9-5-2013

A Dynamically Reconfigurable Parallel Processing Framework with Application to High-Performance Video Processing

John Hoffman

John Colby Hoffman

Candidate

Electrical and Computer Engineering

Department

This dissertation is approved, and is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

MARIOS PATTICHIS ,Chairperson

CHRISTOS CHRISTODOULOU

PAYMAN ZARKESH-HA

CARLOS LOPEZLEIVA

A Dynamically Reconfigurable Parallel Processing Framework with Application to High-Performance Video Processing

by

John Colby Hoffman

Electrical Engineering, Texas Tech University
USA, 1998

M.Sc., Computer Engineering, University of New Mexico,
USA, 2009

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico
Albuquerque, New Mexico

July, 2013

Dedication

I dedicate this dissertation to my family, especially…

My brilliant and outrageously loving and supportive wife, Corrina;

our energetic, inquisitive, and one of the joys of my life boy, Nathaniel;

our exuberant, sweet, and the other joy of my life girl, Breeanna;

to the guy who has always been there for me my brother, Chad;

to Dad and Mom for instilling the importance of hard work and higher education;

to my grandfather, late grandmother, for summers of joy while I grew up on the farm;

to my wife's entire wonderful and supportive family.

Acknowledgements

I heartily acknowledge Dr. Marios Pattichis, my advisor and dissertation chair, for continuing to encourage me through the years of classroom teachings and the long number of months writing and rewriting these chapters. His guidance and professional style will remain with me as I continue my career.

I also thank my committee members for their valuable recommendations pertaining to this study and assistance in my professional development. Gratitude is extended to my friends at Raytheon Company, Xilinx, inc. and Altera, inc. for pushing me to finish this research.

A Dynamically Reconfigurable Parallel Processing Framework with Application to High-Performance Video Processing

by

John Colby Hoffman

Electrical Engineering, Texas Tech University
USA, 1998

M.Sc., Computer Engineering, University of New Mexico,
USA, 2009

Ph.D., Engineering, University of New Mexico,
USA, 2013

Abstract

Digital video processing demands have and will continue to grow at unprecedented rates. Growth comes from ever increasing volume of data, demand for higher resolution, higher frame rates, and the need for high capacity communications. Moreover, economic realities force continued reductions in size, weight and power requirements. The ever-changing needs and complexities associated with effective video processing systems leads to the consideration of dynamically reconfigurable systems.

The goal of this dissertation research was to develop and demonstrate the viability of integrated parallel processing system that effectively and efficiently apply pre-optimized hardware cores for processing video streamed data. Digital video is decomposed into packets which are then distributed over a group of parallel video processing cores. Real time processing requires an effective task scheduler that distributes video packets efficiently to any of the reconfigurable distributed processing nodes across the framework, with the nodes running on FPGA reconfigurable logic in an inherently "Virtual" mode.

The developed framework, coupled with the use of hardware techniques for dynamic processing optimization achieves an optimal cost/power/performance realization for video processing applications. The system is evaluated by testing processor utilization relative to I/O bandwidth and algorithm latency using a separable 2-D FIR filtering system, and a dynamic pixel processor. For these applications, the system can achieve performance of hundreds of 640x480 video frames per second across an eight lane Gen I PCIe bus. Overall, optimal performance is achieved in the sense that video data is processed at the maximum possible rate that can be streamed through the processing cores. This performance, coupled with inherent ability to dynamically add new algorithms to the described dynamically reconfigurable distributed processing framework, creates new opportunities for realizable and economic hardware virtualization.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This dissertation focuses on the application of dynamically reconfigurable hardware to develop effective and novel means to implement adaptable and high performance video processing methods with FPGA technology. The primary motivation for this research comes from the need for flexible, reprogrammable or reconfigurable solutions that can be deployed before design requirements have been finalized, and can then be adapted to changing requirements after deployment.

Prior research enabling exploitation of FPGA reconfiguration technology has begun to address some of the many complexities associated with the development of hardware and configurable hardware whose functionality can be changed during operation. The potential of self-adaptable hardware to enable revolutionary new applications has led to a large and growing body of work in reconfigurable computation. The recent explosive growth of video datasets motivates the exploration of how dynamically reconfigurable methods can be used for video processing. The use of dynamically reconfigurable methods for video processing has received very little attention. This dissertation research will build on recent ivPCL research by extending the dynamic pixel processor and dynamically reconfigurable FIR system in a high-performance video processing system.

1.0 Overview

Dynamic Partial Reconfiguration (DPR) of Field Programmable Gate Array (FPGA) devices represents a major disruptive technology that can be used to meet vital needs for increased system efficiency and resiliency. This value proposition can be

demonstrated for applications such as satellite-based image processing [1] [2]. For these systems, upgrades are typically quite complex, risky, and expensive. Furthermore, available resources are constrained by Size, Weight, and Power (SWAP) requirements. FPGA devices incorporated into these systems represent high-value, finitely-limited resources. Once deployed, the available FPGA logic is strictly finite.

Consequently, an ability to reconfigure the logic of such devices after deployment enables revision of the design logic, as necessary, to correct design deficiencies or to alter or enhance functionality. The logic of an FPGA device is specified (programmed) in the manner of uploading one or more files to the device that specifies the configuration of the device logic. These files are commonly referred to as *bitstream* files. When an electrically reprogrammable FPGA device is initially programmed following power-on or reset, a default bitstream is loaded to that device, often from an associated memory through the device programming [3].

It is obvious that, at a minimum, non-run-time reconfiguration enables corrections and upgrades. Beyond a specified reconfiguration, DPR can be used to correct, enhance, or replace firmware functional blocks in the FPGA reconfigurable logic while that device is operating in the system without disruption of other ongoing functions. Programming these devices with partial bitstreams can now be accomplished over high speed multi-gigabit per second Serializer/Deserializer (SerDes) interfaces and devices which will be configured and programmed specifically to support ongoing device level needs. This capability provides modification flexibility analogous to software while realizing functional density and performance efficiency of hardware. Cost-effective, easily-modified systems are essential to today's complex systems and development success,

which can be accomplished with this approach. However, achieving this outcome requires the right tools and correct techniques; but moreover it most requires a structured (well-posed) yet flexible architectural approach.

For this to be a reality will require the development of the appropriate techniques and necessary tools which leads to a packet based parallel processing framework for complex infrastructures such as an FPGA Application Store for FPGA Intellectual Property (IP) development and horizontal deployment.

This research builds off of recent work in the area of Dynamic Energy/Power-Performance-Accuracy (EPA/PPA) management which provided a management approach for digital signal, image, and video processing architectures. In this work EPA/PPA was demonstrated with the use of Dynamic Partial Reconfiguration (DPR) and Dynamic Frequency Control (DFC) on FPGAs [4] [5] [6] [7].

1.1. Motivation for Dissertation Research

The majority of FPGA hardware designs and implementations tend to be expensive, tedious, and static. The term static is used here to refer to the fact that the designs are completed according to pre-defined (fixed) requirements. Indeed, even if the initial architects use a well-structured architectural design, once detailed design, timing optimization, and final verification, there is considerable risk associated with asking a later design team to attempt modification. Thus, design modification rarely happens, especially for large and complex designs.

Yet device geometries continue to shrink because the density difference between FPGA technology and standard cell Application Specific Integrated Circuit (ASIC) technology is roughly an order of magnitude [8]. In part, this is due to the simple fact

that the logical design of FPGA technology is based on ASIC technology perhaps supplemented with more customized design of the embedded memory structures themselves. Inherently, the logic density of programmable logic will remain at least an order of magnitude lower than that achievable in customized directly instantiated logic.

The implication of this fact is that the achievable density of FPGA technology directly tracks that of the underlying ASIC technology and consequently continues to grow at that same rate. Emerging FPGA devices can achieve or perhaps exceed the capabilities achievable a decade ago in a dense standard cell technology and consequently remain complex to design. Moreover, physical constraints that limit the ability of a signal to propagate across a die become increasingly adverse at the individual logic cell level necessitating much more knowledge of physical placement of function than had been necessary in older FPGA technologies. The end result is that the difficulties in designing FPGAs are driving tool requirements to provide analogous physical optimization required for ASIC design. Yet the end result needs to be far cheaper and faster [9].

Fortunately, the underlying optimization technologies developed for ASIC design needed now to address the issues with FPGA design were available a decade ago and have matured since. Unfortunately widespread adaptation has not occurred due to the additional costs associated with the need for those same more sophisticated tool flows and inertial resistance to make significant investments and change without clear widespread adoption.

The engineering process for firmware can be accelerated with increased use of functional component reuse. Unfortunately, the present FPGA industry process does not

directly lead to reusable components for a number of reasons including design methodology. However, once it becomes inherently advantageous to create physically modular IP blocks for partially reconfigurable designs, reuse through standardization can become a natural and pervasive result. The combination of physical and logical standardization in turn can lead to the natural evolutionary result of macro-scale modular pre-tested/pre-validated components with specific physical layout within a standard, regular shaped region all built on established tools, technology, expertise and conformal with the considered base of products and related IP. Having accomplished that, it is apparent that even enormous applications reasonably decomposable into structured components of smaller, more manageable sizes are more easily created in a manner that converges to an acceptably sub-optimum solution much more effectively and quickly than with non-partitioned single monolithic applications. This divide-and-conquer method is the basis of modern systems engineering practice. This practice produces designs whose individual components and aggregated designs can be far more reasonably revised and refined than large non-structured monolithic implementations.

Furthermore, if those same applications were to be created conformal to a regularized macro-scale tiling with standardized electrical and messaging interconnections, then the FPGA fabrics provisioning of local and global routing resources can be partitioned to accommodate tile-to-tile and longer pass-through routing. Under a suitable framework for defining interconnection, tiling, and resource provisioning, the placement of individual functional tiles becomes a problem in tile assembly.

A primary consideration for the specification of this framework is one of tradeoffs between manageable complexity of individual tiles, nominal average tile utilizations, and total number of unique tiles. Tile size fidelity encapsulates a number of FPGA (Combinational Logic Blocks) CLBs, but partitions the CLB fabric with sufficient granularity to provide sufficient ability to accommodate the majority of smaller re-useable macro-functional elements.

However, the necessity to provide this same design paradigm for systemic creation of standardized physical and electrical macro elements that in turn systemically leads to the necessity to create interoperable and swappable components then directly enables the feasibility of near-real-time DPR device reconfiguration. Enabling rapid and dynamic firmware reconfiguration also suggests that defects can be more easily corrected after the product is fielded and consequently it becomes possible to put the product in the field to begin evaluation sooner. This means that other potentially major issues normally are only discovered by integration, in actual use, or when exposed to real-world circumstances can be discovered and corrected earlier and more effectively leading to cost reduction, faster deployment, and better results.

FPGA DPR firmware will provide immediate or real-time updates to fielded hardware. Rapid responses can be deployed horizontally in near-real time to networks of systems with these capabilities. It provides rapid responses and the capability for immediate solutions to be horizontally deployed across numerous systems and platforms.

Situation tailored capability becomes an important goal. System hardware capabilities can be modified at a given moment to adapt the current environment or

situation. Moreover, dynamic functional adaptations as well as application mobility across networked systems becomes a realizable possibility [10].

