effect of wire reservation on compilation. The nine systems ($p = 0.1$ to $p = 0.9$) from section 4.5 are re-mapped using several different reservation policies.

To observe the general effect of wire reservation we have created a number of wire use policies that reserve 10, 20, 30, 40, 50, and 60 percent of W_{FPGA} for tunnelling. The wire set reservation of these policies is detailed in Table 4.12.

Table 4.12 Policy wire set reservation detail

% W_{FPGA} Reserved	Number of W=2 Sets Reserved	Number of W=3 Sets Reserved
0	0	0
10	0	2
20	0	4
30	8	2
40	4	6
50	4	8
60	8	8

The nine synthetic circuits with increasing levels of routing complexity between $p = 0.1$ and $p = 0.9$ from section 4.5 were re-mapped using the reservation policies. The graphs in Figure 4.10 and Figure 4.11 show the impact of wire reservation on the connection allocation performance for these nine benchmarks with increasing routing complexity.

As expected, increasing the number of wire sets reserved, reduces the routing complexity that can be supported because reserved wire sets cannot be used for routing a component, effectively reducing W_{FPGA} .

Figure 4.10 Resultant total wires used, W_U , for benchmarks with increasing routing complexity mapped with increasing numbers of wires reserved for tunnelling

Generally it can be seen that the total number of wires used increases with increased reservation. This suggests that the average path length is increased as reservation increases because congestion is forcing connections to take longer paths. Furthermore, as

the number of $W = 3$ wires is reduced, more $W = 2$ wires have to be used, two $W = 2$ wires are required to cover the distance of one $W = 3$ wire.

Figure 4.11 shows I_{CA} plotted against increasing routing complexity for each policy used. It is interesting to observe that the number of connection allocation iterations reduces with increased wire reservation. The number of iterations required for the $W_T = 10\%$ policy reduces by around 10% of the iterations required for the $W_T = 0\%$ policy. This behaviour holds up until the $p = 0.8$ benchmark, where the number of iterations for the $W_T = 10\%$ increases by 14% of that required by the $W_T = 0\%$ policy. When $W_T = 10\%$, the router fails to route the $p = 0.9$ benchmark. We observe that the number of connection allocation iterations, I_{CA} , reduces with increased reservation factor up until the benchmark before the router fails, where the I_{CA} increases. The reduction in I_{CA} is explained by the fact that reservation reduces the number of paths that the router has to explore. However, wire reservation can be increased to the point where congestion starts to increase router effort, ultimately resulting in routing failure.

Figure 4.11 Connection allocation iterations, I_{CA} , for benchmarks with increasing routing complexity mapped with increasing numbers of wires reserved for tunnelling

Figure 4.12 shows W_{LP} plotted against increasing routing complexity for each policy used. It can be seen from these results that critical path length will increase with an increased number of reserved wires. The average percentage increase, $100\% \times (W_{LP(W_T)} - W_{LP(W_T=0)}) / W_{LP(W_T=0)}$, was 6, 16, 26, 32, 84, and 52% for a W_T of 10, 20, 30, 40, 50 and 60% respectively.

Thus, wire reservation will degrade the run-time performance of the mapped circuits. It is important to note that the target architecture only has $W = 2$ and $W = 3$ wires. Therefore, we are forced to reserve potentially useful wires within a component's region.

Figure 4.12 Wires in longest path, W_{LP} , for benchmarks with increasing routing complexity mapped with increasing numbers of wires reserved for tunnelling

A real world FPGA interconnect architecture will typically have wire segments that are longer than 2 or 3. We expect that reserving the longer wires will have less of an impact on the run-time performance of mapped circuits than reserving the short wire segments.

4.7.2 Interfaces and Wire Reservation

We now explore the combination of wire reservation for tunnelling and pre-routed interfaces. We take the 20 benchmark systems (five sets: $B = 16, 36, 64, 100$ and 256 , each with four interface bandwidths) defined in subsection 4.6.1 and re-map them using several different reservation policies.

We found in subsection 4.7.1 that reserving 60% of the wire channel bandwidth caused connection allocation to fail in all but the simplest of benchmark circuits. Reserving 50% of the bandwidth allowed the mapping of circuits with a Rent exponent of 0.4. The $W_{IF} = 10\%$ systems from subsection 4.6.1 were mapped with increasing numbers of wires reserved for tunnelling. Considering that these circuits have a Rent exponent of between 0.32 and 0.39 and that they have less elements than the circuits used in subsection 4.7.1 we expect that connection allocation will succeed for all of these benchmark circuits.

Figure 4.13 shows the number of wires used in the critical path of each of the benchmarks, mapped using the MCA approach, and plotted against an increasing percentage of wire channel bandwidth reserved for tunnelling.

As component size increases so does the potential path length. Thus, it can be seen from Figure 4.13 that the benchmark circuits with larger region sizes (the B256 set) experience a greater reduction in performance (increase in critical path length). The average percentage increase in critical path length, $100\% \times (W_{LP(WT)} - W_{LP(WT=0)}) / W_{LP(WT=0)}$, for the benchmark circuits used was 4, 15, 11, 18, and 39% for a W_T of 10, 20, 30, 40, and 50% respectively.

Figure 4.13 Critical path length for the two component systems plotted against the percentage of channel bandwidth reserved for tunnelling

4.7.3 Complementary Policies

In this subsection we take the $W_{IF} = 10\%$ systems from each of the five benchmark sets defined in subsection 4.6.1 and create two complementary systems that use tunnelling links to connect through each other.

For two components to connect across a third component, via a tunnelling link, they must use interface wires that are reserved in the third component. Figure 4.14 illustrates two simple systems that may be identical except for the interface allocation and the wire reservation policy used.

Figure 4.14 Two sets of components with complimentary interface and reservation allocations

The first system consists of component A and B, linked via interface extension AB. The second system is comprised from component C and D, linked by interface extension CD. Interface extension AB uses wires reserved from components C and D allowing the extension to overlay these components. Interface extension CD uses wires reserved from components A and B allowing the extension to overlay these components. The restrictions on interface extension (outlined in subsection 3.4.6) must be observed.

The wire usage of a complementary pair of wire set reservation policies and interface definitions is tabulated in Table 4.13. We have used these to create complementary pairs of systems based on the $W_{IF} = 10\%$ systems from subsection 4.6.1.

Table 4.13 Interface allocation and reservation policy wire set usage

System	Reserved Sets	Interface Sets
AB	W=3 Sets 9, 10	W=3 Sets 1, 2
CD	W=3 Sets 1, 2	W=3 Sets 9, 10

We first composed the two sets of systems without the interface extension to see how the different reservation and interface allocations effects the critical path. We find that the critical path varied between -2 and +3 hops across the five benchmarks when using the MCA approach. We found that there was variation in the critical path length when using the MPA approach of between -3 and +3 hops. Since the MPA approach respects the reservation but does not use interfaces, this indicates that the variation is mainly attributed to the different wires reserved causing the router to find different paths through the interconnect fabric.

A link extension is required to connect across a component (as shown in Figure 4.14). As the interfaces use $W = 3$ wires (as discussed in subsection 3.4.6.1), a link extension will extend all of the paths between the two components by 2, 3, 4, 5, and 8 hops when crossing a component of area 16, 36, 64, 100, and 256 tiles respectively.

4.8 Discussion

In order to properly explore the impact of pre-routed components, we have created an instance of an FPGA architecture within our modelling environment. Using a range of synthetic circuits with predefined interconnect complexities (Rent exponent between 0.1 and 0.9) we have shown that the modelled architecture performs in line with theoretical predictions. In fact we find that the routing difficulty predictor is rather pessimistic, suggesting that a higher channel utilisation factor is necessary (we set it to 50%, whereas 60% may be more appropriate).

We have created automated mapping tools to support our proposed design methodology. We compare the traditional placement and routing approach to system compilation (MPA approach), an approach that only pre-places component resource (MRA approach) and our new MCA approach that pre-routes components. The combination of tools and architecture have been applied to a set of synthetic systems in order to study the difference in compile time that the three system building approaches offer. The synthetic systems used were particularly aggressive in that they required near 100% logic utilisation. There was no attempt to correctly partition the system into components so nets with a high fanout will have traces that pass across component boundaries. Circuits with

an interconnect complexity between $p = 0.32$ and $p = 0.65$ were mapped. Components had between 16 and 256 signals constrained to pass across a single link.

Variation in the routing difficulty predictor (W_{MIN}) from the MPA approach, where no port constraints are applied, to the MCA approach, where port constraints are enforced, suggests that the constraints reduce the quality of the placement solution which will result in added congestion. Despite the congestion effects, we find that we are able to achieve an interface bandwidth utilisation of up to 30% of W_{FPGA} with only an average increase in critical path length of 3 wire hops. Pushing the interface bandwidth utilisation up to 40% of W_{FPGA} had a higher average increase in critical path length.

While we did investigate different interface wire allocation heuristics for their performance, we made no attempt to optimise interface wire allocation to the components in the synthetic systems studies. Thus, it is expected that the interfaces will increase congestion significantly. However, both the increase in routing difficulty predictor and the increase in critical path lengths produced by the MCA approach were not excessive. The synthetic systems were randomly generated which makes optimal interface wire allocation difficult. We expect that with a designed system it is more feasible to plan the wire allocation of an interface and reduce the congestion effects seen by the MCA approach.

Our smallest and largest systems appeared to suffer more than the medium sized systems when using the MCA approach. We surmise that the smallest, $B = 16$ tile, components suffered because their smaller area had less interconnect flexibility than the larger components. Therefore we consider four tiles as a lower bound for a component's dimension in the target architecture used. We believe this to be specific to the interconnect architecture used and a study into the smallest components supported should be carried out when applying these techniques to a new architecture. Furthermore, we found that mapping to non-square component regions produced a lower quality result than square regions. However, we expect that region shape will depend more on the nature of the circuit being mapped into the component.

The largest, $B = 256$ tile, components suffered more because the port area constraints spread the interface signals across a wider area, increasing the amount that net's were stretched from their optimum route. In a later study we found that a centrally located and compressed port area is better than one spread across an interface surface.

In order to increase the potential flexibility of connectivity when composing a system from pre-routed components we have proposed reserving interconnect bandwidth to facilitate links that tunnel through components to connect non-neighbouring regions. Using different wire use policies we were able to reserve interconnect bandwidth for tunnelling links. As expected reserving interconnect bandwidth reduces the amount of interconnect complexity that can be supported by a given architecture. We were able to map all of the synthetic systems with up to 50% of the interconnect bandwidth reserved. Clearly the larger a component the more it suffers from bandwidth reservation. Using complementary bandwidth reservations and interface allocations we have shown the feasibility of our tunnelling link strategy. One weakness of this strategy is the amount by which a tunnelling link increases net length. If tunnelling links were to be used, it is strongly recommended that signals are registered at both ends of a link to maintain performance.

The tunnelling link technique presented here has strong similarities with the domain-based isolation technique presented by Tessier [Tess99]. One major difference is that we do not propose routing at merge time and the complementary edge interfaces help to balance the amount of wire bandwidth required for inter-component communication. Both the *planar isolation* and *domain-based isolation* techniques, presented by Tessier [Tess99], require routing to be performed at the point where pre-routed components are merged into a system. Tessier noted that there was some reduction in compile time when using these techniques. However, comparative figures are difficult to extract. The planar isolation technique suffered resource loss due to routing only areas. While the MCA approach sacrifices flexibility in the position of one component relative to another, MCA has no routing at merge time and uses routing resource within a component so there is less FPGA area lost to routing.

Other approaches, designed for the Xilinx Virtex and Virtex-2, have been based around the tri-state buffer (TBUF) bus macro [Xilap290]. The TBUF bus macro imposes a restricted bandwidth of 4 signals per wire channel, using only 4% of W_{FPGA} in the Virtex architecture and only 2% of W_{FPGA} in the Virtex-II architecture. Furthermore, communication is limited to the horizontal axis. Sedcole et al. [Sedc04], describe LUT bus macros used in the Xilinx Virtex-4 architecture. LUT bus macros have been built into the partial reconfiguration design flow created by Xilinx for Virtex-2 and Virtex-4 architectures [Xilug208]. Each LUT bus provides 8 unidirectional signals per tile along an interface surface or 4% of W_{FPGA} . It is possible to overlay three LUT bus macros to provide an aggregate bandwidth of 24 signals per tile along an interface or 12% of W_{FPGA} . Each signal in the macro uses an input side LUT, a specific routing wire, and an output side LUT which will add delay to signals passing through the macro. While an optional register stage can be added to the output logic element, application logic cannot be mapped to the elements used by the LUT bus macro. Thus, there will always be an area overhead associated with each set of signals between the component regions. If we applied LUT bus macros to our $B = 256$ $W_{IF} = 40\%$ of W_{FPGA} benchmark (from subsection 4.6.1) that uses 256 interface signals, we would need to expand each component region by 64 tiles (four signals per tile). This would represent an area increase of 25%. Sedcole et al. also report a form of wire reservation using all long lines and 20% of hex lines. This would provide a non-neighbour edge bandwidth of 48 wires per interface edge tile in a Virtex-4 providing 29% of the W_{FPGA} . However, the reserved wires are only used in a system specific harness where components are swapped in and out.

So far we have used systems comprising just two components. Our findings show that the MCA approach is able to reduce compile time in a system of two components by up to 20%. An aspect of compile time reduction that we have not illustrated here is the potential for component reuse both within a system and across different systems. In the next chapter we will examine more complex systems and explore the potential for intra-system reuse in systems of many components using a representative high performance computing application.

5 Application

In the last chapter we used our experimental mapping tools to explore the impact of pre-routed components and identified a number of important points to observe when designing a system using these components.

In this chapter we look at a specific HPC application suitable for FPGA. HPC on FPGA is achieved by exploiting the technology's massive parallelism. We use Foster's parallel system design methodology [Fost95] and apply it to the design of parallel computing systems on FPGA. In particular, we show how it can help plan mapping decisions when using our *pre-routed components* methodology.

5.1 Parallel System Design Methodology

Foster's design methodology [Fost95] is based on a *task* and *channel* model of computation. The task abstraction provides a mechanism for talking about locality: local data is contained within a task's local memory and other data is stored remotely outside the task. The channel abstraction provides a mechanism for indicating that a computation in one task requires data from another task before it can proceed. Channels indicate data dependencies between tasks.

