Mapping Recursive Functions To Reconfigurable Hardware

George Ferizis

Acknowledgements

I would like to begin by thanking my supervisor Hossam El Gindy. Hossam has been wonderfully supportive during the tenure of my PhD program, both academically and personally. He has constantly been there to provide a usually gentle guiding hand throughout the last few years. He has been the largest influence on the way I view my career, computer science and that influence has not only touched my academic and professional growth, but also my personal growth. He is somebody who I am lucky and proud to be able to call a friend.

I would like to thank the other people I have the fortunate ability to say I have worked with, bounced ideas off, and engaged in general conversation with over the last three years while sitting at my desk. In particular my fellow PhD candidates, Keith So and Usama Malik who have both been willing to listen too and comment on ideas that I float about no matter how absurd they are. After Hossam it was these two who have provided the most valuable feedback during the last few years.

To my friends and extended family whom over the last few years have been there for me, both to serve as a distraction and to provide moral support in all aspects of my life I give my thanks. I doubt I can begin to list all of the friends and extended family who have made my life easier over the last few years without producing a 600 page document, but I would like to thank two people in particular. I give a hearty salute to John Bowers for the way that he has always been available to lend a sympathetic ear when it was required and to provide many the abominable joke, both when levity needed to come knocking on the door, and when perhaps it should have stayed in the next suburb. I would also like to thank Honghanh Nguyen for the large amount of support that she has provided me over the last few years, and in particular for the pride she has displayed towards me at various milestones over the last few years which has provided me with encouragement to push forward that little bit further.

And now onto the most important people that require acknowledgement - and people who should in all fairness have the letters PhD following their name after the blood, sweat and tears that they have put into my education - my parents Chris and Marcha and my sister Maria who have supported me through my academic career, emotionally when it all felt like it was becoming a bit too hard and financially when it was required. Their belief in my abilities and constant encour-

agement not only over the last few years but also over my entire life, coupled with the warm and caring household that they all contribute towards has enabled me to be in the position to write this acknowledgement. Mum and Dad; I doubt that I will ever be able to repay you for the tools that you have both given me.

Abstract

Reconfigurable computing is a design methodology that provides a developer with the ability to reprogram a hardware device. Current Field Programmable Gate Array (FPGA) systems, in particular, allow for the rapid and cost effective implementation of hardware devices when compared to standard ASIC design, and for the increase in performance when compared to software-based solutions. The advent of development tools such as Celoxica's DK package and Xilinx's Forge package, which support high level languages traditionally associated with software development, allows for software programmers to easily acquire the skills needed to develop reconfigurable solutions that were previously reserved for hardware designers. The desirable goal of cost effective high performance systems and increasing the adoption of FPGA technologies may thus be achieved. Such tools aim to closely mirror current software development tools in terms of language, syntax and methodology, and at the same time take advantage of the increased performance that reconfigurable architectures can provide through parallelism and arbitrary-depth pipelining in a transparent and automated way.

A common feature of many programming languages that is not supported by many higher-level hardware design tools is recursion. Recursion is a powerful method used to elegantly describe many algorithms that is typically implemented by using a stack to store arguments, context and a return address for function calls. However a stack-based implementation limits the hardware to running only a single function at any moment. Such implementation eliminates the possibility of taking advantage of parallelism, provided by the resources in FPGA systems, which can be used to simultaneously process successive iterations of a recursive function. The full potential in performance may not be achieved. We present a method to address the lack of support for recursion in design tools that exploits the parallelism available between recursive calls. Our main approach is to unroll the recursion into a pipeline, in a similar manner to the pipeline obtained from loop unrolling, and then to stream the data through the resulting pipeline. The differences between loops and recursive functions such as multiple recursive calls in a function, and hence multiple unrolling, and post-recursive statements add further complexity to the issue of unrolling as the pipeline may take a non-linear shape and contain heterogeneous stages.

Unrolling the recursive function into an FPGA system increases the parallelism available. However the depth of the pipeline, and thus

the amount of parallelism available, is limited by the finite resources on the FPGA device. We make efficient use of the FPGA resources by unrolling the function in a way to best suit the input and ensure that the function is not unrolled past its maximum recursive depth. A straightforward solution such as unrolling on-demand introduces a latency into the system when a further instance of the function is required that reduces overall performance. We reduce this penalty by using methods to predict the behaviour of the recursive function based on the input data and unroll the function to a suitable length prior to it being required. Accurate prediction is possible in cases where the condition for recursion is a simple function on the arguments. However accurate prediction is not possible in cases where the condition for recursion is based on complex functions. In such situations we use a heuristic to provide an approximation to the correct depth of recursion at any given time. This prediction allows the system to reduce the performance penalty from real time unrolling without over utilization of the FPGA resources.

Our results demonstrate the increase in performance for various recursive functions when compared to a stack-based implementation on the same device. Experiments with polynomail logarithmic algorithms such as quick sort, merge sort and quad tree partitioning have resulted in implementations running in linear time. Further case stud-

ies into a polynomial problem of matrix multiplication has also resulted in an increase in performance. The results, that are presented in this thesis, of binary tree insertion and search being mapped to reconfigurable hardware demonstrates that the pipeline generated by the function unrolling provides further performance benefits by allowing the pipeline to be executing multiple instances of the function in parallel. In the case of binary tree search, constant throughput is obtained regardless of the size of the tree being searched.

In certain instances due to constraints on hardware availability, such as a lack of hardware that properly supports runtime reconfiguration, and hardware with a low number of logic gates, results were gained from device simulation using a simulator developed for this purpose. Details of this simulator are presented in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recursion is an effective design tool that is used to describe many important algorithms and data structures. As the adoption of higher-level language to FPGA design tools increases, the lack of support for recursive functions and the difficulty in supporting such functions without a stack is becoming glaringly obvious.

In this chapter I present an abridged description of the background that motivates the inclusion of recursive functions in higher-level design tools, describe the methodology I adopt to automate the mapping of such functions into hardware suitable for FPGA implementation, and then outline limitations of my methodology. Finally I outline the remainder of the thesis and highlight my original contributions.

1.1 Introduction

Reconfigurable computing is a method of computing where general purpose hardware is configured for a specific task, but may be reconfigured for a different task at a later date.

An example of a reconfigurable computing device is FPGA, which consists of a set of logic cells that can be programmed to form any logic function provided that sufficient placement and routing resources are available on the device.

A comparison of an FPGA-based solutions to an ASIC designs shows that the FPGA based solution has a better cost of development and a lower time to develop, but however does not perform as well as an ASIC design. A comparison of an FPGA-based solution to software based solution shows that the FPGA solution may take longer to develop but does provide better performance results. From these observations it can be seen that when considering the metrics of performance and development cost and time an FPGA based solution resides between ASIC designs and software solutions.

Traditionally, designing for FPGA-based systems has involved the implementation of hardware designs in hardware design languages such as VHDL, or Verilog. While such languages are appropriate for hardware designers and their designs, attempts to expand the use of FPGA technology among software developers, who may lack hard-

ware design background, favoured a move towards familiar software design languages [1, 3, 35, 48, 20], such as C and Java.

These tools are typically designed to mirror the interface of common software IDE packages. The tools mirror the interfaces in terms of "look-and-feel" and provide syntax that is very similar to the software tools. By providing this syntax the tools hide the majority of the hardware details from the developer with the objective of providing a shorter development time for complex designs without the need of extensive hardware design skills. However such approach has the potential to produce designs with lower performance and larger use of FPGA resources. In addition, some of the high-level language tools fall short of their objectives as they force the developer to think of cycle-by-cycle operations during the development of their application and, of particular relevance to this thesis, only support a subset of the language.

1.2 Problem Definition

This thesis addresses the particular problem of support for recursive functions in high-level design tools. Maruyama et al [48] have developed the only design tool to date that supports recursive functions. It supports recursive functions by builiding a memory stack to store

intermediary data.

The use of a stack greatly reduces the amount of parallelism that can be extracted from the recursive call as it reduces the ability of the system to run multiple instances of the function in parallel. Recursion is typically implemented on microprocessor systems with a stack, but with an FPGA-based system that provides the ability for massive amounts of parallelism, a stack-based solution clearly falls short of realising its performance potential. This thesis puts forward a method to automatically generate circuits for recursive functions that will exploit the parallelism available in the entire recursive call.

1.3 Research justification

Recursion is commonly used in many high performance algorithms, with all algorithms belonging to the "divide-and-conquer" class being defined recursively. Such functions can be transformed into loops [46, 7, 6]. However functions that must store variables defined prior to the execution of a recursive call require the use of a memory stack. Such functions cannot be mapped into hardware using regular loop unrolling techniques due to the stack maintenance acting as a loop carried dependency.

Performance issues aside, the lack of support for recursion, even

stack based recursion, forces the developer to either abandon the use of recursion or to implement a stack design of their own in cases where it is necessary. Such situation is not compatible with the prime purpose of these tools, which is to allow the developer to work in a familiar environment. Thus automated support for recursive functions should be an integral part of any high-level design tool.

1.4 Methodology overview

For the context of the research presented in this thesis the execution of a recursive function is modelled as a tree. This tree matches the call graph of the recursive function as it executes. A call graph is a graph representation of the function calls made during the execution of a program. In the case of a call graph for a recursive function, each vertex of the graph corresponds to an instance of the function, and an edge exists from each function to every instance that it instantiates via a recursive call. The call graph obtained for a recursive function is named the recursive tree throughout this dissertation.

Two types of inter-procedural parallelism are present between recursive calls. The first stems from parallel execution of recursive calls made by a function, which exploits the parallelism available between nodes of the recursive tree on the same level. The second type intro-

duces parallelism by having nodes operate on partial data, thus having nodes in different levels operating in parallel. This parallelism is present in functions that operate on streams of data and output data before processing the entire stream.

The research presented in this thesis describes a method that exploits this parallelism by creating a pipeline whose stages contain instances of the recursive function that implement function in-lining on recursive calls. The pipeline matches the shape of the call graph created by the recursive function. Similar techniques have been previously used to map loops to reconfigurable hardware [17, 30, 37] where each stage of the pipeline corresponds to an unrolled instance of the original statements. The differences between the two methods are that the pipeline created for a recursive function is not necessarily linear, as the call graph can be a tree, and the heterogeneity of the stages in the pipeline. We show that our method provides an increase in performance when compared to a stack-based solution, due to the parallel execution of operations in the stages of the pipeline.

Another issue critical to good mapping is the limited hardware resources in an FPGA device. Such constraint derives the need to minimize the amount of area used on the device while unrolling the function. Keeping the amount of area configured to a minimum amount allows for greater utilization of area on the device in cases where the

recursive tree may be unbalanced. Naive allocation of stages in the pipeline may cause hardware to be wasted and therefore the maximum potential amount to be unrolled may remain un-realised. Thus stages in the pipeline are allocated on-demand.

On-demand allocation of new pipeline stages may result in stalling of execution due to large runtime reconfiguration delays. Since runtime reconfiguration is a costly process, when compared to computation time, we attempt to reduce the overhead introduced when reconfiguring further depths of recursion by employing prediction strategies to identify the need for further stages of the pipeline in advance.

1.5 Assumptions and Limitations

Our method for mapping recursive functions into a hardware pipeline is limited by the logic and memory resources on the target FPGA device. Such limitations are identified and some solution techniques are addressed in this thesis.

The parallelism that can be extracted from certain types of recursive functions is also limited. In particular, functions with multiple recursive calls with data dependencies between them and recursive functions that do not operate on input data in a sequential order suffer from reduced performance as buffering the input becomes a necessity.

Finally there is limited support for the mapping of recursive data structures to reconfigurable hardware. A specific case study presented in this thesis for quad-tree partitioning for force approximation does feature recursive data structures, however the function being mapped only considers a single function building the tree and doing operations on it. A data structure such as binary search, which features multiple opertions is not completely supported by the techniques presented in this thesis, as it involves multiple functions operating on the same data structure. While this algorithm is not completely supported, the feasbility of extending the techniques in this thesis to this algorithm are documented in a case study. The case study examines the insert and search operations on a binary search tree by incorporating both functions in the same processing units.

1.6 Thesis Organisation

This thesis is organised into four sections. The first section surveys the FPGA technology and current design tools before reviewing previous literature on mapping recursive functions and loops to reconfigurable hardware. The second section details the methods used to implement recursion on runtime reconfigurable hardware. The third section presents several case studies where the methods introduced

in this thesis are applied to several well-known recursive algorithms. Experimental results that illustrate the performance benefits of using our method when compared to stack-based implementations are presented. The final section summarises the thesis and identifies future work. An appendix to the thesis describes the simulation framework used to obtain some results.

1.6.1 Literature Review

Chapter 2: explains basics of the FPGA technology and current design tools. The chapter details FPGAs, and how they operate before presenting the growth in FPGA devices in recent years. We then survey current design tools detailing the differences between the available tools and what they provide to the developer.

Chapter 3: surveys previous literature on loop unrolling and the recursive function unrolling on FPGA devices and other parallel architectures.

1.6.2 Implementation

Chapter 4: details the function analysis that need to be performed for a given a recursive function before hardware can be created.

The analysis described in this chapter describes how a recursive function is partitioned into smaller functions and the optimizations that are made to the function to reduce the amount of buffering required.

Chapter 5: details the method used to create the pipeline on the FPGA. We present methods to reduce the latency introducted while unrolling the pipeline by predicting the need for further stages in the pipeline before they are required as well as techniques for reducing the number of stages in the pipeline while constant throughput is maintained without the need for buffering data or stalling computation in the pipeline.

Chapter 6: describes the hardware model used to implement recursive functions on FPGA devices. This chapter describes the method used to place the pipeline on the FPGA as well as the method used to route between stages of the pipeline before describing the method used to address limitations placed on the unrolling process by the resources of the FPGA device. The memory model that is used is then described. The bottleneck of memory bandwidth is discussed, before techniques to reduce the impact of this bottleneck are presented.

1.6.3 Case Studies

Chapter 7: describes the case studies that were implemented using the techniques described in this thesis. The algorithm is detailed before stepping through each critical part of the mapping techniques. Results are then presented comparing the implementation with a traditional stack-based implementation.

1.6.4 Conclusion

Chapter 8: presents my concluding remarks. I address how the aims I outlined in this introduction are met before presenting a discussion on research questions that this thesis raises for the future as well as future directions of research that stem from this thesis.

1.6.5 Appendix

Appendix chapter A: describes a simulator that was built to measure the performance of the techniques described in this thesis. The simulator was required in situations where available hardware was insufficient for testing due to a lack of support for runtime reconfiguration. The simulator also has applications for modelling of various general FPGA properties.

1.7 Contributions

This thesis makes the following contributions:

Parallelisation of recursive functions on reconfigurable hardware: The main contribution of this thesis is a method to parallelise recursive functions and map them into reconfigurable hardware. Previous research into mapping recursive functions on reconfigurable hardware has not considered exploiting the high level of parallelism available between instances of the recursive function.

Partitioning of recursive computations: A key contribution of this thesis is a method to partition a recursive function and then describe it as a combination of these smaller partitions.

Previous work into the parallelisation of recursive functions has maintained each instance of the recursive function as an atomic unit, and creating separate processes for each of the function calls made.

"Flattening" of the recursive call graph: Another key contribution of this thesis is the analysis of the runtime of the recursive function, and the ratio of time spent between a recursive function and its recursive calls to itself. This analysis is done to reduce

the size of the call graph by introducing some basic load balancing, which allows a larger call graph and therefore a deeper recursive call to be mapped to the hardware.

Prediction of behaviour: Another key contribution of this thesis is the prediction of the behaviour of the recursive function to reduce the latency introduced while unrolling the function before further stages are required. Previous research into unrolling of pipelines on FPGA hardware has relied on hardware re-use to disguise the latency of the unrolling procedure.

Unrolling of unbound recursion on finite FPGA resources: Another key contribution of this thesis is the unrolling of non-linear pipelines that require more resources than available on FPGAs. Previous research into unrolling of pipelines on FPGA hardware has only considered linear pipelines.

Simulation of runtime reconfiguration: This thesis presents a simulation framework which allows for parameterisable reconfiguration models. Previous simulation frameworks have been tied to specific architectures that limits the ability of researchers to experiment with various reconfiguration architectures and designs.

Chapter 1

Introduction

Recursion is an effective design tool that is used to describe many important algorithms and data structures. As the adoption of higher-level language to FPGA design tools increases, the lack of support for recursive functions and the difficulty in supporting such functions without a stack is becoming glaringly obvious.

In this chapter I present an abridged description of the background that motivates the inclusion of recursive functions in higher-level design tools, describe the methodology I adopt to automate the mapping of such functions into hardware suitable for FPGA implementation, and then outline limitations of my methodology. Finally I outline the remainder of the thesis and highlight my original contributions.

1.1 Introduction

Reconfigurable computing is a method of computing where general purpose hardware is configured for a specific task, but may be reconfigured for a different task at a later date.

An example of a reconfigurable computing device is FPGA, which consists of a set of logic cells that can be programmed to form any logic function provided that sufficient placement and routing resources are available on the device.

A comparison of an FPGA-based solutions to an ASIC designs shows that the FPGA based solution has a better cost of development and a lower time to develop, but however does not perform as well as an ASIC design. A comparison of an FPGA-based solution to software based solution shows that the FPGA solution may take longer to develop but does provide better performance results. From these observations it can be seen that when considering the metrics of performance and development cost and time an FPGA based solution resides between ASIC designs and software solutions.

Traditionally, designing for FPGA-based systems has involved the implementation of hardware designs in hardware design languages such as VHDL, or Verilog. While such languages are appropriate for hardware designers and their designs, attempts to expand the use of FPGA technology among software developers, who may lack hard-

ware design background, favoured a move towards familiar software design languages [1, 3, 35, 48, 20], such as C and Java.

These tools are typically designed to mirror the interface of common software IDE packages. The tools mirror the interfaces in terms of "look-and-feel" and provide syntax that is very similar to the software tools. By providing this syntax the tools hide the majority of the hardware details from the developer with the objective of providing a shorter development time for complex designs without the need of extensive hardware design skills. However such approach has the potential to produce designs with lower performance and larger use of FPGA resources. In addition, some of the high-level language tools fall short of their objectives as they force the developer to think of cycle-by-cycle operations during the development of their application and, of particular relevance to this thesis, only support a subset of the language.

1.2 Problem Definition

This thesis addresses the particular problem of support for recursive functions in high-level design tools. Maruyama et al [48] have developed the only design tool to date that supports recursive functions. It supports recursive functions by builiding a memory stack to store

intermediary data.

The use of a stack greatly reduces the amount of parallelism that can be extracted from the recursive call as it reduces the ability of the system to run multiple instances of the function in parallel. Recursion is typically implemented on microprocessor systems with a stack, but with an FPGA-based system that provides the ability for massive amounts of parallelism, a stack-based solution clearly falls short of realising its performance potential. This thesis puts forward a method to automatically generate circuits for recursive functions that will exploit the parallelism available in the entire recursive call.

1.3 Research justification

Recursion is commonly used in many high performance algorithms, with all algorithms belonging to the "divide-and-conquer" class being defined recursively. Such functions can be transformed into loops [46, 7, 6]. However functions that must store variables defined prior to the execution of a recursive call require the use of a memory stack. Such functions cannot be mapped into hardware using regular loop unrolling techniques due to the stack maintenance acting as a loop carried dependency.

Performance issues aside, the lack of support for recursion, even

stack based recursion, forces the developer to either abandon the use of recursion or to implement a stack design of their own in cases where it is necessary. Such situation is not compatible with the prime purpose of these tools, which is to allow the developer to work in a familiar environment. Thus automated support for recursive functions should be an integral part of any high-level design tool.

1.4 Methodology overview

For the context of the research presented in this thesis the execution of a recursive function is modelled as a tree. This tree matches the call graph of the recursive function as it executes. A call graph is a graph representation of the function calls made during the execution of a program. In the case of a call graph for a recursive function, each vertex of the graph corresponds to an instance of the function, and an edge exists from each function to every instance that it instantiates via a recursive call. The call graph obtained for a recursive function is named the recursive tree throughout this dissertation.

Two types of inter-procedural parallelism are present between recursive calls. The first stems from parallel execution of recursive calls made by a function, which exploits the parallelism available between nodes of the recursive tree on the same level. The second type intro-

duces parallelism by having nodes operate on partial data, thus having nodes in different levels operating in parallel. This parallelism is present in functions that operate on streams of data and output data before processing the entire stream.

The research presented in this thesis describes a method that exploits this parallelism by creating a pipeline whose stages contain instances of the recursive function that implement function in-lining on recursive calls. The pipeline matches the shape of the call graph created by the recursive function. Similar techniques have been previously used to map loops to reconfigurable hardware [17, 30, 37] where each stage of the pipeline corresponds to an unrolled instance of the original statements. The differences between the two methods are that the pipeline created for a recursive function is not necessarily linear, as the call graph can be a tree, and the heterogeneity of the stages in the pipeline. We show that our method provides an increase in performance when compared to a stack-based solution, due to the parallel execution of operations in the stages of the pipeline.

Another issue critical to good mapping is the limited hardware resources in an FPGA device. Such constraint derives the need to minimize the amount of area used on the device while unrolling the function. Keeping the amount of area configured to a minimum amount allows for greater utilization of area on the device in cases where the

recursive tree may be unbalanced. Naive allocation of stages in the pipeline may cause hardware to be wasted and therefore the maximum potential amount to be unrolled may remain un-realised. Thus stages in the pipeline are allocated on-demand.

On-demand allocation of new pipeline stages may result in stalling of execution due to large runtime reconfiguration delays. Since runtime reconfiguration is a costly process, when compared to computation time, we attempt to reduce the overhead introduced when reconfiguring further depths of recursion by employing prediction strategies to identify the need for further stages of the pipeline in advance.

1.5 Assumptions and Limitations

Our method for mapping recursive functions into a hardware pipeline is limited by the logic and memory resources on the target FPGA device. Such limitations are identified and some solution techniques are addressed in this thesis.

The parallelism that can be extracted from certain types of recursive functions is also limited. In particular, functions with multiple recursive calls with data dependencies between them and recursive functions that do not operate on input data in a sequential order suffer from reduced performance as buffering the input becomes a necessity.

Finally there is limited support for the mapping of recursive data structures to reconfigurable hardware. A specific case study presented in this thesis for quad-tree partitioning for force approximation does feature recursive data structures, however the function being mapped only considers a single function building the tree and doing operations on it. A data structure such as binary search, which features multiple opertions is not completely supported by the techniques presented in this thesis, as it involves multiple functions operating on the same data structure. While this algorithm is not completely supported, the feasbility of extending the techniques in this thesis to this algorithm are documented in a case study. The case study examines the insert and search operations on a binary search tree by incorporating both functions in the same processing units.

1.6 Thesis Organisation

This thesis is organised into four sections. The first section surveys the FPGA technology and current design tools before reviewing previous literature on mapping recursive functions and loops to reconfigurable hardware. The second section details the methods used to implement recursion on runtime reconfigurable hardware. The third section presents several case studies where the methods introduced

in this thesis are applied to several well-known recursive algorithms. Experimental results that illustrate the performance benefits of using our method when compared to stack-based implementations are presented. The final section summarises the thesis and identifies future work. An appendix to the thesis describes the simulation framework used to obtain some results.

1.6.1 Literature Review

Chapter 2: explains basics of the FPGA technology and current design tools. The chapter details FPGAs, and how they operate before presenting the growth in FPGA devices in recent years. We then survey current design tools detailing the differences between the available tools and what they provide to the developer.

Chapter 3: surveys previous literature on loop unrolling and the recursive function unrolling on FPGA devices and other parallel architectures.

1.6.2 Implementation

Chapter 4: details the function analysis that need to be performed for a given a recursive function before hardware can be created.

The analysis described in this chapter describes how a recursive function is partitioned into smaller functions and the optimizations that are made to the function to reduce the amount of buffering required.

Chapter 5: details the method used to create the pipeline on the FPGA. We present methods to reduce the latency introducted while unrolling the pipeline by predicting the need for further stages in the pipeline before they are required as well as techniques for reducing the number of stages in the pipeline while constant throughput is maintained without the need for buffering data or stalling computation in the pipeline.

Chapter 6: describes the hardware model used to implement recursive functions on FPGA devices. This chapter describes the method used to place the pipeline on the FPGA as well as the method used to route between stages of the pipeline before describing the method used to address limitations placed on the unrolling process by the resources of the FPGA device. The memory model that is used is then described. The bottleneck of memory bandwidth is discussed, before techniques to reduce the impact of this bottleneck are presented.

1.6.3 Case Studies

Chapter 7: describes the case studies that were implemented using the techniques described in this thesis. The algorithm is detailed before stepping through each critical part of the mapping techniques. Results are then presented comparing the implementation with a traditional stack-based implementation.

1.6.4 Conclusion

Chapter 8: presents my concluding remarks. I address how the aims I outlined in this introduction are met before presenting a discussion on research questions that this thesis raises for the future as well as future directions of research that stem from this thesis.

1.6.5 Appendix

Appendix chapter A: describes a simulator that was built to measure the performance of the techniques described in this thesis. The simulator was required in situations where available hardware was insufficient for testing due to a lack of support for runtime reconfiguration. The simulator also has applications for modelling of various general FPGA properties.

