THE UNIVERSITY
of EDINBURGH

Efficient Runtime Placement Management for High Performance and Reliability in COTS FPGAs

By

Godwin Enemali

A thesis submitted in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

The University of Edinburgh

March 2019

Declaration

I hereby declare that this thesis was composed and originated entirely by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualifications.

Godwin Enemali

To Our Lady Seat of Divine Wisdom

Acknowledgements

I would like to express my immense gratitude to my supervisor, Professor Tughrul Arslan for his enormous support and excellent supervision of my PhD work. His vast knowledge and wide experience are invaluable to the completion of my research work and thesis. In addition to the many opportunities he offered me during my time in Edinburgh, his availability, patience, gentle guidance and kindness were not only helpful to me on the PhD programme, but have inspired me to be a better person.

My sincere appreciation also goes to my second supervisor, Dr Alister Hamilton for introducing me to FPGAs during my MSc programme and his advice throughout my PhD programme. I am also grateful for the teaching experience I gathered from the courses I had the opportunity of tutoring under his guidance. His reminders to finish up the PhD is also very much appreciated.

I also thank members of the research group in which I carried out my research. First, I would like to thank a close colleague Adewale Adetomi for his treasured support on the PhD programme and for sharing many of his experiences on FPGA designs with me. I would also like to thank other members of the group: Riza, Amalina, Yichen, Fengzhou, Yue, Aliyu and Imran for the great time at the SMC Portacabin.

In a very special way, I would like to thank members of my family who are very precious to me and remained my greatest heroes. I cannot thank my parents, Mr and Mrs Isaac Enemali, enough for all they have done for me and for their love. And to you, my siblings, my best friends and highest human support: Dr Felix, Fr Wilfred, Joy, Joseph, Peter, Paul and Emmanuel, may God bless you.

I would also like to thank members of Catholic community at Sacred Heart Church, St Mary Cathedral and Our lady Cause of Our Joy Preasidium of the Legion Mary for the sense of community and worship of God I had among you. I cannot forget my dear friend Johanna for her immense kindness and friendship. May God reward you.

Most importantly, I am grateful to God for all His blessings which I cannot number and the opportunity to have studied in Edinburgh.

Abstract

Designing high-performance, fault-tolerant multisensory electronic systems for hostile environments such as nuclear plants and outer space within the constraints of cost, power and flexibility is challenging. Issues such as ionizing radiation, extreme temperature and ageing can lead to faults in the electronics of these systems. In addition, the remote nature of these environments demands a level of flexibility and autonomy in their operations. The standard practice of using specially hardened electronic devices for such systems is not only very expensive but also has limited flexibility.

This thesis proposes novel techniques that promote the use of Commercial Off-The-Shelf (COTS) reconfigurable devices to meet the challenges of high-performance systems for hostile environments. Reconfigurable hardware such as Field Programmable Gate Arrays (FPGA) have a unique combination of flexibility and high performance. The flexibility offered through features such as dynamic partial reconfiguration (DPR) can be harnessed not only to achieve cost-effective designs as a smaller area can be used to execute multiple tasks, but also to improve the reliability of a system as a circuit on one portion of the device can be physically relocated to another portion in the case of fault occurrence. However, to harness these potentials for high performance and reliability in a cost-effective manner, novel runtime management tools are required. Most runtime support tools for reconfigurable devices are based on ideal models which do not adequately consider the limitations of realistic FPGAs, in particular modern FPGAs which are increasingly heterogeneous. Specifically, these tools lack efficient mechanisms for ensuring a high utilization of FPGA resources, including the FPGA area and the configuration port and clocking resources, in a reliable manner.

To ensure high utilization of reconfigurable device area, placement management is a key aspect of these tools. This thesis presents novel techniques for the management of hardware task placement on COTS reconfigurable devices for high performance and reliability. To this end, it addresses design-time issues that affect efficient hardware task placement, with a focus on reliability. It also presents techniques to

maximize the utilization of the FPGA area in runtime, including techniques to minimize fragmentation. Fragmentation leads to the creation of unusable areas due to dynamic placement of tasks and the heterogeneity of the resources on the chip.

Moreover, this thesis also presents an efficient task reuse mechanism to improve the availability of the internal configuration infrastructure of the FPGA for critical responsibilities like error mitigation. The task reuse scheme, unlike previous approaches, also improves the utilization of the chip area by offering defragmentation.

Task relocation, which involves changing the physical location of circuits is a technique for error mitigation and high performance. Hence, this thesis also provides a functionality-based relocation mechanism for improving the number of locations to which tasks can be relocated on heterogeneous FPGAs. As tasks are relocated, clock networks need to be routed to them. As such, a reliability-aware technique of clock network routing to tasks after placement is also proposed.

Finally, this thesis offers a prototype implementation and characterization of a placement management system (PMS) which is an integration of the aforementioned techniques. The performance of most of the proposed techniques are tested using data processing tasks of a NASA JPL spectrometer application. The results show that the proposed techniques have potentials to improve the reliability and performance of applications in hostile environment compared to state-of-the-art techniques. The task optimization technique presented leads to better capacity to circumvent permanent faults on COTS FPGAs compared to state-of-the-art approaches (48.6% more errors were circumvented for the JPL spectrometer application). The proposed task reuse scheme leads to approximately 29% saving in the amount of configuration time. This frees up the internal configuration interface for more error mitigation operations. In addition, the proposed PMS has a worst-case latency of less than 50% of that of state-of-the-art runtime placement systems, while maintaining the same level of placement quality and resource overhead.

Content

List of Figures

List of Tables

List of Abbreviations

AC	Architecture Checker
AQE	Allocator Quality Evaluator
ASIC	Application Specific Integrated Circuit
ATQ	Arriving Tasks Queue
BF	Best Fit
BUFG	Global clock buffer
BUFH	Horizontal Clock Buffer
BUFIO	I/O clock buffer
BUFMR	Multi-Regional Clock Buffers
BUFR	Regional Clock Buffers
CLB	Configurable Logic Block
CMEM	Configuration Memory
COTS	Commercial Off-The-Shelf
DBR	Direct Bitstream Relocation
DPR	Dynamic Partial Reconfiguration
DRAM	Distributed RAM
DRC	Design Rule Check
DSP	Digital Signal Processor
DVF	Device Vulnerability Factor
EAC	Empty Area Compaction
EADU	Empty Area Descriptor Updater
ECC	Error Correcting Code
EDF	Earliest Deadline First
EUAS	Expanding the Unusable Area Strategy
FAReP	Fragmentation-Aware Replacement Policy
FBR	Functionality-Based Relocation
FC	Fragmentation Coefficient
FPGA	Field Programmable Gate Arrays
FSML	FPGA State Matrix and Layout memory
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
IB	Init Buffer
ICAP	Internal Configuration Access Port
IDB	Input Data Buffer
IDE	Integrated Design Environment
ILA	Integrated Logic Analyzer
LPR	Least Probability of Recurrence
LRU	Least Recently Used
LSB	Least Significant Bit
LUT	Look Up Table
MER	Maximum Empty Rectangle
NTP	New Task Placement
ODB	Output Data Buffer
PIP	Programmable Interconnection Points

PMS	Placement Management System
PRC	Partial Reconfiguration Controller
PTP	Pending Task Placement
FAR	Frame Address Register
RBS	Reuse-Based Scheduling
RER	Reconfiguration-to-Execution Ratio
RM	Reconfigurable Module
ROS	Reconfigurable Operating System
RUP	Runtime Utilization Probability
SA	Simulated Annealing
SEM	Soft Error Mitigation
SUP	Static Utilization Probability
TMR	Triple Modular Redundancy
TSB	Task State Buffer
VBS	Virtual Bitstream
VIO	Virtual Input Output
VLS	Vertex List Set
XDC	Xilinx Design Constraints

Chapter 1: Introduction

Future space systems, robots for nuclear plants and other critical applications as well as general embedded systems have a wide range of constraints which their electronics need to meet. These constraints including reliability, low power, flexibility, area constraints and cost have continued to drive the growth of the electronics industry. This growth has been unprecedented, finding applications in varied aspects of human life, and changing the way we live, interact and work over the last few decades. The growth of electronic components in many fields, including aerospace, is projected not only to be sustained but anticipated to witness the highest growth among other component types over the next five years [1]. The sustenance of the growth in the electronics industry is largely driven by the desire for faster computing capability within these constraints. To meet these constraints, many research efforts are targeted at new computing architectures and hardware platforms.

The main computing hardware platforms including processors, reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) provides varying degree of solution to the constraints above [2], [3]. A significant amount of computing is done on processor platforms, mostly general purpose processors (GPPs). Processors have the highest level of flexibility among the computing platforms identified above, thus making them appliable in a wide range of applications. In addition, the aboundant tool support for GPPs, mostly in the form of operating systems (OS), compilers, libraries and the relative ease with which programmes targeted at GPPs can be written have helped to increase the productivity of GPP-based computers. It remains the most dominant computing platform by far as every server needs them and numberless applications are written to run on them [4]. However, alternative processing platforms are increasingly gaining attention due to new applications domains such as artificial intelligence. Other requirements of electronic equipments especially reliability for systems in hostile environents, low power and high performance are also major reasons for developers to consider other computing platforms.

GPPs are fundamentally sequential elements which are based on a repetitive execution of fetch-decode-execute cycles of a stored programe. Thus improving the speed of processing is achieved either by increasing clock frequency or by using multiple processors. Although modern fabrication technology makes it possible to reduce the dimention of transistors, which in theory could mean that they could be driven at higher freqencies, there is a limitation as to the maximum operating frequency to which transisitors can operate because of the problem of heat extraction. This leads to the so called power wall [5]. On the otherhand, while using multicore processors means that multiple instructions could be executed in parallel, their performance is dependent on a variety of factors. These include: the ability to parallelise the application at hand effectively [6], efficient management of the possibility of increasing percentages of dark or dim silicon and the effect of Amdel's law on multicore architectures and workloads [7].

Due to their flexibility, GPPs are generally targeted at general applications, and may not provide sufficient low power, high performance and adequate reliability especially for hostile environments. Many reseachers foresee a more heterogenous computing platform to meet these challenges, depending on the application [7]. The challenge of reliability of the hardware design is an important one in the context of this thesis. Thus, it is important to state that GPPs do not have an adequate means of managing damage to the underlying silicon on which they are built.

Application Specific Integrated Circuits (ASICs) provide certain advantages in certain domains compared to GPPs. Custom ASICs have much higher throughput and low power compared to processors as they are fine-tunned to the targeted application. Example of ASICs include Graphics Processing Units (GPUs) which are optimized for targeted applications in the domain of image procesing, and Digital Signal Processors (DSPs) which target a wide variety of signal processing applications. In practice, these ASICs often operate in conjuction with GPPs. In a sharp constrast to GPPs, ASICs do not have a high level of flexibility. Thus, the set of applications they handle is significantly less than GPPs.

While ASICs have inherent potentials to have low power and high performance, their lack of flexibility is a huge disadvantage. Due to this lack of flexibility and the increasing cost of fabrication process, ASICs are very expensive and are mostly deployed for large scale applications so that they can benefit from the economy of scale. In addition, like GPPs, their capability to deal with damage to their underlying hardware is also limited. For critical applications, special hardening techniques are use to improve the relaibility of ASICs. This further adds to the cost of an already expensive technology.

Reconfigurable hardware occupies a middle ground, both interms of flexibility and performance, between GPPs and ASICs. Reconfigurable hardware such as FPGAs have a unique combination of flexibility and high performance. Although their flexibility is lower than GPPs and their performance and power efficiency lower than ASICs, their unique combination of these features make them applicable to various applications in a way which neither GPPs nor ASICs can. With FPGAs, an application can be easily updated even from a remote location without having to take down the system.

Reconfigurable hardware has stood in between conventional processors and ASICs for decades. However, their degree of flexibility, which is enabled by their reconfiguration capability, has continually improved. An example of this is the introduction of partial reconfiguration in FPGAs. This has further positioned them closer to processors in terms of flexibility and versatility. Partial reconfiguration allows the behaviour (of part) of a chip to be redefined without interfering with the normal operation of the other parts of the chip. In addition, this can be done in runtime while the other applications continue to operate. This is called Dynamic Partial Reconfiguration (DPR). Hence, it is possible to swap hardware *tasks* (circuits) in and out of the chip, and effectively turning it into a platform which can both be time and area-shared among multiple tasks [8], [9] in runtime.

FPGAs were traditionally used for prototyping ASIC designs; however, their improved flexibility and performance have contributed to making them being harnessed for runtime applications just like GPPs and ASICs. Big player like amazon

and Intel have included FPGAs in their products and services. Amazon uses FPGAs for cloud computing [10], [11], [12] while intel has become a key player in the FPGAs industry with their purchase of Altera [13]. These points to the growing potentials of FPGAs in the electronics industry.

One of the special interests in the context this thesis is that the flexibility offered by Commercial Off-The-Shelf (COTS) FPGAs can be applied to mitigating both transient and permanent faults in critical applications [14] while still maintaining high performance. Techniques have been presented which correct transient faults by *reconfiguration* and permanent faults by *relocation* [8]. Relocation to circumvent parmanet damage is an advantage of FPGAs that GPPs and AISCs are yet to provide an equivalence for in their current architecture.

In terms of the application design process, GPPs have remained the most attractive platform, compared to ASICs and FPGAs. The long years of investment in the development of design tools targeted at GPPs as well as the wealth of knowledge application developers have amassed in the use of these tools are important reasons for the ease of developing applications for processors. Tools for developing applications for GPPs are far more developed than those for FPGAs. This is a reason why many application developers prefer GPPs for some applications which could have benefitted more from the capabilities of FPGAs. In addition to this, productivity, skill and as well as price are other reasons why GPPs are preferred to FPGAs. GPPs design tools are well developed with numerous operating system support, compilers and libraries.

Although FPGAs tools are nowhere close to the GPPs tools, there have been rapid developmental trends in the tools for FPGA platforms. The number of design automation tools have consistently increased in recent years, with support both from the industry and the academia. This is evident with respect to the industry as Xilinx (who own the largest market share of the FPGA market [15]) has continually invested in design tool chain. An example is the developments in the Vivado Integrated Design Environment (IDE), as well as continually increasing their library of IPs which users can simply integrate into designs. In addition to these, there is a growing research to support High Level Synthesis (HLS) by Xilinx chain of tools [16] with commercial

products already available. Another example of industry growing tool support is the recent Intel® Quartus® Prime software [17] for developing application targeted at Intel FPGAs.

In addition to the growing industry tools, many academic efforts aim to provide resources that facilitate the use of FPGAs, both at design time and runtime [18]. Many of these take the industrial devices beyond the traditional support offered by the industry tools, and some aim to improve the versatility of FPGAs by supporting COTS FPGAs for critical applications [19]. Tools for COTS FPGAs for critical application has the potential of significantly lowering the cost of these critical applications in contrast to developing special parts for them.

The idea of operating system support for FPGAs was first envisioned over 2 decades ago [20] and it is generally agreed that reconfigurable operating system (ROS) would revolutionize the entire FPGA industry. However, a lot of research, both on the FPGA hardware platforms themselves as well as the design and implementation of algorithms to harness the potentials of the platform, is required to actualize the dream. State-of-the-art ROSes have not yet reached a stage where reconfigurable computing can appeal to many application designers even when it is obvious that they have better performance and/or cost benefits than GPPs and custom ASICs.

More research effort is required to develop efficient and user-friendly ROS for FPGA platforms. There are many fundamental aspects of an ROS that need to be addressed to make ROS more popular. In particular, the lack of efficient and generic runtime tools to manage the placement of hardware circuits (called hardware tasks) on the FPGA limits the efficiency and adoption of ROS in many scenarios. The main aim of this thesis is to develop a runtime placement management system for ROS targeting high performance and reliability. The thesis explores a wide range of issues relating to runtime placement management on FPGAs, including design time optimization of hardware tasks to enhance their place-ability in runtime, efficient and robust fragmentation minimization techniques in the placement of tasks on COTS FPGAs, achieving low time overhead defragmentation on state-of-the-art FPGAs within the context of their relatively large reconfiguration time, relocation of hardware tasks on FPGA platforms, providing access to clock nets at the right clock frequency during

runtime placement of tasks while ensuring that the placement process does not impact on the performance and reliability of a design. The techniques proposed in this thesis uses Xilinx FPGAs are case studies since Xilinx has the largest share of the FPGA market [15]

1.1 Thesis Objectives

The primary objective of this thesis is to design and implement efficient placement management techniques for reconfigurable computing targeting high performance and reliability. It aims to develop generic routines and procedures as well as provide their implementation strategies that can be integrated into the design of efficient ROS on COTS FPGAs platforms for different applications. The specific objectives of the research presented in this thesis are as follows:

i) To develop an efficient design-time optimization strategy for hardware tasks with a view of enhancing their place-ability on reconfigurable hardware in runtime.
ii) To design and implement efficient and robust fragmentation minimization techniques in the placement of tasks on COTS FPGAs, and comparing these to state-of-the-art fragmentation minimization mechanisms.
iii) To implement hardware task reuse strategies to circumvent the relatively large reconfiguration time of COTS FPGA using a fragmentation-aware task replacement policy
iv) To develop novel techniques for improving the relocation of hardware tasks on FPGA platforms
v) To integrate mechanism of providing access to clock nets at the right clock frequency to hardware tasks placed in runtime while ensuring that the process does not impact on the performance and reliability of the design.
vi) To provide an implementation case study of a placement management system for ROS based on ii) to v).

1.2 Summary of Novelty and Contribution

The first contribution of this thesis is the development of an offline optimization flow that aims to improve the number and distribution of task placement locations on the FPGA in runtime [21]. With the increase in the degree of heterogeneity of COTS FPGA, the technique leads to a reduction in the incidence of overlapping locations for tasks in runtime scenario even when the execution order of the tasks is not known at design time. Moreover, the minimization of the variance in the number of potential matching locations ensure that some application components are not denied placement while others have abundant locations, leading to a pre-mature failure of the application. The proposed optimization technique leads to greater reliability in applications where relocation technique is used to circumvent permanent damage on the chip.

The second contribution of this thesis is the presentation of a task reuse mechanism on COTS FPGAs to circumvent their large reconfiguration overhead in runtime applications [22]. The reuse mechanism is based on a novel replacement policy which not only aim to preserve tasks with large configuration overhead on the chip, but also uses each task replacement window to offer some defragmentation of the FPGA area [23].

In addition, an efficient fragmentation quantification technique suited to heterogeneous FPGA platforms is developed. The aim of the fragmentation metric is to address the limitation in heterogeneous FPGAs where matching locations for tasks are not guaranteed to be found at the border of existing placements or the border of the chip as is the case on homogenous FPGAs. This was reported as part of [23]. In addition, a technique called Expanding the unusable Area Scheme (EUAS) is also presented to further improve chip area utilization and to circumvent the creation of unusable areas due to the heterogeneous nature of the chip is also presented and reported in [21].

The fourth contribution in the thesis is functionality-based runtime relocation technique for hardware tasks on heterogeneous FPGAs [24]. The technique is used to

augment direct bitstream relocation techniques by replacing the functionality of certain tasks by a look-up-table (or memory). This makes it possible for them to be placed on locations which does not match the original task's bitstream due to the heterogeneous nature of COTS FPGA. Hardware task relocation is a beneficial technique in reconfigurable computing which can potentially be applied to circumvent permanent faults on the chip, achieve defragmentation and load balancing. Thus, the proposed technique which improves the number of possible relocations of tasks on COTS FPGA enables ROS to potentially improve the reliability and performance of applications.

The final major contribution of this thesis is an efficient and reliable runtime clock network delivery technique to hardware tasks placed in runtime [25]. The technique is resource efficient as it is done through the configuration layer of the FPGA. This is necessary to support the runtime placement of tasks on any matching location on the FPGA. To this end the architecture of the configuration bitstream was studied and key control bits for clock net routing were identified. Furthermore, to avoid jeopardizing the reliability of the system in the process of editing configuration bitstream, a runtime frame error correcting code (Frame ECC) re-computation controller is implemented to re-compute Frame ECC values after edits in such a manner as not to impact the performance of the system.

Lastly, the techniques developed in the thesis are integrated into an implementation of a prototype placement management system to show their practicability. The performance of most of the proposed techniques are tested using data processing tasks of a NASA JPL spectrometer application. The results show that the proposed techniques lead to improvement in the reliability and performance of applications for hostile environment over state-of-the-art techniques. Hence, they have potentials to contribute to the design of low-cost, high-performance, fault-tolerant multisensory electronic systems for hostile environments such as nuclear plants and outer space.

It is important to note that the work presented in this thesis is part of a larger effort at the Ewireless Research Group, University of Edinburgh aimed at developing a reliable real-time operating system for COTS FPGAs. Therefore, it is necessary to

acknowledge the contribution of Adewale Adetomi who developed and implemented a flexible communication infrastructure that supports dynamic placement and relocation of hardware tasks without the need for pre-defined partitions [26]. The communication mechanism is necessary to support the generic placement techniques presented in this thesis. In addition, he also designed and implemented a configuration controller which is used both for configuring tasks on the FPGA after placement decisions and also for soft error mitigation techniques [27]. However, the reverse engineering experiments carried out as part of this thesis were used in the design and implementation of the configuration controller. The configuration controller is used for the relocation technique presented in chapter 6 to coordinate the copying of data through the configuration memory. It is also used in chapter 8 in the online routing of clock nets to newly placed tasks.

1.3 Publications Arising from Thesis

The following are the publications which have been drawn from the research work contained in this thesis:

Journals

G. Enemali, A. Adetomi, G. Seetharaman and T. Arslan, “A Functionality-Based Runtime Relocation System for Circuits on Heterogeneous FPGAs,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 5, pp. 612–616, May 2018.

Conferences

G. Enemali, A. Adetomi, and T. Arslan, "FAReP: Fragmentation-Aware Replacement Policy for Task Reuse on Reconfigurable FPGAs", in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 202 – 206, 10.1109/IPDPSW.2017.153.

G. Enemali, A. Adetomi, and T. Arslan, "A Placement Management Circuit for Efficient Realtime Hardware Reuse on FPGAs Targeting Reliable Autonomous Systems", in 2017 IEEE International Symposium on Circuit and Systems (ISCAS 2017), 2017, pp. 2030 – 2033, 10.1109/ISCAS.2017.8050796

G. Enemali, A. Adetomi, and T. Arslan, "Expanding the Un-usable Area Strategy for Improved Utilization of Reconfigurable FPGAs", in 2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2017, 10.1109/AHS.2017.8046370.

G. Enemali, A. Adetomi, and T. Arslan, "Efficient Runtime Frame ECC Re-computation for Reliable Task Execution on Xilinx FPGAs ", in 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2018, pp. 59 – 65. 10.1109/AHS.2018.8541471

A. Adetomi, **G. Enemali**, and T. Arslan, "Relocation-Aware Communication Network for Circuits on Xilinx FPGAs”, in 2017 International Conference on Field

Programmable Logic and Applications (FPL), 2017, pp. 1-7, 10.23919/FPL.2017.8056818.

A. Adetomi, **G. Enemali**, and T. Arslan, "A Fault-Tolerant ICAP Controller with a Selective-Area Soft Error Mitigation Engine", in 2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2017, pp. 192-199, 10.1109/AHS.2017.8046378.

A. Adetomi, **G. Enemali**, and T. Arslan, "R3TOS-Based Integrated Modular Space Avionics for On-Board Real-Time Data Processing," in 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2018. Pp. 1- 8. 10.1109/AHS.2018.8541369

1.4 Thesis Outline

This thesis is organized in nine chapters. In this chapter, an introduction to the use of FPGAs in electronics and computing devices as well as the contribution and aim of the thesis have been presented. The remainder of this thesis is organized as follows:

Chapter 2: Introduction to Dynamic Partial Reconfiguration Reliability and Clocking Infrastructure on FPGAs

This chapter presents background information relating to dynamic partial reconfiguration as well as a review of comercial tools to harness the potentials of DPR. The chapter also gives relevant background information on reliability isues in FPGAs. The clocking infrastructure of FPGAs that enable clock network delivery in circuits in runtime is also reviewed.

Chapter 3: Runtime Placement Management and Low Power Computation on FPGAs

A review of related research efforts at developing reconfigurable computing tools to harness the potentials of DPR for high performance and reliability in applications beyond the capabilities offered by comercial tools is presented in the chapter. The focus of the chapter is on placement management as a key part of reconfigurable computing tools, thus a review of the relevenat research work on runtime placment of hardware tasks on FPGAs is presented. The three aspects of placement management reviewed include: managing the FPGA area for efficient utilization, configuration overhead management in runtime placement systems and runtime clock routing to tasks after placement. In addition, the chapter also reviews the methods of power consumption minimization on FPGAs.

Chapter 4: Offline Design Optimization for Efficient Runtime Placement and Reliability

Design-time optimization techniques aimed at improving the performance of the hardware tasks in runtime is presented in this chapter. A series of optimization steps

is presented that transforms an RTL design into optimized partial bitstreams with a selection of synthesis locations for the tasks that ensures an optimal number and distribution of placement location for constituent tasks. With these optimizations, the reliability of applications is improved. In addition, an optional technique for achieving low power computation based on memoization is proposed for tasks with low port width.

Chapter 5: Runtime Placement on FPGAs for High Performance and Reliability

This chapter focuses on the runtime phase of placement management for high performance and reliability. It gives the details of techniques in efficient runtime task placement on heterogeneous COTS FPGAs. The techniques include a novel fragmentation quantification and efficient task reuse techniques. The fragmentation quantification technique is based on measuring the isolation of an area of the chip that can be potentially occupied by hardware tasks and aims to select task placement locations to minimize the fragmentation of the chip area. In addition, a task reuse mechanism is presentation that circumvent configuration of certain tasks to reduce the workload of the configuration engine. The task reuse scheme is based on a novel task replacement policy which offer some defragmentation of the chip area during task replacement. By improving the utilization of the chip area, more application components can be executed on the chip leading to lower task rejection ratio and potential for a greater number of task relocation. In addition, by circumventing task configuration, the configuration port is more available for soft error mitigation.

Chapter 6: Techniques for Task Relocation on FPGAs

The chapter discribes relocation techniques for hardware tasks on COTS FPGAs. A functionality-based relocation technique is proposed to augument direct bistream relocation on hetergoneous FPGAs. The aim of the proposed functionality-based relocation technique is to replicate the functionality of certain hardware tasks at another location on the chip where the original bitstream cannot be configured due to

lack of matching resource. The proposed technique is based on memorizing the computations of tasks' output over the normal execution duration of the tasks and using these to create a look-up-table at a destination location. The techique is limited to only referencially transparent tasks with low port widths.

Chapter 7: Placement Management System Implementation and Characterization

The techniques developed in chapters 5 and 6 are integrated into a case study implementation of a placement management system. The Xilinx 7 series FPGA is used as a case study to show the practicability of the proposed techniques. However, the algorithms and heuristics can be extended to other reconfigurable FPGAs. The implementation is characterized in terms of the timing behavior as well as the resource overhead, and its performance is compared with another state-of-the-art placement system. The results show that the proposed placement management system has a worst-case placement duration of less than 50% of a comparable system, while having a comparable placement quality and resource overhead.

Chapter 8: Towards a Reliability-Aware Efficient Runtime Clock Routing in Reconfigurable Computing.

This chapter address the challenge of delivering clock networks to tasks after placement in runtime. It presents a technique of routing clock networks to hardware tasks in runtime via the configuration layer by editing the configuration bitstream. However, there are reliability issues associated with editing the content of the configuration memory in runtime as the frame error correcting codes stored as part of the bitstream becomes invalid. Thus, the chapter also presents an efficient means of re-computing Frame ECCs after editing configuration bits in such a way that the performance of the system is not degraded.

Chapter 9: Conclusion and Future work

This chapter gives the conclusion of the research work presented in the thesis. It also outlines the significance of the results, identifies the limitations of the work and suggests future works.

Chapter 2: Introduction to Dynamic Partial Reconfiguration, Clocking Infrastructure and Reliability Issues on FPGAs

FPGAs are reconfigurable devices which are used to implement circuits. They offer hardware performance similar to ASICs. Their architecture can be divided into the physical layer (which contains functional resources such as look up tables, flip flops, etc.) and a configuration layer. The functionality of the physical layer at any time is defined by the design programmed into their configuration layer. The process of programming an FPGA can be repeated frequently and a great number of times. In practical terms, the number of times SRAM-based FPGAs can be reprogrammed could be regarded as *indefinite* [28], thus making FPGAs highly flexible. In addition, the performance offered by circuits configured on FPGAs is based on an actual (re)wiring of hardware resources to build circuits. Hence, these circuits can be optimized to have a class of hardware performance close to that of ASICs [29]. Thus, FPGAs have a unique combination of high performance and flexibility which can be harnessed to revolutionize many system designs.

Advances in modern FPGA architectures and tools have equipped them to be used to implement complex systems. From a humble beginning of including only 85,000 transistors (forming only 64 CLBs and 58 I/O block) [30], FPGAs have grown by more than 10^4 times in capacity, 10^2 times in performance, while energy consumption and cost have reduced by more than 10^3 times [31]. They now include dedicated signal processors, block of RAM, hard multi-core processors. Thus, they have great potentials to be used to implement cost-effective complex SoCs in a short time.

Their flexibility is especially desirable for many reasons, including easy update of FPGA-based designs. This can translate to huge savings in cost compared to ASIC based in applications that need to be upgraded to use better (or different) algorithms [32]. Thus, FPGA-based designs are future-proof. In addition, design updates can be carried out much more quickly, reducing system down-time and improving

reliability. Also, the fact that the configuration bitstream can be sent remotely is a great advantage for remote systems where physical access to the system is restricted.

One key technology that has significantly contributed to these feats achieved by FPGAs is DPR [33]. It opened even further possibilities for FPGAs to be harnessed for high performance and reliability applications. DPR makes it possible for a part of an application operating on an FPGA to be changed without affecting the functionality of the other parts of the application. However, runtime management tools and techniques are needed to harness these potentials of modern COTS FPGAs to achieve high performance and reliability.

For the remainder of this chapter, the concept of DPR would be explained with a description of how it could be harnessed for high performance and reliability. Thereafter, an overview of a commercial tool for harnessing DPR on COTS FPGAs in runtime applications will be presented. The limitations of the tool are also identified.

Furthermore, the chapter gives an overview of clocking architecture of FPGAs that enable clock network delivery to circuits in runtime. In addition, an introduction to reliability isues in FPGAs is given with a focus on possible ways of addressing the challenge. Most of the terminologies used in this chapter are for Xilinx FPGAs, however similar terms exist for other classes of FPGAs also and the underlying concepts can be extended to these other FPGA families or even other reconfigurable hardware types in some cases.

2.1 Introduction to Dynamic Partial Reconfiguration

DPR allows the behaviour of part of a chip to be redefined without stopping the operation of the other parts of the chip. Hence, it is possible to swap hardware tasks (circuits) in and out of the chip, and effectively turn it into a platform which can both be time and area-shared among multiple tasks [8] while offering high performance. Circuits (or a subset of circuit(s)) configured on an FPGA with DPR capabilities

could be removed when not needed to make room for other circuits or to modify the functionality of the system. The concept of DPR is illustrated in Figure 2.1.

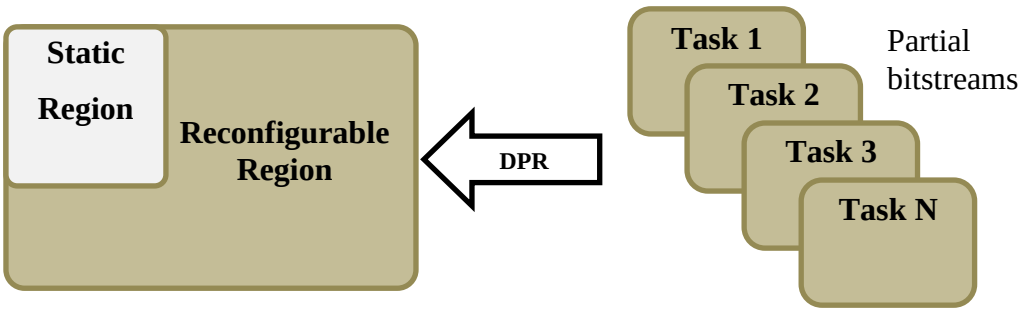


Figure 2.1: Dynamic Partial Reconfiguration in FPGAs

Tasks 1 to N shown in the figure can be loaded onto the reconfigurable region of the FPGA dynamically using DPR without affecting the operation of the static part of the chip. In fact, even the part of the reconfigurable region which the task being loaded does not overlap can retain its functionality. DPR is applicable to SRAM-based FPGAs [34] [35] such as Xilinx FPGAs. SRAM-based FPGAs holds their configuration bits in a Static RAM on the FPGA called the configuration memory spread around the chip in a configuration layer. DPR simply writes a section of the configuration memory, altering the functionality of the part of the chip where that section of memory controls.

While a *full* configuration (over)writes the entire content of the configuration memory and thus alters the behaviour of an entire chip, a *partial* configuration writes only part of the memory. A full configuration is done when a *full bitstream* type is loaded on the FPGA, as opposed to loading a *partial bitstream* on the chip for partial reconfiguration. DPR loads *partial* bitstreams unto sections of the configuration memory in runtime. Furthermore, while a full bitstream must be a specific length for an FPGA chip, partial bitstreams vary in sizes, depending on the amount of functionality desired to be changed in runtime [36]. Theoretically, their size could be as large as the full bitstream and could be as small as just a single *configuration*

frame. A configuration frame is an addressable unit of the configuration bitstream. Further details of the configuration bitstream is presented in section 2.3.2.

While, it is worth noting that any addressable unit of the configuration memory can be written in runtime, in practice, partial bitstreams are created during an off-line design process that involves selecting a set of components as a partial reconfigurable region [37]. This region can only include certain types of components which generally vary from device to device. As an example, Table *2.1* shows an overview of reconfigurable (✔) and non-reconfigurable (✗) components in the Xilinx 7 series and UltraScale devices. Non-reconfigurable components must be left in the static region of a design.

Although some components cannot be included in the reconfigurable region of a design using the regular flow supported by the commercial FPGA design tools, by using special reconfigurable computing techniques, DPR can be extended to these components to control their behaviour in runtime. For example, some clock buffers in Xilinx 7 series devices can be enabled and disabled by writing specific locations of the configuration memory in runtime, even though they cannot be included in a reconfigurable region. Such techniques are used in this thesis to extend the advantages of DPR to components that cannot be placed the in reconfigurable region.

Table 2.1: Reconfigurable Resources in Xilinx 7 Series and UltraScale FPGAs

Component Type	**7 Series**	**UltraScale**
CLB	✔	✔
BRAM	✔	✔
DSP	✔	✔
PCI Express	✔	✔
Clock Modifying logic (e.g. clock buffers, PLL, etc.)	✗	✔
Clock Nets and PIPs	✔	✔
I/O and I/O related components	✗	✔
Serial Transceivers	✗	✔
XADC and System Monitor	✗	✔
Configuration Components (ICAP, Frame_ECC, etc.)	✗	✗

A DPR-based design creates the enabling environment to dynamically multiplex hardware tasks on an FPGA. This leads to a plethora of advantages and possibilities including: reduction of area, power and cost of resources required to implement desired functionality. It is worth noting that the frequency of performing DPR depends on the type of application and the size of the FPGA. For example, an application consisting of 2 units in a reconfigurable partition would require that DPR be performed twice per execution cycle, if both units must be executed in each cycle. However, cleaver algorithms and frameworks have been proposed to reduce the number of reconfigurations in dynamic computation scenario such as [38].

DPR also leads to ease of updating designs, providing flexibility in the algorithms to be implemented for an application and improving reliability [36]. However, it is worth noting that DPR designs have additional overheads. For example, in Xilinx FPGAs, design constraints commonly used in DPR flows results in additional overheads in timing and resource utilization in a DPR-based design compared to an equivalent design without DPR [36]. Two examples of such constraint are "CONTAIN_ROUTING" (used to ensure that routing wires belonging to a specified reconfigurable module are contained within certain boundaries) and restrictions on optimization across reconfigurable module boundaries. Nevertheless, DPR remains a great technique with so much potentials that more design tools are needed to reap many of its benefits.

2.2 Xilinx Tool Support for DPR

Xilinx support for DPR could be classified into two main classes:

a) Flow for creating partial reconfiguration bitstreams for modules (done offline)
b) Provision of partial reconfiguration controller that enables self-programmability from within the FPGA (supports runtime DPR).

2.2.1 Creating Partial Reconfiguration Bitstreams

The procedure for partial bitstream creation for Xilinx FPGAs is documented in [37]. In addition to the partial bitstreams, a full bitstream is also created to be used for the initial configuration of the entire device. An optional 'Black-box' bitstream can also be created for each partition which can be used to wipe out a configuration when not needed.

Figure *2.2* shows a summarised Xilinx partial reconfiguration flow using Vivado (v15.1) IDE. First, the design top module is synthesized after removing all other files from the design. The top module contains those segments of the design that translates to the static part of the design shown in Figure 2.1. The synthesized result for the top module is saved as checkpoint (.dcp file). The process is repeated for each reconfigurable module (RM) present in the design, each time removing all other files from the design, setting the target RM as the 'top' module, running synthesis and saving the design check point file. Next, the saved checkpoints are loaded up, assembled together and each of the RMs are set as *partially reconfigurable.* A floor-plan area is then created and assigned for each partition. Each floor-plan area is referred to as p-block. The area covered by a p-block must include sufficient number of resources required by the RM(s) it is meant to accommodate. A partition could accommodate more than a single RM, in which case, the partition must contain a superset of the resources required by all RMs to be placed in that partition.

Attributes are set for the design before performing a design rule check (DRC) and running implementation. Two examples of attributes set for the design at this stage are 'RESET_AFTER_RECONFIG' to enable dedicated initialization of an RM after its reconfiguration and CONTAIN_ROUTING to instruct the place and route process to keep all routings belonging to a RM within the partition to which it is assigned. DRC step ensures that essential constraints are met before attempting to implement the design.

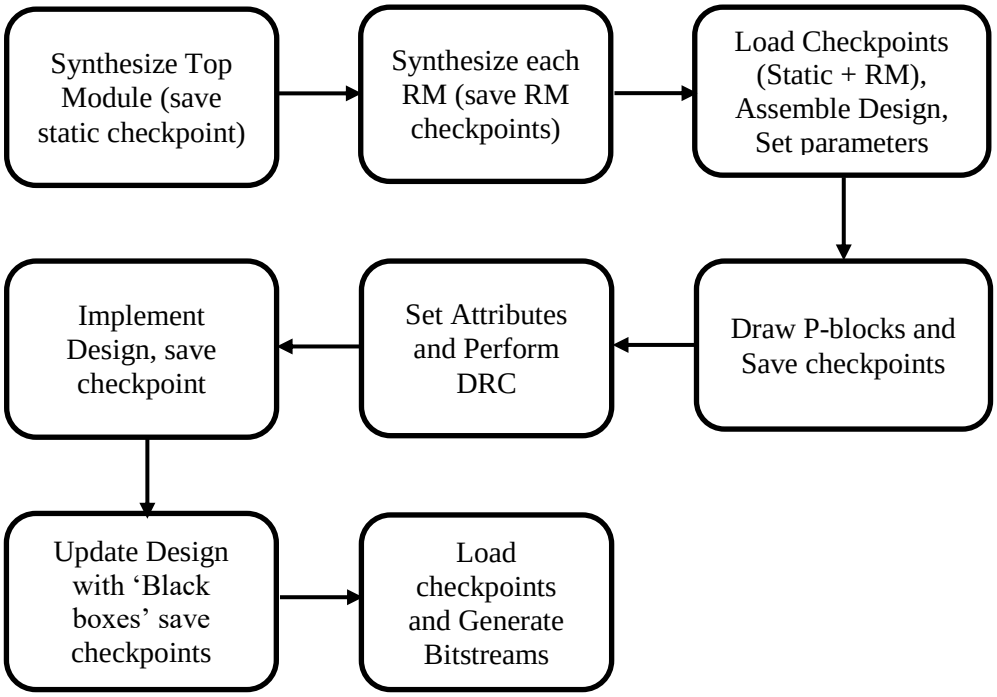


Figure 2.2: Summary of Xilinx Partial Reconfiguration Flow

For example, DRC run indicates an error if a p-block includes invalid sites or have insufficient number of resources to accommodate all RMs assigned to that partition. Users can select which of the design rules to be checked in their design using the *Tools* tab in Vivado. After a successful DRC, the design is implemented, and the current checkpoint saved. This checkpoint is later loaded to generate the design bitstreams. First, a full configuration bitstream is generated for the entire design, and partial bitstreams are generated for each RM in each partition. To generate an optional 'black-box' bitstream in addition to full and partial bitstreams; after the implementation step, the design is updated with 'black-boxes' (essentially empty designs for each partition) and a checkpoint created for the updated design. This later checkpoint is loaded to create 'black-box' bitstreams. These are blanking bitstreams used for removing the functionality an RM.

2.2.2 Xilinx Partial Reconfiguration Controller

A configuration controller fetches the bitstream from memory and delivers it to a configuration port. The configuration controller can either be self-contained in the programmable logic itself or reside in an external device such as a processor. The ICAP is the port available to the internal logic resources after the device is programmed. It enables the chip to be programmed from within itself, and it is the primary port for DPR in Xilinx FPGAs. It is important to note that a full configuration is always required after device power-up before partial configuration is supported. Examples of other configuration ports include: SelectMap, Serial, JTAG interfaces. In addition, processor configuration access port (PCAP) and media configuration access port (MCAP) can be used for downloading bitstreams on Zynq®-7000 SoC devices and UltraScale devices respectively.

Xilinx provides a customizable Partial Reconfiguration Controller (PRC) IP that can be used to manage partial reconfiguration in runtime. Xilinx PRC receives interrupts from a higher system manager, coordinates the fetching of partial bitstreams from eternal memory and deliver them to the ICAP [36]. It supports enclosed designs where the RMs are known to the controller [39]. It has a capacity of managing up to 32 reconfigurable partitions with a maximum of 128 RMs per partition. When the set of RMs to be managed in a system changes in runtime, the PRC must be reconfigured using its AXI-lite register interface.

The PRC's architecture is composed of a set of virtual socket managers. Independent socket managers control different reconfigurable partitions simultaneously as they can operate in parallel. However, the access to the path for fetching the partial bitstream from memory as well as the configuration port can accommodate only a single request at a time. The socket managers respond to external triggers which could originate from a processor or another hardware management source. Figure *2.3* shows the operation flow of a socket manager. After an interrupt is received, any RM in the target partition is first cleared out before initiating the configuration of a new RM on the same partition. This is an optional step as there might be no RM in the target partition. Similarly, after the configuration of a new RM on a partition, start-up

operations such as coupling it to the static part might be required. The optional steps are shown in dotted rectangles in the figure.

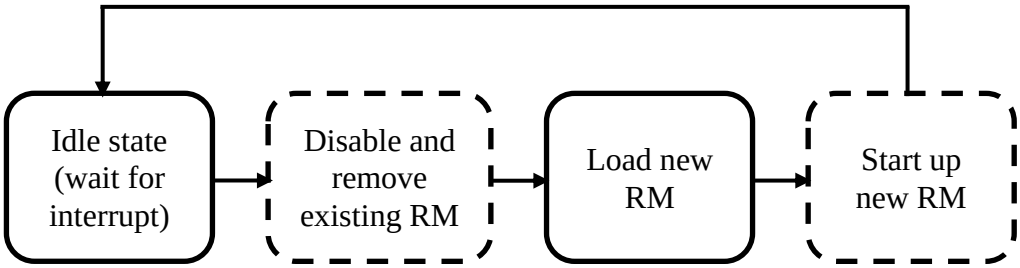


Figure 2.3: Main Steps of a Virtual Socket Manager [Adapted from *[39]*]

2.2.3 Limitations of Xilinx Partial Reconfiguration Controller

Although Xilinx's support for DPR, offers many useful features that explores some of the potentials of DPR, there are several important limitations with the tool support. Three major ones are:

i) Inefficient use of FPGA Area

The architecture supported by the process of generating partial bitstream using Xilinx reconfiguration flow and their runtime placement on the FPGA chip do not optimize the use of FPGA area. A major reason for this is the use of pre-determined slots (called reconfigurable partitions) for RMs. The use of partitions mean that a slot assigned to a set of RMs must contain a superset of all resources of the RMs. This leads to internal fragmentation [18] as smaller RMs would have unused area in the slots they share with larger RMs. Fragmentation leads to increased area usage per application which in turn leads to higher power consumption and cost.

In addition to the reconfigurable partitions themselves being resource inefficient, the resource utilization of the PRC [39] is quite high compared to a custom implementation of a runtime configuration controller. This is illustrated in Table *2.2*. From the table, it is easy to see that the custom PRC implementation [27] has an overhead of only 31.17% and 62.94% of the amount of FFs and LUTs required by

the Xilinx PRC. However, it is important to note that the PRC has functionalities which are not present in [27] and vice versa. For example, the [39] has decoupling functionalities not reported in [27], while the [27] has bitstream relocation feature and capacity to handle encrypted bitstreams not present in the [39]. Nevertheless, most of operations surrounding dynamic loading of partial bitstreams are reported in both.

Table 2.2: Resource Overhead of Xilinx PRC on Kintex7 Device

Resource Type	**Xilinx PRC** [39]	**Custom Controller** [27]
FF	1203	375
LUT	1171	737
BRAM	-	3

ii) Encrypted Partial Bitstream not Supported

Xilinx PRC does not fully support the runtime configuration of encrypted partial bitstreams even on its very recent devices such as the 7 series FPGA [39]. On the UltraScale devices, the PRC offers limited support for the configuration of encrypted partial bitstreams. But in the case of error occurrence during configuration, the system would not be able to recover. In this age when data and application security is very important, such lack of full support for encryption creates opportunity for IP theft and other security threats [40], [41] [42] [12].

iii) Lack of Bitstream Relocation Support

Xilinx PRC does not support bitstream relocation. Hence, a partial bitstream synthesised at one location on the chip cannot be placed at another location of the chip in runtime. Bitstream relocation is a potentially beneficial technique in many FPGA-based applications. It has the advantage that fewer number of partial bitstreams can be stored and configured at different locations when needed in runtime. In addition, bitstream relocation can be used in critical applications to circumvent damages on the chip such that in the event of fault, a circuit can be relocated to another location. Another advantage of bitstream relocation is

defragmentation of the chip area. Without support for runtime bitstream relocation, Xilinx PRC does not meet the need of most modern reconfigurable computing systems.

Given the limitations identified above and the numerous prospects of DPR, several academic efforts have been directed towards developing tools that would better manage the FPGA resources in runtime [43] [18] [44]. In addition to harnessing the prospects offered by DPR in several application domains, many of the tools also aim to simplify the process of deploying DPR. This is to enable DPR to be available to ordinary users by abstracting low level details. Notable examples of proposed ROS and reconfigurable computing tools include: R3TOS [8], ReconOS [45], CAP-OS [46], LEAP FPGA OS [47], RIFFA [48] and RTSM [49]. More details on these tools and techniques is provided in chapter 3.

2.3 Clocking Infrastructure and Bitstream Format of FPGAs

After the placement of task in runtime in a scenario where there are no pre-determined partitions for the tasks, it becomes necessary to route clock networks to the newly placed task. The proposed technique in this thesis is to achieve efficient clock routing through the configuration layer. In this section, first, the clocking architecture of an FPGA is reviewed with a focus on the opportunities the clocking infrastructure offer for efficiently delivering clocking network to a task in runtime. Next, a brief description of the structure of configuration bitstream is presented, identifying sections to be edited in runtime to achieve clock routing and the sections for error monitoring. The discussion uses the Xilinx 7 series FPGA family as an example.