1.2. Related Research

There has been considerable research over the past decade on the subject of reconfigurable computation; however there remains substantial work to be accomplished to create technologies, techniques, and tools that enable exploitation of reconfigurable hardware. A survey of recently published literature testifies to the lack of sophisticated capabilities and tools to use DPR technology within a broad-based mainstream design methodology. Examples that clearly demonstrate limitations within the current art are discussed in following paragraphs and are related to results established during the course of this research.

SRAM-based FPGA technologies in particular have attracted substantial attention due to investments made by companies including Xilinx in reconfiguration technology. Reconfigurable technologies occupy the performance-density /cost space between ASIC and custom silicon and microprocessor-based software solutions and consequently have attracted considerable attention. While the ability to reprogram (reconfigure) SRAM-based FPGAs has existed since these products were first introduced, in practice, it happens very infrequently. With each generation, device complexities continue to grow and consequently the complexity of the potential reconfiguration trade spaces grow even faster. Compounding the issue, the latest generation devices now provide the ability to reprogram portions of a design while the device continues to operate.

Examination of the literature suggests that there are many open issues regarding how to efficiently and effectively use these capabilities. Recent research by Fons [11]

suggests that the existing run-time reconfigurable computing ecosystem is at a sufficient "level of maturity" for the "professional design and development of embedded electronic systems." However, at the same time he qualifies this remark noting that further advances are required, especially regarding automation tools. While the basic low-level technology capabilities exist, sophisticated capabilities to effectively and efficiently exploit hardware reconfiguration do not.

The author states that "when working with reconfigurable hardware technology the development effort required to get any application to produce even modest performance is high compared with a purely software implementation, and any seemingly small detail can easily result in a significant performance degradation [11]."

Fons discusses an open system architecture driven by a reconfiguration engine that in turn supplies inputs to a logic synthesis tool for creating applications on SRAM-based FPGAs. The described framework identifies and labels recognized computational tasks that otherwise would commonly be synthesized into static logic. The author investigates a number of specific application implementations while incorporating on-the-fly reconfiguration techniques into those designs. In one such example, implementation of a CORDIC function is examined. Trigonometric functions are synthesized and a micro-controller is used to reconfigure ran FPGA with the new function. When the associated application software code invokes the function, the associated function is loaded into the FPGA. FPGA execution is performed on demanded when the application invokes one of the supported functions. This then initiates an automated mechanism to affect data transfer with the FPGA as well as with the FPGA for partial reconfiguration download.

In contrast to the highly automated and highly interactive processing techniques investigated in the present research, Fons' approach focused on demand-driven function synthesis that then is used to reconfigure a device to meet a new need. Within this paradigm, reconfiguration performance is a function of the power of available FPGA design automation and synthesis tool performance. The author in fact notes that the time "required by the CAD tools to place and route a design into a physical device can be extremely long" and thus this approach would benefit from on-going work high-level language synthesis for greater abstraction and that "the complexity of embedded systems increases at a rate that is not met by the development of advanced CAD tools for managing such a large design space" and consequently "there exists risk that this progress remains only in research prototypes and some products."

Aggaarwal et al [12]proposed the System Coordination Framework (SCF). This work focuses on heterogeneous computation including FPGAs, CPUs, and other processing elements. Their research primarily addresses inter-processor task communications through message passing between different tools (and languages). This occurs while executing on multiple, different processing devices ranging from embedded to High-Performance Computing (HPC) systems.

Unlike their work addressing more distributed applications, the present research focuses on task coordination related to the direct control of information flows through a highly parallelized FPGA fabric, performing processing tasks on that hardware, and then scheduling and controlling the allocation, reconfiguration, and continual modification of the logic incorporated within that same FPGA hardware fabric. The very high speed and continual DPR techniques described herein show how individual DPR regions within an

array of such regions can be independently modified in an adaptive manner quickly and effectively.

The NSF Center for High-Performance Reconfigurable Computing also has done extensive research in the area of FPGA Design Frameworks for Dynamic Partial Reconfiguration [13] [14]. That research focused on the actual physical layout of FPGA devices and not the communication network fabric or the optimizations needed to achieve an efficient dynamic partial reconfiguration framework. These topics demand better solutions than are currently available and the current research directly addressed both.

Tan and DeMara [15] discuss why a sophisticated partial reconfiguration framework is essential to integrate and optimize existing FPGA technologies in order to exploit the potential of PR and DPR techniques. Towards this end, they suggest standardized APIs, abstracted data structures, and structured access to heterogeneous logic and communication resources. Important framework attributes were suggested to include (1) autonomous FPGA partial reconfiguration without manual intervention, (2) task-level granularity, (3) reconfiguration bitstreams generated at run-time as well as during design. This prior research work serves to reinforce motivations for the current dissertation.

Similarly, Marconi [16] recently investigated issues associated with PR hardware task scheduling and placement assuming pre-designed hardware tasks. Hardware tasks were defined by three parameters: area (width and height), reconfiguration time, and its execution time. This work explores aspects of area utilization, hardware allocation overhead, execution time, and resource scheduling. The author also discussed reconfiguration overhead due to aspects of existing configuration infrastructures. This

work was limited and expanded on in this dissertation with respect to resource scheduling of the data flow through the reconfigurable system before during and after the reconfiguration.

Other research directions related to better DPR methods and process flows include bitstream size compression algorithms [17]. Sudarsanam [18] proposed modification of a niche DPR technique for relocation of blocks tied to a specific family of devices, modifying an approach developed by Carver et al. [19] adding a decompression algorithm for obtaining the next word in the bitstream. One reported limitation of that approach is that current floor planning tools do not allow static routing in the partial regions. This limited their number of achievable partial regions. As will be seen in the results from this work, the approach we have taken towards DPR does not suffer from similar limitations. Indeed, the studied application involved thirty two processing cores that could be individually and dynamically programmed into the device and dynamically reconfigured.

Krill et al [20] proposed a DPR design flow and demonstrated implementation of an Intellectual Property (IP) core for Color Space Conversion (CSC) for image processing. They proposed a DPR environment which they used to generate a CSC IP core optimized in terms of the area/speed ratio, providing both static and multiple reconfigurable areas. One significant difference between the current work and previous published results concerns the number and complexity of DPR regions. This work describes a framework for developing place-able DPR pre-placed macros and assigning placement of those macros, and additionally defines DPR interconnection and a novel reconfiguration task scheduling mechanism. Unlike this work the scheduling mechanism does not address the

actual scheduling of the data being processed during or the changes to the flow of the data which are a result of the reconfiguration process.

In [21], Clemente et al. developed a reconfiguration scheduler for implementing task-graphs at run-time, steering execution in the reconfigurable resources while performing pre-fetch and replacement, avoiding most of the reconfiguration delays. In their proposed scheduling environment, task-graphs are analyzed at design-time to extract useful information. This information requires simple computations at run-time to obtain optimized schedules. The authors developed a hardware implementation of the optimization techniques. The authors discuss efficiency for their evaluated scheduling protocol which manages a specified task-graph for required performance under enforced constraints. They schedule run-time reconfigurations only including one reconfiguration circuit and where reconfiguration latencies are assumed significant. As previously mentioned, this work describes a novel mechanism that provides for adaptive scheduling of reconfiguration tasks.

Consequently, while there remains substantial work to be done in regards to development of a DPR ecosystem that includes sophisticated tools to support machine state based process optimization as well as visualization and understanding tools for future highly concurrent, dynamically changing processing systems. Results from this work can be leveraged to grow existing techniques and tools to begin to create needed solutions.

1.3. Thesis statement

The main objective of this Ph.D. dissertation is to develop an effective framework for dynamically reconfigurable parallel processing. The target application for this

research is a video processing system. The thesis of this research is that a high-performance video processing system can be developed for real-time video streaming applications.

The effectiveness of this system is demonstrated in the high-performance achieved for video processing applications. In summary, the system characteristics include:

- Linux based dynamically reconfigurable system (DRS) for packet-based video processing
- Vendor development flow coupled with a Matlab and Simulink environment which facilitates DPR video processing applications
- Demonstration of the architecture in parallel processing of digital video data streams across a single channel

For reproducibility of the results, the design was implemented on a standard Xilinx FPGA PCIe development card that is widely available. The actual system described within this dissertation utilized a Xilinx Virtex 6 based ML605 development board.

1.4. Innovations and Contributions

The primary contributions of the dissertation include:

1. Partial Reconfiguration System for packet-based processing: This includes the development and implementation of a Linux based Dynamically Reconfigurable System (DRS) for packet-based video processing which facilitate DPR video processing applications and test fixtures.
2. Parallel processing of a digital video data stream across a channel passed into the FPGA fabric.

3. A reconfigurable packet processing bus arbitration engine used for processing digital video data streams.
4. Reconfigurable multi-core processing system that can be adjusted to different video processing sizes (e.g., 64x64 and 32x32).
5. Development of a custom Linux driver for packet processing and dynamic partial reconfiguration.
6. Reconfigurable Packet Processor Round Robin Controller.
7. High-performance video processing system that is demonstrated using a dynamic pixel processor and dynamic, separable 2-D FIR implementation.
8. Overall, optimal performance is achieved in the sense that video data is processed at the maximum possible rate that can be streamed through the processing cores.

1.5. Organization

This dissertation is organized as follows:

- Chapter 2 presents basic background on FPGAs, Dynamic Partial Reconfiguration and Packet processing frameworks.
- Chapter 3 presents motivation for the proposed dissertation research.
- Chapter 4 explains the hardware and software architectures of the implemented Dynamic Partial Reconfiguration systems.
- Chapter 5 presents the implementation results and discussion.
- Chapter 6 provides the conclusion for this dissertation and future work.

Chapter 2
Background

This chapter provides topologic descriptions of the custom DPR scalable systems that were developed and subsequently applied at the University of New Mexico for the purpose of exploring DPR architectures and architectural trade-offs, and a review of the current state of the art DPR approaches in signal, image and video processing.

The specific DPR topology configurations that were evaluated are described. To accomplish that required the development of a DPR development, evaluation, and test environment. That environment is described including features, architecture, and furthermore an analysis of achievable output precision resulting due to limitations associated with various scalable architecture nuances.

The underlying physical structure of the FPGA logic fabric leads to a natural two dimensional tiling problem associated with the assignment of specific logic elements to the actual Combinational Logic Blocks (CLB) that are designated by the specified design to execute that logic function. Ultimately, all of the logic functions that are to be executed by the FPGA must be assigned to a specific CLB which itself would have specific physical coordinates associated with it. From a bitstream programming viewpoint, tile location represents a specific programming address in an address space. The purpose of the software tools that compile abstract hardware descriptive code (typically developed in VHDL or Verilog) is to translate the abstract code into technology specific primitive logic library elements and then to map that logic to specific locations within the CLB fabric of the FPGA device.

This physical mapping is constrained by local and physical routing (wire interconnect) resources as well as the physical loading and signal time delays resulting from the length and impedance of those interconnects. Consequently, the optimized placement of each of those instantiated logic elements results in multiple simultaneous NP-complete optimization problems [22]. As a result, placed designs are never optimum but instead are sub-optimum but optimal in the sense that a finite set of specific design constraints have been met [23].