1.7 Contributions

This thesis makes the following contributions:

Parallelisation of recursive functions on reconfigurable hardware: The main contribution of this thesis is a method to parallelise recursive functions and map them into reconfigurable hardware. Previous research into mapping recursive functions on reconfigurable hardware has not considered exploiting the high level of parallelism available between instances of the recursive function.

Partitioning of recursive computations: A key contribution of this thesis is a method to partition a recursive function and then describe it as a combination of these smaller partitions.

Previous work into the parallelisation of recursive functions has maintained each instance of the recursive function as an atomic unit, and creating separate processes for each of the function calls made.

"Flattening" of the recursive call graph: Another key contribution of this thesis is the analysis of the runtime of the recursive function, and the ratio of time spent between a recursive function and its recursive calls to itself. This analysis is done to reduce

the size of the call graph by introducing some basic load balancing, which allows a larger call graph and therefore a deeper recursive call to be mapped to the hardware.

Prediction of behaviour: Another key contribution of this thesis is the prediction of the behaviour of the recursive function to reduce the latency introduced while unrolling the function before further stages are required. Previous research into unrolling of pipelines on FPGA hardware has relied on hardware re-use to disguise the latency of the unrolling procedure.

Unrolling of unbound recursion on finite FPGA resources: Another key contribution of this thesis is the unrolling of non-linear pipelines that require more resources than available on FPGAs. Previous research into unrolling of pipelines on FPGA hardware has only considered linear pipelines.

Simulation of runtime reconfiguration: This thesis presents a simulation framework which allows for parameterisable reconfiguration models. Previous simulation frameworks have been tied to specific architectures that limits the ability of researchers to experiment with various reconfiguration architectures and designs.

Chapter 2

Background

This chapter surveys reconfigurable hardware, focusing on **Field Programmable Gate Arrays**(FPGAs). FPGA technology is detailed in section 2.2, before presenting information on the recent evolution of FPGA technology in section 2.4.

Design methods are then surveyed in section 2.5, with a focus on **Higher Level Language** (HLL) design flows and **Runtime Reconfiguration** (RTR) design methodologies.

2.1 Reconfigurable Computing

There are two paradigms that dominate the design and execution of algorithms. The first utilises fixed hardware designs in the form of **Application Specific Integrated Circuits**(ASICs). ASICs are designed

to maximize the performance of a single algorithm, thus producing high performance solutions for only a single problem. ASIC devices however suffer from being immutable once fabricated, thus a change in the application requires ASIC manufacturers to repeat the expensive design and fabrication processes to produce a new ASIC device. The immutability of ASIC design also inhibits the updating of deployed devices, and necessitates expensive product recalls if faults are found in the device.

The second paradigm utilizes a general-purpose microprocessor. A general-purpose microprocessor can execute a set of basic instructions that implement boolean, arithmetic and I/O functionalities. A software program, which is a sequence of these instructions, is loaded onto the microprocessor. By executing this dynamic sequence of instructions a large functional domain can be obtained without changes being made to the hardware. A microprocessor requires many control signals and overheads, such as instruction fetching and decoding which reduces the performance of an algorithm implemented on a microprocessor when compared to an ASIC implementation, to support this high degree of functionality.

Reconfigurable computing attempts to reduce the flexibility vs. performance trade-off by providing the programmability of a microprocessor, while approaching the performance of an ASIC solution.

A reconfigurable device, which consists of a sea of configurable logic resources connected by a configurable network, allows the developer to partition an algorithm into blocks to be mapped into the logic resources and to use the configurable network to connect the blocks thus mapping the algorithm into a hardware circuit. This approach combines the flexibility of programmability a software solution provides without the overhead of instruction fetching and decoding thus maintaining a higher degree of performance.

The granularity of reconfiguration varies between various reconfigurable devices. Fine-grained reconfigurable architectures contain **Lookup Tables** (LUTs) and multiplexers as the unit of configuration. Coarse-grained reconfigurable architectures typically contain a series of **Digital Signal Processors** (DSPs) or processing elements. These differing granularities of configuration allow reconfigurable computing to be applied to various domains, which span regular logic synthesis for fine-grained architectures to signal processing applications for coarse-grained architectures.

2.2 FPGAs

Field Programmable Gate Arrays(FPGAs) are a particular type of fine-grained reconfigurable device. An FPGA device contains con-

figurable logic blocks, connected by a reconfigurable network. Given enough logic blocks, an FPGA can implement any function. Configurable logic on an FPGA is typically implemented with a four input LUTs as shown in figure 2.1. A four input LUT stores sixteen values, with a single bit value stored for each permutation of the input lines. Thus any four input function can be implemented with a LUT.

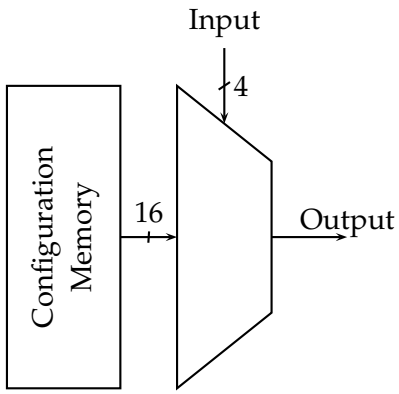


Figure 2.1: Typical lookup table

Logic resources on an FPGA are arranged in collections named tiles. A tile is a fixed set of logic resources that is replicated across the FPGA. There exist various types of tiles, which contain logic for various operations such as programmable logic, I/O and memory access. The majority of the tiles on an FPGA contain programmable logic, which typically consists of a number of LUTs, some flip-flops or latches, several multiplexers and some fixed logic gates. An example of such a tile on Xilinx's Virtex architecture is shown in figure 2.2.

Each tile contains a configurable set of switches. These switches,

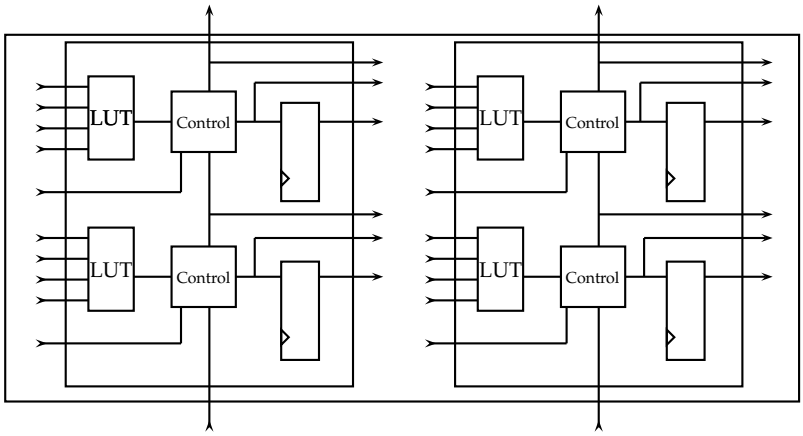


Figure 2.2: Logic resources in a Virtex tile

named **Programmable Interconnection Points**(PIPs), act as junctions between wires connected to the logic resources in the tile and to wires that connect the tiles in the FPGA. By configuring the state of these switches routing can be configured.

Wires that connect resources between tiles are thought of as static routing. There are wires of various lengths on most FPGA devices that span a different number of tiles, providing rapid communication between neighbouring tiles, or tiles that reside on opposite sides of the FPGA. For example Virtex-II routing contains multiple wires that span the entire length and height of the column, wires that span 6 columns, wires than span 2 columns and wires that connect adjacent tiles together.

Device	LUTs	Configuration Bits
XC2V40	512	338,976
XC2V80	1024	598,816
XC2V250	3072	1,593,632
XC2V500	6144	2,560,544
XC2V1000	10240	4,082,592
XC2V1500	15360	5,170,208
XC2V2000	21504	6,812,960
XC2V3000	28672	10,494,368

Table 2.1: LUTs vs Configuration Bits

2.3 FPGA Configuration

A stream of bits that is loaded into the FPGA via I/O pins sets the state of configurable resources on an FPGA device. This stream is typically created on the host computer used to develop the FPGA design and is stored in a file named a bitstream file. Configuration does not necessarily have to be done using I/O pins as some FPGAs have internal configuration ports that allow circuits on the FPGA to configure the device.

The data in a bitstream file describes the configuration state of all configurable logic resources on an FPGA, as well as the state of all routing resources. As a result large FPGAs require larger bitstream files and require a longer time to configure. Table 2.1, shows the number of LUTs in several Xilinx Virtex-II FPGAs and the number of configuration bits required to configure the devices.

The configuration ports that are used to distribute configuration

data to the FPGAs resources run at low speeds. The ports also typically have a small word size that allows only a small amount of data to be loaded in a single clock cycle. The low speed of the port and low amount of parallelism available due to the low width port causes configuration time to be high. For example, a Xilinx XC2V1000 contains a 33MHz configuration port that is 8 bits wide. This sets the theoretical minimum time taken to configure a XC2V1000 to 15msec. Experimental results show that the time taken is longer than this.

For the remainder of the discussion on configuration Xilinx Virtex devices will be used as a reference for two reasons: the author is more familiar with Xilinx configuration architectures than the configuration architectures of devices from any other vendor; and Xilinx devices are the only FPGA devices that support RTR which is critical to the methodology presented in this thesis.

A configuration bitstream contains a series of configuration atomic units named frames. A frame is a fixed amount of configuration data. In Virtex architectures a frame can be thought of as the configuration data for a column with a width of one bit as shown figure 2.3. This leads to situations where a frame in a column containing LUTs will hold one bit of configuration data for every LUT in that column. As a result the configuration data for a LUT is stored in multiple configuration frames, and therefore the configuration of a LUT may re-

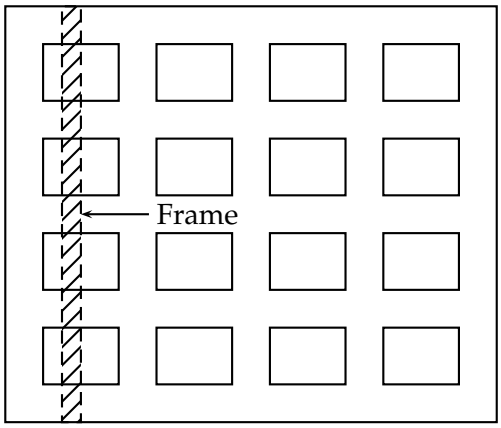


Figure 2.3: Relationship between frames and hardware

quire multiple frames to be loaded. Later Xilinx architectures such as the Virtex-II use frames that span columns of width greater than 1, which allows a LUT to be configured by a single frame. The frame size for Xilinx Virtex devices is dependent on the number of rows in the FPGA, which results in larger devices having a larger unit of configuration.

2.3.1 Partial Reconfiguration

Partial reconfiguration is a reconfiguration mechanism where selected segments of the FPGA can be configured leaving the remainder unchanged. This is desirable in situations where only small changes to a design are required, such as changing keys in encryption applications [54], implementing high speed crossbar switches [63], or in ap-

plications that require certain areas of the device to remain unchanged such as runtime task management [52, 45, 47, 41].

When an FPGA is partially configured the bitstream only contains a fraction of the data required for a complete reconfiguration, which reduces the size of the bitstream and therefore the time taken to configure the FPGA. However it is important to note that configuring specific segments of an FPGA using partial configuration may require a larger bitstream than is needed. For example when changing the state of a single LUT or pip on the device, the frames that the changed bits belong too must be reloaded in their entirety. Therefore to configure only a single bit on a Virtex device, many bits of data must be loaded.

2.3.2 Runtime Reconfiguration

Run-time Reconfiguration (RTR) is a mechanism that allows partial configuration of an FPGA while other circuits on the FPGA are running. RTR allows applications to be developed where circuits on the FPGA configure the FPGA through configuration mechanisms such as **Internal Configuration Access Port**(ICAP). The bitstreams that are loaded must be generated on an external host and stored in memory. This is due to the format of the bitstream on commercial FPGAs being closed which makes it impossible for circuits to create, or even edit configuration data for the device. As such the bitstream for any pos-

sible configurations must already have been generated and stored in memory which may require large amounts of storage.

In cases where a frame loaded during RTR contains a circuit that is to be reconfigured and one whose state is to remain unchanged a feature termed "glitchless configuration" is used. Glitchless configuration guarantees logic or routing in a frame is not changed when it is loaded the actual logic or routing is not configured. Combined with RTR this allows the frame configuration atom to be broken, as static circuits that occupy parts of a frame being reconfigured are not interrupted during the reconfiguration process. It should be noted that the atom is not broken completely as the data must still be loaded.

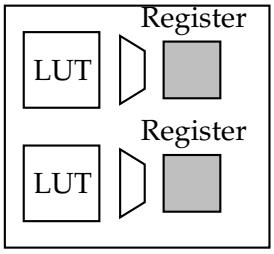


Figure 2.4: Internal Slice Logic

2.4 FPGA Evolution

While FPGA adoption in commercial industry has been on the rise, a major factor limiting its growth has been the amount of logic on the device. However, the size of FPGA devices has grown rapidly

over recent years. As shown in the graph in figure 2.5, the number of LUTs present on top of the range Xilinx devices has grown by approximately 900% over the last three years. These trends signal that any limitations imposed on designs by the limited amount of logic on the device will descrease as the amount of logic that can be synthesized on the device grows over time.

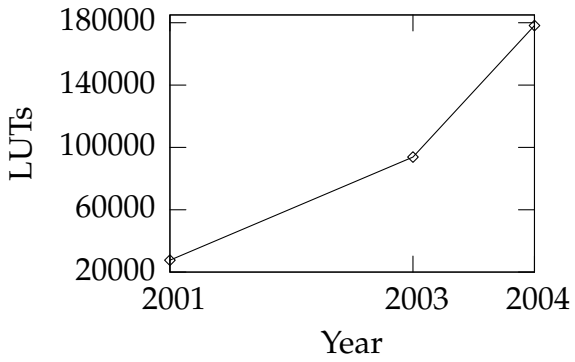


Figure 2.5: Number of LUTs on Xilinx FPGA devices

The number of LUTs available on a device has not been the only item whose growth is of interest. The amount of distributed memory on FPGA devices has also grown rapidly over the last few years since they were introduced increasing a massive 7800% in the last three years. As with the increase in LUTs on the device, these trends signal that any designs limited by the amount of memory on the device will have these limitations decreased as the amount of memory on the device grows over time.

In recent years a shift in commercial FPGAs has seen them trans-

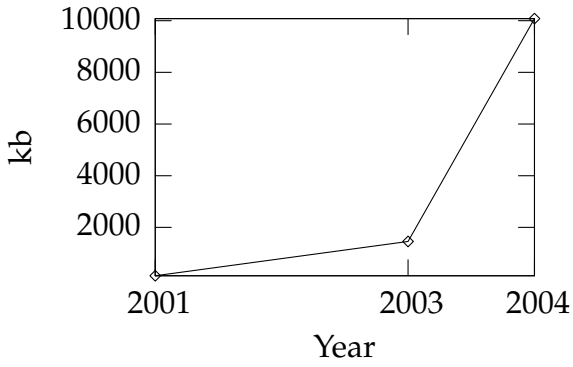


Figure 2.6: Amount of distributed memory on Xilinx FPGA devices

formed from completely fine-grained architectures to partially coarse-grained architectures, with the embedding of multipliers in Xilinx's Virtex-II [24] range of devices, and then the introduction of PowerPC cores on their Virtex-II Pro devices [25]. Later devices by Xilinx [23] now contain dedicated DSP processors.

2.5 Design Tools

2.5.1 HLL Design Tools

Higher level language(HLL) design tools for FPGA synthesis are a well studied area for languages such as C and Java. Available packages aim to reduce the time taken to develop applications when compared to **Hardware Design Language**(HDL) design tools such as Xilinx's foundation series [21], or Altera's Quartus package [18]. They also attempt to minimise the reduction in performance in terms of

clock speed and the amount of resources used when compared to the HDL tool kits.

Regular commercial products, such as Handel-C [1], System- C [3] and Forge [20], produce netlists from C or Java code, which are then used by regular hardware synthesis tools to produce bitstreams to be loaded into the FPGA.

Various research utilities that provide functionality not provided by commercial tools also exist. Utilities such as the Streams-C compiler [35] disguise the low level details of the hardware design from the programmer, and also add interesting constructs to enable streaming which is not supported natively by the standard ANSI C language [5]. The Streams-C compiler does not extract any parallelism from the program leaving it to the programmer to parallelise their code.

The compiler developed by Maruyama et al [48] automatically extracts parallelism from loops and recursive calls. Loops are parallelised by creating a staged pipeline, with each stage corresponding to an iteration of the loop. The process of parallelising recursive calls proceeds in two stages. First recursive calls are transformed into loops that are then parallelized. While this process parallelises the recursive call by having multiple instances in a pipeline at the same time, it is unclear how this method works when the recursive function calls it-

self more than once. If instances are placed in a queue it is possible that this queue grows to a large length, hence causing a large memory bottleneck in the system.

2.5.2 Partial Reconfiguration and RTR Design Flows

There has been a large amount of effort into the development of partial reconfiguration and RTR design flows by the academic community. These design flows generate bitstreams for partial reconfiguration. Most packages are based on Xilinx's JBits package [36], as the format of the bitstream is closed necessitating vendor tools to be used.

As the contents and format of the configuration bitstreams for all commercial FPGAs are closed Xilinx provide a Java API named JBits that directly manipulates the bitstream file. Through a Java program typically running on a host machine, JBits allows a developer to access the reconfiguration data of any configurable element on the FPGA device. This includes configurable logic such as LUTs and multiplexers as well as configurable routing on the FPGA. JBits edits the bitstream directly and as such requires an input bitstream for the changes to be made too. JBits is difficult to use as a design tool. It requires detailed knowledge of the FPGA, requires the developer to think in terms of bits and possibly the biggest difficulty with using JBits is that it requires the developer to place and route their own designs. JBits is not

a complete development tool as it does not provide interfaces for the testing or debugging of a design.

JHDL [12] was developed to tackle some of the JBits deficiencies. It is a HDL language that mirrors the Java language and is capable of modeling partial and runtime reconfiguration. It produces netlists that can be used by traditional FPGA placement and routing tools to create bitstreams. However JHDL does not offer as much functionality as JBits. While it models partial and runtime reconfiguration it does not allow the developer to create or edit bitstreams to use for these types of configuration. An extension to JHDL, JHDLBits [56], attempts to mix the JHDL and JBits functionalities by creating their own placer and router and then creating a bitstream through the use of JBits. This approach retains the ability to create and model high-level designs using JHDL, but also the low-level bitstream manipulation provided by JBits. It is unfortunate that I could not obtain a copy of JHDLBits as it is not available for download as advertised in their published articles.

Other RTR design flows have looked at the issue of runtime task management on self-reconfiguring platforms [13]. The tool created by Blodget et al, named XPART, uses JBits to create partial bitstreams which are then loaded onto the device using RTR. XPART makes it possible to locate a rectangular module in a bitstream and shift it hor-

izontally and vertically creating a new bitstream with a new placement. As with most software related to Xilinx's bitstream it is not possible to get a copy of the software for experimentation.

Chapter 3

Literature Review

This chapter reviews recursion in section 3.1, examining the history of recursion in early programming languages, before reviewing research into recursion to iterative transforms in section 3.1.1 and examining techniques for compiling recursion on parallel architectures in section 3.1.2. What little research that has been done in mapping recursive functions to reconfigurable hardware is discussed in section 3.1.3.

The well studied topic of loop unrolling on reconfigurable computing is then discussed in section 3.2. Loop unrolling is discussed as loop unrolling techniques are similar to the technique described in this thesis for mapping recursion to reconfigurable hardware.

3.1 Recursion

Recursive functions are functions that contain function calls to themselves. Recursion is used to describe algorithms such as "divide-and-conquer" algorithms which recursively split large data sets until the data set is of small enough size so that results can be easily obtained. Dynamic programming [11] also uses recursion and as such recursion is a cornerstone of algorithm development.

Early programming languages such as FORTRAN [8] did not support recursion. This is due to recursion requiring dynamic memory allocation to accomodate for an unknown number of functions being called. As static memory allocation offers increased performance when compared too dynamic memory allocation early computer systems only featured static memory allocation. Without dynamic frame allocation of a function or a procedure in memory only a fixed number of instances of each function can be run as each instance of the function requires a predetermined memory address to be allocated to it at compile time. Maintenance of a stack is also not possible as the stack will be limited in size due to the memory for the stack being statically allocated at compile time. By definition when a recursive function calls itself it will require an unknown number of instances of itself to be allocated, thus without dynamic memory allocation to allocate frames for new functions or procedures and area to store variables

from the calling procedure recursion was not implemented. However eventually due to a desire for increased functionality coupled with the rapid increase in technology, stack based implementations and dynamic frame allocation became common. Thus recursion was implemented on stack based processors by the ALGOL 58 [55] language and was adopted by most other languages, with functional languages such as Haskell providing recursion as the only method of iteration.

3.1.1 Recursive to Iterative Transforms

Transformations from recursive functions to iterative loops have been well studied. Auslander et al [7] propose a method that transforms a recursive call into a loop using a stack to store data as it is required. While they demonstrate improved performance on single processor architectures, whether the method provides any improvements on parallel architectures is unclear. It is doubtful that this method will utilise a parallel architecture without further transformations.

Liu et al [46], propose a method that can possibly eliminate the need for a stack to store arguments by proposing an inverse function. When a recursive function f calls itself, it contains a state that is required after the recursive call A. When the next function executes it is possible that it changes the state to a new state A' using a function g. Their proposal is that if an inverse function g' of g can be derived it is

possible to compute the previous state A from A'. This is not always possible as it is possible that the function g is not bijective. In cases where such a function can be found this technique lends itself to regular loop unrolling techniques as the elimination of the stack ensures that the iterative loops do not contain any loop carried dependencies.

3.1.2 Parallel Recursion

Stack based recursion on uni-processor systems does not attempt to extract inter-procedural parallelism between distinct instances of the function. This has been realised on multiprocessor systems where there has been research into parallel recursion at the compilation level. Powerlist [49], attempts to build control structures that schedule different portions of a recursive function on multiple computing nodes. The system schedules the portions on different computing nodes after considering the load balance on the system at the time.

3.1.3 Recursion on Reconfigurbale Hardware

To this author's knowledge there only exists one higher level language FPGA design tool that supports recursive functions in packages that map from procedural languages to FPGA hardware designed by Maruyama et al [48]. It does this by implementing a memory stack, thus reducing the parallelism available between the recursive calls..

3.2 Loop Unrolling

While the mapping of recursive functions in parallel on reconfigurable hardware is not a well studied problem, similar investigations have been undetaken in previous research. Loop unrolling on FPGAs with the aim that individual iterations of the loop operate in parallel is a well studied problem. Loop unrolling is a similar problem to unrolling recursion functions, in that as discussed previously all recursive functions can be expressed as iterative functions with the aid of a stack. The remainder of this section will present previous research into the mapping of loops onto FPGA hardware and analyse how these techniques may apply to parallel recursion on FPGAs as well as identifying reasons why the approaches used for loop unrolling will not work for recursive functions.

Bondalapti et al [15, 16], as well as Weinhardt et al [62], propose methods to unroll loops that create a linear pipelined array with each stage of the pipeline corresponding to an iteration of the loop, with an example of this being illustrated in figure 3.1. This is similar to the approach to recursive mapping presented in this thesis in that the aim of the approach is to create a pipeline. The approach differs for two reasons: the pipeline array is a linear array whereas the pipeline for a recursive function can be a tree when the recursive function contains multiple recursive call sites; and recursive functions that are not tail

recursive (ie. execute computation on the results of the recursive calls made), require feedback in the pipeline which is not present in loop unrolling.

Similar problems addressed by Bondalapati et al, need to be addressed to unroll recursive functions. Bondalapti et al realise that the finite resources present on a hardware platform constrain the amount of unrolling possible. This is addressed in two ways: forcing stages in the pipeline to operate on multiple iterations of the loop; and the introduction of data context switching [14], which is used for nested loops. Data context switching has a single pipeline compute multiple loops, switching between loops identified by a unique context.

Whereas these methods are appropriate for addressing space issues while unrolling loops, they require extensions to be applicable to recursive mapping. Reuse of stages in the pipeline for a loop can be done as each stage operates on constant sized data. This is not true for recursive functions where data between functions can increase or decrease causing stages to require different times to operate. Further problems exist in grouping nodes in a pipeline that is the shape of a tree, when compared to the problem of grouping nodes in a linear pipeline as the nodes in the pipeline are adjacent, therefore causing no change in shape to the pipeline.