2.3.1 General Structure of Clocking Network on Xilinx FPGA

Like most recent FPGAs, each Xilinx's 7 series FPGA device is divided into units called clock regions. The number of clock regions in a device varies from 2 to 24 depending on the device size [50]. All clock regions have a height of 50 CLBs (equivalent to 10 36-kb Block RAMs or 20 DSPs). The number of columns in a

clock region varies by device, ranging from 20 in the small Artix 7 device to 62 in a Virtex 7 device. A clock region defines the area of the device serviced by dedicated clock nets and buffers.

The major clocking resources present in a clock region of a Xilinx 7 series FPGA consists mainly of clock buffers, clock nets and programmable interconnection points (PIPs). These are briefly described below.

***Clock Buffers**:*

There are 4 types of clock buffers in each clock region. These include 12 horizontal clock buffers (BUFH) and 4 regional clock buffers (BUFR). These can be used to directly drive logic resources such as flip flops, BRAMs and DSPs. The other two clock buffers in a clock region are the multi-regional clock buffer (BUFMR) and the I/O clock buffer (BUFIO). BUFMR and BUFIO cannot be used to feed logic directly. BUFMRs feeds BUFRs which in turn drives an intended logic resource while BUFIO drives the I/O clock tree. Since BUFIOs are not involved in driving reconfigurable logic, they are not used in the process of delivering clock network to placed tasks and hence they have been omitted from the following descriptions and figures. In addition to the buffer types listed above, there are 32 global clock buffers (BUFGs) which are not located in any specific clock region, but are part of the global clock tree and are collectively located at the centre of the device.

It is worth noting that the type of buffer that can be used to deliver clock signal to a task is determined by 3 factors:

i) the size of the task (in terms of the number of clock regions the task spans),
ii) task shape (in terms of the orientation of the task – whether the task spans clock regions in the vertical or horizontal directions) and
iii) the location of the tasks on the chip.

Figure 2.4 shows five tasks with a list of the buffers which can be used to feed clock signal to each. Tasks which span more than one clock region in the vertical directions only but limited to 3 clock regions (such as Task A and Task C) can only be fed by

BUFG and BUFMR. It should be noted that BUFMR must be routed via a BUFR to the task. Tasks which are limited to a single clock region (e.g. Task D) can be fed by all buffers capable of feeding logic. Tasks spanning more than a clock region in the horizontal directions only (such as task E) can be fed by BUFG and BUGH only. Finally, a task which does not conform to any of the above three categories (such as Task B) must be fed with a BUFG. In addition to the *reach* of each clock buffer, characteristics such as clock division capability, runtime Enable/Disable capability as well as limitation on the number of available buffers play a role in the selection of buffer to feed a task in runtime. For example, tasks requiring the frequency of available clock to be divided by a factor must be routed through BUFR as only BUFRs have the clock division capability.

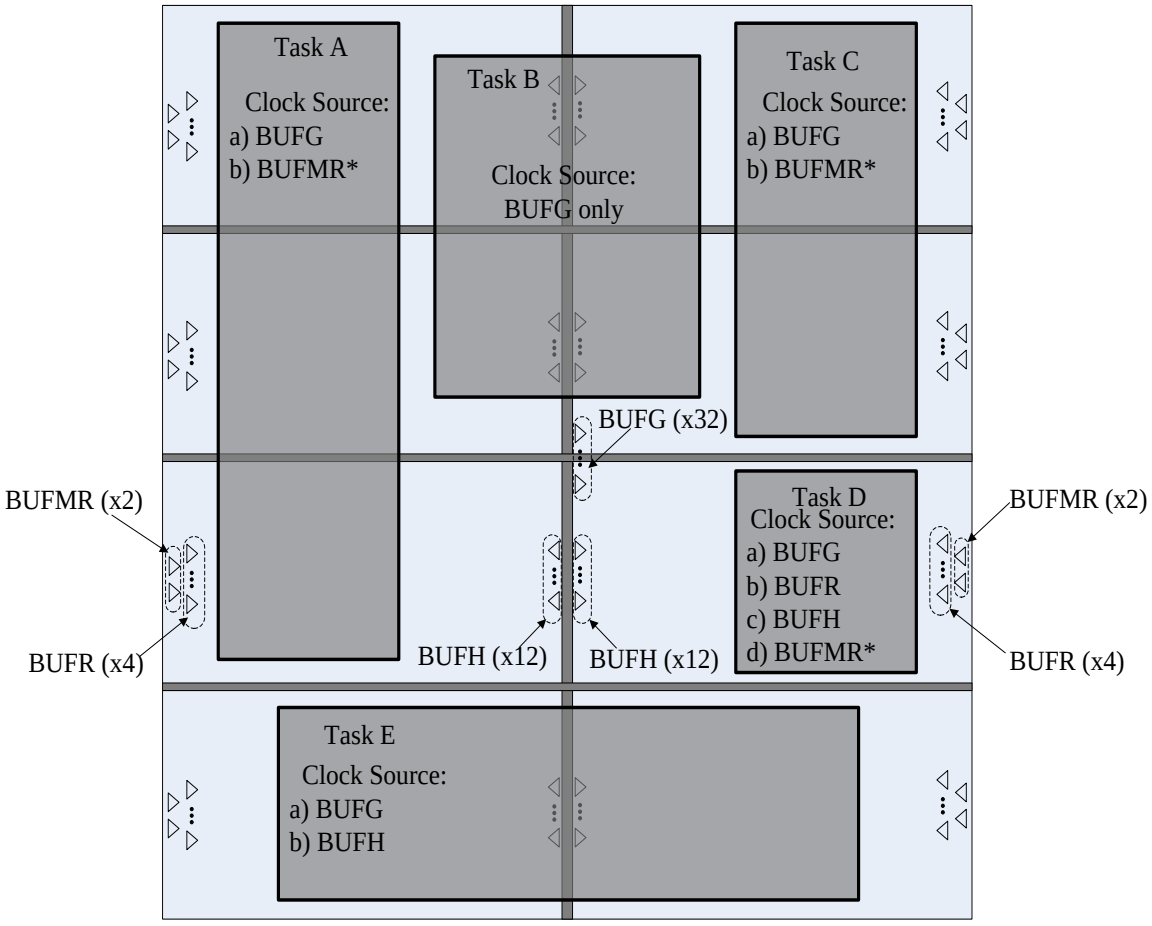


Figure 2.4: Example Tasks and List of Clock Buffers for Clock Network Delivery

Clock Nets and PIPs:

In the 7 series, there are 16 horizontal clock nets in each clock region. These are divided into two groups: 12 nets which are in the horizontal clock row (HROW), and 4 dedicated regional nets. The nets are physically located at the middle of the columns which occur between 25 upper CLBs and 25 lower CLBs in a clock region. They are available to all columns of the region with logic resources. The nets can be routed to all synchronous elements using a set of PIPs. The PIPs serve as connectors from clock buffers to clock nets, and clock nets to other nets/trees. In each column of the device, the PIPs enable the possible routing of the clock nets to the synchronous elements in the column. Figure 2.5 shows a simplified illustration of the arrangements of BUFHs, clock nets and PIPs. This arrangement facilitates the routing of clock nets to any column in the region with synchronous elements. As shown, the 12 BUFHs in a clock region can normally drive 12 of the 16 horizontal nets. These 12 nets can also be driven by BUFGs. The other 4 dedicated nets are driven by the BUFRs in the clock region in which they are located.

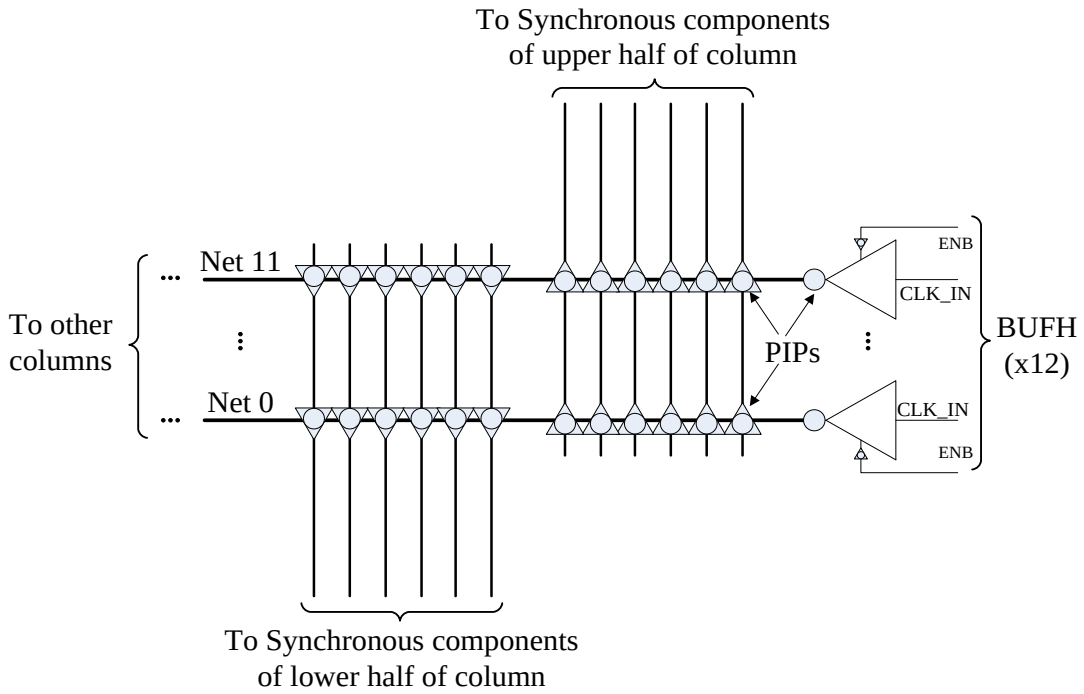


Figure 2.5: A Simplified Illustration of BUFHs, Clock Nets and PIPs

In addition, a set of PIPs can potentially switch a net vertically to deliver any 12 of the 16 horizontal clock nets in a clock region to the synchronous elements in a column. As shown there are 6 nets which enter a column from the HROW for each column: 6 PIPs (TOP0, TOP1, … TOP5) deliver clock signals to the resources in the upper half of the column and 6 to the lower half (BOT0, BOT1, … BOT5). In addition to these, 6 clock nets from an adjacent column can also be delivered to the upper part column and lower parts. Hence, a maximum of 12 clock nets can be routed to synchronous elements in a column simultaneously.

2.3.2 Overview of Relevant Sections of Xilinx Bitstream Format

Figure *2.6* shows major sections of the configuration bitstream of a typical Xilinx 7 series FPGA. Four distinct sections of the bitstream can be identified. The first is a pre-amble and synchronization section which contains data relating to setting up the configuration interface. An example of a set of data contained in this section of the bitstream is the bus width auto detection sequence which is used to adjust the configuration port to a desired width. Supported port width are 8, 16 and 32 bits.

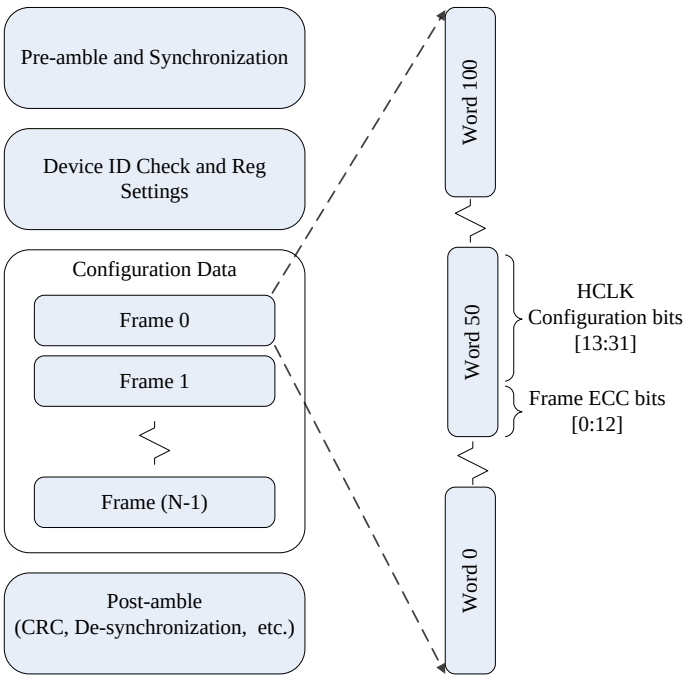


Figure 2.6: A simplified Illustration of Sections of the Configuration Bitstream

The bus width detection logic looks for the following patterns on bit [7:0] in the configuration bitstream pre-amble section to set the bus width for configuration:

i) for 8 bit: D[7:0] = 0xBB, followed by 0x11 on the same bits in the next word

ii) for 16 bit: D[7:0] = 0xBB, followed by 0x22 on the same bits in the next word

iii) for 32 bit: D[7:0] = 0xBB, followed by 0x44 on the same bits in the next word

This section of the bitstream also contains the synchronization word which signals to the device that configuration data is about to be loaded and aligns the configuration data with the configuration logic. For the 7 series and UltraScale FPGAs the synchronization word is: 0xAA995566.

The second section of the bitstream consists of device ID check, setting of configuration registers to control parameters such as number words to be configured, etc. The device ID check avoids the configuration of a bitstream meant for a different device. The ID of a device has the format shown in Table 2.3. As an example, the Xilinx 7 series chip on Digilent's basys3 FPGA board has a unique ID CODE: 0x0362D093.

Table 2.3: Format of Device ID Code in Configuration Bitstream

[31:28]	[27:21]	[20:12]	[11:0]
Version	FPGA Family Code	Array Code*	Company Code

*array code includes 4-bit sub-family and 5-bit device code

The fourth section of the bitstream consist of instructions to confirm CRC values as a means of data integrity check. During loading of configuration data, CRC values are computed for the data loaded unto the device. This section of the bitstream contains instructions to load the value of the computed CRC and compare this with that in the bitstream. The fourth section also contain commands to de-synchronize the configuration port.

The third section is the actual configuration data which are written to the configuration memory (CMEM) of the FPGA. It is the largest part of the

configuration bitstream and contains the data specific to the RTL design being configured as well as all components in the region of the FPGA being configured. The data are organized in configuration frames. A configuration frame is the smallest resolution of configuration on a Xilinx FPGA chip. For the 7 series, a configuration frame consists of 101 words, each word being 32-bits wide. Each frame is identified by a specific frame address and can be written to the configuration memory of the device to change the characteristics of the primitive (or part of it) which the frame controls.

A configuration frame is built up from individual primitives in a unit of the components listed in Table 2.1. The components are organized in columns and rows on most modern Xilinx FPGAs. Several rows are further grouped together to form a clock region. For example, in the 7 series devices, a clock region consists of 50 CLB rows [36]. Thus, a column of CLB has a width of 1 and a height of 50 CLBs. Similarly, a BRAM column has a width of 1 and a height of 20 BRAMs. A number of configuration frames are required to write the configuration memory corresponding to a column of components. The number of frames vary by component type.

The number of configuration frames N is constant for a full bitstream of a specific device. For partial bitstreams, N is directly proportional to the number and type of resources on the chip area to be configured. Table 2.4 shows the number of frames required to configure each reconfigurable resource type [25]. It is worth noting that the table have been organized in pairs of columns because a pair of reconfigurable resource column share a routing network in the 7 series FPGAs.

Table 2.4: Number of Configuration Frames in Reconfiguration Resource Pair on Xilinx 7 series FPGA

Resource Pair	***CLB-CLB***	***CLB-DSP***	***CLB-BRAM***
Number of Frames	72	64	192

Each configuration frame in the bitstream is referenced by a frame address whose format is shown in Table 2.5 The frame address links the data in the frame to physical

resources on the chip by defining its row, column and resource type [51]. It also specifies whether the resource is in the upper or lower half of the device and the specific section of the device to be configured. For example, the frequency of the clock signal routed via a BUFR in a specific clock region may be controlled by re-writing a specific frame in a specific column of the device.

Table 2.5: Frame Address Format in Xilinx 7 series FPGA

Address Type	***Block***	***Top/Bottom***	***Row***	***Column***	***Minor***
Bit Index	[25:23]	22	[21:17]	[16:7]	[6:0]

Each configuration frame consists of 101 32-bit words, labelled word 0 to 100 in Figure 2.6. Word 50 is of special interest here as it contains the ECC values of the data contained in the frame. As shown in the figure, the ECC values are located in the lower 13 bits of the word. These bits are monitored in runtime to detect any changes to the composition of the data contained in the frame. In section 2.4.1, details of how they are used to detect and correct errors in the CMEM is presented.

The 50th word of each frame also contains important information relating to the clocking resources such as clock buffers as well as some information about the clock nets and PIPs in the HCLK. This information is mostly found on bits [31:13] of the word and include the clock frequency division factors of BUFR, clock enable bits for BUFH, etc. Further clock net information is also found in words 48, 49, 51 and 52 of a configuration frame. The locations of these control bits are not disclosed by Xilinx and were obtained using reverse engineering experiments. Details of these are given in chapter 8 of this thesis.

2.4 Reliability Issues in FPGAs

COTS FPGAs based on SRAM configuration memory are susceptible to bit flips. These are called temporal faults. These unwanted bit flips are often caused by effects such as ionizing radiation and extreme temperatures [52]. Temporal faults in the configuration memory can affect the functionality of an application leading to soft errors. However, not every bit flip lead to soft errors. For soft error to occur, the flip must affect a *critical* bit in the design. In addition, the number of configuration bits not used by a design reduces the soft error rate. Xilinx reports the effective soft error rate on their devices using device vulnerability factor (DVF). The DVF for a typical design is reported as 5% with a worst case value of 10% for their devices [53]. The number of failures on 7 series devices was reported as 75 FIT/Mb, were one FIT refer to one failure per one billion device hours, and Mb is 10^6 of memory bits. Nevertheless, when FPGAs are used in critical applications, soft errors need to be managed to prevent application failure. Correction of temporal faults often involve reversing the bits that have flipped.

In addition to temporal faults, there are permanent faults that occur on a chip. These are not easily correctable like temporal faults. Examples include latch-up, damage to the underlying silicon through effect of electromigration, hot-carrier injection and other ageing related effects [54], [55]. Common approach to mitigate the effect of permanent fault in runtime is by using circuit relocation or other application design techniques like triple modular redundancy (TMR). Figure *2.7* shows some of the mains cause of faults in electronic chips.

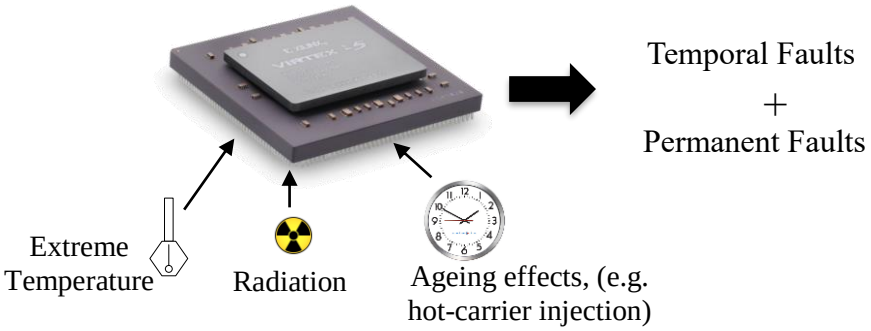


Figure 2.7: Temporal and Permanent Faults Occurrence on Electronic Chips

While electromigration, hot carrier injection and other ageing effects generally lead to permanent faults, ionizing radiation and extreme temperatures generally lead to temporal faults. Both of these poses reliability issues in FPGAs-based designs.

2.4.1 Soft Error Mitigation in FPGAs

Configuration scrubbing is the technique typically used to detect and possibly correct un-wanted bit flips in FPGAs whose CMEM are based on SRAM technology [28]. Xilinx have developed a proprietary solution which monitors errors in a manner which is transparent to users who only need to include the IP in their designs. This is called the Xilinx Soft Error Mitigation (SEM) IP [56]. Many custom solutions have also been developed to detect and/or correct soft errors on Xilinx FPGAs. Notable examples include [27], [57] and [58] which present many variants of scrubbing optimizations. However, all the techniques rely on the same fundamental principle as the SEM IP which is summarized below.

There are generally four types of memories on the FPGAs which can potentially be affected by bit flips. Arranged in decreasing order of size and hence likelihood of bit flip occurrence, these are: CMEM, block RAM (BRAM), distributed RAM (DRAM) and Flip Flops (FF). Generally, soft error mitigation in these memories except the CMEM can be performed in the design itself, by using techniques such as triple modular redundancy [56]. The SEM IP detects errors only in the CMEM. Also, it does not have the capability to avoid soft errors, but only reacts to correct them and thus mitigate their effect. Given that the IP continually scans the CMEM using the Internal Configuration Access Point (ICAP) bandwidth, Xilinx recommends that at least 99% of the ICAP bandwidth be dedicated to the SEM IP's operations [59]. With this recommendation, the typical error detection and correction latency is 25ms [56]. However, error classification can be used to improve the performance while some implementations also restrict the scanning to certain parts of the chip containing user designs and hence improves on the average speed of error detection [27].

Xilinx SEM IP implements five main functionalities: initialization, error injection, error detection, error correction and error classification. These are shown in Table 2.6.

As shown, all the functions except initialization and detection are optional and must be enabled by the user by setting specific registers. To achieve error correction in a design after the initialization stage, the SEM Controller monitors the integrated soft error detection status (based only on calibrated Frame ECC values). When an error is detected and localized, the SEM IP either corrects the error by reversing the bit flip (repair) if a single bit error occurs or reconfigures the entire frame if multiple bits are affected within the same frame (replace). The SEM IP can also operate in an advanced repair mode during which adjacent double bits errors in the same frame can be corrected. However, the advanced repair mode requires the use of both the Frame ECC and the CRC of the configuration bitstream.

At the heart of the error detection capability of the SEM IP is the FRAME_ECC primitive which provides a SYNDROME value used to determine the location of the error in the CMEM. However, the Frame_ECC primitive cannot differentiate between a user operation editing bits and soft errors (which could be caused by ionizing radiations), since it reports any bit flips as errors. Hence, since many operations in reconfigurable computing such as the runtime clock network routing is carried out through the configuration layer, it is important to update the Frame ECC values after editing the content of the configuration memory. The technique of doing this is presented in chapter 8 of this thesis as part of the runtime clock routing mechanism.

Table 2.6: Main Operations of Xilinx SEM IP

Function	***Type***
Initialization	Necessary
Error detection	Necessary
Error injection	Optional
Error correction	Optional
Error classification	Optional

2.4.2 Permanent Fault Mitigation in FPGAs

Unlike temporal faults which can be corrected by reversing a bit flip or (re)writing sections of the configuration memory, permanent faults on the chip are managed differently. In addition to techniques such as TMR, hardware task relocation has been proposed to circumvent permanent faults on a reconfigurable chip. The technique involve the re-configuration of a hardware task affected by a permanent fault at another location on the chip [19]. Circuit relocation technique need to meet certain requirements. For example, it needs to provide a means of communication with the other parts of the system or external ports on the chip. There is also the need to manage the chip area efficiently to ensure availability of free area on which to place circuits during relocation. The challenge of delivering clock networks to relocated tasks also need to be addressed. This thesis provides techniques to address the latter two of these challenges, and hence improve circuit relocation. Details of area management to improve relocation of tasks are covered in chapter 4, 5, 6 and 7 of this thesis while clock net routing is covered in chapter 8.

2.5 Chapter Conclusion

In this chapter, the concept of DPR was introduced and an overview of commercial tools support for DPR was presented, using Xilinx FPGAs as case study. The chapter also identified the limitations of Xilinx tools for DPR especially as it relates to efficient use of the chip resources. In addition, the clock architecture of a typical FPGA chip was reviewed and the resources that enable clock network delivery to tasks in runtime were identified. As the mechanism involves editing the configuration bits in runtime, the structure of a typical Xilinx configuration bitstream was also reviewed in this chapter. A description of the reliability challenges associated with runtime bit editing was also discussed. Finally, an introduction to reliability isues in FPGAs was given with a focus on how they might be addressed.

Chapter 3: Runtime Placement Management and Low Power Computation on FPGAs

Placement management is a key aspect of reconfigurable computing. It helps to harness the potentials of DPR for high performance and reliability. Reconfigurable computing involves performing computations using the area of programmable devices such as FPGAs in a dynamic application scenario [44]. Several reconfigurable computing techniques have been proposed to harness the potentials of DPR for high performance and reliability. Some authors refer to tools for managing reconfigurable computing as reconfigurable operating system (ROS), a convention that is adopted for the remainder of this thesis. Runtime placement management is aimed at ensuring an optimal utilization of the reconfigurable device resources, including the area of the chip as well as the configuration port. Maximizing the utilization of the chip area improves application performance, not only by improving the number of hardware tasks that can be executed on the chip, but also creating room on the chip for task relocation. Relocation is applicable for the purposes of circumventing permanent damage on the chip, thermal balancing, etc. Similarly, managing the reconfiguration port is essential to maintain a healthy balance among the various essential responsibilities of the single port such as task configuration and soft error mitigation.

In addition, low power computation is an important aspect of reconfigurable computing. Reconfigurable computing techniques can potentially be used to lower the energy consumption on FPGAs. In addition to minimizing energy bills, low power consumption also reduces the risk of electromigration and increases the life span of the chip as well as battery life.

In this chapter, a review of runtime placement management and low-power computation on reconfigurable hardware is presented. Regarding placement management, this includes a review of various placement management systems developed for reconfigurable computing tools. A review of underlying issues that affect quality of placements such as fragmentation and configuration overhead is also presented. Finally, an overview of low power consumption on reconfigurable

hardware, with a focus on reusing computation results to lower power consumption is discussed.

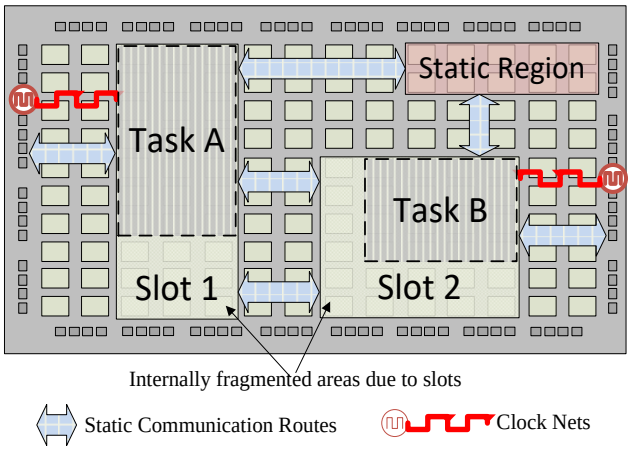
3.1 Review of Runtime Placement Management on FPGAs for Reconfigurable Computing

Most ROS have a form of runtime placement manager in their architectures. R3TOS [8] is composed of 3 main parts; scheduler, allocator (which is its placement manager) and configuration manager. R3TOS uses a non-slotted computing model, where there are no fixed partitions for RMs (or hardware tasks). Thus, tasks can be placed at any free matching location on the chip. The advantage of using a non-slotted (or non-partitioned) model is that it avoids internal fragmentation. Tasks only occupy resources which they require for their computation and hence other areas of the chip are free to be used by other tasks. A major focus of R3TOS is achieving reliability by relocating tasks affected by a permanent damage on the FPGA to another location on the chip and thus increase the reliability of critical applications in hostile environments [60]. In a slot-based model, should a single unit of resource become damaged in a slot, the entire slot becomes unusable thereby wasting all the other resources in the slot. This, the authors argued, reduces the number of relocations of tasks, and hence reduces the degree of fault tolerance of an application [19].

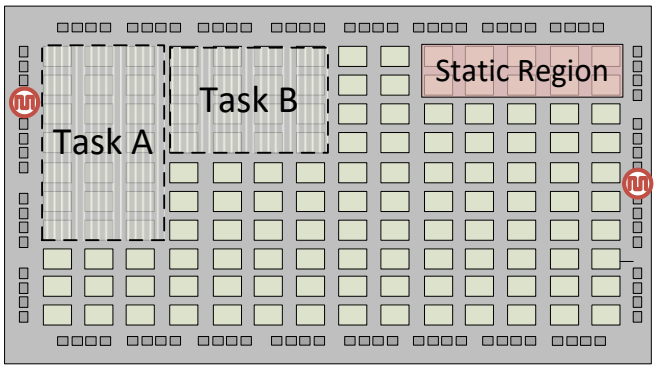
However, the challenge of non-slotted computing model is the associated difficulty of achieving communication among RMs and the FPGA ports, as well as the complex allocation process. Figure *3.1* illustrates advantages and disadvantages of slotted and non-slotted reconfigurable computing architectures.

To address the challenge of inter-tasks communication, R3TOS explored the use of the configuration layer for communicating among tasks [61]. Special wrappers are created for each task which essentially enables the task to read its inputs from an input data buffer (IDB) and save its computation results in an output data buffer (ODB). When task communication is required, the internal configuration access point is used to copy data from the ODB of a source task to the IDB of a destination task.

To address the challenge of delivering clock network to a task in runtime after its placement on the chip, R3TOS also uses the configuration layer to route clock networks to RMs [62]. The technique used by R3TOS involved carefully examining the configuration bitstream to identify configuration bits that control the state of the programmable interconnection points (PIP) in the paths of clock nets. This information is used to route clock signals to all clocking points of each flip flops, BRAM and DSPs in a design.



(a)



(b)

Figure 3.1: Slotted Versus Non-Slotted Reconfigurable Computing

a) Slotted Architecture has clearly defined boundaries reducing the complexity of task allocation, communication and clock networks delivery but leads to inefficient resource usage. b) Non-slotted architecture has potentials for better resource utilization but requires more complex runtime task placement management. Clock network delivery to task and task communication are also more challenging

The placement manager (allocator) in R3TOS keeps track of the resources on the FPGA chip and decides location for incoming tasks with the aim of maximizing the utilization of the chip area. Placement locations are not limited by static communication ports or pre-defined boundaries for tasks [63]. Thus, RMs can be placed on any matching location on the chip which is free. The allocator architecture consists of an architecture checker (AC), an empty area descriptor updater (EADU) and an allocator quality evaluator (AQE). The AC checks the feasibility of placing an RM on a location on the FPGA by comparing the resource layout of the RM's architecture and that of potential locations. The AQE computes the quality of potential locations. It does this by comparing the fragmentation contribution of each of the potential placement location and selecting the location with the least fragmentation. The main algorithm implemented by the allocator to optimize the FPGA area utilization is Empty Area Compaction (EAC) algorithm which is a derivative of the Maximum Empty Rectangle (MER) algorithm [64].

Unlike R3TOS, ReconOS [45] is targeted only at high performance applications, and not reliability of critical applications in hostile environments. The key idea of ReconOs is to extend the capabilities of a traditional host OS to support hardware threads [65] [45]. ReconOS manages threads from a software perspective and delegates appropriate threads to the reconfigurable hardware platform when needed in runtime. ReconOS aims to extend the multithreading programming model on processors to a mixture of processors and reconfigurable hardware platforms. Thus, from the application's appearance, a unified hardware and software thread is presented. To use ReconOS for applications, a user first executes a multithreaded application only in software for functionality testing. Secondly, the application is executed on an embedded CPU in a targeted FPGA platform. Thereafter, the application designer uses profiling to identify threads suited to CPU execution, threads suited to be executed on the reconfigurable hardware and threads that can be executed on both. Finally, the designer synthesizes the hardware threads and configures them on the FPGA ready to be executed when required. In runtime an implementation of each task to assigned to either software or hardware when needed depending on the runtime scenario.

ReconOS uses pre-defined slots for executing RMs (hardware threads) in runtime [66] [18]. Thus, the attendant challenges of runtime placement seem not to be a complex one as in the case of R3TOS which uses a non-slotted architecture. In addition, the use of slots simplifies the communication and clock network delivery process. However, as mentioned above, the efficiency of area utilization in slotted (or partition-based) runtime placement architecture is significantly less than those of a non-slotted architecture.

The CAP-OS targets real time applications [46] [67]. It manages tasks in runtime by coordinating the schedule of tasks' reconfiguration, their runtime allocation to specific processing elements (in the form of processors). The objectives of the scheduling and allocation are: meeting tasks' deadline, improve resource utilization and lower power consumption. The OS manages access to the configuration port with a consideration of its configuration overhead. It includes the possibility to reuse tasks to reduce the workload of the configuration port.

The placement management technique presented by CAP-OS involves checking the availability of free processors to execute a requested task, and in their absence a new processor is configured. No detail of area management technique (such as minimizing fragmentation) is presented. This seem to suggest that a partition-based architecture was used. The implementation of CAP-OS was also reported as partition-based in [18].

Like CAP-OS, LEAP OS and RIFFA are also partition-based reconfigurable operating systems and do not give any detail of runtime placement management relating to optimizing the use of FPGA area. Rather, the main focus of LEAP OS is the abstraction of communication infrastructures on the FPGA chip by using the idea of latency-insensitive design presented in [68]. RIFFA, on its part, provides communication and synchronization by offering a consistent and generic interface between hardware and software on FPGA SoCs. Similarly, GOAHEAD [69] is a design-time tool that offers support for runtime reconfigurable systems. No detailed area management strategy was provided.

Finally, RTSM presented a management system that aims to manage both hardware and software tasks on FPGAs in runtime. Although, the technique – like many of the foregoing – is partition-based, the authors use a mechanism named *best fit in space* to improve the utilization of the partitions in runtime. In essence, *best-fit in space* allocates tasks to partitions which leaves the least unused area in the slot. In addition, RTSM includes a form of task reuse in its runtime placement scheme to achieve better overall execution time. This technique, called *best fit in time,* checks if a task is already present in any of the partitions before configuring another copy. However, to support the possibility of a task's partial bitstream being configured on multiple partitions, different versions of the task's bitstream must be stored, a technique which often require large storage.

It is clear from the foregoing that most ROS use the partition-based (slotted) architecture for runtime placement of RMs on FPGAs despite the potentials for a better area utilization with non-slotted architecture. Only R3TOS uses the non-slotted architecture. As identified above, the main reasons for adopting slotted architecture in most ROS revolve around the complexity in developing novel placement techniques that are practicable for ROS. Other reasons include developing communication and clock network delivery to tasks after their runtime placement. In addition to these, another challenge with using COTS FPGA in runtime applications generally is their large reconfiguration overhead.

In the following sub-sections, a review of some previous relevant research efforts towards addressing the challenges of efficient runtime area management, large reconfiguration time and clock routing to tasks after their placement is given. Communication is not addressed in this thesis.

3.1.1 Review of FPGA Area Management in Runtime Placement Systems

The runtime placement of tasks on homogenous reconfigurable hardware have been well studied by many authors. A foundational work in modern efficient management of FPGA area was presented in [64]. The authors presented both a design time and a

runtime algorithm for task placement on the FPGA. The runtime placement involves keeping track of the MERs on the chip area and using either a best fit or first fit technique to decide how to split the area to accommodate a requested task. The aim of the splitting technique is to minimize fragmentation of the chip area. Another notable work is the use of Vertex List Set (VLS) to keep the contour information of free area on the chip [70]. In a later work [71], the authors proposed an adjacency-based heuristic that uses the information in a VLS to determine a location for a task. The MER and VLS techniques are reported to have high accuracies but have high computational overhead [72]. In addition, these techniques are not inherently targeted at heterogeneous FPGAs. Hence, additional computation is required to apply them to COTS FPGAs which are heterogeneous as done by the placement system reported in [63].

Authors in [73], proposed runtime placement algorithms for heterogeneous reconfigurable platforms. One of the main ideas of the placement system is to speed up the process of scanning the FPGA area to find location for an RM using the locations of the heterogeneous resources such as BRAMs and DSPs on the chip. Since there are typically fewer of these resources, the scanning process can quickly decide if the architecture of an available location on the chip matches that of an arriving RM (or task). When multiple possible locations exist on the chip to place a task, the layout of the other tasks on a scheduled queue is checked to see which of the locations blocks the least number of scheduled tasks. That location is chosen for the task.

However, to simplify the problem the allocation problem, the authors assume that the heterogeneous blocks on the FPGA platforms are regularly spaced-out. In addition, their techniques rely on a Virtual Bitstream (VBS) format which is independent of a task's location on the chip. None of these two assumptions apply to conventional COTS FPGAs. The authors proposed a new FPGA architecture for their algorithm. This is a different approach from the focus of this thesis which is to use COTS FPGA architectures rather than propose new ones.

The work in [74] presents two algorithms for the placement of tasks on heterogeneous platforms (targeted at COTS FPGAs). They are Static Utilization Probability (SUP) Fit and Run-time Utilization Probability (RUP) fit. These correspond to offline and runtime phases of placement management respectively. The basic principle of both is that tasks with many potential placement locations on the chips are not placed on locations required by tasks with few potential placement locations. SUP fit generates a utilization probability for each cell on the chip using the matching location of all tasks to be placed. This was done by analyzing overlap graphs of each cell on the chip. The weights of all feasible positions are determined and sorted in ascending order. This is done offline, at the application design stage. At runtime, when a request is made for a task's placement, the next suitable position for the task which is least probably used by other tasks in the set is assigned to the task.

The authors also presented an alternative RUP fit. RUP fit dynamically computes position weight for the cells at runtime instead of the design time approach used by SUP. A drawback of these techniques is that a foreknowledge of all tasks to be placed on the chip in runtime is required for SUP fit. Furthermore, the technique does not consider the distribution of the number of feasible positions among the constituent tasks of an application. Thus, it is prone to a situation where some tasks have abundant areas and others have too little. In addition, updating the position weights in the RUP fit in runtime is quite time consuming.

The work in [75] presents a technique for relocating a design bitstream synthesized for a location with a DSP to another location with a BRAM replacing the DSP. However, the technique is based on online editing of configuration bitstream which is time consuming. In addition, the routing between the DSP and BRAM are required to be identical, and neither the DSP nor the BRAM must be used by the design. Commercial FPGA do not have the same routing for DSP and BRAM resources.

Furthermore, the work in [71] presented an adjacency-based heuristics for measuring and minimizing fragmentation on chip area. Adjacency based techniques are better suited to homogenous chips, as heterogeneous hardware tasks have definite layout

requirements which mean that a matching location to accommodate a task may not be adjacent to another task or the chip boundary.

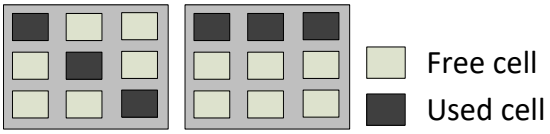
Also, the authors in [76] and [77] presented a design optimization technique to improve place-ability of tasks at runtime using overlap graphs. The main principle of the work is based on the intuition that the placement locations for a task is *determined* by the selected synthesis position in its design (or offline) phase. Thus, synthesis location that are least contested for by other tasks in the application is selected for each task. Since these works are based on minimizing overlap like [74], they are prone to variation in the number of feasible location for constituent tasks.

An integral aspect of runtime placement is the quantification of fragmentation of the chip area. Most existing techniques for quantifying fragmentation on reconfigurable chips are well suited to homogenous ones. They use the assumption that the chips only consist of a single resource type – mostly Configurable Logic Blocks (CLBs). Hence, they are not directly applicable to heterogeneous chips. The work in [63] presented a version of MER which is applicable to heterogeneous chips by proposing additional computation using an architecture checker.

In [78], a metric for fragmentation was presented. The technique measure fragmentation by computing the contribution, f_i of individual slots, i (called hole) using (3.1a). f_i is then used to compute the degree of fragmentation, F of the entire chip using (3.1b). k represents the number of free cells in a slot and N the total number of cells on the chip. An advantage of the technique is that is it fast and hence suitable for runtime placement management systems where fast placement decisions are key. However, the computation of f_i is based only on the *number* of free cells and does not consider the distribution of the occupied or free cells within the hole. Hence the metric cannot discriminate between holes having same number of occupied cells distributed differently. Thus, using the approach as presented, the two slots shown in Figure *3.2* would produce the same value of f_i. Consequently, the approach would not be efficient in deciding the location of a task in runtime.

$$f_i = \frac{k}{N^2} \qquad (3.1a)$$

$$F = 1 - \left(\prod_i f_i\right) \qquad (3.1b)$$



(a) (b)

Figure 3.2: Distribution of Occupied and Free Cells in a Slot

A different approach of quantifying fragmentation was proposed by Handa et al. in [79]. The main idea of the model is to compute the contribution of each cell on the chip to the total fragmentation of the chip. Then, an averaging technique is used to determine the degree of fragmentation of a partition. Using the technique, the fragmentation contribution of a cell is determined by the number of empty cells in its vicinity, both in the horizontal and vertical directions. These are computed using equations 3.2(a) and 3.2(b) where v_x and v_y refers to the sum of the number of empty cells in the horizontal and vertical directions of the cell respectively. The parameters L_x and L_y represent twice the average width and height respectively of the set of tasks to be placed on the chip. As shown in the equations, when there is no empty cell in the vicinity of a target cell in a certain direction (e.g. when $v_x = 0$), the fragmentation contribution in that direction is 1 (maximum).

On the other hand, when the number of empty cells in the vicinity of a target cell is greater than twice the average width (or height) of the tasks being placed, the fragmentation contribution in that direction is 0 (minimum). The total fragmentation contribution of a cell ($TFCC$) is the sum of its horizontal and vertical contributions as shown in 3.2(c). To compute the fragmentation contribution of an area consisting of multiple cells (AFC) such as an area (to be) occupied by a task, or the entire FPGA area, an overage of the fragmentation contributions of the constituent cells is

computed using 3.2(d), where N is the number of cells in the area. Fragmentation contribution is only defined for empty cells as the aim is to determine a potential location for an incoming task.

$$FCC_x = \begin{cases} 1 - \dfrac{v_x}{L_x - 1}, & v_x < L_x \\ 0, & otherwise \end{cases} \qquad 3.2(a)$$

$$FCC_y = \begin{cases} 1 - \dfrac{v_y}{L_y - 1}, & v_y < L_y \\ 0, & otherwise \end{cases} \qquad 3.2(b)$$

$$TFCC = FCC_x + FCC_y \qquad 3.2(c)$$

$$AFC = \frac{1}{N}\sum_{i=0}^{N} TFCC_i \qquad 3.2(d)$$

The technique is in principle more accurate than [78] as it accounts for the state of each cell in a slot. However, the process of computing fragmentation for individual cells in a slot could be expensive and time consuming for a runtime scenario. In addition, the computation only considers the number of empty cells in the vicinity of a target cell and not the distribution of the empty cells. It does not differentiate between cells on the left and right of the target cell. Consequently, the technique does not account for the *isolation* of a slot as a unit from other tasks or the border of the chip in the fragmentation quantification process. To illustrate this, Figure *3.3* shows two chip areas, each with a slot, $Slot\ A$ and $Slot\ B$. Both slots have the same fragmentation coefficient according to [79]. Each slot has 16 cells, and have equal number of empty cells in their vicinity in both horizontal and vertical directions. However, $Slot\ A$ is better aligned to the border of the chip and hence would keep the empty areas on the chip more compacted compared to $SlotB$. In addition, if any of the slots were to be shifted in the same plane (either horizontal or vertical), the proposed technique cannot differentiate between the different locations created.

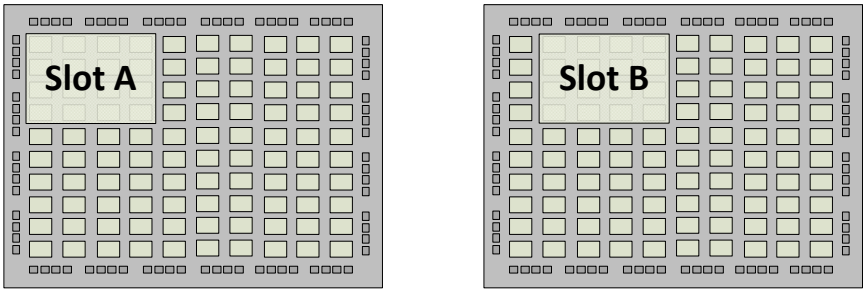


Figure 3.3: Slots with Same Fragmentation Metric but Different Placement Effects

3.1.2 Configuration Overhead Management in Runtime Placement Systems

A major technique required in utilizing FPGAs for runtime applications is the management of their relatively large configuration overhead [38]. Current COTS FPGA have a single internal configuration port which can perform a single task at once [51]. The port is often required to handle many critical duties in addition to writing the configuration memory. An example of such duty is monitoring and correction temporal faults in the configuration memory [27] [80]. Furthermore, when task configurations, which require external memory access operations, are requested frequently, the power consumption of the system could also increase significantly [81]. Thus, it is important to keep the number of reconfigurations as low as possible; not only so that the configuration port can be more available for other critical operations, but also to enhance the performance of the system [59].

Consequently, techniques have been proposed to manage the configuration overhead of COTS FPGAs. The two most important of these as it relates to ROS are: Prefetch and Task Reuse [82]. Prefetch aims to overlap the computation of certain hardware tasks with the configuration of new ones before the new ones are required for computation [73] [83]. Prefetching does not aim to reduce the number of configurations, but rather manages the distribution of the configuration to meet tasks' requirements.

On the other hand, task reuse is aimed at reducing the number of hardware tasks configuration. This is achieved by retaining carefully selected tasks on the chips even after their computation, such that if they are required again in the future, their configuration will be circumvented [38] [23]. This means that the cumulative occupancy of the configuration port is reduced, making it more available for other critical operations [22], while also avoiding frequent external memory access. To this end, many task reuse techniques have been proposed. Notable example are [84] [85] [86] and [38].

Closely linked to the efficiency of any task reuse technique is the policy used to decide which tasks are to be preserved on the chip and which to eject in the case that resources are required to accommodate an arriving task [87] [88]. This is called the replacement policy. As pointed out by [84], an incorrect replacement decision will not only fail to reduce the total number of reconfiguration, but would increase it. Multiple options exist for which task(s) to be replaced on the chip as illustrated in Figure *3.4*.

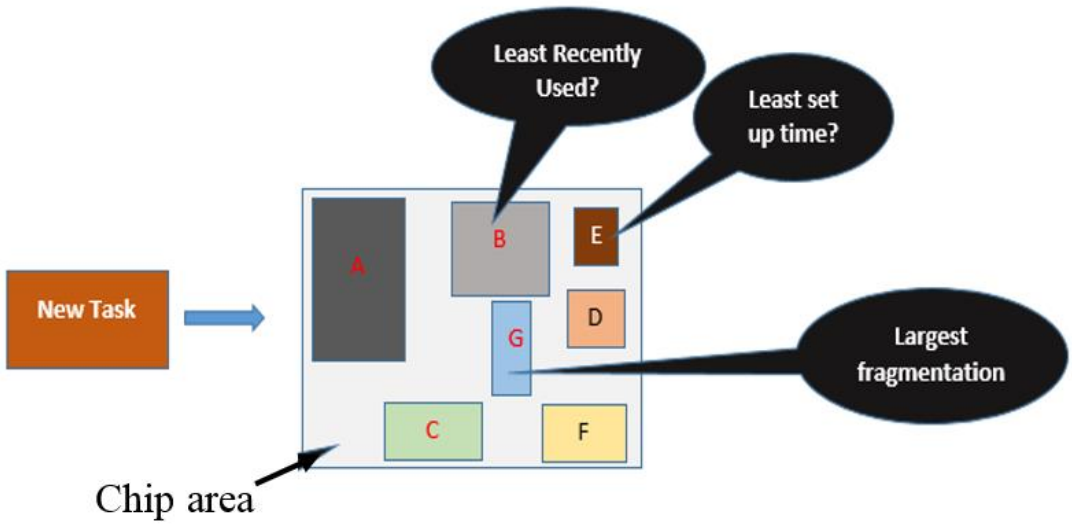


Figure 3.4: Multiple options for Task Replacement

The pioneering work by Compton et al. [84] surveyed a number of replacement policies for different FPGA architectures. These include *Simulated Annealing* (SA),

Least Recently Used (LRU) and *penalty-based* task replacement algorithm. The authors compared the performance of the algorithms on different FPGA types and reported that the penalty-based algorithm had better performance. The penalty-based algorithm used was a combination of the reconfiguration overhead and the number of times tasks were used since their (re)configuration.