Therefore the design of logic for FPGA devices involves the simultaneous requirement to create a logically correct design while also meeting all required physical and timing constraints for the assigned programmed logic. A key difference between FPGA logic design and Application Specific Integrated Circuit (ASIC) design is that ASIC designs result in silicon mask sets which produce transistor based logic directly fabricated in silicon. The logic in a FPGA device can be moved if the device is reprogrammed. This section addresses the structured methodology for moving or altering the placement of logic in an FPGA fabric during the normal run-time operation of that device.

To address the idea of redefining the logic of an FPGA device during operation, it is essential to first provide a structure for defining how, when, and where logic within the functioning device can be modified. Consequently, it is useful to organize physical structuring of the DPR problem organized into high level topological arrays of resources interconnected through specific architectural topologies.

The next section details the set of specific architectural topologies that were explored in the course of this research and details relevant attributes and features for each.

2.1. DPR Resource Interconnection

The FPGA CLBs consist of logic implemented by lookup tables (LUTs) and which are immersed in an associated interconnection fabric. The interconnection fabric itself contains both local as well as global routing resources (wire and driver logic).

Hur et al [24] explored trading hardwired interconnect performance with use of programmable resources. They evaluated performance against configuration costs for soft and hard crossbar interconnection and observed that for their application that the hardwired interconnection achieved significantly higher performance. Thus, it is clear that the reconfiguration overhead can be significant.

An LUT with its configuration storage is clearly larger than the logic it will be programmed to implement. Likewise, the interconnect fabric consists of more gates than would be necessary with a direct connection between logic elements [25].

When a PR region is reconfigured, any associated functional interconnections likewise must be reconfigured. Historically, PR methodology would have required bus macros which would be geographically spread out to provide connection resources available to accommodate multiple module interconnectivity. Lacking pre-optimized resource allocation, use of soft programmable resources for general interconnection is not efficient because resulting interconnection resources are distributed over a sub-optimum larger surface area and consequently consume even more resources.

Furthermore, with specific predetermined wire delays, timing of soft networks are highly unpredictable with wide variance prior to placement and routing. Subsequently, it may be difficult to guarantee or even meet design-time timing constraints due to this unpredictable net delay skew. In the case of PR circuits that are also adaptively changed, managing both the complexity of co-optimization of functional adaptation as well as interconnection synthesis is intractable without imposing structural constraint on allowable combinations.

To compose complex designs in either ASIC and FPGA technologies, it is common to use predesigned libraries, macro functions, or IP building blocks. As with other design paradigms it is common to develop building blocks for general use conformal to standards. Never-the-less compatibility remains a challenge. IP products address specific functional needs and corresponding interfaces are likewise functionally specific. As a result integration with other functions requires translational logic with alternative mappings for each required interface variant. It is for this and other reasons that standardized chip level interconnect strategies and associated IP have emerged to provide structured options.

One such chip level interconnection standard is the Advanced Microcontroller Bus Architecture (AMBA) from Arm [25] developed for ARM processors. ARM Holdings, PLC (UK) [26] licenses silicon IP for a wide range of custom ASICs and standard microcontroller and microprocessor products including some later generation FPGAs. AMBA 4.0 defines five interfaces:

- Advanced eXtensible Interface (AXI)
- Advanced High-performance Bus (AHB)

- Advanced System Bus (ASB)
- Advanced Peripheral Bus (APB)
- Advanced Trace Bus (ATB)

The Advanced eXtensible Interface (AXI) specifically targets high speed designs. AXI is a master/slave interface that is extensible using a switch or fabric. AXI provides separate address/control and data phases to support non-aligned byte-wide data transfers. The bus standard accommodates multi-address burst transactions for efficient bus utilization. The bus specification is flexible providing multiple levels of performance and complexity allowing one or more bus masters.

AXI-Lite provides a lightweight version when full bus capability is not needed. AXI Coherency Extensions (ACE) provides memory cache coherency support. AHB was introduced with AMBA-2 with AHB-Lite added in AMBA-3. AHB provides a single-edge clock protocol supporting multiple bus masters, split transactions, burst transfers, and single-cycle bus master handover. AHB is a non-tristate, pipelined architecture with bus widths up to 128 bits. AHB-Lite is a reduced subset for single master systems. Finally, APB connects to AHB for low bandwidth and register interfaces for peripherals like serial ports. It is similar to AHB but is less complex; requiring fewer resources to implement.

The research associated with the current Ph.D. dissertation did not integrate a standardized interface such as AMBA, but given that the Xilinx Zync family incorporates two ARM processors, integration of this and/or another similar chip-level interconnection standard logically represents a next step in increasing the capability and flexibility of the design framework. The general approach taken to evaluating interconnection

architectures is described below. Bus or switch connected architectures are more suitable for adaptation to standards including AMBA but lattice structures are more suited to pipelined data flows architectures and structurally more closely resemble concurrent filter structures, the application subject of this research.

In addition to the physical and logical issues associated with interconnection, equally important is the physical arrangement and associations of DDPR regions. Consider an FPGA design developed to support DPR where reconfiguration is performed on a by-physical-region basis. To accomplish this, the FPGA fabric is partitioned into a finite set of mutually exclusive physical regions which we generally refer herein to as DPR regions. An example of a regular partitioning of the fabric from the results can be seen in Figure 15 as well as in Figure 16 later on in this work.

A DPR design with multiple simultaneously running DPR regions can be split across a single FPGA fabric and/or across multiple FPGAs. The specific manner in which this would be accomplished would depend on system processing requirements constrained by the finite available interconnection resources between regions. Interconnection resource allocation could either be optimally tuned for a specific application class, or alternately provisioned with sufficient resources to provide flexibility to accommodate a broader range of applications. Once utilizing DPR techniques, even interconnection resource provisioning can be reconfigured such that the system can be conditionally optimized for a particular class of processing techniques depending on its utilization environment. Figure 1 as follows depicts several common interconnection topology schemes that could be considered.

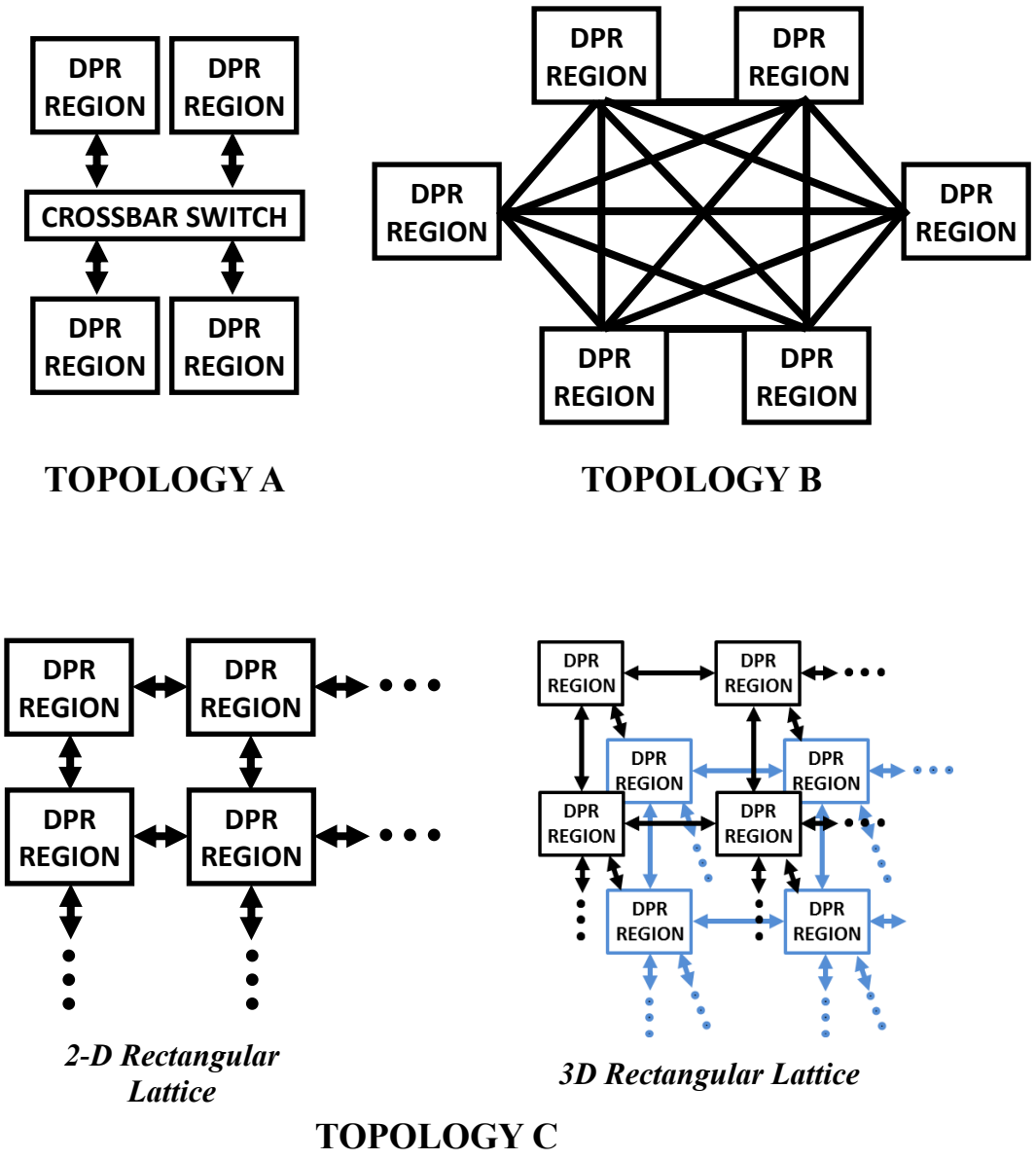

Figure 1. Researched DPR Region Topologies A to C

Topology *A* represents a crossbar style interconnection scheme. Each pair of DPR regions is connected through a central crossbar switch. This crossbar switch provides selectable direct interconnections and consequently allowing each individual DPR region to directly communicate with any other DPR region. The realization of this crossbar switch itself conceptually could be realized by a DPR region in a device. Alternatively a

specialized crossbar switching device or yet another FPGA device dedicated for routing could be used to create this connection capability.

There are various well known forms and architectures of crossbar switches and the properties of those are very well established. Example crossbar realizations can be found in many development systems [26] [27] [28] [29] [30] [31] [32]. In Figure 2, we show an application of this topology that was studied within the Splash 2 [33] development system.