Foster's methodology defines four steps in creating a parallel system design given a computing problem: First the problem is decomposed into primitive tasks; the communication channels between tasks are identified; the primitive tasks are agglomerated in an attempt to both balance communication bandwidth with processing bandwidth and amortise communication overheads across a set of tasks; and lastly each agglomerated task is mapped to available hardware.

Consider that, within our pre-routed component based FPGA design methodology, tasks map to component instances and channels map to links between components. Following Foster's methodology helps to:

- Identify tasks and estimate their associated component resource requirements;
- Identify identical tasks, which could lead to component reuse;
- Identify link bandwidth requirements;
- Provide agglomeration options to amortise encapsulation overhead; and
- Provide agglomeration options to share or reduce link bandwidth requirements.

Furthermore, identifying the communication patterns and component resource requirements provide inputs to floor-planning the system.

Identifying identical functional tasks that will operate in parallel is key to leverage on the potential to reuse pre-routed components offered by our MCA approach. Multiple instances of the same task can be performed by multiple instances of the same component.

When agglomerating tasks, both the resource requirements and the channel requirements of the component, that the task will map to, needs to be considered.

In the previous chapter we found that a square component region was most optimal. For our synthetic target architecture, components below a region size of 36 tiles appeared to suffer from increased congestion. Therefore we place a minimum component region size of 36 tiles. On the other hand, the smaller the components the greater the partitioning of the compilation problem, and therefore the greater potential there is for a reduction in compiler effort.

Component region dimensions effect the link bandwidth available between components. For our synthetic target architecture we found that using 20% of the total interconnect bandwidth available along an interface surface was acceptable, however it could be pushed up to 40%.

Our pre-routed component methodology places restrictions on the pattern of communication that is directly mapped to links between components. Links between components are strictly point-to-point. Generally components have four sides, and thus, four interface surfaces. Therefore, anything other than a grid does not easily scale. One option is to extend the surface of a component to abut with a number of communicating tasks to support a one-to-many pattern.

Tunnelling links, while still point-to-point provide slightly more flexibility by linking non neighbouring components. However, the amount of tunnelling bandwidth supported by a component will effect its computational performance. For our synthetic target architecture we found that using more than 10% of the wire channel bandwidth for tunnelling had an adverse effect on performance.

Rather than directly mapping the communication infrastructure, point-to-point links may be coupled within components to create more complex communication patterns. For example a many-to-many communication may be implemented with each component supporting a router. Alternatively, a central component may have the sole purpose of distributing data among a set of components.

The application must be adapted to fit within the constraints of the pre-routed component methodology. If it is found that the available wire bandwidth is restricting the number of signals in a channel between two tasks one may consider multiplexing data onto a link. If the interface bandwidth is restricting the speed of computation within a component, agglomerating the two communicating tasks to absorb the link inside a component may be an option. Another option is to expand a component region, reducing its internal resource utilisation, purely to extend an interface surface and increase the available bandwidth. Shaping components to meet the resource requirements and the communication pattern leads to possible floor-plans.

Using our pre-routed component methodology, we will now develop a high performance computing application on FPGA that accelerates a biological database search algorithm by up to two orders of magnitude over a general purpose desktop processor. This application demonstrates the level of speed-up that is achievable when implementing integer based algorithms in FPGA and the potential for increased flexibility and performance when utilising the fine grain re-programmability of FPGA technology. While this application exhibits a somewhat simple communication structure (a linear array) and uses only integer arithmetic, the performance is sensitive to communication efficiency and area utilisation. The simplicity does not detract from the underlying issues

that we wish to focus on, namely scalability, communication efficiency, and resource efficiency. Thus, it is an excellent case for studying the impact of our pre-routed component methodology and illustrating the interaction between compile time and application run-time performance in a highly reconfigurable system.

5.2 Biological Sequence Database Scanning

Scanning protein sequence databases is a common and often repeated task in molecular biology. The scan operation consists of finding similarities between a particular query sequence and all sequences of a bank and allows biologists to point out sequences sharing common sub-sequences. From a biological point of view, it leads to identifying similar functionality.

5.2.1 Motivation for FPGA Acceleration

The need for speeding up this application comes from the exponential growth of the bio-sequence banks: every year their size scales by a factor of 1.5 to 2. Comparison algorithms whose complexities are quadratic with respect to the length of the sequences detect similarities between the query sequence and a subject sequence. One frequently used approach to speed up this time consuming operation is to introduce heuristics in the search algorithm [Alts90]. The main drawback of this solution is that the more time efficient the heuristics, the worse is the quality of the result [Pear95].

Another approach to get high quality results in a short time is to use parallel processing. There are two basic methods of mapping the scanning of sequence databases to a parallel processor: one is based on the systolisation of the sequence comparison algorithm; the other is based on the distribution of the computation of pair-wise comparisons. Systolic array architectures have been proven as a good candidate structure for the first approach [Chow91], [Guer97], [Sing96], while more expensive supercomputers and networks of workstations are suitable architectures for the second [Glem97], [Lave98].

Special-purpose systolic arrays provide the best area/performance ratio by means of running a particular algorithm [Hugh96]. Their disadvantage is the lack of flexibility with respect to the implementation of different algorithms. Several massively parallel single instruction multiple data (SIMD) architectures have been developed in order to combine the speed and simplicity of systolic arrays with flexible programmability [Bora94], [Dahl99], [Alts90]. However, because of the high production costs involved, there are many cases where announced second-generation architectures have not been produced. Instead we pursue the possibility of realizing high performance sequence database scanning on an FPGA platform, which provides flexibility for fine-grained parallel computing based on re-configurable hardware. Since there is a large overall FPGA market, this approach has a relatively small price-per-unit cost and also facilitates upgrading to state-of-the-art FPGA technology as soon as it is available.

Taking full advantage of hardware reconfiguration, we present modular designs that are tailored towards particular query parameters. We will show how this leads to a high-speed implementation on a Virtex-II FPGA. We will then explore the use of our pre-routed component methodology when mapping to an FPGA architecture model.

5.2.2 Previous Approaches

A number of parallel architectures have been developed specifically for sequence analysis. In addition to architectures specifically designed for sequence analysis, existing programmable sequential and parallel architectures have been used for solving the sequence alignment problems. Special-purpose hardware implementations can provide the fastest means of running a particular algorithm with very high PE density. However, they are limited to a single algorithm, and thus cannot supply the flexibility necessary to run the variety of algorithms required for analysing DNA, RNA, and proteins. P-NAC was the first such machine and computed edit distance over a four-character alphabet [Lopr87]. More recent examples, better tuned to the needs of computational biology, include BISP, SAMBA and BIOSCAN [Chow91], [Guer97], [Sing96].

An approach presented in [Schm02] is based on instruction systolic arrays that combine the speed and simplicity of systolic arrays with flexible programmability. Several other approaches are based on the SIMD concept, including MGAP [Bora94], Kestrel [Dahl99] and Fuzion [Schm02]. SIMD and instruction systolic array architectures are programmable and can be used for a wider range of applications, such as image processing and scientific computing. Since these architectures contain more general-purpose parallel processors, their PE density is less than the density of special-purpose ASICs. Nevertheless, SIMD solutions can still achieve significant runtime savings. However, the costs involved in designing and producing SIMD architectures are quite high. As a consequence, none of the above solutions has a successor generation, making upgrading impossible.

Recently, a number of bio-sequence analysis applications have been developed for re-configurable systems [Hoan93], [Time08], [Gokh95], [Yama02], [Oliv05]. Re-configurable systems are based on programmable logic such as FPGAs or custom-designed arrays. They are generally slower and have lower PE densities than special-purpose architectures, but are more flexible. The main drawback is that the configuration must be changed for each algorithm, which is generally more complicated than writing new code for a programmable architecture. Several solutions including Splash-2 [Hoan93] and Decypher [Time08] are based on FPGAs while PIM has its own re-configurable design [Gokh95]. Solutions based on FPGAs have the additional advantage that they can be regularly upgraded to state-of-the-art-technology. This makes FPGAs a very attractive alternative to special-purpose and SIMD architectures.

Compared to the previously published FPGA solutions, we are using a new partitioning technique for varying query sequence lengths. The design presented in [Yama02] is closest to our approach since it also uses a re-configurable platform. Unfortunately, it only allows for linear gap penalties and global alignment, while our implementation considers both linear and affine gap penalties and is able to compute local alignments.

5.2.3 Sequence Comparison Algorithm

Surprising relationships have been discovered between protein sequences that have little overall similarity but in which similar sub-sequences can be found. In that sense, the identification of similar sub-sequences is probably the most useful and practical method for comparing two sequences. The Smith-Waterman algorithm [Smit81] finds the most similar sub-sequences of two sequences (the local alignment) using dynamic programming.

The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment (or subsequence) into another. Two elementary operations are used: substitution and insertion/deletion (also called a gap operation). Through a series of such elementary operations, any segment can be transformed into any other segment. The smallest number of operations required to change one segment into another can be taken as the measure of the distance between the segments.

Consider two strings S_1 and S_2 of length l_1 and l_2 . To identify common subsequences, the Smith-Waterman algorithm computes the similarity $H(i,j)$ of the two sequences ending at position i and j of the sequences S_1 and S_2 , respectively. The computation of $H(i,j)$ is given by the following recurrences:

$$\begin{aligned} H(i,j) &= \max\{0, E(i,j), F(i,j), H(i-1,j-1) + Sbt(S_{1i}, S_{2j})\}, & \text{for } 1 \leq i \leq l_1, 1 \leq j \leq l_2. \\ E(i,j) &= \max\{H(i,j-1) - \alpha, E(i,j-1) - \beta\}, & 0 \leq i \leq l_1, 1 \leq j \leq l_2. \\ F(i,j) &= \max\{H(i-1,j) - \alpha, F(i-1,j) - \beta\}, & 1 \leq i \leq l_1, 0 \leq j \leq l_2. \end{aligned} \quad (5.1)$$

where Sbt is a character substitution cost table. Initialization of these values is given by $H(i,0) = E(i,0) = H(0,j) = F(0,j) = 0$ for $0 \leq i \leq l_1, 0 \leq j \leq l_2$. Multiple gap costs are taken into account as follows: α is the cost of the first gap; β is the cost of the following gaps. This type of gap cost is known as *affine gap penalty*. Some applications also use a *linear gap penalty*, i.e. $\alpha = \beta$. For linear gap penalties the above recurrence relations can be simplified to:

$$\begin{aligned} H(i,j) &= \max\{0, H(i,j-1) - \alpha, H(i-1,j) - \alpha, H(i-1,j-1) + Sbt(S_{1i}, S_{2j})\}, \\ & \text{for } 1 \leq i \leq l_1, 1 \leq j \leq l_2. \\ H(i,0) &= H(0,j) = 0 \text{ for } 0 \leq i \leq l_1, 0 \leq j \leq l_2. \end{aligned} \quad (5.2)$$

Each position of the matrix H is a similarity value. The two segments of S_1 and S_2 producing this value can be determined by a trace-back procedure. Table 5.1 shows an example of the Smith-Waterman algorithm used to compute the local alignment between two DNA sequences ATCTCGTATGAT and GTCTATCAC. The matrix $H(i,j)$ is shown for the linear gap cost $\alpha = 1$, and a substitution cost of +2 if the characters are identical and -1 otherwise.

Table 5.1 Example of the Smith-Waterman algorithm

	∅	A	T	C	T	C	G	T	A	T	G	A	T
∅	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	1	0	0	2	1	0
T	0	0	2	1	2	1	1	4	3	2	1	1	3
C	0	0	1	4	3	4	3	3	3	2	1	0	2
T	0	0	2	3	6	5	4	5	4	5	4	3	2
A	0	2	2	2	5	5	4	4	7	6	5	6	5
T	0	1	4	3	4	4	4	6	5	9	8	7	8
C	0	0	3	6	5	6	5	5	5	8	8	7	7
A	0	2	2	5	5	5	5	4	7	7	7	10	9
C	0	1	1	4	4	7	6	5	6	6	6	9	9

In order to extract the local sequence alignment from the matrix a trace-back procedure is performed. A trace-back starts from the highest score in the matrix. The characters from S1 and S2 corresponding to this position in the matrix are the last in the alignment. The trace is formed by iteratively selecting the highest of the three cells up, left and diagonally up and left. When a zero is reached the trace-back terminated. From the highest score (+10 in the example), the trace-back delivers the corresponding alignment (the shaded cells) as the two sub-sequences:

```
TCT---ATCA
TCTCGTATGA
```

Only the maximum alignment value in the matrix is required to determine the significance of a pairwise alignment between the two sequences. Trace-back will only be performed on a handful of the most significant alignments. Thus, we have added the maximum matrix, M , and associated recurrence relation to find the maximum without having to search the H matrix. The maximum alignment value seen at position i and j is calculated. The computation of the maximum value matrix M is given by the following recurrence relation:

$$\begin{aligned} M(i,j) &= \max \{0, M(i,j-1), M(i-1,j), H(i,j)\}, \text{ for } 1 \leq i \leq l_1, 1 \leq j \leq l_2. \\ M(i,0) &= M(0,j) = 0 \text{ for } 0 \leq i \leq l_1, 0 \leq j \leq l_2. \end{aligned} \quad (5.3)$$

The value at $M(l_1, l_2)$ will be the maximum score for the pairwise alignment of sequences S1 and S2.

5.2.4 Application Scenario

We have created a database scanning application that is provided as a web service, allowing the user to enter a database query using a form-based web page similar to MPSrch [Stur93]. The service architecture of such a system is illustrated in Figure 5.1.

Figure 5.1 Web service architecture

The web front end interface of this service is shown in Figure 5.2. A MySQL server is used to store the Swiss-Prot database and queue user queries. A query has a unique sequence and may have a unique matrix, gap penalties, and database selection. The parameter set from a user query is given a unique ID and stored in a MySQL table. When the search is finished the results are returned to the MySQL database where the web server identifies them using the unique ID and displays them to the user.