Bondalapati et al, also realise the need to use RTR to place further

stages onto the reconfigurable device as they are required. Methods to pipeline the RTR with computation, and therefore reduce the latency introduced by RTR, are presented that reuse stages in the same way stages are reused to conserve space. The impact of the latency introduced by RTR is significant in the case of unrolling recursion, with recursive functions typically requiring more configuration than a single stage of a loop. The techniques presented by Bondalapti et al, reduce the amount of parallelism available, thus techniques that involve initiating RTR before it is required are investigated and presented in this thesis.

```
SUM-ARRAY(A, n)
1  i ← 0
2  sum ← 0
3  while i < n
       do
4          sum ← sum + A[i]
5          return sum
```

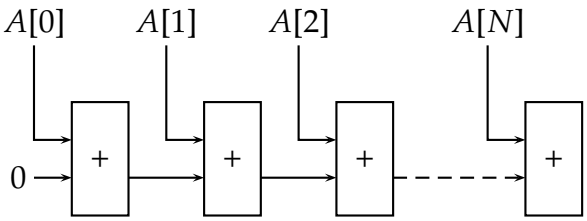


Figure 3.1: An unrolled loop.

Styles et al [59], realise the reduction in parallelism that results from **Loop Carried Dependencies**(LCDs). A LCD is a data depen-

```
sum = 0;
while(i<n)
{
   sum+=a[i];
   i++;
}
```

Figure 3.2: An example of a LCD

dency that occurs between different iterations of a loop. Such a dependency is shown in figure 3.2. The dependency occurs with the value `sum`. No iteration of the loop can execute in parallel as any stage j is dependent on the value of `sum` from every stage k where $0 < k < j$. Styles et al propose a method to parallelise multiple instances of loops with LCDs, using a tagged token system similar to that of the dataflow machine [61]. The problem of LCDs is applicable to recursion unrolling as each stage of the pipeline in recursion is dependent on the previous stages.

Chapter 4

Function Analysis

This chapter begins by discussing function inlining (as defined in definition 4 and why traditional function inlining is not appropriate for the case of recursive functions in section 4.3. A method to partition recursive functions into smaller functions is then presented in section 4.4. This partitioning creates functions that can be inlined using techniques described in this chapter.

Section 4.5, presents two optimisations that are done to the recursive function: ordering the input arguments to reduce the amount of buffering required and fixing the recursive cases to a constant no matter the inputs into the function.

The chapter will now begin with the definition of some terms that will be used throughout this chapter and the rest of this thesis. These terms are generic compiler terms [50] and terms that have been used

to describe properties this thesis examines.

4.1 Chapter Aims

- Introduce relevent compilation terms.
- Introduce inlining and why it is not appropriate for mapping recursion to FPGAs.
- Introduce and justify the method described in this thesis.
- Describe the analysis done on the recursive function that is to be mapped.

4.2 Definition of Terms.

Definition 1. *Basic Block: A maximal sequence of instructions or statements that can only be entered from the first statement and exited from the final statement.*

Definition 2. *Final Block: A basic block that contains statements that terminate a function. An example of this is a block containing a return statement.*

Definition 3. *Flowgraph: A rooted directed graph representation of a function, with a vertex for each basic block in the function and two additional entry and exit blocks, and a set of edges that connect the basic blocks with a*

flow corresponding to that of the function. The entry block contains a single edge to the entry point of the function, and the exit block has an edge to it from every final basic block in the function.

Definition 4. *Inlining: The substitution of code at a function call with the statements or instructions contained in the function being called.*

Definition 5. *Reachability: A vertex $u \in G$ is reachable from $v \in G$ if there exists a path in G from v to u.*

Definition 6. *Succ: In a graph G, the set succ(v) for any vertex $v \in G$, is the set of vertices in G that are reachable from v excluding v.*

$$succ(v) = \{u \in G \mid reachable(v, u) \wedge u \neq v\}$$

Definition 7. *Pred: In a graph G, the set pred(v) for any vertex $v \in G$, is the set of vertices in G which v is reachable from excluding v.*

$$pred(v) = \{u \in G \mid reachable(u, v) \wedge u \neq v\}$$

Definition 8. *Live Variable: A variable is live at a point in a program if there exists a path to the exit block such where its value is used before being redefined.*

Definition 9. *Tail Recursion: A recursive function where no statements occur after a recursive function call, except for a recursive function call.*

Definition 10. *Recursive Condition: The boolean expression with the function's arguments as it's terms that evaluates to true for all inputs that lead to the function calling itself.*

Definition 11. *Terminating Condition: The boolean expression with the function's arguments as it's terms that evaluates to true for all inputs that cause the function to terminate.*

Definition 12. *Recursive Tree: A tree that is used to describe the call graph created by the recursive function as it is executed. A node in the tree corresponds to a call to the recursive function. The recursive tree for the fibonacci function shown in figure 4.1, when the initial function is called with $n = 4$ is shown in figure 4.2.*

```
FIBONACCI(n)
1   if n ≤ 1
       then
2           return 1
    else
3           return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Figure 4.1: The fibonacci algorithm.

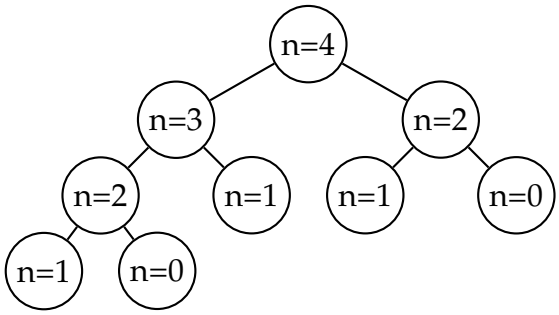


Figure 4.2: Recursive tree for the fibonacci function when $n = 4$

Definition 13. *Recursive Case: The function evaluated when a recursive call is made.*

Definition 14. *Base Case: The function evaluated that returns a value without having made a recursive call.*

Definition 15. *Recursive Depth: The height of the recursive tree.*

Definition 16. *Recursive Level: The depth of a node in the recursive tree.*

Definition 17. *Recursive Degree: The out-degree of any node in the recursive tree.*

An example function and the corresponding flowgraph is shown in figure 4.3. The numbers within the blocks refer to the line numbers of the statements that block contains. The function given is the factorial function $n!$.

An extension is made to the definition of the basic block to aid analysis specific to the process of the recursive function analysis described in this thesis. A basic block containing a statement that contains a recursive function call must only contain that statement. Such a basic block is termed a **recursive block**. An application of this extension to the flowgraph in figure 4.3 is showed in the flowgraph in figure 4.4. The recursive block is hashed so as to distinguish it from the other basic blocks. This method of marking a recursive block is used throughout this document.

```
FACTORIAL(n)
1  if n = 1
       then
2          return 1
   else
3          RecursiveResult ← FACTORIAL(n − 1)
4          return n × RecursiveResult
```

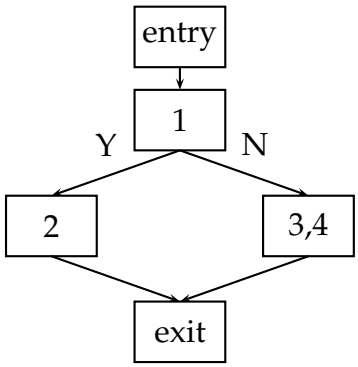


Figure 4.3: The factorial function and its corresponding flowgraph.

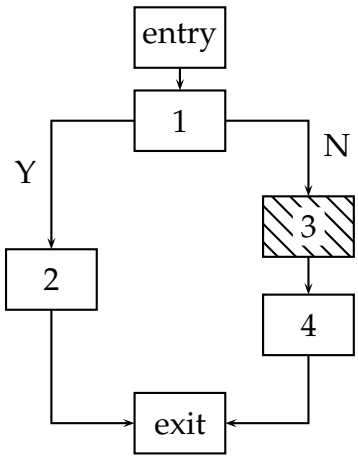


Figure 4.4: Flowgraph for the recursive factorial function showing the recursive basic block

4.3 Inlining Procedure

Function inlining is a process where a call to a function is substituted by the entire code for the function.

By inlining a function, optimizations that may not be able to be applied across multiple functions, such as constant propogation and common sub-expression removal may be applied.

Compilers such as gcc [43] perform function inlining but not for recursive functions. Optimising compilers like LLVM [42] inline a recursive function only a fixed number of times in an attempt to reduce the overhead from the function call. These limitations are due to the inlining process never terminating for a recursive function as each inlined function will contain the statement it replaced. However as a terminating recursive function must have a sequence of statements that causes the function to terminate there must be a way to inline the recursive function such that this sequence is obtained.

A recursive function terminates when the terminating condition of the function is met. In the example function in figure 4.3, the terminating condition is $n = 1$.

Observing this, it can be seen that when a recursive function is inlined, either a set of instructions that lead to termination is inlined, or a set of instructions that lead to a further recursive call is inlined. These two different instruction sets can be thought to belong to two

separate functions: $f_{non\text{-}recursive}$, the function containing the set of instructions that lead to termination, , or $f_{recursive}$, the function containing the set of instructions that leads to a further recursive call.

As the selection of the function which is inlined effects termination, selecting the appropriate function to inline will shape how the recursive function evaluates. An example application of this inlining proceduce for the factorial function is shown in figure 4.5.

The call graph in figure 4.5 contains one function that calls another and then waits for data to be returned. This return of data is shown by the presence of a back edge (labelled 1) in the graph. The mutual data dependency that results from this will cause both stages of the pipeline to stall while waiting for data from its neighbouring nodes. This in effect allows only a single stage of the pipeline to be operating at any one time.

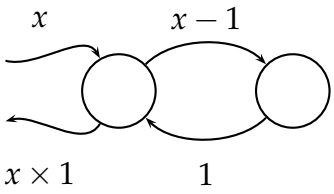


Figure 4.5: Call graph showing a mutual dependency

This is solved by further splitting the function $f_{recursive}$ into two functions: $f_{pre\text{-}recursive}$, which contains all statements prior to any recursive call, and $f_{post\text{-}recursive}$ which contains all the statements that may occur after any recursive call. This allows another stage to be added

to the pipeline that corresponds to the operations in $f_{post\text{-}recursive}$.

Figure 4.6, shows an example of this extra stage being added. The final stage in the graph evaluates the multiplication statement that occur after the recursive call in the factorial program. It receives as input the result from the previous recursive call and the original argument to the recursive call. Arranging the pipeline in this manner eliminates all cycles and therefore any mutual data dependencies between stages of the pipeline. This eliminates the stalling scenario described earlier.

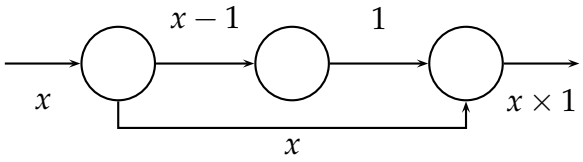


Figure 4.6: Call graph extended to remove a mutual dependency

4.4 Partitioning The Function

To begin partitioning a recursive function f into its sub-functions the flowgraph G for the function f is generated. R is an ordered subset with a cardinality of n that contains all the recursive blocks in G. Using G the sub-functions of f: $f_{recursive}$ and $f_{non\text{-}recursive}$, which correspond to the recursive function and the non-recursive function, are determined. After obtaining $f_{recursive}$, the sub-functions of $f_{recursive}$: $f_{pre\text{-}recursive}$ and $f_{post\text{-}recursive}$, which correspond to the pre-recursive call

function and the post-recursive call function, are then determined.

The method used to determine these functions uses the flowgraph G to obtain the flowgraphs for the functions. In all flowgraphs created new entry and exit blocks must be added to obtain a correct flowgraph, and as such, the entry and exit blocks are not considered for addition to the newly derived graphs. It should also be noted that after obtaining the vertices of the new subgraph G_{sub} all the edges $< v, u >$ in G where both $v \in G_{sub}$ and $u \in G_{sub}$ are added to G_{sub}.

The first definition to be made is the flowgraph G' which is a subgraph of the flowgraph G that has all the recursive blocks removed.

$$G' = \{g \in G \mid g \notin R\} \tag{4.1}$$

The flowgraph $G_{non-recursive}$ for the function $f_{non-recursive}$ is defined as the set of vertices in the graph G' that are both reachable from the *entry* block, and can reach the *exit* block.

$$G_{non-recursive} = \{g \in G' \mid g \in succ(entry) \wedge g \in pred(exit)\} \tag{4.2}$$

The flowgraph $G_{recursive}$ for the function $f_{recursive}$ is defined as the set of vertices in G that can either reach a recursive block, or are reachable from a recursive block.

$$G_{recursive} = \bigcup_{i=1}^{n} \{g \in G \mid g \in succ(R_i) \vee g \in pred(R_i) \vee g = R_i\} \quad (4.3)$$

The flowgraph $G_{pre-recursive}$ for the function $f_{pre-recursive}$ is defined as the set of vertices in $G_{recursive}$ that can reach any recursive block.

$$G_{pre-recursive} = \bigcup_{i=1}^{n} \{g \in G_{recursive} \mid g \in pred(R_i) \vee g = R_i\} \quad (4.4)$$

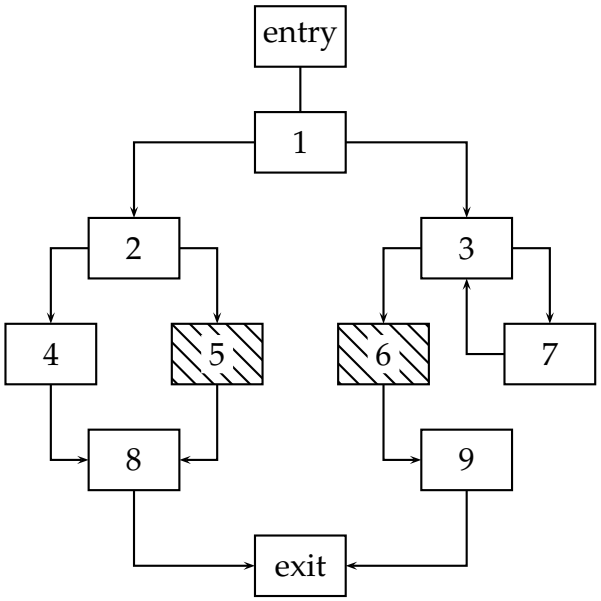
The flowgraph $G_{post-recursive}$ for the function $f_{post-recursive}$ is defined as the set of vertices in $G_{recursive}$ that are reachable from any recursive block.

$$G_{post-recursive} = \bigcup_{i=1}^{n} \{g \in G_{recursive} \mid g \in succ(R_i) \vee g = R_i\} \quad (4.5)$$

Using these definitions it can be shown that the data presented in figure 4.7, for the accompanying flowgraph is true.

Determining the input and output variables for each function unit and the order in which a function unit receives and transmits data is the next necessary step.

The input variables for the *pre-recursive* and *non-recursive* func-



$$
\begin{aligned}
R &= \{5,6\} \\
G' &= \{1,2,3,4,7,8,9\} \\
G_{non-recursive} &= \{1,2,4,8\} \\
G_{recursive} &= \{1,2,3,5,6,7,8,9\} \\
G_{pre-recursive} &= \{1,2,3,5,6,7\} \\
G_{post-recursive} &= \{5,6,8,9\}
\end{aligned}
$$

Figure 4.7: Example flowgraph showing results of partitioning

tions are the input variables for the original function as they correspond to new instances of the recursive function. The input variables for the *post-recursive* function however differ as it corresponds to a function beginning midway through the recursive call and thus the initial state of the *post-recursive* function corresponds to the state of the original function after the recursive call, which is the state of the

pre-recursive function at its exit block. Transmitting the initial function arguments would require large amounts of the *pre-recursive* function to be recomputed to reach the state of the function after the recursive call. To eliminate the need for repeating this computation each *pre-recursive* function unit transmits the live variables at the recursive blocks to its corresponding *post-recursive* function unit. These live variables are computed by finding all the variables that are live on exit from the recursive blocks. This is described in equation (4.6). The *post-recursive* unit also receives arguments that correspond to the results of all recursive calls made by that function.

$$Args_{post\text{-}recursive} = \bigcup_{i=1}^{N} LV_{out}(R_i) \tag{4.6}$$

4.5 Optimisations

This section will present two optimisations that are made to the recursive function being mapped: the reordering of the function arguments to reduce the amount of buffering that the function requires and removing call sites from conditional blocks, which causes the recursive tree to become proper thus simplifying the process of implementing it in hardware.

4.5.1 Input Ordering

The order that input variables are streamed into a function unit can effect the amount of buffering required by the function. To minimise the amount of buffering required all scalar arguments are streamed before arrays. This results in the amount of buffering required being reduced as scalar variables require a constant amount of storage and thus can be stored in LUTs while an array of unknown length requires a memory module, and possibly off-chip memory for storage.

The effect that the order of input variables can have on buffering can be seen in the functions in figures 4.8 and 4.9. Both functions operate on two input arguments A and v, and both read these arguments by calling the READ-INPUT function. Both functions compute the number of elements in the array A that are greater than v and output this result using the WRITE-OUTPUT function.

The function in figure 4.8 receives the array A before the scalar value v. This ordering requires that the function unit buffers the entire array A, of unknown size. This requires at least one memory module to be allocated to the function unit, with the possibility of more being allocated depending on the size of the array.

The function in figure 4.9 receives the array A after the scalar value v. As A is accessed sequentially only one element from it is ever buffered. This constant amount of buffering allows the data to be

buffered in LUTs, conserving the limited on-chip memory resources. This demonstrates the effect reordering can have on the amount of data a function buffers and therefore the amount of memory required by a function.

```
NUMBERGREATER()
 1  value ← READ-INPUT()
 2  while value is not the end of the array
        do
 3          append value to A
 4          value ← READ-INPUT()
 5  v ← READ-INPUT()
 6  greaterThan ← 0
 7  for all a in A
        do
 8          if a > v
              then
 9                  greaterThan ← greaterThan +1
10  WRITE-OUTPUT(greaterThan)
```

Figure 4.8: Algorithm to compute the number of elements greater than v in array A

As a recursive function calls itself, the order in which it defines the variables it uses for the function call must also be considered when the order of the input variables is determined. Figure 4.10, which as with figure **??** replaces the recursive call with the function WRITE-OUTPUT to simulate sending arguments to the next recursive call, contains a recursive function that uses and defines arguments in a way that requires buffering no matter the input argument sequence. The algo-

```
NUMBERGREATERREORDERED()
1  v ← READ-INPUT()
2  greaterThan ← 0
3  value ← READ-INPUT()
4  while value is not the end of the array
       do
5          if value > v
             then
6                greaterThan ← greaterThan +1
7          value ← READ-INPUT()
8  WRITE-OUTPUT(greaterThan)
```

Figure 4.9: Algorithm to compute the number of elements greater than v in array A with reordered inputs

rithm does the same operation as the algorithm NUMBERGREATER, but calls itself again with an array only containing the values greater than v and the value v incremented. The algorithm also takes in another variable n which corresponds to the length of the array. Keeping the previous ordering to reduce the buffering of input, the variable are sent in the order n, v, A, however the definition for the argument n(*greaterThan*) depends on the definition of the argument $A(A_{next})$. This orders the definitions of the variables for the next function call v, A, n, which does not match the order of input. This requires that all the values for A_{next} must be buffered before *greaterThan* is finally defined.

The effect of argument definitions on buffering is determined by

```
NUMBERGREATERRECURSIVE()
 1  n ← READ-INPUT()
 2  v ← READ-INPUT()
 3  A_next ← ∅
 4  greaterThan ← 0
 5  value ← READ-INPUT()
 6  i ← 0
 7  while i < n
        do
 8          if value > v
              then
 9                  greaterThan ← greaterThan +1
10                  append value to A_next
11          i ← i +1
12  WRITE-OUTPUT(v +1)
13  WRITE-OUTPUT(greaterThan)
14  WRITE-OUTPUT(A_next)
```

Figure 4.10: Increment algorithm and the streamed version for the same input order

examining the order of definition for the arguments for the function. If there is no way to order the definitions to be the same as the order of the input variables, and such that the function is still correct then buffering is unavoidable. Arguments that are dependent in this way occur most often in situations such as this where the size of the array is required as an input. This is a result of using languages such as C that do not have constructs that allow the user to determine the length of an array. Languages such as Java have the *.length*() construct which allows the user to determine the length of an array. This is

one argument against the C language for the application of hardware development, however a lengthy discussion about the best procedural language to use for hardware development is not in the scope of this thesis.

The algorithm for ordering the input arguments is shown in figure 4.11. The algorithm ORDERARGS has as an input the set of arguments *Args*. The algorithm first splits the set into two ordered sets *Scalars* and *Aggregrates*, which contain scalar and aggregrate arguments respectively. These sets are then ordered in respect to their first use before the *Aggregrates* set is appended to the *Scalars* set. The ordering function SORTARGS orders the arguments based on their first use in the function. The result of the FIRST-USE(a) is defined as the block in the call graph that contains the first use of the value a.

4.5.2 Constant Number Of Recursive Call Sites

A recursive function can be written in a way such that it can call itself a different number of times depending on the input arguments. This occurs when a conditional statement has one branch that contains a recursive call and one that does not. This produces an improper recursive tree which complicates building hardware to implement the recursive function.

The recursive tree is made proper by introducing recursive calls

```
ORDERARGS(Args)
 1  Aggregrates ← ∅
 2  Scalars ← ∅
 3  for A in Args
        do
 4          if A is scalar
              then
 5                  Scalars ← Scalars ∪ {A}
 6          else
 7                  Aggregrates ← Aggregrates ∪ {A}
 8  SORTARGS(Scalar)
 9  SORTARGS(Aggregrates)
10  return Scalar : Aggregrates

SORTARGS(Args)
 1  SortedArgs ← ∅
 2  i ← 0
 3  while Args ≠ ∅
        do
 4          A_first ← Args[0]
 5          for B in Args
                do
 6                  if FIRST-USE(A_first) ∈ SUCC(FIRST-USE(B))
                      then
 7                          A_first ← B
 8          SortedArgs[i] ← A_first
 9          i ← i +1
10          Args ← Args − {A_first}
```

Figure 4.11: Algorithm to sort input arguments for a function

with *null* arguments into branches with no recursive calls. To do this each recursive block in a conditional branch is moved out of the conditional branch. Each branch of this condition then sets arguments

for the recursive call, either corresponding to the call it would make or *null* arguments if no call is made in that branch. To do this both branches of the condition must be "cut" at the recursive block, or if non exist at the root of the branch. A conditional branch with the same condition is then placed after the new recursive block with its branches being the remainder of the branches that were cut. This process is repeated until no recursive block in the function occurs in a conditional branch.

```
BALANCERECURSIVESITE(rb)
 1  origin ← block at origin conditional branch for rb
 2  end ← block ending the conditional branch for rb
 3  R ← recursive block with arguments newArgs for call
 4  N ← empty block with the same conditional branch as origin
 5  for all branches br from origin
        do
 6          if br contains a recursive block rb
              then
 7                  set rb to the first recursive block in br
 8                  replace call with assignments from args of rb to newArgs
 9                  remove rb from Recursive-Blocks
10                  add child of N to children of rb
11                  set child of rb to R
12          else
13                  add null assignments for newArgs to first block in br
14                  set child of N to end
15                  set child of final block in br to R
16  set child of R to N
```

Figure 4.12: Algorithm to remove a recursive call site from a conditional branch

The algorithm for this is shown in figure 4.12.

An example of the application of this to the function in figure 4.13 will now be shown. The flowgraph in figure 4.14 is the flowgraph for the function with the branch in question being the branch beginning at 1 and ending at 6. The graph that results from application of BALANCERECURSIVESITE to every recursive block not dominating *exit* is shown in figure 4.15.