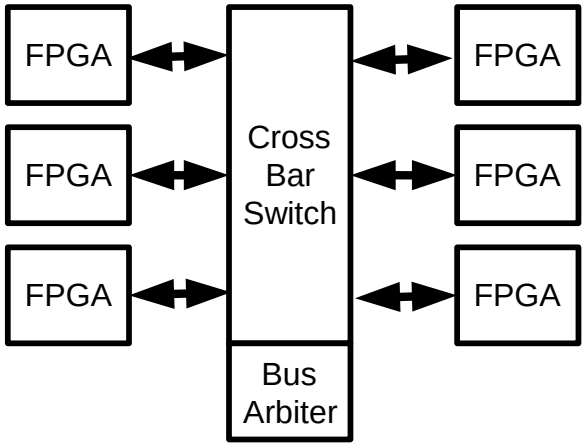

Figure 2. Splash 2 Interconnection Architecture

The Splash 2 architecture consists of sixteen (16) FPGAs individually connected to a common central crossbar. Switch connections between the DPRs are programmable and controlled by an additional seventeenth FPGA device. This architecture can be applied to DPR regions with either intra or external communication requirements. The advantage that DPR has in this type of architecture is the interconnect can remain static during multiple DPR configurations and by updating only the Bus Arbitration scheme to match the processing requirements the system can be optimized.

A fully connected interconnect topology can be seen in Topology *B* where all of the DPR regions connect to each other. This topology has the advantage of removing the crossbar latency from interconnections but the number of connections to each region grows as $N!$ where N is the number of fully interconnected regions.

Both Topologies *A* and *B* have each FPGA connected to every other so they represent the most flexible topologies [34] [35] [36]. However, this flexibility comes at a cost. The ability to scale the number of region interconnections is nominally a linear function of the region boundary length, L. Hence the number of viable nodes N that can be interconnected in this manner is bound by a relationship of form $O(N!) < \alpha \cdot f(L)$ where α is some finite constant. Moreover, intra-region information flow and control complexity for each independent region would be anticipated to grow $O(N!^2)$ whereas the region area would be anticipated to be $o(L^2)$. Thus this imposes an even more restrictive asymptotic bound of $O(N!^2) < \beta \cdot f(L)^2$ for some finite constant β. Conversely, as the number of regions increases in a system, the number of possible connections required grows with $O(N!)$. Since routing resources are finitely bound, the average number of available connections between any two regions consequently must be reduced.

The inability to scale combined with necessity to develop DPR macros that are themselves individually dependent on N and consequently topology dependent. This means that DPR region must be developed in a system-dependent manner. A fully interconnected topology increases system power and heat dissipation due to the total number of wire and drivers comprising the interconnection resources required as well as increasing interconnection control and flow logic.

2-D or 3-D mesh (or lattice) interconnection is represented by Topology *C*. The bottom of Figure 1 depicts rectangular 2-D or 3-D lattices. Interconnect routing for 2-D or 3-D lattices are either direct node to node interconnected or otherwise pass-through routed. These topologies allow a given DPR region to communicate directly with its neighbors. However, communication non-neighboring regions must be relayed through an intermediate region. Currently this type of topology is not realistic in DPR systems because of the large and complex nature of the communication requirements between nodes. However, with newer stacked die devices like the Virtex 7-2000T, which comprises of four of the largest FPGAs in the 7 Series devices on a Stacked Interposer, a hybrid of this type of interconnect, may be adapted in the near future. Examples of systems that use 2-D meshes are [37] [38] and systems that use a 3-D mesh are [39].

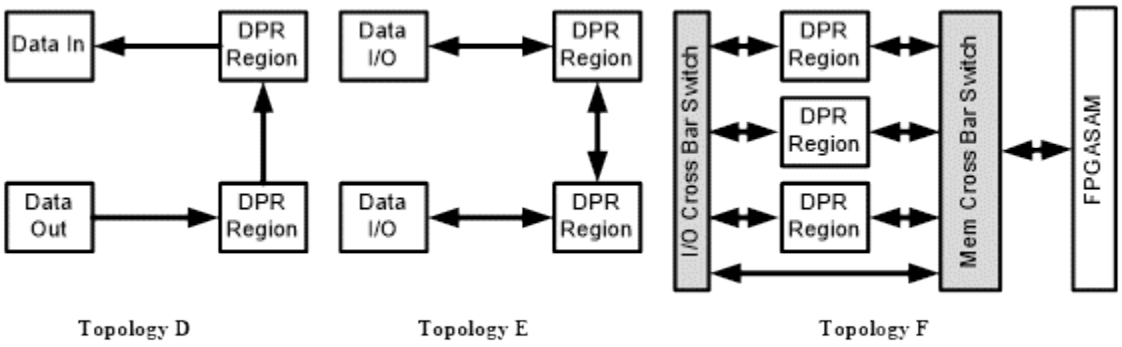

Figure 3. Researched DPR Region Topologies D to F

In systems where the algorithm is designed to be ran in a pipelined manner, for which a series of independent algorithms are applied, such as encryption and decryption of data. A linear interconnection scheme between DPR regions comprising of the stages, can efficiently implement pipelined applications. In this type of application data is sent into the system from one end of the chain for each DPR region to process in order. The result is then output from the last FPGA. One feature of this architecture, seen in

topology *E*, is (if designed to do so) for problems such as encryption/decryption, is that the data can be fed into the stages in one direction for encryption and in the reverse order for decryption.

A hybrid topology of the cross bar and pipeline architecture is the shared access memory topology. This topology allows the architecture to take advantage of a wide variety of algorithmic optimizations and still maintain the ability to share information between DPR regions. An adaptation to this topology can be seen in Topology *F* where the I/O Cross Bar can directly load the independent DPR regions and directly access the shared access memory through the memory cross bar. One of the primary concerns with this type of topology is both the I/O and memory cross bar must communicate with one another else data corruption can occur. For this architecture to be the most efficient, the bus between the memory cross bar should be scaled such that data through put is maintained. This type of architecture has been used in multi node single instruction multiple data applications. One example of this can be found in [40] the Texas Memory Systems S4 Vector Processing system.

The realized architecture can be best described as a layered hybrid interconnection architecture where the video data distribution and routing layer is represented by a crossbar connected network as discussed previously. The next section details the numerous possible complexities and tradeoffs that are associated with packet-switched architectures and additionally explains a novel approach to scheduling with specific details provided in Section 3.3.1.2.

However, it additionally should be understood that underlying that layer, at the DPR fabric layer, Filter Processing Cores can be viewed as a 2-D lattice network. This is described in 3.2.2 and depicted in Figure 15.

2.2. Data Switching and Packet Routing

Because this work focuses on packet-based processing and involves distributed queuing and routing of data packets, it is essential to understand the architecture in terms of its characteristics related to packet switch technologies and techniques. A significant result from this research was to propose and develop a novel approach to data queue allocation and scheduling using a novel Adaptive Weighted Round Robin (AWRR) scheduling protocol that is optimized for the adaptive video FPGA DPR application described by this work. The implementation of the AWRR protocol is detailed in Section 3.3.1.2.

There are many possible technologies, topologies and architectural approaches to provide data flow mechanisms in a system. When data flows require aggregation or de-aggregation, those flow points become shared resources for data producers or consumers. Historically, the primary means for parallel accessibility to shared common resources involves either a common exchange media (such as a bus or other common, arbitrated media) or a switched network. Bus architectures for processing applications have largely dominated all but the very high-end applications for the last several decades.

System level performance is bounded by interconnectivity capabilities and capacities. However, there are increasingly challenging physical bounds on achievable speed and bandwidth of bussed technologies. Combined with the emergence of high-

speed serial interconnect technologies, this has resulted in increased emphasis on switching architectures.

A significant attribute and potential benefit of switch based architectures is the concentration of connection management complexity together into the control for that switch fabric. For a single, centralized switch, the majority of the complexity of system interconnection is concentrated in that switch. Consequently, viable solutions must be compact and efficient [41].

When considering the general topic of switch technologies, there are a few general categories that are widely discussed in the literature. These include:

- Cell Based Switching
- Packet Based Switching
- Time Division Multiplexing (TDM)
- Circuit Switching
- Connection/Connection-less Switching
- Dynamic path routing
- Memory-less Crossbar Switch

Especially when examining complex or distributed switching systems, often the system can be functionally decomposed into multiple layers of switching that are themselves constructed from one of the other categories of switches. Thus the higher level represents an abstraction of a service that is provided by the lower layer of the switch fabric. Consequently, to really understand switching technology, it is helpful to first understand the primitive constructs that are common to switch architectures.

Data that flows through a system must be buffered at any point associated with delay. Delay occurs due to competition for use of a shared element or path through the switch. For a switched network, there are three general forms of buffering. Buffering can occur at the input to the switch, at the center of the switch, or the output of the switch. This concept is illustrated in Figure 4 below.

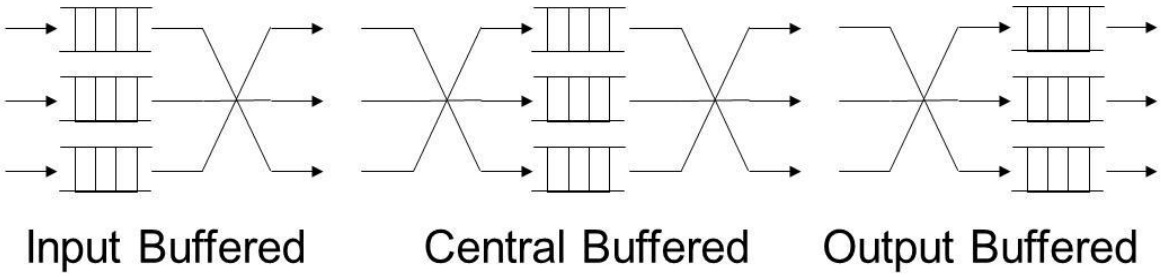

Figure 4. Switch Buffer Architectures [41]

For purpose of this work, we explain some of the primary considerations involved with architectural aspects of designing the queuing structures for packet switching and we will explain the corresponding considerations and issues that contribute to the architectural choices made during this work.

The problem of understanding switch data flow complexity can be discussed in terms of the allocation and use of the data buffering used to provide the queuing structures in the switch. Understanding switch behavior (capability, capacity, and limits) can be discussed in terms of the queuing mechanisms. Consequently, it is helpful to consider the types of queues that can be constructed. For instance, queues can be allocated with fixed memory resources or can draw their memory resources from a pool of memory buffers. This pool of buffers could either be made up of fixed-size blocks that can be linked and unlinked together to provide the memory for the queue, or a contiguous

block of memory could be allocated. Either case requires a mechanism to organize the pool, to allocate and de-allocate use of memory, and also to manage cleanup if irregular sized blocks of memory are permitted.

As discussed previously, queues can be associated with inbound traffic, outbound traffic, or centrally located and consequently associated with a combination of inbound or outbound traffic. Queues can be used for data collection from inbound traffic. They can be used to buffer data as a Transmission Queues for outbound traffic. Queues can be used to replicate traffic to provide multi-cast or broadcast service. They also can be used to organize, prioritize, sort, or reorder traffic.

Unless queues are assigned fixed memory resources, they must share them. Fixed assignment can mean permanent or hardwired allocation or it can simply mean statically fixed (meaning that resources can later be re-allocated). If resources are sharable and not statically allocated, then it is necessary to consider different protocols for queue provisioning. Allocation and management can occur on a per-path, per flow, or per traffic class/sub-class basis, or even user defined. Sharing can involve demand-driven, dynamic allocation, or statically-based allocation.

Shared resources consequently involve considerations including thresholds and limitations. When differentiation occurs, it can be on a per-path, flow, or class basis. Algorithms can be applied to shape the traffic flows, or applied to encourage resource utilization, such as the case with “greedy flow” algorithms.