In a similar fashion, the authors in [85] proposed a penalty-based replacement technique called Reconfiguration-to-Execution Ratio (RER). To choose a candidate to be replaced, they compute an RER value for each potential candidate as the ratio of its reconfiguration time to execution time, multiplied by execution frequency. However, the architecture proposed by the authors composed of both FPGA and CPU. Hence, before candidates for replacement are evaluated for the FPGA, a factor called *speedup* is computed. The factor measures the performance gained by executing a task on the FPGA over executing it on the CPU. A task can only be replaced by an arriving task if its speedup is lower than that of the arriving task. Thus, only tasks with lower speedups than the arriving task are evaluated for replacement using their RER values. The candidate with lowest RER value is replaced first. The efficiency of their technique was evaluated by comparing it with other placement routines which did not include task reuse.

A Reuse-Based Scheduling (RBS) algorithm is presented in [86]. In the proposed scheme, only *significant tasks* are preserved on the chip. The significance of tasks is computed using the configuration overhead and the probability of recurrence in the future. The configuration cost is computed as the ratio of configuration latency to execution latency, while probability of recurrence of a task, p_i, is computed using equation 3.3. λ_i is the average number of arrived instances of task T_i in the past time interval Δt. The authors indicate that two threshold values k_1 and k_2 are defined for configuration overhead and probability of reuse respectively. A task is designated as significant if both the configuration cost and the probability of reuse exceed the threshold values.

$$p_{\Delta t} = \lambda_i \, x \, e^{-\lambda_i} = p_i \qquad 3.3$$

To minimize fragmenting the chip area due to the preservation of certain tasks while others are removed, the authors propose to have two resizable regions on the chip. One of the regions is reserved for significant tasks (which are preserved) and the other for non-significant tasks (which are not preserved). The sizes of the regions are resized in runtime using the relative number of significant and non-significant tasks. Two replacement policies were surveyed: Best-Fit (BF) and Least Probability of Recurrence (LPR). The first favours the replacement of the task with the smallest size that can accommodate the newly arriving task. LPR on the other hand favours the replacement of tasks with low values of p_i. The authors reported a better performance for LPR.

3.1.3 Review of Runtime Clock Routing Techniques

The authors in [62] proposed an online clock routing technique for a runtime task placement scenario. The technique is based on finding the location of PIPs relating to the clock tree and controlling their states in runtime. The location of the essential configuration bitstream information obtained by reverse engineering experiments was proposed to be used to route clock signals to newly placed tasks and to switch from a failed clock buffer to a functional one. In addition, the authors proposed techniques for dividing clock frequencies of regional clock buffers in runtime, making it possible for tasks designed for different clock frequencies to be placed in the same clock region.

However, one of the drawbacks of their technique is that the number of bits to route a clock signal to a task is quite high, leading to large time overhead in online clock routing. For example, according to the proof of concept of the work and its Table II, to route a clock signal from a buffer to a task occupying a single CLB column on a Xilinx 7 series FPGA chip, a total of 98 bits, distributed as follows is required. First, the appropriate net is selected (1 bit), then the clock signal must be routed to all the sequential components in the column (96 bits). Finally, the clock buffer is switched on (1-bit). These bits are located in different configuration frames and since the smallest resolution of reconfiguration is a frame, a large amount of configuration is required. This is a very significant overhead, given that these operations are carried

out by a single configuration port in runtime, adding to the already large configuration overhead of state-of-the-art FPGAs [38]. A second weakness of the paper is that no practical means of mitigating the reliability issue caused by online bit editing was presented. As discussed in chapter 2, the configuration memory of SRAM-based FPGAs is subject to bit flips (known as transient faults) which are mostly managed using frame ECC values. Changing the values of bits in a frame at runtime without recomputing ECC values would put the reliability of the entire design at risk. Although this was acknowledged in the work, no solution was presented.

The work in [89] also proposed an online bit editing strategy as a means of routing clock networks to circuits placed in runtime. No clear information on the number of bits that need to be activated or deactivated to route a clock signal to a circuit was presented. However, the statement by the author that the clock connection to each CLB in a design need to be examined to route a signal suggests a significant amount of routing bits similar to [62]. In addition, the paper did not offer any means of dealing with the loss of reliability due to runtime editing of configuration bitstream.

3.2 Power consumption on FPGAs

Minimizing power consumption is one of the top goals of many system designers. The desire for low power designs on many systems is not only aimed at reducing energy bills and increasing battery life, but also to increase the life span of the device and its reliability, as well as reduce the burden of cooling systems [90]. Low power consumption also reduces the risk of electromigration. FPGA based applications are not left out. Consequently, many techniques have been proposed to reduce various components of energy consumption of both applications [91] [92] and FPGA platforms [93].

3.2.1 Components of Power Consumption of FPGAs

The main components of power consumed by a circuit on an FPGAs are static power and dynamic power [93]. In addition to these, extra power is drawn during

(re)configuration [94]. This is associated with SRAM based FPGAs. It includes the power of the configuration engine and memory access when configuration is done through the Internal Configuration Access Port.

Static power is consumed by the FPGA even when no active computation is being done by the chip – when no signals are changing [93]. It is basically due to leakage current in transistors. For Xilinx FPGAs, most of the processing elements, including CLBs and DSPs, contribute a constant value to the static power consumption of the device whether they are used by a design or not. However, the BRAMs in the 7 series and ultra-scale devices only contribute to the static power when they are used by the design [95]. Consequently, the static power consumption of many FPGAs tends to be relatively constant for a specific chip irrespective of the circuit configured on it. Generally, static power for FPGAs varies with device size. This is illustrated in Figure 3.5. It shows the power consumption of a Xilinx CORDIC IP computing the hyperbolic Tangent on various Xilinx FPGAs.

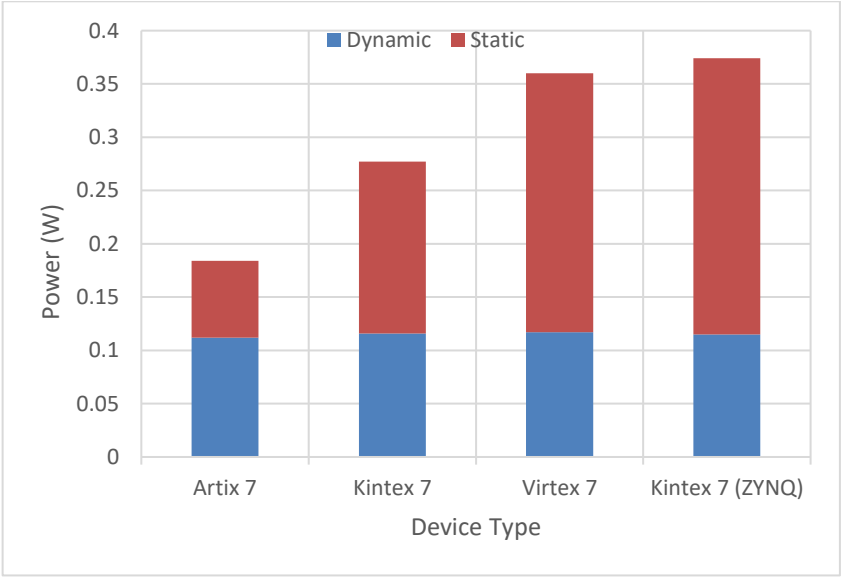


Figure 3.5: Power Consumption Components of a CORDIC Circuit on Different COTS FPGA

Table *3.1* shows a summary of the resources present on the devices. As shown, the static power of the chips in Figure 3.5 varies with the amount of resources present in the chips shown in Table *3.1*. Techniques to minimize static power in not considered here. Rather, to minimize it, the smallest device size that can fit an intended design is selected.

Table 3.1: Resource Distribution of Selected Xilinx's 7 Series FPGA

Device	**IOB**	**LUTs**	**FFs**	**BRAMs**	**DSPs**
Artix-7	106	20800	41600	50	90
Kintex-7	500	203800	407600	445	840
Virtex-7	700	303600	607200	1030	2800
ZYNQ Kintex-7	362	277400	554800	775	2020

Dynamic power is consumed only when signals of a design are switching. Its value is affected by the clock frequency, the loading capacitance and the supply voltage. This is shown in equation 3.4. The term v is the supply voltage, c the capacitance, α the (switching) activity on the components and f the clock frequency. Dynamic energy component is dependent on the specific design implemented on the FPGA, and in many designs, constitute the most significant energy consumption component of an entire system. Therefore, minimizing this component is not only an effective way of saving significant energy, but it is also one that can be achieved in runtime, unlike static energy of the device.

$$P = v^2 . C_l . f . \alpha \qquad (3.4)$$

To minimize this component of energy, the v component may be reduced using adaptive voltage scaling [96] or power gating [97]. It is worth noting that dynamic or adaptive voltage scaling can significantly reduce static power consumption in FPGAs. The work is [96] reported over 85% saving in energy compared to nominal

voltage designs. Also, power savings can be obtained by gating the f term when a circuit is not required to operate, or reduced if possible. In addition, the activity on the circuit may also be minimized [98].

3.2.2 Minimizing Dynamic Power Using Memoization

Memoization involves reusing the result of a previous computation when a request is made for computation with the same set of inputs that produced them. Thus, the process of re-computing the result is circumvented – together with its attendant energy consumption.

Although many advantages have been advanced for the technique of using memoization either to speed up computations or save energy, their actual implementation on FPGAs have remained challenging. One of the most important challenge is the balance between the overhead of the memoization logic and the power saving it offers. These overheads include its own energy consumption per transaction. This is important as the memoization circuit is always executed before the original circuit for each transaction. Therefore, to benefit from the technique, it is important to ensure that the memoization block's overhead is significantly less than the circuit it is meant to work with, and that a high Miss rate is avoided. To maximize the benefit of the technique, the block should also be able to carry out a fast comparison of input(s) in a limited number of clock cycles. This would reduce its energy consumption per transaction as well as reduce the total delay in the application's path, especially in the case of a Miss. A greedy search procedure of the block's memory would greatly increase the energy consumption per transaction, even for a moderately sized memory.

3.2.3 Review of Memoization Techniques for Low Power on FPGAs

Memoization techniques in processors and declarative languages have been well investigated. The work in [99] gives a good summary of memoization as it relates to software scenarios. In this section, the focus would be the implementation and

evaluation of the technique as it relates to FPGA platforms. The author is not aware of any prior work that gives the low-level implementation details of a memoization block which are applicable to FPGA-based proprietary IPs as presented in this thesis.

Many prior works regarding power saving in hardware circuits on FPGA have used approaches involving the design of imprecise hardware. The works in [91] and [92] are good examples. The imprecise hardware generally produces acceptable results while their power consumption is significantly lower compared to circuits producing accurate results. The authors in [100] presented a similar technique. However, unlike [91] and [92], an imprecise hardware is generated from an original behavioural (RTL) description of the circuit and not manually. They reported an up to 50% saving in power. These approaches are different from the approach in this thesis since they are focused on altering the architecture of the original circuit, and do not involve any memoization.

The work in [101] uses a different approach. Although new circuit architectures are generated as in [100], the new circuits use memoization to achieve lower power in addition to their imprecision. The work presented a design flow that uses a high-level synthesis tool to generate memoization based circuits. The technique involves specifying an input C routine together with a threshold for accuracy and power. Using these, the flow iteratively synthesizes hardware circuits until the specified constraints are met. The quality of each of the circuits that meets the specified criteria is then evaluated by computing the ratio of its dynamic power saving to its area overhead (power saving per Area, PSPA). The circuit with the highest PSPA is chosen as the best candidate. The experimental results presented show up to 20% saving in dynamic power consumption. Their technique relies heavily on the accuracy of the test data provided. It does not benefit from any runtime information of the system as it is completely a design time approach. For instance, if the accuracy threshold changes during the runtime phase of the circuit, there is no means of using this information to improve the design. This could lead to design failure or spend resources and power needlessly. In addition, the quality (in terms of resources) of circuits generated using HLS is known to be lower than designing directly with HDL. Also, their technique

does not provide a means of incorporating memoization and approximation capabilities into a proprietary IP which one simply wishes to reuse without changing its architecture. In [102], the implementation of a memoization technique for an image processing application was presented. The design presented is very specific to the application, and no detail is given about extending the technique to other applications.

3.3 Chapter Conclusion

In this chapter, runtime placement management techniques for reconfigurable computing have been reviewed, including the underlying aspects of runtime placement such as fragmentation and managing reconfiguration overhead. In addition, power minimization techniques in FPGAs were discussed with a focus on reducing dynamic power consumption using memoization.

As shown in the review above, a large percentage of existing runtime placement management systems are not targeted at state-of-the-art COTS FPGAs. They either assume a homogeneous architecture, or a regular heterogeneous architecture, both of which is not the case with COTS FPGAs. Although few existing works such as [74] and [75] targets COTS FPGAs, the techniques presented require a foreknowledge of all tasks to be placed on the chip in runtime or can have prohibitively high execution times, which are not suitable for runtime scenarios. In addition, they are not optimised for reliability.

Similarly, existing techniques for managing the relatively large configuration overhead of COTS FPGAs does not include a consideration of the spatial features of the chip area such as ongoing fragmentation. A shown in chapter 5 of this thesis, addressing ongoing fragmentation leads to better configuration overhead management.

Furthermore, existing techniques for routing clock networks to hardware circuits in runtime are not only expensive, taking up a significant bandwidth of configuration port, but also do not address a major reliability concern associated with the method

used. In chapter 8, a better online clock routing scheme which addresses these two limitations of existing runtime clock routing technique is presented.

In the next chapter, efficient design-time optimization techniques are presented. The optimizations are aimed at improving the quality of task placement achieved in runtime, increase the reliability of applications and improve the efficiency of computation in a dynamic reconfiguration environment.

Chapter 4: Offline Design Optimization for Efficient Runtime Placement and Reliability

Efficient offline optimization of designs is central to runtime placement management system for high quality and fast placement decisions. In many circumstances, runtime placement decisions are required to be made fast so as not to impact the performance of the overall application in a dynamic reconfiguration environment. For many reconfigurable hardware such as COTS FPGAs, this mean that it is often infeasible to change the physical layout of the hardware task in runtime as (re)synthesis and implementation of tasks take a long time. Hence, for fast and efficient placement, the layout of tasks should be optimized such that pre-synthesized tasks (in form of their partial bitstreams) can be efficiently placed on the chip in runtime with minimum adjustments.

In addition, the programmable logic of COTS FPGAs are practically heterogeneous. This is due to the presence of hard blocks such as BRAMs and DSPs which are spread around the chip often in an irregular manner. They place greater limitations on runtime placement of hardware tasks on heterogeneous chips compared to their homogeneous counterparts. Their presence limits the maximum number of possible locations on the chip where a pre-synthesized circuit can be placed in runtime. Unlike task placement on homogeneous FPGAs where the task's area requirement is simply to be satisfied in terms of length and breadth of task alone, on heterogeneous chips, the layout – that is, the specific order of the resources – also need to match those of the original implementation location of the task. Therefore, it is important to optimize a hardware task at design time to obtain better placement quality in runtime.

In this chapter an offline optimization flow is presented. The proposed flow aims not only to improve the maximum number of locations for each task on the chip, but also achieve a fair distribution among all tasks which will share the chip area concurrently in runtime. Furthermore, during this phase, the task is provided with a wrapper to support communication after placement. In addition, an optional wrapper based on memoization for low power computation is proposed for tasks with low port width.

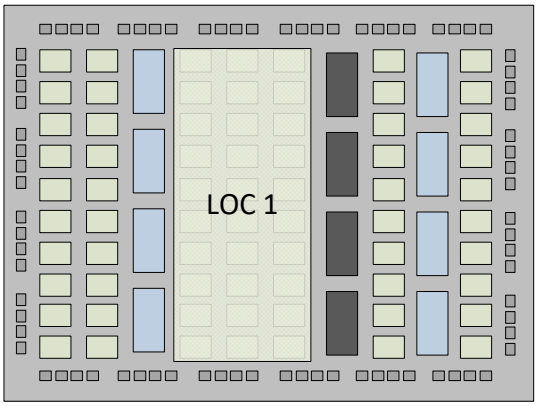
4.1 Offline Optimization to Improve the Number of Placement Locations

In reconfigurable computing, hardware tasks are usually pre-synthesized in an offline design, ready to be loaded on demand in runtime. This is necessitated mainly by the large synthesis and implementation time required to turn an RTL designs into configuration bitstreams, which could be in the order of hours [103]. However, one limitation that comes with pre-implementation of a task offline is that once implemented, most of its features are fixed and cannot be easily changed in runtime. For example, its *shape* and *layout* remain fixed. However, this offline design stage can be harnessed to optimize some of the features of the tasks to make their performance in runtime more efficient.

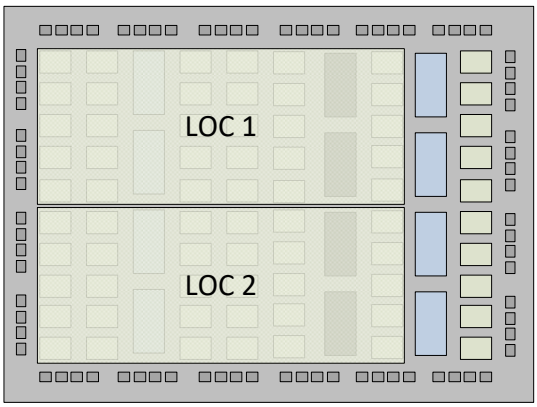
With regard to runtime placement on the chip, the maximum number of locations on the chip on which a task can be placed in runtime is determined by the implementation location chosen during the offline stage of the task design [76]. Figure 4.1 illustrates this point. The two floor plans shown are intended for a hypothetical circuit with a utilization of 30 CLBs. Using the floor plan in Figure 4.1(a), only one location (LOC 1) exists for the task in runtime. Should this location become permanently damaged or be occupied by another task in runtime, it would become infeasible to execute the task on the chip. On the other hand, Figure 4.1 (b) shows that the same task can be floor-planned differently such that two non-overlapping locations (LOC 1 and LOC 2) are available for the task. This later floor plan gives the task a higher chance of being placed on the chip in runtime in the presence of other tasks and the possibility of permanent damage occurrence. Though, it is worth noting that the later floor plan comes with an overhead of a slightly higher overall area. Thus, it is important to select implementation location for each task that would maximize the chance of finding a location for the tasks on the chip in a dynamic runtime scenario.

In the following sub-sections, a flow that aims to select optimized implementation location for a set of tasks sharing chip area simultaneously is presented. It is a set of

stages that transforms RTL description of tasks to their configuration partial bitstreams.



(a)



(b)

Figure 4.1: Implementation Location Determines Number of Runtime Placement Locations

a) Only 1 location available
b) 2 locations available

The optimization procedure is divided into 5 stages which are:

a) Synthesis (using Vivado IDE) to obtain estimate of resource utilization of task(s)
b) Conversion of the resource estimation to FPGA resource columns
c) Determination of optimized implementation location for task(s)

d) Execution of script-based Partial reconfiguration routine for partial bitstream generation

e) Copying of bitstream to a desired bitstream storage for ready for runtime execution

These stages are shown in Figure 4.2. The details of each stage are provided below.

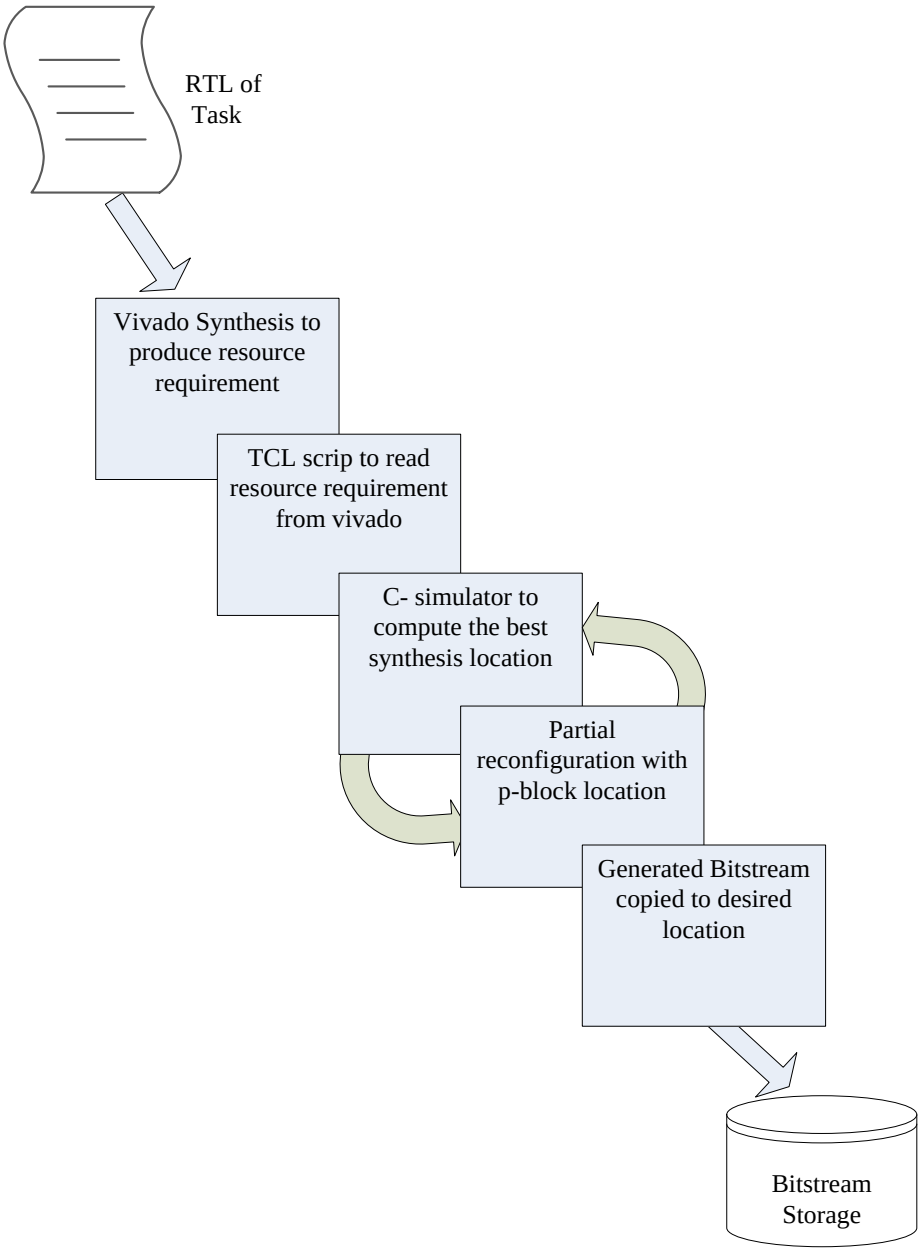


Figure 4.2: Stages of Offline Optimization of Tasks

4.1.1 Initial Synthesis to Determine the Resource Utilization of Task(s)

Commercial FPGA CAD tools can be used to transform an RTL description of a task to a netlist. The Xilinx Vivado IDE is a good example. After synthesis and implementation step, an estimate of the amount of FPGA resources required to implement the task is also reported. With the default setting of Vivado, this is available in the project folder, and can be read directly with a script. As an example, Figure 4.3 shows a section of the resource utilization report after implementation of a CORDIC application implemented using Xilinx's CORDIC IP [104]. This report provides a good starting point in the selection of a suitable implementation location for a task. First, it provides an initial estimate of the amount of resources required by the task. The estimate obtained by this step is refined to obtain a suitable resource layout and subsequently used to select an implementation location by further stages of the optimization flow described below.

4.1.2 Conversion of Resource Estimation to FPGA Columns

The resources reported by Vivado CAD tools, usually in terms of numbers of FFs, LUTs, BRAMs and DSPs, are converted to device rows and columns. To simplify the optimization analysis and implementation, the resource utilization is converted to number of columns and rows, assuming that the task physical area will have a minimum resolution of 1 device column and 1 device row.

It is worth noting that the task could be easily designed such that the number of columns can include fractions of a column, with the resource utilization contained in a section of a column without occupying an entire column. That is, without making the resolution of the task's width equal to the row height of the chip. However, selecting an entire column of resources, aligned to the height of the clock region of the device has three major advantages:

a) The configuration engine is naturally aligned to the configuration of an entire device column within each row [27]. Although, it is possible to choose task configurations that overlap clock regions heights, it means that when the