```
int f(int a, int n)
{
   int g, h;
   a = a + 1;
   if(n%2==1)
   {
      g = f(a, n-1);
   }
   else
   {
      g = 1;
   }
   h = f(a, n-3);
   return g+h;
}
```

Figure 4.13: Function with independant call sites

4.6 Summary

This chapter has presented the analysis done on a recursive function. This analysis was motivated by a discussion on why traditional func-

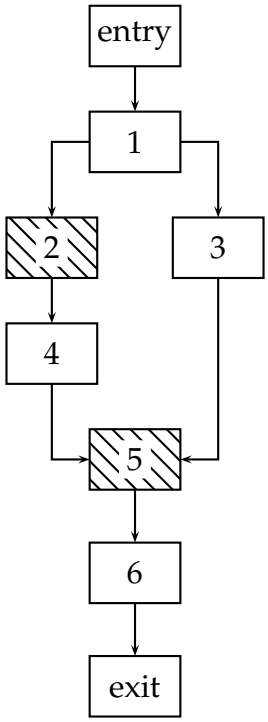


Figure 4.14: Flowgraph for a function with a recursive call site in a conditional branch

tion inlining is not appropriate for unrolling recursive functions.

The analysis described partitions a recursive function into three smaller functions that are used for inlining. The analysis done also includes optimisations that reduce buffering by reordering input arguments and forcing the number of recursive calls made to be constant, making the recursive degree constant for all nodes in the recursive tree.

The important points in this chapter are: the problems that arise when applying regular function inlining techniques to recursive func-

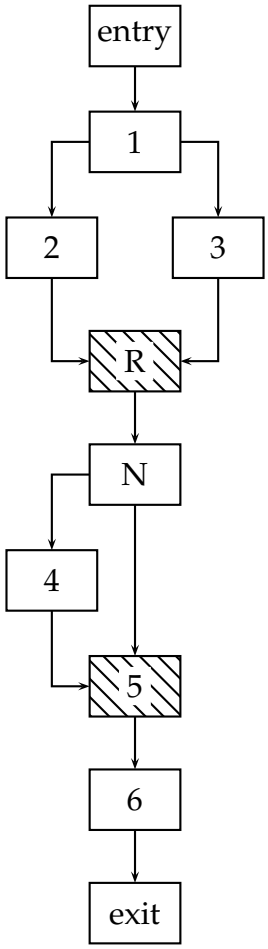


Figure 4.15: Result of removing recursive call site from the conditional branch

tions; the decomposition of a recursive function into smaller functions to allow the unrolling process to terminate and to remove the need to block; the reordering of function arguments and the effects on buffering and the suitability of the C language to the problem of streamed applications based on observations made during experimentation.

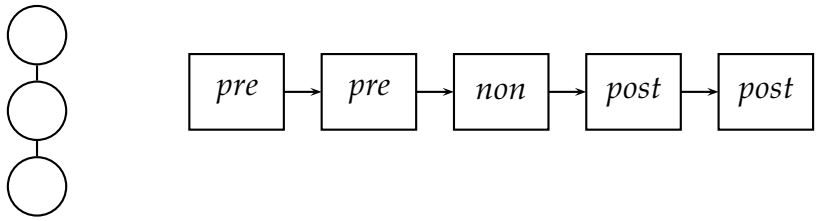
Chapter 5

Drawing the pipeline

Once the analysis of the recursive function has been completed and the *pre-recursive*, *post-recursive* and *non-recursive* functions have been obtained they must be arranged in a pipeline to compute the result of the original recursive function. The pipeline that is created must match the recursive tree of that instance of the recursive function. The challenges that this poses will be discussed in this chapter.

Figure 5.1(a), shows a recursive tree with a recursive depth of three and a width degree of one. As the leaf node of the tree corresponds to a terminating case of the recursive function, the stage in the pipeline corresponding to this node will require a *non-recursive* function unit. In a similar way all internal nodes will require an instance of the *recursive* function which is made of the *pre-recursive* and *post-recursive* functions. The pipeline in figure 5.1(b), is the required pipeline for the

recursive tree in figure 5.1(a). As can be observed the pipeline is heterogenous, with different stages of the pipeline computing different functions.



(a) Recursive Tree (b) Pipeline

Figure 5.1: Recursive tree and the corresponding pipeline

As the recursive tree grows as the function unrolls, the shape of the recursive tree is unknown when computation begins. This requires that the FPGA be configured during runtime. Configuring during runtime requires the pipeline to be dynamically drawn on the FPGA, which requires runtime reconfiguration. Using runtime reconfiguration allows the logic on the FPGA to be configured to match the recursive tree at any given time of the computation.

The use of runtime reconfiguration introduces a latency to the unrolling of the function, as the amount of time required to configure a stage being significanly higher than computation time on the FPGA. This latency is reduced by beginning runtime reconfiguration as early as possible, which allows the reconfiguration to be pipelined with

computation. This is done by predicting the need for runtime reconfiguration before it is required. Techniques to predict this are discussed in section 5.2.

As the recursive tree grows, the number of nodes in the tree grows exponentially. This can result in massive amounts of hardware being required to compute recursive trees of short height. A technique that reuses function units on the device that still maintains constant throughput through the pipeline relative to the first node is discussed in section 5.3.

5.1 Chapter Aims

- Introducing the latency that arises from unrolling the pipeline on-demand.
- The use of heuristics to predict the need to unroll before further stages are required, thus reducing the latency introduced by the unrolling process.
- Analysis of the ratio of work done between nodes in the recursive tree with the aim of reducing the number of nodes in the recursive tree by balancing the work load between nodes.

5.2 Prediction Techniques

Before discussing methods to predict growth of the pipeline some terms that will be used throughout this section will be defined.

Definition 18. *Triggering Argument: The first argument into the system that requires a deeper pipeline than the one currently configured.*

Definition 19. *Overflow: When the final stage in the pipeline detects that it requires further stages to be configured and stalls the system.*

Definition 20. δ_p*: The time taken for a single stage in the pipeline for a recursive function to execute.*

Definition 21. Δ_R*: The time taken to reconfigure a single instance of the recursive pipeline.*

Definition 22. S_D*: The number of stages in the pipeline after the stage that detects the need for reconfiguration, with* $S_D = N$ *at the first stage of a pipeline of length* N*.*

If configuration was initiated only when overflow occured the system must stall while configuration of a new stage occurs. As the configuration of even a modest sized function unit takes a time measured in milliseconds $\Delta_R \gg \delta_p$, which leads to a lot of execution time being wasted while the system stalls during reconfiguration.

If configuration was initiated before overflow occured the performance penalty that occurs while stalling can be reduced by a value of $S_D \cdot \delta_p$. For a sufficiently large S_D the reconfiguration can be completely pipelined with computation thus eliminating latency.

Ideally $S_D = N$ would be the goal for any prediction technique. Prediction where $S_D = N$ is termed optimal prediction and is discussed in section 5.2.1.

Circumstances where $S_D \neq N$ occur when optimally predicting the need for reconfiguration requires too much hardware or involves computing the entire recursive function. Heuristics are empolyed for prediction in this instance. This is termed non-optimal prediction and is discussed in section 5.2.2.

5.2.1 Optimal Prediction

Predicting the need for a new stage of the pipeline involves evaluating the terminating condition on the input arguments. If a triggering argument is found when it enters the pipeline the system has the time for that data item to reach the final stage of the pipeline to configure a new stage. If the stage is still not configured the system may have to stall but the latency has been reduced.

To apply the terminating condition to the input arguments the variables in the condition must be substituted with their definitions

until they are expressed only in terms of the input arguments of the function. The algorithm for this is shown in figure 5.2.

```
REDUCE-EXPRESSION(E, Arg-List)
 1  Used-Expressions ← all expressions used in E
 2  Changed ← FALSE
 3  for UE in Used-Expressions
 4       do
 5          if UE ∉ Arg-List
 6            then Changed ← TRUE
 7                 UE-Def ← DEF(UE)
 8                 Replace UE with UE-Def
 9  if Changed
10    then
11         returnREDUCE-EXPRESSION(E, Arg-List)
12  return E
```

Figure 5.2: Algorithm for reducing expression to arguments

This procedure can produce a value that changes over the evaluation of a single instance of a function. Consider the function in figure 5.3. The function reduces the size of the array in half each time with a base case of size 1. The relationship between the depth of the pipeline D and the input arguments is shown in equation (5.1). Assuming that the size of the input array A was unknown and that it was inputted into the pipeline as a stream, the depth of the pipeline would change over time. It can be shown that at a time t the depth of the pipeline corresponds to equation (5.2), where $A.length(t)$ corresponds to the size of the input stream for the array A at time t. This

value is recorded and the equation computed by keeping a counter on the size of the stream. The triggering argument for this function becomes the argument into the system which causes $D(t)$ to increase.

```
HALF-ARRAY(A)
1  if A.length = 1
2     then return A
3  else
4          i ← 0
5          while i < B.length
6               do B[i] ← A[i] + 1
7                  i ← i +1
8          return HALF-ARRAY(B)
```

Figure 5.3: Algorithm with arguments to be reduced

$$\frac{A.length}{2^D} = 1 \tag{5.1}$$

$$D(t) = \lceil log_2(A.length(t)) \rceil \tag{5.2}$$

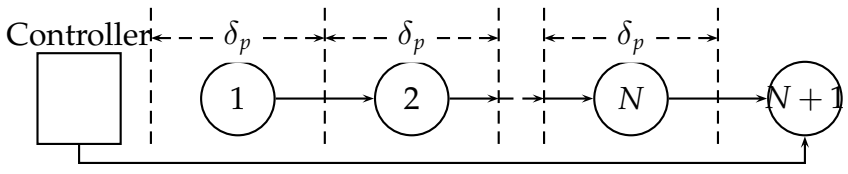


Figure 5.4: Overflowed recursive pipeline reconfigured by controller

Prediction that directly calculates the depth from the terminating condition and the arguments is precise. The function is evaluated by

a controller at the beginning of the pipeline as shown in figure 5.4. This allows the system to predict the possibility of overflow as soon as a triggering argument enters the system, i.e. when $S_D = N$. This reduces the latency from Δ_R to $\Delta_R - N\delta_p$. With a sufficiently large value of N, $\left(N \geq \frac{\Delta_R}{\delta_p}\right)$, this term is reduced to 0.

5.2.2 Non-Optimal Prediction

When the function for predicting the depth of the recursion is too complex to realise in hardware because it would require too much logic to be synthesised or as in some cases involves evaluating the entire recursive function, heuristics are used to approximate the depth of the recursion. The majority of algorithms that exhibit the need for such a heuristic are partitioning algorithms such as quick sort that partition the input data into smaller sets to use as arguments for the recursive calls. In the example of quick sort with an input set of size N, the size of the partitioned sets can range between 1 and N. If the sets are partioning into sizes of $\frac{N}{2}$, the depth of recursion will be $log_2(N)$, however if the sets are partitioned into sizes 1 and N, the depth of recursion will be N.

Two different approaches were considered to predict the depth of the pipeline: an aggressive approach that predicts the maximum possible depth of the pipeline and a conservative approach that predicts

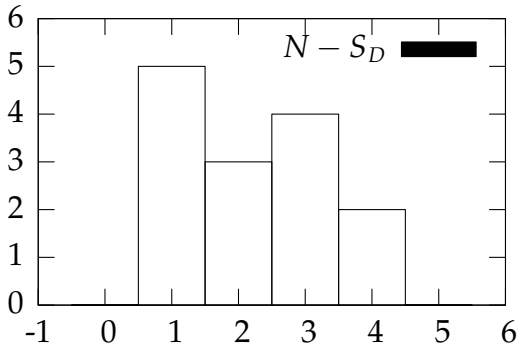


Figure 5.5: S_D during the operation of Quick Sort

the minimum possible depth of the pipeline, given certain inputs. In the case of quick sort the aggressive approach will predict the need for N stages in the pipeline. This however is a poor approximation as analysis shows that the depth of recursion on average should be $O(log(N))$. This poor approximation results in inefficient use of FPGA resources. A conservative approach would allocate the minimum size pipeline, which would be a recursive depth of $O(log(N))$, but it is unlikely that the function call graph will correspond to a properly balanced tree and as such it will need to be reconfigured which introduces latency into the system. However, it is better to introduce latency but realise the maximum possible depth of recursion than to reduce latency at the cost of realising a low depth of recursion.

To improve the accuracy of the conservative approach the approximation is applied at each level of the recursion. The results for this are shown in figure 5.5, which shows how many stages were left in

the pipeline after the stage that predicted the need for a new stage in the pipeline, with 0 being the final stage of the pipeline. This graph shows that the need for reconfiguration was always found before the final stage of recursion, however with a relatively small value for δ_p, the reduction of latency is not that great.

5.3 Reducing Hardware Use

Allocating a function unit to each node of the recursive tree can result in poor utilisation of FPGA resources, as this allocation method does not ensure proper work balancing. This unbalanced work load results from function units at deeper points of recursion most likely working on smaller data sets, thus requiring less time to compute results. In circumstances such as this it is possible to allocate a function unit that controls multiple nodes of the recursive tree, thus reducing the number of function units that are needed to implement the entire recursive tree.

It is desirable that the throughput of each node is similar to that of the root node of the recursive tree, as if a later stage of a pipeline requries more time to compute than previous stages, either previous stages must stall or an infinite sized FIFO buffer must be introduced between stages of the pipeline. As only a finite amount of memory

and logic exists on FPGA devices and stalling is not desirable, to maintain constant throughput each stage in the pipeline should do at most the same amount of work as the first stage in the pipeline. Furthermore maintaining the invariant that all stages do approximately the same amount of work means that it is desirable that no stage in the pipeline stalls while waiting for prior stages to communicate results. If this invariant were not maintained stages of the pipeline would be idle during execution thus not making full use of the logic that has been configured. To maintain this invariant and still maintain constant throughput analysis is required into the amount of work done at each stage of the recursion and the amount of logic, in this case the number of function units, that should be allocated to do this work.

To efficiently allocate resources the ratio of work done between the first level of recursion and every other level of recursion must be determined, from which the ratio of function units between levels of recursion that will produce constant throughput can be determined. This is done recursively by comparing the work done between subsequent levels of recursion.

Consider a recursive function with a complexity of $O(f(N))$. A node in its recursive tree that receives D sized input and has N children where the i^{th} child receives D_i sized input is shown in figure 5.6. The ratio of work done between the root node and its children is

shown in equation (5.3).

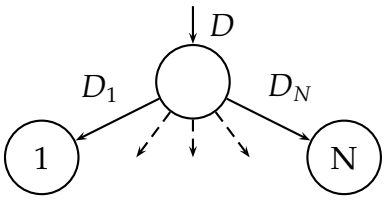


Figure 5.6: Node in recursive tree noting amount of data between nodes

$$\frac{f(D)}{\sum_{i=1}^{N} f(D_i)} \tag{5.3}$$

The result of the ceiling function on this ratio is the number of function units that need to be allocated to compute the result for all the children nodes with the same throughput as the previous node. This will be termed the **recursive growth rate**, as it describes the growth in the number of function units between levels of recursion.

Algorithms with a recursive growth rate of one require only a single function unit to be allocated per level of recursion. Such algorithms match the following four criteria:

1. $f(A) \geq f(B), \forall A \geq B$
2. $D \geq \sum_{i=1}^{N} D_i$
3. $f(0) = 0$
4. $\frac{df(N)}{N}(A) \geq \frac{df(N)}{N}(B), \forall A \geq B$

All algorithms belonging to the *divide-and-conquer* class of algorithms match the first, second and third criteria, and the author of this thesis is yet to find a *divide-and-conquer* algorithm that does not match the fourth criteria.

A reduction in function units allocated per level allows deeper recursive trees to be allocated on an FPGA device, with algorithms such as merge-sort that contain N nodes in its recursive tree requiring only $log_2(N)$ function units. Using this method also addresses a large problem in mapping recursive functions in hardware, where the height of most recursive trees is small compared to the width as the width grows exponentially in respect to the height.

5.3.1 Context Switching

Allocating a single function unit to control multiple nodes in the recursive tree forces the function unit to be executing multiple instances of the recursive function in "parallel". This parallelism is the same parallelism found in conventional uni-processors systems that switch between processes to simulate multi-tasking. Uni-processor systems implement *context switching* to enable this operation. A context switch in a microprocessor flushes the state of all CPU registers to memory and then loads the state or context of another process that is ready to execute. A similar process is implemented in the recursive func-

tion units. When receiving data for an instance of the recursive function that differs from the current instance being computed the state is flushed to memory and the state of the new instance is loaded before execution continues.

Context is controlled in a method similar to that used by the tagged dataflow architecture [61]. The tagged dataflow architecture contains function units that operate on a set of arguments. A function unit will receive arguments for many instances of the function, with a unique tag describing the instance of the function the argument belongs too. Upon receiving a data token, the function unit checks memory to see if it has the rest of the arguments for that instance of the function. If it does it retrieves the arguments from memory and executes the function, otherwise it stores the new data token in memory.

The method adopted in this thesis differs from the dataflow architecture as processing begins as soon as an argument enters the system. Upon receiving an argument the function evaluates all expressions that it can using this argument and the results of expressions from previous arguments, before storing the results of these expressions that are to be used later in the function in memory.

Context Tag

To enable context switching a context tag is sent with each data item by a communicating function unit. The tag is an integer that exists in the range $[0, \infty)$, with the initial data into the first stage of recursion having a tag of 0. Every subsequent function unit implements the bijective function in equation (5.4) to generate new tags. A bijective function is necessary in the same way the function return addess is necessary in a regular stack based solution to recursion, in that from the tag the *post-recursive* function can identify what instance of the function it should output data to and create the appropriate tag. A *pre-recursive* function unit for a recursive function that contains R recursive sites, outputs data with tags $(Rt, Rt+1, \cdots, Rt+R-1)$ for input data with a tag t; the corresponding *post-recursive* function unit will output data with a tag t, for all input data with tags in the range of $[Rt, Rt+R-1]$.

$$
\begin{aligned}
&f(tag, r) = (tag \times t) + r \\
&\text{where} \\
&tag: \text{ the inputted tag} \\
&r: \text{ the index of the recursive call site in the function} \\
&t: \text{ the number of recursive call sites in the function}
\end{aligned}
\tag{5.4}
$$

5.3.2 Unbalanced Recursive Trees

Function units controlling nodes in the recursive tree from different branches may result in the function unit being allocated to nodes that do not exist. For the remainder of this discussion the case of a single unit being allocated per recursive level will be used as it is the easiest to conceptualise.

The pipeline for the recursive function must be unrolled to the height of the recursive tree; however, an unbalanced recursion tree will result in branches having different depths. If the data were to be passed through the pipeline for a branch of less height than the entire tree the data will have the *pre-recursive* function operated on it too many times. An example of this is shown in figure 5.7, where node 3 requries a *non-recursive* function unit at the second stage of the pipeline but there is a *pre-recursive* function unit configured in the second stage of the pipeline. Therefore a mechanism is required that allows the data to be passed through unnecessary levels of recursion until it reaches the *non-recursive* function unit.

Skip Tag

The mechanism to pass data along results in another tag named *skip tag* being added to data being transmitted, which is initially set to 0. Each *pre-recursive* function unit checks if the terminating condition

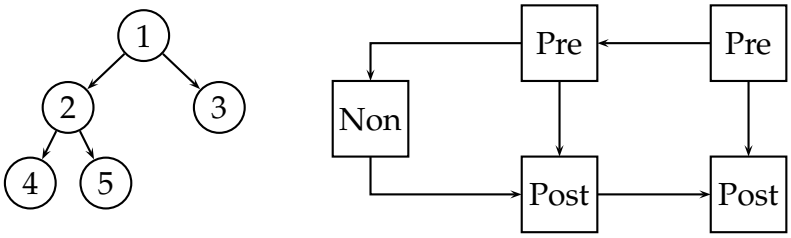


Figure 5.7: Example of a branch of depth 2 executing on a pipeline of depth 4

has been met for any input arguments. If the terminating condition is met this tag is set to 1 and the arguments are sent out. Any subsequent *pre-recursive* function unit that receives a tag ≥ 1 increments the tag and passes the data along. The *non-recursive* function unit operates irrespective of the value of the tag, however the *post-recursive* function units only operate on data where the tag is set to 0, decrementing and passing along the data for any tags ≥ 1. This results in the correct depth pipeline being simulated for each branch of recursion. An example of this operation for a branch of depth two operating on a pipeline of depth four is shown in figure 5.8, where the value of the skip tag for the arguments for node 3 are shown. As can be seen the value is set to 0 only at one *pre-recursive* and one *post-recursive* function unit, thus calculating the correct function.

This procedure is also used to simplify circumstances where recursive functions do not result in proper recursive trees. This occurs

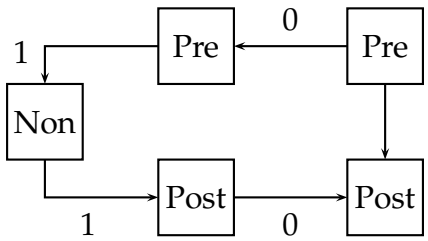


Figure 5.8: Example of a branch of depth 2 executing on a pipeline of depth 4

in functions which may call themselves anywhere between 0 and N times instead of only 0 or N times. Null arguments are passed through the pipeline in the event of a recursive call not being made with the skip tag being set to 1. This allows the *post-recursive* function unit to determine when it has received all the results from the recursive calls as there will always be N results for it to gather.

5.4 Summary

This chapter presents the methods used to create the pipeline using the function modules obtained in chapter 4, which included methods to reduce the performance penalty incurred while unrolling the pipeline as well as methods to decrease the size of the pipeline without reducing performance.

The important points made in this chapter are: the use of prediction heuristics used to predict the need to unroll the function before

the new stage is required, thereby reducing the latency introduced while unrolling the function; the relationship between the number of nodes in the recursive tree a function unit can control, the amount of work it does and the amount of data; the possibility of reducing the growth of hardware used by a recursive function from exponential to linear and the use of a tagging system to facilitate context information and the ability to realise unbalanced recursive trees on linear pipelines.

Chapter 6

Hardware Implementation

Once RTL descriptions for the different function units have been generated, areas on the FPGA device must be configured to compute their operations with interconnections made between the function units that require the ability to communicate with each other.

Placing a static pipeline on the FPGA to achieve optimal logic use can be reduced to the 2-d bin packing problem, which has been shown to be NP-complete [34], however techniques that perform well on small problems [44], and approximations to optimal placement are possible for larger problems [28, 40]. These techniques cannot be applied to the problem described in this thesis as the pipeline changes over time.

A technique that places function units onto the FPGA device as they are needed must attempt to place the function units such that

routing to future function units that may be placed is possible with a minimum number of wires used and a minimal wire length. Specific research into 2-d on runtime reconfigurable FPGA hardware [60], has resulted in the use of heuristics that attempt to reduce the fragmentation introduced but this research has not dealt with tasks that are dependent on communication. This problem is addressed in this thesis by examining the placement problem from the point of view of the network on the device in section 6.3. Techniques to simplify the problem, such as partitioning of the area on the device into columns and grouping function units together are discussed before this in section 6.2

While unrolling a homogenous pipeline further stages may be appended to the pipeline with no requirement that already configured stages must be changed [17]. This property is not true for heterogenous pipelines such as the pipeline created while unrolling recursive functions. The pipeline created by the unrolling process requires that a $non\text{-}recursive$ function unit must be at a position no less than the recursive depth of the branch of recursion it is controlling. A function that requires unrolling from a depth of d to a depth of $d+1$, now requires the $pre\text{-}recursive$ operation to be done at the stage corresponding to the node at depth d. This requires the $non\text{-}recursive$ function unit that was configured to control the stage for depth d to be recon-

figured to perform the operations of a *pre-recursive* function unit. The use of Runtime Reconfiguration(RTR) to address this problem is discussed in section 6.4.

As the function is unrolled the likelihood of the function requiring more function units that can be placed on the device increases. Aborting the recursive function as it is computing and reporting a lack of logic space is not a desirable solution, so a solution that successfully computes the results of recursive functions that require more function units that can be placed on the device by altering the number of recursive nodes a function unit operates on while still maintaing the throughput invariants, is presented in section 6.5.

6.1 Chapter Aims

- The description of placement and routing algorithms.
- The use of RTR for configuration and its impact on the performance on the technique described in this thesis.
- Reuse of function units to compensate for the finite space available on an FPGA.

6.2 Reducing Complexity

Given a constrained area on an FPGA device modules must be placed on the FPGA such that logic fragmentation is minimised while at the same time ensuring that placement always allows for routing between the placed resource and current resources as well as resources that are to be placed in the future. While it is critical to make efficient use of resources, it is also crucial that the placement and routing algorithms do not consume large amounts of time, thereby reducing their contribution to reconfiguration latency.

To simplify the placement and routing problem to a one dimensional problem it is decided that all function units are synthesized in columns that span the entire height of the FPGA. This requires the constrained area for the function to be the entire height of the FPGA. The area is partitioned into vertical columns, each of which is the width of the largest function unit, with buses for communication on the cell boundaries as shown in figure 6.1. This partitioning into columns reduces the placement problem to a one dimensional problem, as opposed to having rectangular shaped function units that require a two dimensional problem. This reduces the solution space while trying to place a function unit.

To facilitate equal sized columns all function units have equal area constraints that requires all function unit modules have a width the

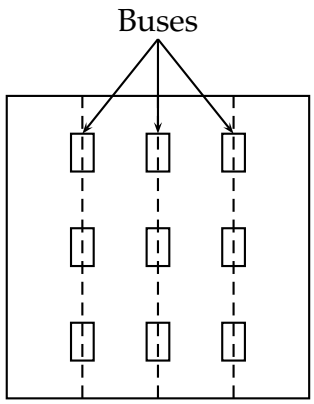


Figure 6.1: Column partitioned area with buses

same as the widest module. This results in fragmentation as the area in the modules corresponding to the smaller function units is not entirely utilized as shown in figure 6.2, where the hatched areas in the device are utilized areas and blank areas unused areas.

As the fragmentation is relative to the width of the column, the only way to reduce the fragmentation is to alter the width of the column. In certain circumstances, it is possible to reduce fragmentation by increasing the width of the column and grouping function units into the same column. Analysis of this also gives insight into the potential value that can be gained by partitioning the recursive function into *pre-recursive* and *post-recursive* functions.

For the purposes of the remaining discussion on the effect of grouping function units on fragmentation the following variables will be defined:

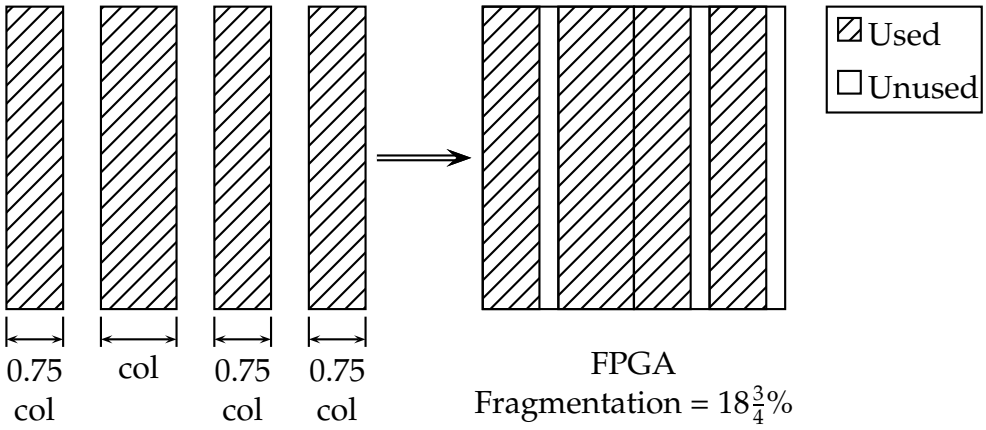


Figure 6.2: Example placement with fragmentation

- *pre* for the width of the *pre-recursive* module
- *post* for the width of the *post-recursive* module
- *non* for the width of the *non-recursive* module
- *max* the largest of *pre*, *post* and *non*
- *total* for the total width of the area the function can unroll in

The recursive tree that is drawn is always proper, as null function calls pass to a dummy function. Any proper *k*-ary tree with *n* internal nodes has $nk - n + 1$ external nodes. Using this property it can be seen that a *k*-ary recursive tree with *n* internal nodes will be required to place *n* *pre-recursive* and *post-recursive* function units and $nk - n + 1$ *non-recursive* function units onto the FPGA. Allocating a column to each of these function units will require $nk + n + 1$ column partitions on the device.

For the remainder of this discussion, n will correspond to the number of internal nodes in the maximal sized tree that can be drawn on the device. As each of the $n + nk + 1$ columns will be of size max, $total \geq (n + nk + 1)\, max$.

The first grouping that will be considered is the grouping of two function units together, g_1 and g_2 with the remaining function unit u remaining alone.

Grouping any two function units reduces the number of columns required for a tree with n internal nodes to $nk + 1$, as each function unit in the column can operate independently which enables each to operate for different nodes in the tree. For better utilization to occur $max_{grouped}(nk + 1) \leq total$, where $max_{grouped}$ is the value of max after grouping and $max_{ungrouped}$ is the value of max prior to grouping. There are three possibilities for the value of $max_{grouped}$ and $max_{ungrouped}$ that must be examined to determine whether this change allows for a greater number of function units to be place onto the device:

1. $max_{ungrouped} = u$ and $max_{ungrouped} = u$, ie. when $u \geq (g_1 + g_2)$
2. $max_{ungrouped} = u$ and $max_{grouped} = g_1 + g_2$, ie. when $u < (g_1 + g_2)$
3. $max_{ungrouped} = g_1$ or $max_{ungrouped} = g_2$

If condition 1 is met utilization is always improved as the equation reduces to the inequality $(nk + 1) < (nk + n + 1)$, which is always true.

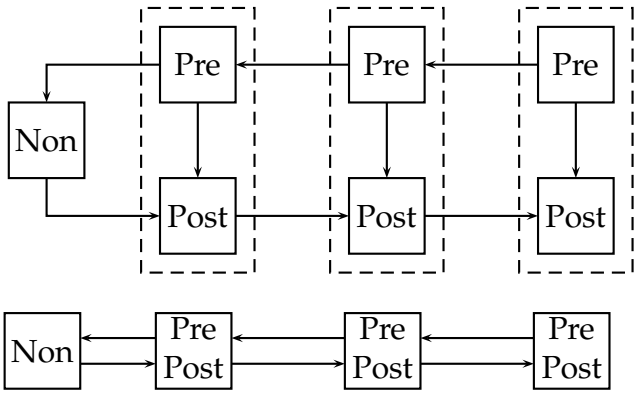
If condition 2 is met, better utilization occurs when the inequality $(nk+1)(g_1+g_2) < total$ is satisfied.

If condition 3 is met, better utilization occurs when the inequality $(nk+1)(g_1+g_2) < total$ is satisfied. This can be reduced to when $\frac{g_1}{g_2} \gtrapprox k$ if $g_1 > g_2$ as shown in equation (6.1). As similar proof exists to show that this can be reduced too $\frac{g_2}{g_1} \gtrapprox k$ if $g_2 > g_1$.

$$\begin{aligned} nk(g_1+g_2) &< total \\ total &= g_1(nk+n+1) \\ nk(g_1+g_2) &< g_1(nk+n+1) \\ nk(g_2) &< g_1(n+1) \\ \frac{g_1}{g_2} &> \frac{nk}{n+1} \\ \frac{g_1}{g_2} &\gtrapprox k \text{ as } n \gg 1 \end{aligned} \tag{6.1}$$

Grouping *pre-recursive* and *post-recursive* function units together, as shown in figure 6.2 is an interesting case of grouping which merits further discussion. Grouping these two function units together reduces the amount of routing required on the device as the route that is present between the two blocks is removed. The complexity of routing and placement is also reduced as the pair of function units in the group communicate with the same function units, thus reducing the

number of different routes that are required to half. It should be noted that this does not reduce the number of wires required for routing as the previous routes are now required to be bi-directional. To maintain parallelism this bi-directionality is implemented by doubling the wires in a connection.



Grouping the function units together also allows them to share the same RAM module on the FPGA. This is possible in most modern FPGA devices as the embedded RAMs are dual port. As a result of both function units accessing the same memory module it is possible that the *post-recursive* function unit may not require a RAM dedicated to its operation, thereby reducing the total number of memory modules allocated to the pipeline. In designs where the limiting factor to the size of the pipeline is the number of distributed RAMs this may aid in realising a deeper pipeline.

For these reasons the *pre-recursive* and *post-recursive* modules are

always grouped together. The analysis in equation 6.1, applies to this situation to gauge whether this is detrimental to fragmentation. This grouping reduces the number of columns required to be $nk + 1$.

For the analysis as to whether grouping all modules will reduce fragmentation, one new variable will be introduced $pre\text{-}post$ for the combined size of pre and $post$. The analysis in equation (6.2) assumes that $pre\text{-}post > non$, however a similar analysis holds true if that assumption is false.

$$\begin{aligned}
(nk - n + 1)(pre\text{-}post + non) &< total \\
total &= pre\text{-}post(nk + 1) \\
(nk - n + 1)(pre\text{-}post + non) &< pre\text{-}post(nk + 1) \\
(nk - n + 1)(non) &< pre\text{-}post(n) \\
\frac{pre\text{-}post}{non} &> \frac{nk - n + 1}{n} \\
\frac{pre\text{-}post}{non} &\gtrapprox k - 1 \text{ as } n \gg 1
\end{aligned} \tag{6.2}$$

6.3 Communcation Model

The placing of dependent modules onto the FPGA device requires the routing of signals over some network between the modules to enable communication.

The FPGA device already contains a configurable network that is

used to communicate between logic resources on the device that introduces little latency to computation. Implementing function units that make use of these routing resources means that no logic is used to implement a network-on-chip, thus allowing for a greater number of function units to be placed onto the device.

However the network on the FPGA is complex, and as such routing on the device is an expensive process in terms of time. This further increases the latency introduced when reconfiguring new stages.

Keeping this in mind it is decided that a network-on-chip is used for routing between function units. A connection-orientated network can allow for simple and rapid point-to-point routing and a connectionless network self-routes data, thus eliminating the need to route at runtime. A network-on-chip does however introduce some problems: extra logic is required to implement the network which reduces the maximum depth of the pipeline, and the latency that is introduced as LUTs are used as switches.

From this there are four requirements for any network-on-chip that is to be used:

1. Low delay between connections
2. Simple routing algorithms are available for the network
3. Little logic is used for implementation

4. Logic used scales efficiently as the number of nodes on the network increases

6.3.1 Network

The network used contains toroidal segmented multiple buses that run across each column. This network is based on the Reconfigurable Mesh Bus(RMB) network [29]. The RMB is a circuit switched multiple segmented bus network that guarantees connectivity when it is available. The RMB is too general purpose for the problem being described in this thesis, thus some of the functionality was removed to conserve hardware on the device.

In the network implemnted each horizontal bus line is segmented at each column, with the segments defaulting to being connected. When a connection is created the bus segmented in the appropriate directions. The buses are bi-directional to take advantage of the simplified routing that can be obtained by the grouping together of the *pre-recursive* and *post-recursive* function units. An optimal placing for a tree with 5 nodes, and with the *pre-recursive* and *post-recursive* function units grouped is shown in figure 6.3.

The number of vertical buses is directly proportional to the recursive width growth (g) of the algorithm, with every column corresponding to an internal node of the recursive tree requiring g I/O

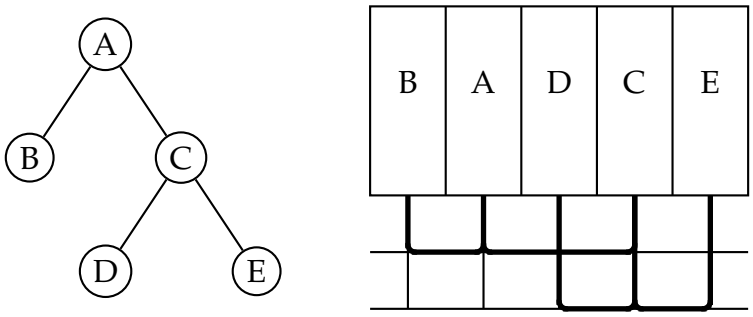


Figure 6.3: Segmented bus with 5 columns

buses to communicate with its children and one I/O bus to communicate with its parent.

Connections can only be made across segments on the same horizontal bus. While this reduces the utilization of resources it greatly simplifies the routing algorithm and therefore the time taken to route.

6.3.2 Placement Algorithm

The case of placing a pipeline with a recursive width growth of 1 is a trivial one, as it simply requires drawing a linear pipeline. As such it will not be discussed in any detail except to mention emperical measurements later.

Drawing a balanced tree on a linear array is typically done by placing a child node on either side of a parent node an equal distance between the parent node and its surrounding nodes as shown in figure 6.4.

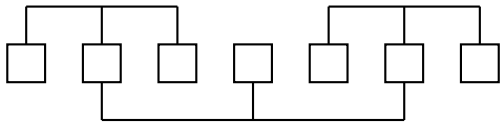


Figure 6.4: H-Tree layout of a balanced binary tree of height 3 in a linear array

This algorithm places rigid constraints on the placement of the tree, with it requiring a balanced tree while drawing. An area that can hold n modules will only be able to have a tree of depth $log_2(n)$ placed onto it. The worst case unbalanced tree with n nodes will have a height of $\frac{n-1}{2}$, and as such $\frac{n-1}{2} - log_2(N)$ nodes will not be placed due to the placement algorithm, even though area is available. To this end the algorithm is modified to the algorithm shown in figure 6.5.

The algorithm PLACE in figure 6.5, places the two children nodes for a node $parent$, based on information in the array $Placed$, which is TRUE at every index corresponding to a column that has a tree node placed into it, and the array $Congestion$ which holds an integer at each index of the array describing how many wires pass through the column corresponding to that index. It attempts to place each of the children nodes in between the parent node and the nodes to the left and right of it. If there are no free columns to place the nodes it places the node in the largest free area available in the device.

The algorithm LARGESTAREA in figure 6.6 finds the columns at the boundary of largest number of contiguous free columns in the area. It