Figure 5.2 Web data entry screen

5.3 Parallel Algorithm Design

We will adapt Foster's methodology, which usually focuses on mapping computing problems to a number of sequential ISA processors running in parallel, to map to small specialised Processing Elements (PE). Taking advantage of the reconfigurable hardware platform, we can tailor the individual PE design towards different gap penalty functions (linear or affine). This approach allows us to include only as much computational hardware and local memory as required. There are a number of ways to exploit the parallelism in the sequence alignment data base scanning problem.

5.3.1 Identifying the Parallelism

Following the four steps in Foster's methodology to architect our application:

First, we perform a functional decomposition on the web service architecture. We use a desktop processor and an FPGA on a plug-in card. We first perform functional decomposition to identify the tasks split between FPGA and processor.

- Handling the web front end and database streaming
- Calculating maximum alignment score for query and each database sequence
- Sorting and ranking the top *100* or so scoring alignments
- Performing trace-back and displaying the alignment

The bulk of the computational effort is consumed by the alignment calculation, which uses a small repeating set of integer operations making it suitable for FPGA implementation. Therefore, it is clear that the alignment calculation will map to the FPGA with the database streaming task, score sorting task and trace-back running on the processor.

One can consider the parallelism within the alignment task at several levels. Each search request is a unique task. This contains a task for each alignment of the query sequence

with each database sequence. Consider alignment of two sequences $A = a_1a_2\dots a_N$ and $B = b_1b_2\dots b_K$, using the Smith-Waterman dynamic programming algorithm. The cell values $H(i, j)$, $E(i, j)$, and $F(i, j)$ are produced for all cells in matrices with dimensions $N \times K$. By associating a primitive task, as shown in Figure 5.3 with each cell update we have our first domain decomposition. On a large enough FPGA it may be possible to create a PE for every cell in the matrix. If a task updating one cell needs a value from a task updating another cell, we connect them via a channel. However, it can be seen from the recurrence relations (5.1), (5.2) and (5.3), that each cell update requires values from the cell above, from the left and from the cell diagonally up and left. Each task has three incoming channels and three outgoing channels. Therefore, PEs would only be actively calculating along a diagonal wave front, only reaching maximum efficiency when the wave reaches the longest diagonal.

Figure 5.3 Single Cell Update Task Assigned to Processing Elements

Instead of one PE per cell, we agglomerate the primitive tasks associated with the same column of the matrix absorbing communication channels along the vertical axis, as shown in Figure 5.4. Communication is folded into a pipeline along a linear array of PEs [Kung88]. Each PE passes its last calculated values to the next PE in the array. Every PE in the pipeline is employed doing useful computations. Therefore, the mapping of columns to each PE is far more efficient than mapping a PE to each cell in the matrix.

Figure 5.4 Whole matrix column assigned to processing elements

So far we have only considered communication of the cell values. The alignment calculation requires the query sequence, database sequence, gap penalties and substitution table to be communicated to each PE. Rather than a one-to-many communication structure, global communication is distributed so that values are passed through the linear array neighbour-to-neighbour in the same way that intermediate values are.

Because each PE needs access to the substitution table, we distribute it to every PE. Rather than every PE having a complete copy of the substitution table, each PE only requires the column of the substitution table relating to its assigned query sequence character reducing the storage requirements of a PE.

Assuming we are aligning query sequence A with subject sequence B from the database, where $A = a_1a_2\dots a_N$ and $B = b_1b_2\dots b_K$. We use a linear PE array of size N with affine gap penalties (the procedure for linear gap penalties is similar). As a preprocessing step, symbol a_i is assigned to PE i , with $1 \leq i \leq N$. After that the row of the substitution table corresponding to the respective character is loaded into each PE as well as the gap penalties α and β . PE 1 receives b_j in step j with $1 \leq j \leq K$. B is then completely shifted through the array in $N + K - 1$ steps as shown in Figure 5.5. In iteration step k , $1 \leq k \leq N + K - 1$, the values $H(i, j)$, $E(i, j)$, and $F(i, j)$ for all i, j with $1 \leq i \leq N$, $1 \leq j \leq K$ and $k = i + j - 1$ are computed in parallel in all PEs $1 \leq i \leq N$, within a single clock cycle. For this calculation PE i , $2 \leq i \leq N$, receives the values $H(i-1, j)$, $E(i-1, j)$, and b_j from its left neighbour $i - 1$, while the values $H(i-1, j-1)$, $H(i, j-1)$, $F(i, j-1)$, a_i , α , β , and a column of the substitution table for a_i are stored locally. Additions are performed using saturation arithmetic. The look-up of $Sbt(a_i, b_j)$ and its addition to $H(i-1, j-1)$ is done in one cycle. The values $H(i, j)$, $E(i, j)$, and $F(i, j)$ are calculated in the next cycle and passed to the next PE in the array. In the following cycle the value of $M(i, j)$ is calculated. After the last character of a subject sequence has been processed in PE N , the maximum of matrix H is stored in PE N , which is then written into the off-chip memory. In this way only the first and last PE in the linear array need to be connected to the FIFO buffer and external host interface.

Figure 5.5 Linear array of pairwise sequence alignment processing elements

Thus, it takes $N + K - 1$ steps to compute the alignment score of the two sequences with the Smith-Waterman algorithm. Now, assuming $N \leq K$, in iteration step k , where $1 \leq k \leq N - 1$, only k PEs are actively processing. In iteration step k , where $N \leq k \leq K$, all N PEs are active. In iteration step k , where $N \leq k \leq N + K - 1$, $K + N - k$ PEs are active. If $K \leq N$ then only K PEs will ever be active, leaving $K - N$ PEs inactive for the entire alignment. However, notice that after the last character of B enters the array, the first character of a new subject sequence can be input for the next iteration step. Thus, all subject sequences of the database can be pipelined with only a one-step delay between two different sequences. By making the effective value of K large, the computational efficiency is maintained so that all N PEs are employed in useful computation.

Figure 5.6 shows our PE designs for both a linear gap penalty and for an affine gap penalty. The data width (dw) is scaled to the required precision. The LUT depth is scaled to hold the required number of substitution table rows. The substitution width (sw) is scaled to accommodate the dynamic range required by the substitution table. The look-up address width (lw) is scaled in relation to the LUT depth.

Figure 5.6 (a) Shows a linear gap penalty PE_i (b) Shows an affine gap penalty PE_i.

Because of the very limited memory of each PE, only the highest score of matrix H is computed on the FPGA for each pair-wise comparison. The host PC carries out the ranking of the compared sequences and the reconstruction of the alignments. Because this last operation is only performed for a very few subject sequences, its computation time is negligible.

The application management software in our web server handles communication between the MySQL server and the FPGA board. When the application manager sees the query in the list, it processes it for transfer to the FPGA and configures the FPGA with a linear array suited to the request. The sequence data is extracted from the database and compiled into a form suitable for fast streaming to the FPGA. The query parameters are loaded onto the linear array and the compiled database is streamed across the PCI interface. Only the maximum value found in each sequence alignment is required to indicate the significance of a pairwise match between the query and a particular database sequence. Results are collected and ranked in the application manager.

5.3.2 Query Length Scaling

So far we have assumed a PE array equal in size to that of the query sequence length. In practice, this rarely happens. Since the length of the sequences may vary, the computation must be partitioned on the fixed size PE array. The query sequence, which may have a length of several thousands in some cases and commonly has a length in the hundreds, is usually larger than the PE array. For sake of clarity we firstly assume a query sequence of length N and a PE array of size P where N is a multiple of P , i.e. $N = k \cdot P$ where $k \geq 1$ is an integer. A possible solution is to split the computation into k passes: The first P characters of the query sequence are assigned to the PE array and the corresponding substitution table columns loaded. The entire database then passes through the array; the H -value and E -value computed in PE P in each iteration step are output. In the next pass the following P characters of the query sequence are loaded into the array. The data stored previously is loaded together with the corresponding subject sequences and sent again through the PE array. The process is iterated until the end of the query sequence is reached.

Unfortunately, this solution requires a large amount of memory (assuming 16-bit accuracy for intermediate results then four times the database size per pass is needed). The memory requirement can be reduced by a factor p by splitting the database into p equal-sized pieces and computing the alignment scores of all subject sequences within each piece. However, this approach also increases the loading time of the substitution table columns

by a factor of p . In order to eliminate this loading time we have slightly extended our PE design. Each PE now stores k columns of the substitution table instead of only one. Although this increases the area per PE, it allows for the alignment of each database sequence with the complete query sequence without additional delays. It also reduces the required memory for storing intermediate results to four times the longest database sequence size (again assuming 16-bit accuracy). Figure 5.7 illustrates our solution. The database sequences are passed in from the host one by one through a FIFO to the S2 interface. The database sequences have been pre-converted to LUT addresses. For query lengths longer than the PE array the intermediate results are stored in a FIFO of width $2 \times dw + lw + 1$ for affine gap penalty. For linear gap penalty the FIFO width is $dw + lw + 1$. The FIFO depth is sized to hold the longest sequence in the database. The database sequence is also stored in the FIFO. On each consecutive pass an LUT offset is added to address the next column of the substitution table stored within the PEs. The maximum score on each pass is compared with those from all other passes and the absolute maximum is returned to the host. We can again take advantage of reconfiguration and design different configurations for different values of k . This allows us to load a particular configuration that is suited for a range of query sequence lengths.

Figure 5.7 System implementation

So far we have assumed that the query sequence length N is a multiple of the PE array size P , i.e. $N = k \cdot P$ where k is an integer. If this is not the case, we can still use our design by filling substitution table columns in the remaining PEs with zeros. However, by switching off the PEs in this way we are wasting computing bandwidth. This presents a problem of how to maintain efficiency when there is not enough parallelism due to a short query length.

We could exploit the parallelism in the database scan. The PE array is divided into query length segments and different parts of the database are streamed to each segment. One issue with this approach is the increased bandwidth requirement of the link from database storage to FPGA. This approach is only feasible when the number of available PEs is a multiple of the query length.

Another option is to process multiple short query sequences in parallel. It is possible to adapt the PE design to be able to handle the assignment of several short query sequences. Note that short sequences do not require the FIFO of intermediate results. A start of query flag is set in the PE to indicate that it has been assigned the first character in a query sequence. When the start of query flag is set, PE i will use initial values instead of values from PE $i-1$.

The difficulty comes when trying to extract the final result from an intermediate PE that has been assigned the last character in a query sequence. A channel is required from every PE that could be assigned the last character in a sequence back to the host interface. This suggests a many-to-one structure that could prove to be a bottleneck, preventing the architecture from scaling.

Another restriction of this approach is that we cannot exploit the individual query parameters to specialise the PEs. Instead we must select a PE array that will work for all the queries allocated. With more flexibility in composing multiple customised linear arrays we could customise each segment to the query parameters.

5.3.3 Mapping to Xilinx Virtex-II Technology

We have described the PE design in Verilog and targeted it to the Xilinx Virtex-II FPGA architecture.

We found that by providing region placement constraints for each PE improves both run-time performance and compile time. However, the PE region must be shaped to maximize the utilization internal to the PE and minimize the fragmentation of the FPGA resource. We allocate a 192×160 array of logic slices for the PE array on a Virtex II XC2V6000 FPGA. We specify an area constraint for each PE region. The linear PE array is placed in a zigzag pattern as shown in Figure 5.8.

We are able to accommodate 252 linear PEs or 168 affine PEs using $k=3$. This allows handling of query sequence lengths up to 756 and 504 respectively, which is sufficient in most cases (74% of sequences in Swiss-Prot are ≤ 500 [Boec03]). For longer queries we have implemented a design with $k = 12$, which can accommodate 168 linear PEs or 119 affine PEs. The corresponding clock frequencies are 55 MHz for a linear gap cost and 44 MHz for an affine gap cost.

Figure 5.8 System floor plan in the XC2V6000 on the Alpha-Data ADM-XRC-II board

We use on-chip RAM for the partial result FIFO. The FIFO depth has been sized to 8192 entries and uses almost all of one column of block SelectRAM. Database sequences longer than 8192 are aligned on the host (only 2 sequences in Swiss-Prot are > 8192 [Boec03]). The host interface takes up some of the FPGA space in the bottom right-hand corner.

A performance measure commonly used in computational biology is *cell updates per second* (CUPS). A CUPS represents the time for a complete computation of one entry of the matrix H , including all comparisons, additions and maxima computations. The CUPS performance of our implementations can be measured by multiplying the number of PEs by the clock frequency. Table 5.2 summarizes our results.

Table 5.2. Linear array performance for different PE designs mapped to a Virtex-II XC2V6000

Design	Max PEs Fitted	Max Speed (MHz)	Max. Query length (PEs $\times k$)	Peak Performance (GCUPS)
Linear, $k=3$	252	55	756	13.9
Linear, $k=12$	168	55	2016	9.2
Affine, $k=3$	168	45	504	7.6
Affine, $k=12$	119	44	1428	5.2

Since CUPS does not consider data transfer time, query length and initialization time, it is often a weak measure that does not reflect the behaviour of the complete system. Therefore, we will use database scans for different query lengths in our evaluation.

Table 5.3 reports the performance for scanning the Swiss-Prot protein data bank (release 43, which contains 146720 sequences comprising 54093154 amino acids [Boec03]) for query sequences of various lengths using our design on an ADM-XRC-II FPGA Mezzanine PCI-board with a Virtex-II XC2V6000 from Alpha-Data [Alph02]. The query sequence lengths have been chosen to illustrate the effect that length has on performance. Maximum performance is achieved when the query length is closely matched to an integer multiple of the PE array length. Note how the performance increases as the database-streaming overhead is amortized when the PE array is required to perform more processing passes. These scan times omit just 2 sequences in Swiss-Prot Release 43 because they are longer than 8192 amino acids.

For the same application an optimized C-program on a Pentium IV 1.6 GHz has a performance of 52 MCUPS for linear gap penalties and 40 MCUPS for affine gap penalties. Hence, our FPGA implementation achieves a speed-up of approximately 170 times for linear gap penalties and 125 times for affine gap penalties.

Table 5.3 The mean performance of an affine gap penalty PE array of length 119 ($k=12$) when scanning Swiss-Prot Release 43 for several query length ranges.