tasks are to be configured, any other tasks in a shared device column must be stopped, their context saved and their bitstream $ORed$ with the new task's configuration bitstream. This is a time-consuming process which, in addition to disrupting the execution of already computing tasks, adds to the challenge of large reconfiguration overhead of COTS FPGAs as explained in chapter 5 of this thesis.

```
| Tool Version : Vivado v.2015.1 (win64) Build 1215546 Mon Apr 27 19:22:08 MDT 2015
| Date         : Wed Mar 07 18:50:55 2018
| Host         : ENG-4934 running 64-bit Service Pack 1  (build 7601)
| Command      : report_utilization -file CordicTasksGeneric_utilization_placed.rpt -pb
CordicTasksGeneric_utilization_placed.pb
| Design       : CordicTasksGeneric
| Device       : xc7a35t
| Design State : Fully Placed
------------------------------------------------------------------------------------------------
---------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Slice Logic Distribution
3. Memory
4. DSP
5. IO and GT Specific
6. Clocking
7. Specific Feature
8. Primitives
9. Black Boxes
10. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 2226 |     0 |     20800 | 10.70 |
|   LUT as Logic             | 2213 |     0 |     20800 | 10.64 |
|   LUT as Memory            |   13 |     0 |      9600 |  0.14 |
|     LUT as Distributed RAM |    0 |     0 |           |       |
|     LUT as Shift Register  |   13 |     0 |           |       |
| Slice Registers            | 2920 |     0 |     41600 |  7.02 |
|   Register as Flip Flop    | 2920 |     0 |     41600 |  7.02 |
|   Register as Latch        |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                   |    0 |     0 |     16300 |  0.00 |
| F8 Muxes                   |    0 |     0 |      8150 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Figure 4.3: Section of a Typical Resource Utilization Report from Vivado IDE

b) It leads to a significant reduction in the amount of time required by the online placement routine to scan the chip area to find a location for a task. A reduction of approximately 50 times is obtained on the 7 series FPGA chip.

c) The memory required to keep the state of the FPGA resources by the runtime placement system is significantly reduced. This is also in addition to a proportional reduction in the amount of time needed to update the state of the chip in memory after each placement activity.

The area required by the task is computed in terms of number of target device columns (aligned to a clock region height) using (4.1).

$$N_{CLB\_col} = \max\left(\left\lceil\frac{n_{FF}}{P}\right\rceil, \left\lceil\frac{n_{LUT}}{Q}\right\rceil\right)$$
$$N_{BRAM\_col} = \left\lceil\frac{n_{BRAM}}{R}\right\rceil \quad (4.1)$$
$$N_{DSP\_col} = \left\lceil\frac{n_{DSP}}{S}\right\rceil$$

Where $N_{CLB\_col}, N_{BRAM\_col}$ and N_{DSP_col} refer to the number of CLBs, BRAMs and DSP columns respectively required by the task. The terms, $n_{FF}, n_{LUT}, n_{BRAM}$ and n_{DSP} denotes the number of flip flops, LUTs, BRAMs and DSPs obtained from the synthesis report of the RTL explained in section 4.1.1 above. Also, P, Q, R and S are the number of the respective primitives in a column of the device. For the 7 series, these are 800, 400, 10 and 20 respectively. As an illustration, Table 4.1 shows a conversion of the resource utilization of the data processing tasks of a NASA JPL Fourier Transform Spectrometer (FTS) application [105] and the corresponding number of device columns estimated using 4.1. These initial estimations are fed into the next stage of the proposed flow to determine optimized implementation locations for the tasks shown.

Table 4.1: Resource Utilization of JPL Spectrometer Application and Corresponding Number of Device Columns*

Tasks	n_{FF}	n_{LUT}	n_{BRAM}	n_{DSP}	n_{CLB_col}	n_{BRAM_col}	n_{DSP_col}
STAT	576	868	3	15	3	1	1
FFT	20,521	18,325	66	132	46	7	7
ZPD	1,080	9,729	14	32	25	2	2

*The resource requirement for task communication have been included (Details section 4.2)

4.1.3 Determination of the Optimized Implementation Location of Task(s)

For certain design scenarios, tasks are known to occupy the FPGA area alone or are such that other tasks which could share the chip area with them concurrently are unknown at design time. In that scenario, an implementation location is chosen to maximize the total number of such positions present on the chip. On the other hand, as in most practical cases, tasks sharing the chip area simultaneously (or at overlapping times) are known. For this later case, implementation positions are selected such that:

a) a maximum number of non-overlapping positions for each task is obtained
b) An optimum distribution in the number of location of the tasks is achieved.

This is done iteratively as summarized by Algorithm 4.1. First, a function $DetRsIds()$ is used to determine all possible implementation position for each task. These are saved in an array (Array). This is an iterative process (line 3 - 5), with the function returning only after a valid location is found, or the end of the chip is reached. The value returned by the function is the index of the start column of a matching location of the chip ($RsId$), the length of the task (l), and its width (w). These are collected in an array indexed using $RsId$ values. Thereafter, for each combination of implementation location for the constituent tasks, a function ($Quality()$) computes a measure of the quality of placement achievable using the selected locations (line 11 – 24). The terms $k_1, k_2, \dots k_n$, refers to the number of distinct possible start locations for each task.

The function, $Quality()$, consists of another function, $scan()$ which scans the chip area to determine the number of locations matching each of the current implementation positions on the chip (n_i). This is then used to compute a term that measures the quality, q of the selected implementation locations. Equation 4.1 shows how q is computed. The process is repeated until all possible combinations of implementation locations for the constituent tasks have been examined, each time comparing the current q to a previous value, Q and updating the value of Q when a better combination of implementation location (i.e. a higher value of q) is found. In

addition, an array ($opArray$) keeps the potential implementation locations of the tasks corresponding to the value currently stored in Q.

It is worth noting that for the 7-series device, a pair of resource column share a common set of routing resources. Typically, all columns of CLB, BRAM and DSP have orientations designated as *left* or *right*. The general and clock routing networks are located between a left and a right column as shown in Figure 4.4. This technique helps to improve density of resources on the chip, improving the quality of automatic place and route operations [106].

Algorithm 4.1: Pseudo code for selecting optimized implementation location for improved placement

Inputs: FPGA Model, Number of Tasks in application (N), Resource requirement for each task (expressed in number of columns e.g. of CLBs, BRAMs and DSPs)

Output: Task Layout ($oppArray$) $T_i = \{l_i;\ w_i;\ RsId_i\}, i = 0\ to\ N-1$

1. **for** ($i = 0$ to N)**{**
2. $k = 0$
3. **while** ($RsId[k] \neq Null$)**{**
4. $RsId[k] \leftarrow DetRsIds(N_{CLB_{col}}, N_{BRAM_{col}}, N_{DSP_{col}},\ FPGA\ Model)$
5. k++
6. **}**
7. $Array[N] \leftarrow \{RsId[k]\}$
8. **}**
9. OpArray [N] $\leftarrow Null$
10. $Q = 0$
11. **for** (i = 0 to k_1) {
12. for(j = 0 to k_2){
⋮
17. for(k = 0 to k_n){
18. q_i= $Quality$ ($Array[k_1], Array[k_2] \ldots Array[k_n]$, FPGA Model)
19. **if** ($q_i > Q$){
20. $Q \leftarrow q_i$
21. OpArray $\leftarrow$ {Array[k_1], Array[k_2], …Array[k_n]}
22. **}**
23. }
24. }
25. **}**

However, it introduces a constraint in the location of P-block for partial reconfiguration. To use the Vivado PR-flow, the boundaries of a P-block cannot be located between a left-right pair of resources; a P-block must begin with a column of resource with *left* orientation and end with a column with *right* orientation. Consequently, the l parameter is increased by 1 in the right direction whenever a potential location ends on a column with *left* orientation.

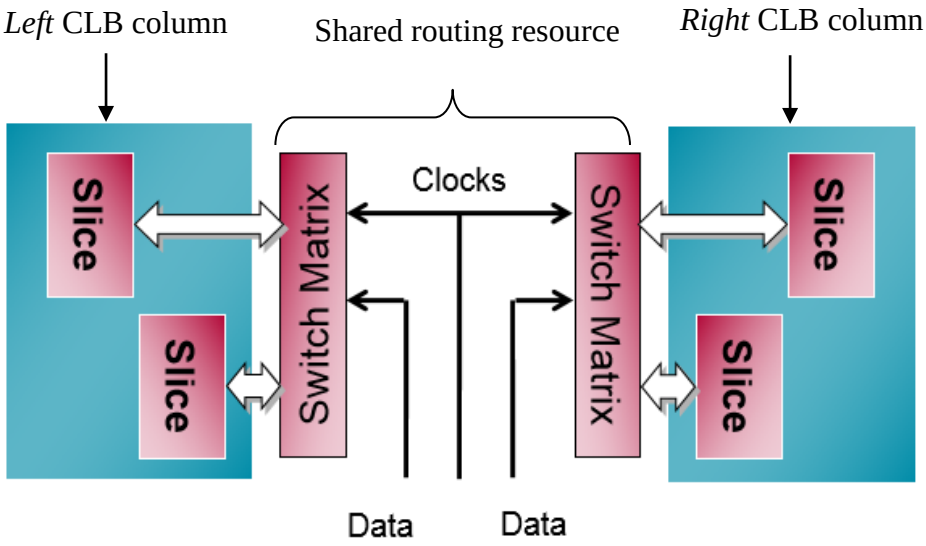


Figure 4.4: Routing Structure in a pair of CLB Columns of Xilinx 7 Series FPGA *[106]*

The quality term, q in Algorithm 4.1, is computed using equation 4.2 for all combinations of implementation locations for the constituent tasks. The set of locations with the maximum value of q is used for implementing the tasks and generating partial bitstreams. In Equation 4.2, n is the number of tasks occupying the chip area concurrently, L is the sum of locations for the tasks, EL is sum of non-overlapping locations, ϵ is a small constant term to avoid ambiguous 0 products, and y is chosen to give relative significance to the terms. In the simulations, y=2 was used to give a greater significance to the sum of non-overlapping locations (EL) over the sum of all locations (L). Algorithm 4.1 takes the model of the FPGA, number of task (N), and resource requirement of each task as inputs. The model of the FPGA is

required for the scan operation. During the scan operation the layout of the task is compared to segments of the chip to determine a matching location for the task.

The FPGA platform is modelled using equation 4.3. The model defines the heterogeneous columns in the chip which are reconfigurable. The final output of Algorithm 4.1 is an array of the optimized layout information of each task, consisting of the start column index of the task (called $RsId$ for the rest of this thesis), the length (l) and width (w) of the task expressed in the number of adjacent device columns and rows respectively required by the task. The parameters l and w specify the area requirement of a task. $RsId$, specifies a column on the target chip such that the next $(l-1)$ contiguous columns to the right of $RsId$ is the matching position for the task on an intended heterogeneous chip with the optimum number of locations on the chip. This parameter makes the model applicable for heterogeneous tasks and chips.

$$q = \left[\left(\sum_{i=0}^{n} L_i\right) + \left(\prod_{i=0}^{n} (EL + \epsilon)_i\right)^{y}\right] \tag{4.2}$$

$$P = \left\{L\ x\ W;\ (B_i), (D_j) \,\middle|\, i \neq j; 0\ \leq i, j, \leq L\right\} \tag{4.3}$$

To illustrate this step, the resource utilization of the data processing tasks of a NASA JPL spectrometer application developed in [105] on Xilinx's xc7z100ffg900-2 FPGA chip (shown in Table 4.1) was fed into Algorithm 4.1. Figure 4.5 gives an overview of q values for some of the combinations using the technique described above. Only combinations with q ≥ 0.2 have been shown for clarity purposes. The values of q were computed with (4.2) using a heuristic approach with ϵ chosen to be 0.1 and $y = 2$. Possible combinations are represented on the horizontal axis. The number of non-overlapping locations (EL) for each of the three tasks are shown in bars, utilizing the vertical scale on the left, while the total sum of locations (including overlapping ones) for tasks (L) and the computed Quality, q values use the right vertical axis. As can be seen, combination 6 which gives both the peak quality and maximum sum is the

preferred combination. The individual tasks are then assigned locations on the chip corresponding to combination 6.

To validate the effect of the optimization technique, the ability of the tasks in Table 4.1 to cope with errors by relocation to different locations on the chip in the event of permanent damage on the chip was evaluated. 1000 set of 200 errors were simulated and applied to two different implementations of the tasks in Table 4.1. The first implementation used the maximum number of potential locations as criteria to determine the implementation positions of the tasks. In this case, several layouts of each task were constructed and the number of potential locations for each layout was computed. For each task, a matching location of its layout with the highest number of potential locations was chosen as implementation location. The implementation location of the second implementation of tasks used Algorithm 4.1 described above. During the simulated error injection for both implementations, if a location occupied by a task is affected by an error, the task is relocated to a different location on the chip. Figure 4.6 shows the number of errors each implementation survived before failure. The assumption used was that a system fails if any of its component tasks can no longer be relocated on the detection of a fault at its current location. As shown in the figure, it was observed that using the proposed optimization technique improves the relocation capability of the tasks of an application compared to selecting implementation locations that only maximizes the total number of locations for each task. An average of 48.6% more errors were survived due to relocation.

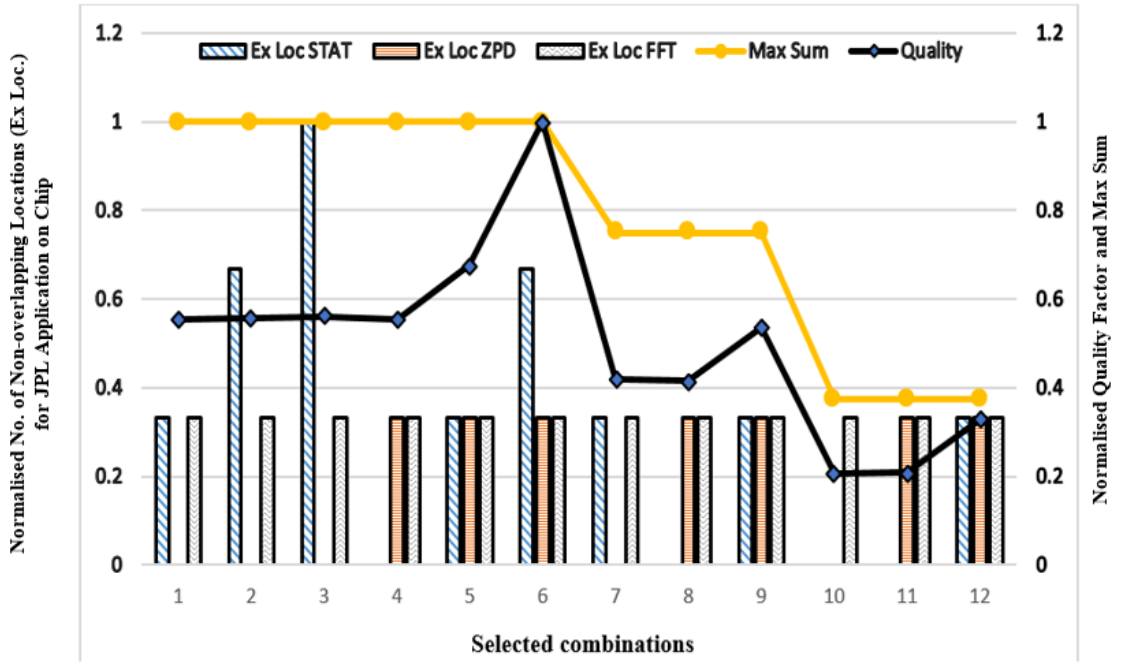


Figure 4.5: Optimal Implementation Location Selection for Spctrometer Tasks on Xilinx's 7z100 Chip

It is important to state that the combination in Figure *4.5* refer to the set of potential implementation location for JPL application consisting of CLB, DSP and BRAM columns. For each of the three tasks, all the matching locations are determined by comparing the resource requirement of the task to the model of the chip layout. All possible combinations of the chip are then evaluated by computing the quality term described above. A $'C'$ routine was written for the evaluating the combinations. Only the twelve best combinations have been shown in the figure for clarity purposes.

*Using same comparison base as [77]

Figure 4.6: Effect of Offline Optimization on Tasks Number of Successful Relocation

4.1.4 Execution of Script-Based Partial Reconfiguration Routine for Bitstream Generation

TCL scripting is used to execute a partial reconfiguration routine to generate partial bitstreams for the tasks. The script is essentially based on the tool-chain of the Xilinx's Vivado 2015.1, augmented with information from the optimization procedure above. In addition to the standard partial reconfiguration flow (PR-flow) of Vivado, the TCL script created for this step includes two additional features:

a) This step takes the optimized implementation location of the tasks as input and uses them to draw p-blocks for the process.

b) Allowance is made for overlapping optimized implementation location (p-block) locations.

The second feature is necessary as the output for stage 3 could include overlapping positions as optimal implementation location. The standard Xilinx DPR flow cannot

handle overlapping p-block locations. To support this feature, the augmented script considers alternate matching locations for the task on the chip and selects it as a synthesis location if it does not overlap another tasks' location. In the case when no other identical location exists, the tasks with overlapping implementation locations are placed on the chip one after the other and PR-flow repeated for each.

It is important to note that partial reconfiguration designs often have additional area overhead compared to flat designs that does not use partial reconfiguration [107]. The actual additional resource requirement varies from design to design [36]. Examples of factors which lead to increased resource utilization is: number of interface pins and shape of the p-block. In addition, the application of partial reconfiguration constraints such as "Contain Routing" mean that the density of the resource in an area is less than that of an equivalent flat design [36]. Therefore, it is possible that the bitstream generation process could fail due to the P-block size estimated using algorithm 4.1 (in section 4.1.1) being smaller than the resource requirement of the PR-procedure. To account for this, the stages 3 and 4 in the proposed flow are repeated until stage 4 is successful. That is, when stage 4 fails due to insufficient P-block size, stage 3 is repeated.

To re-execute stage 3 after a failure resulting from insufficient P-block size, additional column(s) is added to the failing P-block. The amount of resource added is the minimum resolution supported by the target FPGA family. For the 7 series chips, this is often a pair of columns due to the routing structure discussed above (see Figure 4.4). It is worth stating, however that the chances of failures due to insufficient P-block resources in the stages described here-in is low. In fact, for the spectrometer application shown in Table 4.1, there was no failure in stage 4 as the implementation location outputted by algorithm 4.1 was sufficient to generate partial bitstreams for the tasks. Two factors leading to the low chance of this failure occurring are:

a) The computation of the number of columns takes the upper bound of the number of columns (this can be seen in equation 4.2).

b) The resources computed using (4.2) are aligned to end on a device column with a right-orientation, thereby increasing the amount of estimated resources by a whole device column (within each row) in some cases

Thus, the outputs of the Algorithm 4.1 already contain extra resources which in most cases can cater for the resource overhead due to the PR-flow. However, in cases where failure occurs, step 3 is repeated.

4.1.5 Configuration Bitstream Storage and Task Model

The bitstreams of all tasks are each assigned a serial number and saved in a memory off-chip. Without loss of generality, the DDR memory present on the Xilinx's 7z100 was used in this thesis, although any suitable off-chip memory could be used in a similar fashion provided a controller is available to transfer the bitstream to the chip in runtime. In addition to this, an on-chip memory is used to store runtime information about the tasks including its spatial properties: $RsId$, l and w; as well as its temporal properties: configuration time (t_c) and execution time (t_e). The parameter, t_c refer to the time to setup the task on the chip by a configuration manager; t_e is the duration required by the task for active computation. An additional timing parameter, task deadline (t_d) is included in the task model, however, its value is determined in runtime. The deadline, t_d, is the maximum time before which a task's output must be available to be useful.

The values of the parameter $RsId$ for the tasks are obtained after stage 3 (section 4.1.3), while the l and w are obtained after a successful PR in section 4.1.4. The t_c of the task is computed using the:

a) number of frames in the configuration bitstream and
b) timing characteristics of the configuration controller.

The number of frames in the configuration partial bitstream of a task is dependent on both the composition and size of the implementation location selected for the task. Table *2.4* (chapter 2) shows the number of configuration frames in Xilinx's 7 series FPGA for each selectable type of device pair. It is worth noting that there are 128

BRAM content frames per column for *partition* selections including BRAM columns, as against 72 frames for 2 columns of CLB. Thus, a smaller area including BRAM columns have larger partial bitstreams compared to equivalent area composed of only CLB columns, and thus have larger configuration time.

The configuration controller used in this thesis [27] has the timing characteristics given in equation (4.4) and (4.5) respectively for a non-BRAM frame and a BRAM frame for the same FPGA series. Where N is the number of frames to be written to the configuration memory, M is the number of instances of the task to be configured and t_c is the number of clock cycles required to write the configuration memory with N frames and M instances. The model also accounts for the initial data in the pre-amble section of the configuration bitstream, thus N is the number of frames in the configuration data of the partial bitstream which is determined purely by the properties of the reconfigurable resources. For example, to configure one instance of a task with $w = 1$ and l=2 consisting of a CLB-BRAM pair, a total of 25676 clock cycles is required. This consists of 8230 clock cycles for writing the 64 non-Bram frames and the remaining 17446 clock cycles configure the 128 BRAM content frames.

$$t_{c(Non-Bram\,Frame)} = 27 + 128N + 11M \qquad (4.4)$$

$$t_{c(Bram\,Frame)} = 19 + 136N + 19M \qquad (4.5)$$

The parameter t_e for a task is measured by executing the tasks and examining its characteristics. For tasks with varying execution time, the worst-case value is chosen. The deadline parameter (t_d) is determined in runtime when the task is scheduled by a top application. Thus, the task, modeled as a collection of these six parameters as shown in (4.6).

$$T = (l, w, RsId, t_c, t_e, t_d) \qquad (4.6)$$

Table 4.2 is an example of these parameters for selected hardware tasks after the execution of the steps 4.1.1 to 4.1.5 described above. The components tasks of each of the two applications shown were optimized such that they can be placed on the chip area currently in an efficient way under runtime scenarios. It worth noting the adjustments made by Algorithm 4.1 to some of the parameters of the tasks compared to their initial values in Table 4.1. For example, the sum of the number of resource column for the STAT task is 5 from Table 4.1, however, its final output after step 3 and 4 of the flow was augmented to 6 as shown in Table 4.2. This was because the p-block for the STAT task attempted to split interconnects between a *left-right* pair of resources.

Table 4.2: Example of Task Hardware Parameters after Optimization Steps*

Application	**Component tasks**	$\boldsymbol{l}$	$\boldsymbol{w}$	$\boldsymbol{RsId}$	$\boldsymbol{t_c(\mu s)}$	$\boldsymbol{t_e\ (\mu s)}$
FTS Application	STAT	6	1	2	430.84	100
	FFT	76	1	4	4629.24	200
	ZPD	30	1	82	1587.96	750
CORDIC	Square Root	2	1	8	92.54	15
	Sine/Cosine	2	1	16	92.54	19
	Hyperbolic Tan	6	1	0	441.08	56

*The resource requirement for task communication have been included (Details section 4.2)

4.2 Communication Interface Wrapper

A non-slotted model is used for the placement techniques presented in this thesis. Thus, tasks are not constrained to pre-determined slots. This helps to improve the utilization of the chip area. The alternative model, slotted architecture, is faced with the challenge of internal fragmentation as pointed out in chapter 3. Slotted ROS architecture such as [45] places pre-synthesized circuits in pre-defined slots in runtime. The slots are fixed in size and resource layout and can accommodate one hardware task at a time. There are several disadvantages with the slotted architecture. First is the determination of an appropriate size of the slots as hardware tasks would

generally have different sizes and layout requirements. In addition, the use of slotted architecture is susceptible to internal fragmentation in the placement of tasks which in turn leads to wastage of resources. This is because a smaller task would have to use an entire slot. Also, should a part of a slot become damaged, the entire slot could become useless. These leads to inefficient resource usage which is a major goal of ROS. However, slotted architecture has the advantage that tasks placed in slots can easily maintain communication and clocking access with other tasks and the FPGA ports.

On the other hand, non-slotted ROS architecture such as [8] has the potential advantage of better area utilization as circuits can be placed on any matching location on the chip. This translates to less fragmentation and better area utilization, as well as better reliability of an application where relocation is used in the avoidance of permanent damage. However, the non-slotted ROS architecture faces two challenges (in addition to more complex area management): maintaining communication with other circuits or with the FPGA ports and clock network delivery to circuits placed in runtime. It has been argued that the slot-less architecture has the potential to have far better performance than the slotted architecture if these limitations could be addressed [8]. To address the challenge of communication, a technique which uses the clock buffers and nets on the FPGA chip for communication is adopted [26]. The technique developed by Adewale, a fellow researcher in the group, adapts the unused clock buffers and nets on Xilinx FPGAs for communication. An overview of the technique is shown in Figure *4.7*. The technique is quite scalable and can be easily adopted for different designs by using a custom wrapper. The wrapper adds an overhead of 249 LUTs and 87 FF for each pair of 32-bit input and output. These resource overheads are included in the resource utilization of the tasks before executing the design optimization flow in section 4.1. Complete details of the communication technique can be seen in [26], [108] and [109].

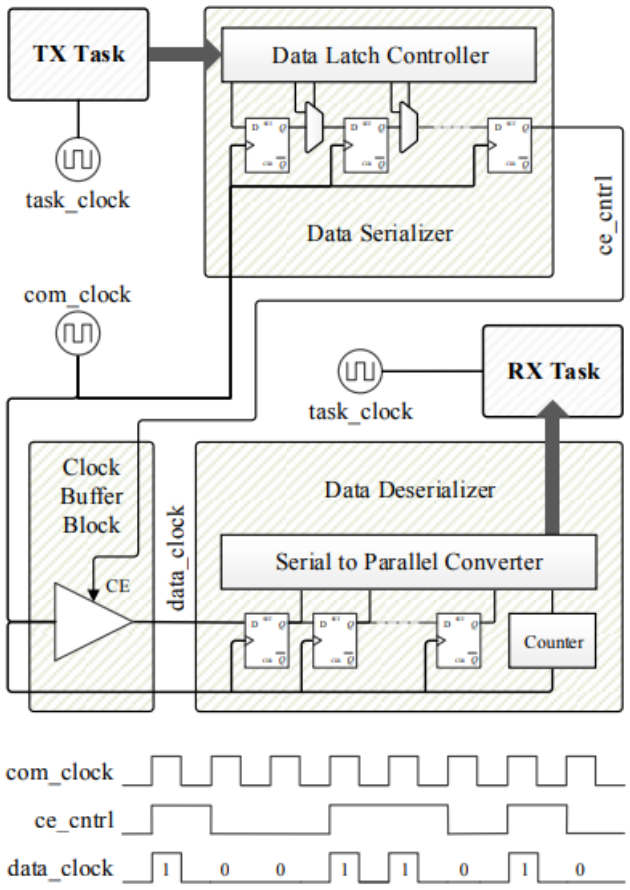


Figure 4.7: Mechanism of Data Transfer Using Clock Buffers as Serial Bit Transceivers *[109]*.
The signal transitions show an example of the transmission of an 8-bit binary data 10011010

4.3 Additional Optimization for Low Power for Low Porth Width Applications

Certain applications can be optimized to benefit from power savings by keeping track of previous computations. In this section, an explanation of how this could be achieved for a task is given. Like the communication interface wrapper, this technique is applied to qualifying tasks before the optimization flow to improve runtime placement outlined in section 4.1

Memoization is a technique that has been proposed for low power designs on FPGAs, though it has been previously applied to other fields, especially software.

Memoization involves reusing the result of a previous computation when a request is made for computation with the same set of inputs that produced them. Thus, the process of re-computing the result is circumvented – together with its attendant energy consumption. This advantage is, of course, at the expense of additional storage and logic resources. Therefore, the gains of memoization must be balanced against its overheads. Figure 4.8 shows an illustration of a circuit and its memoization block. In this architecture, the original circuit is only enabled to compute new results for a set of inputs if the memoization block fails to find the result(s) for the input(s) in memory. The results of the computation are saved to the memoization block's memory after each computation if not already present.

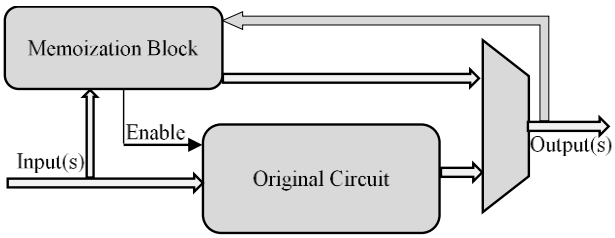


Figure 4.8: A circuit and its Memoization block

Memoization is only applicable to systems which are referentially transparent – systems that produce the same outputs for the same set of inputs. Systems whose outputs depend on some internal states or are determined by other factors than the current input(s) are not directly implemented by the memoization technique presented in this section.

The focus in this section is to minimize dynamic power using memoization technique. It involves disabling a circuit if the result of a requested computation is already present in memory. For the memoization technique to be gainful, the additional resources required for its implementation, together with its power consumption must be balanced against that of the original circuit(s) on which dynamic power is aimed to be saved.

The low-level details of implementing a low power memoization wrapper for a design while retaining the architecture of the original circuit is presented below. The

technique presented is directly applicable for applications with low interface width. The main contributions are:

i) An actual implementation of a memoization technique for applications with low interface port widths. A low-level implementation details of a memoization block architecture which can interface to FPGA-based circuit(s), including proprietary IPs is presented.

ii) A technique for achieving *negligible* energy overhead for a memoization block, by using memory space reservation to achieve a fast decision in a fixed number of clock cycles.

4.3.1 Architecture and Operation of Memoization Wrapper

a) Task Memoization Module Architecture

The memorization module adds a pre-processing step to the computations of an application. The application could consist of a single or multiple circuit (or tasks). Figure 4.9 shows the block diagram of a memoization module which is easily adaptable to applications with both single and multiple tasks. Its architecture consists of a task memory, an input data memory, output data memory and memoization logic. During an application's initialization stage, its component tasks are loaded into the task memory with each task assigned a unique ID. This unique ID also doubles as the address of the task in the Task Memory. The data stored for each task corresponds to the start of the address space allocated to the task in the memoization's block input and output memories. The depth of the task memory is determined by the maximum number of memoizable tasks in the application(s). Its width is determined by the number of tasks, sum of the number of inputs of the constituent tasks that are memoizable and the tolerance of the tasks. For example, for an application (or a set of applications) consisting of 4 tasks, with each task having a single 8-bit memoizable input and an input tolerance of 0, the task memory in Figure 4.9 is configured with a depth of 4 and a width of 10 bits. For the same number of tasks and inputs, but with tolerance of 2 bits, a task memory of depth 4 and width 8 bits would suffice. In the implementation, 5 additional bits are added to the width of

the task memory – 4 are used to specify the tolerance of the task and 1 (at the LSB) is checked for validity of the value stored at an address.

The memoization FSM checks the task memory to decide if a task has been saved or not. This check only takes 2 clock cycles as the task ID used for the check corresponds to the input address of the task memory. The output of this memory corresponds to the beginning of the section of the input memory hosting the inputs of the addressed task. Thus, it is the base address (base_addr) in the input memory. An offset value is added to this base address to form the input address of the input memory. The offset is determined using the task's current input and its tolerance value. Information of the tasks' tolerance are stored in the task memory as steps, where a step of 1 correspond to tolerance 0, step of 2 corresponds to tolerance of 1, etc. A straight forward approach is to reserve sufficient memory for all potential data inputs. In this way, the offset value is determined as follows. The data input is augmented (approximated) to the nearest reference value address by taking its tolerance into account. For a data input of 4 and a tolerance of step 2, the offset of (4/2 = 2) is obtained. This offset is added to the base address and the sum forms the input address to the memoization block's input memory.

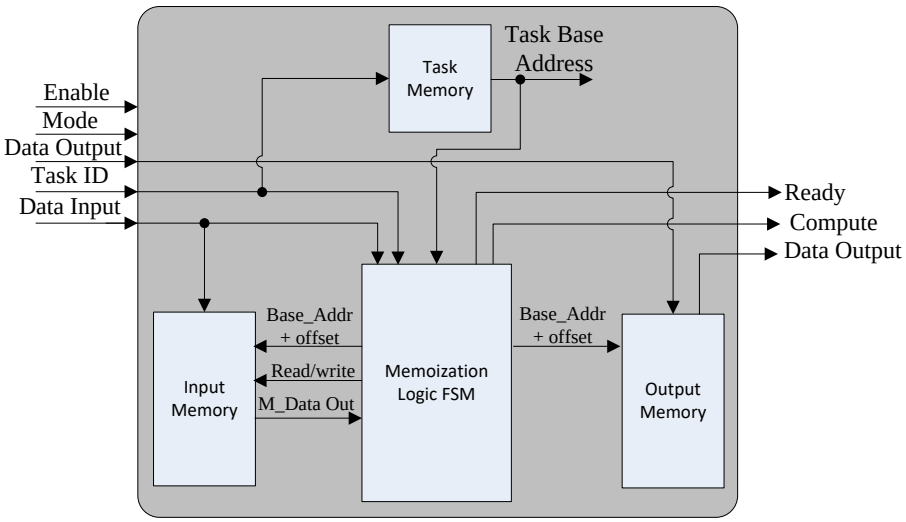


Figure 4.9: Block Diagram of Memoization Module

The output of the input memory is checked to determine if current data input has been saved or not. In the proposed memory space reservation technique, only the LSB of the output of the input memory is checked to determine if the value stored at the address is valid (‘1’) or not (‘0’). Hence, there is no greedy search procedure, where a series of values from memory are compared against the current input, involved. In addition, the addresses of the results in the memoization block’s output memory corresponds to the addresses of inputs in the input memory. For example, if an input for a task is saved at address 8 in the input memory, the results of that input would be saved at address 8 in the output memory. If a ‘valid’ is returned, the corresponding address is read out as the result of the computation. Otherwise, the task would have to be enabled to compute an output for the current input.

For a set of tasks to share a single memoization block, two conditions need to be satisfied. First, all the constituent tasks must have similar interface, with the same data width. Second, no two tasks will be required compute at the same time. Groups of tasks which fulfil these conditions could share a single memoization block to save logic. When a single task owns a memoization block, the task memory sub-block in Figure 4.9 is removed from its architecture.

b) Memory Size Management

The amount of memory reserved for a task in the input memory of the memoization block is determined by the number (and size) of its potential inputs as well as its tolerance. For example, a task with single input of 8 bits and a tolerance of 0 (step 1) would require 255 address spaces in input memory. As can be seen, the memory requirement increases very rapidly with the input size and number. Two suggestions are proposed for reducing this (huge) memory. First by saving only certain reference values of the inputs within the tolerance of task as mentioned earlier. For example, if the same task above has a tolerance of 1 (step 2) then the input memory requirement is reduced by 50% compared to the step 1 case. For this strategy, an additional resource saving can be obtained in the output memory as follows. Recall that the LSB of the output of the input memory is reserved to be checked if the current task

input has been saved or not. The other bits at the same address are free to be used for other purposes. They may be used to save address of the corresponding output in the Output Memory. In this way, there is no need to duplicate outputs that are the same. A single result value can be stored, and all inputs producing that output would have this address at their $(n-1)$th bit positions. However, some greedy search would need to be done to identify duplicate outputs, thus leading to higher timing and energy overhead.

A second possible technique of reducing the memory requirement of the memoization block is to reserve memory only for those inputs which are regularly assessed. A fixed amount of memory is reserved for the task's inputs to be memorized, and their corresponding outputs. In this case, all arriving inputs are initially saved, and when the predetermined memory is filled, a replacement policy is used to displace least frequently used ones. However, some search would be necessary to decide if the computation result for the current input has been saved or not. In addition, some logic overhead is incurred in implementing both the search and replacement schemes. None of these two techniques were implemented in the proposed wrapper as the target in this section is low port width applications.

c) Task Memoization Module Operation

The operations of the memoization module is in 2 modes: CHECK mode and SAVE mode. Figure 4.10 shows a generalized operation of the memoization block shared by a single task. Its operations are controlled by an FSM in its logic. In the CHECK mode, inputs to the application are evaluated by the memoization module to see if the result for the requested computation is already present in memory. It takes the input of the task (Data Input) as well as the task number (Task ID). These inputs are checked against the memorized data. The outcome of this check is either a HIT, when result(s) is present in memory for this input(s), or a MISS otherwise.

Misses are expected (frequently) at the early stages of the applications execution but expected to decrease over the lifespan of the application. After a long period of execution, the miss rate is proportional to the size of the memory and, in the case

where the memory is insufficient for all inputs, the efficacy of the policy used to select which inputs to save and which not to save. When a MISS occurs, the COMPUTE and READY signals of the memoization block are set high. If the input qualifies to be memorized, the FSM waits for the computation to complete and then switches to SAVE mode to save the results of the computation. The total energy for a MISS operation is higher for a task with memoization than just that of the original task since both the memoization block and the task are executed. It is important that the energy of the memoization block is significantly lower than that of the original task to minimize the overhead of a miss. This requires that the memoization check must be done in very few clock cycles, in order to minimize power consumption.

In the case of a HIT no computation is required by the original task. READY is set high, the OUTPUT corresponding to the current input is set on the OUTPUT port and COMPUTE signal is set low. The memoization module remains in the CHECK mode and the application is ready for a new input. This case circumvents computation energy. Memoization blocks are designed to achieve high HIT rate in the steady state of the tasks execution. In the SAVE mode, the module monitors the input to the task and its output. Both are saved respectively in the input and output memory of the module in the sections corresponding to their Task ID. To activate this mode, the CHECK mode must have been evaluated and resulted in the results of the computation not being found in its input memory.

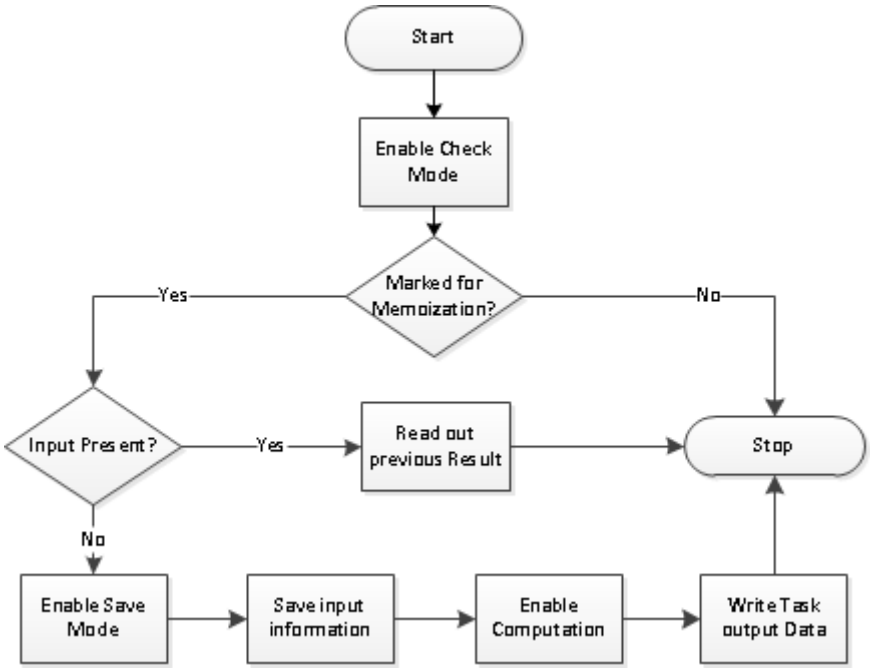


Figure 4.10: Flow chart of a memoization block

4.3.2 Energy Efficiency of Memoization Wrapper

The anticipated energy implication of the outcomes of the memoization block's operation is shown in Table 4.3. As shown, the value of the energy of the memoization block affects the average energy consumption significantly. Given that the factors that determine the HIT and MISS rates are often not completely known at design time, reducing the energy of the memoization block is one of the best ways of ensuring that the average energy consumption of an application with a memoization block remains considerably low, even in the case of frequent misses. Considering a hypothetical situation in which a memoization block has 50% miss rate. Figure 4.11 shows that the energy per transaction of the block needs to be less than 50% of the original task's energy to make any energy gain. In the figure both axes are expressed as percentages of the original task energy consumption. Considering that additional resources and time overhead are incurred by the memoization block, even more significant saving in the energy overhead of the memoization block is always required.

An important aim here is to lower the energy consumption of a memoization block by significantly reducing the number of its operating clock cycles. This is achieved by using input values as address offsets both for the input and output memories of the memoization block. This avoids the need for a time-consuming search step, thus reducing dynamic energy. In addition, the proposed technique offers the advantage that the processing time of the memoization block is independent of the size of the memory or number of inputs present in it. For a task with multiple inputs, the inputs are concatenated so that the check time remains constant. Thus, it offers predictability both in processing time and its energy consumption. However, this technique requires that space be reserved for all potential inputs. This seeming disadvantage in fact means that decisions about miss operations are reported after very few clock cycles and thus lower energy. Nevertheless, it must be acknowledged that to keep the miss ratio reasonable, the port width of the task must be small. However, there are many applications which can benefit from our scheme even with this limitation. Examples include a CORDIC task designed to compute the trigonometry of radian inputs, an RGB to YCrCb colour conversion task, and multiplier circuits just to mention 3.

Table 4.3: Possible Outcomes of Memoization Wrapper and Energy Implication

Status	**Energy Change (with respect to original task)**	
	Static	***Dynamic***
MISS	No change	Increased by energy of Memoization block
HIT	No change	Replaced by energy of Memoization block

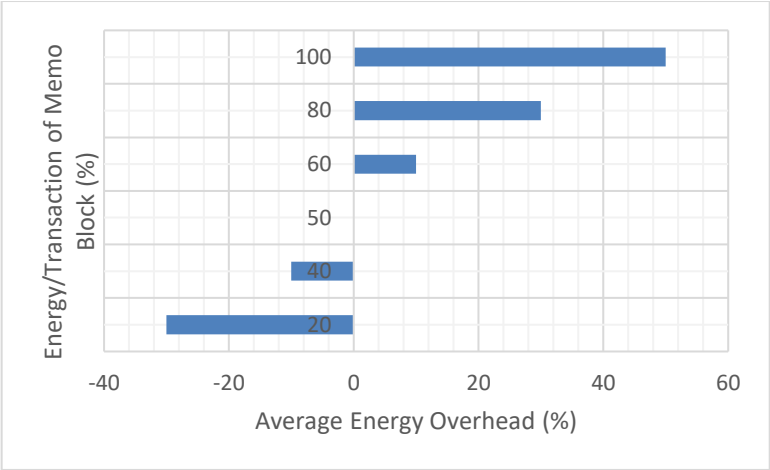


Figure 4.11: Variation of Memoization Wrapper Energy with Average Energy of Task and memoization Wrapper

4.3.4 An Implementation and a Case Study

The proposed memoization flow was implemented for a low port width application to test the efficacy of the technique. First, a simple CORDIC task which computes the hyperbolic tangent of an input angle was implemented. Two Xilinx IPs: a CORDIC Sinh/Cosh IP and a Division IP were used for the implementation. The first takes an 8-bit angle input ($Phase\_In$) and generates a 16-bit output, with the 8 MSBs of the output representing $Cosh(Phase\_In)$ and the remaining 8 bits representing $Sinh(Phase\_In)$. The division IP takes these two 8-bit vectors and perform a division to give an output of $Tanh(Phase\_In)$. Table 4.4 shows the resource utilization of the task and the number of clock cycles required for a single round of processing or a transaction. The implementation of the task has a dynamic power consumption of 112mW on the Xilinx's XC7A35T FPGA chip on which the corresponding static power is 72mW.

In addition, the memoization wrapper architecture and flow described above was implemented for this task. Being a single task, the task memory in Figure 4.9 was removed. The memory requirement for this block is 256 8-bit data locations for all potential inputs, and the same memory size for maximum distinct outputs. The resource utilization of the memoization module is shown in the second row of Table

4.4. It takes an average of only 5 clock cycles to complete its operation for a transaction. For a hit, it takes 5 clock cycles to produce the result from memory. A miss takes 3 clock cycles to report, and an additional 2 clock cycles to store the result of a computation of a task (in SAVE MODE). The power consumption of the memoization module is only 9mW. Given that each operation takes 5 clock cycles, the energy per transaction of the block is only 0.45nJ with a 100MHz clock.

Table 4.4: Implementation Data of a CORDIC Circuit and its Memoization Wrapper

Module	**Resources utilization**			**Clock Cycles**	**Power (mW)**
	LUTs	***Flip Flops***	***BRAMs (18Kb)***		
Cordic (Tanh)	1569	2243	-	25	112
Memo Wrapper	10	11	2	5	9
Wrapper overhead (%)	0.64	0.49	-	20	8.04

4.3.5 Results and Discussion

Table 4.5 shows the total (dynamic) energy per transaction for the design described in section 4.3.4 running at 100MHz. It shows that the energy overhead of the memoization block is only 0.96% in case of a miss, which is very small compared to the saving of 98% in the case of a hit. If the computed result is to be saved by the memoization wrapper in the case of a miss, the energy overhead increases to 1.6%, which is still significantly smaller than the savings obtained in a hit situation. In addition, the resource overhead is only 0.64% and 0.49% of the number of LUTs and FFs used by the original application, in addition to two 18kb BRAM. This is very small compared to the huge saving in energy. The timing overhead is only 5 clock cycles which is an increase of 20% compared to the original task. With these values, even if the miss rate is as high as 90% an energy savings of over 8% is still obtained.

Table 4.5: Energy Overhead/Transaction of CORDIC Task with Memoization Wrapper

Module	Energy per Trans. (nJ)	Energy Diff (%)
Cordic Task Only	28.00	-
Task and Memo Wrapper (Miss)	28.27	+ 0.96
Task and Memo Wrapper (Hit)	00.45	- 98.4
Memo Wrapper (Save Mode)	00.18	+ 0.64

In addition, Figure 4.12 shows the variation of average energy per transaction as the number of transactions progress for the CORDIC task with a memoization block. Each point on the graph corresponds to an average for 16 transactions. The input data used were randomly generated from Excel and had about 33% repetition. 33% is chosen in keeping with the common practice in data sets used for simulations in approximate computing [101] [110]. After 256 transactions, the average energy consumption was 18.35nJ which is 34.5% less than the task without memoization. It is worth noting that the energy consumption decreases significantly with subsequent input data, hence after a long period of execution, energy consumption would reduce significantly due to decrease in miss rate.

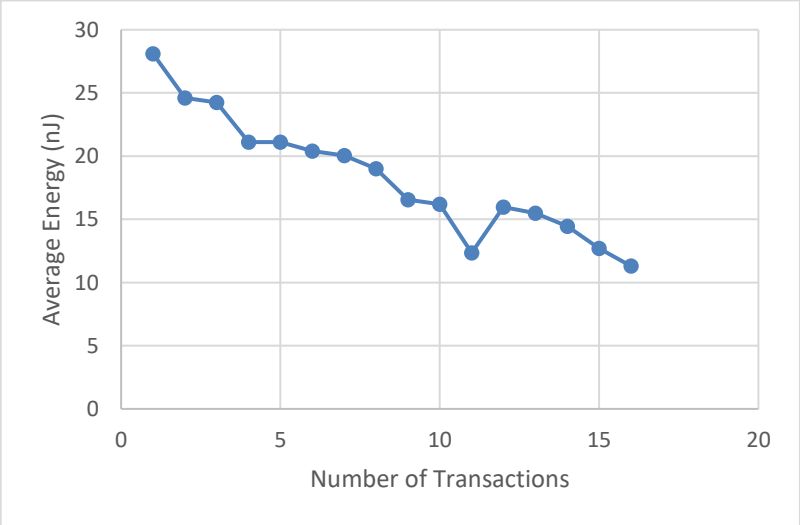


Figure 4.12: Variation of Average Energy/Transaction

In this section, an architecture for implementing very low power memoization block for applications with low port-widths has been presented. The technique uses memory space reservation to achieve a fast decision in a fixed number of clock cycles. It can be used with proprietary IPs; whose internal workings is unknown, provided its output at any point is a function of only the current inputs. Its architecture is directly applicable to many low port width applications, with only minor adjustment to memory sizes and address widths. The implementation results show that the technique leads to a significant reduction in average energy consumption at the expense of few resource overhead in many low port-width applications. For example, in a CORDIC circuit, the energy savings achieved in the case of a hit was over 96% and the energy overhead of a miss is only 1.6%. An average energy saving of 34.5% was obtained after 256 transactions. Its resource overhead was only 10 LUTs, 11 Flip Flops, a single 18Bb BRAMs. The wrapper introduced only a fixed 5 clock cycles overhead.

4.4 Chapter Conclusion

In this chapter, an overview of the various techniques applied to tasks in order to improve their runtime placement and efficient computation in a dynamic reconfiguration environment was presented. The chapter described an offline flow to improve the placement quality of tasks in runtime. The technique is based selecting implementation locations for tasks to minimize overlap in the potential placement locations of tasks occupying the FPGA area simultaneously. Placement quality optimization also aim to minimize the variance in the number of potential locations of each task to avoid a situation where some tasks have abundant areas and others have too little. This leads to an improvement in the number of placements in runtime for each task and increases the fault tolerance of an application.

The chapter also considers wrapping tasks according to format that supports communication in runtime without using pre-determined slots for tasks. In addition, a power optimization technique using memoization is applied to the task to reduce tasks' dynamic power consumption. The resource overhead of both the wrapper and

the memoization implementation are added to tasks' resource utilization before optimizing them for placement locations. The content of this chapter was published as part of the following papers:

- G. Enemali, A. Adetomi, and T. Arslan, "Expanding the Un-usable Area Strategy for Improved Utilization of Reconfigurable FPGAs", in 2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2017, 10.1109/AHS.2017.8046370.

In the next chapter, the runtime placement of task is considered and techniques to mitigate runtime time fragmentation and manage the relatively large reconfiguration overhead of COTS FPGAs are presented.

Chapter 5: Runtime Placement on FPGAs for High Performance and Reliability

Efficient runtime placement techniques are required in reconfigurable computing to achieve high performance and reliability. For state-of-the-art COTS FPGA platforms, two key challenges that need to be addressed by runtime placement management systems are fragmentation and the limitation of relatively large reconfiguration time overhead of COTS FPGAs. Fragmentation leads to wastage of chip area, which in turn leads to hardware tasks rejection in runtime. For application scenarios where hardware tasks are relocated on the chip to circumvent permanent damage, better chip area utilization positively impacts application reliability [19]. On the other hand, large reconfiguration time overhead, if not well managed, could lead to missed deadlines in real-time applications. Frequent reconfiguration can also increase energy consumption of applications. It also impacts the availability of the configuration port for error correction, and hence affects the reliability of applications. As mentioned chapter 1, although specialized FPGAs can be used for each application demand, this adds to the cost of designs and leads to longer application development time.