```
PLACE(parent, Placed, Congestion)
 1  closestRight ← first used column to the left of parent
 2  closestLeft ← first used column to the right of parent
 3  if closestLeft = parent −1
       then
 4          leftPlace ← LARGESTAREA(parent, Used, Congestion)
 5  else
 6          leftPlace ← (closestLeft + parent) / 2
 7          Placed[leftPlace] = TRUE
 8  if closestRight = parent +1
       then
 9          rightPlace ← LARGESTAREA(parent, Used, Congestion)
10  else
11          rightPlace ← (closestRight + parent) / 2
12          Placed[rightPlace] = TRUE
```

Figure 6.5: Placement algorithm

then decides which boundary to place the new node at by finding the congestion between the *parent* and both boundary columns, such that there is no wire in the freed area, and returning the column that has the least congestion to it. If the congestions are equal it returns the closest column.

The algorithm MAXCONGESTION in figure 6.7 finds the maximum value in the array *Congestion* betwen the indexes *from* and *to*, wrapping around the boundary of the array.

```
LARGESTAREA(parent, Used, Congestion)
 1  largestContig ← the largest series of contiguous unused columns
 2  contigSize ← the number of columns in this series
 3  oppositeContig ← largestContig + contigSize −1
 4  c_left ← MAXCONGESTION(Congestion, largestContig, parent)
 5  c_right ← MAXCONGESTION(Congestion, parent, largestContig)
 6  if c_left < c_right
       then
 7          place ← largestContig
 8  elseif c_right < c_left
       then
 9          place ← oppositeContig
10  else
11          place ← CLOSEST(Parent, largestConfig, oppositeContig)
12  return place
```

Figure 6.6: Algorithm to find the boundary of the largest free area that produces the better placement

```
MAXCONGESTION(Congestion, from, to)
1  i ← from
2  cong_max ← 0
3  length ← Congestion.length
4  while i ≠ to
       do
5          if Congestion[i] > cong_max
             then
6                cong_max ← Congestion[i]
7          i ← (i +1)% length
8  return cong_max
```

Figure 6.7: Algorithm to find the maximum congestion between two points

6.3.3 Routing

Routing between two points in the segmented bus network has two possibilities as the buses are toroidal. When selecting a route, a route that will not require use of an unused wire is favoured, and if a new wire must be used the direction that uses the shortest length of wire is used. The algorithm for this is shown in figure 6.8.

```
ROUTE(Congestion, Point_1, Point_2)
 1  cong_1-to-2 ← MAXCONGESTION(Congestion, Point_1, Point_2)
 2  cong_2-to-1 ← MAXCONGESTION(Congestion, Point_2, Point_1)
 3  length ← Congestion.length
 4  if cong_1-to-2 < cong_2-to-1
       then
 5          route_start ← Point_1
 6          route_end ← Point_2
 7  elseif cong_2-to-1 < cong_1-to-2
       then
 8          route_start ← Point_2
 9          route_end ← Point_1
10  else
11          Set route_start and route_end for minimum number of hops
12  i ← route_start
13  while i ≠ route_end
        do
14          Congestion[i] ← Congestion[i] + 1
15          i ← (i +1)% length
```

Figure 6.8: Algorithm for routing between two points

After selecting the positions for the children nodes for *k*-ary recursive trees, where $k \leq 3$, all combinations are enumerated to find the

best result. As the number of results for a k-ary tree is 2^k the value grows exponentially with k. For larger values of k, the children are partitioned into $\frac{k}{2}$ sets of size two, in any arbitrary order and each set has the best permutation worked out for it in turn.

Placement and routing is controlled by the arbitrator controlling reconfiguration as it is the logic that determines the need for a new stage of recursion. It controls the states of all segments in the bus by feeding data into each column in the bus describing the state of the segments in it.

Experiments show that this algorithm requires at most $\frac{n-1}{2}$ to place and route any k-ary tree with n nodes.

6.4 Runtime Reconfiguration

The modules are placed onto the device using RTR. RTR allows for partial configuration of logic on the FPGA without interrupting otherlogic that is operating. For this section Xilinx's Virtex architectures will be referred too during discussion as they are the only commercial runtime reconfigurable FPGA devices currently available.

The selection of column orientated module suits the configuration architecture of the Virtex devices, which implements column based frames. Changing one bit on the FPGA requires reloading the entire

frame that bit belongs to. As frames span an entire column, editing one bit in a column requires the entire column to be reconfigured. Thus arranging the modules in a way such that as many bits in the column are reconfigured at the same time reduces the number of frames being configured and hence the time required to configure the entire module.

The modular RTR design flows recommended and supported by Xilinx [26], places the following constraints on the developer:

1. All modules span the entire height of the FPGA
2. Horizontal placement of the modules must be on a four slice boundary
3. All modules have a width that is a multiple of four slices
4. All communication out of modules is done using a special bus macro.

Using the modular design flow presents an interesting problem: routing cannot be placed into the same column as a module as configuring a new module will destroy any routing configuration. Using a technique such as readback of the current configuration to obtain the state of the network in the area for that module before editing the bitstream is a possiblity, however this increases the time required to

do runtime reconfiguration and either requires the ability to manipulate the bitstream which requires an external host, or the bitstream to be recreated which takes far too long. To solve the problem the routing is placed in static modules spanning columns between modules as shown in figure 6.9. Bus macros are used to connect each module with its neighbour. This method results in logic being underutilized as typically the control logic for the network does not require a module the width of four slices.

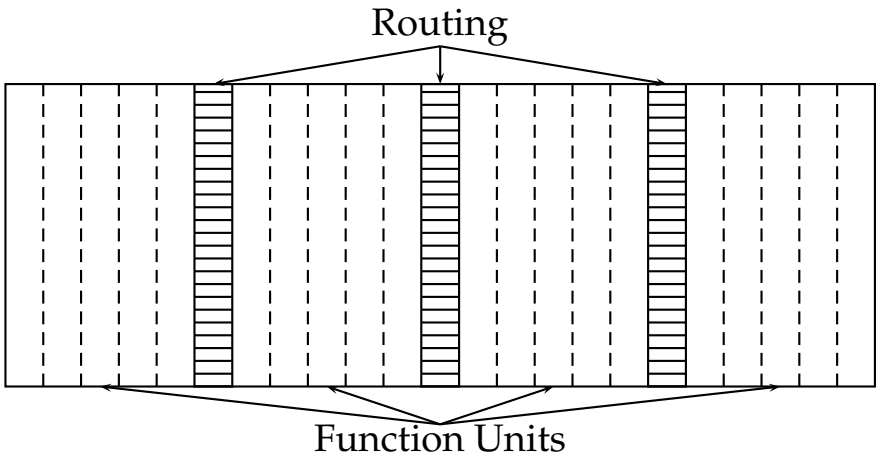


Figure 6.9: Configurable modules and routing modules placement

Generating bitstreams for each location a module can occupy may result in many bitstreams being created. Doing this can require a large amount of memory and as such may not be a satisfactory solution. A solution such as the solution presented in the XPART package [13], that when given a bitstream and a bounding rectangle can generate the bitstream required to configure the logic in that bounding rectan-

gle in a shape of equal size in another location on the FPGA would be more suitable. This solution would allow a module to be replaced into a different area on the FPGA. XPART however relies on the presence of Power PC cores which are not available on all FPGA devices, and most importantly the XPART software itself is not readily available. This makes testing of such techniques difficult on real devices. To overcome the lack of availablity of these tools, and the lack of hardware to support them at any instance, a simulator was developed for testing of the RTR techniques presented in this document. This simulator is described in section A.

6.5 Limitations On Logic Size

The amount of logic resources on an FPGA device places a limit on the number of function units that can be placed on the device. This limit restricts the size of the pipeline that is created and therefore the amount of unrolling that can be done.

Limiting the number of times a function can be unrolled reduces the performance benefit that can be gained from using the unrolling method described in this thesis. While the finite logic resources restricts the depth of the recursive function that can be realised by physically placing function units on the device, methods can be used that

will facilitate recursive depths greater than that which would fit on the device.

Past techniques into circuits that are larger than the FPGA rely on hardware virtualization techniques [32, 31, 57, 4], that "page" computational contexts out of the FPGA and place other contexts into the FPGA in a similar way to a virtual memory system. This technique cannot be applied to the problem of unrolling recursion as it is inefficient to save context and then reconfigure the device and reduces the parallelism available in the system.

Bondalapati et al address this problem when mapping iterative loops onto reconfigurable devices [17, 14] in two different ways. The first way they adopt reuses creates a cycle at each stage of the pipeline, with each stage feeding back onto itself multiple times before passing data to the next stage. While this reduces the parallelism in the operation it is the technique that is extended and will be described in this section.

Functions that map into a pipelined linear array exhibit the property that each stage does no more work than the last stage. This allows for each stage of the pipeline to work on multiple stages of recursion without requiring buffering to compensate for unconstant throughput.

Consider the recursive tree in figure 6.10, which shows that every

level is controlled by a single function unit and the time taken at each stage of the pipeline is less than the last. The accompanying pipeline illustrates the pipeline for this tree.

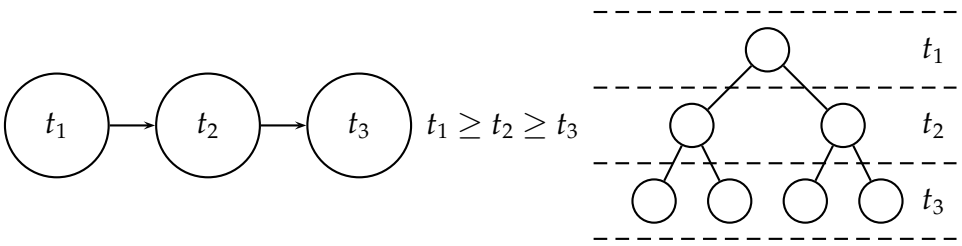


Figure 6.10: Recursive tree and its corresponding linear pipeline

If the tree increased to a depth of five and this could not be placed due to a lack of available logic on the device, each stage of the pipeline will now control two consecutive levels of the tree, as shown in figure 6.11 with the invariant that every stage in the pipeline take less or the same amount of time to do a unit of work being maintained.

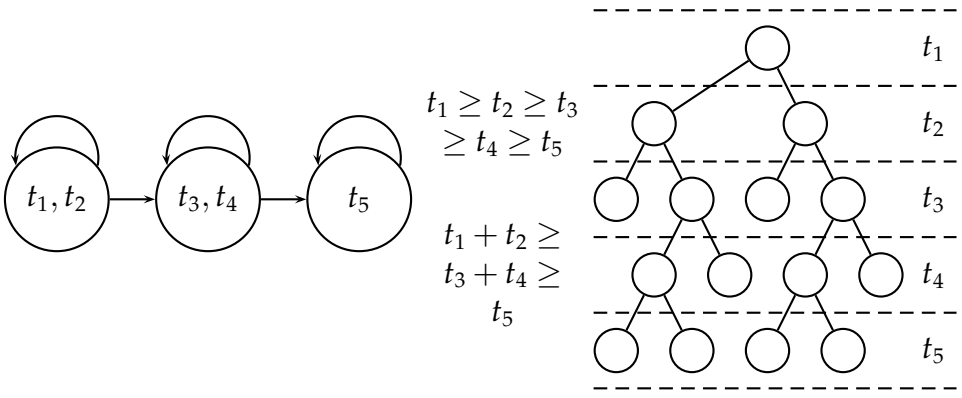


Figure 6.11: Recursive tree and its corresponding linear pipeline implementing reuse

The same process cannot be applied to functions with recursive growth rates (g) that are greater than one which result in tree shaped pipelines as grouping subsequent levels of recursion with their parent level either causes throughput to become unconstant, or results in an increase in the degree of the tree corresponding to the pipeline which increases the complexity of routing.

Instead all sibiling nodes in the tree are grouped together as shown in figure 6.12. This maintains a tree of degree the same as that of the original tree, thereby only ever requiring routing to be removed not added. This does not however maintain constant throughput, as the work done in the first level is less than the total work done in the second level due to the recursive growth rate. To maintain constant throughput the first two levels are grouped together. The proof that the result of grouping these two levels and the grouping on the subsequent level maintains contant throughput is in equation (6.3). For the equation two variables are introduced g, for the recursive growth rate, and d which is the ratio of work done between a recursive node and all of its children. As shown in equation 5.3 in section 5.3, $g = \lceil d \rceil$, so therefore $g \leq d$.

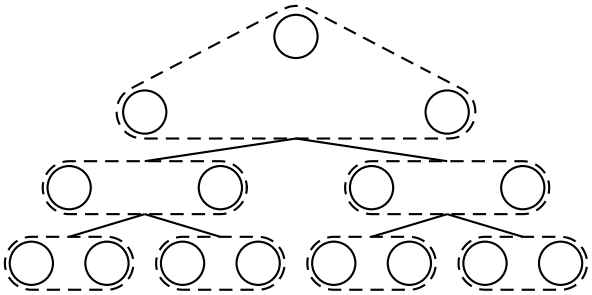


Figure 6.12: Tree shaped pipeline reusing nodes in the tree

$$
\begin{aligned}
t_1 + gt_1 &\geq \frac{g^2 t_1}{d} \\
g^2 - dg - d &\leq 0 \\
|\, g \,| &\leq \frac{d + \sqrt{d^2 + 4d}}{2}
\end{aligned}
\tag{6.3}
$$

This is true as $g \leq d$ and $d \geq 0$

Throughput between subsequent stages in the pipeline is constant, as the work done by two nodes from stage i, $2W_i$ is greater than the work done by two nodes from stage $i+1$, $2W_{i+1}$, as $W_i \geq W_{i+1}$.

6.6 Memory

Many streamed applications do not require memory for operation. These are applications that access an input stream sequentially. However some applications exist that require memory to buffer data.

Applications that require the buffering of data do not access data

in the input stream sequentially, or receive multiple streamed inputs that arrive sequentially and are dependent on each other. These circumstances require the data to be buffered in memory while waiting for the necessary area of the stream to be inputted.

Memory is also required to store context information when a function unit is controlling multiple nodes in the recursive tree. In circumstances where input arguments for the multiple nodes it controls do not arrive in a contiguous stream, a context must be saved which allows the function unit to remember its state when it receives an argument for a particular instance of a function. As the number of contexts stored by a function unit grows exponentially as the recursive depth of the node increases memory is required to store the context information.

As with the von Neumann bottleneck [9], the bandwidth between the numerous function units on the FPGA and memory becomes critical to the performance of the method described in this thesis. To provide this bandwidth the large number of distributed memory modules that are available on current commercial FPGA devices is utilized.

This chapter begins by presenting information on the memory that is currently available on current commercial FPGA devices in section 6.6.1. Memory allocation schemes that are then presented in section 6.6.2.

6.6.1 FPGA Memory

Modern FPGA devices contain embeded memory blocks on the chip. These blocks can be accessed by the circuits that implemented on the FPGA device. The memory blocks are typically arranged around the edge or in the centre of the FPGA as shown in figure 6.13. The memory contained in the blocks is usually dual port SRAM memories.

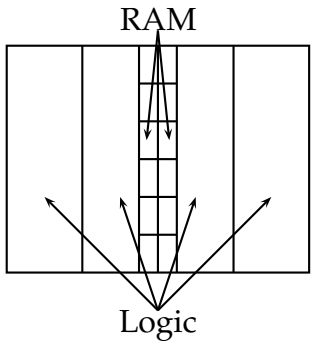


Figure 6.13: Memory layout on Xilinx FPGAs

The amount of memory and the number of modules that this is split into varies between manufaturer and amongst devices depending on device size and cost. A brief survey containing the number and size of distributed memory blocks in current Altera and Xilinx FPGAs is shown in table 6.1. It is worth noting that Altera devices contain a great number of small memory blocks, and a small number of large memory blocks while Xilinx devices contain memory blocks

of only one size. The greater number of logic blocks in Altera devices gives the user a large amount of bandwidth to the smaller memory blocks, however due to the bulk of the memory being contained in larger memory blocks the amount of bandwidth available for larger data sets is reduced.

Device	512bit Blocks	4Kb Blocks	18Kb Blocks	512Kb Blocks	Total Kbs
EP2S15	104	78	0	0	384
EP2S30	202	144	0	1	1,189
EP2S60	329	255	0	2	2,208
EP2S90	488	408	0	4	3,924
EP2S130	699	609	0	6	5,857
EP2S180	930	768	0	9	8,145
XC4VFX12	0	0	36	0	648
XC4VFX20	0	0	68	0	1,224
XC4VFX40	0	0	144	0	2,592
XC4VFX60	0	0	232	0	4,176
XC4VFX100	0	0	376	0	6,768
XC4VFX140	0	0	552	0	9,936

Table 6.1: Size and amount of distributed memory available on Altera Stratix-II [19], and Xilinx Virtex-4 Devices [22]

It is desirable to use the on-chip memory resources over off-chip memory resources as there is a greater number of distributed memory blocks on-chip than there are memory ports to RAM off-chip. As many function units may require parallel memory access to not reduce the parallel operation of the function units it is desirable to increase the number of memory ports available for each function unit to access. It is also worth nothing that the disparate addresses accessed

by the function units does not require any two function units to be accessing the same memory.

A single ported, or even a dual-ported, off-chip memory module would require a MMU with time multiplexing to facilitate parallel access by distinct function units to the memory. This reduces the amount parallel operation between the function units as the become dependent on each others memory access as a consequence of the time multiplexing forcing them to stall while accessing the memory. Having a large number of distributed RAM modules meets the first criteria necessary of providing many memory ports, whilst taking advantage of the disparate memory access patterns which not requiring that different function units access the same RAM module does not requre the memory modules to be of large size.

6.6.2 Allocation

Allocation of the large amount of memory bandwidth available on an FPGA device using algorithms that observe the behaviour of recursive functions and differing circumstances in which memory is required is crucial to using memory efficiently on the FPGA.

This section presents the allocation method used to allocate memory blocks for context information in section 6.6.2, before presenting the method used to allocate memory blocks while buffering in sec-

tion 6.6.2.

Allocation for context information

The amount of memory used to store context information per function unit is related to the number of contexts that function unit computes. If the recursion were balanced a function unit operating on a deeper level of recursion will control more contexts than a previous level of recursion, and the number of contexts is predictable. The number of contexts that can be obtained from assuming balanced recursion is the maximum possible contexts that can be controlled by a node at that level.

Allocation of memory based on the maximum number of contexts that may be controlled by a function unit can result in memory being allocated ineffeciently when the recursion is unbalanced. The value is not useless however as for function units that at most control a small number of contexts it is possible to use LUTs to control their logic. All other function units are allocated a memory module, with *pre-recursive* and *post-recursive* pairs sharing the same module if it is a dual port memory. The remaining modules are placed into a "memory pool". When requiring more storage capacity than available in a single module a function unit request a module from the pool to store subsequent data.

An arbitrator in between the function units and the memory pool manages all incoming requests. Upon receiving a request the arbitrator finds any free modules in the pool. The first free module found is allocated to that function unit and marked as allocated. If no free modules exist, off-chip memory must be used.

For function units that will at maximum control a small number of contexts no memory module is allocated to them and they use LUTs to store data. This reduces the number of memory modules that are allocated.

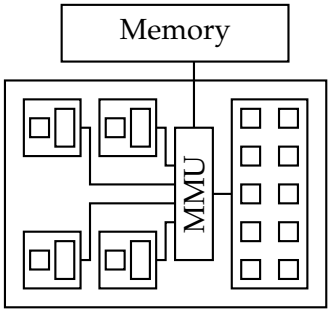


Figure 6.14: Memory architecture of device

Allocation for buffering

When buffering unbounded arrays the maximum amount of memory required is unknown. Applying the allocation method used for the context memory allows the first function unit in the array to be allocated all the memory modules in the pool in situations where it must buffer an input array that will not fit in the memory available on the

device. This inhibits subsequent levels of recursion from accessing memory.

The allocation scheme chosen for this buffering problem attempts to allocate memory modules to function units more fairly by analysing the behaviour of the function. If the change in the amount of data between recursive functions can be easily determined it is possible to determine the number of modules allocated to a function unit relative to another function unit to ensure fairness. In circumstances where this cannot easily be determined an equal number of modules is allocated per level.

The modules allocated to each function unit act as a cache, with larger off-chip memory used by all function units as the major storage element. This greatly reduces the parallelism between function units as they contend for memory access and is why the proposed method for parallel recursion exhibits a reduction in performance when array accesses are not sequential. It is worth noting that a function with irregular array indexing is examined as a case study in section 7.5, with a proposal for reordering the input stream to reduce the need for buffering.

6.7 Summary

This chapter presented the methods that are required to place the modules required to compute the result of a recursive function onto an FPGA device and the method used to route between the modules. It then presents how memory bandwidth is allocated to the function units on the FPGA.

The important points made in this chapter are: the partitioning of the device into columns to simplify placement; the grouping of modules to reduce fragmentation and routing overhead; the use of a low-logic on-chip network to route between modules without impacting on latency; the use of RTR to place new modules onto the device and to alter modules previously placed onto the device; and the pipeline stage reuse techniques used to overcome the lack of logic available on the FPGA device.

It is worth noting that the network used falls well short of the optimal number of wires required, which is $O(log(N))$. Techniques such as compaction of connections, which allows a connection to be shifted vertically "upwards" through buses after being made, which is a core feature of the RMB model, could be used to approach this value. However without a larger amount of hardware dedicated to routing it would be difficult to approach this value.

Chapter 7

Case Studies

7.1 Introduction

This chapter presents several case studies which demonstrate the application of the techniques that have been described in this document.

The case studies include merge sort, quick sort, strassen's matrix multiplication algorithm, quad tree partitioning for force approximation and tree search. For all function descriptions the symbol $\oplus$ corresponds to the concatenation operation. When this statement appears in the form $a \oplus b$, all the values in the array b are concatenated to the end of the array a. If b is a scalar its value is concatenated to the end of a.

All measurements for the case studies were either done using a Celoxica RC200 board [2] which contains a Xilinx XC2V1000 Virtex-II

device, or a device simulator. The device simulator was used as in some circumstances the device being tested on did not support run-time reconfiguration. The measurements that were made ran on the highest clock speed that the stack implementation would run at even if the unrolled function could be clocked at a higher rate. For measurements made on the device simulator results are given in terms of clock cycles.

As a comparison the results obtained were compared with a stack based implementation on the same device. A stack based implementation was chosen as it allows the results to be compared with how recursion has been previously implemented. Implementing the stack on the same device allows for a more fair comparison between results, with the same benefits and limitations being imposed on both measurements.

Comparing the unrolling method described in this thesis with a stack based implementation also highlights the differences between the methodologies presented in this thesis and the traditional methods of thinking associated with implementing recursion.

7.2 Merge Sort

7.2.1 Algorithm

Merge sort is a simple recursive algorithm that has a predictable depth given the length of the sequence to be sorted. For the implementation discussion that follows it is assumed that the length of the entire sequence is not known. The function implemented corresponds to the code in figure 7.1, which has been adapted from the merge sort algorithm by Orenstein et al [53]. This algorithm is based on a bottom-up description of merge sort. The top-down approach of merge sort only partitions the input array into smaller arrays. The bottom-up approach described here can be thought of as partitioning the array "in-place", as well as doing the merge operation "in-place".

This case study has been chosen as it is simple to understand, and serves as an introductory point for the case studies.

The flowgraph for the algorithm is shown in figure 7.2. Function analysis on this flowgraph obtains the following basic blocks for the sub-functions:

- $non\text{-}recursive$: $\{1, 4\}$
- $pre\text{-}recursive$: $\{1, 2, 3, 5\}$
- $post\text{-}recursive$: $\{3\}$