Query length range	Number of Processing Passes	Mean Performance (GCUPS)	Percentage of Peak Performance	Mean Scan Time (s)
3 – 119	1	2.6	50%	3.8
120 – 238	2	2.8	54%	5.2
715 – 833	7	4.8	92%	11.4
1310 – 1428	12	5	96%	17.9

For the comparison of different massively parallel machines, we have taken data from [Dahl99], [Guer97], [Schm02], [Yama02] for a database search with the SW algorithm for different query lengths. The Virtex II XC2V6000 is around ten times faster than the much larger 16K-PE MasPar. Kestrel, Fuzion and Systola 1024 are one-board SIMD

solutions. Kestrel is 12 times slower [Dahl99], Fuzion is two to three times slower [Schm02], and Systola is around 50 times slower [Schm02], than our solution. All these boards reach a lower performance, because they have been built with older CMOS technology (Kestrel: 0.5- μm , Fuzion: 0.25- μm , Systola 1024: 1.0- μm) than the Virtex-II XC2V6000 (0.15- μm). However, the major difference between these approaches and our reconfigurable FPGA solution, is not so much in the improved performance, but in the guaranteed technology upgrade path. The high production costs associated with the development of dedicated massively parallel SIMD processors has resulted in many cases where announced second-generation architectures have not been produced. This is not the case for FPGA devices, for instance, targeting our design to a Virtex-II XC2V8000 would improve the performance by around 30%. Further performance improvements could be achieved by targeting the design to newer devices, such as the Virtex-4 and Virtex-5 FPGAs

Our implementation is around three times faster than the FPGA implementation presented in [Yama02] which implements only global alignment on a Virtex XCV2000E. Our design is slightly slower than the FPGA implementations described in [Gucc02], [Hoan93], [Yu03]. However, these designs only implement edit distance. This greatly simplifies the PE design and therefore achieves a higher PE density as well as a higher clock frequency. Although of theoretical interest, edit distance is not used in practice because it does not allow for different gap penalties and substitution tables.

5.3.4 Precision Scaling

The precision of the PE array is dictated by the bit-width of the data path. The amount of resource used by the data path scales linearly with its bit width. The operations in the data path are all additions. The propagation delay of an adder scales linearly with its bit-width. Thus, for a lower bit-width, we should be able to fit more PEs onto the FPGA area and run them faster. Table 5.4 shows the performance of $k = 12$ affine gap penalty designs with a range of precisions, synthesized and mapped to the FPGA. Every PE is fitted into the same region shape. By adjusting the aspect ratio of a PE region we are able to fit more PEs than if we simply used a square PE region.

Table 5.4 The effect of precision scaling

Precision (Bits)	Speed (MHz)	PE slices	PEs (square)	Fitted PE height	Fitted PE width	PEs (fitted)	Peak Rate (MCUPS)
9	57	165	168	12	14	176	10032
10	56	170	143	10	17	171	9576
11	54	179	143	9	20	168	9072
12	53	191	143	12	16	160	8480
13	53	204	120	16	13	144	7632
14	52	211	120	24	9	136	7072
15	51	225	120	24	10	128	6528
16	50	230	120	17	14	121	6050

However, given the query parameters, we need to determine an adequate precision for the PE data path. Note that the PE design uses zero limited saturation arithmetic, and thus the minimum value is always 0 while the maximum value is $(2^W - 1)/2$, where W is the bit-

width. The negative values summed in the PE naturally saturate to zero. The score will only increment when two sequences match. The largest increment in any one processing step is the highest positive value in the substitution table (Sbt_{MAX}). For example the Blosom62 matrix has a Sbt_{MAX} of 11 for matching two Tryptophan amino acids. The absolute worst case would be comparing a sequence of all Tryptophan with an equal or longer length sequence of all Tryptophan. This is clearly an unlikely occurrence, but gives us a fast way to estimate the required precision. The maximum score that could occur when aligning two sequences of length N and K is the product of the shortest sequence length, $\min(N, K)$, and the maximum value in the substitution table, Sbt_{MAX} . Therefore, the required PE precision, b_{PE} , is calculated as in equation (5.4).

$$b_{PE} = \log_2((2 \times \min(N, K) \times Sbt_{MAX}) + 1) \quad (5.4)$$

5.3.5 Dynamic Precision Scaling

In the previous examples, the FPGA computing performance is limited by the amount of logic resource available. If we can increase the utilization of this finite resource we can increase the performance. The queries' parameters can be used to customise the array and make better use of the available resource. It is only worth employing run-time re-configuration if the time saving is greater than the time to re-configure.

We will first assume that all the necessary configurations have been pre-compiled and the bit-streams are available on the server. Therefore we only consider the time it takes to select the correct bit-stream and configure the FPGA. It takes approximately 80ms to configure the XC2V6000 over the PCI bus. A Swiss-Prot database scan will take several seconds so it is feasible to re-configure the FPGA for each query.

Protein databases exhibit a Gaussian curve length distribution centred around 300 amino acids. The Swiss-Prot database is no exception with 50% of its sequences below 300 amino acids. Figure 5.9 shows the distribution of sequence lengths in Swiss-Prot Release 43 and the number of Amino Acids (sequence characters) that each length bin contributes to the computing problem. It can be seen in Figure 5.9 that the bulk of the data to be processed in the area centred around sequences with a length of 400 characters, with very little data contributed by sequences over 1500 characters in length.

Figure 5.9 Sequence data distribution of Swiss-Prot Release 43

It can be seen from Table 5.4 that a low precision PE array has a higher performance than one that has a higher precision. The query sequence is known, and by scanning the database from shortest to longest sequence we can track the PE precision required by using equation (5.4). What we find is that we are able to use a lower precision PE array at the beginning, gearing up to a higher precision as the sequence length increases. We estimate the average sequence length in a bin with upper bound (N_{UPPER}) and lower bound (N_{LOWER}) is $N_{AVERAGE} = (N_{UPPER} + N_{LOWER})/2$. Given a query sequence length (K), the number of sequences in a bin (S), and the processing rate ($CUPS$), we can estimate the time to process the sequences in a bin as $T_{BIN} = (N_{AVERAGE} \times K \times S) / CUPS$.

Figure 5.10 illustrates how dynamic scaling would give a higher processing rate at the start of the scan, slowing down towards the end. The time taken to process each length bin is plotted in Figure 5.10. It can be seen that most of the processing time is spent processing sequences with a length in the region of 500 characters. A full device re-configuration is required each time the PE array precision is changed which takes approximately 80ms. The reconfiguration time required each time the precision of the PE array is changed is included in the time shown in Figure 5.10. A total of 0.4 seconds would be spent reconfiguring the FPGA.

Figure 5.10 dynamic precision scaling scan of the Swiss-Prot database

Using this technique with a query sequence of length $K = 1452$, $Sbt_{MAX} = 11$, and a $k = 12$ PE array, in a scan of the Swiss-Prot Release 43 database we would achieve a 11% increase in the overall performance, resulting in a 4% reduction in the time taken to do the database scan when compared to doing the entire scan with one precision PE.

5.4 Mapping to Pre-Routed Components

We have explored how specialising the PE design to the query parameters can afford an increase in performance. If all of these specialised PE array configurations are to be made available to the web service they must be compiled and stored. This represents a significant overhead in terms of both designer effort and storage capacity.

As the sequence scanning application requires a system of repeated PEs, there is the potential to cut compile time by orders of magnitude by pre-routing a single PE for each specialisation and composing the linear array from that single pre-routed PE. However,

this assumes that one pre-routed PE will compose into a linear array. In actual fact, we will need several versions of the pre-routed PE to cope with different geometric constraints. To illustrate the PE geometry, Figure 5.11 shows a conceptual mapping of a linear array of 16 PEs. A linear array may be folded onto the FPGA in any number of ways. The PE boundaries are shown as well as one possible data flow through the PEs.

Figure 5.11 Conceptual quantised system floor for a 16 PE system

Considering the interface surface across which data flows in and out of each PE region, we can build up the set of *port-edge combinations* that are required in the example floor plan in Figure 5.11. This floor plan requires 8 unique port-edge combination of the PE (annotated 1 to 8 in Figure 5.11). Effectively, we would only have to compile half of the design because we compile 8 PEs and use a total of 16 in this system. In a larger system, such as the 12-bit precision array of 160 PEs (from Table 5.4), we would still only have to compile 8 unique pre-routed PEs. This represents a potential $160 / 8 = 20$ times reduction in compiler effort.

So far, we have not considered the restrictions on interface bandwidth and geometry. We will first illustrate some of the restrictions placed on pre-routed components by the available tools. Following this we will map the application to our experimental compiler environment and investigate the more subtle effects of wire level pre-routing.

5.4.1 Pre-Routing on Virtex-II

The first commercially available design environment for pre-routing components was designed to facilitate module based partial FPGA reconfiguration [Xilap290]. Module based partial reconfiguration has the added restriction that component regions must align with the independently reconfigurable regions of the configuration memory. Since we are not attempting partial reconfiguration we do not consider this restriction. Furthermore, we will assume that once a component has been pre-routed we are able to relocate the resource and interconnect configuration to any location on the device.

Bus macros created using tri-state buffer primitives provide the interfacing point between components [Xilap290]. Tri-state buffer lines are only available along horizontal interconnect channels. Therefore, interfaces can only be created in the horizontal direction. Furthermore, using tri-state buffers provides a maximum of four interface signals per CLB row on a Virtex-II device.

We will use the 12-bit PE from Table 5.4 to illustrate how these restrictions need to be considered. We will assume we have a contiguous array of 96 CLB tiles by 80 CLB tiles, which is representative of the XC2V6000 device. While BRAM columns add further constraints to component mapping we will not consider them in this discussion. The 12-bit PE requires 191 Virtex-II logic slices. Since a Virtex-II CLB tile contains four logic slices, we would be able to pack a PE into 48 CLB tiles (assuming we will achieve 100% logic utilisation). By shaping the PE region to be 6 CLBs high by 8 CLBs wide, allows us to fit $16 \times 10 = 160$ PE regions in the available space. Using the floor plan illustrated in Figure 5.11, we require a set of PEs with 8 port-edge combinations to build the linear array. Only the total area of this set of PEs ($8 \times 48 = 384$ CLB tiles) would have to be compiled for a design that actually covers $96 \times 80 = 7680$ CLB tiles, yielding a potential reduction of 20 times in compiler effort.

So far we have not considered the port geometry. Table 5.5 shows the detail of the interface used in our affine gap penalty PE, with signal widths for a 12-bit precision.

Table 5.5 Interface and number of signals for the 12-bit precision PE

Name	Signals	Description
sequence_character	5	Database sequence character code
e_cell	12	E matrix cell value
h_cell	12	H matrix cell value
m_cell	12	Maximum matrix cell value
reset	1	Reset pulse between database sequences
write_enable	1	Write enable to query sequence storage
overflow	1	Adder overflow indication
Total	44	

A total of 44 signals will require an interface surface of 11 CLB tiles when using the tri-state lines that provide 4 wires per tile. The result of our first fitting exercise is an interface surface of 8 CLB tiles. Rather than needlessly expanding a PE region or stretching the interface surface, we will agglomerate two PEs into one component. A component of two PEs requires 96 CLB tiles (382 slices). Shaping this to be 8 tiles wide affords an interface surface that is 12 tiles high, enough to fit our tri-state buffer mapped interface. We would be able to fit $8 \times 10 = 80$ dual PE components, providing a total of 160 PEs in an array.

Note that in Figure 5.11 we have both vertically and horizontally directed links. Tri-state buffer mapped interfaces do not support vertically directed links. Note that if the availability of tri-state buffers was not a hard restriction on vertically directed links, our component region only provides an interface surface that is 8 tiles wide for vertically directed links, not wide enough to support our interface at 4 signals per tile. In order to remove the vertical interfaces, we agglomerate four PEs into a component to absorb vertical links. Figure 5.12 shows a representative system floorplan that uses a set of 6 agglomerated PE components to handle the restrictions of the tri-state lines.

Figure 5.12 Linear array of 40 PEs that requires a set of 6 port-edge combinations

For this set of 6 components the total area compiled is $(2 \times 96) + (4 \times 192) = 960$ CLB tiles. While we have managed to fit the same number of PEs, the reduction in compiler effort has dropped from 20, to $7680 / 960 = 8$. Less than half the reduction we predicted in the best case.

Improvements to the Xilinx modular partial reconfiguration design flow have been reported [Sedc04]. Instead of using tri-state buffers, the bus-macros use a slice LUT at either end of a fixed wiring pattern to provide an interfacing point. A *slice based bus-macro* uses one CLB and provides a path between components for up to 8 signals. It is not possible to insert application logic into the CLB used by the bus-macro.

As an example, we take our 12-bit precision PE design that requires 48 CLB tiles and has two interfaces (in and out) each with 44 signals. Each interface requires 6 slice based bus-macros to transport 44 signals. Thus we must increase the single PE component region to $(6 \times 2) + 48 = 60$ CLB tiles. By shaping this to be 6 tiles high and 10 tiles long we are able to fit $16 \times 8 = 128$ PEs into the space available. Note that a PE component has an effective resource utilisation of $48 / 60 = 80\%$ of the CLB tiles.

While only horizontally directed slice bus-macros are available for the Virtex-II, there are vertically directed macros available for the Virtex-4 and Virtex-5. We will assume that vertically directed macros are available for the Virtex-II. The slice macro provides an interface bandwidth of 8 signals per CLB column or row. Therefore, an interface of 44 signals requires an interface surface of 6 tiles in width. There is no need to stretch the PE or agglomerate more than one PE into a component because both edges of the PE are able to accommodate an interface that is 6 tiles wide.

Using the floor plan pattern from Figure 5.11 we require 8 port-edge combinations to be able to create this system. Thus, the reduction in compiler effort is $7680 / (60 \times 8) = 16$. While this is close to the theoretically achievable reduction of 20, we have lost 20% of the PEs in the array to the overhead of slice macros. This would translate into a direct loss of 20% in peak performance.

5.4.2 New Pre-Routing Approach

We will not study the existing approaches to pre-routed components in any further detail. Instead we will map a simplified version of the PE design to our synthetic architecture using our new compilation environment to investigate the more subtle effects of pre-routing on design performance.