In this chapter, novel techniques which minimizes fragmentation and the effects of large reconfiguration time on COTS FPGAs in runtime are presented. First, a method of quantifying fragmentation which takes the heterogeneous nature of state-of-the-art COTS FPGAs into consideration is presented. Unlike previous and most current techniques of quantifying fragmentation, the proposed technique is based on *the degree of isolation* of the area occupied by a hardware task rather than its *degree of contact* with other tasks (or the chip boundaries). Second, an expansion strategy is proposed to avoid placement decisions that could create pockets of unusable resources on the chip, considering the heterogeneous nature of the chip and tasks dimensions. Finally, a task reuse strategy that aims to reduce the number of reconfigurations carried out in a runtime application scenario is proposed. The task reuse strategy includes a novel task replacement policy, FAReP. FAReP not only aim to circumvent reconfigurations and thus make the configuration port more available

to other important operations like error monitoring and correction, but also offer some defragmentation on the chip area as part of the replacement process.

5.1 Fragmentation on Heterogenous FPGAs

In runtime, the optimized tasks are scheduled to be placed on the chip, often in a dynamic manner decided by an application. Placements can also be requested when there is a need to relocate a task due to occurrence of faults on the chip. In addition (re)placement or relocation can be requested to balance system workload on the chip. In each of these cases, pre-synthesized tasks, in the form of configuration bitstreams are loaded onto the chip. As already discussed in section 4.1, the bitstream can normally be loaded on locations which matches the implementation location of the hardware task on the chip. However, since more than one of such locations exists on the chip, it is necessary to choose locations which will minimize the fragmentation of the chip area considering the current state of the chip area. Area fragmentation has been identified as the greatest obstacle to good chip utilization [111]. Finding a location for a task on the chip with minimal fragmentation in the shortest possible time is the goal of most runtime placement management systems. This is because minimizing fragmentation improves the utilization of the chip which in effect translates to better fault tolerance for critical applications, and lower task rejection ratio in other scenarios. It is also important that locations are decided quickly so that scheduled tasks do not miss their deadlines.

A major step in minimizing fragmentation is quantifying it. Therefore, a method of quantifying fragmentation on heterogeneous FPGA chips is presented with the objective of reduced computational complexity compared to state-of-the-art approaches while still leading to superior or comparable placement decisions. In addition to a fast and efficient fragmentation computation scheme, an additional technique called Expanding the unusable Area Scheme (EUAS) is also presented to further improve chip area utilization and to circumvent the creation of unusable areas due to the heterogeneous nature of COTS FPGA is also presented.

5.1.1 Quantifying Fragmentation

As pointed out in chapter 3, many state-of-the-art techniques for computing fragmentation such as the adjacency [63] or vertex based heuristics [71] do not suit a heterogeneous chip (which is the target in this thesis). This is because tasks' location on heterogeneous chips have definite start and end points which, often, do not fall at the border of existing placements [23]. The approach proposed in this sub-section, is based on computing the degree of isolation (as well as the adjacency – if it exists) of the possible location area (to be) occupied by task(s). The location with the minimum isolation is chosen. This is done using equation 5.1, which computes the average degree of isolation in the horizontal and vertical direction. This term is referred to as fragmentation coefficient (FC) for the remainder of this chapter.

In 5.1, $\bar{d}_{fh}$ and $\bar{d}_{bh}$ refer to the average distance – in number of CLB cells – in the forward and backward horizontal directions of a potential placement area. Similarly, $\bar{d}_{fv}$ and $\bar{d}_{bv}$ is the number of unit cells to the top and bottom of the area. The term r_h (r_v) is the range of the distances, and α is chosen to give appropriate relative significance between the distance and range terms. The parameters m and n are the horizontal and vertical dimensions of the chip. Number of CLB is used here as most FPGA chips are organized in grids and CLBs represents the smallest resolution of configurable physical units. The physical span of other reconfigurable units such as DSPs and BRAMs can be easily expressed in multiples of CLBs. In addition, only rectangular areas are considered.

$$FC = \left[\frac{\left((\bar{d}_{fh} + \bar{d}_{bh}\right)/m)^{\alpha}}{r_h} + \frac{((\bar{d}_{fv} + \bar{d}_{bv})/n)^{\alpha}}{r_v}\right] \qquad (5.1)$$

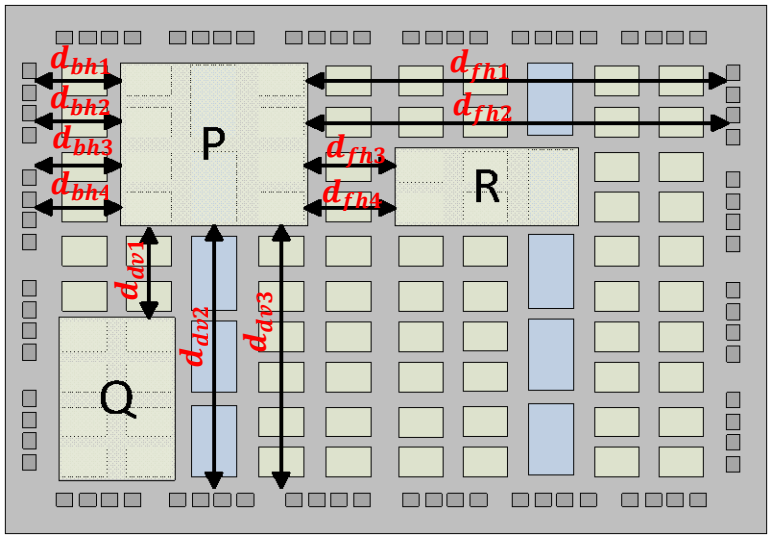


Figure 5.1: Quantifying Task Area Fragmentation

Figure *5.1* shows how the FC of an area is computed. As shown, there are 3 tasks' area on the chip marked P, Q and R. The distances around *P* are shown for 3 directions: forward horizontal (d_{fh}), background horizontal (d_{bh}) and downward vertical (d_{dv}). The distance in the upward vertical direction in this case is 0 as the area is located at the border of the device. The average distance in the forward direction, $\bar{d}_{fh}$ is computed using equation 5.2, where n is the height of the task area and d_{fhi} represents individual distance of the task segment from the closet task or the device border as shown in Figure 5.1. The average distances in the other 3 directions are computed similarly. The horizontal range term (r_h) is computed as the difference the forward and background horizontal distances of the area. The vertical range term is similarly computed.

$$\bar{d}_{fh} = \frac{1}{n}\sum_{i=0}^{n} d_{fhi} \qquad (5.2)$$

As an illustration, consider Figure 5.2 which shows different areas for tasks on a chip. For the chip shown, $m = 10$ and $n = 8$. The value of $\alpha = 3$ was used. This was chosen to give appropriate relative significance between the distance and range terms and thus increase the accuracy of the FC. Although higher value of α leads to

greater accuracy, it would nevertheless increase the computational complexity of calculating FC.

To illustrate how we selected the value of α for the chip shown. Consider α = 1. The values of the FC for the 4 areas on the chip would be: 0.425, 0.282, 0.725 and 0.425. Hence, the accuracy is quite low as only half of the areas has a unique FC. Increasing α to 2, all the areas have unique FC values (0.145, 0.150, 0.471 and 0.250) respectively. However, the first two values are quite close. α = 3 give unique values as shown in Table 5.1 below.

Table 5.1 shows the fragmentation coefficients for the scenarios shown in Figure 5.2 using our technique. As shown, Task 1 has the least FC (0.051) maintaining the least isolation and having the highest contact at its borders. Task 4 has the highest FC as shown. It is worth noting that although both tasks 1 and 4 have the same adjacency at their borders, their FC values are very different, hence, adjacency alone may not be a good metric for fragmentation on heterogeneous chips.

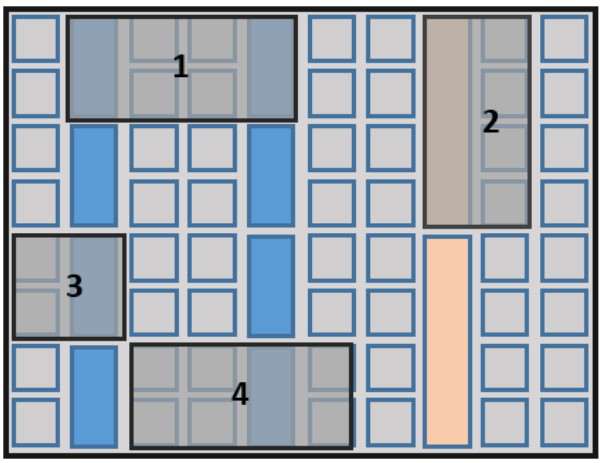


Figure 5.2: Task Areas on a Chip

Table 5.1: Fragmentation Computation

Tasks	**Fragmentation Coefficient**		
	Horizontal	***Vertical***	***Resultant***
1	0.027	0.024	0.051
2	0.048	0.031	0.079
3	0.064	0.244	0.308
4	0.108	0.024	0.132

There are many other techniques of measuring fragmentation on FPGA chips. As noted in chapter 3, many fragmentation metrics gives the fragmentation of the entire chip, and hence are closely aligned to applications requiring the state of the chip. An example of the application scenario where the fragmentation state of the entire chip is necessary is to monitor when to trigger defragmentation. However, in many scenarios fragmentation metrics are used as cost functions to decide location of task on the chip in runtime. This later scenario is the aim of the fragmentation coefficient presented here. In order to test the effectiveness of the proposed fragmentation quantification scheme, a comparison is done with some approaches where fragmentation metrics is used as a cost function in runtime placement of tasks.

The most widely reported metric for quantifying a form of fragmentation in deciding location for incoming task on a chip is the MER approach [64]. Although, the technique does not compute fragmentation directly, it uses the MER technique for the purpose of task placement in runtime. However, it is known to be a very expensive process even though it has a very high accuracy in deciding locations for tasks [72]. Many variants of MER have been introduced, which are approximations to improve the computational intensity of the original MER algorithm. An example of this is presented by Iturbe et al. in [63] where a version of MER called Empty Area Compaction (EAC) is used in deciding placement locations for tasks. On the other hand, in [78], Ejnioui et al. presented a fast fragmentation metric which is suitable for runtime applications is presented. The major disadvantage of the

fragmentation metric presented is that its accuracy is very low. The technique can be used to compute the fragmentation of different areas quickly without considering the entire chip area. The technique presented by Handa et al. in [79] though require more computation than [78], is more accurate. It also measures the fragmentation of an area to enable fast placement but does so by considering the contribution of each cell in the area.

Figure 5.3 shows a comparison of the level of accuracy of three fragmentation metrics together with the proposed technique using the tasks in Figure 5.2 as a case study. The accuracy is computed based on the capacity of the metrics to assign different values to each of the four tasks' area shown on the chip (called *Discrimination* in the figure). As shown, the EAC technique (Iturbe) has the highest accuracy. In selecting tasks for each of the areas on the chip, the metric values are all unique. The technique in [78] (Ejnioui) has the least accuracy of 25%, 3 out of the 4 task areas on the chip have the same metric value. The fragmentation metric used by the technique is $f = \frac{k}{N^2}$, where f is the number of (free) cells in an area, and N^2 is the number of cells in the chip. For the case study in Figure 5.2, Task 1, 2 and 4 all have fragmentation metric of 0.1, with only task 3 having a different metric of 0.05. Thus only 1 of the 4 areas have a distinct value. The fragmentation metric in [79] (Handa) is shown in equation 3.2. Using those equations, the values of the fragmentation metric for the tasks shown in Figure 5.2 were computed. Since the equations require information about the task size (given that the parameter L_x and L_y represent the average width and height respectively of the set of tasks to be placed on the chip), we assumed an upper band performance and subtracted those occasions where, in principle, the metric cannot differentiate between the tasks given its very form. For the case at hand, Task 1 and 4 would have the same number of empty cells in their vicinity (v_x and v_y). Thus, the upper bound performance of the metric in this case would be 75%.

As shown in Figure 5.3, the proposed technique also has distinct values for all the Task locations. It can be observed that the improvement of the proposed technique in terms of accuracy of differentiating the fragmentation metric of similar but different potential task location over [79], is based on the fact that the technique does not only

compute the empty cells around a cell, but also considers if the empty cells are located to the right or left (for horizontal direction) or in the upper or lower direction (for the vertical direction) of the slot. Additionally, the computation does not consider each cell in the slot but treats the slot as a unit – and thus have lower computational intensity. Table *5.2* shows the time complexity of the four techniques whose accuracy are compared in Figure *5.3*. As shown, although Iturbe [63] has a high accuracy, its time complexity is also high. On the other hand, while Ejnioui [78] has a constant time complexity, its accuracy is low. The complexity of the proposed technique is $O(n)$ where n is (half) the perimeter of the task area evaluated, which is comparable to that of Handa [79], however the proposed technique have a better accuracy.

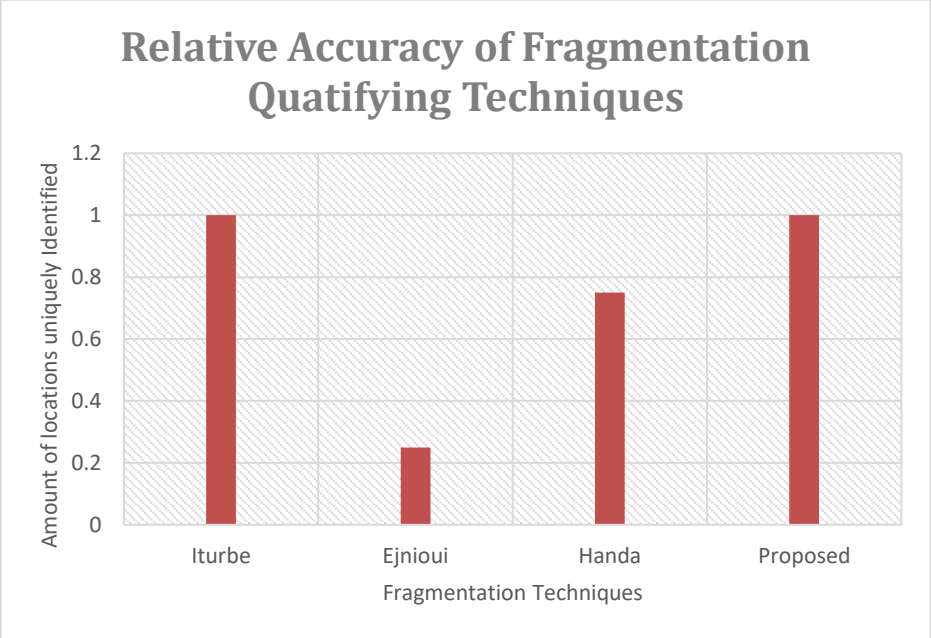


Figure 5.3: Comparison of Accuracy of Fragmentation (Cost) Quantifying Techniques for Task Areas in Figure 5.2

Table 5.2: Fragmentation Computation Complexity

Technique	Complexity
Iturbe [63]	$O(n^2)$
Ejuioui [78]	$O(1)$
Handa [79]	$O(n)$
Proposed	$O(n)$

5.1.2 Expanding the Unusable Area Strategy (EUAS) for Improved Utilization

During runtime placement, tasks are configured on the FPGA by choosing positions for them that would not hinder subsequent tasks as much as possible. For applications where all component tasks do not fit into the chip at once, when a task finishes its computation, and another demands the resources it holds, it is removed from the chip, and the new task configured. For all placement requests, whether initial or due to a task being removed from the chip to make room for another or due to a damage to (part of) the resource on which a task is located, a new location is found by the placement scheme with the aim of avoiding the creation of unusable resources. This is the aim of the fragmentation metrics as detailed in section 5.1.1. Unusable resources in this context mean those chip resources (CLBs, BRAMs or DSPs) which occur between the border of two placements and which cannot accommodate any of the constituent tasks of the application. Whenever possible (that is, when other placement locations exist for the task on the chip) placements which create unusable locations are avoided.

Fragmentation metrics such as adjacency is not a good way to quantify fragmentation when aiming to avoid the problem of creating pockets of unusable resource segments on a heterogeneous chip. This is because these metrics are often based on the *amount* of contact a task has with other tasks or the device. This is unreliable in heterogeneous devices because the tasks have start positions which are constrained by their layout, and often do not begin at the border of an existing placement [21].

On the other hand, fragmentation techniques which tries to evaluate the quality of a placement by computing the degree of fragmentation of the entire chip (e.g. EAC) is not only time consuming, but also not able to deal with the problem at hand efficiently. Computing the degree of fragmentation of the entire chip does not reveal complete information about the size or nature of individual contiguous free locations. Moreover, other fast metrics, such as the proposed technique could produce two areas with same metric under strict conditions. Hence, additional techniques need to be integrated to refine placement decisions in runtime, in addition to conventional fragmentation quantification. In this section, one of such technique is proposed called Expanding the Unusable Area Strategy (EUAS).

The basic idea of EUAS is illustrated as follows. Consider Figure 5.4 which shows a chip with section A of the chip containing a task which is involved in active computation. Suppose a new task X is scheduled to be placed on the same chip, and that in the set of tasks to be placed, the minimum width is known to be 2. Suppose also that locations C and D are matching locations for the task and have comparable fragmentation metrics. Although both C and D are matching positions for X on the chip, choosing C makes section B of the chip unusable for the duration for which tasks occupying A and C remain in their positions. A cumulative effect of many such unusable portions of the chip can lead to waste of the chip area. On the other hand, placing X at position D means that all columns between A and D are potentially usable. The aim of integrating EUAS with the fragmentation metric is to avoid placements which render certain portions of the chip unusable whenever possible.

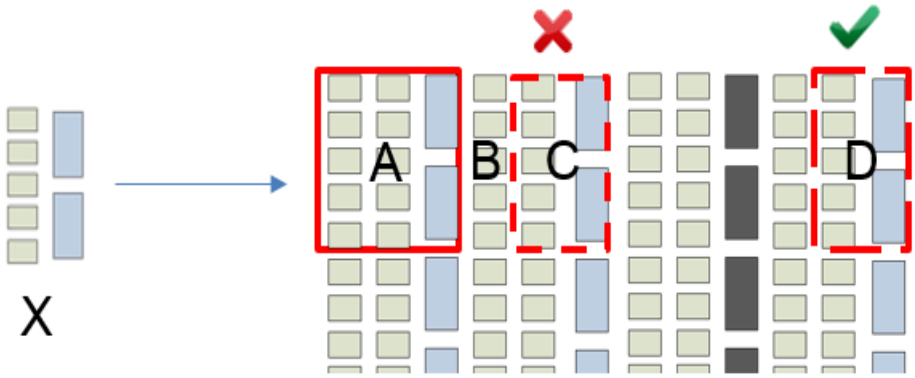


Figure 5.4: Effect of a Placement on the Usability of Adjoining Resource

It is worth noting that often, the size of all tasks to be placed is known after their design phase, hence the minimum dimension of tasks is known. Alternatively, an approximation could be obtained using the history of tasks requested to be placed by the placement scheme. This is used to determine if a location would be unusable or not.

Performance Evaluation

i) Experimental Set-up

To evaluate the effect of integrating EUAS with the fragmentation metric for placement of tasks with varying properties, we implemented a simulation framework on an Intel(R) Core™ i7 processor, running at 3.40GHz. The reconfigurable platform simulated was that of the Programmable logic of the Xilinx xc7z100ffg900-2 chip. In that chip, each row is made up of 134 columns, consisting of 12 BRAM columns, 15 DSP columns and 107 CLB columns. It has 7 identical rows.

The tasks used for the simulations were based on utilization and estimates of execution time of common hardware tasks, obtained from [86]. 20 sets each consisting of 100 tasks were generated. The results presented is an average of these. Each set varied in the range of values for l, w and s$RsId$. For example, set 1 has parameters in the following range $2 \leq l \leq 8; 1 \leq w \leq 3$ and $0 \leq RsId \leq 20$ while set 20 has their area parameters in the range: $58 \leq l \leq 64; 4 \leq w \leq 7$ and $40 \leq RsId \leq 64$. Random values were then generated for these parameters within assigned limits for each set. Configuration time, t_c was computed based on the equivalent resource utilization of each task using Table *2.4*. Execution time was randomly generated within the limits of $50 - 200\mu s$ and task deadlines were randomly assigned during placement requests. Task placement requests were generated randomly. As soon as request is received, and the placer and configuration manager were ready, the placement scheme is executed. If more than one request is received per time, an Earliest Deadline First (EDF) scheme [112] which first services the task with the nearest deadline was used. The placement scheme simulated included a task reuse strategy such that tasks are not deallocated from the chip except when their area is required by another task.

ii) Result and Discussion

Figure 5.5 shows the variation of the task rejection ratio for three device sizes. It compares the relative performance of the proposed EUAS scheme and that without it. The FPGA sizes A, B and C refer to approximately 25%, 50% and 100% of the xc7z100ffg900-2 chip. The corresponding logic equivalent are shown in *Table 5.3*.

The FPGA sizes were chosen to reflect the effect of small, medium and large FPGA sizes on the placement schemes. It can be seen, that for all three cases, EUAS has a less task rejection ratio. It is worth noting that the performance of EUAS is more pronounced for medium device size. For small sizes, due to limited placement positions on the chip, both schemes end up with similar placement decisions. Similarly, for large device sizes, a greater number of tasks can be accommodated. However, for medium device sizes, a difference of almost 10% in task rejection ratio exist between the schemes.

It is worth noting that although EUAS leads to an improvement in the number of successful placements, it does not guarantee that the '*freed'* resources will be usable by an incoming task. Basically, the technique aims to avoid the existence of unusable resources when an alternative location exists, but it was observed that some of these freed areas have resource layouts which do not match the layout of an arriving task, and hence remain unused. However, the scheme increases the probability of their usage. This explains why the performance of the scheme is offers only 9.4% improvement.

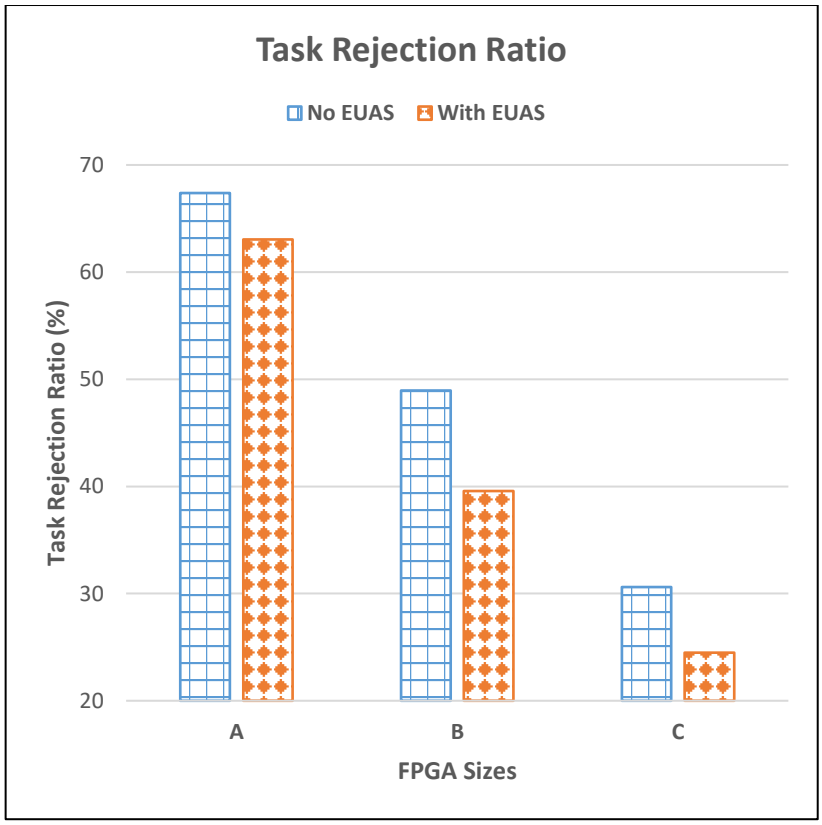


Figure 5.5: Effect of EUAS on Task Rejection Ratio

Table 5.3: Reconfigurable Resources of Simulation Platform

FPGA Size	CLBs	BRAMs (36kb)	DSPs
A	9400	200	600
B	18800	400	1120
C	34675	755	2020

5.2 Task Reuse to Circumvent Large Reconfiguration Overhead on COTS FPGAs

Online placement management systems on reconfiguration hardware such as COTS FPGAs must circumvent constraints associated with the target hardware platforms. For COTS FPGAs, two main interrelated constraints are *large reconfiguration*

duration and *ongoing fragmentation*. The former of these need to be well managed to avoid tasks missing deadlines in critical applications with real-time constraints and to free up the configuration port for other important activities. The later, if not well managed would lead to under-utilization of the chip resources due to fragmentation. The reconfiguration duration of state-of-the-art COTS FPGAs is quite significant, often in the order of milliseconds [38]. It is worth noting that COTS FPGAs have a single configuration port which is charged with multiple responsibilities including critical duties like being used for error mitigation [113] [56], freeing it up from some configuration activities will make it more available for these other important tasks. This in turn leads to more reliable designs for example.

Large configuration duration is not well amenable to the runtime scenario of many embedded systems requiring frequent context switches of its tasks [114] as it could lead to longer delays which can translate to missed deadlines. It also translates to large system down time where relocation is used to circumvent permanent damage on a chip. To cope with this, hardware task reuse have been proposed [38] [82] [86] [84]. Task reuse aims to preserve tasks with high configuration duration on the chip even after their execution, if they are likely to be required again soon.

However, the frequent addition and removal of circuits while preserving others on the chip will, very often, lead to fragmentation of its area, in an *ongoing* manner. Ongoing fragmentation happens even when individual tasks are carefully well-placed on their initial arrival as it is a result of the dynamic runtime activities of the chip [71]. Therefore, defragmentation of the chip area is required to achieve an efficient use of the chip resources in addition to good offline optimization and runtime placement techniques. Ongoing fragmentation would have been grossly reduced by defragmentation, which involves a time-to-time rearrangement of tasks on the chip [88]. However, large reconfiguration time makes such defragmentation very challenging on the current COTS FPGA architecture. In the following sub-sections, details of task reuse mechanism that aims to reduce the workload of the configuration port while also achieving a form of defragmentation is presented.

5.2.1 Task Reuse on COTs FPGAs

A highly promising technique aimed at addressing the problem of large reconfiguration time in run-time placement of hardware circuits on reconfigurable chips is *circuit reuse* [85]. It aims to circumvent (re)configuration overhead of certain tasks by retaining them on the chip after completing their execution, so that they do not have to be reconfigured for subsequent executions. In essence, once configured on the chip, hardware tasks are not deallocated until the resources they occupy on the chip is required by another task. Thus, on arrival of a new task (NT), the placement scheme checks if an instance already present on chip can be used to execute the task. It is only in the event that none is present that a new location is sought for the task on the chip, and the configuration engine used to write the configuration memory.

Under such condition, the possible outcomes in an attempt to place an arriving task are as follows: first, it could be assigned to an idle instance if any is found capable of executing the task. This is possible as circuits are not removed from the chip until their location is required by another task. The second option is to queue it, waiting on a computing task so as to reuse an already configured instance. This is possible if a suitable instance is configured on the chip but is actively involved in computation, and would become free in time for the execution of NT without violating its deadline requirements. This possibility is checked by verifying that the required wait time, t_w of the new task (equation 5.3) is less than the remaining computation time, t_r of the busy instance. These two options leverages task reuse, and configuration time is circumvented, freeing the configuration engine for other activities.

$$t_w = t_d - t_e \qquad (5.3)$$

The third possible option and last resort in the placement of NT is to scan for a new location for the task on the chip. To determine an initial placement, a scan function obtains the set of available locations. For each of these positions, a *fragmentation coefficient* (FC) is computed as described in section 5.1.1. The available location with the least FC is chosen for placement of the NT.

If all three of the above fail to allocate a position to the task, the task is not yet rejected since there are some idle instances on the chip which could be deallocated to accommodate the task, depending on the its criticality relative to the idle instances. The policy used to select a candidate for replacement is presented in section 5.2.2.

5.2.2 FAReP: Fragmentation-Aware Replacement Policy for Task Reuse on COTs FPGAs

Preserving hardware circuits on the chip to enhance task reuse have potentials to circumvent reconfiguration overhead for tasks. However, since all tasks cannot be preserved on the chip, some would need to be replaced at some point. As stated earlier, the choice of which task to replace is key to the performance of any reuse scheme [88]. State-of-the-art schemes uses the reconfiguration cost (often a product of reconfiguration time and likelihood of future reuse) as the criteria for replacement [85], [88]. However, this does not account for the fragmentation which the chip area undergoes due to frequent addition and removal of tasks. Our work differs from these because, in addition to reconfiguration cost minimization, we use each replacement window as an opportunity to also offer *defragmentation*. This leads to a replacement policy which in addition to preserving costly reconfiguration tasks on the chip, uses each replacement window as a defragmentation opportunity.

The proposed task replacement policy scheme considers, in addition to reconfiguration overhead and likelihood of future reuse, the degree to which a possible candidate contributes to fragmenting the chip is considered to determine which to replace. The basic idea is this: for a set of potential candidates of replacement with *comparable* reconfiguration cost, that which contributes most to the fragmentation of the device is considered for removal ahead of others. FAReP relies on three parameters to determine which of the idle instances to replace:

i) reconfiguration overhead, c,
ii) likelihood of reusing an instance in the near future, and
iii) fragmentation coefficient of the hardware task.

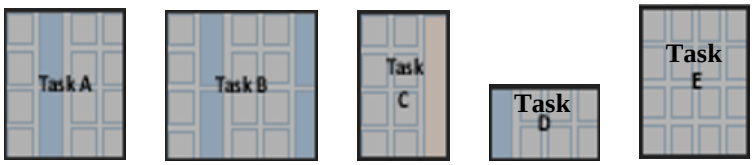
The reconfiguration overhead (t_c) is a function of the initial implementation area of the task and the characteristics of the runtime configuration controller. The value of t_c is stored in addition to the task's other parameters. The likelihood of re-use of an instance can be projected using its execution history. This is done as follows: a parameter, number of reuse (NU) is maintained for each configured instance. This parameter is initialized for each task after each (re)configuration. It is incremented each time the instance is used to execute a task. The cost of reconfiguration, ψ, is computed as: $\psi = t_c \; x$ NU.

When a task replacement is required, ψ is computed for all idle instances which could be potentially replaced to place an incoming task. Now, a threshold α is defined such that all instances whose differences in ψ is less than α have *comparable* reconfiguration cost. The instance of these with the highest FC is chosen to be replaced ahead of others, provided their replacement will enable the placement of the requested task. The effect of α values is discussed in the result section.

Figure 5.6 shows an example that points outs the benefit of the proposed replacement policy. A set of tasks ($A - E$) are requested to be executed on the chip twice, in the order A to E with the configuration overhead of B and C assumed to be comparable, but with B's slightly lower. The figure shows a comparison between $FAReP$ and another replacement policy called RER [85]. RER is based on reconfiguration duration and execution rate of configured instance.

Each stage in the figure represents a new task placement (and removal of another if necessary). As shown in the figure, during the first execution cycle, the three initial placements ($I_{1,2,3}$) are the same for both schemes as there is no need for any replacement. For stage I_4, to place D, FAReP (figure a) chooses to replace C because its FC (computed with equation 5.1) is 0.1 as against 0.08 for B. However, RER (figure b) chooses B since its reconfiguration overhead is smaller than that of C. The advantage of replacing instance C is that both D and E can be configured onto the chip without having to remove B in addition as is the case with the RER. This is because the removal of C made other fragmented spaces around it useable, thus constituting a form of defragmentation activity. In the second cycle of execution (after

t_1), only two configurations are required for FARePP, those of instances C and D. On the other head, RER requires 4 configurations (B, C, D and E) in the second cycle. Not only will the reconfiguration activities occupy the single configuration port longer, keeping it from other essential activities like error monitoring, but it also leads to delayed execution of the tasks themselves.



Tasks A- E to be placed on chip in 2 cycles

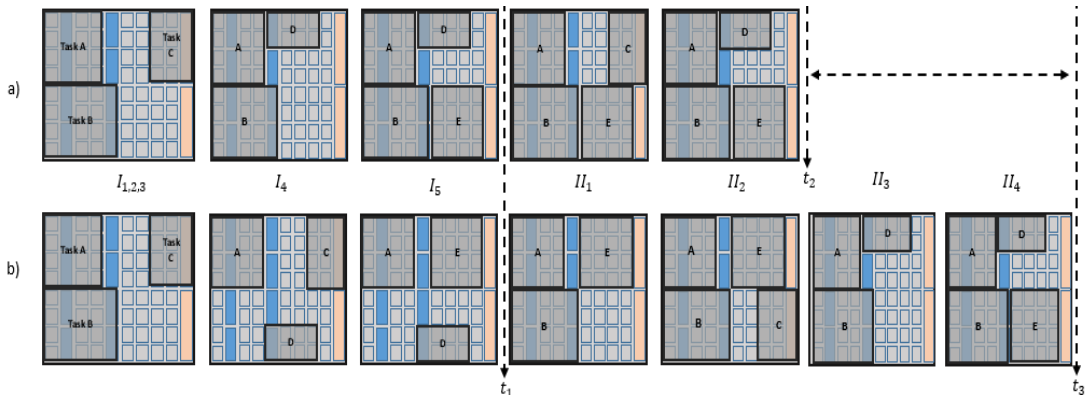


Figure 5.6: Comparison of Replacement Policies for Task Execution on a Chip
a) FARePP Policy b) RER policy

Performance Evaluation

To evaluate the performance of the proposed replacement policy, the same set of tasks and simulation platform in section 5.1.2 were used, with the exception that the algorithm implemented was the replacement policies. The performance of FARePP was compared with 2 other major hardware task re-use schemes. These are the RER in [85] and RBS in [86]. RER replacement policy is based on configuration cost and frequency of reuse. RBS uses a policy which is based on a form of frequency of reuse (called Least Probability of Recurrence, LPR). RBS preserves only certain tasks (called significant tasks) in a region on the chip while another region is used to

execute less significant tasks which are not preserved. The comparison is based on: task rejection ratio (TRR), average unused area at task rejection (AUATR), and Average Configuration Clock Cycles Saved (ACCCS). We define TRR as the ratio of the number of task rejections to the total of placement requests; AUATR= ratio of sum of unused area when task rejection due to area occurs to number of tasks rejected for lack of area. Finally, ACCCS refers to the sum of the configuration clock cycles of all tasks which reused idle instances, and hence did not have to be configured.

Figure 5.7 shows the variation of the TRR for RER, FAReP and RBS. FAReP is evaluated for various values of α. For the results shown, α is computed as 10%, 20%, 40%, 80% and 100% of the difference between the least and the largest configuration times of the tasks. As shown, FAReP with α = 0.1 has the least TRR of 6%, compared to the 10% and 11% respectively for RER and RBS. It can also be seen that the performance of FAReP degrades with increase in α. The successive degrading performance of FAReP with increase in α is due to the loss in the significance of configuration cost and frequency of reuse relative to fragmentation with increasing α. This conforms with [84] and [85] which found that configuration time and frequency of use of any instance are major factors in any replacement policy.

Table 5.4 shows additional performance data for the replacement policies. The values shown have been normalized to a base of the RER values to present a clearer comparison. As shown, $FAReP$ saves the configuration engine about 29% of configuration clock cycles compared to RER. This is a significant improvement as it could translate to amount of time the configuration port is freed up for other important tasks. On the other hand, the performance figure for RBS shows that the configuration port was occupied at about 16% more than the case of RER. In addition, the average wasted area when a task is rejected (AUATR) is lower for FAReP than RER by 14% showing a better utilization of the chip area. For RBS, its value is higher by 5%. This is due to the defragmentation offered by FAReP However, the computation time of $FAReP$ is slightly larger (9% more) than that of RER. This is due to extra computational steps required to compute fragmentations. However, it is faster than RBS by 7%.

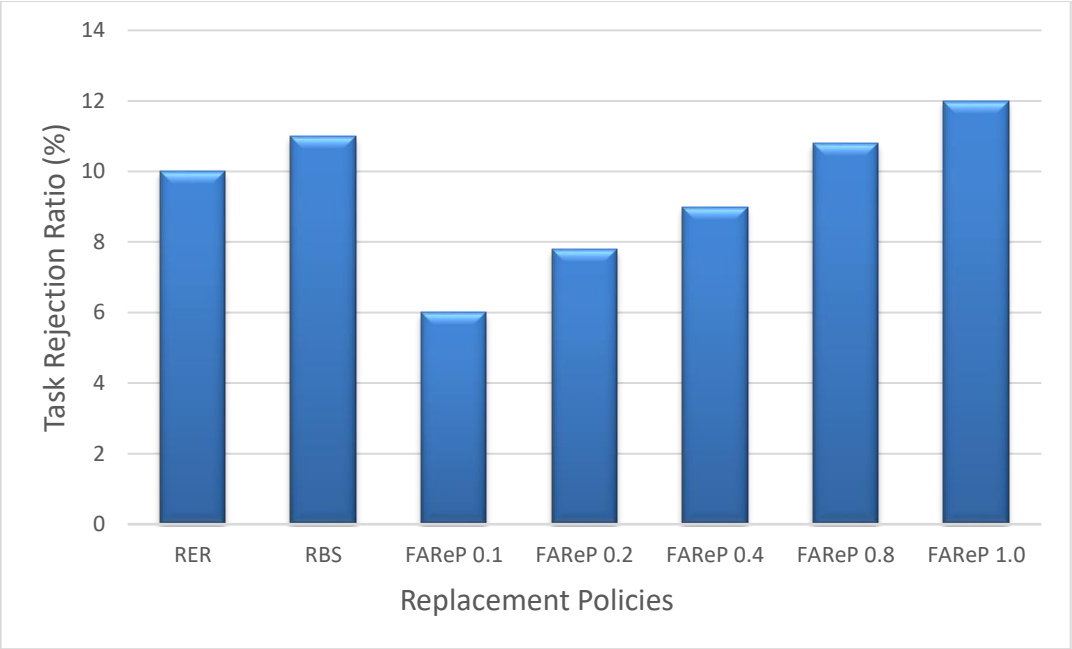


Figure 5.7: Variation of the Task Rejection Ratio for Replacement Policies

Table 5.4: Relative Performance Metrics of Replacement Policies*

	RER	**RBS**	**FAReP**
ACCCS	1	0.84	1.29
Runtime of Scheme	1	1.16	1.09
AUATR	1	1.05	0.86

*Normalised to a base of RER values

5.3 Chapter Conclusion

In this chapter, two major techniques relating to the runtime phase of a placement management on COTS FPGAs have been presented. First a method of quantifying fragmentation on heterogeneous FPGAs was proposed. The fragmentation quantification technique measures the isolation of potential task areas to choose a candidate for an arriving task in runtime. The accuracy of the proposed technique was compared to other techniques for quantifying fragmentation in runtime and it

was found to have high accuracy compared to state-of-the-art techniques. In runtime, this is augmented by a technique, EUAS – expanding unusable area strategy. EUAS involves using the information on the minimum size of tasks to be placed to avoid placement decision that creates pockets of unusable areas on the chip. This is necessitated by the heterogeneous nature of COTS FPGA where tasks placements do not fall at the border of existing placements due to layout requirements.

The second technique presented in the chapter is a task reuse strategy to circumvent the large configuration overhead of COTS FPGAs. The task reuse flow includes a task replacement policy, FAReP which in addition to preserving costly reconfiguration tasks on the chip, uses each replacement window as a defragmentation opportunity. FAReP choose candidate to be replaced from the chip using the reconfiguration overhead of instances, the likelihood of future reuse and the contribution of an instance to the fragmentation of the chip. The proposed replacement policy was tested by comparing its performance with state-of-the-art replacement policies. The comparison results showed that with FAReP, a reduction in the number of reconfigurations can be obtained and a greater number of tasks can be placed in runtime compared to other reuse schemes with different replacement policies. Reduced number of configurations translates to reduction in the occupancy of the single configuration port of COTS FPGAs. Thus, the configuration port can be more available to other important tasks like error monitoring. In addition, due to the defragmentation offered by FAReP a better utilization of the chip area is obtained. This leads to better placement quality which not only translate to greater higher performance for applications, but also a higher reliability in applications where relocation is used to circumvent permanent faults on the chip. The content of this chapter has been published in the following papers:

- G. Enemali, A. Adetomi, and T. Arslan, "FAReP: Fragmentation-Aware Replacement Policy for Task Reuse on Reconfigurable FPGAs", in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 202 – 206, 10.1109/IPDPSW.2017.153.

- G. Enemali, A. Adetomi, and T. Arslan, "A Placement Management Circuit for Efficient Realtime Hardware Reuse on FPGAs Targeting Reliable Autonomous Systems", in 2017 IEEE International Symposium on Circuit and Systems (ISCAS 2017), 2017, pp. 2030 – 2033, 10.1109/ISCAS.2017.8050796

In the next chapter, techniques relating to task relocation on heterogeneous FPGAs is discussed. The chapter will explore techniques of dealing with lack of matching locations on heterogeneous FPGAs by proposing a functionality-based relocation system to augment direct bitstream relocation on COTS FPGAs.

Chapter 6: Techniques for Task Relocation on FPGAs

Relocation of hardware tasks on FPGAs is a key technique used in reconfigurable computing to manage many desirable features. It is beneficial for many reasons. Three important ones are: to circumvent permanent damages on chips and consequently improve fault tolerance of critical applications in hostile environments such as space [19], to achieve defragmentation of the chip area [22] and hence provide a better utilization of the chip, and to maintain a desired thermal distribution on the chip [115]. Task relocation on COTS FPGA involve the movement of a circuit from one physical location on the chip to another. Although task relocation has many potentials, its implementation is constrained by provision of dynamic on-chip communication support for relocated tasks and finding suitable locations on the chip that matches the architecture of the tasks' original implementation location. The second of these constraints is becoming increasingly challenging to manage on COTS FPGA which are increasingly heterogeneous.

The aim of this chapter is to present techniques that improve the number of runtime task relocations on heterogeneous COTS FPGAs. The proposed relocator augments Direct Bitstream Relocation (DBR) with a Functionality-Based Relocation (FBR). As shown in the chapter, DBR alone is quite limited on COTS FPGAs due to their heterogeneity, while the proposed FBR technique is limited in relocating tasks which are not referentially transparent or have large port width. Hence, the chapter proposes to merge both techniques for improved performance.

This chapter is organized as follows: first, an overview of DBR is presented. This includes two approaches: generating unique bitstreams for different locations and manipulation of selected location-dependent information in the bitstream to change its location during configuration. Second, detail of the proposed FBR scheme is presented. The FBR technique presented in this chapter relies on the mechanism of replicating the functionality of a circuit with a look-up-table (LUT) or a memory

block in runtime for selected circuits. The chapter concludes with a comparison between the traditional DBR and the proposed FBR to show the improvement in number of relocations obtained by a merger of the two using a case study.

6.1 Direct Bitstream Relocation

Bitstream relocation techniques allow the use of a single partial bitstream at different locations on the FPGA. However, each partial bitstream has location specific information embedded within it. This information enables the configuration controller to determine which location to configure the bitstream on. For a bitstream to be directly relocatable to another location, the location information in the bitstream must match that of the intended destination. Since runtime placement management techniques determine location of hardware tasks in runtime, it is important to consider how the location dependent information in a partial bitstream is managed to enable relocation.

6.1.1 Methods of Direct Bitstream Relocation

One possible method of DBR is to synthesize a partial bitstream at all potential locations that it might be placed on in runtime. In that case, the process of relocation essentially reduces to searching which version of the partial bitstream matches an intended location. This approach is not only time consuming but has large memory overhead for storing the various versions of partial bitstreams [116].

A more efficient technique of bitstream relocation is to modify the location dependent portions of a bitstream in runtime [6]. For most modern COTS FPGAs, this modification does not constitute a huge overhead as only a single reference information needs to be modified in a partial bitstream for each resource type. For example, all CLBs configuration bits share a common location reference information which points to the beginning of the collection of CLBs to be configured by the bitstream. In addition, in most recent Xilinx FPGAs, CLB and DSP resources share a common block type and hence have the same reference location information.

BRAMs have a separate location reference. To this end, there are a maximum of only two location dependent information (typically called Frame Address) that need to be edited in runtime to relocate a bitstream, provided the destination location is physically identical to the source location of the bitstream without considering communication and clocking nets. Just after each frame address reference, are several frames required to configure the area of the FPGA whose functionality is represented by configuration data in that section. This is illustrated in Figure 6.1. Each location-dependent field to be replaced is a 32-bit word and can be efficiently replaced by the location found by the placement manager in runtime using the configuration controller in [27] adopted in this thesis. The configuration controller looks for the unique FAR command and once detected, the next word is replaced by that of the desired destination. It is worth noting that each frame has a unique frame address, but this is not stored in the bitstream as the reference frame address is increased by 1 automatically after each frame is loaded.

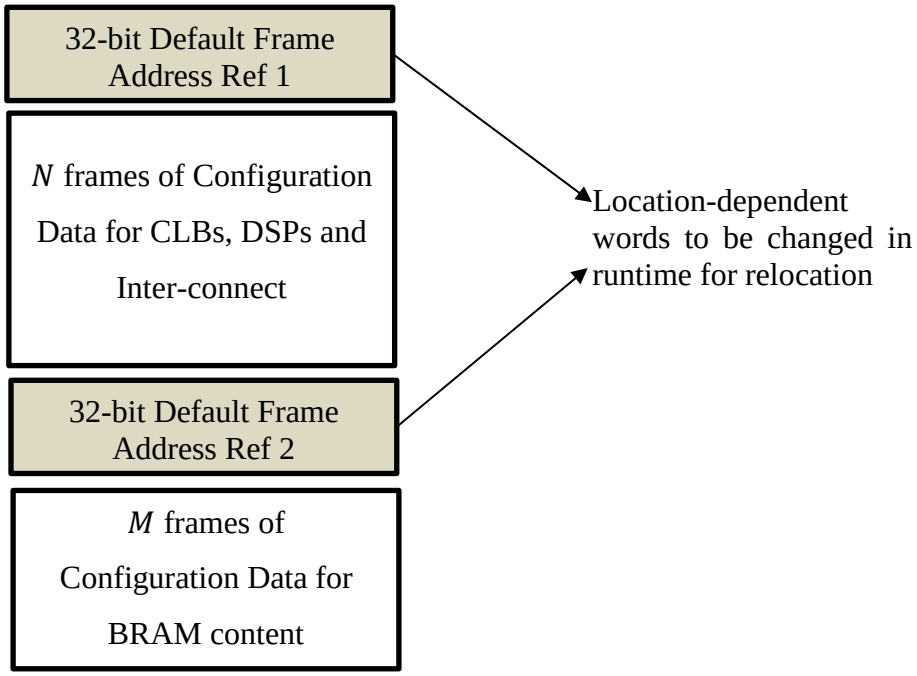
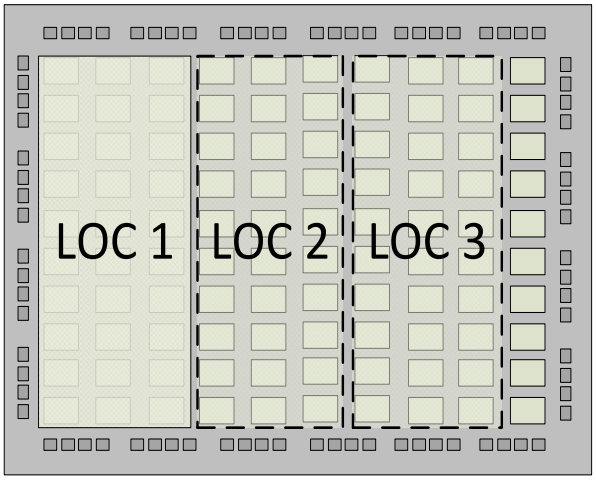


Figure 6.1: Achieving Direct Bitstream Relocation Using Runtime Frame Address Modification

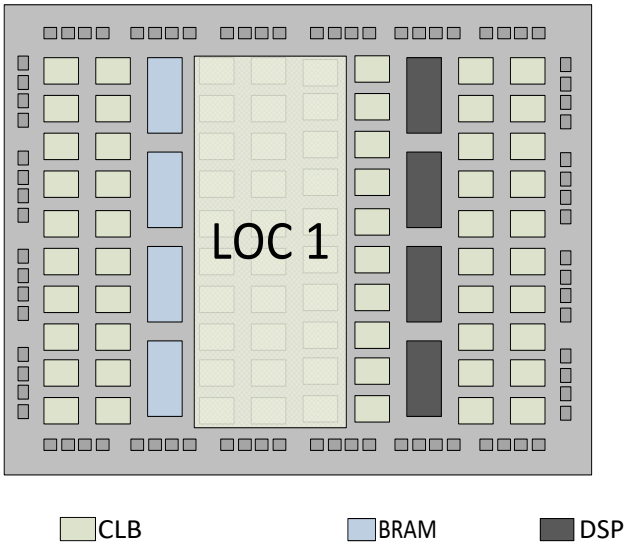
6.1.2 Limitations of Direct Bitstream Relocation

A major condition that needs to be satisfied for relocation of a circuit to be possible in runtime using DBR is that the resource composition of the original location for which the task was synthesized should be the same as the intended destination location. That is, the source and destination are required to have identical chip area, not only in size, but also in the type, number, order and orientation of the resources they contain. This condition was easily satisfiable in older versions of FPGAs which were essentially homogeneous. Modern FPGA chips, in a bid to improve performance and lower power consumption, have hard blocks such as memory blocks (BRAMs) and digital signal processors (DSPs) sandwiched between the conventional CLBs [117]. In addition, these BRAMs, DSPs and CLBs sometimes have different orientations (left and right) which differ in routing types as in the Xilinx 7 series FPGAs. Thus, FPGAs have become increasingly heterogeneous, and this places greater restrictions on the relocation of circuits. The result of this increase in heterogeneity is that the number of direct bitstream relocations possible for typical tasks has reduced with newer generations of FPGAs. Figure 6.2 illustrates this point. The figure shows that while on a homogeneous chip the circuit on LOC 1 could be relocated to 2 additional identical locations (LOC 2 and LOC 3), on the heterogeneous chip no identical location can be found.

Therefore, in the next session, a novel functionality-based relocation is proposed to improve the number of relocations possible for a circuit in runtime.



(a)



(b)

Figure 6.2: Number of relocations on homogeneous and heterogeneous FPGAs
a) Up to 2 relocations are possible b) No relocation is possible

6.2 Functionality-Based Relocation

The basic idea of FBR is to memorize the outputs of tasks during their normal execution, so that its functionality can be mimicked using a look up table (LUT) or a memory block in runtime. The basic idea of the concept is illustrated in Figure 6.3. It shows an original logic-based circuit on the left transformed into a memory-based circuit on the right. The aim of the transformation is so that the new circuit can be

relocatable to additional locations which do not match the resource layout of the implementation location of the original circuit.

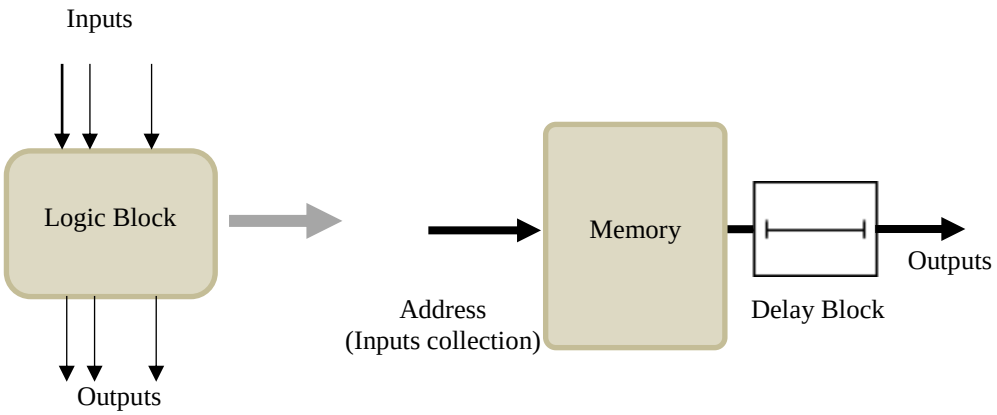


Figure 6.3: Transformation of Logic Block to Memory Block

6.2.1 FBR: Operation and Architecture

The proposed functionality-based relocation is done when an exact matching position for the circuit's original bitstream is either not available, or would lead to undesirable effects, such as increased fragmentation of the chip area. A circuit to be relocated using this technique has its computation results memorized during its normal operation using a dedicated system which is referred to as relocation module hereafter. In addition, a generic bitstream of an LUT or memory resource template is pre-synthesized and stored in an off-chip memory at design time. When relocation is required in runtime, a destination location is configured with the bitstream template, and its memory content filled with the outputs of the original circuit previously memorized.

The operational flow of the relocation mechanism is shown in Figure 6.4 and can be summarized as follows. When a request is received to relocate a circuit (after attempts to find an exact location for the original bitstream on the chip is found to be infeasible or unprofitable), a duration evaluator carries out a check to see if the

timing constraints associated with the relocation request can be met. Next, an area check is done to find a suitable location for a pre-synthesized memory template. The details of the time required for a relocation procedure is given in section 6.2.1b below while Section 6.2.1c explains the procedure for the area check. If both checks are successful, then the relocation request is accepted and executed in 3 additional steps:

i) The outputs of the circuit not present in memory are computed and saved

ii) A memory template is configured on the chip

iii) Data is copied from the original circuit's memory unto the already configured template.

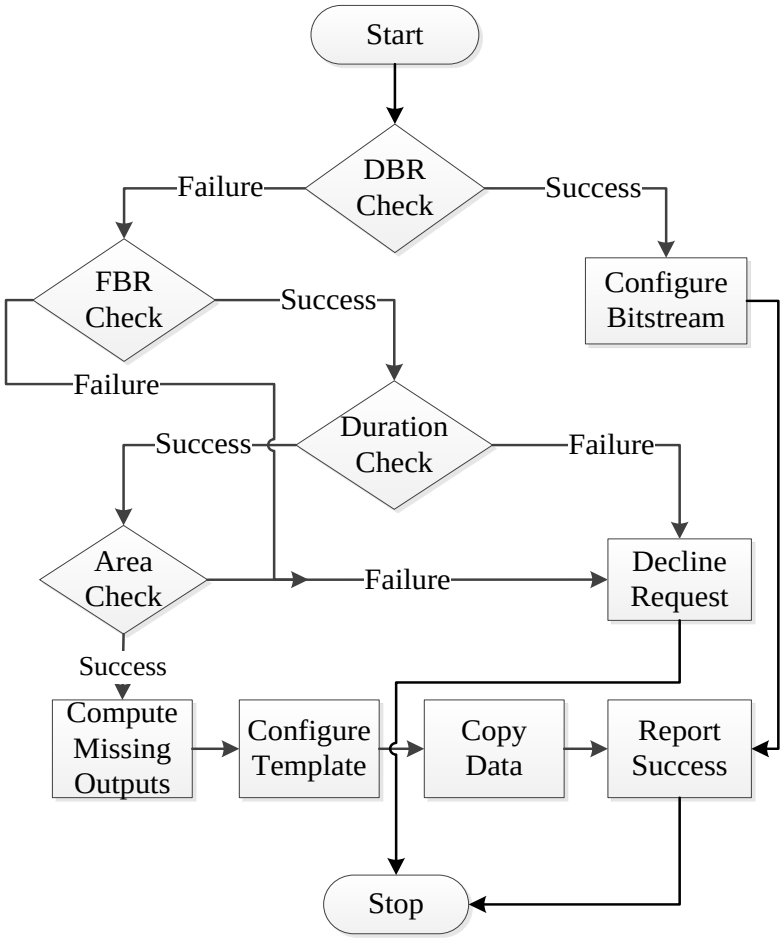


Figure 6.4: Operational Flow of the Proposed Functionlaity-Based Relocation Technique

These operations are managed by various units of a relocation module discussed below. The architectural composition of the proposed relocation module consists of an Output Memorizer, Duration Evaluator, and Area Finder.

a) Output Memorizer