```
MERGE-SORT(A, window)
1  if window < A.length
2     then index ← 0
3          while index ≠ A.length
4               do index ← index +2 × window
5                  MERGE(A, index, window)
6
7          return MERGE-SORT(A, 2 × window)
8  else
9          return A

MERGE(A, index, window)
1  A[index] ← MERGE-LISTS(A[index], A[index + window], window)
```

Figure 7.1: Merge Sort Algorithm

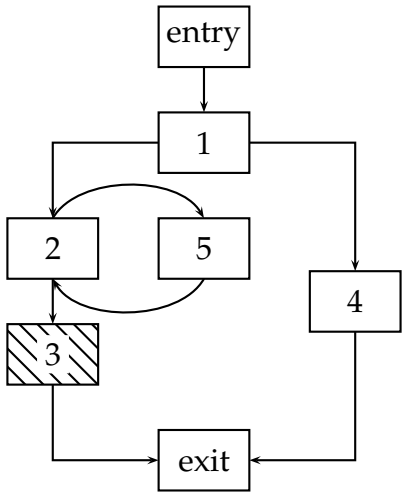


Figure 7.2: Flowgraph for the merge sort algorithm in figure 7.1

After removing recursive blocks the function *post-recursive* contains no blocks, therefore there is no need to place any modules to operate for this.

7.2.2 Recursive Growth Width

As only a single function call is made the recursive growth width is 1. This means that only one processor is allocated per level of the recursive tree to maintain throughput. This results in a linear pipeline being placed onto the hardware.

7.2.3 Grouping

Grouping the *non-recursive* and *pre-recursive* function units reduces the fragmentation in the system as the size of the *non-recursive* function unit is negligible when compared to the size of the *pre-recursive* function unit. Following this RTR was not required.

7.2.4 Context

As every function unit operates on a single instance of the recursive function no context is required.

7.2.5 Buffering

Every stage of the function is required to buffer up to $2 \times window$ values before operating the MERGE operation as the merge operation does not have sequential accesses on the arrays being merged. Therefore a function unit corresponding to the recursive depth of i will re-

quire enough space to buffer 2^i data items. Following this the entire system will require enough storage for $N-1$ items.

7.2.6 Prediction

The number of function units that are required to compute this function can be calculated with the logarithmic function shown in equation (7.1), where s is the current size of the inputted sequence.

$$pred_{merge}(s) = \left\lceil log_2 \left(\frac{s}{2}\right) \right\rceil \tag{7.1}$$

The time for these elements to be required by the newly configured module, and hence the largest possible time that can be devoted to reconfiguration before the need to stall the system or buffer data is proportional to the configured depth of the recursive tree. Thus with a pipeline of sufficient depth initially configured the reconfiguration can be completed by the time required to place data at the new module and thus the cost of reconfiguration can be completely hidden.

7.2.7 Results

The results of implementing this function on the RC200 board are shown in table 7.1 and the graph in figure 7.3. The comparison was made between this implementation and a stack based implementation

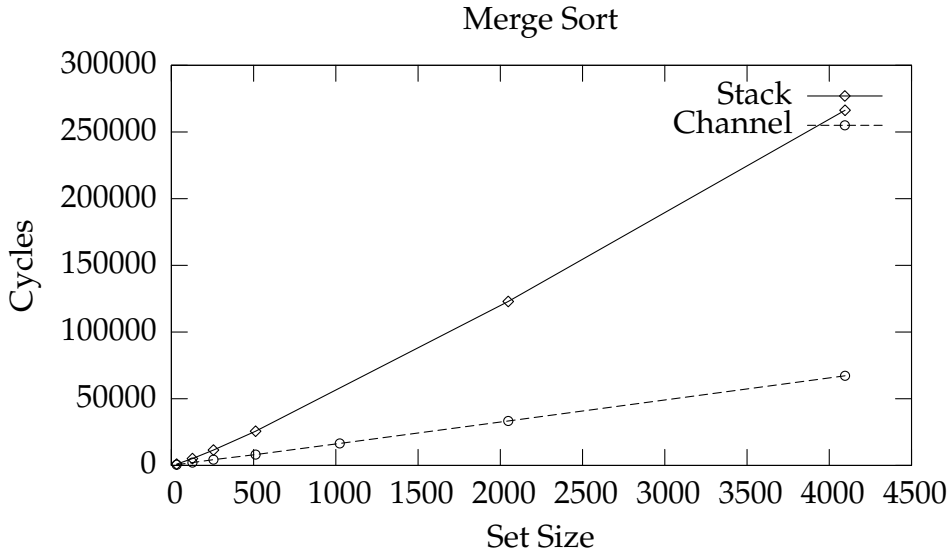


Figure 7.3: Merge Sort Results

of merge sort.

As expected the stack based implementation runs in $O(N \log N)$ time, while the unrolled function runs in linear time.

Data Size	Stack Performance (Cycles)	Unrolled Performance (Cycles)
32	970	500
128	5136	2116
256	11539	4314
512	25622	8117
2048	122908	33310
4096	266271	67207

Table 7.1: Merge Sort Results

7.3 Quick Sort

7.3.1 Algorithm

Quick Sort is a recursive algorithm that sorts a set. Given an unsorted set S, the algorithm determines a pivot $p \in S$, and divides the S into two smaller sets:

- $S_{left} = \{v \mid v \in S \wedge v < p\}$
- $S_{right} = \{v \mid v \in S \wedge v > p\}$

These two sets are then recursively sorted using the quick sort algorithm, until a set with a size less than or equal to 1 is reached. p is then concatenated onto S_{left} before S_{right} is concatenated to the resulting set. This produces a sorted set. This algorithm is shown in figure 7.4.

It is worth noting that the pivot that is selected in our algorithm is always the first element in the list. To balance the recursion the pivot is normally chosen such that it splits the set in half, however finding the median value or even approximating it with the average value before splitting the input set requires the input set to be read multiple times. Doing this will require the entire set to be buffered in memory which is undesired.

```
QUICKSORT(S)
 1  if S.length ≤ 1
 2     then return S
 3  else
 4          p ← S[0]
 5          i ← 1
 6          while i ≤ S.length
 7               do if S[i] < p
 8                     then S_left ⊕ p
 9                  else
10                            S_right ⊕ p
11                  i ← i +1
12          S_left ← QUICKSORT(S_left)
13          S_right ← QUICKSORT(S_right)
14          return (S_left ⊕ p) ⊕ S_right
```

Figure 7.4: Quick Sort Algorithm

The flowgraph for the algorithm is shown in figure 7.5. Function analysis on this flowgraph obtains the following basic blocks for the sub-functions:

- $non\text{-}recursive$: $\{1, 2\}$
- $pre\text{-}recursive$: $\{1, 3, 4, 5, 6, 7, 8, 9\}$
- $post\text{-}recursive$: $\{8, 9, 10\}$

The reason for presenting this algorithm as a case study is that it contains $pre\text{-}recursive$ function units, unlike the merge sort algorithm. It also features a prediction heuristic that is not optimal.

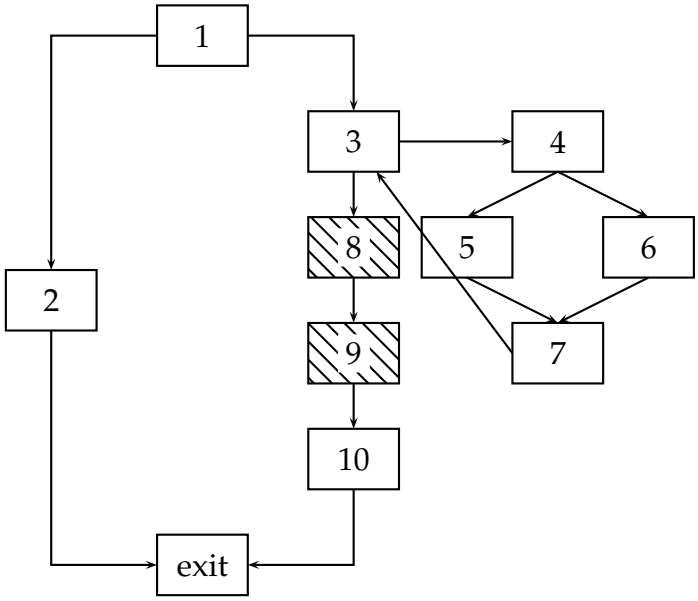


Figure 7.5: Flowgraph for the quick sort algorithm in figure 7.4

7.3.2 Recursive Growth Width

As the amount of data between levels remains constant, and the complexity of the function is $O(N)$, the recursive growth width is 1. This means that only one processor is allocated per level of the recursive tree to maintain throughput.

7.3.3 Grouping

Grouping all the function units reduces the fragmentation in the system as the size of the *non-recursive* function unit is negligible when compared to the size of the *pre-recursive* and *post-recursive* function

units. Following this RTR was not required.

7.3.4 Context

Each *pre-recursive* function unit must store the pivot for each instance of the recursive function it is operating on. This will at most store $\frac{N}{2}$ data items as a recursion tree that is balanced for the quick sort operation will have a width of $\frac{N}{2}$ at its widest level as the height of the tree is $\log_2(N)$.

Each *post-recursive* function unit must store the pivot value also. As the *post-recursive* function unit shares the same context memory as the *pre-recursive* function unit as detailed in section 6.2.

7.3.5 Buffering

No buffering is required. This is due to the values in S_{left} always arriving before the values in S_{right}.

7.3.6 Prediction

The prediction heuristic for quick sort is a non-optimal prediction heuristic. Each level of the recursion predicts the minimum depth of recursion required to sort the data is has inputted into it. The recursive tree of minimum height required to sort N elements using quick

sort can be found using equation (7.2).

$$pred_{quick}(N) = \lceil log_2(N) \rceil + 1 \tag{7.2}$$

As the maximum height of the recursion tree can be N, this value can be incorrect by a large amount at higher levels of recursion when the recursion is very unbalanced. It is worth noting that for random data the height of the recursion tree is on average $O(\log_2(N))$ [38, 39].

7.3.7 Results

The results of implementing this function on the RC200 board are shown in table 7.2 and the graph in figure 7.6. The comparison was made between this implementation and a stack based implementation of quick sort.

As expected the stack based implementation runs in $O(N \log N)$ time, while the unrolled function runs in linear time.

Data Size	Stack Performance (Cycles)	Unrolled Performance (Cycles)
512	47041	8117
1024	101778	16540
2048	214473	30624
4096	558672	59810
8192	1249370	118686

Table 7.2: Quick Sort Results

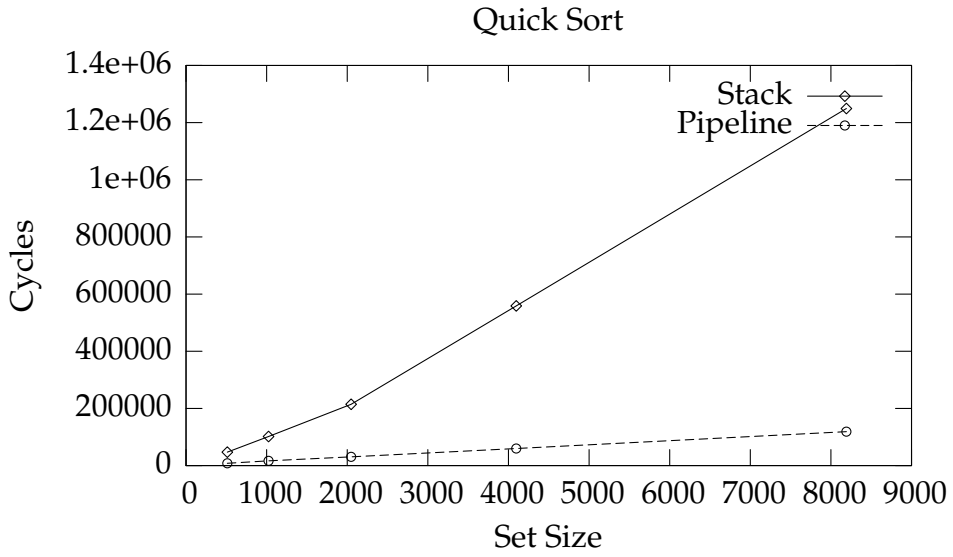


Figure 7.6: Quick Sort Results

7.4 Quad Tree Partitioning

7.4.1 Algorithm

The Quad Tree partioning algorithm is a recursive algorithm that recursively divides a 2-dimensional plane into quadrants until only a fixed number of points exist in each sub-division. This algorithm is not of much interest unless something is done with the data that is partitioned. The algorithm in this section is based on the Barnes and Hut force approximxation algorithm [10] that divides a plane into recursively quadrants to approximate the force exerted by the sections of the plane on points in the plane. A quadtree is used to store the approximation data, with exact data (quadrants with single points) being stored in the leaves. The data at each node of the tree corresponds to the (x, y), of a point and other data necessary to the force calculation(for example mass when calculating gravitational force) at that point. Each internal node describes a point in the plane that approximates the data for all other points in that quadrant, and as such the data stored at an internal node is an approximation based on the data stored in its children nodes. For gravitational force calculations this is the centre of mass of the quadrant. The algorithm for this is shown in figure 7.7.

The flowgraph for the algorithm is shown in figure 7.8. Function