Looking at an individual queue, a first consideration is when it is possible to simply put data into the queue. The queue is intended to be the place where data delay can be accommodated and consequently it is assumed that data cannot wait on queue

availability. However, data is not always available to place in a queue or else there would not be a rate matching or delay need to begin with. Queues are typically designed to underflow their input capacity, at least slightly. However, because the queue represents a finite memory resource, only a finite amount of data can exist in the queue at any moment. Hence, once the queue fills, data must either be overwritten or else input discarded. There are various protocols for queue admission management including [41]:

- Drop Tail
    - Packets are dropped from end if queue fills
    - Lost packets are an arbitrary result of resource over-allocation
- Random Early Discard (RED)
    - Packets discarded when queue depth exceeds threshold
    - Probability of drop increases as depth grows
    - When queue fills acts like drop tail
- Weighted Random Early Discard (WRED)
    - Drop probability is class dependent
- Drop Head
    - Not frequently used
    - Prioritizes new packets over old
- Class based versus flow based policies

Queues are "serviced" when data is read (removed) from the queue. For output buffered queues, switching control mechanisms must control physical path connectivity to the queue input port such that the port is guaranteed available whenever data arrives. One physical mechanism often used to provide that guarantee is to use a multiport input

such that there is always and available port. For output buffered switches, output buffers are serviced by the corresponding data consumer per the rate and service mechanism for that specific consumer. In contrast, for input-buffered switches, the actual act of switching occurs through the process of selecting which source queue is to be serviced next. Commonly used queue servicing or scheduling mechanisms include:

- FIFO (First-In-First-Out) or FILO (First-In-Last-Out)
    - Queue service is based on packet arrival time
- Periodic versus Aperiodic; Virtual Clock or Bounded Time
- Weighting - selection prioritization based on queue depth (un-serviced content)
    - Per packet
    - Per word
- Flow / Class differentiation
- Prioritization
    - Strict Prioritization
    - Round Robin / Weighted Round Robin / Deficit Round Robin
- Hybridization

As will be seen in later sections, individual data flows incorporate FIFO queue structures. However, there are multiple service scheduling mechanisms that occur concurrently. At the highest level of task process coordination, this work describes an adaptive variation of a Weighted Round Robin (WRR) scheduler.

Of course there are many possible tradeoffs involved in selection of queue structures, queue use policies, and queue service policies [41]. The current application drove selection of a set of structures and techniques. As the DPR framework evolves, it

will prove necessary to provide a wider range of scheduling mechanisms in order to optimize resource use for other anticipated flow traffic and processing algorithm characteristics.

Commonly considered tradeoffs include:

- Algorithm Complexity. In this research, the desired types of processing were mostly known a priori.
- Memory cost, organization, and provisioning. In the Ph.D. dissertation, performance is optimized to match fixed resources
- Fairness. Fairness is a concern when there a multiple flows with differing priorities and variation in traffic statistics. Fairness allows all consumers some amount of resource utilization. This is a more sophisticated need than required for this Ph.D. dissertation.
- Flow isolation or protection. When processing multiple video flows, it may be desirable to preclude variable processing in one flow to adversely impact the ability of another to meet service guarantees.
- Efficiency:
    - Temporary over-subscription. It is desirable in some cases to permit over-subscription of resources to occur, especially if back pressure or other means can be applied to provide overall elasticity
    - Permanent over-subscription. This work is “virtually” permanently over-subscribed and adaptively adjusts resources to maximize the overall efficiency, fully using available resources

The design of a full featured framework for packet processing DPR is very analogous to design of a packet routing switch fabric. Indeed, the data distribution and routing of application data is a packet switching problem. Overlaid to that is the hardware virtualization of logic resources. These resources must be scheduled for FPGA reconfiguration at that time where they are needed for processing and then replaced with other functions. Thus the total footprint of available logic functions can greatly exceed the instantaneous logic actually present in the device at a given time.

One of the significant achievements of this work was achieving the DPR reconfiguration programming over a multi-lane high speed serial interface consequently leading to region reconfiguration turn-around times that are of an order similar to task processing times.

2.3. Round Robin / Weighted Round Robin / Deficit Round Robin

Because of the limitations of strict priority protocols an alternative approach commonly referred to as “Round Robin” scheduling is popular. In a simple round robin protocol, queues are organized in a linear ordered manner and each queue is serviced in order (if data available) until all have been serviced at which point the scheduling mechanism goes back to the first of the list [41]. Scheduling essentially begins examining the service request state of the first queue in the list. If service is requested, then a packet is removed from the queue, otherwise the queue pointer moves to the next in the list and the process repeats until all queues have been visited, in which case the pointer goes back to the start of the list. Each queue serviced has exactly one packet removed from the queue.

Simple Round Robin does not provide a capability to differentiate among queues with high traffic rates. Since each queue essentially gets equal treatment, then data arrivals in low data rate queues are more likely to be immediately serviced. Arrivals in relatively high data rate queues therefore experience congestion and backup with corresponding increase in average latency.

To overcome this limitation of Simple Round Robin, the idea of Weighted Round Robin is to apply a weighting factor to the selection process. The weight determines the average number of packets to service from each non-empty queue. Therefore, the number of packets to service would be the weight divided by an average packet size.

A limitation of Weighted Round Robin is that the algorithm is not well suited when there does not exist any a priori knowledge of packet sizes, packet sizes have widely varying sizes, ill-suited statistical characteristics, packet sizes are unpredictable, or change over time. Shreedhar and Varghese [42]originally proposed the idea of Deficit Round Robin (DRR) to overcome associated issues. In DRR, a threshold size is used to hold back first-in-line packets whose length exceeds this threshold. This threshold value is used as a deficit counter for the queue and is increased by a default deficit amount for the next schedule time. If the size still exceeds the current count (also referred to as the credit), the count is again increased and this continues until the packet is serviced. Whenever a packet is serviced from a DRR queue, its credit is decreased by the packet size.

2.3.1. Queue Service Prioritization Policies

Among the most commonly used methods for queue selection for service are those policies that are priority-driven. A strict prioritization protocol assigns a linear priority

to each possible selection and whenever the highest priority queue demands service, it is serviced prior to any other queue [41]. If that queue is empty then the service goes to the next highest priority queue that requires service. Strict prioritization guarantees that the highest priority data is always serviced first. The shortcoming is that if the system flows and priorities are not carefully considered, it is possible that lower priority traffic can experience extensive delays, might be discarded, or may not be guaranteed to ever be serviced.

A strict priority protocol can be augmented with additional rules that either promote/demote data traffic priorities, or allow limited amounts of low priority data to be serviced. The later rule is commonly referred to as a leaky bucket protocol [41].

2.4. State of the Art for Reconfigurable Computing

In today's systems, reducing system size, weight, and power (SWaP) is critical for operational life and budgetary constraints. SWaP budgets are driven down to extend battery life, reduce form factor for better mobility and logistics, and expand the market. In platforms such as cell phones, SWaP constraints are different because there is a need to support fast embedded processors for the increasing image and video processing needs while still staying within the SWaP budget.

Reconfigurable Computing (RC) has been identified as the solution to provide substantial performance benefits while maintaining the tight SWaP budgets compared to traditional microprocessors or GPUs [43]. The Dynamic Instruction Set Computer (DISC), first proposed by Wirthlin and Hutchings [44] was one method introduce to address this issue by using a medium grain Configurable Logic ArraY (CLAY) device from National semiconductors. This computer had an instruction set made up of

independent hardware units configured into the device as needed. The architecture's performance was limited by the reconfiguration time overhead and the number and size of the processors was fixed.

Other research has been concerned with detection of bottle necks in RC. These include High-level synthesis tools increase FPGA productivity [45], Characterization and Optimization of Atomic Operations in GPUs [46] and Integrated Development Toolset and Implementation Methodology for Partially Reconfigurable System-on-Chips [47]. These systems have been primarily focused on RC development systems and not creating a low SWaP framework to apply general or specific algorithms.

For RC, one of the primary areas of interest is reducing the Data Movement this has been looked at in both the reconfiguration time, data dependency between tasks, inter-task communication as well as task resource utilization [48]. However, in addressing these scheduling issues, the authors did not consider dynamically reconfigurable systems.

A large amount of research has also been associated with the use of multiple chips, ranging from the use of processors, DSPs and/or FPGAs including Multi-Level Performance Prediction of Multi-FPGA Systems, Multilevel-PGAS Programming Model for Reconfigurable Supercomputing [49] and Bridging Parallel and Reconfigurable Computing with Multilevel PGAS and SHMEM [50] [51]. These approaches addressed issues associated with working with multiple devices and modeling the data flow through these parallel systems. However, these prior approaches fall short of looking at the issues of data movement and reconfiguration of multiple RC engines in a single heterogeneous low SWaP architecture, as done in this dissertation research.

2.4.1. Embedded Reconfigurable Computing

Dynamic Partial Reconfiguration (DPR) refers to an on-going process reconfiguring portions of an FPGA during run time while other portions remain active. Research continues towards developing more useful methodologies that can realize the potential utility of FPGA DPR [18]. The existing literature speaks to the considerable potential for fundamental change in flexibility, effectiveness and efficiency. Perhaps the most important advantage afforded by DPR is an ability to employ smaller or few FPGAs reducing cost, packaging, and power [15]. Capabilities for PR have existed for several years, yet the design methodologies, flows, and tool capabilities capable of effectively using PR remain limited. Existing methodologies for PR previously have been largely ad hoc, cumbersome, or poorly integrated into the regular design flow [18].

Fons [11] however, noted that the first commercial products/systems driven by PR technologies are beginning to enter the marketplace. In his dissertation, the author provides an introduction into the history of reconfigurable computational technologies. The author also provides an extensive and detailed explanation of the differing FPGA technologies. Most relevant to this work, the author discusses the different forms of PR technologies (including those from Xilinx.) The author discusses PR in terms of context planes for FPGA reconfiguration defined as planes of programmable resources. FPGAs are envisioned as reconfiguration capabilities multiplexed in a manner that only one of the configuration planes is active at a given time. Consequently, creating a multi-context FPGA requires multiple memory bits for each programming bit location. These memory bits would be used to define the multiple planes of configuration information. The model allows for background loading of one of the contexts while another is active and

executing. In this model, one configuration plane is active at a given time and the device switches among alternate context of already programmed configurations. The author provides extensive information on the body of work accomplished and his published work serves as a good reference to understand DPR technologies.

With dynamically and partially reconfigurable designs, it is necessary that the speed of the reconfiguration be accomplished in a time that is sufficiently small such that the operation of reconfiguration is not the limiting factor in the process. This was accomplished by designing and implementing a High Speed Dynamic Partial Reconfiguration Controller (HSDPRC) core that maximized the bandwidth of the ICAP. The work used on chip temperature and voltage active feedback for Dynamic Partial Reconfiguration. This approach created a DPR controller that is tightly coupled with the FPGA silicon and the system behavior than previous work had not accomplished [4].