The architecture of the output memorizer is similar to memorization module in section 4.3.1. It basically saves the results of computations of selected circuits in memory in runtime. Thus, it connects to the circuits whose outputs it memorizes. It has 3 units: task memory, evaluation logic (which is called memo logic for the remainder of this chapter) and output memory. These are shown in Figure 6.5. The task memory saves the list of circuits which are currently configured on the FPGA chip and are potentially relocatable by functionality. The memo logic manages the conversion of the raw inputs to address values, determines if the output for an input has been previously saved and switches mode to save the current output of the application when it has not been saved previously. Each circuit has a unique identifier (Circuit ID) which corresponds to its address in the task memory (Base_Addr). The memo logic has a fixed 3 clock cycle overhead when operating in the CHECK mode where it verifies if an input has been previously saved, and an overhead of 2 clock cycles when in the SAVE mode where it saves an output unto its output memory if not already saved.

Basically, the fixed number of clock cycles is achieved by concatenating the inputs of a circuit into a unique address value (Base_Addr + offset), with Base_Addr being the start of the memory location assigned to the circuit and offset determined using information on the circuit's input and tolerance. Hence, our proposed technique is based on memory space reservation (since each input translates to a unique address) rather than a greedy search procedure, where a series of values from memory is compared against the current input. In the CHECK mode, the memo logic operates in parallel with the operation of the original circuit, and thus does not add any pre-processing overhead to circuits which take at least 3 clock cycles for their normal

operations. In the SAVE mode (executed only when an input has not been previously saved in memory), 2 post-processing clock cycles are needed.

The output memory contains the results of computations. An application with multiple outputs has these outputs concatenated and saved at an address. The Least Significant Bit (LSB) of each output memory location is reserved to be checked for validity of the value stored at that address as shown in Figure 6.6. This bit is checked to determine if results of a computation are available in memory or not. A value of '1' at that location indicates that a previous value has been saved and is valid and a '0' means that valid output is missing for this input and the original circuit would have to compute it.

To compute missing outputs in runtime after a request to relocate a circuit is received, the memo logic iterates through the LSBs of the section of its own output memory dedicated to memorizing the circuit's outputs. The LSB of a missing output has a value of '0'. The address indices (which correspond to inputs) of missing outputs are then each decoupled and fed into the original circuit as inputs for it to compute corresponding outputs. It is worth restating that the LSBs of the output memory are used to keep track of valid outputs. This is because in reconfigurable computing, the functionality of a circuit could be changed in runtime, for example, when a part of that circuit is reconfigured with a different functionality in runtime using DPR. Under such conditions, the memo logic refreshes previously computed outputs by resetting the LSBs of the output locations to '0'.

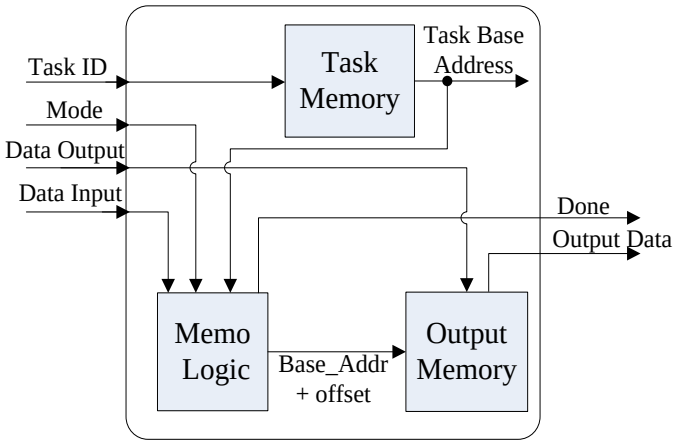


Figure 6.5: Architectural overview of the output memorizer

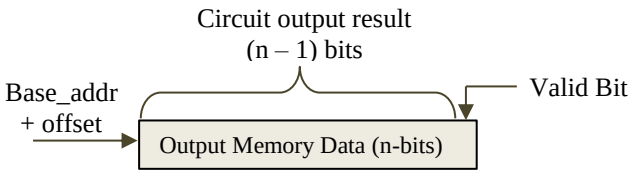


Figure 6.6: Data Distribution in Output Memory of Output Memorizer

The sizes of the task and output memories of the Output Memorizer are determined by the number of relocatable circuits on the chip, the sum of the number of inputs of the constituent circuits and the *tolerance* of the circuits. By tolerance, it is meant permissible variation in a circuit's outputs. Since this technique requires that space is reserved for all potential outputs, its memory overhead could be a major bottleneck for large-port-width applications that require numerous distinct outputs to be saved. Hence, we acknowledge that to keep the memory requirement reasonable, the port width of the circuits which can be relocated using this mechanism must be small, or if the port width is large, then the application tolerance must be large as well. Moreover, the functionality-based relocation proposed in this work is only applicable to circuits which are referentially transparent – that is, circuits implementing systems that *produce the same set of outputs for the same set of inputs*. Circuits whose current outputs depend on some internal states, or are determined by factors other than the current input(s) are not directly relocatable by the technique proposed in this work. Nevertheless, there are many applications which can profit from the proposed scheme even with these limitations. Three Examples are: an RGB to YCrCb colour conversion circuit which is widely used in computer graphics, CORDIC circuits designed to compute the trigonometry of angular inputs, and multiplier circuits which form the basis for many other applications.

b) Duration Evaluator

This unit checks if the requested relocation can be completed within the time constraint associated with the request. Its architecture consists of an LUT RAM which contains the essential parameters of the circuits, including the duration associated with the circuit's operations such as the configuration time, the number of clock cycles for computation of outputs (e) as well as the duration of data transfer from the output memorizer's memory to a memory template. The time constraint of a relocation request is evaluated using equation 6.1. The term R_t in equation 6.1 is the total time required for relocation, C_t is the time required for the memory template to be configured on the chip; and e is the time required to compute a missing output of the circuit(s) to be relocated, with n being the number of the missing (yet to be saved) outputs. M_t is time required for the memorized memory content to be transferred to the template. It is worth noting that the operation of the area finder and the computation of the missing outputs of a circuit to be relocated are done in parallel, thus equation (6.1) uses the value of the greater of time required to complete these two operations. e is initially measured at design time just like the configuration duration of the circuit. However, since e depends on the architecture and functionality of a circuit, when these are changed by DPR, its new value is measured (by observing the duration required by the updated circuit to change a set of inputs into outputs) and updated in runtime.

$$R_t = \sum_{i=0}^{n} e_i + C_t + M_t \qquad (6.1)$$

c) Area Finder

The area finder basically checks if there is an area on the chip for a template to be placed on. It has access to a RAM containing the state of the chip (State Memory), as well as a memory containing all the potential locations of the template. The State Memory represents the state of all resources on the chip by an M x N Matrix, where M and N are respectively the number of rows and columns in the device. An available resource is represented by a '0' and a used or damaged resource by a '1'.

Thus, each element in the matrix defines the state of a specific reconfigurable resource on the chip. A scan function is used to check the availability of potential locations for the circuit in the light of the current state of the chip. Further details of the scan procedure can be seen in chapter 7.

Finally, it is worth stating that the memory template consists of a generic memory block capable of holding all potentially required output data of the circuit(s) it is designed to replace. It also contains associated logic to manage functionalities such as memory read and delay management. Its memory size is determined like the output memory of the output memorizer discussed in section 6.2.1(a) above. The delay management block manages the difference between the timing behavior of the memory template and the original circuit so as to maintain the timing characteristics of the entire system. It does this by delaying the assertion of 'done' by the difference in the number of clock cycles between the operation of the memory template and that of the original circuit.

6.2.2 FBR Implementation Details

a) Case-Study Application: CORDIC

To test the proposed FBR flow a CORDIC application was implemented using Xilinx IP blocks. The application consists of 3 independent circuits: Square root, Sine/Cosine trigonometric operations and the hyperbolic tangent (Tanh) computing circuits. CORDIC was chosen as it is an important algorithm for various mathematical functions [118]. Details of the circuits' operations as well as their data format can be found in [104]. A custom wrapper was created for the circuits for easy compatibility with the proposed FBR model. Each circuit was optimized to take an 8-bit $DataIn$ and produce an 8-bit $DataOut$ and ap_done signal. The application and its components, along with a top wrapper module were synthesized using Xilinx Vivado suite for the Xilinx xc7a35tcpg236-1 FPGA chip. The top wrapper includes a $TaskId$ signal that is used to select a particular circuit. Table 6.1 shows the resource

utilization of the circuits, while Table 6.2 shows the number of clock cycles for each operation. The partial bitstream of the application is 140 kB in size.

Table 6.1: Resource Utilization of a CORDIC Circuit Case-Study Application

Circuits	**LUTs**	**Memory LUTs**	**Flip Flops**	**BRAM**
Square Root	71	1	100	-
Sine/ Cosine	277	4	307	-
Hyperbolic Tangent	1583	4	2218	-
Wrapper + All modules	2226	13	2920	-
Memo Block Template	14	14	21	1

Table 6.2: Latency of CORDIC Circuit Case-Study Application

Circuits	**Clock Cycles (e)**
Square Root	15
Sine/Cosine	19
Hyperbolic Tangent	56
Memo Block Template	2

b) Relocation Module

The relocation module, comprising of an output memorizer, duration checker and area finder described in section 6.2.1 was implemented using the Xilinx Vivado 15.1 design tools. Its resource utilization is shown in Table 6.3. A total of 66 LUTs, 58

flip flops and a single 18-Kb BRAM were used on the xc7a35tcpg236-1 chip. It is worth noting that the size of the memory used is dependent on the application. We chose an 18-kb memory because it is sufficient to save all the outputs of our target case-study application. The relocation module connects to the inputs and outputs of application(s) to memorize new computations by the application. It is also worth noting that practical relocation techniques require access to the configuration memory of the FPGA, as well as a means of communicating between a relocated module and other parts of the chip. Thus, a self-reconfiguration controller [27] with the required access to the configuration memory was instantiated. The controller is used for configuring the chip, as well as copying of data between block memories of the relocation module and the relocated module via the configuration layer. To address the need of a communication technique that supports relocation, the technique described in [26] which makes use of those clock buffers not used by applications for on-chip communication was adopted. The CORDIC case-study application used a single BUFG out of the 32 available on the xc7a35tcpg236-1 for clock network delivery. Thus, the remaining 12 BUFH and 2 BUFMR per clock region present on the chip are available for on-chip communication without any conflict with our case-study application or relocation management module. The technique is used to maintain communication between the relocated circuits and other circuits on the chip and/or the FPGA ports.

Table 6.3: Resource Utilization of Proposed Relocation Module

Unit	**LUT**	**FF**	**BRAM**
Output Memorizer	10	11	1
Duration Checker	36	30	-
Area Finder	20	17	-
Total	66	58	1

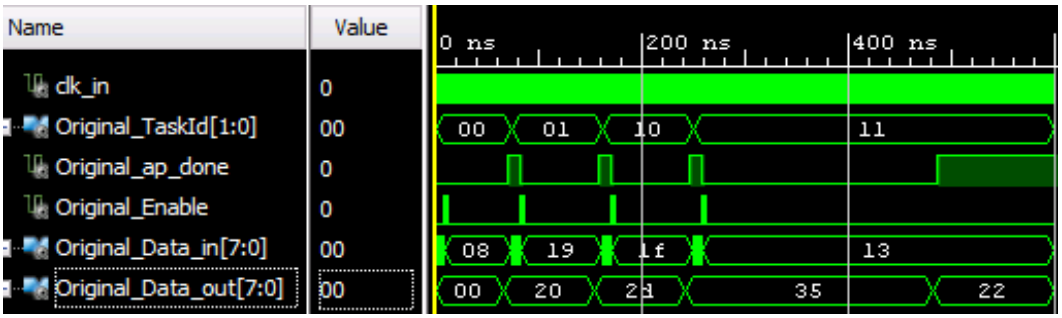
Next, a memory template for relocation was implemented. This template reserves 10 kB of memory and manages the delay of the application it replaces. This memory size was determined by the maximum memory requirement of the circuits whose functionality it is intended to replace. The actual resource utilization of the memory template on the target FPGA is 14 LUTs, 21 Flip flops, and 18-kb RAM and it has a delay of 2 clock cycles. The bitstream size of the template is 76.9 kB. The delay mechanism is used to ensure that the relocated equivalent does not alter the timing of the relocated application so as not to lose synchronization with the entire system.

6.2.3 Performance Evaluation and Comparison with DBR

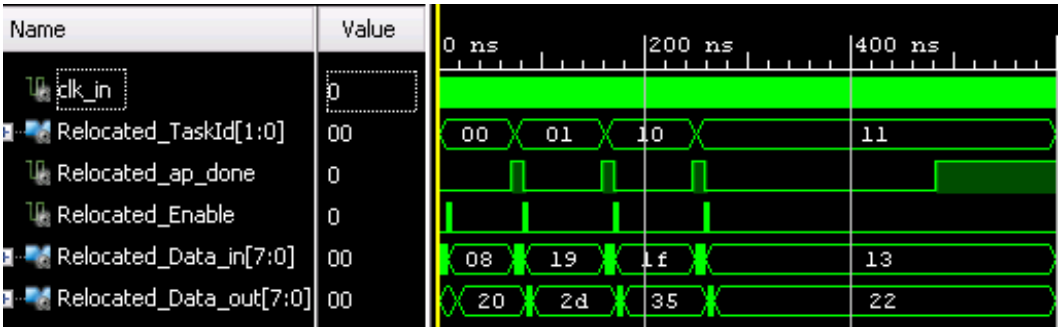
At runtime, a relocation request was initiated when 50% of the outputs of the application have been saved by the output memorization module. The floor plan of the application required a pattern of 8 contiguous CLB columns on the xc7a35tcpg236-1 chip. This pattern occurs only once on that chip. Hence, only a functionality-based relocation was possible. The timing constraint associated with the relocation request was such that the relocation was required to take a maximum of 1ms. The total time duration for the relocation was measured as 306.80 µs at 100 MHz, with the configuration of memory template taking 82.30 µs, the computation of missing outputs taking 175.36 µs, and the copying of data from memorization unit memory to the template taking 49.14 µs. The worst-case relocation duration for this module was also measured as 361.86 µs and best case as 131.44 µs. This was done by generating relocation requests when 0% (worst case) and 100% (best case) respectively of the outputs had been saved. The time required for the configuration of the memory template and the copying of data is constant for an application, irrespective of when a relocation request is received.

It was also observed that the outputs of both the original circuit and the relocated equivalent for the same inputs. The results were the same for both circuits – in both cases, the value of $Data\_out$ when the ap_done signal goes high was the same. This is shown in Figure 6.7. In addition, the improvement in the number of possible

relocations brought about by incorporating the proposed technique into the state-of-the-art direct bitstream relocation technique was evaluated. Table 6.4 shows the result for different Xilinx FPGA chips. As shown, the proposed technique leads to a significant improvement in the number of relocations. For the chips compared, an average of about 36 more relocations (an increase of over 260%) of the case-study circuits could be obtained using the proposed technique. This is a great advantage in applications which aim to improve reliability by circumventing permanent damage on the chip. It means that augmenting the traditional direct bitstream relocation with the proposed functionality-based technique would significantly improve the fault tolerance of a design.



(a)



(b)

Figure 6.7: Output Waveforms of Original and Funtionality-Based Relocated Circuits.
a) Original CORDIC Circuit b) Relocated Equivalent

Table 6.4: Improvements in Number of Possible Relocations Due to FBR

Target Chip	**Only DBR**	**DBR+FBR**
Artix-7 (xc7a35tcpg236-1)	1	8
Kintex-7(xc7k325tffg900c-2)	19	64
Virtex-7 (xc7vx485tffg1761c-2)	21	77
Total	41	149

As already noted above (and shown in the relocation case study used), the relocator resorts to a functionality-based relocation when the bitstream of the original design cannot be placed on a matching location on the chip, leads to undesired effects, or where access to the location information of the bitstream is not possible (such as in encrypted bitstreams). The technique is especially suitable on modern heterogeneous FPGA chips, such as the Xilinx 7 Series and UltraScale FPGAs, which are rich in memory resources, many of which are sometimes unused. It has also been noted above that relocation by functionality is only applicable to circuits with low port width. This is due to its memory overhead not scaling well with port width, and thus resulting in large overheads for large-port-width circuits. To this end, it is important to restate that the relocator system presented is also capable of bitstream relocation for circuits which *cannot* be memorized.

In addition, the time overheads of direct bitstream relocation and the proposed functionality-based relocation were compared. Table 6.5 shows the relocation time for both techniques for 3 different circuits: CORDIC [104], RGB to YCrCb colour converter [119] and a multiplier circuit [120]. All the circuits were implemented using Xilinx IPs from Xilinx Vivado 15.1 for the xc7a35tcpg236-1 chip. As shown, functionality-based relocation technique has a larger time overhead than direct bitstream relocation for a majority (2 out of 3) of the cases. For example, direct bitstream relocation duration for a 12-bit RGB to YCrCb colour converter circuit would only require 174.46 µs as against a minimum of 326.15 µs required for the

functionality-based technique. It is worth noting that the relocation time for the functionality-based technique is proportional to the port width of the circuit. Hence, for the CORDIC circuit with 10-bits inputs, its relocation time is smaller than direct bitstream relocation. With increase in port width, the relocation time for direct bitstream relocation has better performance. A major disadvantage of functionality-based relocation technique is that it does not scale well with increase in port width. In fact, the memory requirement doubles for each bit increase in port width. However, since direct relocation is *impossible* in certain cases such as for encrypted bitstreams and when an identical location is not present on the chip, servicing relocation requests whose time constraint can be satisfied in those cases is always an advantage. Therefore, it is an added layer of advantage to relocate circuits by functionality whenever direct bitstream relocation is *impossible* or leads to undesired effects.

Table 6.5: Comparison of the Relocation Time Overhead of Different Relocation Techniques

Circuit	**Port Width (no of bits)**	**Relocation Time Overhead (µs)**		
		Direct Bitstream	**Functionality-based (best case)**	**Functionality-based (worst case)**
CORDIC	10	369.02	131.44	361.86
RGB to YCrCb	12	174.46	326.15	367.63
Multiplier	16	92.54	774.88	1430.26

Finally, the size of the additional memory template bitstream required for functionality-based relocation is only 55% of that of the original circuits' in our case study. Hence, in terms of additional memory required, the functionality-based technique would be better compared to having to store multiple bitstreams of the original circuit, not to mention that since it is an empty memory template most of the bits in its bitstreams are '0's and would be much smaller when compressed compared to the original circuit's bitstream.

6.3 Chapter Conclusion

In this chapter, a novel functionality-based technique has been proposed to complement direct bitstream relocation on COTS FPGAs. This aims to alleviate the effect of lack of matching locations on heterogeneous FPGAs programmable logic which limits DBR. The additional FBR capability is based on replicating the functionality of the original circuit by memorizing its previous computation results. The memorized results are then used to mimic the functionality of the original circuit at another location on the chip where the original circuit cannot be configured due to lack of matching resource, but which supports the memory template. The performance evaluation of the proposed technique shows that it has the potential to increase the amount of relocation that can be carried out on heterogeneous COTS FPGAs. Given that relocation is a major technique used by ROS for achieving both high performance and reliability, the FBR proposed technique has the potential to improve the degree of performance and reliability of ROS based on a combined DBR and FBR.

However, the proposed FBR is only applicable to certain circuits – those whose outputs do not depend on internal states, but rather on only the current input. In addition, its memory overhead is significant for applications with large port width. Hence, the proposed technique is limited to circuits with low port width and are referentially transparent. Thus, FBR is recommended to be used to augment DBR. The content of this chapter is included in the following publication:

- G. Enemali, A. Adetomi, G. Seetharaman and T. Arslan, “A Functionality-Based Runtime Relocation System for Circuits on Heterogeneous FPGAs,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 5, pp. 612–616, May 2018.

In the next chapter, some of the low-level practical implementation details of a prototype placement management system incorporating the proposed techniques in the previous chapters are presented. It aims to show the practicality of the proposed techniques on an actual COTS FPGAs.

Chapter 7: Placement Management System Implementation and Characterization

This chapter presents a prototype hardware implementation of the proposed placement management system (PMS). The techniques relating to runtime placement of circuits discussed in the previous chapters are implemented and characterised in this chapter. The low-level implementation details are also discussed. In addition to bringing together the various components of the placement system described above, it also shows the practicality of the ideas proposed. Unlike many other proposed runtime placement systems, one of the aims of this work is to avoid proposing an ideal system for an ideal platform, rather, the proposed system is designed for actual COTS FPGAs. In this implementation, Xilinx 7 series FPGA platforms have been used, but this is easily extensible to other versions of reconfigurable FPGAs. In addition, Xilinx Vivdao v15.1 design tool has been used to carry out the experiments, including timing and resource overhead measurements.

The implementation presented here relates only to the run-time phase of the placement management system. Details of the design phase of the system is presented in chapter 4 of this thesis. Also, it has been assumed that placement requests are generated by a higher application which requires pre-synthesized tasks to be placed in runtime. A task graph has been used to replace the function of such an application to enable easy testing of the placement module presented here. In addition, the routines for delivering clock networks to place tasks, and the means of maintaining communication with a newly placed task (or a relocated task) are not discussed here. Efficient clock network delivery routine via the configuration layer discussed in the next chapter. Task communication is not covered in this thesis. Details of the communication architecture adopted in this thesis can be found in [26].

This chapter is organized as follows: first, a flow of the runtime phase of the placement request management is presented. This includes the various stages the PMS goes through in a bid to service a typical placement request, leading to task

placement location or task reject. Thereafter, details of the architecture of the proposed PMS is presented with each component discussed and analyzed. Finally, the hardware implementation results are discussed and compared with other placement systems on FPGAs, with a discussion on the balance between the overhead of the proposed PMS and its features.

7.1 Summary of Runtime Placement Flow

Figure 7.1 presents a summarized flowchart of the runtime placement management system. An initialization step is required by the placement system. This step is executed at the beginning of placements when changing the set of tasks which the placement system manages. The step consists in reserving space for the parameters of each of the tasks to be managed on a memory by saving the tasks' parameters on the Task Memory.

After initialization, the system waits in an idle state for interrupts. Two types are anticipated in regard to tasks placement on the chip: New Task Placement (NTP) and Pending Task Placement (PTP). When the former is requested (by a scheduler), the timing constraints of the request is checked to see if its deadline can be met. The possible result of this test and their associated decisions are:

i) Insufficient compute time ($t_e > t_d$) – in this case the task is rejected immediately.

ii) Sufficient compute time, but insufficient configuration duration ($t_e > t_d; t_e + t_c < t_d$) – the placement system proceeds to check if an already configured instance can be used to execute the task. The instance could be idle, or executing another task which would end *soon.* If an idle instance exists, it is assigned to the newly arriving task. In the absence of an idle instance, the arriving task is set to queue if a computing instance is found which satisfy the condition that: ($t_r < t_d - t_e$), where t_r is the remaining execution time of the computing instance. If none of these two possibilities exists, the task is rejected for timing reasons.

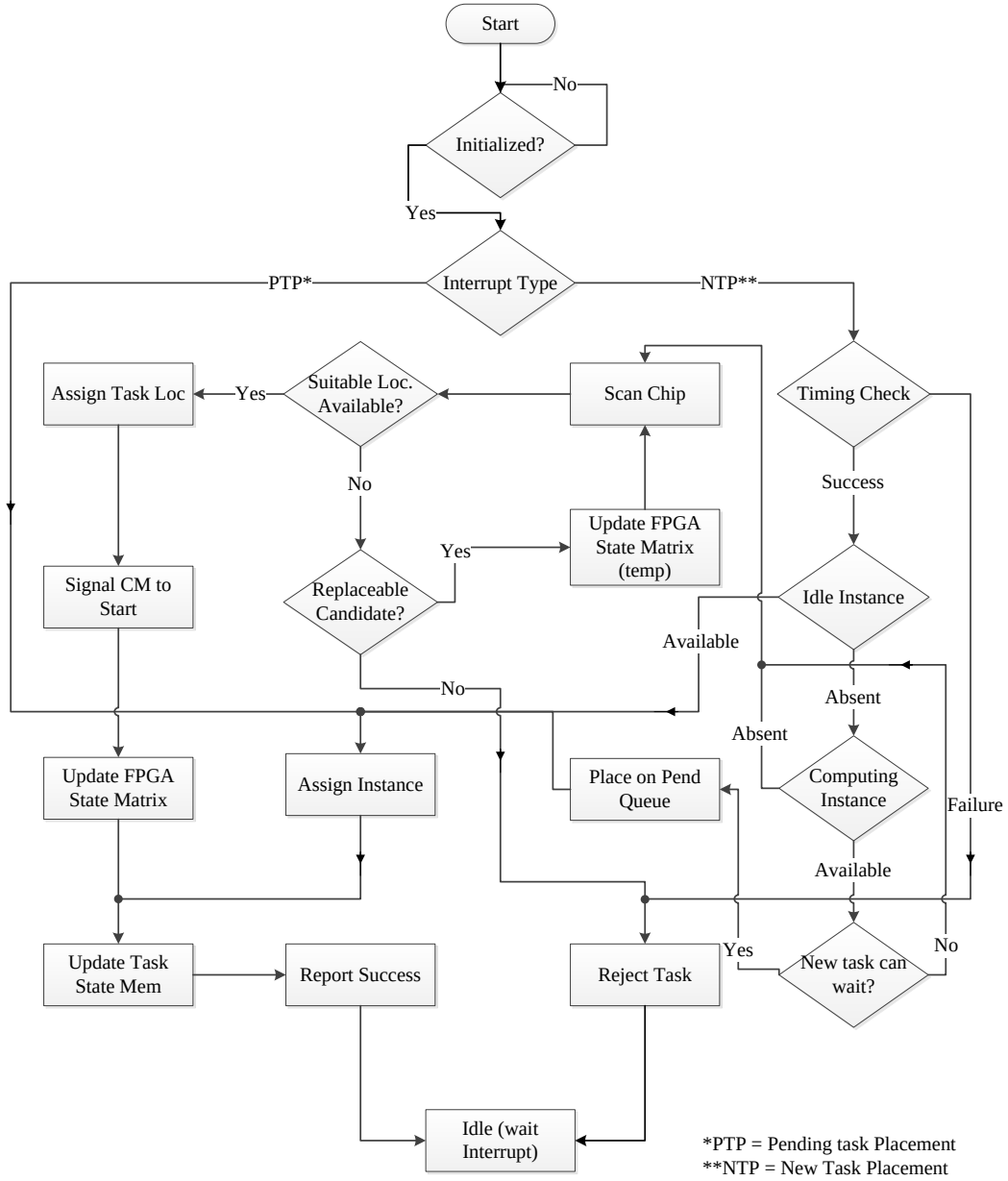


Figure 7.1: Summary of Main Operations of runtime Placement Management System

iii) Sufficient compute time and configuration duration ($t_e + t_c > t_d$). In this case, both of the tests in (ii) are carried out, however if both of these fail,

instead of rejecting the task, a third option is considered: a scan of the chip area is done to find a location of the chip where the task's bitstream can be configured. Failing to find an unoccupied location on the chip, a final option of replacing idle instance(s) to accommodate the new task according to the replacement policy described in chapter 5 of the thesis is considered. The task is rejected if none of these is successful.

A task can be successfully placed by being assigned to an already configured instance or by a new location being assigned to it on the chip. In the former case, the placement system updates only the states of the task in the memory to correctly designate them as either computing or pending. In the later, an FPGA State Matrix, which keeps the states of each resource, is also updated to reflect the current state of the chip. The process of updating the memory of the placement system is done concurrently with the process of task configuration. Thus, the configuration manager is signalled to begin writing of the configuration memory as soon as a new location is found for an arriving task.

7.2 Placement System Architecture

Figure 7.2 shows a block diagram of the architecture of the placement management system. It consists of 5 main modules and 3 blocks of memory. The modules are: initialization, reuse, scan, replace and update modules. In addition to these, 3 memory blocks are used to keep track of the state of the tasks and the chip area. They include: Arriving Tasks Queue (ATQ), FPGA State Matrix and Layout memory (FSML), and a Task State Buffer (TSB). The details of these modules, from an implementation perspective, are discussed below.

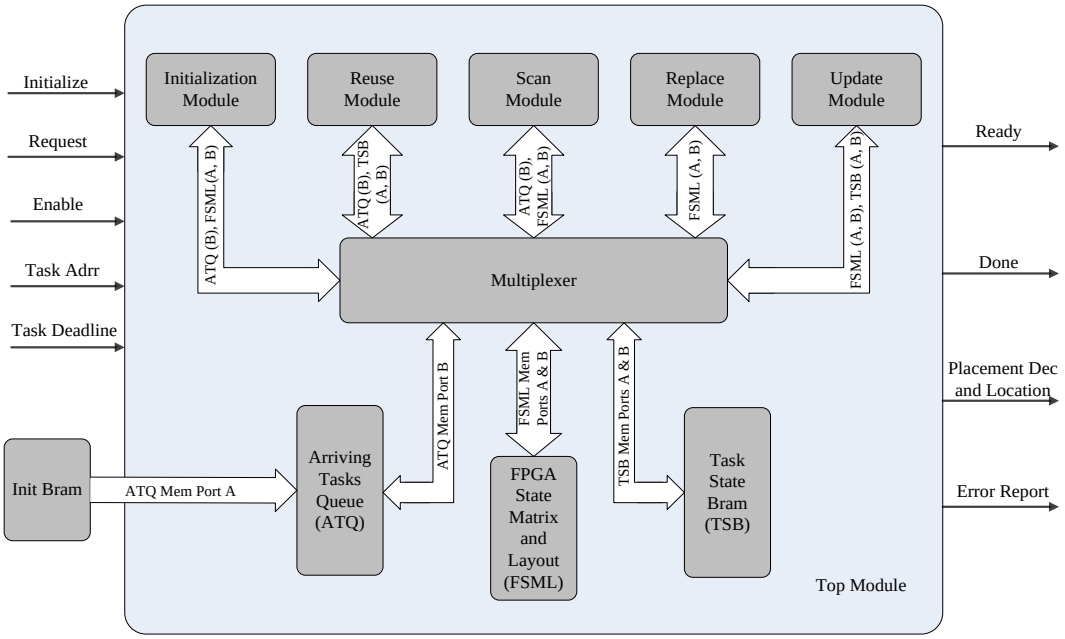


Figure 7.2: Block Diagram of the Placement Management System

7.2.1 Initialization Module:

The architecture of this module consists of a simple state machine which reads the content of the constituent tasks in an *init buffer* (IB) and uses the information to reserve space and initialize the parameters of the tasks in the TSB. Prior to the assertion of the 'Initialize' signal, the IB must contain all the essential parameters of all potentials tasks.

The parameters of a task in this case study implementation include a set of area related information: Length, l, Width, w, and a start column on an intended heterogeneous chip, $RsId$, such that the next $(l-1)$ contiguous columns to the right of $RsId$ is a matching position for the task on an intended heterogeneous chip. These are distributed in the IB as shown in Figure 7.3 (a). In addition, a set of time-related information: configuration time (t_c) and execution time (t_e) are also stored in the IB as shown in Figure 7.3 (b). A task is stored in the IB using two 33-bit words with the first (Word A) containing the area related information and the second (Word B) for time related information. Some of the bits in the IB are reserved (R) as the memory

has been configured to be compatible with other memory blocks in the system for ease of data exchange. The first address of the buffer contains the number of tasks (N) to be managed by the system.

	[32:20]	[19:17]	[16:9]	[8:1]	[0]
Word A	R	w	l	$RsId$	R

(a)

	[32:17]	[16:6]	[5:0]
Word B	t_c	t_e	R

(b)

Figure 7.3: Data Distribution in Init Buffer of Placement System

During the initialization process, the $FSML$ which holds a matrix corresponding to the resources on the chip, is initialized to '0' with the exception of damaged resources which are marked-off with a '1' at their location. The $FSML$ Memory is configured as 32-bit wide, with its useable depth dependent on the size of the chip. For example, the content of the section of the state matrix in the $FSML$ buffer for a Xilinx xc7z100ffg900-2 chip is shown in Figure 7.4. The device consists of 134 columns of dynamically reconfigurable resources organised in 7 rows as shown in the figure. It is worth noting that the portion of the memory between the end of the 134th columns of a row and beginning of the 1st column of the next row is marked off permanently as not available as shown.

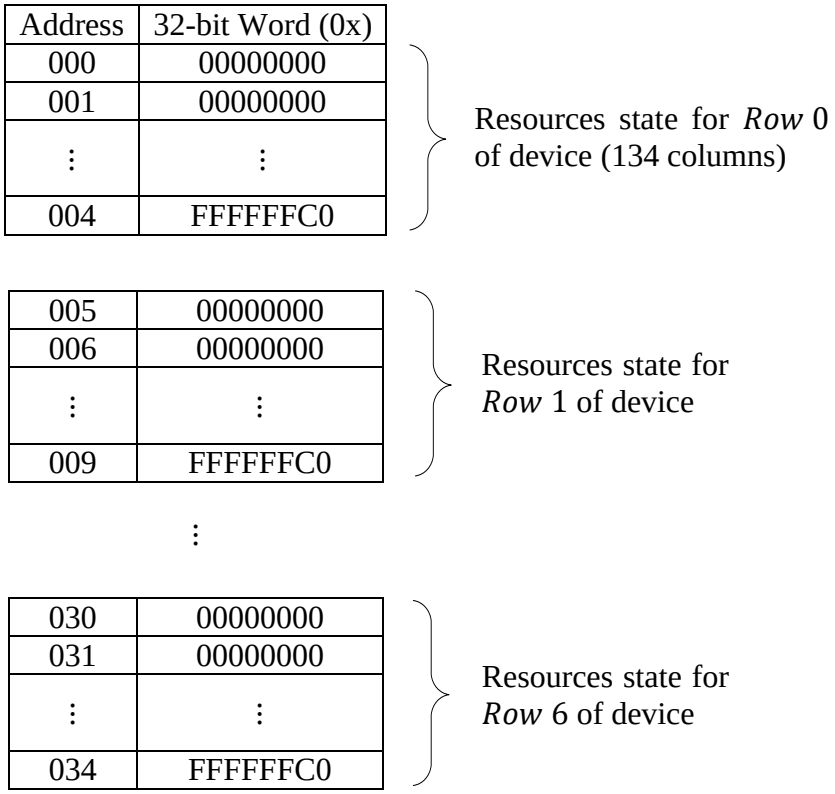


Figure 7.4: Example of Initialized State Matrix in the $FSML$ Buffer for a Xilinx xc7z100ffg900-2 FPGA

The TSB is also initialized. The memory, organised as 33-bit wide similar to the IB, is divided into 3 sections. The first holds parameters of idle instances, the second those of computing instances and the third contains pending tasks, waiting on computing instances. The idle instance section is designed to contain information relating to:

1. the physical location of the instance on the chip, expressed as start column and row (x, y)
2. area properties of the instance, including the width (w), length (l) and the start column of first matching location on the chip $(RsId)$
3. configuration time of the task
4. number of times the instance has been reuse since its latest configuration on the chip

The first 2 of the information are contained in the first word (Word A) of each instance, with the bits distributed similar to that in of the IB, but with the exception that the reserved fields (R) are now allocated for the desired content. Bits [32:25] are designated to store the horizontal location information (x) of the instance, bits [24:22] store the vertical location information (y), bits [21:20] of Word A is reserved (R) for idle instance, while bit [0] (valid bit, v) is monitored to ascertain if the instance is idle (value '1') or not (value '0'). The last 2 information saved for an idle instance (i.e. 3 and 4 in the list above) is contained in the second word (Word B) of the instance, with its upper 16 bits [32:17] used to save the configuration time and lower 16 bits [15:0] containing the number of instance reuse, N_r. Bit [16] is reserved.

During the initialization stage, bit [0] of Word A is set to '0' for all tasks as placement is yet to commence, bits [19:1] of Word A are copied from corresponding bits in the IB, bits [21:20] are not used since they are reserved, hence they are each set to '0', bits [24:22] and [32:25] (location x and y respectively) are set to a value equivalent to an invalid location on the chip. For example, for the xc7z100ffg900-2 FPGA, they are set to all '1's. In runtime, when a task finishes, and their instance is marked as idle, v is set to '1', and the location of the instance is updated to instance's current location. For word B, bits [32:17] are copied from corresponding bits in the IB while bits [15:0] which correspond to number of reuse are all set to '0'. Figure 7.5 shows the distribution and initialization values a typical idle instance section of TSB.

The computing instance section of the TSB contains 5 words (Words A – E) per task. Like the idle instance section, Word A of the computing task section contains the information about the current location of the instance on the chip (x, y) as well as the area properties of the instance (l, w and $RsId$) while its lsb indicates the state of the task (currently computing or not). However, bit [20] is used to indicate whether another task is waiting on the currently computing instance or not. This bit is set ('1') if a task in a pending state is waiting to use the same instance, otherwise its value remains '0'. The entire bits in the second word (B) for an instance in this section contain the start time of the task currently computing on it.

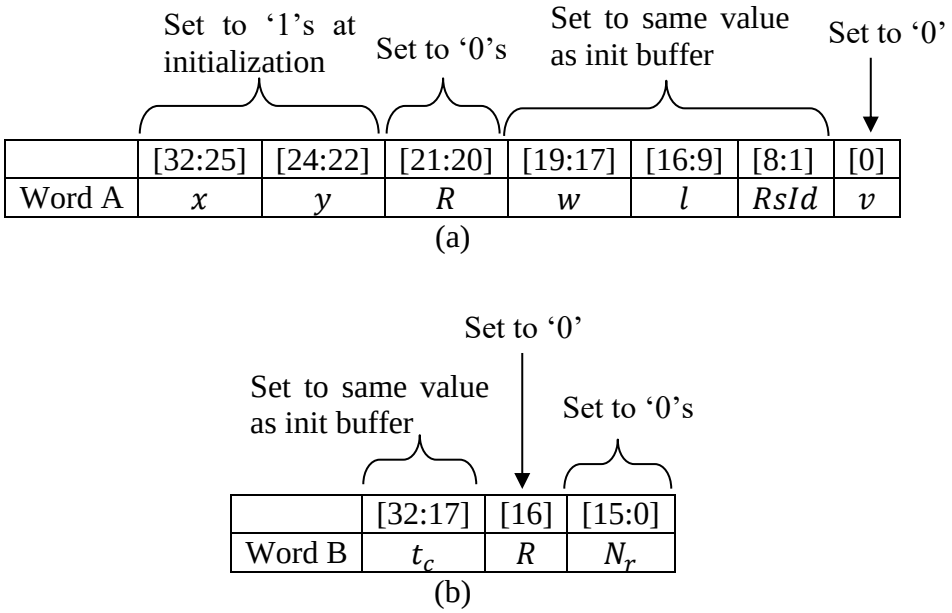


Figure 7.5: Data Distribution and Initialization Value of an Idle Instance

Word C store the configuration and execution duration of the task, similar to Word B of the IB, with the exception that bits [5:0] which are reserved in the IB are used to keep the unique ID of the task computing on the instance. Word D contains the deadline information of the task and Word E contains the total execution duration of all tasks waiting on the instance. During the initialization process, Word A is set to the same bits as word A of the idle instance section of the TSB, word C is copied from the IB, while all bits in Words B, D and E are set to '0'.

The pending task section of the TSB uses three 33-bit words per task to save the state of a task waiting to use a configured instance which is busy. The bits in the first word are distributed exactly like Word A of the idle instance section. Similarly, the bits distribution in the second and third words are the same as the Word C and Word D of the computing task section respectively and are initialized to the same values.

7.2.2 Reuse Module

The reuse module aims to reduce the number of configurations carried out during runtime placement of tasks on a chip. As explained in chapter 5, reconfiguration overhead constituent a bottleneck that need to be addressed when FPGAs are used for runtime applications. Task reuse is a technique used to reduce the amount of reconfiguration needed for runtime applications and thus make the single configuration access port more available to other duties as such as soft error mitigation [22].

Before new tasks are configured on the chip, the reuse module checks to see if any configured instance is suitable to execute the requested task. As shown in Figure 7.1, this module checks a list of idle instances and the computing instances. In order to avoid a greedy search of the entire list of idle and computing instances, a memory reservation technique is adopted where the lsb of the word A of the idle instance and computing instance are checked to see if the idle or computing. To do this, request for task placement is done by the scheduler using an address of tasks in the ATQ. The reuse module decodes the address and requests the states of instances in the idle and computing instance sections of the TSB. This check takes 3 clock cycles, with the first being used to set address to the TSB and 2 clock cycles needed to read the content of the memory. TSB is configured as a true dual port memory [95], thus the states of both the matching idle and computing instance are checked simultaneously.

7.2.3 Scan Module:

The scan module looks for a matching physical location for a task on the chip. This module is enabled only after the reuse module reports an absence of an idle instance that can immediately execute the task and there is either no matching computing tasks or the timing constraints of the placement request is such that the task cannot wait for a computing instance to become free.

When a task placement request is received, a suitable location is found for the task on the chip. This is done by first scanning the chip (with the scan module) to determine the availability of matching positions, which is a subset of the total number of possible locations. To do this, an up-to-date state of the chip is maintained in a buffer called *Chip State* as a 2D matrix, M. Each element, $M_{(x,y)}$ in the matrix corresponds to a unit of resource located at the horizontal distance x and a vertical distance y from the top left corner of the chip. For instance, $M_{(0,\ 0)}$ refer to the resource in the first row of the first column. The value at each of these positions indicate whether the resource is available (value '0') or not (value '1'). Both resources temporary occupied by a task or permanently damaged have same value of 1, but a record of permanently damaged resources are maintained differently.

For a chip scan operation, the *Chip State* is read and compared with the resource requirements of the task to be placed. The parameter $RsId$ is used as the start position of the first scan. The scan begins by checking the state of the resource at the position corresponding to the horizontal position of the first row (i.e. $M_{(RsId,\ 0)}$) and progresses by checking other elements to the right of $RsId$ until the length requirement l of the task is satisfied. (i.e. $M_{(RsId+1,\ 0)}$, $M_{(RsId+2,\ 0)} \cdots M_{(RsId+(l-1),\ 0)}$). Thereafter, the vertical term, y is incremented (with x reinitialised to $RsId$) until the width requirement w of the task is satisfied. At every stage in the comparison, if any value of the matrix term is '1', the search is aborted, the values of l and w reinitialised and the scan restarted using the next row. This is repeated for each encounter of '1' until all vertical locations corresponding to the current $RsId$ are scanned. Next, a new horizontal scan location is determined by computing $RsIds$ for the task.

New $RsIds$ for a task can be computed first by constructing the layout of the task using its original $RsId$ and l. This is compared with the static layout of the first row of the chip until a matching layout is found. This process can often be time consuming. Since this is done in runtime, it is necessary to optimize the process for speedy determination of placement locations. Two approaches are considered depending on the application requirements. The first is an extension of the technique

proposed in [73] which is to search for matching locations using the heterogeneous resources which has the least number of occurrence in the task's layout. Searches are made using locations of heterogeneous resources (mostly BRAMs and DSPs) first, before checking the location of the more abundant homogenous resources (CLBs). However, they assumed that the heterogeneous resources are regularly spaced on the chip which is not the case for COTS FPGAs. Nevertheless, the technique can be extended to chips with irregularly-spaced heterogeneous resources as well. To do this, the location of the heterogeneous resources in a task's layout are checked against the known location of corresponding heterogeneous resource on the chip.

This is illustrated as follows: for the STAT task in Table 4.2 on a Xilinx xc7z100ffg900-2 chip. The task has $l = 6$, $w = 1$, and a possible matching of location of the task has $RsId = 4$ and resource layout of $BRAM - CLB - CLB - DSP - CLB - CLB$. The chip layout consists of 12 BRAM columns (located in columns $(4, 15, 20, 32, 43, 57, 77, 94, 108, 115, 121, 128)$ of the device. The search for a new $RsId$ begins from the BRAM location on the chip next to the previous $RsId$ location. This is illustrated in Figure 7.6. In this case, the search for a new start location ($RsId_1$) begins from column 15 of the device, then successively checking if column 16, 17 … matches the layout of the task. If a matching location does not result from the search, then the search restarts from the next BRAM column of the device which is column 20 ($RsId_2$). In this example, a matching location would be found for with a start scan from $RsId_2$.

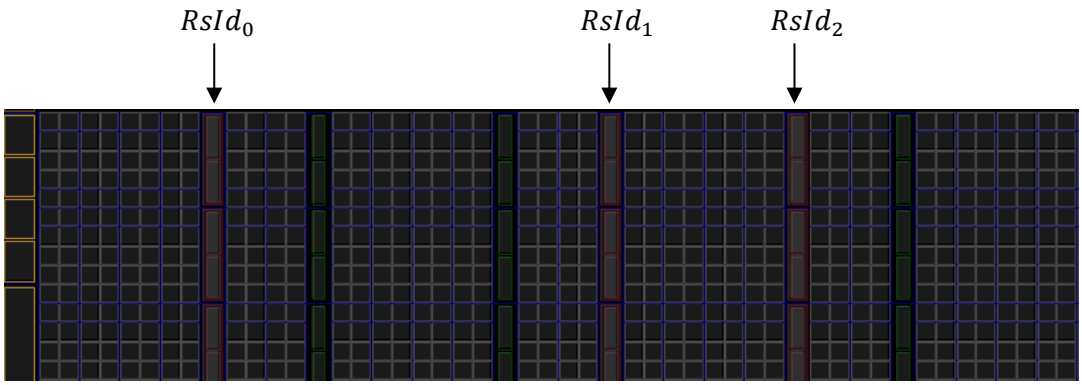


Figure 7.6: Start Scan Locations to Accelerate Resource Scanning

The location of the heterogeneous resource used as a start column does not necessarily have to be the first column of the task. In cases where the heterogeneous resource is located at the kth location in the task's resource layout, the chip is scanned first in the forward direction by $l - k$ locations beginning from the heterogeneous resource before scanning k locations to the $left$ of the heterogenous resource location. A speed up of approximately 11 times is obtained by beginning search from the locations of BRAMs and 9 times by beginning search from DSPs on the xc7z100ffg900-2 chip compared to linearly searching all columns on the chip.

Another means of speeding-up finding potential placement locations is to precompute all possible $RsId$s for the task on the chip at design time and store these in addition to the task parameters described in section 4.1. This simplifies the runtime phase to reading the potential locations from memory and scanning the chip to determine if all the columns in the location are available or not. The main disadvantage of this over runtime computation of $RsIds$ is its increased memory overhead. However, it is a good choice for applications which require faster runtime placement where the required memory overhead is not a constraint.

An important aspect for the scan module is being able to *quickly* find an optimal location for the task on the chip if it exists. It is worth noting that the reuse module has a constant execution time, due to the memory reservation technique. However, the scan duration of the module depends on the task properties and the state of the chip at the time of scan, and therefore varies for each operation. Hence, it is important to minimize the worst-case scan time of this module. To achieve this, a method similar to that proposed in [6] was adopted where possible horizontal start locations, $RsId'_i$, of tasks on the chip are stored along with the task parameters. These are read into the TSB during the initialization process. In addition, for each $RsId$, scans are conductor vertically first as each row on our target device have the same repeated pattern in all rows.

The worst-case timing requirement for a scan depends on three factors:

i) the number of $RsId'$ available for the task on the chip,

ii) the length, l, and width, w, of the task

iii) amount of time required to compute the fragmentation of a potential location.

Table 7.1 shows the number of clock cycles required for the constituent operations of the scan module. As shown, a total of 6 clock cycles are required to read a word from the $FSML$ and determine if the corresponding location of the chip is available or not. In addition to those shown, two additional clock cycles are required at the beginning of the scan operation (used to compute initial read address to the $FSML$ memory). An additional clock cycle is also required to assign a location to the PMS port. These are incurred once and hence have been omitted from the table since the analysis will first focus on recurrent operations. They would be added at the final stage.

For a task with length, l and width, w the total number of clock cycles require to scan an area t_{scan} equivalent to the task is given by the expression in 7.1. The ceiling operation in 7.1 represents the number of times the $FSML$ is read to cover the length of the task when the memory is configured as 32-bit. For example, a task with $l = 32$ and $RsId = 2$, $FSML$ would be read twice per row per scan. Since scan are done vertically first, the worst-case scan duration ($t_{wc\ scan}$) for each $RsId$ is given by equation 7.2, where $N_{Device\ row}$ is the number of rows in the device.

Table 7.1: Clock Cycles Required for Constituent Operations of Scan Module

Operation	Duration Clock cycles)
Read State (32-bit word)	1
Align word to task	2
Check availability	1
Task width check	1
Task length check	1
Total	6

$$t_{scan} = 6w * \left\lceil \frac{l + \left\lceil \frac{RsId}{32} \right\rceil}{32} \right\rceil \quad (7.1)$$

$$t_{wc\,scan} = (N_{Device\,row} - (w-1))t_{scan} \quad (7.2)$$

It is worth noting that since the proposed placement module aims to minimize fragmentation of the chip area, a fragmentation coefficient (FC) is computed for each successful scan before proceeding to the next. The details of how FC is computed is shown in chapter 5 section 5.1.1. Hence, the overall worst-case scan duration for the scan module is shown in equation 7.3, where the t_{fc} is the duration required to compute the fragmentation co-efficient of a potential location and $N_{RsId'}$ is the number of possible horizontal start locations for the task. Two additional clock cycles are incurred to retrieve subsequent $RsId's$. Equation 7.4 is an estimation of the worst-case overhead for computing the fragmentation coefficient for each location. As shown by equations 5.1 and 5.2, the computation is a function of the dimension of the chip and the task size. Three fixed clock cycles are incurred: 2 at the beginning of reading the $FSML$ content and an additional clock cycle for computing FC.

$$T_{wc\,scan} = \left(t_{wc\,scan} + t_{fc} + 2\right)N_{RsId'} \quad (7.3)$$

$$t_{fc} = \frac{1}{2}((N_{column} - l)l + (N_{Device\,row} - w)w + 3) \quad (7.4)$$

7.2.4 Replace Module

In the event that none of the instances present on chip can be used to execute the requested task and a vacant location cannot be found in the current state of the chip, the $Replace\ Module$ is activated to select idle instance(s) to be deallocated from the

chip area to accommodate new placement requests. It uses FAReP, a fragmentation aware replacement policy [23] to select a candidate instance to be replaced. To minimize the total amount of configuration required by an application as well enhance a better area utilization in the process, this module uses three main factors to decide a candidate for eviction:

- The configuration duration of an instance (instance with large configuration duration are more likely to be preserved)
- The frequency of instance reuse (more frequently used instances are more likely to be preserved)
- The fragmentation coefficient of an instance (instances whose location are such that they contribute lower FC to the chip are more likely to be preserved).

This module sorts idle instances (potential candidates for replacement) using the above factors. The sorting process is done concurrently with the scanning of the chip. The sorting duration is proportional to the number of idle instances on the chip, as well as the number of duplicates. The worst-case duration for the sorting process ($t_{wc\ sort}$) is given by equation 7.5 where N_{idle} is the number of potential candidates for replacement.

$$t_{wc\ sort} = 2 + 2N_{idle}(N_{idle} - 1) \qquad (7.5)$$

The replace module sends out one victim per time from the sorted list until the new task can be placed or all candidates have been tested.

7.2.5 Update Module

After each placement (or deallocation) operation, the update module updates the TSB and the $FSML$. In addition, it uses interrupts from a computing task which has finished to update the TSB. In the case when a computing task which is waited upon

finishes, a PTP interrupt (shown in Figure 7.1) is generated for the PMS. There are seven scenarios involving the update manager and Table 7.2 gives a summary of the operations and the duration required to update the memories.

Table 7.2: Summary of Operations and Time Overhead for Update Module

	Placement Outcome	Memory	Duration (Clock Cycles)	Operations/Remark
1	New Task on Idle Instance	TSB	8	Set lsb of idle instance (to '0'), increase number of reuse of idle instance, Set lsb of computing instance (to '1'), update timing parameter of computing instance
2	New Task on Pending List	TSB	5	Write task to corresponding address on pending task section of TSB, mark computing task as waited upon, increase duration of tasks waiting on instance
3	New Task on New Location	TSB, $FSML$	$7 + 8 * w$	Set lsb of computing instance (to '1'), update location and timing parameters of computing instance, mark corresponding location of new task in $FSML$ as unavailable
4	Idle Instance Replaced	TSB, $FSML$	$10 + 8 * w$	Set lsb of idle instance (to '0'), reset location and number of reuse of idle instance, update location and timing parameters of computing instance, mark corresponding location of new task in $FSML$ as unavailable
5	New Task Rejected	-	-	-
6	Pending Task on Idle Instance	TSB	11	Same as 1

7.3 Hardware Implementation Results

The component modules described above were implemented for Xilinx's xc7z100ffg900-2 FPGA using Vivado 15.2. VHDL was used to code the routines in order to maximize the performance of the placement system tapping into the