```
QT(left, right, top, bottom, Points)
 1  if Points.length ≤ 1
 2     then Tree.mass ← Points[0].mass
 3          Tree.x ← Points[0].x
 4          Tree.y ← Points[0].y
 5          return Tree
 6  else
 7          middle ← (left+right)/2
 8          centre ← (top+bottom)/2
 9          i ← 0
10          while i ≤ Points.length
11              do if Points[i].x < middle
12                    then if Points[i].y < centre
13                            then Points_sw ⊕ Points[i]
14                            else
15                                 Points_nw ⊕ Points[i]
16                    else
17                            then Points_se ⊕ Points[i]
18                            else
19                                 Points_ne ⊕ Points[i]
20                    i ← i+1
21          Tree_sw ← QT(left, middle, centre, bottom, Points_sw)
22          Tree_se ← QT(middle, right, centre, bottom, Points_se)
23          Tree_nw ← QT(left, middle, top, centre, Points_nw)
24          Tree_ne ← QT(left, right, top, centre, Points_ne)
25          Tree.mass ← Σ Tree_xx.mass
26          Tree.x ← (Σ Tree_xx.mass . Tree_xx.x) / Tree.mass
27          Tree.y ← (Σ Tree_xx.mass . Tree_xx.y) / Tree.mass
28          Tree.sw ← Tree_sw
29          Tree.se ← Tree_se
30          Tree.nw ← Tree_nw
31          Tree.ne ← Tree_ne
32          return Tree
```

Figure 7.7: Force Approximation Algorithm

analysis on this flowgraph obtains the following basic blocks for the sub-functions:

- *non-recursive*: $\{1, 2\}$
- *pre-recursive*: $\{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}$
- *post-recursive*: $\{4, 5, 6, 7, 8\}$

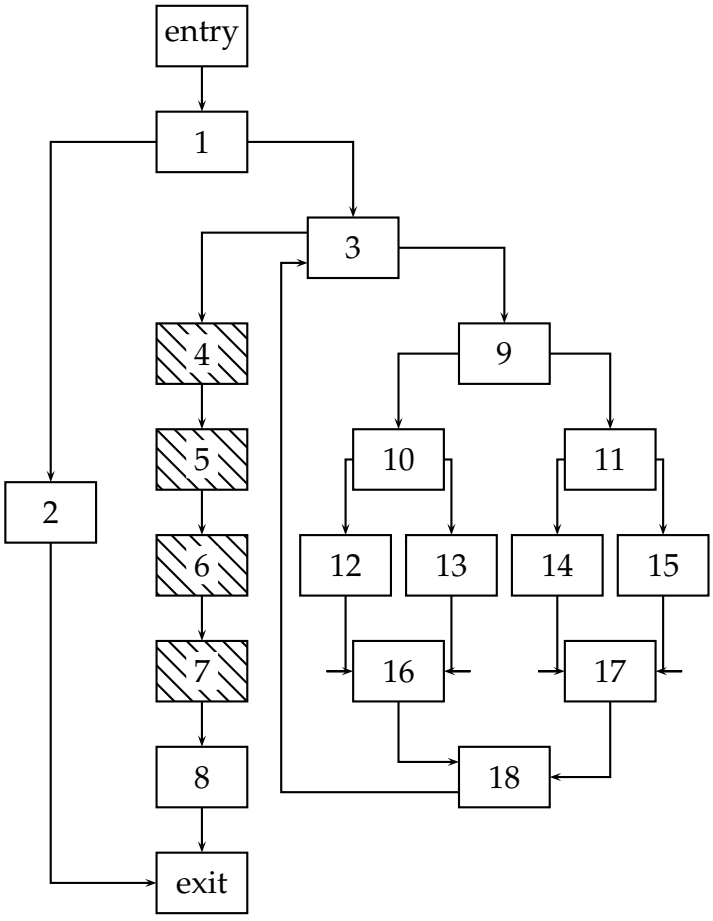


Figure 7.8: Flowgraph for the quick sort algorithm in figure 7.4

The reason for presenting this algorithm as a case study is that it contains *post-recursive* function units that actually execute instructions on the data, unlike the quick sort algorithm. With the exception of this processing the algorithms are very similar as shown in the flow graph.

7.4.2 Recursive Growth Width

As the amount of data between levels remains constant, and the complexity of the function is $O(N)$, the recursive growth width is 1. This means that only one processor is allocated per level of the recursive tree to maintain throughput.

7.4.3 Grouping

Grouping all the function units reduces the fragmentation in the system as the size of the *non-recursive* function unit is negligible when compared to the size of the *pre-recursive* and *post-recursive* function units. Following this RTR was not required.

7.4.4 Context

Each *pre-recursive* function unit must store the *middle*, *centre*, *left*, *right*, *top* and *bottom* values for each recursive function it is comput-

ing. This will at most store $\frac{N}{2}$ data items as a recursion tree that is balanced for the quad tree operation will have a width of $\frac{N}{2}$ at its widest level as the height of the tree is $\log_2(N)$.

The *post-recursive* does not require any context information.

7.4.5 Buffering

No buffering is required, as the array accesses in the algorithm are completely sequential, and the array data that is returned is not operated on.

7.4.6 Prediction

A similar prediction heuristic to the one used for the quick sort algorithm is applied to this algorithm. Each level of the recursion predicts the minimum depth of recursion required to sort the data is has inputted into it. The recursive tree of minimum height required to sort N elements using quick sort can be found using equation (7.3).

$$pred_{quick}(N) = \lceil log_4(N) \rceil \tag{7.3}$$

As the maximum height of the recursion tree can be N, this value can be incorrect by a large amount at higher levels of recursion when the recursion is very unbalanced. It is worth noting that for random

data the height of the recursion tree is on average $O(\log_2(N))$.

7.4.7 Results

The results of implementing this function on the RC200 board are shown in table **??** and the graph in figure 7.9. The comparison was made between this implementation and a stack based implementation of quad tree partioning.

As expected the stack based implementation runs in $O(N \log N)$ time, while the unrolled function runs in linear time.

Data Size	Stack Performance (Cycles)	Unrolled Performance (Cycles)
4	498	8639
16	2726	10534
64	12278	15235
256	53366	28794
1024	229238	70618
4096	972686	209960

Table 7.3: Quad Tree Results

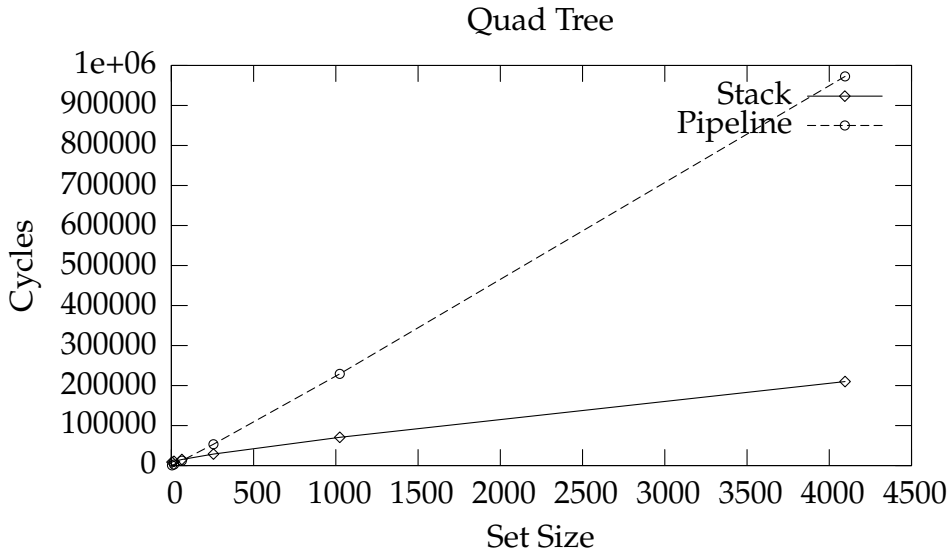


Figure 7.9: Quad Tree Results

7.5 Strassen's Matrix Multiplication

7.5.1 Algorithm

Strassen's matrix multiplication algorithm is a simple recursive algorithm with a predictable depth when given the size of the matrices to be multiplied. The algorithm corresponds to the code in figure 7.10, which has been adopted from [58].

The algorithm is a divide and conquer matrix multiplication algorithm that reduces the complexity of matrix multiplcation from $O(N^3$ to $O(N^{log_2 7})$. The division phase of the operation splits each of the two $N \times N$ matrices that are being multiplied into seven $\frac{N}{2} \times \frac{N}{2}$ matrices, by taking the four quadrants of each matrix and adding them in various permutations. The resulting $\frac{N}{2} \times \frac{N}{2}$ are then multiplied using the same algorithm with each other to result in seven $\frac{N}{2} \times \frac{N}{2}$ matrices, $P_1, P_2, \cdots, P_7$. These are then have a merge operation done on them which again is a permutation of addition of these matrices. They are then placed into quadrants of a $N \times N$ matrix and returned.

The reasons for choosing this algorithm are that it is the only case study that demonstrates the use of multiple function units per level of the recursive call. The results presented in this case study show that if the number of function units allocated per level does not match the recursive growth width a performance penalty is incurred, and that

STRASSEN-MULTIPLY(A, B, n)

1 **if** $n = lower$
  then
2 **return** MULTIPLY-MATRICES(A, B, n)
3 **else**
4 $A_1, A_2, \cdots, A_7 \leftarrow$ SPLIT$_A(A, n)$
5 $B_1, B_2, \cdots, B_7 \leftarrow$ SPLIT$_B(B, n)$
6 **while** $i \leq 7$
  do
7 $P_i \leftarrow$ STRASSEN-MULTIPLY$(A_i, B_i, \frac{n}{2})$
8 $i \leftarrow i + 1$
9 $R_1, R_2, R_3, R_4 \leftarrow$ MERGE-MATRICES$(P_1, \cdots, P_7)$
10 **return** $\begin{pmatrix} R_1 & R_2 \\ R3 & R_4 \end{pmatrix}$

SPLIT$_X(M, n)$

1 $\begin{pmatrix} M_1 & M_2 \\ M3 & M_4 \end{pmatrix} \leftarrow M$
2 $i \leftarrow 1$
3 **while** $i \leq 7$
  do
4 $P_i \leftarrow a_i M_1 + b_i M_2 + c_i M_3 + d_i M_4$
5 $i \leftarrow i + 1$
6 **return** $P_1, P_2, \cdots, P_7$

MERGE-MATRICES$(P_1, P_2, \cdots, P_7)$

1 $i \leftarrow 1$
2 **while** $i \leq 7$
  do
3 $R_i \leftarrow a_i P_1 + b_i P_2 + \cdots + g_i P_7$
4 $i \leftarrow i + 1$
5 **return** R_1, R_2, R_3, R_4

Figure 7.10: Strassen's Matrix Multiplication Algorithm

allocating more function units than the recursive growth width does not result in any significant performance increase.

7.5.2 Recursive Growth

The total amount of data that is processed per level does increase. The first level operates on $2N \times N$ matrices, which means it processes $2N^2$ elements of data. The next level operates on $14\frac{N}{2} \times \frac{N}{2}$ matrices, or $\frac{7}{2}N^2$ elements of data. It can be shown that the amount of data increases per level at a rate of $\frac{7}{4}$ per level. Therefore the recursive growth ratio is 2.

7.5.3 Grouping

As the *non-recursive* function units contain multipliers they require a large amount of area. Thus only the *pre-recursive* and *post-recursive* function units are grouped together.

7.5.4 Context

The context that is required to be stored are the input matrices for all the instances of the recursive function that a function unit is controlling. A function unit at depth i will at most store contexts for 4^i instances of the recursive function.

As the depth of a node increases, the number of contexts it stores increases, but the size of the data in each context decreseas. This is due to the matrix size reducing from N^2 to $\frac{N^2}{4}$. A function unit at a depth i multiplies matrices of size $\frac{N^2}{4^i}$. Following this the most data that each function unit will store is N.

7.5.5 Buffering

The amount of data that is to be buffered is $O(\frac{N^2}{2})$ for each $N \times N$ matrix multiplication. The first level of recursion will require enough memory storage for $\frac{N^2}{2}$ data items, the next will require $7\frac{N^2}{8}$ as it multiplies 7 $\frac{N}{2} \times \frac{N}{2}$ matrices. The i^{th} level of recursion will require enough memory storage for $7^i \frac{N^2}{2 \times 4^i}$. For a depth of d this results in the amount of storage in equation (7.4).

$$\begin{aligned} storage &= \sum_{k=0}^{d} \frac{7^k N^2}{2 \times 4^k} \\ &= \frac{N^2}{2} \sum_{k=0}^{d} \left(\frac{7}{4}\right)^k \\ &\approxeq \frac{N^2}{2} \left(\frac{7}{4}\right)^{d+1} \end{aligned} \tag{7.4}$$

7.5.6 Prediction

In a similar manner to the merge-sort algorithm this algorithm continues to split the size of the data given to the next instances of the algorithm in half, relative to the dimensions of the matrices. Given a truncation condition where the dimensions of the matrix is *lower*, we find that equation (7.5) matches the depth of recursion.

$$pred_{strassen}(s) = \left\lceil log_2 \left(\frac{N}{lower} \right) \right\rceil \tag{7.5}$$

7.5.7 Results

The results in table 7.4, demonstrate the performance of the unrolled pipeline when compared to a stack implementation of Strassen's algorithm. Results from the stack implementation when the data was of size 256 was not possible as there was not enough memory available on the board.

Data Size	Stack Performance (Cycles)	Unrolled Performance (Cycles)
4	185	177
16	1437	1447
64	10134	8235
256	N/A	62163

Table 7.4: Strassen Matrix Multiplication Results

The effects of changing the recursive growth width on the perfor-

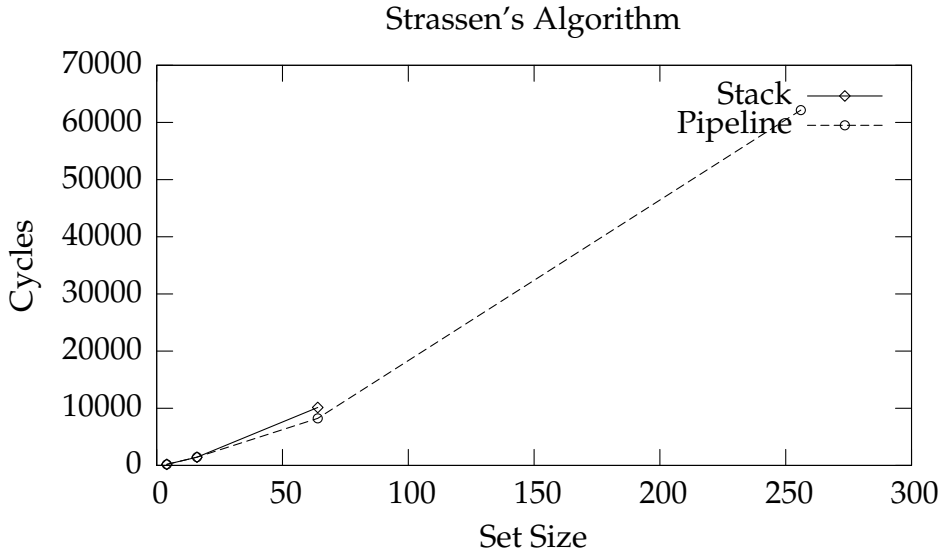


Figure 7.11: Strassen's Algorithm Results

mance of the algorithm can be shown in figure 7.5. While the performance when the growth rate is set to 2 and 3 features similar ratios to each other, the performance gap when the value grows at a higher rate.

Data Size	Growth = 1 Performance (Cycles)	Growth = 2 Performance (Cycles)	Growth = 3 Performance (Cycles)
4	159	177	171
16	1256	1447	1290
64	8812	8235	7809
256	72857	62163	59235

Table 7.5: Comparison of growth width

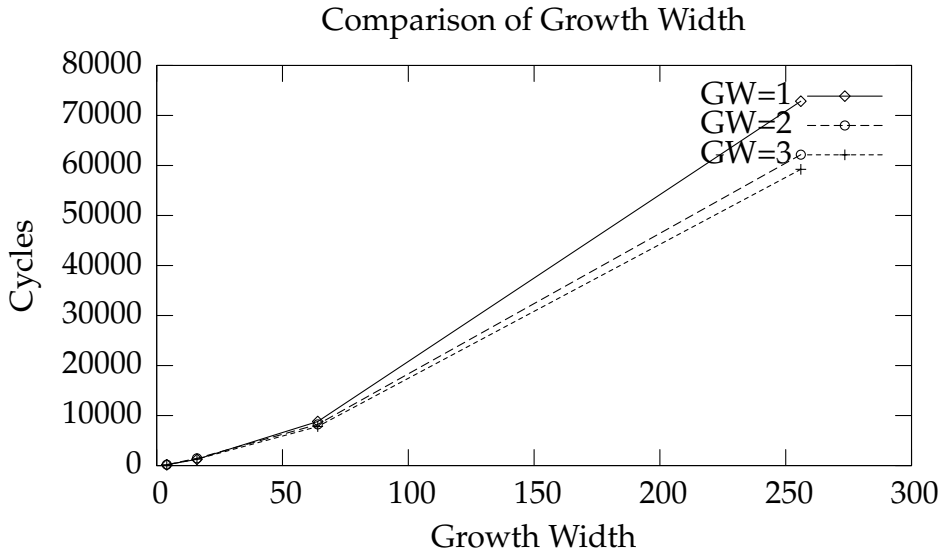


Figure 7.12: Comparison of Growth Widths

7.6 Parallel Tree Search

This section presents a case study which shows how the methods in this thesis may be applied to the mapping and maintenance of recursive data structures to reconfigurable systems. The recursive data structure that is being mapped in this case is a binary tree, with each node of the tree containing a positive integer as a key.

7.6.1 Algorithm

There are two algorithms that are mapped in this section. The first is the insertion algorithm shown in figure 7.13, and the second is the search algorithm shown in figure 7.14.

```
INSERTVALUE(Node, k)
 1  if k < Node.key
 2     then if Node.left = NULL
 3             then create new node at Node.left with key set to k
 4          else
 5                  INSERTVALUE(Node.left, k)
 6  else
 7          if Node.right = NULL
 8             then create new node at Node.right with key set to k
 9          else
10                  INSERTVALUE(Node.right, k)
```

Figure 7.13: Binary Tree Insertion Algorithm

Both of these algorithms will be run on the same hardware mod-