There is no value in adding all of the Virtex-II features to our architectural model. We have not supported elements, such as LUT based RAM, as they present a significant challenge to the mapping tools and have little relevance to pre-routing performance. Furthermore, we have used a commercial synthesis tool when mapping to the Xilinx Virtex-II. However, our design environment uses an open source synthesis tool with a very different level of optimisation performance. The interconnect parameters of our synthetic FPGA architecture are very different from the Virtex-II. Therefore, we do not intend to make any direct comparisons between the performance and resource utilisation of our approach and the mapping to Virtex-II technology. Instead, we will focus on the routing performance between the three approaches (MPA, MRA and MCA) within our compilation environment.

The restrictions of the architectural model and the synthesis tools require a simplified PE design. Rather than comparing protein sequences, which requires a larger lookup table to support 21 different amino acids, we will compare DNA sequences which only requires a small lookup table to support 4 bases. The PE design has been further simplified to support only one query character per PE ($k = 1$). A small lookup table makes it feasible to map the RAM structure to registers. We will use the smaller linear gap penalty PE design with a precision of 8-bits. Table 5.6 shows the details of the interface used in this PE design.

Table 5.6 Interface and number of signals for the 8-bit simplified PE

Name	Signals	Description
sequence_character	2	Database sequence character code
h_cell	8	H matrix cell value
m_cell	8	Maximum matrix cell value
reset	1	Reset pulse between database sequences
write_enable	1	Write enable to query sequence storage
overflow	1	Adder overflow indication
Total	21	

The simplified 8-bit precision PE requires 52 logic resource tiles in our synthetic architecture. Shaping the PE to be 7 tiles wide by 8 tiles high provides a component region of 54 logic tiles (96% resource utilisation). An interface surface of 7 tiles will need 3 wires per tile to fit the interface signals outlined in Table 5.6. The W2W3.1 wire allocation heuristic (see Table 4.3) was used to create the interface allocation. We compress our interface allocation to be 6 tiles wide and use 10% of the available wires in the interconnect channel. From our previous results, we expect that a W_{IF} of 10% will not adversely affect performance. The high fanout signals were moved to the centre of the port region.

Using our pre-routing technique there is enough interface bandwidth to meet the requirements of the interface. Furthermore, there is provision for links along both axis. Thus, there was no need to agglomerate PEs when using our pre-routing approach. By using bisected wires as the isolation point between components, there was no need for extra resource to implement the interface. Thus, we are able to achieve the predicted

reduction in compiler effort. In the next subsection we will compare the differences in performance of the three compilation approaches.

5.4.3 The Impact of Pre-Routing

We do not need to create large, 160 PE, systems to see the effect of pre-routing. Instead we create systems of 2, 4, 6, 8, 10, and 20 PEs. This is enough to see the trends in critical path length and compiler effort that may be extrapolated to system with a larger number of PEs. We have not considered the FIFO in this system architecture. The host interface is simply a set of interface signals connected via registers to a set of device pads. Figure 5.13 shows the schematic layout of a 20 PE system that requires 9 unique port-edge combinations.

Figure 5.13 Schematic layout and screen capture of the 20 PE system mapped to our synthetic architecture using the MCA approach

We have created 6 systems, the details of which are recorded in Table 5.7. The MPA approach does not take advantage of the fact that only a fraction of the total PEs have unique port-edge combinations. The MRA approach is only able to take advantage of the unique PEs to reduce resource allocation effort, where as the MCA approach is able to reduce the connection allocation effort too.

Table 5.7 Details of the six systems used in this study

System	Total PEs	Unique PEs	Reduction in Compiler Effort	W_{MIN} (MPA)	Maximum W_{MIN} (MRA and MCA)
1	2	2	1	12.8	18.1
2	4	4	1	15	18.1
3	6	4	1.5	16.3	18.1
4	8	4	2	17.5	18.1
5	10	4	2.5	18.2	18.1
6	20	9	2.22	17.6	18.3

The routing difficulty predictor values (W_{MIN}) are shown in Table 5.7. Note that the routing difficulty steadily increases with system size for the MPA approach because the placement problem increases at the same rate. The maximum routing difficulty across all the unique components is reported for each system. This stays constant across systems 1 to 5 because the component with $W_{MIN} = 18.1$ is used in all these systems.

We expected that breaking a system into separately pre-routed components would adversely effect its performance. Figure 5.14 shows the critical path length for each system when using the three compilation approaches. The MCA approach produced a consistent critical path length. Contrary to our expectation, the critical path length produced by the MCA approach was equal or less than that produced by both the MPA approach for all but one system. System 2 (with four PEs) produced a critical path length that was longer than that produced when using the MPA approach.

Figure 5.14 Critical path length for the six systems using the three different approaches

We found that, when using the MCA approach, the critical paths were always across a link between ports. System 1 (with 2 PEs) has a longer critical path length because the combination of ports on the corner PE and the host interface yields a longer path length than any other port combination. This combination of ports does not occur for any other system in the set.

Figure 5.15 shows the resource allocation (placer) effort for each system when using the three different compilation approaches. It can be seen from this graph that the placer effort increases linearly with system size when using the MPA approach. Whereas, the placer effort increases linearly with the number of unique PE components for the MRA and MCA approaches.

Figure 5.15 Placement effort for the six systems using the three different approaches

Figure 5.16 shows the connection allocation (router) effort for each system when using the three different compilation approaches. It can be seen from this graph that the router effort increases linearly with system size when using the MPA approach. Whereas, the router effort for the MRA approach, while less than MPA, still increases linearly with system size. However, the router effort for the MCA approach is much lower, increasing linearly with the number unique PEs.

Figure 5.16 Routing effort for the six systems using the three different approaches

The potential reduction in compiler effort (through reuse) for these systems, reported in Table 5.7, is small because of the small number of components reused within each system. We would expect that the reduction in compiler iterations would be in line with the potential reduction. However, due to the fact that the compilation problem is broken into smaller sub-problems the reduction in placer effort is greater than the predicted potential reduction. While the 20 PE system had a potential reduction of 2.22, the actual reduction in placer effort for the MRA and MCA approaches compared to the MPA

approach was 6.48 times. The reduction in router effort when using the MCA approach was 4.56.

Although we have created a system with a small number of PEs, it is possible to create a linear array of any length with the same set of PEs. For example, using the MCA approach to compile a system of 160 PEs would require a set of 8 port-edge combinations, yielding a reduction in compiler effort that would be in excess of 20 times.

In the next subsection, we will adapt the holistic functional density metric to explore the trade off between compiler effort and system performance when considering how long the system will run.

5.4.4 Compiler Effort and Computation Time

The performance of each approach is measured and compared in terms of construction time and the quality of the mapped circuit. For the purpose of comparing two compiler approaches, the holistic functional density metric (equation (3.2)) is simplified as follows: Assuming that the silicon area used for the two approaches being compared is the same and that the transfer time, t_T , is the same we can remove these from the equation and simplify it to:

$$D_n = \frac{1}{\left(t_{En} + \frac{t_P}{n}\right)} \quad (5.5)$$

This implies that for a short lived “instance specific” system where the number of circuit execution cycles, n , is small, the system preparation time, t_P , is more important than the circuit cycle time, t_{En} . For a long running system, where n is very large, the circuit cycle time, t_{En} , becomes more important than the system preparation time, t_P . Using this simplified equation for holistic functional density we are able to explore the trade-off between compiler effort and system performance for different system compilation approaches.

For the purpose of comparison, the preparation time, t_P , is estimated as the sum of I_{CA} and I_{RA} . Note we have not used absolute time because this is dependent on the speed of the system used for compilation. Furthermore, absolute time is irrelevant when comparing two techniques using the same base algorithms.

The circuit cycle execution time, t_{En} , is indicated by the critical path length. We are focusing on a fully buffered architecture where the interconnect delay is proportional to the number of wires used. Therefore, the critical path length, W_{LP} , is measured in wire hops from sink to source. An increase in W_{LP} indicates a proportional increase in t_{En} .

The graphs in Figure 5.17 are plotted using the compilation results from the 20 PE system in the previous subsection as an example. The holistic functional density is calculated over a range of values of execution cycles, n .

An increased holistic functional density indicates an overall improvement. The values for W_{LP} have been normalised to the MPA approach. Therefore the plot of D_n for the MPA approach tends towards 1.0 as n increases.

Figure 5.17 Holistic functional density plot for the 20 PE system

Because the critical path length produced by the MCA approach is less than that produced by the MPA approach the plot of D_n for the MCA approach tends towards 1.07. The plot of D_n for the MCA approach rises at an earlier value of n indicating that, because the compiler effort is less, the compilation is easier to amortize over a smaller number of useful compute iterations than when using the MPA approach.

5.5 Summary

Applying FPGA technology to a high performance computing application highlights the aspects of pre-routed components that require consideration. We have taken the Smith-Waterman dynamic programming algorithm used to score the similarity between genetic sequences and, using Fosters methodology, exploited the inherent parallelism to produce an FPGA based accelerator. The cell update tasks are agglomerated into fine grain PEs. The PEs communicate values along a pipelined linear array. This communication pattern maximises the utilisation of both the PE computational circuitry and the communication infrastructure. We found that we were able to fit hundreds of PEs onto current FPGA technology. Our FPGA based accelerator is competitive with other existing solutions, achieving a speed-up in the region of two orders of magnitude over a desktop processor, without compromising the accuracy of the algorithm by using heuristic methods.

FPGA technology lends itself to further performance increases through specialisation, on a per query basis and dynamically through a database scan. However, the compilation time and storage requirements for large numbers of specialised bit-streams makes the exploitation of specialisation less attractive. The regularity in the linear array of PEs allows the potential to apply a pre-routed component methodology to achieve a reduction in the compiler effort. Furthermore, we could reduce storage requirements by performing on-the-fly composition of systems from sets of pre-routed components.

The most significant outcome of our work was that by using our proposed pre-routed component methodology we are able to achieve a 20 times reduction in compiler effort for a system with 160 PEs. This is significantly better than existing techniques using tri-

state buffers or slice logic to lock signals between components. We expected a loss in performance, due to an increase in critical path length caused by locking signals to wires. However, we found that our MCA (pre-routed component) approach consistently had a lower critical path length as system size increased. Whereas, the MPA and MRA approaches saw an increase in the critical path length. Using the modified holistic functional density metric we showed that our proposed methodology has the potential to reduce the overhead of specialisation, giving a higher effective performance even for "small" computation runs of ten to a hundred million iterations. As FPGA devices and designs become larger, the time required for place and route has become more significant. This will eventually require the adoption of techniques to reduce this overhead. Our study has shown that significant reductions in place and route effort are possible with only a relatively modest reduction in performance.

6 Conclusions and Future Work

6.1 Conclusion

FPGA technology advances every 18 months, doubling the gate capacity available to the designer. The reconfigurability of FPGA technology enables the creation of many specialised high performance computing architectures. However, specialisation implies a design process. Realising the potential of FPGA technology is threatened by the gap between the number of available gates and the ability of designers to use these gates and meet time-to-market pressures.

Third party component reuse is one way of tackling design productivity. Providing libraries of commonly used functions is a recognised approach to improving design productivity. However, the third party reuse schemes available for FPGA design operate at either the source code or the net-list level. We investigate the use of pre-routed FPGA components to provide a method to capture and reuse the placement and routing effort used to map components. We then set about proving the following hypothesis:

“The productivity benefit of using pre-routed components outweighs any performance impact that may arise”

Firstly, the productivity benefit must be established and quantified. Allowing any performance impact to be weighed against it. Designer productivity is somewhat difficult to quantify due to its strong ergonomic aspects. Thus, we have focused on the aspect that is quantifiable: Compiler effort. We have developed the *holistic functional density metric* to illustrate the trade off between compiler effort and computing performance.

We find several aspects of component encapsulation that effect the computing performance of a system: *Fragmentation* of the FPGA surface between component regions will reduce the amount of resource that is available for useful computation; *communication overheads* will effect performance if links between components require resource that could have been usefully employed in computation; limitations on the *communication bandwidth* between components will restrict the rate of computation; and *routing congestion* caused by locking external signals will extend the critical path length, effectively restricting the maximum frequency of operation.

Pre-routing will only improve designer productivity when it is supported within an existing EDA work flow. Previous approaches to pre-routed components [Tess99], [Sedc04], [Tayl02], [Kalt04], [Patt03], [Xilap290], have only served as a proof of concept. Thus, it is somewhat difficult to ascertain their effect on designer productivity. With the exception of Tessier [Tess99], there has been few attempts at integrating pre-routed component support into EDA tools for FPGA mapping.

We have successfully integrated wire level component isolation into a conventional FPGA design flow. Although extra design artefacts such as *wire use policies*, *interface definitions* and *component templates* must be defined, these do not need revisiting once

they are written. Furthermore, the added design artefacts are reusable in the same way as other source code.

We created a real world FPGA application and applied our pre-routing technique to judge both the productivity benefit and the impact on performance. We took the Smith-Waterman dynamic programming algorithm used to score the similarity between genetic sequences and, using Fosters methodology, exploited the inherent parallelism to produce an FPGA based accelerator. Our FPGA based accelerator is competitive with other existing solutions, achieving a speed-up in the region of two orders of magnitude over a desktop processor, without compromising the accuracy of the algorithm by using heuristic methods. The regularity within the application was exploited to achieve a reduction in the compiler effort by using our pre-routed component methodology.

The application is sensitive to communications overheads because we are able to usefully employ almost all of the logic resource to increase computing performance, having to pass resource over to enable the communication infrastructure will reduce the potential for acceleration. Furthermore, communication between neighbouring components is performed during every step of the algorithm, a reduction in available communication bandwidth will strongly effect performance.

In a system of 160 PEs there is the potential to use a set of eight PE port-edge combinations to reduce the compiler effort by a factor of 20. We found, by analysis, that existing pre-routing methodologies are not able to achieve this level of compiler effort reduction. Using our newly proposed pre-routed component methodology that enables the mapping of interface signals to the wires bisected by component boundaries we would be able to achieve the 20 times reduction in compiler effort for a system with 160 PEs. We have found, because the placement and routing problems are broken into smaller problems, that we actually achieve a greater reduction in compiler effort than predicted.