There is strong interest in the development of dynamically reconfigurable systems that can meet real-time constraints in energy/power-performance-accuracy (EPA/PPA). Recently, in his Ph.D. dissertation research, Dr. Llamocca described a research framework for implementing dynamically reconfigurable digital signal, image, and video processing systems. However, the proposed architecture was not scalable since it did not use a video packet based approach for dealing with system scaling issues [52] [53] [54] [55]. Additionally, this research was built on an embedded processing system where a processor running in the fabric moved the data in and out of the processing cores.

The proposed dissertation architecture provides a scalable architecture for sustaining high-performance in multi-core, dynamically reconfigurable systems.

Moreover, the run-time adaptability of the scheduling protocol described here is uniquely targeted towards dynamically reconfigurable applications.

2.4.2. FPGA Reconfiguration Methodology

Programming the soft logic of an FPGA is significantly different than writing software for a processor. Low level design is typically accomplished in Register Transfer Level (RTL) Verilog or VHDL and is similar to ASIC design. The primary difference is that RTL for FPGA design includes additional limitations on language use to guide proper circuit inference within the logic synthesis tools. In addition to RTL design, there is dramatic increase in design complexity.

Over the previous decade, high-level FPGA and ASIC design languages have evolved (e.g., SystemC). The popularity of SystemC is due to the fact that it is based on the C programming language. On the other hand, this advantage is not important since the development of highly concurrent FPGA logic is completely different than writing software applications for a von Neumann architecture processor.

Xilinx's Vivado provides direct support for AXI-based components adding required interconnect logic automatically [25]. Perhaps one challenge of standards based interconnection design is the degree of extensibility. The tool environment needs to provide designers with the ability to both customize to their need while also providing a range of automatic optimizations. It remains very challenging to provide tool flows that accommodate developers who want the flexibility and power of an FPGA but whose focus and understanding of applications is primarily based on software application programming. Reconciling these two views of computation within a single programming and tools environment will remain a challenging problem.

This research demonstrates both the need to establish even better programming environments as well as explaining why it is essential for DPR to establish a structured programming framework to guide DPR application development. Furthermore, important results are explained in our result for tenants of framework definition. This subject is discussed in the next section.

2.4.3. **Frameworks for FPGA Reconfigurable Computing**

Networks on a chip have been looked at for rapidly moving data efficiently across multiple processing nodes and various topologies for these architectures have been proposed [56] [57].

These include mesh, torus, ring, butterfly, octagon and irregular interconnection networks. Various researchers have used telecommunication type of interconnection structures for example a star-based network was used that communicated using the principle of CDMA (Code Division Multiple Access) [58] and others have proposed a tree-based implementation [59], where each node in the tree behaves as a router in a network. The primary issue with these approaches is that they are focused on routing packets across vast distances with specific hardware to move the packets from processing node to node. In FPGAs, this is not an option due to the limited ability to add large routing tables and controllers to the reconfigurable matrix.

The Reconfigurable Round Robin Controller (RRRC) described in this dissertation takes a different approach. For the RRRC, the configuration of the fabric matches the configuration of the fabric data flow engine. This approach optimizes both the fabric resources used by the routing engine and the data flow through the framework.

Tan and DeMara [15] evaluated methodology tradeoffs for Partial Reconfiguration using the Xilinx Virtex II Pro FPGA technology node. The authors discuss significant challenges that existed for dynamic reconfiguration for the Virtex II node. Accordingly, some of their results do not directly apply to current generation devices, but general observations are relevant. They observed that a considerable number of open research issues existed then although this work addresses some open issues, many others will remain. Now as then, there are no sophisticated tools or tool flows that support a general partial reconfiguration strategy. This is due, in part, to the complexity and diversity of aspects. They discuss methods for resource relocation using PR techniques. The authors discuss techniques that include bitstream manipulation, but the FPGA manufacture's bitstream generation represent important intellectual property and not consequently not available for open research. As Tan and DeMara point out, it was possible to reverse engineer Virtex II bitstream file and directly manipulate them, but that is no longer a reasonable consideration.

Run -time capabilities must address resource allocation and task scheduling, and logic flow control. Indeed, the authors stress a need for hierarchical encapsulation such that layered control would be self-contained and ideally exposing a minimalist pre-defined interface to other layers of the hierarchy. This property is essential to minimize potential side effects that might occur due to modifications to one layer that in turn influence behavior in other layers in unanticipated manners.

Design automation tools are the enablers for efficient exploitation of reconfigurable capabilities for the construction of adaptive, self-adaptive and self-healing systems. In the recent past, developing reconfigurable systems had been "performed only through heroic

acts [11] ” due to the clear lack of sophisticated tools. The quality of currently available tools can be “catalogued as valid to exploit” available technology. The problem is not well solved. While the “latest PR tools released by Xilinx have dramatically improved usability. However, it is admitted there is a gap for improvement that is still necessary to cover in order to boost all the potential of run-time reconfigurable hardware technology. The research community, aware of the fact that the lack of suitable design tools can slow the ramp up of this technology, is working hard on this topic [11].”

2.4.4. **Multi-Soft-Core Reconfigurable Computing**

Clemente et al [21] developed a reconfiguration scheduler for implementing task-graphs at run-time, steering execution in the reconfigurable resources while performing pre-fetch and replacement, hiding most of the reconfiguration delays. In their proposed scheduling environment, task-graphs are analyzed at design-time to extract useful information. This information requires simple computations at run-time to obtain optimized schedules. Authors developed a hardware implementation of the optimization techniques. The authors discuss efficiency for their evaluated scheduling protocol which manages a specified task-graph for required performance under enforced constraints. They schedule run-time reconfigurations only including one reconfiguration circuit and where reconfiguration latencies are assumed significant.

NASA has also looked at Reconfigurable Computing Multi-soft-core frameworks in Space applications [60]. The goal of this research was to provide enhance the availability of reconfigurable radiation tolerant Field Programmable Gate Arrays (FPGAs) for space applications. In this research a high-level architecture for a reconfigurable computing system for space was outlined and a multi-processor

architecture using soft core processors with operating system and message passing support was coupled with the capability to utilize custom cores for specialized processing. This research also addressed the challenges of the space environment through fault mitigation techniques, parts selection, and physical design.

While this research provided significant forward progress in the area of Multi-soft-core Reconfigurable Computing it did not show how the ability to tune the frame work dynamically can produce a more refined method to data movement into and out of a system. In addition this research was pointed at a specific space application of the multi-soft-core framework where the primary focus was to mitigate Singe Event Upsets (SEUs) rather that performance and power optimization.

2.4.5. Video Processing and Reconfigurable Computing

Due to the inherent processing attributes with video applications and their ever increasing need for parallel processing more data and required adaptability, video processing applications are ideal for further development. Recent research is now showing the value of dynamic partial reconfiguration of the FPGA fabric for embedded video processing applications [61].

Fons [11] examined a number of different applications including filtering and pixel processing and previously had described a DPR technique applied to 2-D convolution for adaptive image processing [62]. Sudarsanam et al [18] evaluated polymorphic Systolic Array (SA) implementations in FPGA technology for filtering regular and synchronized data. Specifically they were interested in Extended Kalman Filters (EKF) and convolution kernel-based Discrete Wavelet Transforms (DWT). The DWT consisted of one-dimensional (1-D) filters separable among horizontal and vertical directions, which

is then followed by sub-sampling. The image processing algorithm is typically iterated multiple times in a process referred to as decomposition which results in a data structure that can be efficiently compressed. More to the point, algorithms including EKF and DWT involve structured mathematic operations such as matrix addition, multiplication, and inversion and hence are well suited for efficient acceleration on hardware systolic arrays. Accordingly, their approach involved mapping algorithms into a linear systolic array regular array of Partial Reconfiguration (PR) regions. A Xilinx Virtex4 SX35 FPGA was used for their study and performance benchmark results were compared to software results from a PowerPC 750 system, with observed achievable speed-up of 4x to 6x.

As mentioned previously, Krill et al. [20] work included recommendations for a DPR design flow. Their flow was used to implement an image processing macro core. Their approach provided flexible DPR area/size placement with standardized interfaces and proposed a DPR environment to generate optimized IP cores in terms of the area/speed ratio, thus allowing for both static and reconfigurable areas. Their approach resulted in a custom design; however, they also proposed a framework requiring user provided DPR modules for FPGA implementation.

Some of the most promising research is in the area of Dynamically Reconfigurable Management for energy, performance, and accuracy and where it applies to Digital Signal, Image, and Video Processing Applications [63]. In this research a framework was presented for the generation of optimal implementations from the Power/Energy, Performance, and Accuracy (PPA/EPA) space based on PPA/EPA constraints. The framework allows for dynamic PPA/EPA management for digital signal,

image, and video processing applications [64] [7] [53]. Implementation of this scheduling protocol is described in Section 4. Accordingly, this work describes a novel and powerful Adaptive Weighted Round Robin (AWRR) scheduler for run-time adaptive optimization of FPGA DPR reconfiguration operations.

General problems that are associated with the temporal allocation and utilization of resources under scheduling constraints have been widely studied within statistical queuing theory and also within various aspects of embedded computing and telecommunications. Obviously there are a number of scheduling protocols that could be applied for the purpose of scheduling PR operations. Application and system use constraints guide a system architect to select techniques that meet specific optimization needs or goals. This topic is expanded on in greater detail in Section 4 to provide sufficient background to explain the rationale for our implementation of an AWRR scheduler, explain the novelty of the implemented mechanism, as well as to explain why this scheduling mechanism is particularly powerful for our application.

2.4.6. Description of FPGA Cores

The following is a description of the two video processing cores used to test the framework a (i) high performance 1-D FIR filter based on distributed arithmetic [6] [63] and (ii) pixel processor. The both cores were coded in VHDL, so as to achieve a level of portability. Logic wrappers were added to both cores so that they could be integrated into the frame work. A description of the logic wrappers for each core can be found in (Implantation and Results) Chapter 5.

2.4.6.1. FIR Filter Core

The 1-D FIR filter VHDL core used as the basis of the 1-D and 2-D FIR filter implementations in this work was originally developed and used for ivPCL's paper on *Partial Reconfigurable FIR Filtering System Using Distributed Arithmetic* [65], and later in Dr. Llamocca's dissertation on *Dynamically Reconfigurable Management of Energy, Performance, and Accuracy applied to Digital Signal, Image, and Video Processing Applications* [64]. The FIR filter cores are configured to perform a Gaussian filter operation on the video data.