parallelism offered by the hardware platform while sacrificing some of the chip resources. In this section, the implementation results are discussed.

7.3.1 Interface Signals

The case-study implementation of the placement management system has 5 main inputs and 4 outputs. Figure 7.7 shows the prescribed order for the assertion of the signals in order to perform operations using the system. As shown, the '*Initialize'* signal which is used to control the set-up of the various memories of the system is asserted for one clock cycle to trigger a one-time initialization process, which is only needed when new set of applications begin to execute, or a complete system reset is necessary, say, after power-down. After pulsing the '*Initialize'* signal, the user is required to wait for the system to finish initialization which is indicated by '*Ready'* going high. Table 7.3 shows the main interface signals' properties as well as possible values for this example implementation.

The '*Request Type*' signal dictates the operations the placement system will perform. Its value must be set before asserting the '*Initialize'* or '*Enable'* signals. Except when set to a value of "Initialization", where it must be held at that value for a minimum of one clock cycle, the '*Request Type*' must be held at its value after the assertion of '*Enable'* until the going high of the output signal '*Done'*. This constraint is also true for '*Task Address'* and '*Task Deadline*' as well. '*Task Address'* and '*Task Deadline*' are respectively used to state the location of the task parameters in ATQ and the deadline of the task. The values shown in the figure are random values chosen for illustrative purposes only. *Enable* is pulsed for each operation after setting all the other input values.

New placement requests are queued until *'Ready'* and '*Done'* are both high. '*Done'* goes high once a placement decision and location have been obtained. Task configuration can begin after that, while the placement system updates its memories. '*Ready'* goes high after all memory updates are completed and the system is ready to accept new placement request. It is worth noting that the *'Update Status'* signal is an internal signal and is shown in figure 7.7 for clarity purposes only.

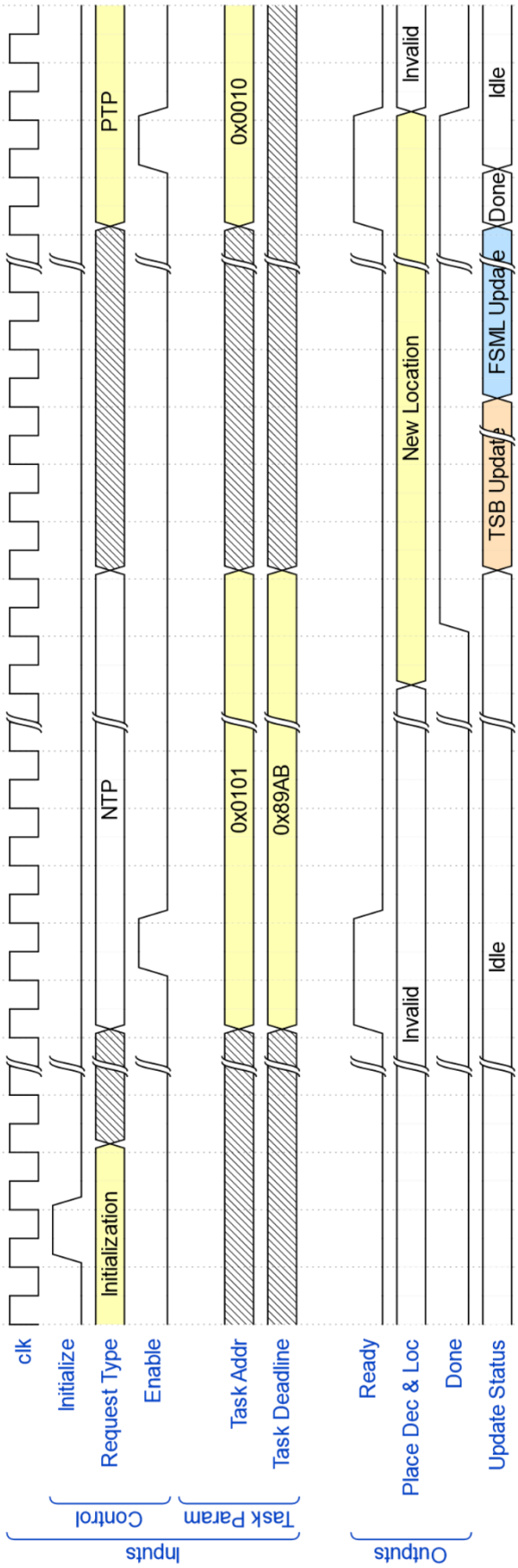


Figure 7.7: Example Waveform for Placement System Interface Signals

Table 7.3: Interface Signal Properties of PMS

Signal	Direction	Width	Remark
Initialize	In	1	—
Request Type	In	2	"00" – No operation "01" – PTP "10" – NTP "11" – Initialization
Enable	In	1	—
Task Address	In	9	—
Task Deadline	In	33	—
Ready	Out	1	—
Place Dec & Loc	Out	20	[19:16] – Placement decision: "0xF" – invalid; "0x2" – New Task Assigned to idle Instance; "0x3" – Pending Task Placed on idle instance; "0x4" – New Task Placed on Pending List "0x5" - New Location for new Task "0x6" – Idle instance replaced for new task "0x7" - New task Rejected (timing issues) "0x9" – New task Rejected (no area) [15:0] – Location of task (if successful placement)

7.3.2 Resource Utilization

As a potential core component of an ROS, the proposed placement management system has been implemented to support a variety of features. Not all these features are necessarily required by all ROS. Thus, the system has been designed to support easy removal of features not needed to reduce its resource utilization as desired.

Table 7.4 shows the resource utilization of the different components in the full implementation of the proposed PMS. The full PMS utilizes 2155 FFs and 2818 LUTs in addition to 4 BRAMs on the 7 series FPGA. This represent only about

5.18%, 11.55% and 5.33% respectively of the total FFs, LUTs and BRAMs present in the smallest 7 series device (xc7a35t). On the largest device in the series (xc7v2000t), these represent 0.09%, 0.23% and 0.31% respectively of the total resources present. An additional DSP is used in the top module due to address computation. These figures represent the resource overhead when the full features of the PMS are desired in an ROS. Hence, the resource overhead is lower in other scenarios. For example, when an ROS does not support task reuse, the Reuse and Replace modules could be stripped from the PMS reducing its resource overhead by 293 FF and 606 LUTs and 1 BRAM. In addition, the section of the update module's resources dedicated to updating the TSB as well as the TSB memory could be stripped off, including some resource savings from the initialization module. This leaves a resource utilization of the version without task reuse at 615 FF, 776 LUTs and 1 BRAM. The equivalent number of slices on the 7 series is 198.

Table 7.4: Resource Utilisation of PMS on a 7 Series FPGA

Component module	**Flip Flop**	**LUT**	**BRAM**	**DSP**
Initialization and Top	1258	614	3	1
Reuse	69	196	0	0
Scan	274	570	0	0
Replace	224	410	1	0
Update	330	1028	0	0
Total	2155	2818	4	1

Figure *7.8* shows the floorplan of an implementation of the proposed PMS (with other associated circuits) with a case study application on the xc7z100ffg900-2 chip. The chip area is divided into a static region and a reconfigurable region. The static region contains the core PMS, a configuration manager and a mechanism of

transferring configuration data from an off-chip memory (DDR Memory) to the CMEM which was implemented using the Xilinx's Direct Memory Access (DMA) IP. The DMA engine have a resource overhead of 1020 Flip Flops and 918 LUTs. These are included in a static region shown in the figure. The top 3 rows (row 0 to 2, with a read border in figure 5) of the chip is reserved for the static region due to the large IO requirement of the DMA engine. The other rows (row 3 to 6) is reserved as reconfigurable region and is shown hosting the data processing tasks of the NASA JPL spectrometer application The details of the application can be found in [105] and [121]. The tasks have been shown in their initial placement location on the chip with empty area for relocation in the case of permanent faults.

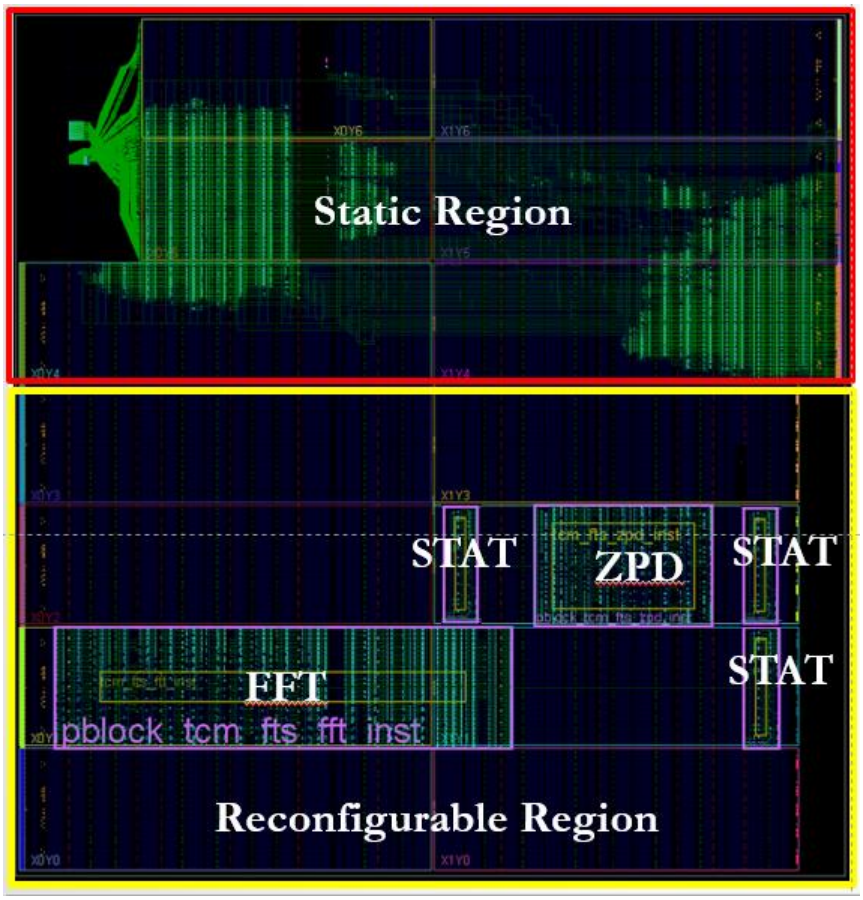


Figure 7.8:Floorplan of PMS, Configuration Controller, DMA Engine and A Case Study FTS Application

7.4 Comparison with Another Placement Management Module

Compared with existing solution, the PMS described above has a fair resource overhead considering the features supported by the PMS. The implementation results of the proposed PMS is compared with that of the R3TOS allocator in [63] which is also aimed at a non-slotted based ROS as well as being targeted for practical implementation on a COTS FPGA. Table 7.5 shows a comparison in terms of the supported features, the timing overhead and the hardware resource utilization of the two implementations. As shown, the R3TOS allocator does not include a task reuse strategy which can lead to significant configuration overhead in runtime applications. Configuration overhead also lead to a higher occupancy of the configuration interface, keeping it away from critical duties such as soft error mitigation. In addition, the proposed PMS has a faster placement time than the R3TOS allocator. The worst-case placement time for the R3TOS allocator was reported as 300 µs at 100 MHz for an FPGA consisting of 15 columns and 12 rows. This includes a 200 µs duration used to update the MER information on the chip (which the authors refer to as empty area descriptor updating). For the same FPGA size at the same frequency, the proposed PMS has a worst-case placement duration of less than 50% of that of R3TOS allocator. With a faster placement time, the proposed PMS is potentially able to react to permanent fault on the chip at least 2x faster than the R3TOS allocator, thus reducing system down-time in the case of fault occurrence.

Table 7.5: Comparison of Features and Overheads of PMS with Similar Schemes

Placement Scheme	**Feature**		***Worst-case Placement Time (Clock Cycles)***	**Resource Utilisation**	
	Heterogenous FPGA Support	***Task Reuse***		***Slices***	***BRAM***
R3TOS Allocator [63]	YES	NO	30,000	459+	4
PMS	YES	YES	14,868	198	1

+ Virtex 4 FPGAs have 4 input LUTs while 7 series used for PMS implementation have 6 input LUTs

The resource utilization of the two implementations were also compared. The R3TOS allocator has an overhead of 459 slices and 4 BRAMs. For the proposed PMS, the resource overhead are 198 slices and 1 BRAM without considering the reuse functionalities. The resource overhead of the reuse functionality was stripped before comparison as the R3TOS allocator does not include reuse. However, it is important to note that R3TOS allocator utilization was reported for a Virtex 4 FPGA which has two 4 input LUTs in a slice while 7 series used for PMS implementation have four 6 input LUTs in a slice. It follows that the number of LUTs come to approximately 918 4 input LUTs for R3TOS allocator, while 776 6-LUTs were used for the proposed PMS.

7.5 Chapter Conclusion

An implementation of the proposed runtime PMS was presented in this chapter. The aim of the chapter is to show the practicability of the techniques proposed in the thesis on COTS FPGA. The performance of the proposed PMS with respect to resource and timing overhead was also presented. To achieve a close connection between the PMS and other components of an ROS, part of the FPGA resources is sacrificed for the implementation of the PMS. A summary of the flow chart of PMS was presented and then detailed architecture of the proposed PMS system was discussed and analyzed providing low level implementation steps adopted in the prototype. A comparison of the hardware implementation result was also made with a similar runtime placement system. The results show that the proposed PMS is capable making placement decisions at least twice as fast as comparable systems, while having comparable resource utilization. It addition, it has additional features not present in the comparable system.

In the next chapter, a technique of routing clock networks to hardware tasks in runtime via the configuration layer is presented. The major control bits in the configuration layer that can be edited to route clock signals are identified, and a proposal of how the routing can be achieved in runtime is described. This is achieved

without jeopardizing the reliability of an application. Hence, the chapter also presents a means of avoiding the loss of reliability by an efficient means of re-computing Frame ECCs after editing configuration bits. This helps an application to retain its capacity to benefit from soft error mitigation techniques.

Chapter 8: Towards a Reliability-Aware Efficient Clock Routing for Reconfigurable Computing

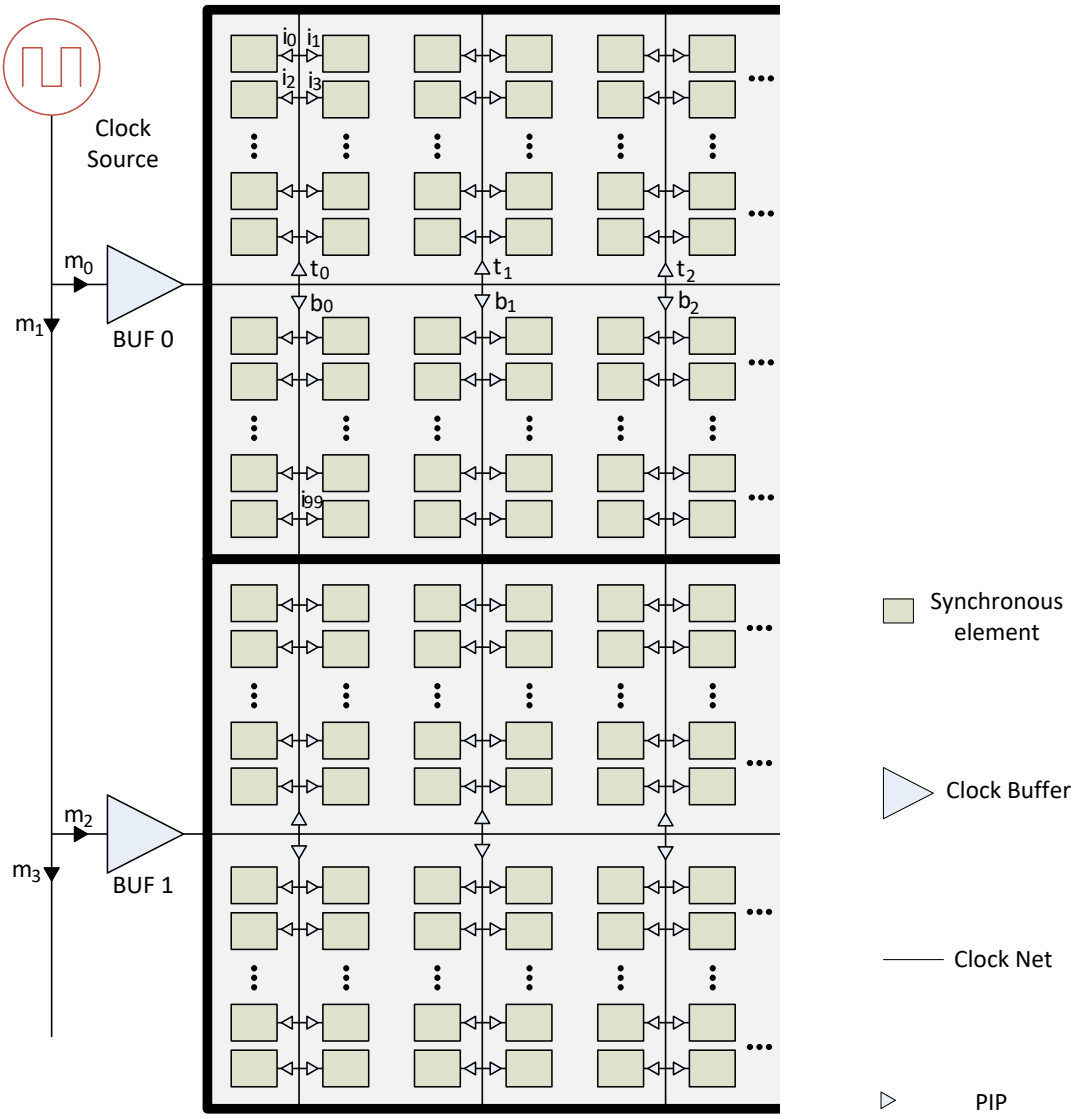
In this chapter, a runtime mechanism of clock delivery to tasks in runtime after placement on the chip is presented. When the locations of tasks are changed in runtime, the question of how clock nets can be delivered to the tasks reliably arise. As a step towards addressing this challenge, the proposed approach in this chapter is based on manipulating essential bits in the bitstream of an application in runtime. This involves identifying key control bits in the bitstream of the FPGA and controlling them in runtime.

The process of runtime editing of configuration bitstream is one which need to be done with care. First, the exact bits required to route a clock signal must be discerned to avoid editing wrong bits which can constitute a major damage to both the application and the device. As the locations of these bits are not provided by Xilinx, careful reverse engineering experiments are required.

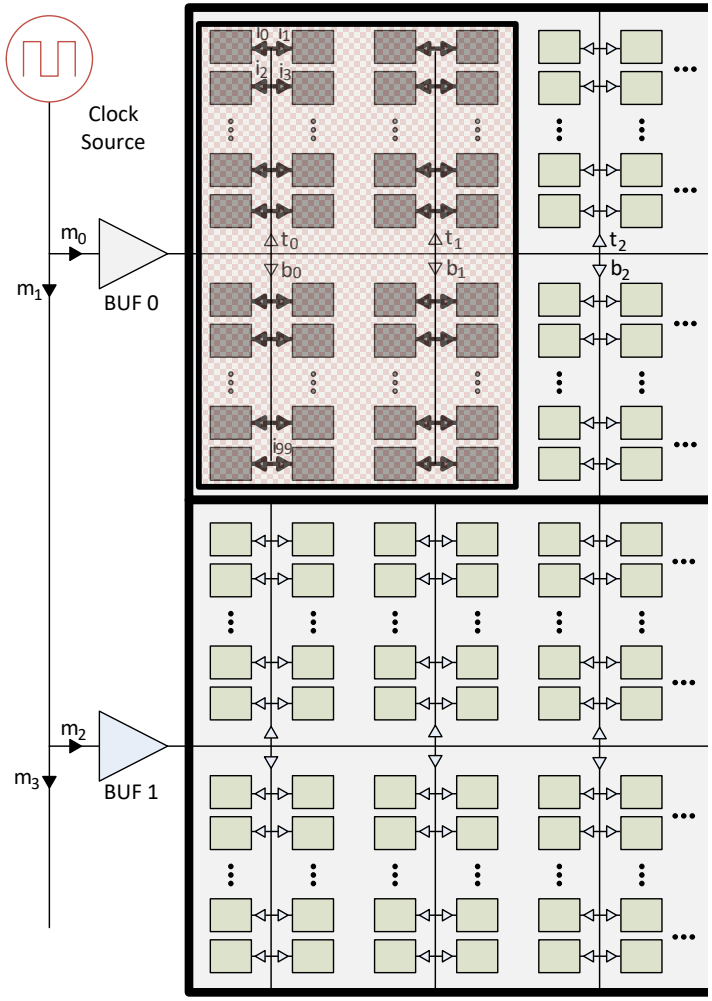
In addition, the process of editing the content of the configuration memory of an SRAM FPGA in runtime can lead to a major reliability risk for critical applications. This is because the configuration memory itself is affected by soft errors in the form of unwanted bit flips due to causes such as ionizing radiation. Unwanted bit flips in the configuration memory are monitored and corrected in critical applications using SEM techniques. These techniques use information stored as part of the bitstream generation process (called Frame ECC) to check if any bits have flipped. This correction mechanism cannot differentiate between intentional bit edits and soft errors. Hence, this chapter also present an efficient implementation of a runtime Frame ECC re-computation controller that enables soft errors to be tracked in designs where bitstream editing is used. The Xilinx 7 series FPGA is used as the target architecture in this chapter.

8.1 Efficient Runtime Clock Delivery

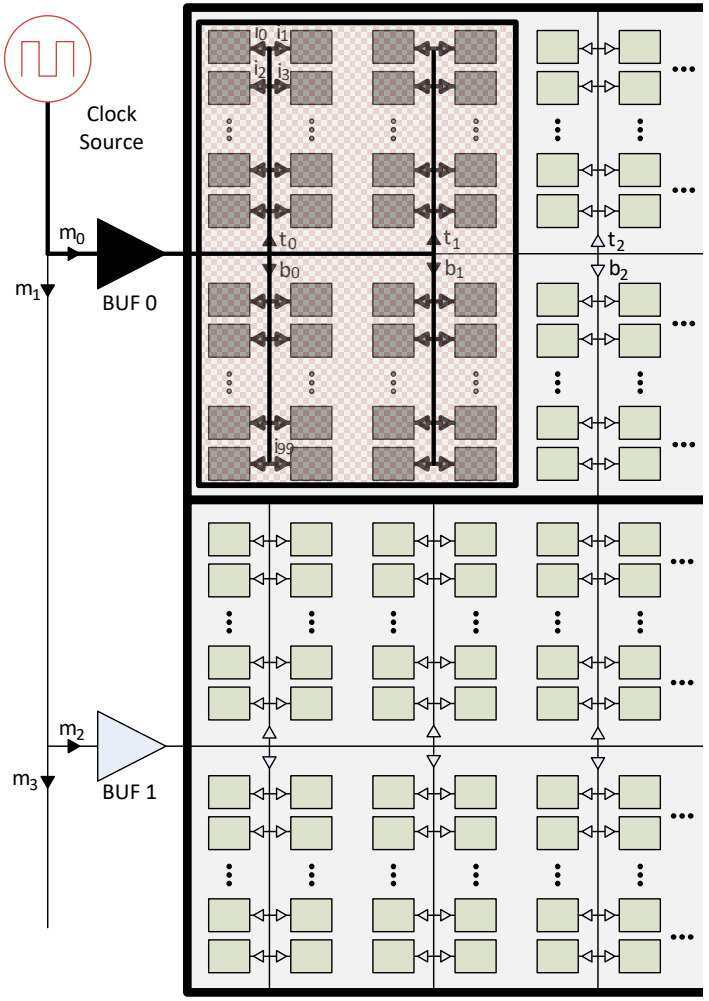
Figure 8.1 gives an illustration of the proposed clock routing process. In (a) the chip area is shown with the task yet to be configured. Sections of two clock regions have been shown containing buffers, clock nets and PIPs and the potential paths to synchronous elements. Each synchronous element on the chip can be routed to the clock source by enabling the PIPs in its path. For example, to deliver clock net to the CLB in the top left corner of the chip, PIPs m_0, t_0 and i_0 as well as BUF 0 must be enabled.



(a) FPGA Area before Task Placement



(b) Task Configured on Chip



(c) Clock Net Completely Routed

Figure 8.1: Runtime Clock Routing Process

A single clock buffer has been shown for each clock region. Also, a single clock net path is shown for each column and synchronous element. This is for illustration purpose only, the actual number/arrangement of clock buffers and nets in the 7 series was discussed in section 2.3.1, and a more detailed illustration shown in Figure *2.5*. More details are also given later in this chapter. In addition, the routing of PIPs and nets within the processing elements (all represented with CLBs) have been omitted for simplicity.

In Figure 8.1 (b), a task is shown configured on the chip, but the clock nets are yet to be completely routed to it. The task occupies 4 columns on the chip area within one row. During the design-time synthesis and implementation of the task, clock nets are routed to all used synchronous elements automatically. However, to make it possible for different clock buffers and nets to be routable to tasks, as well as to ease task relocation, the PIPs in the HROW (e.g. t_0 and b_0) are cleared in the task's partial bitstream at design time so that the desired buffers and nets can be routed to the task in runtime. Hence, after the task configuration shown, PIPs $i_0 - i_{99}$ are already enabled as required while t_0, t_1, b_0 and b_1 are disabled. Finally, Figure 8.1 (c) shows the clock net completely routed. The PIPs t_0, t_1, b_1 and b_2 are enabled to completely route the clock net to the task in runtime after its placement. The clock buffer (BUF 0) is also enabled at this stage. As shown in Figure 8.1 (a) PIPs $m_0 - m_3$ are enabled during the design phase of the application as they are in the static part.

As stated in chapter 2, the major clocking resources present in a clock region of a 7 series FPGA consists mainly of clock buffers, clock nets and PIPs. The arrangement and types of clock buffers, nets and PIPs in FPGAs provides a unique opportunity to address the challenge of clocking management in reconfigurable computing. It is possible to switch from one network to another on the fly, or even route new nets completely to a reconfigurable module. When a task's location is changed in runtime, it is challenging to deliver and maintain clock signals in an efficient way to applications. Three important challenges that need to be addressed include:

i) Ensuring that the right clock frequency is delivered to a task in runtime. This is important as different tasks allocated to the same clock region might require different operating clock frequencies

ii) Actual (re)routing of the clock signals to tasks. This might be done through the configuration layer by changing the states of the bits which control PIPs on clock routes to a task. These bits need to be identified by reverse engineering experiments as their locations are not disclosed by manufacturers.

iii) Power saving considerations. Given that clock buffer primitives must be in a static part of the design ready to be connected to tasks [62], adequate measures are required to minimize their power consumption.

In the next two sub-sections, details of the proposed clock network delivery architecture to address these challenges on Xilinx FPGAs are presented. This involves identifying the location of the bits in the configuration bitstream, an information not provided by Xilinx. The location of essential clock controlling configuration bits are determined by experimenting with various design variants.

The proposed clock delivery architecture involves instantiating all potentially useable clock buffers in the static part of the design. This is necessitated by the fact that clock buffers cannot be included in the reconfigurable part of a design on most FPGAs, including the 7 series. Clock signals are fed to the buffers' inputs, but their outputs are left unconnected. In runtime, their outputs are then *routed* to a reconfigurable module after its configuration by enabling appropriate bits in the configuration memory to activate corresponding PIPs via the configuration layer. The static part of the design also includes a configuration controller (CC) and the PMS. These manage the execution of the reconfigurable modules placed on the reconfigurable part of the design on demand. The CC is used for task configuration as well as writing bit locations to route clock nets while the PMS determines a location for a task on the chip.

Clock buffers which can be instantiated for runtime clock routing include all the BUFHs, BUFRs and BUFMRs in a clock region. These buffers are fed by the

BUFGs or the clock capable inputs of the chip. There are 12 BUFHs, 4 BUFRs and 2 BUFMRs in each clock region of a Xilinx 7 series FPGA. The BUFMR cannot drive logic directly; they must be routed through another buffer such as the BUFR. There are 16 dedicated clock nets which can be used to connect the clock buffers to synchronous elements such as flip flips, BRAMs and DSPs. These run across all columns in the clock region. They are routed all the way to each synchronous element.

Outputs of clock buffers cannot be left unconnected during the synthesis and implementation phase of a design. Tool optimization would remove the instantiated buffers, and if optimization is turned off (or a DONT_TOUCH attribute is set on them), *bitGen* would report "*partial antennae*" error and would not run. Hence, to ensure that the synthesis tool does not optimize the buffers, dummy reconfigurable modules can be included in the floor plan of static module, which are driven by the instantiated clock buffers. These modules are blanked-out immediately after the initial configuration of the static part. To ensure that a consistent architecture is adapted for connecting the buffers and nets in all clock regions, the BUFH are initially connected to the lower 12 of the 16 nets in a clock region. The upper 4 nets are driven by the BUFRs.

The choice of which net to route to which task is made in runtime according to the clocking requirement of each task. For example, a task placed in a clock region where it requires a different clock frequency than that present there would be connected to a net driven by a BUFR since BUFRs have clock division capability. Similarly, a task which extends to the adjacent clock region can be routed to a net driven by a BUFH. The routing is achieved by editing configuration bits that control PIPs in the path of the chosen nets to FFs, BRAMs and DSPs. The location of synchronous elements can be floor-planned to limit the number of bits to route to a task. However, the intersection of the HROW nets and the columns containing the synchronous elements must be routed via PIPs. A set of PIPs can potentially switch a net vertically to deliver any 12 of the 16 clock nets in the HROW to the synchronous elements in a column. Six nets enter a column from the HROW for each column

which are connected by 6 PIPs (TOP0, TOP1, … TOP5). These deliver clock signals to the upper half of the column and 6 to the lower half (BOT0, BOT1, … BOT5).

8.1.1 Selecting the Right clock Frequency for a Task

To select an appropriate net and buffer for a task, the frequency requirement of the task is considered. For tasks with special clock frequency requirement, a net driven by a BUFR is chosen to feed the task. The output frequency of the BUFR can be divided by any integer between 1 and 8 by writing specific 4-bit values to specific locations in the configuration memory. Table 8.1 shows the bit positions in the configuration memory used to divide the clock frequency, and Table *8.2* shows the values to be written for each division factor. The bits positions in Table 8.1 refer to the 50th word of frame address Minor 33 of the IOB column type. the subscript r in the table refer to the row of the device in which the BUFR is located while c denotes the specific index of the buffer.

Table 8.1: Bit Positions for BUFR Clock Frequency Division Factor

BUFR	***Bit Positions***
X_rY_c	18 – 21
X_rY_{c+1}	14 – 17
X_rY_{c+2}	23 – 26
X_rY_{c+3}	27 - 30

Table 8.2: Clock Division Factors and Corresponding Values

Clock Division Factor	***Value to be written to bit positions (0x)***
1	8
2	9
3	A
4	B
5	C
6	D
7	E
8	F

As an example, consider a VGA controller tasks which requires 25MHz for correct operation. When placed in a clock region on the Xilinx's basys3 board running at 100MHz, a value 0xB would be written to the bit location of one of the four BUFRs to achieve a frequency division of a factor of 4 and then it is routed to the task. For tasks where the general frequencies available on the chip would suffice, a BUFH is normally chosen, reserving the BUFRs for tasks requiring clock division. It is also important that tasks requiring frequency division must either be contained in a single clock region, or driven by multiple BUFRs (one for each clock region) connected to a BUFMR and must have a maximum height of 3 clock regions.

8.1.2 Routing a Clock Net to a Task

After choosing a net, driven by the appropriate clock buffer based on the requirement of a task, the chosen net is connected to the clocking point(s) of the task. During the floor-planning and implementation of a task, the sequential elements are constrained to pre-determined locations and routed to clock points in HROW. It is recommended that all timing constraints be addressed at design time. In runtime, an appropriate active clock net must be selected and routed. This is done via the configuration layer by activating the set of PIPs needed to route the clock signal from the buffer to the net in the HROW of each the columns of the FPGA occupied by the task. The

columns in the Xilinx FPGA are organized in pairs, classified as left (L) and right (R) columns. An L-R pair share common routing resources, and thus the bits to enable/disable PIPs are located in either the 'L' or 'R' column. In addition, any of the 16 nets can be routed to a column via a set of *Routes*. Table 8.3 shows the set of bits in the configuration memory to be activated to route any of the 16 clock nets in the HROW to a column through the *BOT0* PIP. Because of the similarity in the bit positions between 'even' and 'odd' nets, the rows are organized in pairs as shown.

It can be seen from the table that each net is routed via BOT0 by activating 3 bits in the configuration memory. However, two of these are shared by groups of nets, with only one being unique to each net. One of the shared bits may be described as a 'group selection bit' as it determines the 'group', G to which the net belongs. The 16 nets may be grouped into 4: Net 0 – Net 3 (G1) which are selected by writing a '1' to bit position 14 of frame minor 1 (M1), Nets 5 – 7 (G2) are controlled by bit position 15 of minor 0 (M0). Similarly, Nets 8 – 11 (G3) and Nets 12 – 15 (G4) are controlled by bit 15 of M1 and bit 16 of M0 respectively.

Table 8.3: Bit Position and Frame Address Minors of PIPs via BOT0

	Bit Position in Word 50 of Frame					
Net	**M0**	**M1**	**M2**	**M3**	**M4**	**M5**
0, 2	14, 22	14, 14		—	14, 15	
1, 3	—	14, 14; 19, 22		14, 15	—	
4, 6	15, 15; 23, 31	—		—	14, 15	
5, 7	15, 15	23, 31		14, 15	—	
8, 10	14, 22	15, 15		—	14, 15	
9, 11	—	15, 15; 19, 22		14, 15	—	
12, 14	16, 16; 23, 31	—		—	14, 15	
13, 15	16, 16	23, 31		14, 15	—	

The second shared bit may be described as 'regular distance bit', **D-bit**. The D-bit is shared by every fourth net, such that Nets 0, 4, 8 and 12 (D1) are controlled by bit position 14, and Nets 2, 6, 10 and 14 (D2) controlled by bit 15 of minor 4 (M4). The odd-numbered nets are controlled by the same bit positions in minor 3 (M3). That is, Nets 1, 5, 9 and 13 (D3), and nets 3, 7, 11 and 15 (D4) are respectively controlled by bit 14 and 15 of M3.

The third bit is unique for each net. Even numbered nets between 0 and 7, i.e. Net 0, 2, 4 and 6 are controlled by bit positions 14, 22, 23 and 31 of M0, while odd numbered nets are controlled by positions 19, 22, 23 and 31 in frame M1. Nets 8 to 16 are controlled by the same bit positions in the same minors, but of the adjacent column. Recall that an L-R pair of columns share a routing resource in the 7 series.

To use another route such as BOT1, … BOT5 or TOP1 to TOP5, the bit positions are organized in a similar fashion to that of BOT0 shown in Table *8.3*. The unique bits remain the same for all routes, both in position and frame address, to that of BOT0 described above. The location of the two other shared bits – the **G-bit** and the **D-bit** relating to each net for all other routes except BOT0 are shown in Table 8.4 and Table 8.5 respectively.

Table 8.4: Bit Position for G- bit of Clock Net in HROW

	G1		**G2**		**G3**		**G4**	
	Bit	**M**	**Bit**	**M**	**Bit**	**M**	**Bit**	**M**
BOT1	16	3	16	5	16	4	28	2
BOT2	18	0	17	1	17	0	16	1
BOT3	17	2	17	4	17	5	17	3
BOT4	20	0	21	1	21	0	21	1
BOT5	22	3	22	5	22	4	22	2
TOP0	30	1	29	0	29	0	29	1
TOP1	29	2	29	4	29	5	29	3
TOP2	26	0	28	1	28	0	28	1
TOP4	25	1	24	0	24	1	24	0
TOP5	23	2	23	4	23	5	23	3

Table 8.5: Bit Position for D- bit of Clock Net in HROW

	D1		**D2**		**D3**		**D4**	
Route	**Bit**	**M**	**Bit**	**M**	**Bit**	**M**	**Bit**	**M**
BOT1	14	2	14	5	15	2	15	5
BOT2	19	5	19	2	18	5	18	2
BOT3	19	3	19	4	18	3	18	4
BOT4	20	4	20	3	21	4	21	3
BOT5	20	2	20	5	21	2	21	5
TOP0	31	5	31	2	30	5	30	2
TOP1	31	3	31	4	30	3	30	4
TOP2	26	4	26	3	27	4	27	3
TOP3	26	2	26	5	27	2	27	5
TOP4	25	5	25	2	24	5	24	2
TOP5	25	3	25	4	24	3	24	4

It is worth noting that only 7 bits per column are required to be modified to route a clock signal to a task. These are: 1 bit to turn-on the buffer routed to the desired net, 3 bits to route the net to a PIP feeding the upper half of the column and 3 bits to feed its lower part. This is a significant improvement compared to the 98 bits required by the technique in [62] especially as the bits are located in different configuration frames. However, to achieve this, additional design-time steps are needed.

8.1.3 Low Power Considerations

With the architecture described above, it is noted that the instantiation of the clock buffers would lead to increased power consumption of the system. Hence, it is important to turn off the clock buffers which are not currently required. This can be done via the configuration layer. Table 8.6 shows the bits positions that control the enabling and disabling of the BUFHs. Writing a ‘0’ to the respective bit positions turns off the buffer while writing a ‘1’ to that location turns it on. The BUFHs are

located in the middle column of the device (e.g. column 23 for the basys3 board). The word and bit position shown are in frame MINOR = 26. It can be observed that only the even-numbered buffers are shown, the location for the odd-number buffers are the same as that of the even ones, except that the frame MINOR = 28 for the odd numbered buffers.

Table 8.6: Enable/Disable Bit Position for BUFHs in a Row

	C = 0		**C = 1**	
	Word	**Bit**	**Word**	**Bit**
XcYr+0	48	19	47	3
XcYr+2	49	3	47	19
XcYr+4	49	19	48	3
XcYr+6	51	3	52	19
XcYr+8	51	19	53	3
XcYr+10	52	3	53	19

There are 2 BUFMR per clock region. The ON/OFF bit of the first of these is in bit 28 of word 5, minor 27 of the IOB frame type. The ON/OFF bit for the second is found at the same location of minor 28. It is worth noting that the BUFRs cannot be switched on/off via the configuration layer. Thus, to control them, the buffer driving a BUFR is turned off. BUFRs are normally driven by BUFMR.

8.2 Reliability Considerations

It can be observed that the entire clock delivery technique presented in the first section of this chapter is hinged on changing specific bits in a configuration frame in runtime. In addition to delivering the right clock frequency to a task placed in runtime or route clock signal to tasks, a variety of techniques used by ROS depend on editing the content of the configuration bitstream in runtime. Examples include

the following: Runtime bitstream modification has also proposed for the relocation of circuits to non-matching locations on the FPGA [75]. Establishing communication with tasks whose location on the FPGA have changed in runtime have also been proposed to be done using techniques that involve runtime bitstream editing [61] [122]. These examples involve editing the bit values inside a configuration frame.

However, there is a major reliability concern with changing bits in a frame of a configuration bitstream in runtime especially for safety-critical applications. SRAM-based FPGA CMEM are *volatile*, and the bits stored in them could be flipped due to undesired effects such as radiation and extreme temperatures [52]. To mitigate the effects of unwanted bit flips, each configuration frame in the bitstream of Xilinx FPGAs is protected by a Frame Error Correcting Code (Frame ECC). A Frame ECC is a set of bits representing a value computed based on the parity of the data in the frame and stored as part of the frame. It is monitored for changes in the content of the frame and can be used to correct single bit errors and detect multiple bit errors [123]. The bit-flip detection and correction technique (usually implemented via the SEM IP [56] or custom scrubbing techniques [27]) does not distinguish between intentional changes in bits and soft errors due to radiation or extreme temperatures. This is illustrated in Figure 8.2.

One means of addressing this challenge is to re-compute the Frame ECC values each time a bit is intentionally changed. In this way the soft error mitigation technique in place would continue to function normally so that the design does not lose the protection offered by the Frame ECC. It is worth noting that the Frame ECC values are generated as part of the undisclosed *bitgen* process when Xilinx design tools are used to implement a design. Indeed, the publicly available technical information on the Frame ECC for the Xilinx 7 series FPGA is limited to the number of bits reserved for the frame ECC, their location in the frame of a configuration bitstream and how custom Xilinx IPs uses these values to report bit flips. A clear information as to how their values are generated is not provided.

Hence the strategy proposed in this work is to re-compute the Frame ECC bits and include updated values in a frame after bit editing, just before configuration of the

updated frame. Soft errors in Xilinx FPGAs are monitored using both Frame ECC (which monitors bits flips in a frame) and CRC values (which monitor bit flips in the entire configuration data). However, this section focuses only on the use of Frame ECC bits to detect errors since it is uncommon that errors not caught by the ECC mechanism are detected by CRC [56].

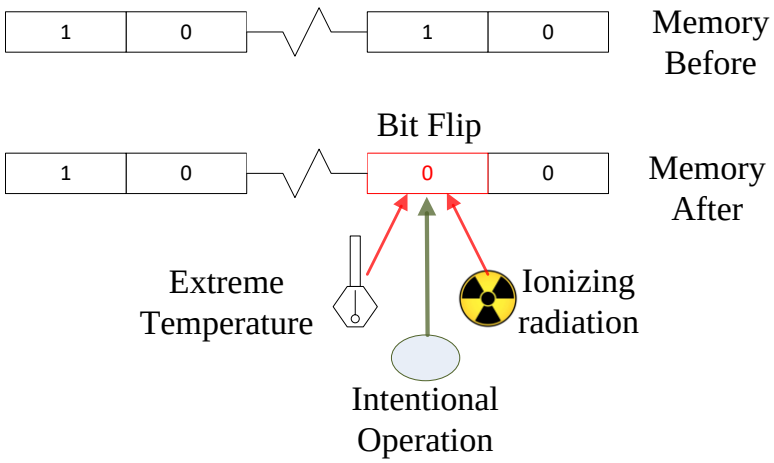


Figure 8.2: Bit flips in Memories of SRAM-based FPGA

In this section, an efficient implementation of the mechanism for recomputing the Frame ECC of Xilinx FPGA configuration bitstream in runtime is presented. Without loss of generality, the implemented algorithm will target Xilinx 7 series FPGAs, but is easily extensible to other FPGA architectures as well. It is worth noting that since the complete routine for computing Frame ECC in Xilinx FPGAs is not completely disclosed, we first present our findings on how to completely re-compute the Frame ECC before moving on to its implementation.

8.2.1 Frame ECC Re-computation Routine

As stated in the last section, it is important to re-compute the Frame ECC values each time a bit is intentionally changed, so that these changes are not interpreted as soft errors and overturned by the SEM IP or similar soft error mitigation mechanisms. One

alternative that can be used to avoid intended bit edits being overwritten is to disable the SEM mechanism. However, this would make the entire design lose the protection offered by the frame ECC, and thus constituent a significant reliability concern in critical applications. Xilinx offers only very limited details as to how the frame ECC in their devices are computed. Generally, the Frame ECC values are computed as part of an undisclosed bitgen step done by the design tools such as Vivado. For the 7 series FPGA, the information provided is limited to the location of the frame ECC and the number of bits reserved for its value. Basic idea of computing ECC for a block of data such as a configuration frame may be seen in [123]. In the following, the details of our findings as to how to exactly re-compute the values of the Frame ECC in runtime is presented.

The computation of the frame ECC is an iterative process carried out on all the words in the frame, where the ECC bits computed for a word are XORed with that for the next word until all the words in the frame have been used. For the Xilinx FPGAs, configuration frames are organized in 32-bit words which are indexed by an integer number, I. Each bit in a frame can be referenced by using the relation: $32.I + k$, where k ranges from 0 to 31 for each I. The range of the values of I is determined by the number of words in a frame of the configuration bitstream and the number of powers of 2 in the range. Values of I corresponding to powers are 2 are skipped as these are reserved for the frame ECC indexes [123]. It was observed that lower values of I are avoided since powers of 2 occur more frequently in that range. For Xilinx virtex 4 FPGA which have 41 words in a configuration frame, the ECC is computed with I ranging from 22 to 63. For the 7 series with 101 words in a frame, it was found that it ranges from an initial value, $I_i = 25$ to terminal value, $I_t = 127$. In general, we found that for an FPGA series, the value of I_t is obtained by aligning the last bit in the frame $(32.I_t + 31)$ to the position $2^{(n-1)} - 1$, where n is the number of bits required to store the frame ECC values. Thus, for the 7 series since 13 bits are reserved for the frame ECC values [56], the value of I_t would computed to 127. Consequently, to account for 101 words, I_s evaluates to 25, with 64 and 32 skipped since there are powers of 2.

Finally, an ECC polynomial defines which bits of each of the words in a frame are XORed at every stage of the iterative process. Equation (8.1) – (8.6) define the polynomial for the computation of each stage of ECC for a 32-bit word. Each bit of the ECC value (bit 1 to 12) is computed by XORing selected bits of the current word. The selected bits are determined using (8.1) – (8.6). Each equation defines the set of bit positions of the word which participates in the computation of that specific ECC bit. For example, for the computation of ECC bit 1, *E(1),* only the odd bit positions in the current word are selected. Thus, all odd bits of the 32-bit word are XORed together to determine the current value of *E(1).* Similarly, for the computation of bit 3 of the ECC bits, *E(3),* bits 4 to 7, 12 to 15, 20 to 23 and 28 to 31 of the current word are XORed together. The computation of the other ECC bits follow similar pattern, using the bit positions dictated by the corresponding equation.

As can be seen from the equations, the computation of ECC bits 1 to 5 is dependent only on the value of the current word and not on its position in the frame, I. This is different for bits 6 to *X* where both the value of the word and its relative location in the frame contribute to determining the value of the ECC bit. For example, to compute bit 6 of the ECC for a word, all the bits of the word are XORed if the least significant bit of the word's location is 1, otherwise that ECC bit is simply 0. In (6), j refers to the bit index of the current word's location in the frame. For example, for $I = 25\ (= 011001_b)$, $I_{j=0} = 1$, $I_{j=1} = 0$, etc. In addition, *X* is determined by the number of bits reserved for the final ECC value. For Xilinx's Virtex 4 and 7 series FPGA, X = 5 and 6 respectively. All bits in each word are XORed to determine E(0) for that word, thus bit 0 of the ECC is the parity of the entire frame.

An iteration step consists in computing all the bits of ECC in a word. These bits are XORed with the values obtained from the previous word of the frame to get the current partial ECC. The process is repeated until all the words in the frame have been considered. It is worth mentioning that process described above is applied to all words in the frame. However, for word 50 in which the ECC bits are located. i.e., for $I = 50$, k iterates from 13 to 31, omitting the location of the ECC bits.

$$\boldsymbol{E(1) = \{k, \forall\, k \neq even, k \leq 31\}} \qquad (8.1)$$

$$\boldsymbol{E(2) = \left\{\frac{1}{2}\left(4(k+1) + (-1)^{k+2} - 1\right), 0 \leq k \leq 15\right\}} \qquad (8.2)$$

$$\boldsymbol{E(3) = \left\{\frac{1}{2}[4(k+1) - (1-i)(-i)^{n+1} - (1+i)(i)^{n+1} + (-1)^{n+2} + 1]: 0 \leq k \leq 15, i^2 = -1\right\}} \qquad (8.3)$$

$$\boldsymbol{E(4) = \{k: 7 \leq k \leq 15, 24 \leq k \leq 31\}} \qquad (8.4)$$

$$\boldsymbol{E(5) = \{k: 16 \leq k \leq 31\}} \qquad (8.5)$$

$$\boldsymbol{E(6+j) = \begin{cases} k, if\ I_j = 1 \\ \emptyset, if\ I_j = 0 \end{cases} 0 \leq k \leq 31, 0 \leq j \leq X} \qquad (8.6)$$

8.2.2 Implementation Case Study

To test the performance of the proposed Frame ECC re-computation scheme, a design consisting of the frame ECC re-computation scheme, a custom configuration controller and a case study application were implemented. Details of each of these is give below. In addition, Xilinx Integrated Logic Analyzer (ILA) as well as the Virtual Input Output (VIO) probes were included in the design. The ILA was used to observe internal signals of Frame ECC primitive, the Frame ECC re-computation engine and the configuration controller. The VIO was used to send commands to the configuration controller and the Frame ECC re-computation engine such as to initiate configuration and read-back operations on the configuration controller and enable the Frame ECC engine.

i) Frame ECC Re-computation Engine

The frame ECC re-computation routine described in above was implemented on a Xilinx xc7a35tcpg236-1 chip using Vivado 15.1 design tool. Table 8.7 shows the resource overhead of the implementation in terms of FPGA resources. The implementation of runtime Frame ECC re-computation has a latency of 104 clock cycles. A BRAM is used to buffer the frame data which contains the configuration bits to be edited (e.g. the clock division bits). The buffer was configured to be 96-bit wide and 34 words deep. It is capable of holding a frame of configuration bitstream (101 32-bit words) at a time. Its output feeds into 3 instances of the ECC re-computation engine which requires 3 clock cycles to obtain the partial ECC for each word, thus obtaining a partial ECC for 3 words in 3 clock cycles. Two additional clock cycles are used to write the final ECC values to the buffer of the configuration controller. The time overhead of the Frame ECC (re)computation routine does not impact the timing behaviour of the task configuration as the re-computation of ECC can be done concurrently with configuration as explained in the next section.

Table 8.7: Resource Utilization of Frame ECC Re-computation Routine

Resource	**Used**	**Available**	**% Utilization**
FF	364	41600	0.875
LUT	193	20800	1.159
Bram 18kb	3	150	2.000

ii) Configuration Controller

A task Configuration controller described in [27] was also instantiated in the design. In addition to task configuration, that controller implements an optimized version of soft error mitigation strategy using the Frame_ECC primitive. As mentioned in chapter 2, the basic principle of soft error mitigation depends on monitoring the value of ECC in the CMEM. The difference between the strategy in [27] and the SEM IP

[56] is that the former limits the region of CMEM monitored for error to only those parts of the chip with actively computing circuits. Technical details of the soft error mitigation of the controller can be found in [27].

The major operations and their associated latencies of the controller used are shown in Table 8.8. It is worth mentioning that the timing characteristics of the controller and that of the Frame ECC re-computation engine is such that no delay is introduced to task configuration by the Frame ECC re-computation engine. As shown in the table, the configuration controller has a minimum configuration latency of a frame to be 166 clock cycles. This consists of 65 clock cycles overhead at the start of a configuration and 101 cycles for writing the 101 words in a frame. Since the ECC word is located at word 50, a total of 115 clock cycles is spent by the configuration controller before getting to the Frame ECC word. Thus, initiating the re-computation of Frame ECC at the same time as the configuration process, no additional clock cycle is incurred in the configuration of tasks. This is illustrated in Figure 8.3. As shown, the re-computed Frame ECC value is available 10 clock cycles before it is required.

Table 8.8: Time Overheads for the Operations of the Configuration controller at A Frequency of 100 Mhz [27]

Operation	**Minimum Time (µs)**	**Time for N Frames and M Replicas (µs)**
Readback	2.37	$1.36 + 1.01N$
Configuration (non-BRAM frame)	1.66	$0.27 + 1.28N + 0.11$M
Configuration (BRAM frame)	1.74	$0.19 + 1.36N + 0.19M$
Blanking	1.56	$1.39 + 0.17N$
Register Read	0.29	0.29
Custom Write	0.23	$0.23+ 1.01N$
Operation Abort	0.05	0.05
SEM Scan	2.37	$1.36 + 1.01N$
SEM Correction (Repair)	1.66	1.66
SEM Correction (Replace)	2.68	2.68

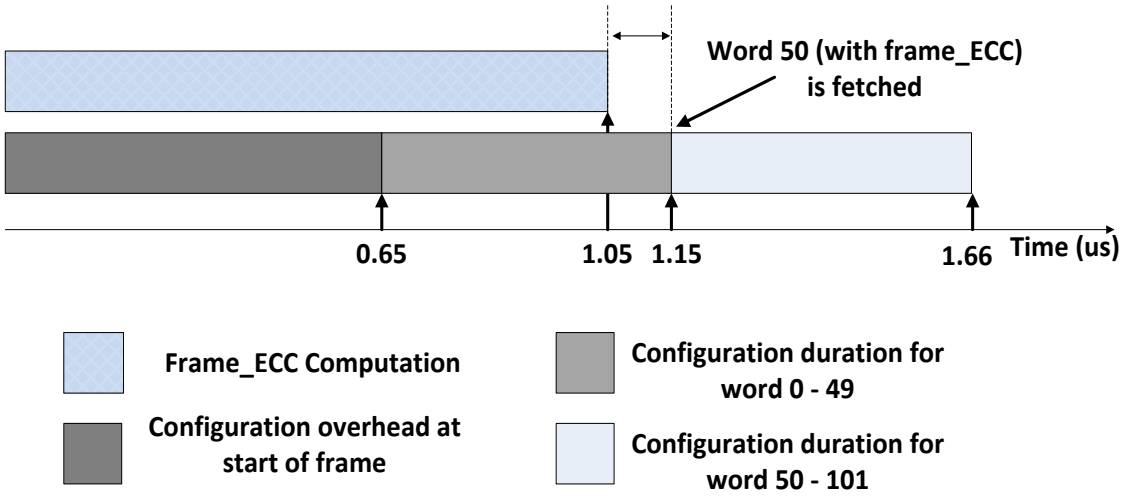


Figure 8.3: Timing Characteristics of Configuration and Frame_ECC re-computation Controllers

iii) Case Study Scenario: Online Clock Frequency Control.

A simple 4-bit counter whose outputs can be easily observed on LEDs on the Xilinx xc7a35tcpg236-1 chip was used for the test. The counter increments every second. The aim in this experiment was to change the clock frequency delivered to the counter in runtime using bit editing and observe the change in its count rate. At the same time, the output of the soft error mitigation mechanism would be observed to see if any error is detected. As a control, the Frame ECC re-computation routine will then be disabled and the same changes will be attempted and then the result of scenarios would be compared.

The output frequency of the BUFR in the Xilinx 7 series FPGA can be divided by any integer between 1 and 8 by writing specific 4-bit values to specific locations in the configuration memory. Therefore, a BUFR was instantiated and its output clock signal was routed to the counter as its clock source. The clock input to the BUFR was routed via a BUFMR so that the clock can disabled and enabled in runtime as BUFRs do not have clock enable pins. The location of the clock division bits for BUFRs in the configuration bitstream are shown in Table 8.1 while the division factors are

shown in Table *8.2*. The first BUFMR in a clock region is enabled by writing a '1' to bit position 28 of frame Minor 27 of word 50 of the IOB column type. The second is BURMR is enabled by similar location of frame Minor 28. The locations of the buffers were constrained to the first BUFMR (BUFMRCE_X0Y0) and the second BUFR (BUFR_X0Y1) of the upper left clock region of the chip in this experiments using constraints in XDC file.

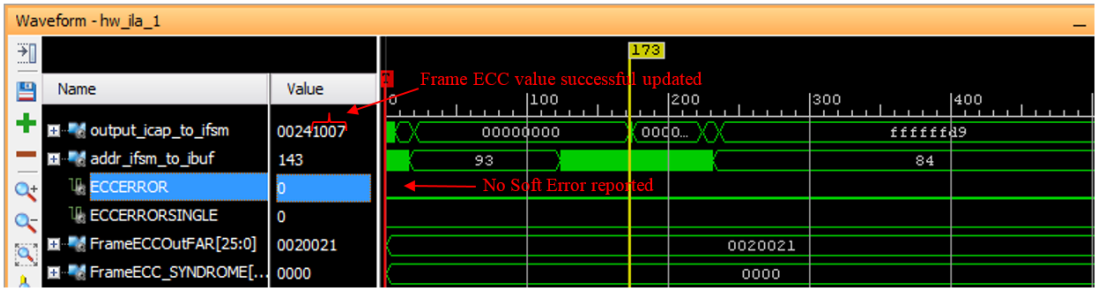
8.2.3 Result and Discussion

The initial design was programmed on a basys3 FPGA board running at 100 MHz with the counter incrementing every second. During the normal operation of the system a frequency division process was initiated. Frame minor 33 of column 0 row 0 in the bottom part of the design of block type IOB was read back. This was done by issuing a readback command to the configuration controller using the VIO. The bitstream read back (101 words) was saved in a buffer. As shown in *Table 8.1* and *Table 8.2*, the clock division bits are in word 50, bits [14:17]. The value of the bits was observed to be **0x8.** The value of the clock division bits was updated to **0x9** in the buffer in accordance with *Table 8.2*, aiming to divide the frequency of the clock by 2. In addition, the frame ECC bits were reset to 0s in the buffer. At this point, the design was not affected.

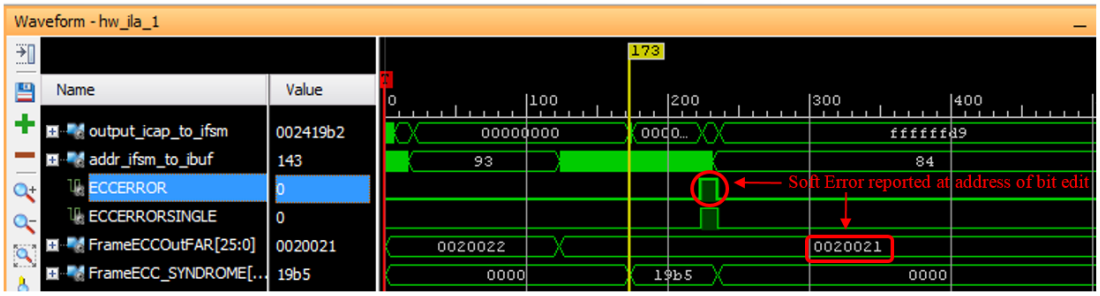
Thereafter, the Frame ECC re-computation engine was enabled and the configuration of the frame to the CMEM was also enabled. After the configuration was done, it was observed that the rate of the counter was reduced to 0.5 seconds. The output of the soft error monitoring scheme (Frame ECC primitive) was observed after a readback operation was performed. No error was reported. This is shown in Figure *8.4* (a). As shown in the figure, the ECC value (shown on bit 12:0 of the signal output_icap_to_fsm) in the frame was updated (from "0x19b2" to "0x1007"), and the signal "ECCERROR" remained at '0' to indicate no soft error. In addition, an *error* was injected in the CMEM by writing '0' to the Enable/Disable bit of the BUFMR (i.e. bit 28 of Minor 27 of column 0) without enabling the re-computation of the

Frame ECC. No effect was noticed on the counter – the design continued to function normally at the rate of 0.5 seconds.

Next, the FPGA board was power-cycled and reprogrammed with the original bitstream of the design. The same steps as above were repeated except that the Frame ECC re-computation engine was not enabled. That is, the frame containing the clock division factor was readback into the buffer and the clock division parameter was updated to 0x9. Then the frame was written to the configuration memory. After configuration was done, it was observed that the counter continued to increment at the initial rate of 1 second. It was also observed that the frame ECC primitive reported a soft error. This is shown in Figure *8.4* (b). However, the soft error mitigation routine over-wrote the entire frame with the golden configuration bitstream. Thus, without re-computing the ECC values, runtime bit editing can be classified as a soft error and over-ridden. Also, Writing '0' to the Enable/Disable bit of the BUFMR also did not have any effect on the counter as it was also corrected by the soft error mitigation mechanism.



(a)



(b)

Figure 8.4: Waveform of Frame ECC Primitive

Finally, the experiment was repeated with both the Frame ECC re-computation routine and the SEM routine disabled. This time after changing the frequency of the clock, the counter rate was found to decrease to 0.5 seconds as expected. However, writing a '0' to the Enable/Disable bit of the BUFMR make the counter completely freeze and stopped incrementing.

Table 8.9: Summary of Features in Designs with and without SEM and Frame ECC Re-computation Engines

Features	**Design Only**	**Design + SEM Controller**	**Design + SEM ad Frame ECC Controllers**
Soft errors recovery	☒	☑	☑
Support for runtime configuration bit editing	☑	☒	☑
Soft error recovery and runtime configuration bit editing	☒	☒	☑