Dynamic reconfiguration has driven the desire to pre-route components in many of the approaches [Sedc04], [Tayl02], [Kalt04], [Patt03], [Xilap290]. Thus, the concept of static and dynamic regions is a common feature. Due to the way the configuration memory relates the resource tiles, dynamic reconfiguration forces restrictions on component region size, position and inter component communication. Restrictions on component region size will cause *fragmentation*. We see pre-routed components as having a wider application. Thus, component region geometry is not constrained by the configuration memory.

Using our general purpose approach to pre-routing, this thesis provides a more thorough performance analysis that pushes inter-component bandwidth and resource utilisation with an aggressive set of benchmark systems. Fragmentation is kept to a minimum with components using over 90% of the resource within their regions. No extra logic resource was required to implement inter-component links. We found that an interface bandwidth utilisation of up to 30% of W_{FPGA} is achievable with an average increase in critical path length of 3 wire hops. We found that the lower bound for component region area was 16 tiles because a smaller area has less interconnect flexibility to deal with the added congestion of wire constraints. We found that placing *port areas* correctly has the potential to reduce the impact that pre-routing has on the performance of a mapped circuit.

Although our aggressive benchmarks showed an increase in critical path length of 3 wire hops, we found that in our real world HPC application the critical path length scaled better with system size when using our pre-routed component approach as compared to a normal approach to placement and routing.

Using the modified holistic functional density metric we showed that our proposed methodology has the potential to reduce the overhead of specialisation, giving a higher effective performance even for "small" computation runs of ten to a hundred million iterations.

Our results indicate that the productivity benefit does indeed outweigh the impact on performance. However, the world of FPGA design is far more complex than the handful of systems presented here. One aspect that is difficult to address is the restrictions on flexibility that pre-routed components impose on different system architectures.

6.2 Future Work

This work lays the foundations for systems composed of pre-compiled FPGA components that offer deterministic reuse without the overhead and complexity of further optimisation. There are a number of avenues for improvement and further study that could build on this thesis.

There are several aspects of the design flow that could be improved. We explicitly defined components and links in a separate system description. This system level description could be extracted from a HDL by the synthesis tools. There is further scope for integrating the interface definitions presented here with the new interface constructs added to HDL languages such as System Verilog.

The placement and routing algorithms used were intentionally basic. There is opportunity to add improvements such as clustering and range limiting in the SA algorithm, net sorting and sink sorting in the routing algorithm. With access to architectural details, the techniques presented here could be applied to commercial architectures, or indeed any architecture that uses a programmable interconnect built using segmented wires and switch boxes.

We have only considered FPGA resource of a single size. While the increased coarseness of devices may impact on the number of positions a pre-routed component can be relocated to, the approach presented in the thesis is based on identifying certain wires in the interconnect as isolation points. This is still relevant for coarse grain devices.

We have not attempted to optimise the wire allocation of an interface. Internal interfaces do not have the hard constraints that pad locations present. However, conceptually interfaces are designed for a set of components before they exist. Therefore, it could be argued that such an interface cannot be optimised for these components. It could be argued that the chances are that an interface optimised for one set of components will perform more poorly for another set of components than the original unoptimised interfaces. However, there is value in optimising the signal to wire allocation of an interface that will only be used for a finite set of known components. How to solve such an optimisation problem is left as future work.

We have only considered rectangular component regions in order to simplify the presentation of ideas. There is the potential to support components shaped into any polyomino. However, the more irregularly shaped the components are, the harder it will be to minimise fragmentation when composing a system.

One very interesting avenue for further study is pushing the granularity of isolation to even finer levels. In this thesis we presented isolation between rectangular regions, using bisected wires as the isolation point. We then went on to reserve a uniform set of wires within component regions for the purpose of tunnelling links. Tunnelling links were created by overlaying an appropriate interface extension that only used the reserved wire resource. Effectively, we shared the wire resource within a region between two circuit mappings. We could share equal portions of the interconnect between two circuit mappings. Now consider we extend the partitioning to include logic resource. By identifying the nodes of the interconnect graph that belong to each partition we could identify the edges (representing switch paths) that provide connectivity between the two. By predefining an interface between two such partitions, each could be compiled independently of the other. Such an approach would provide a higher number of connecting points than the connection points provided by wires bisected by the boundary of a rectangular region. There is no reason for such an approach to be restricted to two partitions.

Many previous works that created pre-routed components were inspired by dynamically reconfigurable systems. There is scope extending our newly developed framework for use in dynamic systems for rapid reconfiguration with a low computing overhead. Rapid composition of systems from libraries of binary components then becomes a reality. This leads to dynamic composition of a system from binary components. Such a system is able to reuse FPGA area, better adapt to a range of parameters, and allows designers to move specialisation decisions from compile time to run-time, thus extending the usefulness of a design.

7 Terminology

A_E	Silicon area dedicated to computational execution (See 3.1)
b	Cluster Level, the number of blocks in each cluster (See 4.2)
B	Circuit Blocks, measured in number of FPGA resource tiles (See 4.2)
B_{IF}	Percentage of circuit resource connected to component interfaces (See 4.6.1)
B_{OC}	Percentage of B resource tiles that are occupied within a component region (See 4.6.1)
B_{PA}	Percentage of circuit resource that occupies a port area (See 4.6.1)
CUPS	Cell Updates Per Second, the number of matrix cell calculations completed in a second (See 5.3.3)
D	Depth of a port area, measured in tiles (See 3.4.1)
D_n	Functional density of a system performing n computing iterations (See 3.1)
E	Interface edge length, measured in tiles (See 3.4.1)
F_{CO}	Flexibility of output connections, the number of wires a tile resource output can connect to (See 2.2.3)
F_{CI}	Flexibility of input connections, the number of wires a tile resource input can connect to (See 2.2.3)
F_S	Switch flexibility, the number of wire inputs each wire output can connect to (See 2.2.3)
I_{CA}	The number Connection Allocation iterations required to completely connect a system net-list (See 4.3)
I_{RA}	The number Resource Allocation iterations required to completely place a system net-list (See 4.3)
MPA	System construction approach that creates a system level net-list by merging all component instance net-lists after Primitive Allocation or “packing” has been performed on each component (See 4.1)
MRA	System construction approach that creates a system level net-list by merging all component instance net-lists after Resource Allocation or “placement” has been performed on each component (See 4.1)
MCA	System construction approach that creates a system level net-list by merging all component instance net-lists after Connection Allocation or “routing” has been performed on each component (See 4.1)
n	Number of computing execution iterations to be performed by a given system configuration (See 3.1)
NX	Component boundary perpendicular to the X axis, on the negative side (See 3.4.1)

7 Terminology		Page 119
NY	Component boundary perpendicular to the Y axis, on the negative side (See 3.4.1)	
PX	Component boundary perpendicular to the X axis, on the positive side (See 3.4.1)	
PY	Component boundary perpendicular to the Y axis, on the positive side (See 3.4.1)	
p	Rent exponent (See 4.2)	
T	The number of terminals on a circuit (See 4.2)	
T_b	The number of terminals on a cluster of size b blocks (See 4.2)	
t_{En}	The time taken to perform a computing execution iteration (See 3.2)	
T_{FPGA}	The number of terminals that a given area of FPGA architecture can support (See 4.4)	
t_p	System preparation time, including configuration compilation time (See 5.4.4)	
t_T	The time taken to transfer FPGA configuration (See 3.2)	
W	The number of wires in a wire set, equal to the number of tiles a wire spans when not truncated by the boundary of a device (See 3.2.3)	
W_{FPGA}	The number of interconnect channel wires in a given FPGA architecture (See 2.2.3)	
W_{IF}	Wire channel bandwidth allocated to interfacing, given as a percentage of W_{FPGA} (See 4.7)	
W_{IFmax}	The maximum number of wires available on a given interface surface (See 3.4.1)	
W_{IN}	Wire channel bandwidth allocated for internal connectivity, given as a percentage of W_{FPGA} (See 4.7)	
W_{LP}	The number of wires in the longest path, measured from sink to source (See 4.3)	
W_{MIN}	The predicted minimum number of interconnect channel wires required to connect a net-list (See 2.3.5)	
W_T	Wire channel bandwidth allocated to tunnelling inter-component links, given as a percentage of W_{FPGA} (See 4.7)	
W_U	Total number of wires used in the system (See 4.3)	

8 Glossary

ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory (Xilinx architectural element)
CLB	Configurable Logic Block (Xilinx architectural element)
CPU	Central Processing Unit
CSL	Configurable System Logic
CUPS	Cell Updates Per Second
DES	Data Encryption Standard
DNA	DeoxyriboNucleic Acid
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EDIF	Electronic Data Interchange Format
FF	Flip Flop
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HLL	High Level Language
HPC	High Performance Computing
IOB	Input Output Block (Xilinx architectural element)
IP	Intellectual Property
ISA	Instruction Set Architecture
ISEF	Instruction Set Extension Fabric
LUT	Look Up Table
MAC	Multiply ACcumulate
MCA	Merge system after Connection Allocation
MGT	Multi-Gigabit Transceiver
MPA	Merge system after Primitive Allocation
MRA	Merge system after Resource Allocation
MUX	MUltipleXer
NOC	Network On Chip
NRE	Non Recurring Engineering
PE	Processing Element
PIP	Programmable Interconnect Point (Xilinx architectural element)
RNA	RiboNucleic acid
RTL	Register Transfer Language
RTR	Run-Time Reconfiguration
SIMD	Single Instruction Multiple Data
SME	Small to Medium Enterprise
SRAM	Static Random Access Memory
VHDL	VHSIC (Very High Speed Integrated Circuit) HDL

9 Bibliography

- [Abra98] D Abramson, P Logothetis, A Postula and M Randall, "FPGA Based Custom Computing Machines for Irregular Problems," Int. Symp. High-Performance Computer Architecture, Las Vegas, NV, 1998, pp. 324-333.
- [Agil08] "Algorithms to Implementation," Agility Design Solutions Inc., Palo Alto, CA , 2008. [Online]. <http://www.agilityds.com>
- [Ahma04] A Ahmadinia, C Bobda, M Bednara and J Teich, "A New Approach for On-line Placement on Reconfigurable Devices," IEEE Int. Parallel and Distributed Processing Symp. Reconfigurable Architectures Workshop, Santa Fe, New Mexico, 2004, pp. 134a.
- [Ahma04b] A Ahmadinia, C Bobda, SP Fekete, J Teich and JC van der Veen, "Optimal Routing-Conscious Dynamic Placement for Reconfigurable Devices," Springer LNCS Field Programmable Logic and Applications, Vol. 3203, pp. 847-851, Aug., 2004.
- [Alph02] "Alpha-Data," Alpha-Data, Edinburgh, Scotland, 2002. [Online]. <http://www.alpha-data.co.uk>
- [Altep1s] "Stratix Architecture," Altera Corporation, San Jose, CA, S51002-3.2 2005.
- [Altep3a] "APEX II Programmable Logic Device Family," Altera Corporation, San Jose, CA, DS-APEXII-3.0 2002.
- [Altep3s] "Stratix III Device Family Overview," Altera Corporation, San Jose, CA, SIII51001-1.4 2008.
- [Altex02] "Excalibur Device Overview," Altera Corporation, San Jose, CA, DS-EXCARM-2.0 v2.2 2002.
- [Altmega] "MegaWizard Plug-Ins," Altera Corporation, San Jose, CA, 2008. [Online]. <http://www.altera.com/products/ip/altera/megawizd>
- [Altni02] "Nios 2.1 CPU," Altera Corporation, San Jose, CA, DS-NIOSCPU-1.1 2002.
- [Alts90] SF Altschul, W Gish, W Miller, EW Myers and DJ Lipman, "Basic local alignment search tool," Journal of Molecular Biology, Vol. 215, No. 3, pp. 403-410, Oct., 1990.
- [Altsopc] "SOPC Builder," Altera Corporation, San Jose, CA, 2008. [Online]. <http://www.altera.com/sopcbuilder>
- [Arno92] J Arnold D Buell and E Davis, "SPLASH II," ACM Annu. Symp. Parallel Algorithms and Architectures, San Diego, CA, 1992, pp. 316-322.
- [Atti05] M Attig and J Lockwood, "A Framework for Rule Processing in Reconfigurable Network Systems," IEEE Annu. Symp. Field-Programmable Custom Computing Machines , Napa, CA, 2005, pp. 225-234.