The 1-D FIR core was developed using Distributed Arithmetic (DA) combined with a look-up table (LUT) reduction technique which allows the direct mapping to reconfigurable LUTs of Xilinx and Altera FPGAs. An FIR filters' fundamental operation with *N* taps is the inner product of two vectors. Below is the mathematical function which is a sum-of-products of a FIR filters components.

$$y = C * X = \sum_{n=0}^{N-1} C_n * X_n. \quad (1)$$

In this function C_n are typically constants and X_n are the time-shifted input samples. If each X_n is represented as a binary *Bx* bit 2s complement number, where $X_{n,b}$ denotes the *b*th bit of X_n. Expanding the equation produces gives:

$$y = \sum_{n=0}^{N-1} C_n \sum_{b=0}^{B_X-2} 2^b X_{n,b} - 2^{B_x-1} X_n, B_X - 1. \quad (2)$$

and

$$y = \sum_{b=0}^{B_X-2} 2^b \sum_{n=0}^{N-1} C_n X_{n,b} - 2^{B_x-1} \sum_{n=0}^{N-1} C_n X_{n,} B_x - 1. \quad (3)$$

Where $f(X_b^N) = \left(X_{0,b}, \ldots, X_{N-1,b}\right)^T$ is the bitvector of length *N* containing the *b*th bit of each element of *X*. Finally, below is the function that is precompiled in the DA core and stored in a LUT with *N* inputs.

$$f(x_b^N) = \sum_{n=0}^{N-1} C_n X_{n,} b. \quad (4)$$

A detailed description of the core and its implementation can be found in [63] [6]. The Table 1 is a description of the filters I/O and VHDL parameters that are used to connect to the packet processing framework.

Table 1: FIR Filter I/Os

I/O	Description
N	The number of taps
NH	The input/coefficients bit width
L	The LUT input size
OP	Used for controlling the output truncation scheme: (i) LSB Truncation then Saturation (ii) LSB and MSB Truncation (iii) No Truncation.
NO NQ	Are used to denote the fixed-point output format for *NO* bits with *NQ* fractional bits.

For the VHDL implementation parameter variables used through this work were set to *N* = 8, *NH* = 16, *L* = 4, *OP* = 0, *NO* = 16 and *NQ* = 15.

2.4.6.2. Pixel Processor Core

The pixel processor VHDL core used in this work was originally developed and used for Dr. Llamocca's paper on *A Dynamically Reconfigurable Parallel Pixel Processing System* [64], and later in his dissertation on *Dynamically Reconfigurable*

Management of Energy, Performance, and Accuracy applied to Digital Signal, Image, and Video Processing Applications [5]. The core is composed of reconfigurable units set to process several pixels in parallel. The pixel processors use a LUT-based architecture and directly map it into the Xilinx FPGA CLB primitives.

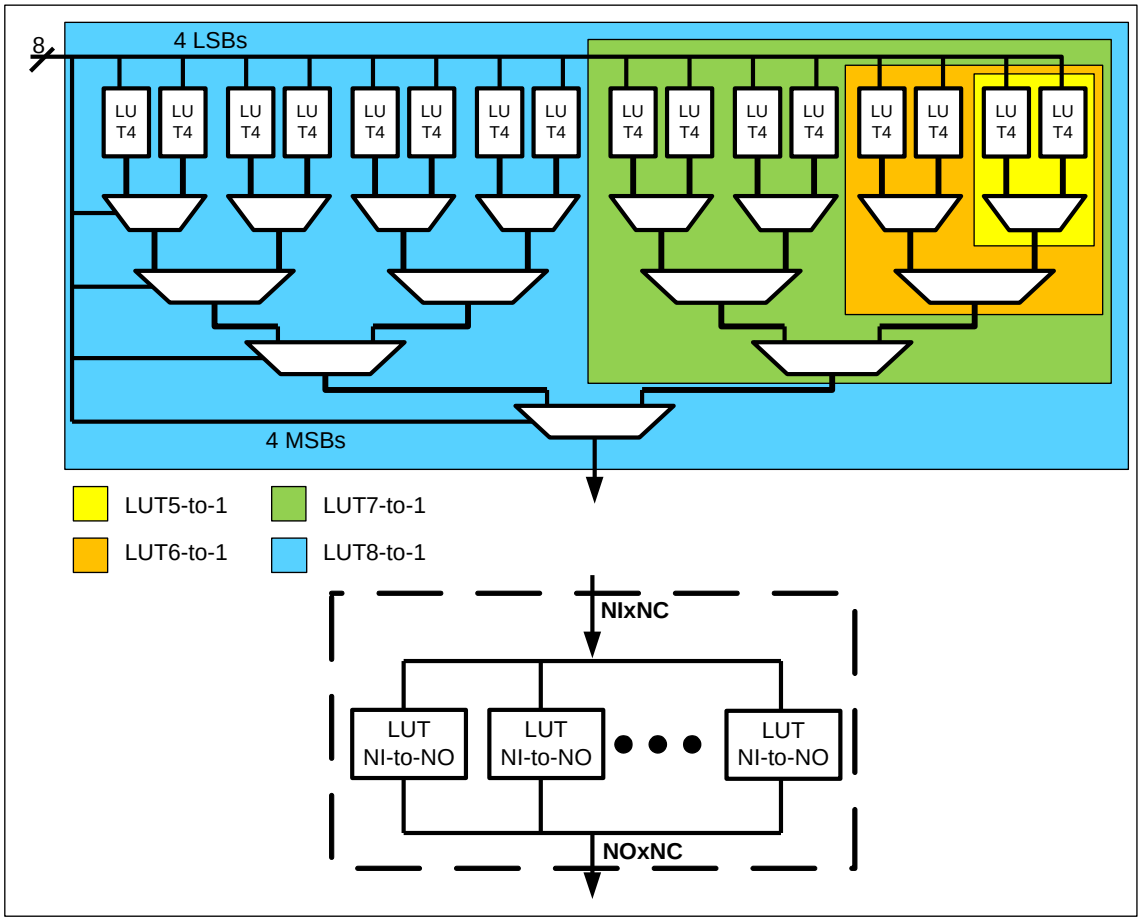

Figure 5: Pixel Processor Architecture

The pixel processor core leverages the structure of the Xilinx FPGA fabric Look-Up Table (LUT) with $L = 4$ (Virtex-II Pro, Virtex-4), and $L = 6$ (Virtex-5, Virtex-6). The core consists of '*NC*' *NI*-to-*NO* Look-Up Table (LUT) which uses *NI* input bits and one output bit.

In this architecture LUTs with higher number of input bits are built by combining the basic LUT primitives with multiplexers. Figure 5 shows the implementation of a

LUT8-to-1. LUTs with *NI* > 8 are implemented by recursively combining two '*NI*-1-to-1' LUTs with a multiplexer, as in Fig. 1(b). The hardware complexity grows exponentially as *NI* increases, and thus there is a point at which a *NI*-to-1 LUT becomes unfeasible. LUT *NI*-to-*NO*: Figure 2.1(c) depicts how a LUT *NI*-to-*NO* is built based on '*NO*' LUTs *NI*-to-1. Each LUT *NI*-to-1 implements one column of the LUT *NI*-to-*NO*.

When building the core into a design the following parameters must be defined:

Table 2: Pixel Processor I/Os

I/O	Description
NC	Number of single-pixel processor cores
NI	Number of input bits of each single-pixel processor (or the number of bits of the input pixel)
NO	Number of output bits of each single-pixel processor (or the number of bits of the output pixel).
LUT Contents	These values specify a unique single-pixel function (e.g. gamma correction, contrast stretching, etc).

Depending on the implementation, the LUT contents of each core can be identical or different. In addition, there might be applications in which *NI* and *NO* need to be different for each single-pixel processor core. However, for the vast majority of applications, *NI* and *NO* remain constant for all the cores.

For this work the Pixel Processor configurable VHDL parameters were configured as follows *NC* = 4, *NI* = 8, *NO* = 8 and the LUT Contents were configured to perform video gamma correction ($\gamma = 0.5$).

Chapter 3
Methodology

3.0 Overview and Motivation

The dissertation research introduced the following concepts that represent improvements over prior research:

- The results for the novel Partial Reconfiguration System for packet-based processing. This research demonstrates the performance of a Linux based Dynamically Reconfigurable Systems (DRS) for packet-based video processing for DPR video processing applications and test fixtures.
- New techniques were developed to demonstrate parallel processing of a digital video data stream across a channel passed into the FPGA fabric.
- A reconfigurable packet processing bus arbitration engine used for processing digital video data streams.
- A reconfigurable multi-core processing system was developed, demonstrating how such a system can be flexibly adjusted for different video processing sizes. Examples included 64x64 and 32x32 data block sizes.
- This research explains the development of a custom Linux driver for packet processing and dynamic partial reconfiguration.
- An Adaptive Weighted Round Robin (AWRR) queuing protocol that was applied to develop a Reconfigurable Packet Processor Round Robin Controller. The ability to accommodate variable size video frames and variable flow rates as well as

interspersed DPR reconfiguration message traffic in an efficient manner is accommodated by this adaptation of a WRR scheduling mechanism. This result is particularly novel and a similar mechanism does not presently exist in the literature.

- Results are provided for a high-performance video processing system based on the use of a dynamic pixel processor and dynamic, separable 2-D FIR implementation. These results directly build on the prior research of Llamocca [61] into the design of the scalable filter cores.

3.1. Hardware Architecture

The block diagram depicted in Figure 6 is a high level view of the hardware architecture developed to evaluate the Video Processing Packet Processing Engine.

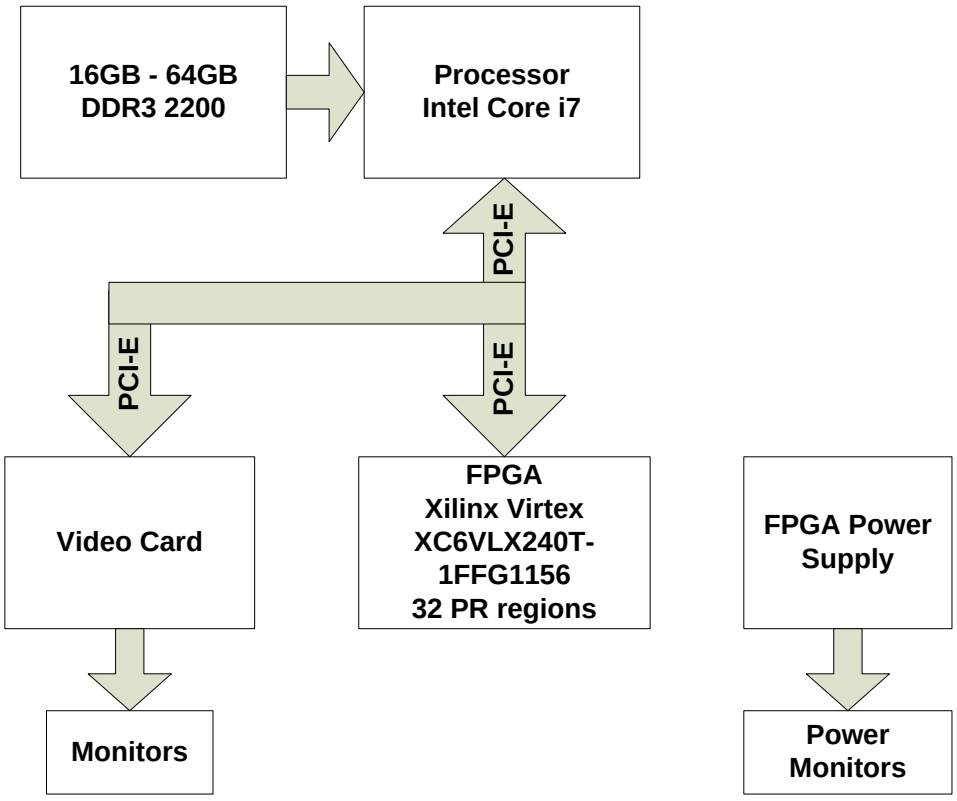

Figure 6. DPR Video Packet Processing Architecture

It is comprised of an Intel mother board, chip set, an Intel I7 processor, on-board video chip set, and Xilinx Virtex 6 evaluation boards. The system is configured with 16 GBytes of memory out of a possible 32 GBytes. The video card and two PCIe slots on the mother board are connected to the processor via the Intel mother chip set.