```
SEARCH(Node, k)
1  if k = Node.key or Node = NULL
2     then return Node
3  else
4          if k < Node.key
5             then return SEARCH(Node.left, k)
6          else
7                  return SEARCH(Node.right, k)
```

Figure 7.14: Binary Tree Search Algorithm

ules, with the insertion code updating the memory blocks for each hardware module and the search algorithm reading the data in the memory blocks.

7.6.2 Recursive Growth Width

As both functions only ever call each other once the recursive growth is 1. This scheme has a single function unit allocated to each level of the binary tree, with it storing all the values in that level of the binary tree.

7.6.3 Grouping

The *non-recursive* for the INSERTVALUE algorithm only stores the data in memory, hence this is incoroporated into the *pre-recursive* function unit. In a similar fashion, the *pre-recursive* function unit for the

SEARCH algorithm only reads memory, and hence is also incoroporated into the *pre-recursive* function unit for the same algorithm. Following this RTR was not required.

7.6.4 Context

The information stored at each function unit is the data for that level of the binary tree. This will at most store $\frac{N}{2}$ items.

7.6.5 Buffering

There is no buffering required as all the input data is scalar.

7.6.6 Prediction

As all function units are identical and allocated when the function beings, no prediction mechanism is required.

7.6.7 Results

The results for insertion into the tree show that insertion for the unrolled version is linear. This does not hold true for the stack implementation where the graph in figure **??**, shows a slightly increasing gradient.

Data Size	Stack Performance (Cycles)	Unrolled Performance (Cycles)
512	15871	7153
1024	34815	13297
2048	75775	28657
4096	163839	53233
8192	352255	102385
16384	752663	204785

Table 7.6: Insert Times

The same holds true for searching, with an item being returned from the search algorithm every 12 cycles. This linear relationship is not true for the stack implementation with smaller trees requiring fewer cycles than large trees.

Data Size	Stack Performance (Cycles)	Unrolled Performance (Cycles)
512	10239	6227
1024	22527	12382
2048	49151	24681
4096	106495	49268
8192	229375	98431
16384	491519	196746

Table 7.7: Search Times

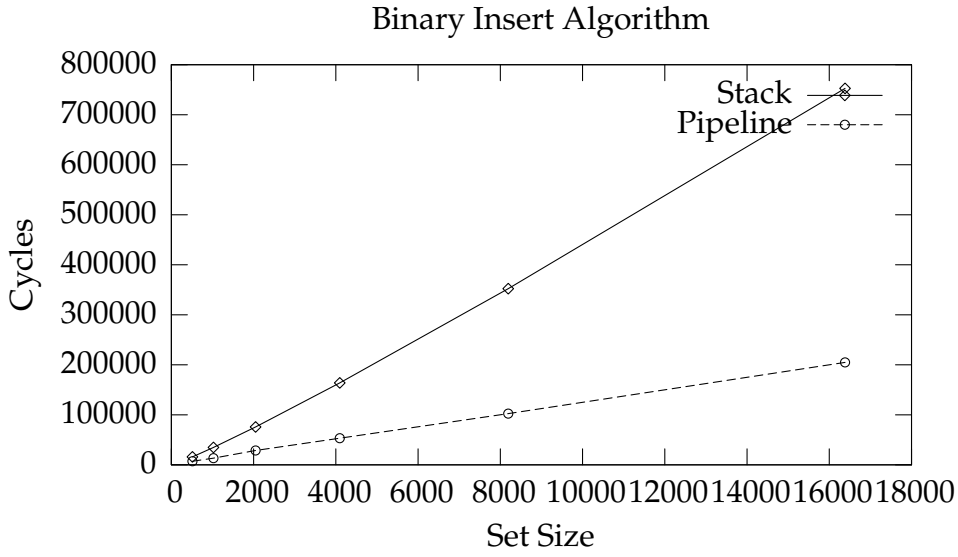


Figure 7.15: Binary Tree Insert Results

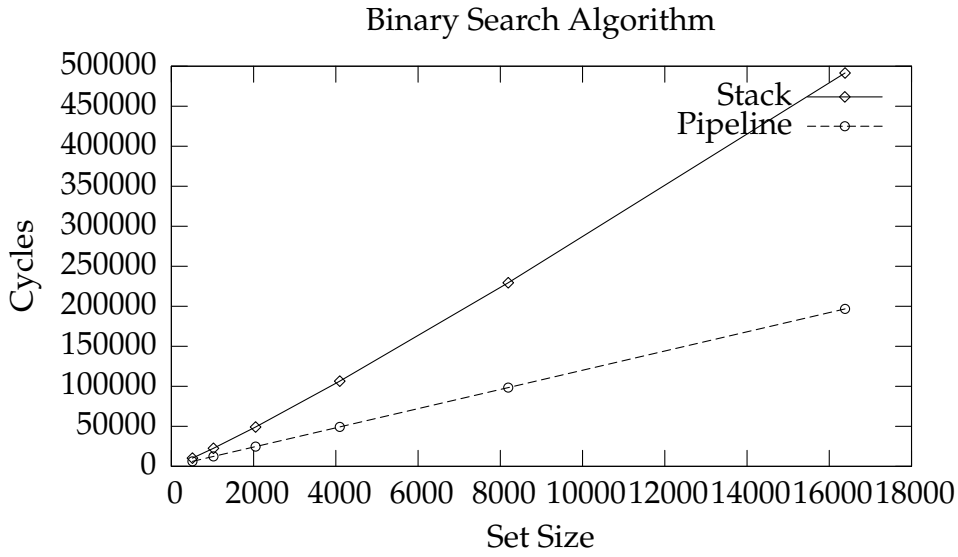


Figure 7.16: Binary Tree Search Results

Chapter 8

Conclusion

This chapter presents our conclusions from the research presented in this thesis, as well as presenting possible future research directions.

8.1 Conclusions and Comments

Mapping recursion to runtime reconfigurable systems, with the aim of introducing parallelism to the recursive function is possible by unrolling the function at run-time. Experimentation shows that for many well known $O(N \log(N))$ sequential algorithms (sections 7.2, 7.3, 7.4), parallelism can be extracted to produce hardware that runs in linear time.

Experimentation on a recursive algorithm that is not truly divide-and-conquer which runs in polynomial time has also demonstrated

a performance improvement(section 7.5). This function differed from the $O(N\log(N))$ divide-and-conquer algorithms in that the data size increased between recursive calls, thus necessitating more than one function unit to be allocated per level of the recursive tree. The experiments run on this algorithm also confirm the claim made in this thesis in regards to the effects of the calculated recursive growth width on the allocation of function units to a recursive function.

The techniques presented in this thesis for mapping recursion to reconfigurable hardware, such as the partitioning of the function into smaller functions, and analysis of the work done between instances of the function has made the mapping more feasible by ensuring that as much of the configured hardware is being utilised at any time, thus alleviating the limitations imposed by the finite area on an FPGA device. In particular, collapsing of the recursive tree as a result of analyzing the ratio of work between recursive calls has shown that it is possible to map many well structured recursive calls to reconfigurable hardware using a number of processors that is linearly related to the height of the tree. This eliminates the exponential explosion of processors if one is allocated to node in the tree as its depth increases.

We have employed methods for predicting the need for further unrolling to alleviate the latency introduced by runtime reconfiguration of the device. The prediction techniques presented in this thesis are

fairly basic. However they illustrate the effective of employing prediction mechanisms, and highlight the need of further research as prediction heuristics and techniques are possible through function profiling and statistic gathering by the controlling hardware. Advanced techniques may further increase the performance benefits of the methods outlined in this thesis.

The physical limitations imposed on the mappings described in this thesis, such as the logic density of the FPGA device as well as the memory bandwidth on the device, have been addressed in this thesis also. However the mapping technique described in this thesis still suffers from inefficient memory utilisation which can result in multiport RAM modules being allocated to a single function unit, thus reducing the amount of memory bandwidth available to the function units computing the result of the recursive function. Further research into the behaviour of recursive functions in respect to memory use, and more efficient uses of the memory on the FPGA device would greatly benefit the methods described in this thesis.

The simulation framework described in Appendix-A provides a foundation for further research into the simulation of runtime reconfigurable architectures. The simulator also provides a test bed for the simulation of designs that directly generate, or edit, the configuration bitstream. The development of this simulator itself acts as a starting

point for several avenues of research, including: the development of a better front end in the form of a GUI, the optimisation of the code produced with an aim to reduce size and the memory footprint. Research into the inclusion of techniques for more accurate timing of designs is another area of interest.

While the recursive growth width gives a good estimate for processor allocation when the recursive function being mapped is a function with predictable behaviour, the technique may not give a good estimate for functions with unpredictable behaviour. Research into load balancing techniques which ensure that all function units are being utilised may aid in such circumstances and provide a further increase to the performance of the mapping.

The mapping presented in this thesis shows that it is possible to build a circuit containing homogenous function units to compute the result of the recursive function. Further research into load balancing techniques that balance the work over a homogenous "sea" of function units may also be a stepping stone to using the methods described in this thesis for hardware that is not runtime reconfigurable.

The routing protocols presented in this thesis are designed for simplicity, not efficiency or to decrease latency. Further research into the network that is used for communication between function units could possibly reduce the communication latency, as well as reduce

the number of wires required for communication, thereby reducing the overheads of the network. Again further research into the network increases the feasibility of using the methods presented in this thesis on hardware that does not allow reconfiguration.

8.2 Future Research Directions

A major future body of research that will benefit the research in this thesis greatly is the mapping of recursive data structures to reconfigurable hardware. The case study in section 7.6, has shown how the methods presented in this thesis can be used to map recursive data structures to reconfigurable hardware, with each node in the mapped structure being able to execute the operations that are required. Further research into this area and its application to more complex optimisation problems would be of great benefit to scientific computing.

While the main thrust of the research presented in this thesis has been the mapping of recursive functions in procedural languages such as C and Java, research into the suitability the same methods being applied to functional languages such as Haskell or Miranda, and how this may impact previous research into functional language to hardware compilation [51, 33], provides a direction of interesting research.

Bibliography

[1] Handle-C language reference manual. *http://www.celoxica.com/*.

[2] RC203 Hardware and Installation Manual. *http://www.celoxica.com/*.

[3] SystemC v2.0.1 functional spec. *http://www.systemc.com/*.

[4] Miron Abramovici and Jose T. De Sousa. A sat solver using reconfigurable hardware and virtual logic. *J. Autom. Reason.*, 24(1-2):5–36, 2000.

[5] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[6] J. Arsac and Y. Kodratoff. Some Techniques for Recursion Removal from Recursive Functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, 1982.

[7] M. A. Auslander and H. R. Strong. Systematic recursion removal. *Communications of the ACM*, 21(2):127–134, 1978.

[8] J. W. Backus. The IBM 701 Speedcoding System. *Journal of the ACM*, 1(1):4–6, 1954.

[9] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[10] J. Barnes and P. Hut. A Hierarchical O($N \log N$) Force-Calculation Algorithm. *Nature*, 324:446–449, 1986.

[11] Richard Bellman. Dynamic Programming and Stochastic Control Processes. *Information and Control*, 1(3):228–239, September 1958.

[12] Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[13] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A self-reconfiguring platform. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *FPL*, volume 2778 of *Lecture Notes in Computer Science*, pages 565–574. Springer, 2003.

[14] Kiran Bondalapati. Parallelizing DSP nested loops on reconfigurable architectures using data context switching. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 273–276. ACM Press, 2001.

[15] Kiran Bondalapati and Viktor K. Prasanna. Mapping loops onto reconfigurable architectures. In Reiner W. Hartenstein and Andres Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 268–277. Springer-Verlag, Berlin, / 1998.

[16] Kiran Bondalapati and Viktor K. Prasanna. Dynamic precision management for loop computations on reconfigurable architectures. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 249–258, Los Alamitos, CA, 1999. IEEE Computer Society Press.

[17] Kiran Bondalapati and Viktor K. Prasanna. Loop pipelining and optimization for run time reconfiguration. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 906–915. Springer-Verlag, 2000.

[18] Altera Corporation. *Quartus*.

[19] Altera Corporation. *Stratix II Device Handbook, Volume 2, Chapter 2. TriMatrix Embedded Memory Blocks in Stratix II Devices*.

[20] Xilinx Corporation. Forge. *www.xilinx.com/ise/advanced/forge.htm*.

[21] Xilinx Corporation. Foundation Series Software. *www.xilinx.com/*.

[22] Xilinx Corporation. *Virtex-4 Family Overview*.

[23] Xilinx Corporation. Virtex 4 Platform: Complete Data Sheet.

[24] Xilinx Corporation. Virtex-II Platform: Complete Data Sheet.

[25] Xilinx Corporation. Virtex-II Pro Platform: Complete Data Sheet.

[26] Xilinx Corporation. XAPP290 Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations.

[27] Xilinx Corporation. XDL (Xilinx Design Language. 1998.

[28] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. pages 46–93, 1997.

[29] H. ElGindy, A. K. Somani, H. Schroder, H. Schmeck, and A. Spray. Rmb – a reconfigurable multiple bus network. In *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*, page 108. IEEE Computer Society, 1996.

[30] Jong eun Lee, Kiyoung Choi, and Nikil D. Dutt. An algorithm for mapping loops onto coarse-grained reconfigurable architectures.

In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 183–188. ACM Press, 2003.

[31] W. Fornaciari and V. Piuri. General methodologies to virtualize FPGAs in HW/SW systems. *Circuits and Systems*, pages 90–93, 1998.

[32] W. Fornaciari and V. Piuri. Virtual FPGAs: Some Steps Behind the Physical Barriers. *Reconfigurable Architectures Workshop*, 1998.

[33] Simon Frankau and Alan Mycroft. Stream processing hardware from functional language specifications. In *HICSS*, page 278, 2003.

[34] M.R. Garey and D.S Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman, 1997.

[35] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 49. IEEE Computer Society, 2000.

[36] S. A. Guccione and D. Levi. XBI: A Java-based interface to FPGA hardware. In J. Schewel, editor, *Configurable Computing Technol-*

ogy and its uses in High Performance Computing, DSP and Systems Engineering, pages 97–102, Bellingham, Washington, November 1998. SPIE – The International Society for Optical Engineering.

[37] Frank Hannig, Hritam Dutta, and Jürgen Teich. Mapping of Regular Nested Loop Programs to Coarse-grained Reconfigurable Arrays – Constraints and Methodology. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, U.S.A., April 2004. IEEE Computer Society.

[38] C. A. R. Hoare. Algorithm 63: partition. *Commun. ACM*, 4(7):321, 1961.

[39] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961.

[40] E. Hopper. *Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods*. PhD thesis, Cardiff University, UK, 2000.

[41] Jack S.N. Jean, Karen Tomko, Vikram Yavagal, Jignesh Shah, and Robert Cook. Dynamic reconfiguration to support concurrent applications. *IEEE Transactions on Computers*, 48:591–602, June 1999.

[42] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Pro-*

ceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[43] Douglas Lea. libg++, the GNU C++ library. In USENIX Association, editor, *USENIX proceedings: C++ Conference, Denver, CO, October 17–21, 1988*, pages 243–256, Berkeley, CA, USA, October 1988. USENIX Association.

[44] Neal Lesh, Joe Marks, A. McMahon, and Michael Mitzenmacher. Exhaustive approaches to 2d rectangular perfect packings. *Information Processing Letters*, 90(1):7–14, 2004.

[45] L. Levinson, R. Manner, M. Sessler, and H. Simmler. Preemptive Multitasking on FPGAs. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 301. IEEE Computer Society, 2000.

[46] Yanhong A. Liu and Scott D. Stoller. From recursion to iteration: what are the optimizations? In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 73–82. ACM Press, 1999.

[47] R. Maestre, F. J. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida, and M. Fernandez. Kernel scheduling in reconfigurable computing. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 21. ACM Press, 1999.

[48] Tsutomu Maruyama and Tsutomu Hoshino. A C to HDL Compiler for Pipeline Processing on FPGAs. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 101. IEEE Computer Society, 2000.

[49] Jayadev Misra. Powerlist: a structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, 1994.

[50] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[51] Alan Mycroft and Richard Sharp. Hardware/software co-design using functional languages. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 236–251. Springer, 2001.

[52] Juanjo Noguera and Rosa M. Badia. Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. *Transactions. on Embedded Computing Systems*, 3(2):385–406, 2004.

[53] J.A. Orenstein, T.H. Merret, and L. Devroye. Linear sorting with O(log N) processors. *BIT*, 23:170–180, 1983.

[54] Cameron Patterson. High performance des encryption in virtex(tm) fpgas using jbits(tm). In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 113, Washington, DC, USA, 2000. IEEE Computer Society.

[55] A. J. Perlis and K. Samelson. Preliminary report: international algebraic language. *Communications of the ACM*, 1(12):8–22, 1958.

[56] Alexandra Poetter, Jesse Hunter, Cameron Patterson, Peter M. Athanas, Brent E. Nelson, and Neil Steiner. JHDLBits: The merging of two worlds. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *FPL*, volume 3203 of *Lecture Notes in Computer Science*, pages 414–423. Springer, 2004.

[57] Herman H. Schmit, Srihari Cadambi, Matthew Moe, and Seth C. Goldstein. Pipeline Reconfigurable FPGAs. *J. VLSI Signal Process. Syst.*, 24(2-3):129–146, 2000.

[58] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[59] Henry Styles, David Barrie Thomas, and Wayne Luk. Pipelining Designs with Loop-Carried Dependencies. In *International Conference on Field-Programmable Technology*. ACM Press, 2004.

[60] Jesus Tabero, Julio Septien, Hortensia Mecha, and Daniel Mozos. A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management. In Serge Vernalde Jurgen Becker, Marco Platzner, editor, *Field-Programmable Logic and Applications*, pages 241–250. Springer-Verlag, Heidelberg, 2004.

[61] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Survey*, 18(4):365–396, 1986.

[62] Markus Weinhardt and Wayne Luk. Pipeline vectorization for reconfigurable systems. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 52. IEEE Computer Society, 1999.

[63] Steve Young, Peter Alfke, Colm Fewer, Scott McMillan, Brandon Blodget, and Delon Levi. A high i/o reconfigurable crossbar switch. In *FCCM*, pages 3–10. IEEE Computer Society, 2003.

Appendix A

Simulator

This chapter describes a device level simulator that was built to measure the effects of the latncy introduced by RTR on the performance of the mapping method described in this thesis.

The simulator is not fixed to one device. The simulator is build from a plain text description of an FPGA device. This plain-text description matches the output from the `xdl` utility that is provided with Xilinx's ISE Software package [21]. From this plain-text description source code for a simulator and bitstream file generator is created. Providing the source code allows for a generic skeleton to be augmented to suit the intended architecture.

As the simulator provides open code the format of the bitsttream is known. This allows for the simulation of components on the FPGA that directly edit the bistream which is used to relocate modules on

the device.

This chapted will begin by providing a description of the plain-text file in section A.1, before describing the method used to generate the source code for the simulator from the plain-text description in section A.2. The operation of the simulator is then described in section A.3. The bitstream generator and its implementation is then described in section A.4. The user interface is then described in section A.5.

A.1 Plain Text Description

The plain-text description is a hierarchical description of the components on the device. The top level of the hierarchy contains a listing of tiles, and a listing of primitive definitions. The hierarchy of components under *tiles*, will be detailed first in section A.1.1, before a description of *primitive definitions*, is given in section A.1.2. The method that connectivity is described will then be presented in section A.1.3.

A.1.1 Tiles

A tile is a regular collection of resources that is placed in multiple locations on the FPGA. Tiles are typed; tiles of the same type contain the same logic resources and the same routing resources, with the excep-

tion of bordering tiles which may contain a small number of routing resources that differ from tiles of the same type not on the edge of the FPGA.

The tile container containts routing and logic resources. The routing resources include configurable routing(pips) and static inter-tile routing which connects pins in the tile to pins in other tiles.

The logic resources in a tile are termed *primitive sites*. A *primitive site* contains a typed logic resource which corresponds to a *primitive definition* which will be described in section A.1.2, and a listing of the static connections from pins on the logic resource to pins in the tile.

The plain text description we use for input to produce the simulator matches exactly the output of the `xdl` tool provided by *Xilinx's ISE* software which gives a description of a Xilinx device. This description names all tiles and their type before giving a listing of the logic resources within each tile, the configurable routing resources in each tile and all the inter-tile connectivity that has sinks or sources in this tile. We will give a brief overview of the syntax of this text description file as an aid in describing how the simulator is built.

A condensed grammar for a tile and all the elements hierarchicaly beneath it is shown in figure A.1.

While a tile's description does not contain the logic resources directly it contains the name of a container which is named a primitive

$$\text{tile} \rightarrow (\text{prim. site})^*(\text{inter-tile conn})^*(\text{pips})^*$$
$$\text{prim. site} \rightarrow (\textit{site name})(\text{pin wires})^*$$

Figure A.1: Tile grammar

Device

Tile

Primitive Site

Element

Figure A.2: Hierarchical organisation of FPGA resources

site. The contents of the primitive sites are defined in the plain-text file in what is termed a primitive definition.

A.1.2 Primitive definitions

Primitive definitions are containters that contain atomic logic elements named *primitive elements* and static routing between the elements.

The gramar for *primitive definitions* is shown in figure A.3.

prim. def. → (*site name*)(pin)*(prim. element)*
prim. element → (element name)(element data)
element data → (elem. pin)*(config)?(conns)*

Figure A.3: Primitive site

The *element data* atom describes properties of the *primitive element* it belongs too. It names the pins that this pin reads and writes to, noting their direction, the connections between these pins and pins in other *primitive elements* in the current *primitive definition*, and if the *config* atom exists, all possible configuration states of this *primitive element*. If the *config* atom does not exist the *primitive element* corresponds to static logic.

The behaviour of the *primitive element* can often be inferred automatically from either the name of the *primitive element*, or the contents of the *config* atom.

Table A.1 shows the percentage of elements whose behaviour cannot be determined automatically for two Xilinx devices. While the numbers do look large it should be noted that the majority of the un-

known elements were to do with reconfiguration resources such as ICAP and BSCAN. The majority of IO-Block and Slice logic resources were implemented automatically.

Device	Number of Unique Elements	% of Unique Elements
xcv50bg256	68	15.77
xc2v1000bg456	127	14.53

Table A.1: Number and percentage of elements with unknown behaviour in various simulated devices.

A.1.3 Connectivity

Two types of connectivity exist on FPGA devices: inter-tile and intra-tile connectivity.

Inter-tile connectivity is described by the grammar shown in figure A.4. For every pin the number of inter-tile connections it has is listed followed by the pins it is connected to in other tiles.

$$\text{inter-tile conn.} \rightarrow (\textit{wire name})(\textit{no. conns.})(\text{conn})^*$$
$$\text{conn} \rightarrow (\textit{tile name})(\textit{pin name})$$

Figure A.4: Inter-tile Grammar

Intra-tile connectivity is described by the grammar shown in figure A.5. The relationship between *pin1* and *pin2* is a $N - N$ relationship, but all pairs only occur once.

pip → (*tile name*)(*pin1*)(dir.)(*pin2*)

Figure A.5: Intra-tile Grammar

Intra-tile connectivity is configurable routing. The connection between two pins can be set to on or off and this is set when the device is configured by the bitstream file. This reconfigurable connection is termed a pip (programmable interconnection point). In this paper inter-tile connectivity is considered static as the connection between pins in tiles is always present. Thus the control of these wires is dependent on the configuration state of the pips at either end of the wire.

A.2 Simulator Generation

Generating code for each individual element and tile on an FPGA results in code that is too large for compilation on many systems. To reduce the size of the code that is generated the high amount repetition of resources on FPGA devices is exploited.

To exploit the repetition of resources on FPGA devices the hierarchical grouping of these resources that can be ovserved in the grammars in figures A.1 and A.3, are reproduced in the code. A description of how code is produced for each level of the hierarchy is described in this section.

A.2.1 Elements

Elements are created to control the operation of the atomic logic resources on the FPGA. These resources include boolean gates, multiplexers, LUTs, flip flops, ram modules and IO pads.

These source code that implements each element is placed into a library, with elements that are common between various different primitive sites only having a single entry in the library. When creating the source code for an element the library is checked to see if code that matches this element already exists. If no source code exists in the library code to implement the element is added to the library. During this process a map is maintained between each element and the function in the library controlling its behaviour.

A.2.2 Primitive Sites

Primitive sites are maintined in a library in the same way elements are. The source code for each primitive site must call code in the element library so that the elements in it operate on input data and generate output data. The map created while generating the element library is used to link the elements in the site with the appropriate function in the library. The source code for primitive sites is created by parsing the primitive definitions in the output file. As each primitive definition is unique with no repetition a map is not maintained at

this level of the hierarchy.

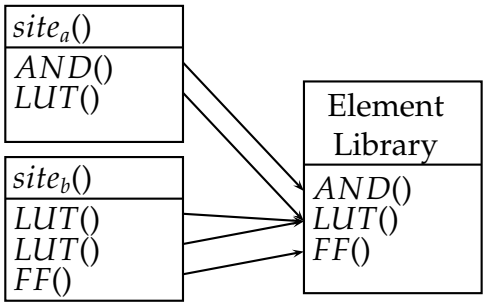


Figure A.6: Library hierarchy of primitive sites and their elements

A.2.3 Tile Implementation

Tiles are also implemented in a library in the same way primitive sites are. Intra-tile communication is implemented in the tile also. As this is configurable it is necessary that all communication be indexed. To facilitate indexing each pin is added to an AVL tree and is indexed by the name of the pin.

Tiles contain two sets of resources: logic and routing. Tiles exist on the FPGA device with the same logic resources but different routing resources. To reduce the code that is produced two libraries are maintained: one for logic and one for routing. Each resource is compared to each library individually, with a map being created between the tile name and the function in both libraries.

It is worth noting no comparison is requried for logic resources as

the tiles are typed, with all tiles of the same type containing equivalent logic resources.

A.2.4 Device Implementation

The tile map obtained while generating the tile library is used for the implementation of the final device. This level also controls inter-tile routing. Inter-tile routing is controlled by indexing the relevant pin in each tile using the AVL tree in the tile structure and creating the appropriate connection.

A.3 Operation

Elements on an FPGA contain different timing delays, and therefore in a period of time will operate a different number of times. The method used to simulate this timing model is described in this section.

A.3.1 Timing Model(s)

The timing model is parameterisable and is specified by the user. It is loaded at runtime and therefore allows the user to operate a described device with different timing delays without the need to reproduce the simulator. The timing model is described in a file that contains a series of named sets. The sets contain the names of elements the chip

contains, followed by the timing delay that is to be associated to every element in that group. This allows for a reduction in the amount of data a user is required to input.

Elements with a delay that varies depending on configuration such as pips are not supported in this simulator with all timing delays being static. The only element the author of this thesis knows of on FPGAs that exhibit this property also happens to be the most common element, the pip. Pips timing delay increases as the number of connections out from it increase due to the increase in capacitance that occurs with a higher fan out.

A.3.2 Functionality

The system functions around an event based model. Initially all configured logic elements and the function that describes their behaviour are added to a min-heap using their timing delay as the key. During operation the top value in the heap is popped off the heap and operated on. A depth first search is then done on the heap and the timing delay of the element that just was popped is subtracted from the key at every node of the heap before it is placed back into the heap. This preserves the shape of the heap but allows a event-triggered type structure to be implemented, which is more desirable than implementing a finishing time event queue, due to the limitations on the size of an

integer on a machine. To reduce the number of elements that require this subtraction the size of the heap is reduced by placing all elements with the same key into the same node of the heap. With the repetition of logic elements with the same timing delay such as pins, wires and many logic resources due to duplication this greatly reduces the size of the heap and thus the cost of the depth first search.

The process for networking resources is different. When a pin is set by an I/O block or a logic element, that pin is added to the min-heap with a temporary flag set. When this pin fires we repeat the same process as we did for a logic element, but do not place the element back into the heap. Instead the direct sinks for this pin, be it wires or other pins in the case of pips, are added on the heap with a temporary flag set. This allows us to further reduce the size of the heap and still have signals propagating through the system without the need to trace networks at runtime.

A.4 Configuration Stream

A.4.1 Bitstream Format

The bitstream file that is created is order in a way such that all the configuration data for each tile is organised in a contiguous block in the bitstream file, and the order of the tiles in the bitstream file matches

their order in the plain-text description of the device.

The simulator considers each tile to contain an ordered configuration list. This list is made up of data to describe the configuration data of all configurable primitive elements in that tile and all pips in that tile. The ordering of this configuration list is determined by the order in which the configurable elements occur in the listing for the tile. All tiles of the same type share the same ordered list.

Currently the bitstream is arranged such that the data for each tile is in contiguous blocks in the bitstream. This however can be changed to simulate different configuration architectures. It should be noted that when the chip listing is parsed to obtain these lists the size of the list contained in each tile is computed. This is used to obtain an index into the bitstream file for the next tile.

A.4.2 Logic Element configuration

When processing a configurable primitive element the `config` (figure A.3) option is processed to extract all the possible states of the primitive element and determine a numerical enumeration that can be mapped to these states. This can be done due to the configuration line for a primitive element containing all the possible states of that element. Certain elements require more information such as programmable LUTs operating in LUT mode, which require a boolean

equation based on the inputs as well as information on the operational state of the LUT. This process typically requires one byte of configuration data per element, with the exception of the LUTs which require three bytes of configuration data.

When this data is being created and added to the configuration list a map is kept which returns an index into the configuration list based on the name of the primitive definition and the primitive element within it. This is later used so that the configuration data for this element can be reported and edited efficiently.

A.4.3 Pip configuration

A similar process is followed for the pips with the pins that each pip can be connected to being assigned to a numerical enumeration. These mapped values in the enumeration are used as indexes into the a bitmap, which contains a bit for each pin the pip can be connected too, as shown in figure A.7. A constant number of bytes are allotted to describe each pip. This constant number is determined by finding the pip that contains the maximum number of possible connections. This constant is determined when parsing the chip listing.

In a similar way to the primitive elements a map is created between the name of the pip and the index into the configuration list of it's configuration data.

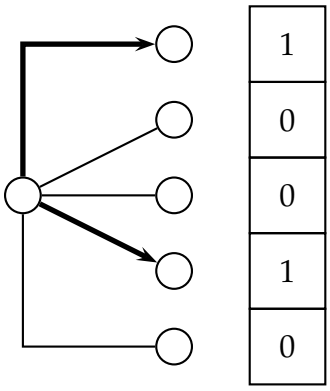


Figure A.7: A pip connection with 5 possible connections, two of which are turned on.

A.5 User Interface

A.5.1 Simulator Interface

The device simulator currently has a text based command-line interface. This interface allows for basic operations such as bypassing on-chip reconfiguration methods to load the bitstream, executing the functions to operate data on the heap and the loading and reading of pins and wires from and into files.

The decision to provide a method for bypassing on-chip reconfiguration methods was made to allow for easier device specification without the need to design reconfiguration mechanisms. The user is free to choose between this method or implementing their own model. It should be noted that the reconfiguration logic in the Virtex devices is not automatically derived from the chip listing, however the user can

implement it using the existing reconfiguration infrastructure.

A.5.2 Bitstream Generation

A bitstream is extracted from a netlist by first converting the `.ncd` format file into a Xilinx Design Language (`.xdl`) [27] format file. The `.xdl` file is a text description of a netlist that is easy to read. It consists of two types: instances and nets.

An instance describes the state of a configurable element. Figure A.8 shows a portion of the text description of a slice. This instance named `res_1` is a slice that has been placed into slice `CLB_R16C5.S0` in tile `R16C5`. It then describes the configuration state of each element in this site, which is to be set by the bitstream generator. It should be noted that every element is described in the `.xdl` file, with elements that are not used being set to `#OFF`.

Using the maps that have been derived the index of this instance and every element within it in the bitstream file can be determined. The value that can be gained from the enumeration derived earlier that describes all possible configuration states of an element is then written into this index of the file. This becomes slightly more complicated for configured LUTs where the content of the LUT is created by evaluating the expression described by the equation for the LUT for every value from 0 to 15.

```
inst "res_1" "SLICE" ,
   placed R16C5 CLB_R16C5.S0,
   cfg "CYSELF::#OFF CYSELG::#OFF
   CKINV::#OFF COUTUSED::#OFF YUSED::0
   F::#OFF: G::#LUT:D=(A2@A4)
   RAMCONFIG::#OFF
```

Figure A.8: Example configurable element.

A net describes the configurable portions of an entire routed network. Figure A.9 shows a description of the network between pin `I` in instance `val2` and `O` in `carry`. We can see three pips configured, one of which creates a network between `BOT_I2` in tile `BC4` that is configured to load pin `BOT_N1`.

```
net "val2_c" ,
  outpin "val2"             I,
  inpin  "carry"            O,
  pip BC4 BOT_I2 -> BOT_N1 ,
  pip BC5 BOT_N21 -> BOT_N_BUF21 ,
  pip BC5 BOT_N_BUF21 -> BOT_O2 ,
```

Figure A.9: Example configurable network.

A.5.3 Runtime Reconfiguration

Runtime reconfiguration is implemented by the command-line interface described earlier. It can also be implemented in the configuration mechanisms of the device by the user. In the case of the Virtex devices this corresponds to various elements such as the ICAP and boundary

scan elements.

In the case of partial reconfiguration, bitstreams are created with a flag stating they are partial bitstreams. Before the configuration for each tile the name of the tile is encoded into the bitstream. This presents a fine grained partial reconfiguration, however if the bitstreams that are generated are limited to a column based approach with each tile in the column written into the file coarse grained partial reconfiguration schemes such as those present in Virtex devices can be realised.

Partial bitstreams can be generated by the bitstream generator in two ways. The xor operation that is present in Xilinx's bitgen software is implemented, as well as a mechanism that only writes out configuration data for tiles that have configuration data in the netlist. The xor operation finds the difference between two bitstreams and generates the minimal configuration bitstream to effect these changes taking into account the granularity of reconfiguration.

It should be noted that the behaviour of elements being reconfigured during the process of runtime reconfiguration is undefined. We have implemented a predictable behaviour, where both the operation of a logic element and reconfiguration are atomic. Hence the output of an element being reconfigured is dependent on which of the two events occurs first.