- [Aziz04] N Azizi, I Kuon, A Egier, A Darabiha and P Chow, "Reconfigurable Molecular Dynamics Simulator," IEEE Annu. Symp. Field-Programmable Custom Computing Machines , Napa, CA, 2004, pp. 197-206.
- [Baza99] K Bazargan and M Sarrafzadeh, "Fast Online Placement for Reconfigurable Computing Systems," IEEE Annu. Symp. Field-Programmable Custom Computing Machines , Napa, CA, 1999, pp. 300-302.
- [Beau06] MJ Beauchamp, S Hauck, KD Underwood and KS Hemmert, "Embedded Floating-Point Units in FPGAs," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2006, pp. 12-20.
- [Beck07] J Becker, A Donlin, and M Hubner, "New tool support and architectures in adaptive reconfigurable computing," IEEE Int. Conf. On Very Large Scale Integration of System-on-Chip, Atlanta, GA, 2007, pp. 134-139.
- [Betz05] D Lewis, E Ahmed, G Baeckler, V Betz, M Bourgeault, D Cashman, D Galloway, M Hutton, C Lane, A Lee and others, "The Stratix II Logic and Routing Architecture," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2005, pp. 14-20.
- [Betz96] V Betz and J Rose, "Directional Bias and Non-Uniformity in FPGA Global Routing Architectures," IEEE/ACM Int. Conf. Computer-Aided Design, San Jose, CA, 1996, pp. 652-659.
- [Betz99] V Betz and J Rose, "FPGA routing architecture: segmentation and buffering to optimize speed and density," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 1999, pp. 59-68.
- [Betz99a] V Betz, J Rose and A Marquardt, "Architecture and CAD for Deep-Submicron FPGAs," Kluwer Academic Publishers, 1999.
- [Bitt97] R Bittner and P Athanas, "Wormhole Run-time Reconfiguration," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 1997, pp. 79-85.
- [Bjes98] P Bjesse, K Claessen, M Sheeran and S Singh, "Lava: Hardware Design in Haskell," ACM SIGPLAN Int. Conf. Functional Programming, Baltimore, Maryland, 1998, pp. 174-184.
- [Blod00] B Blodget, "Pre-route Assistant: A Routing Tool for Run-Time Reconfiguration," Springer LNCS Field Programmable Logic and Applications, Vol. 1896, pp. 797-800, Aug., 2000.
- [Bobd04] C Bobda, M Majer, D Koch, A Ahmadinia and J Teich, "A Dynamic NoC Approach for Communication in Reconfigurable Devices," Springer LNCS Field Programmable Logic and Applications, Vol. 3203, pp. 1032-1036, Aug., 2004.
- [Bobd05] C Bobda, A Majer, A Ahmadinia, T Haller, A Linarth and J Teich, "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms," IEEE Int. Conf. Field-Programmable Technology, Singapore, 2005, pp. 37-42.
- [Boec03] B Boeckmann, A Bairoch, R Apweiler, MC Blatter, A Estreicher, E Gasteiger, MJ Martin, K Michoud, C O'Donovan, I Phan and others, "The

	9 Bibliography	Page 123
	SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003," Nucleic Acids Research, Vol. 31, No. 1, pp. 365-370, 2003.	
[Bora94]	M Borah RS Bajwa S Hannenhalli MJ Irwin, "A SIMD solution to the sequence comparison problem on the MGAP," IEEE Int. Conf. Application Specific Array Processors, San Francisco, CA, 1994, pp. 336-345.	
[Breb03]	G Brebner and D Levi, "Networking on Chip with Platform FPGAs," IEEE Int. Conf. Field-Programmable Technology, Tokyo, Japan, 2003, pp. 13-20.	
[Breb97]	GJ Brebner, "The swappable logic unit: a paradigm for virtual hardware," IEEE Annu. Symp. Field-Programmable Custom Computing Machines , Napa, CA, 1997, pp. 77-86.	
[Call00]	TJ Callahan, JR Hauser and J Wawrzynek, "The Garp Architecture and C Compiler," IEEE Computer, Vol. 33, No. 4, pp. 62-69, 2000.	
[Cami02]	NJ Camilleri and ES McGettigan, "Partial reconfiguration of a programmable gate array using a bus macro," US Patent 6462579, Oct., 8, 2002.	
[Chan03]	PK Chang MDF Schlag, "Parallel Placement for Field Programmable Gate Arrays," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2003, pp. 43-50.	
[Chan96]	YW Chang, DF Wong and CK Wong, "Universal switch-module design for symmetric array-based FPGAs," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 1996, pp. 80-86.	
[Char03]	SM Charlwood and SF Quigley, "The Impact of Routing Architecture on Reconfiguration Overheads," Int. Conf. Engineering of Reconfigurable Systems & Algorithms, Las Vegas, NV, 2003, pp. 102-110.	
[Chen04]	G Chen and J Cong, "Simultaneous Timing Driven Clustering and Placement for FPGAs," Springer LNCS Field Programmable Logic and Applications, Vol. 3203, pp. 158-167, Aug., 2004.	
[Chen94]	CLE Cheng, "RISA: Accurate and Efficient Placement Routability Modeling," IEEE/ACM Int. Conf. Computer-Aided Design, San Jose, CA, 1994, pp. 690-695.	
[Chow91]	E Chow, T Hunkapiller, J Peterson and MS Waterman, "Biological Information Signal Processor," IEEE Int. Conf. Application Specific Array Processors, Barcelona, Spain, 1991, pp. 144-160.	
[Cray05]	"Cray XD1 Data sheet," Cray Inc., Seattle, WA, Release 1.3 2005.	
[Dahl99]	D Dahle, L Grate, E Rice and R Hughey, "The UCSC Kestrel general purpose parallel processor," Int. Conf. Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, 1999, pp. 1243-1249.	
[DeHo00]	A DeHon, "The Density Advantage of Configurable Computing," IEEE Computer, Vol. 33, No. 4, pp. 41-49, 2000.	
[DeHo99]	A DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 1999, pp. 69-78.	

- [Dijk59] EW Dijkstra, "A note on two problems in connexion with graphs," Springer numerische Mathematik, Vol. 1, No. 1, pp. 269-271, Dec., 1959.
- [DRC06] "DRC The Coprocessor Company," DRC Computer Corporation, Santa Clara, CA, v2.2 2006.
- [Durb01] LJK Durbeck and NJ Macias, "The Cell Matrix: an architecture for nanocomputing," Nanotechnology, Vol. 12, No. 3, pp. 217-230, 2001.
- [Dyer02] M Dyer, C Plessl and M Platzner, "Partially Reconfigurable Cores for Xilinx Virtex," Springer LNCS Field Programmable Logic and Applications, Vol. 1896, pp. 292-301, Aug., 2000.
- [Ebel97] C Ebeling, "Whither Configurable Computing?," IEEE Hawaii Int. Conf. System Sciences, Maui, Hawaii, 1997, pp. 713.
- [Egur05] K Eguro, S Hauck and A Sharma, "Architecture-adaptive range limit windowing for simulated annealing FPGA placement," ACM Conf. Design Automation, San Diego, CA, 2005, pp. 439-444.
- [Estr05] G Estrin, "Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable Computer," IEEE Annals of the History of Computing, Vol. 24, No. 4, pp. 3-9, 2002.
- [Estr60] G Estrin, "Organization of Computer Systems: The Fixed Plus Variable Structure Computer," IRE-AIEE-ACM Western Joint IRE-AIEE-ACM Computer Conf., San Francisco, CA, 1960, pp. 33-40.
- [Fan01] H Fan, J Liu, YL Wu and CC Cheung, "On optimum switch box designs for 2-D FPGAs," ACM Conf. Design Automation, Las Vegas, NV, 2001, pp. 203-208.
- [Fend04] J Fender and J Rose, "A High-Speed Ray Tracing Engine Built on a Field-Programmable System," IEEE Int. Conf. Field-Programmable Technology, Brisbane, Australia, 2004, pp. 188-195.
- [Fost95] I Foster, "Designing and Building Parallel Programs," Addison-Wesley, 1995.
- [Frig01] J Frigo, M Gokhale and D Lavenier, "Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2001, pp. 134-140.
- [Gais08] "Processors," Gaisler Research AB, Goteborg, Sweden, 2008. [Online]. <http://www.gaisler.com/leonmain.html>
- [Gajs83] DD Gajski and RH Kuhn, "New VLSI Tools," IEEE Computer, Vol. 16, No. 12, pp. 11-14, 1983.
- [Gaya04] A Gayasen, Y Tsai, N Vijaykrishnan, M Kandemir, MJ Irwin and T Tuan, "Reducing Leakage Energy in FPGAs Using Region-Constrained Placement," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2004, pp. 51-58.
- [Glem97] E Glemet and JJ Codani, "LASSAP, a Large Scale Sequence compArison Package," Bioinformatics, Vol. 13, No. 2, pp. 137-143, 1997.

	9 Bibliography	Page 125
[Gokh95]	M Gokhale, B Holmes and K Iobst, "Processing in memory: The Terasys massively parallel PIM array," IEEE Computer, Vol. 28, No. 4, pp. 23-31, 1995.	
[Gucc02]	SA Guccione and E Keller, "Gene Matching using JBits," Springer LNCS Field Programmable Logic and Applications, Vol. 2438, pp. 1168-1171, Feb., 2002.	
[Gucc99]	S Guccione and D Levi, "Run-Time Parameterizable Cores," Springer LNCS Field Programmable Logic and Applications, Vol. 1673, pp. 215-222, Aug., 1999.	
[Guer97]	P Guerdoux-Jamet and D Lavenier, "SAMBA: hardware accelerator for biological sequence comparison," Bioinformatics, Vol. 13, No. 6, pp. 609-615, 1997.	
[Guo04]	Z Guo, W Najjar, F Vahid and K Vissers, "A Quantitative Analysis of the Speedup Factors of FPGAs over Processors," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2004, pp. 162-170.	
[Hand04]	M Handa and R Vemuri, "Area Fragmentation in Reconfigurable Operating Systems," Int. Conf. Engineering of Reconfigurable Systems & Algorithms, Las Vegas, NV, 2004, pp. 77-83.	
[Haus00]	JR Hauser, "Augmenting a Microprocessor with Reconfigurable Hardware," Ph.D dissertation, University of California, Berkeley, CA, 2000.	
[Hero01]	JP Heron, R Woods, S Sezer and RH Turner, "Development of a Run-Time Reconfiguration System with low reconfiguration overhead," Journal of VLSI Signal Processing, Vol. 28, No. 1, pp. 97-113, 2001.	
[Hoan93]	DT Hoang, "Searching genetic databases on Splash 2," IEEE Workshop FPGAs for Custom Computing Machines, Napa, CA, 1993, pp. 185-191.	
[Hoar04]	R Hoare, S Tung and K Werger, "An 88-Way Multiprocessor Within An FPGA With Customizable Instructions," IEEE Int. Parallel and Distributed Processing Symp. Workshop on Massively Parallel Processing, Santa Fe, New Mexico, 2004, pp. 258b.	
[Holl05]	B Holland, M Vacas, V Aggarwal, R DeVille, I Troxel and AD George, "Survey of C-based Application Mapping Tools for Reconfigurable Computing," Int. Conf. Military and Aerospace Programmable Logic Devices, Washington, DC, 2005, [Online]. http://klabs.org/mapld/ .	
[Hort02]	EL Horta, JW Lockwood, DE Taylor and D Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," ACM Conf. Design Automation, New Orleans, LA, 2002, pp. 343-348.	
[Hubn06]	M Hubner, C Schuck and J Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs," IEEE Int. Parallel and Distributed Processing Symp., Rhodes Island, Greece, 2006, pp. 192.	
[Hugh96]	R Hughey, "Parallel Hardware for Sequence Comparison and Alignment," Bioinformatics, Vol. 12, No. 6, pp. 473-479, 1996.	

- [Hwan98] J Hwang, C Patterson, S Mohan, E Dellinger, S Mitra and R Wittig, "Generating Layouts for Self-Implementing Modules," Springer LNCS Field Programmable Logic and Applications, Vol. 1482, pp. 525-529, Aug., 1998.
- [Hype08] "HyperTransport Consortium," HyperTransport Consortium, Sunnyvale, CA, 2008. [Online]. <http://www.hypertransport.org>
- [Imra99] M Imran Masud and SJE Wilton, "A New Switch Block for Segmented FPGAs," Springer LNCS Field Programmable Logic and Applications, Vol. 1673, pp. 274-281, Aug., 1999.
- [ITRS08] "ITRS Update 2008," The International Technology Roadmap for Semiconductors, 2008. [Online]. <http://www.itrs.net/Links/2008ITRS/Home2008.htm>
- [Kalt04] H Kalte, M Porrmann and U Rückert, "System-on-programmable-chip approach enabling online fine-grained 1D-placement," IEEE Int. Parallel and Distributed Processing Symp. Reconfigurable Architectures Workshop, Santa Fe, New Mexico, 2004, pp. 141a.
- [Kast02] R Kastner, "Synthesis Techniques and Optimizations for Reconfigurable Systems," Ph.D dissertation, University of California, Los Angeles, CA, 2002.
- [Kell00] E Keller, "JRoute: A Run-Time Routing API for FPGA Hardware," Springer LNCS Int. Parallel and Distributed Processing Symp. Reconfigurable Architectures Workshop, Vol. 1800, pp. 874-881, May, 2000.
- [Kell03] E Keller and S McMillan, "An FPGA Wire Database for Run-Time Routers," Int. Conf. Military and Aerospace Programmable Logic Devices, Washington, DC, 2003, [Online]. <http://klabs.org/mapld/>.
- [Koes05] M Koester, M Porrmann and H Kalte, "Task Placement for Heterogeneous Reconfigurable Architectures," IEEE Int. Conf. Field-Programmable Technology, Singapore, 2005, pp. 43-50.
- [Kung88] SY Kung, "VLSI array processors: designs and applications," IEEE Int. Symp. Circuits and Systems, Espoo, Finland, 1988, pp. 313-320.
- [Lamo03] J Lamoureux and SJE Wilton, "On the Interaction Between Power-Aware FPGA CAD Algorithms," IEEE/ACM Int. Conf. Computer-Aided Design, San Jose, CA, 2003, pp. 701-708.
- [Land71] BS Landman and RL Russo, "On a pin versus block relationship for partitions of logic graphs," IEEE Trans. On Computers, Vol. 20, No. 12, pp. 1469-1479, 1971.
- [Lave98] D Lavenier and JL Pacherie, "Parallel Processing for Scanning Genomic Data-Bases," Conf. Parallel Computing: Fundamentals, Applications and New Directions, Vol. 12, pp. 81-88, Sep.1997.
- [Lemi02] GG Lemieux and DM Lewis, "Analytical Framework for Switch Block Design," Springer LNCS Field Programmable Logic and Applications, Vol. 2438, pp. 122-131, Sep., 2002.