Table *8.9* summarises the features of designs depending on the presence of SEM and runtime frame ECC re-computation mechanism. Comparing the results of the three experiments above, and assuming that editing the frequency of the counter was intended by a user, while disabling the BUFMR was a soft error, it can be concluded that re-computing the frame ECC as in the first scenario make it possible to control a design protected by soft error mitigation techniques as desired. It allows users to make intended runtime bit editing while the design is still robust against un-intended bit flips. In the first scenario, the frequency of the counter was successfully updated while the error injected by disabling the BUFMR was detected and corrected. In the second scenario, the soft error mitigation technique interpreted all bit edits as soft errors and reversed them because the Frame ECC was not re-computed. In the third scenario when the soft error mitigation mechanism was disabled, the design responds to all changes to the content of its CMEM. The frequency of the design was successfully updated but the design also failed due to a soft error on an essential bit.

8.3 Chapter Conclusion

In this chapter, we have presented an efficient runtime mechanism of clock delivery to circuits placed in runtime without jeopardizing the reliability of the system. The technique depends on identifying key clock signal routing bits in the configuration bitstream of the class of FPGA. These are controlled in runtime to route a clock signal to circuits. To avoid losing the error monitoring offered by the frame ECC, the value of the ECC is recomputed for any frame in which bits are changed to route a clock signal. The distinctive features of the proposed technique are that it is aimed at minimizing the number of bits changed in runtime and recalculating the ECC for the affected frames. By using different design variants, the location of the essential clock routing and buffer enable/disable bits in Xilinx 7 series FPGA were determined. The frame ECC computing scheme for the same class of FPGAs was implemented. Based on these, it is possible to route a clock signal in runtime by bit editing without losing the protection offered by the Frame ECC.

The results show that for clock routing, only 7 bits per column are required to be modified to route a clock signal to a task as against the 98 bits required by similar approaches. The implementation of frame ECC re-computation controller occupies only 364 LUT, 193 flip flops and 3 18-Kb BRAM on the Xilinx xc7a35tcpg236-1 chip and has a latency of only 104 clock cycles for each frame. It was also shown that its latency does not introduce any delay in configuration procedure.

A major limitation of the proposed approach is its dependence on a family of FPGA. The format of the configuration bitstream changes from one FPGA family to another. Hence, the reverse engineering experiments will have to be repeated to apply the technique to another family of FPGA. The content of this chapter is included in the following publication:

- G. Enemali, A. Adetomi, and T. Arslan, "Efficient Runtime Frame ECC Recomputation for Reliable Task Execution on Xilinx FPGAs ", in 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2018. pp. 59- 65

Chapter 9: Conclusion and Future Work

This thesis has presented techniques towards the development of future high-performance, fault-tolerant electronic systems for hostile environments such as nuclear plants and outer space within the constraints of cost, power and flexibility. The placement management system presented in this thesis is the design and implementation of several techniques to achieve efficient runtime placements in COTS FPGAs which have a high degree of heterogeneity. Techniques relating to optimizing the utilization of the FPGA area, managing its relatively large configuration overhead, relocating tasks on its heterogeneous area and managing clock network routing to placed tasks were presented.

These techniques provide a means that enables future ROS to better harness the capabilities of COTS FPGA using DPR to achieve reliability and high performance. With the continual increase in the degree of heterogeneity of COTS FPGAs, runtime placement and task relocation which are techniques used by ROS to achieve high performance and reliability have become increasing challenging to implement. In addition, circumventing the relatively large reconfiguration overhead of COTS FPGA while managing fragmentation of the device area is an important requirement in ROS. Moreover, delivering clock networks to tasks after placement on the FPGA is also challenging. The proposed placement techniques address these challenges.

While many traditional placement techniques are based on ideal models which are not well suited to COTS FPGA platforms, the proposed techniques not only aim to have better performance but also to be practicable. The practicality of the proposed techniques is demonstrated by presenting its implementation details in chapter 7, thus addressing many design and implementation issues in runtime placement management on COTS FPGAs.

This chapter gives a summary of the research work presented in this thesis. It highlights the main results and draws conclusion from them, showing the potential

impacts and significance of the techniques. The limitations of the techniques presented are identified, and future works are also suggested.

9.1 Summary of Thesis

The first three chapters of this thesis presented background information on the unique attributes of COTS FPGAs – especially on the potentials provided by DPR and reviewed support tools and techniques developed to harness the huge potentials of DPR with a focus on runtime placement. The major contributions of the thesis are contained in chapters 4 to chapter 8. Chapter 4 presented a design-time optimization for reliability. The design flow presented is aimed at not only improving the maximum number of locations for each task on the chip, but also to achieve a fair distribution among all tasks which will share the chip area concurrently in runtime. The technique is based selecting implementation locations for tasks to minimize overlap in the potential placement locations of tasks occupying the FPGA area simultaneously. The offline placement quality optimization also aims to minimize the variance in the number of potential locations of each task and thus avoid a situation where some tasks have abundant potential placement locations and others have too little. A balanced distribution and minimized overlap of implementation location leads to an improvement in the number of placements in runtime for each task and increases the performance and fault tolerance of applications.

Chapter 4 also presented an architecture of a generic task wrapper based on memoization for achieving low power computation on FPGAs for tasks with low port width. The power optimization technique using memoization is applied to tasks to reduce their dynamic power consumption. The technique involves reusing the result of a previous computation when a request is made for computation with the same set of inputs that produced them. Thus, the process of re-computing the result for the input is avoided – together with its dynamic energy consumption. To achieve this, results of previous computations are remembered, leading to memory and logic overheads. Hence, it is imperative that these overheads of the memoization wrapper

are minimized. To achieve energy minimization, a place reservation technique is used which ensures that the search for previous results are done efficiently in few (and fixed) number of clock cycles. Space reservation technique also keeps the MISS rate low, leading to greater energy savings. However, to keep the memory overhead reasonable, the wrapper is only suitable for tasks with low port width. The chapter also includes a discussion of a communication wrapper for all tasks. The resource overheads of both the wrapper for dynamic power minimization and that for communication are added to the task's resource utilization before the optimization procedure for selecting implementation locations.

Unlike the design time techniques presented in Chapter 4, Chapter 5 gives two key techniques relating to the runtime phase of placement management for high performance and reliability. These are: efficient minimization of chip area fragmentation and efficient task reuse to reduce the amount reconfiguration engaged in by the configuration port. To minimize fragmentation on the chip, a fragmentation quantification technique suitable for use on heterogeneous FPGAs was proposed. The method of quantifying fragmentation aims to balance speed and accuracy such that it could be fast enough for runtime placement and yet produce accurate results. The fragmentation measure is based on the isolation of a task placement location from other tasks on the FPGA as well as the FPGA borders. A comparison of the proposed technique with others showed that its accuracy is better than those schemes with comparable computational overhead, while being comparable to others with higher computationally intensity. It was shown that since tasks' location on heterogeneous chips are constrained by their layout, the placement location of one task may not fall at the border of another even with good fragmentation quantification techniques. Hence, an expansion strategy, EUAS, was proposed. EUAS uses information on the dimension of the tasks to be placed to decide the amount of *expansion* during placement. Using simulations, it was shown that a lower task rejection ratio is obtained when EUAS was used.

Chapter 5 also presented a task reuse strategy to circumvent the reconfiguration of carefully selected tasks. The tasks to retain were decided using a novel

fragmentation-aware replacement policy – FAReP. The replacement policy selected tasks to be retained on the chip based on their reconfiguration overhead, frequency of reuse and the amount of fragmentation which their current location contribute to the chip area. Thus, in addition to preserving tasks with costly and frequent reconfigurations on the chip, FAReP offers some degree of defragmentation of the chip area during each task replacement. The chapter results showed that the number of reconfigurations circumvented using FAReP is greater than that of other task replacement policies. A reduction in the number of reconfigurations leads to greater availability of the configuration interface for other very key operations in critical applications such as soft error mitigation.

Chapter 6 addressed the challenge of task relocation on COTS FPGAs. The chapter first introduced the concept of DBR on FPGAs and explained the process involved in DBR. DBR is commonly achieved by either generating partial bitstreams for all potential locations of the task on the chip or by modifying the location dependent sections in its partial bitstream in runtime. The later has the advantage that fewer number of partial bitstreams are required to be managed in runtime. The chapter further identified a major limitation of DBR which is that the resource constituents of the original implementation location on which the partial bitstream was generated must match a destination location for most practical cases. However, COTS FPGAs have heterogeneous columns arranged in no particular order. In fact, even a single resource column type typically has *left* or *right* orientations further increasing the number of different resource columns on the chip and decreasing the chance of finding a location matching the original location of partial bitstream. This reduces the number of locations a task can be relocated to using DBR. Another limitation of DBR is that it cannot be applied to encrypted bitstreams when access to the location information of the bitstream is not available. The chapter thereafter proposed FBR strategy.

The chapter described the process of FBR which essentially involve transforming the logic represented by the task into a look-up-table or a block of memory. The advantage is that the LUT or memory block which replicates the functionality of the

original circuit can be relocated to locations on the chip which do not match the resource arrangement of the original implementation location of the task. However, the chapter also identified the limitation of FBR. FBR cannot be applied to tasks which are not referentially transparent and have huge memory overhead for applications with large ports. Therefore, the chapter proposes a merger of both DBR and FBR and showed that augmenting the later with the former would lead to a significant increase in the total number of task relocations that can be obtained on COTS FPGAs. Since relocation is a central technique used by ROS to achieve high performance and reliability, improving the amount of relocation obtainable on the chip is a huge potential.

Chapter 7 shows the practicality of the proposed techniques by describing the implementation of a prototype PMS which includes the techniques proposed in previous chapters. Low level implementation issues of the PMS were presented. The chapter also characterized the implementation of the PMS and reported its performance including timing and resource overheads. The performance results of the implementation were compared with a similar runtime task placement scheme. The comparison showed that the proposed PMS has more features and is more than 2 times faster than a comparable runtime placement system. Based on this, faster placement decisions can be made which reduces the chances of missed deadlines in runtime scenarios. In addition, new placement locations can be decided more quickly and hence reduce an application down-time in a case where a task need to be relocated due to occurrence of permanent faults on the FPGA.

Chapter 8 discusses clock network delivery to tasks after their placement in runtime. A method of runtime clock routing was presented that involve controlling configuration bits in the configuration memory to change the states of PIPs in the path of clock signal to tasks. The frequency of the clock is also adjusted in some cases to accommodate tasks of different clock frequencies in the same clock region. The chapter described an architecture that supports runtime clock routing by instantiating clock buffers in the static part of an application during its design phase. The chapter also presented the results of reverse engineering experiments to

determine the location of the essential configuration bits that need to be controlled in runtime to achieve clock network routing or frequency division operations.

In addition, chapter 8 also presented a technique of re-computing the frame ECC in the configuration bitstream after editing bits in runtime. The chapter presents the implementation of efficient frame ECC re-computation routine to address the challenge posed by editing configuration bits in runtime. COTS FPGAs based on SRAM configuration memory are susceptible to bit flips (soft errors). These are managed by using soft error mitigation techniques. However, since these techniques do not differentiate between unwanted bit flips and intentional bit edits, the chapter proposed a re-computation of the frame ECC after bit edits. This makes it possible for SEM techniques to continue to track and correct soft errors while still benefiting from techniques that involve configuration bitstream editing.

9.2 Significance of the Research

The research presented in this thesis is significant in three main domains. These are: improving the reliability of FPGA-based applications, high-performance and low-power computation on FPGAs. These are summarized below.

9.2.1 Impact on the Reliability of FPGA-Based Applications

The techniques presented in this thesis address both permanent and temporal faults in COTS FPGA-based applications. The keys ways in which the reliability of applications is improved are as follows:

i) The design-time application optimization technique presented in chapter 4 leads to better capacity to circumvent permanent faults on COTS FPGAs. This was tested by using data from a practical application, namely data processing tasks of a NASA JPL spectrometer application. The results presented in Figure *4.6* show that an average of 48.6% more errors were survived due to the proposed optimization techniques. In addition, the

functionality-based runtime relocation technique presented in chapter 6 have potentials to improve the relocatability of hardware tasks with low-port widths on modern COTS FPGAs.

ii) The task reuse scheme presented in chapter 4 provides a means of reducing the occupancy of the ICAP and thus leaves more of its resources to be devoted to soft error mitigation techniques. As the simulation results in Table *5.4* shows, the proposed task reuse scheme leads to approximately 29% saving in the amount of configuration compared to state-of-the-art techniques. This saving in the occupancy of the ICAP can be devoted to soft error mitigation operations to ensure that soft errors are detected and corrected more readily.

iii) The frame ECC re-computation engine presented as part of chapter 8 also improves the reliability of applications. Specifically, it enables designs to benefit from a variety of reconfigurable computing techniques which rely on runtime bitstream editing without losing the protected offered by soft error mitigation strategies. By ensuring that the value of ECC for each frame is correct after each operation involving bitstream edit(s), unwanted bit flips due to ionizing radiations, extreme temperatures, etc. can be tracked and corrected in reconfigurable computing applications.

9.2.2 Potentials for Low Power Computation and High Performance

The task wrapper based on memoization presented in chapter 4 leads to significant saving in power consumption for referentially transparent tasks with low port widths. For a case study CORDIC circuit, an average of 34.5% of power saving was obtained using the proposed task wrapper. In addition, the proposed task reuse scheme in chapter 5 also has the potential to reduce power consumption as memory accesses associated with task configuration is an energy intensive operation [124]. Low power computation is a major goal of many system designers. It not only reduces energy bills and increases battery life, but also increases the life span of devices and reduces the risk of electromigration.

In addition, the placement techniques presented in this thesis targets high performance. The worst-case latency of the proposed PMS is less than 50% of that of a state-of-the-art runtime placement system. This leads to shorter placement overheads of hardware tasks thus reducing task's overall execution time. Furthermore, the task rejection ratio of the proposed PMS is lower than that of comparable placement systems. This means that more application components can be executed on the chip in a dynamic runtime placement scenario leading to better performance compared to similar placement systems.

9.3 Limitations and Future Work

There are some limitations associated with the placement management system proposed in this thesis. One limitation is that certain aspects of the techniques presented in the PMS are specific to an FPGA family. An example of this is the proposed runtime clock network routing technique. The technique involves the use of runtime configuration bit editing. This is a technology dependent technique which cannot be directly applicable to other FPGA families. The bit locations for routing clock nets presented in chapter 8 are specific to the Xilinx 7 series FPGA family. To apply the runtime clock routing to another family of FPGA such as the UltraScale FPGA family, the reverse engineering experiments must be repeated to identify the location of the clock buffer and PIP control bits for that FPGA family. This limitation also applies to the Frame ECC re-computation technique presented in chapter 8. Bitstream specific information are necessary for the implementation of the Frame ECC re-computation process, and hence must be re-implemented for another family of FPGA to be useful on them. Another technology dependent technique is DBR using frame address modification. To relocate a bitstream in runtime, the location information in the bitstream must be identified and changed in runtime. Usually, location-dependent information in the bitstream changes between FPGA family, and a relocation controller would need to be updated for each new family of device.

Another technique that is quite limited is the proposed functionality-based relocation technique in chapter 6 and the low-power wrapper proposed in chapter 4. Both of these can only be applied to tasks whose output does not depend on internal states, but only on the current inputs. Some practical applications have outputs which depend on internal states and hence this technique cannot be used to relocate them or minimize their power consumption. Additionally, the techniques use a place reservation technique to ensure that the checks for previously computed outputs are carried out in a pre-determined number of clock cycles. This leads to a high memory overhead for tasks with large port width. Essentially, the size of the memory required to save each output of a task doubles with every increase in the number of its input bits.

In addition, many of the techniques in this thesis require the use of an internal configuration circuitry (which uses the ICAP). This enables an FPGA to be programmed from within itself. However, the configuration circuitry or even the ICAP can be affected by both soft and hard errors which can result in system failure.

To address the challenges identified above, several possibilities can be explored as future work. The following are some possible recommendations for future work:

- **System Integration and Application Testing:** The proposed PMS was implemented as a prototype in this thesis and was tested with a separately implemented communication infrastructure based on the clock buffers [26] and a configuration controller [27] also separately implemented. A next natural step would be to integrate these units into a complete stand-alone ROS and test its performance with real life/critical applications. This would be a next major step in the development of future high-performance, fault-tolerant multisensory electronic systems for hostile environments such as nuclear plants and outer space within the constraints of cost, power and flexibility.

- **Support Across Different FPGA Family:** As identified above, FPGA architecture is continually evolving, and this leads to changes in the configuration bitstream format. Hence, techniques which are dependent on specific formats of the bitstream is not directly applicable across different device families. One way to address this limitation could be for FPGA vendors such as Xilinx to standardize the configuration bitstream format so that designs can be future proof by forecasting locations of essential bits. An additional possibility is for FPGA designers to adopt the virtual bitstream format recommended in [73] which makes tasks bitstream independent of their location on the chip.
- **Improving the scope of low power and functionality-based relocation techniques:** The wrapper for low task computation using memoization as well as the relocation techniques presented in chapters 4 and 6 respectively are practicable for only low port width applications because of the huge memory requirement for applications with large port width. Future work could explore the possibility of using data compression mechanisms to extend the proposed technique to circuits with larger port widths. Fast data compression algorithms which is targeted at in-memory data such as [125] is worth investigating for this.
- **External configuration for reliability:**

  To mitigate the effect of the configuration engine or the ICAP failing, it will be important to extend the reliability of the configuration process by implementing a fall-back configuration engine. A future work could explore the implementation of an efficient configuration using an external processor by extending techniques such as the one illustrated in [126]. The external processor could be radiation hardened to reduce the chances of fault occurrence on it.
- **Performance Testing:** Fault injection was used to test many of the experiments presented in this thesis. As fault injection is not enough to reveal all fault conditions, a future work for this project would be to carry out more testing in actual hostile environments. It is expected such testing would be

carried out in the next phase of the project using the existing collaboration between the University of Edinburgh and NASA JPL.

References

[1] N. Desk, “Highest Growth Forecast for Electronics Components in Mil-Aero Market,” EPS News, 13-Mar-2018. Retrieved from: https://epsnews.com/2018/03/13/highest-growth-forecast-electronics-components-mil-aero-market/ [Accessed] May 25, 2018.

[2] C. Cullinan, C. Wyant, and T. Frattesi, “Computing Performance Benchmarks among CPU, GPU, and FPGA.” 2012. Retrieved from https://web.wpi.edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking_Final.pdf [Accessed] May 25, 2018.

[3] M. Vestias and H. Neto, “Trends of CPU, GPU and FPGA for high-performance computing,” in 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, pp. 1–6.

[4] The Economist “The rise of artificial intelligence is creating new variety in the chip market, and trouble for Intel,” The Economist, 25-Feb-2017. Retrieved from https://www.economist.com/business/2017/02/25/the-rise-of-artificial-intelligence-is-creating-new-variety-in-the-chip-market-and-trouble-for-intel [Accessed] May 25, 2018.

[5] M. Fagan, J. Schlachter, K. Yoshii, S. Leyffer, K. Palem, M. Snir, S. M. Wild, and C. Enz, "Overcoming the power wall by exploiting inexactness and emerging COTS architectural features: Trading precision for improving application quality,” in 2016 29th IEEE International System-on-Chip Conference (SOCC), 2016, pp. 241–246.

[6] A. Ebrahim, “Dynamic Partial Reconfiguration Management for High Performance and Reliability in FPGAs,” PhD Thesis, University of Edinburgh, United Kingdom, 2015.

[7] C. Märtin, “Multicore Processors: Challenges, Opportunities, Emerging Trends,” in proc. Embedded World Conference 2014.

[8] X. Iturbe, Khaled Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez, ‘R3TOS: A novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs’, IEEE Trans. Comput., vol. 62, no. 8, pp. 1542–1556, Aug. 2013

[9] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, “A new approach for on-line placement on reconfigurable devices,” in Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, 2004, pp. 134-140.

[10] F. Cornevaux-Juignet, M. Arzel, P. H. Horrein, T. Groléat, and C. Person, “Open-source flexible packet parser for high data rate agile network probe,” in 2017 IEEE Conference on Communications and Network Security (CNS), 2017, pp. 610–618.

[11] L. D. Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio, “The Role of CAD Frameworks in Heterogeneous FPGA-Based Cloud Systems,” in 2017 IEEE International Conference on Computer Design (ICCD), 2017, pp. 423–426.

[12] A. Adetomi, G. Enemali, and T. Arslan, “Towards an efficient intellectual property protection in dynamically reconfigurable FPGAs,” in 2017 Seventh International Conference on Emerging Security Technologies (EST), 2017, pp. 150–156.

[13] “Intel FPGA and SoC.” [Online]. Available: https://www.altera.com/. [Accessed: 03-Jun-2018].
[14] X. Iturbe, D. Keymeulen, P. Yiu, D. Berisford, K. Hand, R. Carlson and E. Ozer., “A Highly-Efficient, Adaptive and Fault-Tolerant SoC Implementation of a Fourier Transform Spectrometer Data Processing,” in 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, 2015, pp. 231–231.
[15] H. Chauhan, “Can Intel Dominate This Market by Overcoming This Smaller Rival?,” The Motley Fool, 24-Nov-2017. [Online]. Available: https://www.fool.com/investing/2017/11/24/can-intel-dominate-this-market-by-overcoming-this.aspx. [Accessed: 19-Aug-2018].
[16] “Vivado High-Level Synthesis.” [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. [Accessed: 03-Jun-2018].
[17] “Intel® Quartus® Prime Software - Overview.” [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html. [Accessed: 03-Jun-2018].
[18] M. Eckert, D. Meyer, J. Haase, and B. Klauer, “Operating System Concepts for Reconfigurable Computing: Review and Survey,” International Journal of Reconfigurable Computing, vol. 2016, pp. 11 - 22, 2016.
[19] X. Iturbe, A. Ebrahim, K. Benkrid, C. Hong, T. Arslan, J. Perez, D. Keymeulen, M. D. Santambrogio, ‘R3TOS-Based autonomous fault-tolerant systems’, IEEE Micro, vol. 34, no. 6, pp. 20–30, Nov. 2014.
[20] G. Brebner, “A virtual hardware operating system for the Xilinx XC6200,” in Field-Programmable Logic Smart Applications, New Paradigms and Compilers, vol. 1142, R. W. Hartenstein and M. Glesner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 327–336.
[21] G. Enemali, A. Adetomi, and T. Arslan, “Expanding the un-usable area strategy for improved utilization of reconfigurable FPGAs,” in 2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2017, pp. 139–144.
[22] G. Enemali, A. Adetomi, and T. Arslan, “A placement management circuit for efficient realtime hardware reuse on FPGAs targeting reliable autonomous systems,” 2017, pp. 1–4.
[23] G. Enemali, A. Adewale, and T. Arslan, “FAReP: Fragmentation-Aware Replacement Policy for Task Reuse on Reconfigurable FPGAs,” in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Orlando, 2017.
[24] G. Enemali, A. Adetomi, and T. Arslan, “A Functionality-Based Runtime Relocation System for Circuits on Heterogeneous FPGAs,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 5, pp. 612–616, May 2018.
[25] G. Enemali, A. Adetomi, and T. Arslan, “Efficient Runtime Frame ECC Recomputation for Reliable Task Execution on Xilinx FPGAs,” in 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), Edinburgh, 2018. In Press.
[26] A. Adetomi, G. Enemali, and T. Arslan, “Relocation-aware communication network for circuits on Xilinx FPGAs,” in proc. of 2017 International

Conference on Field-Programmable Logic and Applications (FPL), 2017, pp. 1–7.
[27] A. Adetomi, G. Enemali, and T. Arslan, “A fault-tolerant ICAP controller with a selective-area soft error mitigation engine,” in 2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2017, pp. 192–199.
[28] I. Kuon, R. Tessier, and Jonathan Rose, FPGA Architecture: Survey and Challenges. Now Publishers Inc, 2008.
[29] L. Kechiche, L. Touil, and B. Ouni, “Toward the Implementation of an ASIC-Like System on FPGA for Real-Time Video Processing with Power Reduction,” International Journal of Reconfigurable Computing, 2018. [Online]. Available: https://www.hindawi.com/journals/ijrc/2018/2843582/abs/. [Accessed: 06-Mar-2019].
[30] P. Alfke, I. Bolsens, B. Carter, M. Santarini, and S. Trimberger, “It’s an FPGA!,” IEEE Solid-State Circuits Mag., vol. 3, no. 4, pp. 15–20, Fall 2011.
[31] S. M. S. Trimberger, “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology,” IEEE Solid-State Circuits Mag., vol. 10, no. 2, pp. 16–29, Spring 2018.
[32] R. Singh, “FPGA vs ASIC: Differences between them and which one to use? | Numato Lab Help Center.” [Online]. Available: https://numato.com/blog/differences-between-fpga-and-asics/. [Accessed: 27-Feb-2019].
[33] K. Vipin and S. A. Fahmy, “FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications,” ACM Comput. Surv., vol. 51, no. 4, pp. 1–39, Jul. 2018.
[34] J. L. Nunes, “Improving the dependability of FPGA-based real-time embedded systems with partial dynamic reconfiguration,” in 2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W), 2013, pp. 1–4.
[35] J. Vliegen. Partial and dynamic FPGA reconfiguration for security applications. PhD thesis, KU Leuven, 2014. Nele Mentens and Ingrid Verbauwhede (promotors).
[36] I. Xilinx, “Vivado Design Suite User Guide: Partial Reconfiguration (UG909).” 2016.
[37] “Vivado Design Suite Tutorial: Partial Reconfiguration (UG947),” 2016.
[38] J. A. Clemente, J. Resano, C. Gonzalez, and D. Mozos, “A Hardware Implementation of a Run-Time Scheduler for Reconfigurable Systems,” IEEE Trans. Very Large Scale Integr. VLSI Syst., vol. 19, no. 7, pp. 1263–1276, Jul. 2011.
[39] I. Xilinx, “Partial Reconfiguration Controller.” 2018.
[40] A. Adetomi, G. Enemali, and T. Arslan, “Relocating Encrypted Partial Bitstreams by Advance Task Address Loading,” in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 188–191.
[41] G. Bloom, B. Narahari, R. Simha, A. Namazi, and R. Levy, “FPGA SoC architecture and runtime to prevent hardware Trojans from leaking secrets,” in 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2015, pp. 48–51.

[42] Z. Zhang, Q. Yu, L. Njilla, and C. Kamhoua, “FPGA-oriented moving target defense against security threats from malicious FPGA tools,” in 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2018, pp. 163–166.
[43] G. Wigley, D. Kearney, and others, “Research issues in operating systems for reconfigurable computing,” in proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms (ERSA), 2002, pp. 10–16.
[44] R. Tessier, K. Pocek, and A. DeHon, “Reconfigurable Computing Architectures,” Proc. IEEE, vol. 103, no. 3, pp. 332–354, Mar. 2015.
[45] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, “ReconOS: An operating system approach for reconfigurable computing,” IEEE Micro, vol. 34, no. 1, pp. 60–71.
[46] D. Gohringer, M. Hubner, E. N. Zeutebouo, and J. Becker, “CAP-OS: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures,” in 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, pp. 1–8.
[47] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, “The LEAP FPGA operating system,” in 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, pp. 1–8.
[48] M. Jacobsen and R. Kastner, “RIFFA 2.0: A reusable integration framework for FPGA accelerators,” in 2013 23rd International Conference on Field programmable Logic and Applications, 2013, pp. 1–8.
[49] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, “Hardware Task Scheduling for Partially Reconfigurable FPGAs,” in Applied Reconfigurable Computing, vol. 9040, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2015, pp. 487–498.
[50] Xilinx Inc, “7 Series FPGAs Data Sheet: Overview (DS180)”, 2018.
[51] Xilinx Inc, “7 Series FPGAs Configuration User Guide (UG470),” 2018.
[52] C. R. Julien, B. J. LaMeres, and R. J. Weber, “An FPGA-based radiation tolerant SmallSat Computer System,” in 2017 IEEE Aerospace Conference, 2017, pp. 1–13.
[53] Xilinx Inc, “Device Reliability Report, Seconf Half 2017" (UG116),” 2018
[54] K. Vittala, M. Niamat, and S. Vemuru, “Early lifetime failure detection in FPGAs using delay faults,” in NAECON 2014 - IEEE National Aerospace and Electronics Conference, 2014, pp. 391–395.
[55] A. Amouri, F. Bruguier, S. Kiamehr, P. Benoit, L. Torres, and M. Tahoori, “Aging effects in FPGAs: an experimental analysis,” in 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, pp. 1–4.
[56] I. Xilinx, “Soft Error Mitigation Controller v4.1 (PG036).” 2015.
[57] H. Michel, A. Belger, T. Lange, B. Fiethe, and H. Michalik, “Read back scrubbing for SRAM FPGAs in a data processing unit for space instruments,” in 2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2015, pp. 1–8.

[58] Q. Martin and A. D. George, “Scrubbing optimization via availability prediction (SOAP) for reconfigurable space computing,” in 2012 IEEE Conference on High Performance Extreme Computing, 2012, pp. 1–6.
[59] M. Welter, “Demonstration of Soft Error Mitigation IP and Partial Reconfiguration Capability on Monolithic Devices - XAPP1261 (v1.0)’.” Xilinx Inc, 2015.
[60] X. Iturbe, K. Benkrid, T. Arslan, C. Hong, A. T. Erdogan, and I. Martinez, “Enabling FPGAs for future deep space exploration missions: Improving fault-tolerance and computation density with R3TOS,” in 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2011, pp. 104–112.
[61] X. Iturbe, K. Benkrid, T. Arslan, R. Torrego, and I. Martinez, “Methods and Mechanisms for Hardware Multitasking: Executing and Synchronizing Fully Relocatable Hardware Tasks in Xilinx FPGAs,” in 2011 21st International Conference on Field Programmable Logic and Applications, 2011, pp. 295–300.
[62] X. Iturbe, K. Benkrid, R. Torrego, A. Ebrahim, and T. Arslan, “Online clock routing in Xilinx FPGAs for high-performance and reliability,” in 2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2012, pp. 85–91.
[63] X. Iturbe, K. Benkrid, Chuan Hong, A. Ebrahima, T. Arslan, and I. Martinez, “Runtime Scheduling, Allocation, and Execution of Real-Time Hardware Tasks onto Xilinx FPGAs Subject to Fault Occurrence,” Int. J. Reconfigurable Comput., pp. 1–32, Jan. 2013.
[64] K. Bazargan, R. Kastner, and M. Sarrafzadeh, “Fast template placement for reconfigurable computing systems,” IEEE Des. Test Comput., vol. 17, no. 1, pp. 68–83, Jan. 2000.
[65] E. Lubbers and M. Platzner, “ReconOS: An RTOS Supporting Hard-and Software Threads,” in 2007 International Conference on Field Programmable Logic and Applications, 2007, pp. 441–446.
[66] D. Andrews and M. Platzner, “Programming models for reconfigurable manycore systems,” in 2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2016, pp. 1–8.
[67] D. Göhringer, M. Hübner, L. Hugot-Derville, and J. Becker, “Message Passing Interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC,” in Modeling and Simulation 2010 International Conference on Embedded Computer Systems: Architectures, 2010, pp. 357–364.
[68] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, “Theory of latency-insensitive design,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 20, no. 9, pp. 1059–1076, Sep. 2001.
[69] C. Beckhoff, D. Koch, and J. Torresen, “Go Ahead: A Partial Reconfiguration Framework,” in 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, 2012, pp. 37–44.
[70] J. Tabero, J. Septién, H. Mecha, and D. Mozos, “A low fragmentation heuristic for task placement in 2D RTR HW management,” Field Program. Log. Appl., pp. 241–250, 2004.

[71] J. Tabero, J. Septién, H. Mecha, and D. Mozos, “Allocation heuristics and defragmentation measures for reconfigurable systems management,” Integr. VLSI J., vol. 41, no. 2, pp. 281–296, 2008.
[72] J. Septien, D. Mozos, H. Mecha, J. Tabero, and M. A. G. de Dios, “Perimeter quadrature-based metric for estimating FPGA fragmentation in 2D HW multitasking,” in 2008 IEEE International Symposium on Parallel and Distributed Processing, 2008, pp. 1–8.
[73] Q.-H. Khuat, D. Chillet, and M. Hubner, “Considering reconfiguration overhead in scheduling of dependent tasks on 2D reconfigurable FPGA,” in 2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2014, pp. 1–8.
[74] M. Koester, M. Porrmann, and H. Kalte, “Task placement for heterogeneous reconfigurable architectures,” in Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on, 2005, pp. 43–50.
[75] T. Becker, W. Luk, and P. Y. Cheung, “Enhancing relocatability of partial bitstreams for run-time reconfiguration,” in Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on, 2007, pp. 35–44.
[76] M. Koester, W. Luk, J. Hagemeyer, and M. Porrmann, “Design optimizations to improve placeability of partial reconfiguration modules,” in Proceedings of the Conference on Design, Automation and Test in Europe, 2009, pp. 976–981.
[77] M. Koester, W. Luk, J. Hagemeyer, M. Porrmann, and U. Ruckert, “Design Optimizations for Tiled Partially Reconfigurable Systems,” IEEE Trans. Very Large Scale Integr. VLSI Syst., vol. 19, no. 6, pp. 1048–1061, Jun. 2011.
[78] A. Ejnioui and R. F. DeMara, “Area Reclamation Strategies and Metrics for SRAM-Based Reconfigurable Devices.,” in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA’05), 2005, pp. 196–202.
[79] M. Handa and R. Vemuri, “Area Fragmentation in Recon?gurable Operating Systems,” In Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms. CSREA Press, Jun. 2004.
[80] A. Ebrahim, T. Arslan, and X. Iturbe, “On enhancing the reliability of internal configuration controllers in FPGAs,” in 2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2014, pp. 83–88.
[81] Y. Lu, K. Bertels, and G. Gaydadjiev, “Efficient hardware task reuse and interrupt handling mechanisms for FPGA-based partially reconfigurable systems,” in 2010 International Conference on Field-Programmable Technology, 2010, pp. 324–327.
[82] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, “Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs,” in 2016 Design, Automation Test in Europe Conference Exhibition (DATE), 2016, pp. 1505–1508.
[83] A. Lifa, P. Eles, and Z. Peng, “Minimization of average execution time based on speculative FPGA configuration prefetch,” in 2012 International Conference on Reconfigurable Computing and FPGAs, 2012, pp. 1–8.

[84] Z. Li, K. Compton, and S. Hauck, “Configuration caching management techniques for reconfigurable computing,” in Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on, 2000, pp. 22–36.
[85] K. Sigdel, C. Galuzzi, K. Bertels, M. Thompso, and A. D. Pimentel, “Runtime task mapping based on hardware configuration reuse,” in Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on, 2010, pp. 25–30.
[86] M. Mansub Bassiri and H. Shahriar Shahhoseini, “Configuration Reusing in On-Line Task Scheduling for Reconfigurable Computing Systems,” J. Comput. Sci. Technol., vol. 26, no. 3, pp. 463–473, May 2011.
[87] J. A. Clemente, D. Mozos, and J. Resano, “A Replacement Technique to Maximize Task Reuse in Reconfigurable Systems,” in 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 250–257.
[88] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck, “Configuration relocation and defragmentation for run-time reconfigurable computing,” IEEE Trans. Very Large Scale Integr. VLSI Syst., vol. 10, no. 3, pp. 209–220, Jun. 2002.
[89] C. Schuck, B. Haetzer, M. Hubner, and J. Becker, “Online Routing of FPGA Clock Networks for Module Relocation in Partial Reconfigurable Multi Clock Designs,” 2011, pp. 181–188.
[90] Y. Wu, S. Thomson, H. Sun, D. Krause, S. Yu, and G. Kurio, “Free Razor: A Novel Voltage Scaling Low-Power Technique for Large SoC Designs,” IEEE Trans. Very Large Scale Integr. VLSI Syst., vol. 23, no. 11, pp. 2431–2437, Nov. 2015.
[91] I. Qiqieh, R. Shafik, G. Tarawneh, D. Sokolov, and A. Yakovlev, “Energy-efficient approximate multiplier design using bit significance-driven logic compression,” in Design, Automation Test in Europe Conference Exhibition (DATE), 2017, 2017, pp. 7–12.
[92] J. Huang, J. Lach, and G. Robins, “A methodology for energy-quality tradeoff using imprecise hardware,” in DAC Design Automation Conference 2012, 2012, pp. 504–509.
[93] Altera, “Reducing Power Consumption and Increasing Bandwidth on 28-nm FPGAs.” Retrieved from https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01148-stxv-power-consumption.pdf. Accessed 26 Sept 2017.
[94] A. Nafkha and Y. Louet, “Accurate measurement of power consumption overhead during FPGA dynamic partial reconfiguration,” in 2016 International Symposium on Wireless Communication Systems (ISWCS), 2016, pp. 586–591.
[95] I. Xilinx, “7 Series FPGAs Memory Resources.” 2016.
[96] J. L. Nunez-Yanez, “Adaptive Voltage Scaling with In-Situ Detectors in Commercial FPGAs,” IEEE Trans. Comput., vol. 64, no. 1, pp. 45–53, Jan. 2015.
[97] H. Qi, O. Ayorinde, and B. Calhoun, “An Energy-Efficient Near/Sub-Threshold FPGA Interconnect Architecture Using Dynamic Voltage Scaling and Power-Gating,” in International Conference on Field-Programmable Technology (FPT), China, 2016, pp. 20–27.

[98] H. Park, S. Vijayvargiya, and A. DeHon, “Energy minimization in the time-space continuum,” in 2015 International Conference on Field Programmable Technology (FPT), Queenstown, New Zealand, 2015, pp. 64–71.
[99] C. Alvarez, J. Corbal, and M. Valero, “Dynamic Tolerance Region Computing for Multimedia,” IEEE Trans. Comput., vol. 61, no. 5, pp. 650–665, May 2012.
[100] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, “ABACUS: A technique for automated behavioral synthesis of approximate computing circuits,” in 2014 Design, Automation Test in Europe Conference Exhibition (DATE), 2014, pp. 1–6.
[101] S. Sinha and W. Zhang, “Low-Power FPGA Design Using Memoization-Based Approximate Computing,” IEEE Trans. Very Large Scale Integr. VLSI Syst., vol. 24, no. 8, pp. 2665–2678, Aug. 2016.
[102] F. Khalvati and M. D. Aagaard, “Window memoization: an efficient hardware architecture for high-performance image processing,” J. Real-Time Image Process., vol. 5, no. 3, pp. 195–212, Sep. 2010.
[103] M. Gort and J. Anderson, “Design re-use for compile time reduction in FPGA high-level synthesis flows,” in 2014 International Conference on Field-Programmable Technology (FPT), 2014, pp. 4–11.
[104] I. Xilinx, “CORDIC v6. 0 LogiCORE IP Product Guide.” 2017.
[105] X. Iturbe, D. Keymeulen, P. Yiu, D. Berisford, K. Hand, R. Carlson and E. Ozer, “Towards a generic and adaptive System-on-Chip controller for space exploration instrumentation,” in 2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2015, pp. 1–8.
[106] I. Xilinx, “7 Series FPGAs Configurable Logic Block User Guide (UG474).” 2016.
[107] I. Xilinx, “Partial Reconfiguration User Guide.” 2013.
[108] A. Adetomi, G. Enemali, and T. Arslan, “Clock Buffers, Nets, and Trees for On-Chip Communication: A Novel Network Access Technique in FPGAs,” in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 219–222.
[109] A. Adetomi, G. Enemali, and T. Arslan, “Characterization of Clock Buffers for On-Chip Inter-Circuit Communication in Xilinx FPGAs,” in 2018 IEEE International Symposium on Circuits and Systems (ISCAS), 2018, pp. 1–5.
[110] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, “Analysis and characterization of inherent application resilience for approximate computing,” 2013, p. 1.
[111] S. Joseph and K. Baskaran, “Performance analysis of various fragmentation techniques in runtime partially reconfigurable FPGA,” Int. J. Comput. Appl., vol. 94, no. 8, 2014.
[112] F. Dittmann and S. Frank, “Hard Real-Time Reconfiguration Port Scheduling,” in Automation Test in Europe Conference Exhibition 2007 Design, 2007, pp. 1–6.
[113] M. Walter, “Demonstration of Soft Error Mitigation IP and Partial Reconfiguration Capability on Monolithic Devices.” Jun-2015.
[114] A. DeHon and S. Hauck, Reconfigurable Computing: Theory and Practice of FPGA based computation. Amsterdam: Morgan Kaufmann, 2008.

[115] L. Kirischian, V. Kirischian, and D. Sharma, “Mitigation of Thermo-cycling effects in Flip-chip FPGA-based Space-borne Systems by Cyclic On-chip Task Relocation,” in 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2018, 2018.

[116] A. Lalevée, P. H. Horrein, M. Arzel, M. Hübner, and S. Vaton, “AutoReloc: Automated Design Flow for Bitstream Relocation on Xilinx FPGAs,” in 2016 Euromicro Conference on Digital System Design (DSD), 2016, pp. 14–21.

[117] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 26, no. 2, pp. 203–215, Feb. 2007.

[118] H. T. Nguyen, X. T. Nguyen, and C. K. Pham, “A Low-Power Hybrid Adaptive CORDIC,” IEEE Trans. Circuits Syst. II Express Briefs, vol. 65, no. 4, pp. 496–500, Apr. 2018.

[119] Xilinx, “RGB to YCrCb Color-Space Converter v7.1 LogiCORE IP Product Guide.” 2015.

[120] Xilinx, “Multiplier v12.0 LogiCORE IP Product Guide.” 2015.

[121] A. Adetomi, G. Enemali, and T. Arslan, “R3TOS-Based Integrated Modular Space Avionics for On-Board Real-Time Data Processing,” in 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2018, Edinburgh, 2018. In press

[122] A. DeHon, R. Huang, and J. Wawrzynek, “Hardware-assisted fast routing,” in Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002, pp. 205–215.

[123] C. E. Warren, D. P. Schultz, and S. P. Young, “Error checking parity and syndrome of a block of data with relocated parity bits,” US12188935, 2008.

[124] J. A. Clemente, E. P. Ramo, J. Resano, D. Mozos, and F. Catthoor, “Configuration Mapping Algorithms to Reduce Energy and Time Reconfiguration Overheads in Reconfigurable Systems,” IEEE Trans. Very Large Scale Integr. VLSI Syst., vol. 22, no. 6, pp. 1248–1261, Jun. 2014.

[125] S.-J. Kwon, S.-H. Kim, H.-J. Kim, and J.-S. Kim, “LZ4m: A fast compression algorithm for in-memory data,” in 2017 IEEE International Conference on Consumer Electronics (ICCE), 2017, pp. 420–423.

[126] M. Nielson, “Using a Microprocessor to Configure 7 Series FPGAs via Slave Serial or Slave SelectMAP Mode,” Xilinx ® 2012.