	9 Bibliography	Page 127
[Lemi04]	G Lemieux, E Lee, M Tom and A Yu, "Directional and Single-Driver Wires in FPGA Interconnect," IEEE Int. Conf. Field-Programmable Technology, Brisbane, Australia, 2004, pp. 41-48.	
[Lewi03]	D Lewis, E Ahmed, G Baeckler, V Betz, M Bourgeault, D Cashman, D Galloway, M Hutton, C Lane, A Lee and others, "The Stratix TM Routing and Logic Architecture," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2003, pp. 12-20.	
[Lewi98]	DM Lewis, DR Galloway, M Van Ierssel, J Rose and P Chow, "The Transmogrieffier-2: A 1 Million Gate Rapid Prototyping System," IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 6, No. 2, pp. 188-198, 1998.	
[Lopr87]	DP Lopresti, "P-NAC: A systolic array for comparing nucleic acid sequences," IEEE Computer, Vol. 20, No. 7, pp. 98-99, 1987.	
[m2000]	"FlexEOS Embedded FPGA Cores," m2000, Bièvres, France, FlexEOS 2003.	
[MacB01]	J MacBeth and P Lysaght, "Dynamically Reconfigurable Cores," Springer LNCS Field Programmable Logic and Applications, Vol. 2147, pp. 462-472, Aug., 2001.	
[Mali05]	U Malik and O Diessel, "A configuration memory architecture for fast run-time reconfiguration of FPGAs," IEEE Int. Conf. Field Programmable Logic and Applications, Tampere, Finland, 2005, pp. 636-639.	
[Mare02]	T Marescaux, A Bartic, D Verkest, S Vernalde and R Lauwereins, "Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs," Springer LNCS Field Programmable Logic and Applications, Vol. 2438, pp. 795-805, Sep., 2002.	
[McMu95]	L McMurchie and C Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 1995, pp. 111-117.	
[Menc02]	O Mencer, "PAM-Blox II: Design and Evaluation of C++ Module Generation for Computing with FPGAs," IEEE Annu. Symp. Field-Programmable Custom Computing Machines, Napa, CA, 2002, pp. 67-76.	
[Moha98]	S Mohan, R Wittig, S Kelem, and S Leavesley, "The Core Generator Framework," Canadian Workshop on Field-Programmable Devices, Montreal, Canada, 1998, pp. 1-6.	
[Naka99]	H Nakada, K Oguri, N Imlig, M Inamori, R Konishi, H Ita, K Nagami and T Shiozawa, "Plastic Cell Architecture: A Dynamically Reconfigurable Hardware-based Computer," Springer LNCS Int. Parallel and Distributed Processing Symp. Reconfigurable Architectures Workshop, Vol. 1586, pp. 679-687, Apr., 1999.	
[Nall08]	"Nallatech, High Performance FPGA computing for Defence and HPC," Nallatech, Glasgow, Scotland, 2008. [Online]. http://www.nallatech.com	
[OCP02]	"The Importance of Sockets in SOC Design," Open Core Protocol International Partnership (OCP-IP), Beaverton, Oregon, 2002. [Online]. http://www.ocpip.org/data/sockets_socdesign.pdf	

- [Oliv04] TF Oliver, S Mohammed, NM Krishna and DL Maskell, "Accelerating an Embedded RTOS in a SOPC Platform," IEEE Int. Region 10 Conf., Chiang Mai, Thailand, 2004, pp. 415-418.
- [Oliv05] TF Oliver, B Schmidt and DL Maskell, "Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2005, pp. 229-237.
- [Open08] "OpenRISC 1000: Overview," OpenCores.Org, Sweden, 2008. [Online]. <http://www.opencores.org/?do=project&who=or1k>
- [OSCI08] "Open SystemC Initiative (OSCI)," OSCI, USA, 2008. [Online]. <http://www.systemc.org/home>
- [Patt00] C Patterson, "High Performance DES Encryption in Virtex FPGAs Using JBits," IEEE Annu. Symp. Field-Programmable Custom Computing Machines, Napa, CA, 2000, pp. 113-121.
- [Patt01] CD Patterson, EF Dellinger, LJ Hwang, S Mitra, S Mohan and RD Wittig, "Method for Constraining Circuit Elements Positions in Structured Layouts," US Patent 6237129, May, 22, 2001.
- [Patt03] C Patterson, "A Dynamic Module Server for Embedded Platform FPGAs," Int. Conf. Engineering of Reconfigurable Systems & Algorithms, Las Vegas, NV, 2003, pp. 31-40.
- [Pear95] WR Pearson, "Comparison of methods for searching protein sequence databases," Protein Science, Vol. 4, No. 6, pp. 1145-1160, 1995.
- [Pell05] D Pellerin and S Thibault, "Practical FPGA Programming in C," Prentice Hall, 2005.
- [Plun04] B Plunkett and J Watson, "Adapt2400 ACM Architecture Overview," Quicksilver Technology Inc., San Jose, CA, 40202-BPAF 2004.
- [Rose91] JS Rose and S Brown, "Flexibility of Interconnection Structures for FPGAs," IEEE Journal of Solid-State Circuits, Vol. 26, No. 3, pp. 277-282, 1991.
- [Russ72] RL Russo, "On the tradeoff between logic performance and circuit to pin ratio for LSI," IEEE Trans. On Computers, Vol. 21, No. 2, pp. 147-153, 1972.
- [Sank99] Y Sankar and J Rose, "Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 1999, pp. 157-166.
- [Schm02] B Schmidt, H Schröder and M Schimmmler, "Massively Parallel Solutions for Molecular Sequence Analysis," IEEE Int. Parallel and Distributed Processing Symp. Workshop on High Performance Computational Biology, Fort Lauderdale, Florida, 2002, pp. 186-193.
- [Sedc04] P Sedcole, PYK Cheung, G Constantinides and W Luk, "A Structured System Methodology for FPGA Based System-on-a-Chip Design," IEEE Annu. Symp. Field-Programmable Custom Computing Machines, Napa, CA, 2004, pp. 271-272.
- [Sedc05] P Sedcole, B Blodget, J Anderson, P Lysaght and T Becker, "Modular dynamic reconfiguration in Virtex FPGAs," IEEE Int. Conf. Field

- Programmable Logic and Applications, Tampere, Finland, 2005, pp. 211-216.
- [Sedc06] NP Sedcole, "Reconfigurable Platform-Based Design in FPGAs for Video Image Processing," Ph.D dissertation, Dept. Electrical and Electronic Engineering, Imperial College of Science, Technology and Medicine, University of London, England, 2006.
- [Seki91] Y Sekiyama, Y Fujihara, T Hayashi, M Seki, J Kusuhara, K Iijima, M Takakura, and K Fukatani, "Timing-oriented routers for PCB layout design of high-performance computers," IEEE/ACM Int. Conf. Computer-Aided Design, Santa Clara, CA, 1991, pp. 332-335 .
- [SGI08] "SGI RASC Technology," SGI, Sunnyvale, CA, 2008. [Online]. <http://www.sgi.com/products/rasc>
- [Shir98] N Shirazi, W Luk and PYK Cheung, "Automating Production of Run-Time Reconfigurable Designs," IEEE Annu. Symp. Field-Programmable Custom Computing Machines , Napa, CA, 1998, pp. 147-156.
- [Sing00] S Singh, "Death of the RLOC?," IEEE Annu. Symp. Field-Programmable Custom Computing Machines , Napa, CA, 2000, pp. 145-152.
- [Sing02] A Singh and M Marek-Sadowska, "FPGA Interconnect Planning," IEEE/ACM Int. Workshop System-Level Interconnect Prediction, San Diego, CA, 2002, pp. 23-30.
- [Sing04] S Singh, "Designing reconfigurable systems in Lava," IEEE Int. Conf. VLSI Design, Mumbai, India, 2004, pp. 299-306.
- [Sing96] RK Singh, DL Hoffman, SG Tell and CT White, "BIOSCAN: a network sharable computational resource for searching biosequence databases," Computer Applications in the Biosciences , Vol. 12, No. 3, pp. 191-196, 1996.
- [Smit81] TF Smith and MS Waterman, "Identification of common molecular subsequences," Journal of Molecular Biology, Vol. 147, No. 1, pp. 195-197, Mar., 1981.
- [Star05] "Hypercomputers," Starbridge Systems Inc., Salt Lake City, UT , 2005. [Online]. <http://www.starbridgesystems.com>
- [Stre08] "Stretch Technology," Stretch Inc., Sunnyvale, CA , 2008. [Online]. <http://www.stretchinc.com/technology/index.php>
- [Stro00] D Stroobandt, P Verplaetse and JV Campenhout, "Generating Synthetic Benchmark Circuit for Evaluating CAD Tools," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 19, No. 9, pp. 1011-1022, 2000.
- [Stro98] D Stroobandt and FJ Kurdahi, "On the characterization of multi-point nets in electronic designs," IEEE Great Lakes Symp. VLSI, Lafayette, LA, 1998, pp. 344-350.
- [Stur93] S Sturrock and J Collins "MPsrch version 1.3," Biocomputing Research Unit, University of Edinburgh, Scotland, 1993.

- [Swar98] JS Swartz, V Betz and J Rose, "A Fast Routability-Driven Router for FPGAs," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 1998, pp. 140-149.
- [Tava97] D Tavana, WK Yee, and VA Holen, "FPGA architecture with repeatable titles including routing matrices and logic matrices," US Patent 5883525, Oct., 3, 1997.
- [Tayl02] DE Taylor, JS Turner, JW Lockwood and EL Horta, "Dynamic hardware plugins: exploiting reconfigurable hardware for high-performance programmable routers," Computer Networks, Vol. 38, No. 3, pp. 295-310, Feb., 2002.
- [Tess01] R Tessier and W Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey," Journal of VLSI Signal Processing, Vol. 28, No. 1, pp. 7-27, 2001.
- [Tess99] RG Tessier, "Fast Place and Route Approaches for FPGAs," Ph.D dissertation, Dept. Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1999.
- [Time08] "TimeLogic Corporation," TimeLogic Corporation, Carlsbad, CA , 2008. [Online]. <http://www.timelogic.com>
- [Todm05] T Todman, JGF Coutinho and W Luk, "Customisable hardware compilation," The Journal of Supercomputing, Vol. 32, No. 2, pp. 119-137, 2005.
- [Tris5] "Triscend E5 Customizable Microcontroller Platform," Triscend Inc., Mountain View, CA, TCH300-0001-001 v1.07 2003.
- [Tris7] "Triscend A7S Configurable System-on-Chip Platform," Triscend Inc., Mountain View, CA, TCH305-0001-002 v1.2 2002.
- [Tsen92] B Tsent, J Rose and S Brown, "Improving FPGA routing architectures using architecture and CAD interactions," IEEE Int. Conf. Computer Design: VLSI in Computer and Processors, Cambridge, MA, 1992, pp. 99-104.
- [Unde04] KD Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," ACM Int. Symp. Field-Programmable Gate Arrays, Monterey, CA, 2004, pp. 171-180.
- [Wald03] H Walder and M Platzner, "Online Scheduling for Block-partitioned Reconfigurable Devices," Conf. Design, Automation and Test in Europe, Munich, Germany, 2003, pp. 290-295.
- [Wald04] H Walder and M Platzner, "A Runtime Environment for Reconfigurable Hardware Operating Systems," Springer LNCS Field Programmable Logic and Applications, Vol. 3203, pp. 831-835, Aug., 2004.
- [Wang03] M Wang, A Ranjan and A Raje, "Multi-Million Gate FPGA Physical Design Challenges," IEEE/ACM Int. Conf. Computer-Aided Design, San Jose, CA, 2003, pp. 891-898.
- [Wigl05] G Wigley, "An Operating System for Reconfigurable Computing," Ph.D dissertation, School of Computer and Information Science, University of South Australia, Adelaide, Australia, 2005.

	9 Bibliography	Page 131
[Will06]	"Icarus Verilog," S Williams, Online, 2006. [Online]. http://www.icarus.com/eda/verilog	
[Wilt97]	SJE Wilton, "Architectures and Algorithms for FPGAs with Embedded Memory," Ph.D dissertation, Dept. Electrical and Computer Engineering, University of Toronto, Ontario, Canada, 1997.	
[Wirt95]	MJ Wirthlin and BL Hutchings, "DISC: The dynamic instruction set computer," SPIE Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Philadelphia, PA, 1995, pp. 92-103.	
[Wirt97]	MJ Wirthlin, "Improving Functional Density Through Run-time Circuit Reconfiguration," Ph.D dissertation, Dept. Electrical and Computer Engineering, Brigham Young University, Provo, UT, 1997.	
[Xilap290]	"Two Flows for Partial Reconfiguration: Module Based or Difference Based," Xilinx Inc., San Jose, CA, v1.4 2004.	
[Xilds003]	"Virtex 2.5 V Field Programmable Gate Arrays," Xilinx Inc., San Jose, CA, DS003 v2.5 2001.	
[Xilds031]	"Virtex-II Platform FPGAs," Xilinx Inc., San Jose, CA, DS031 v1.9 2002.	
[Xilds083]	"Virtex-II Pro Platform FPGAs: Introduction and Overview," Xilinx Inc., San Jose, CA, DS083 v2.4.2 2003.	
[Xilds100]	"Virtex-5 Family Overview," Xilinx Inc., San Jose, CA, DS100 v3.0 2007.	
[Xilds112]	"Virtex-4 Family Overview," Xilinx Inc., San Jose, CA, DS112 v2.0 2007.	
[Xilfs07]	"Fast Simplex Link (FSL) Bus," Xilinx Inc., San Jose, CA, DS449 v2.11a 2007.	
[Xilm08]	"MicroBlaze Processor," Xilinx Inc., San Jose, CA, 2008. [Online]. http://www.xilinx.com/microblaze	
[Xilp08]	"PicoBlaze," Xilinx Inc., San Jose, CA, 2008. [Online]. http://www.xilinx.com/picoblaze	
[Xilps08]	"Platform Studio and the EDK," Xilinx Inc., San Jose, CA, 2008. [Online]. http://www.xilinx.com/edk	
[Xilug208]	"Early Access Partial Reconfiguration User Guide," Xilinx Inc., San Jose, CA, UG208 v1.1 2006.	
[Xtre06]	"XD1000 FPGA Coprocessor Module for Socket 940," XtremeData Inc., Schaumburg, IL, v1.2 2006.	
[Xu03]	W Xu, R Ramanarayanan and R Tessier, "Adaptive Fault Recovery for Networked Reconfigurable Systems," IEEE Annu. Symp. Field-Programmable Custom Computing Machines, Napa, CA, 2003, pp. 143-152.	
[Yala02]	S Yalamanchili, "The Customization Landscape for Embedded Systems," Int. Conf. High Performance Computing, Bangalore, India, 2002, pp. 693-696.	
[Yama02]	Y Yamaguchi, T Maruyama and A Konagaya, "High Speed Homology Search with FPGAs," Pacific Symp. Biocomputing, Lihue, Hawaii, 2002, pp. 271-282.	

- [Ye00] ZA Ye, A Moshovos, S Hauck and P Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," Annu. Int. Symp. Computer Architecture, Vancouver, BC, Canada, 2000, pp. 225-235.
- [Yosh82] T Yoshimura and ES Kuh, "Efficient Algorithms for Channel Routing," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, Vol. 1, No. 1, pp. 25-35, Jan., 1982.
- [Yu03] CW Yu, KH Kwong, KH Lee and PHW Leong, "A Smith-Waterman Systolic Cell," Springer LNCS Field Programmable Logic and Applications, Vol. 2778, pp. 375-384, Sep., 2003.