Digital video packets are streamed from the DDR 3 memory to the FPGA PCIe static logic Gen 1 PCIe core (Figure 11) via the PCIe bus. After processing is complete, (described in the Static Packet Processor Region section) if data is to be returned to the host, the FPGA static logic Gen 1 PCIe core returns the digital video packets to the DDR 3 memory via the PCIe bus. Ultimately, the performance of the FPGA system cannot exceed the bandwidth that is available through the PCIe bus.

PCIe is a high performance, fully scalable, well defined interconnection standard for a wide variety of computing platforms. It is designed for point-to-point connections from chip-to-chip, inter-board, and short cabling applications. The PCIe protocol was designed to provide a serial connection implementation of the PCI bus standard and consequently the protocol logically behaves as a bussed topology. As a result PCIe connections have a master-slave relationship and separate master controlled domains are associated with independent address spaces.

Utilization of PCIe provides software compatibility with existing PCIe drivers and operating systems. Being a packet based serial technology; PCIe greatly reduces the number of required pins and simplifies board routing and manufacturing. PCIe can be simply used as a point-to-point technology, as opposed to the multi-drop bus in PCI. Each PCIe device has the advantage of full duplex communication with its link partner to greatly increase overall system bandwidth. The basic PCIe Gen I data rate of 250 MB/s

for a single lane doubles that of the 32 bit / 33 MHz PCI bus. A four lane link has eight times the data rate in each direction of a conventional bus.

The system was developed to have the ability to monitor the power consumed by each individual evaluation board or by the entire system. This is accomplished by monitoring the power to the PCIe boards supplied by the ATX power supply. This capability is valuable when characterizing both the performance and power tradeoffs of different DPRS states, configuration regions and data types [66] [67].

3.2. Software Architecture

This section describes the Video Packet Configuration Software (VPCS) architecture use in the development system. It is comprised of a Linux Fedora 16 base open operating system, a custom Linux driver and several programs used to test the operation of the PPE.

3.2.1. Linux Partial Reconfiguration Platform Driver

When interfacing the FPGA development board it was necessary to develop a custom driver that would allow a user application to communicate through the Linux Kernel to the Dynamic Partial Reconfiguration enabled Hardware. The following is an overview of how the driver worked in the system.

After the system is started, the new driver is associated with the FPGA connected to the PCIe bus. This is done automatically by checking the registers in the FPGA PCIe interface to verify the Vendor ID, Device ID and Revision ID which are programmed into the FPGA bitstream when it is generated. The information is then used by the Linux kernel to invoke the drive during PCIe bus enumeration. The driver provides the following services:

- Accesses IP hardware registers
- Allocates physical memory for throughput transfers
- Returns PCIe link specific info

By using the service access points (standard file operations such as open, close, read, write and ioctl) the application code can access the FPGA and perform the User Application Instructions.

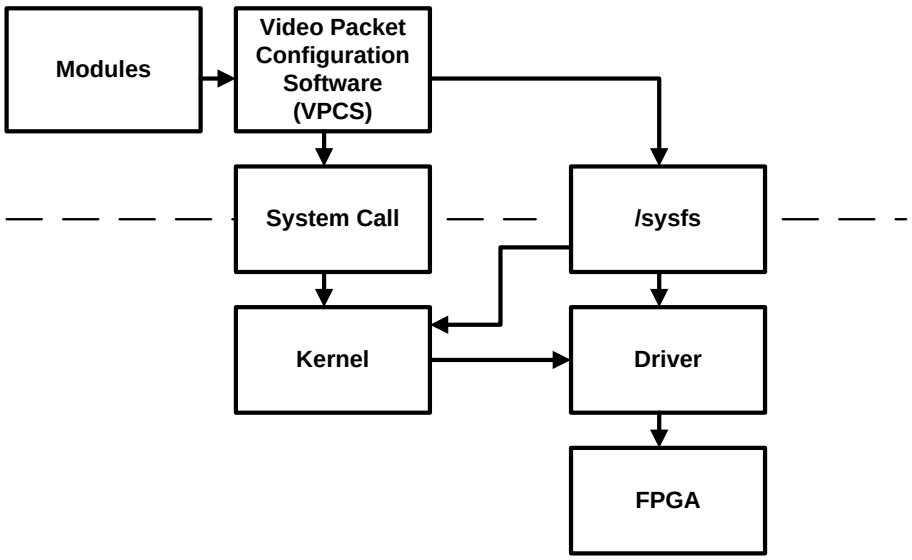

Figure 7. DPR Service Composition

Figure 7 shows the relationship between the application, the driver and the FPGA. The Application must use the OS system calls to gain access to the driver. The driver in turn uses Kernel functions to map hardware memory into kernel and user space.

In this figure, the /sysfs represents the /sys, /dev and /proc file systems found in Linux that allow the driver and user space to exchange information. The driver actually creates entries in the /sys/class and /proc areas. This is done when the driver is installed and initialized. When the driver detects and initializes an FPGA, it creates an entry in the /sys/class. It also registers a major and minor device number with the kernel. The udev

service then creates a file entry in the /dev device tree. This FPGA will have the character device major and minor number associated with the driver and board.

When the user opens the device, via an open ("filename"), the kernel knows which driver handles that file (via the major/minor) and invokes the driver's entry points. The driver uses the minor number to understand which board the user wants to access (if more than one in system). The /dev filename and the major/minor numbers are the translations between user space and kernel space.

3.2.2. Video Packet Processing

For the system to efficiently distribute the processing of the video frame data across the available thirty two reconfigurable video processing cores in the FPGA, the video frames are converted into processing packets by the VPCS (see Figure 7). Breaking the packets into properly sized elements for processing is a critical step. If the data packets are sized too large for a processing element, the PCIe bus or system memory the data will not fit in the data buffers. This will cause one of two issues. If the system element or processing core is configured for back pressure, the system will stop incoming data until the data has been processed and more data can be received by the system, causing slow performance. If the processing core is not configured for back pressure, the system will drop data, causing pour video processing results.

Figure 8. Video Processing Chain

Therefore, the first operation the VPCS running on the PC does is to determine the video frame size of the incoming video, optimally configure the reconfigurable hardware to match and determine the appropriate packet sizes to send data to and from the reconfigurable hardware. Figure 8 shows the video processing chain.

3.3. FPGA Packet Processing Regions

The Packet Processor framework was developed to allow for rapid development and ease of a cross domain hardware and software implementation solution. Figure 7 provides a representational layout of the actual device used in this dissertation.

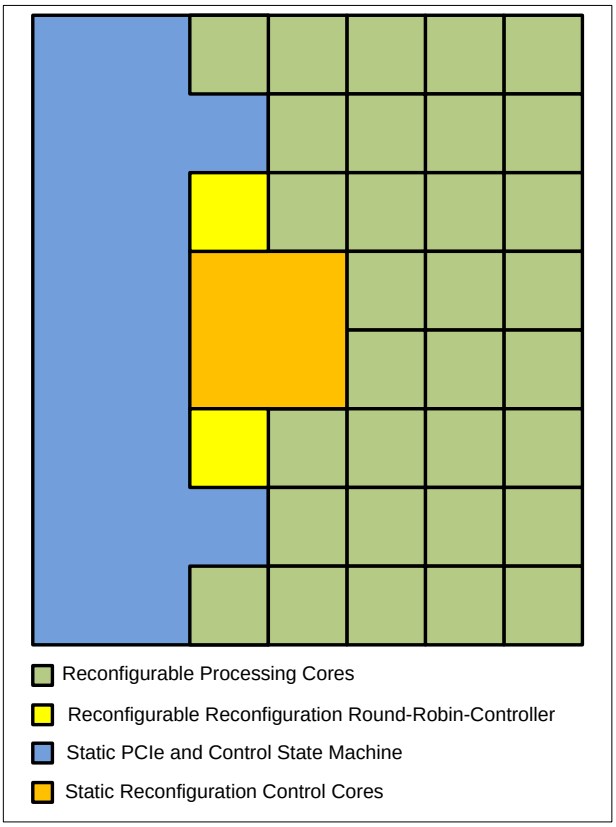

Figure 9. FPGA Device Layout

The FPGA device is laid out with four basic types of reconfiguration regions comprising of two static regions and thirty four dynamically reconfigurable regions. The

layouts of the FPGA regions are tied to the FPGA hard IP blocks and operation of the FPGA. This can be seen when looking at the PCIe hard IP block (located on the far left side of the die) and the Static PCIe and Control State Machine logic (which must also be tied to the left side of the die). Other constraints include that the FPGA hard IP block ICAP controller is located in the center of the FPGA die and the Static Reconfiguration Control Core logic is tied to the center of the FPGA die. Detailed description of these regions and their operations are provided in following sub-sections.

3.3.1. Static Packet Processor Region

This section describes the layout of the static area within the FPGA logic fabric. The static region is connected to and must likewise connect the thirty two separately dynamically reconfigurable regions. Figure 10 represents a high-level depiction of how the static framework interacts with the individual packet processing nodes. The 20 Gbps data links to the Ingress FIFO and Egress FIFO come from the 8 lane Gen 1 PCIe bus.

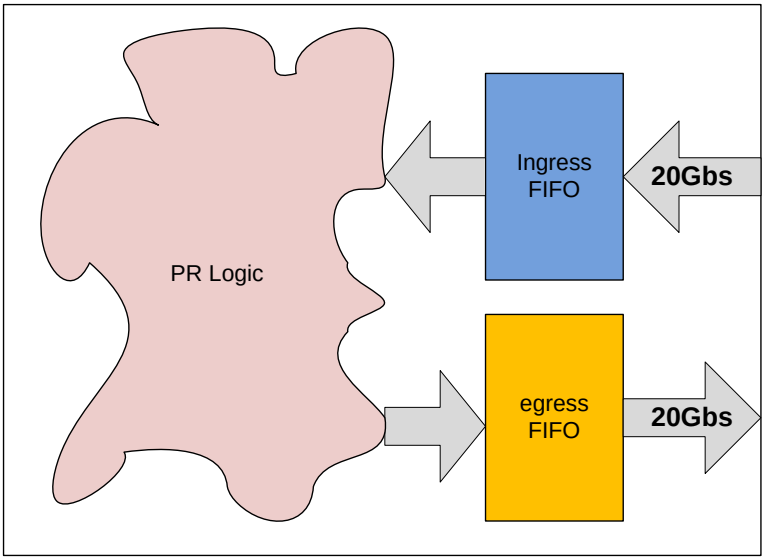

Figure 10. Notional Static Framework

The static firmware core is initially configured during system power up and does not change thereafter. This core allows video packets and DPR packets to be sent to the FPGA and processed over the same channel. The block diagram of Figure 9 shows how the data and control are connected in the core.

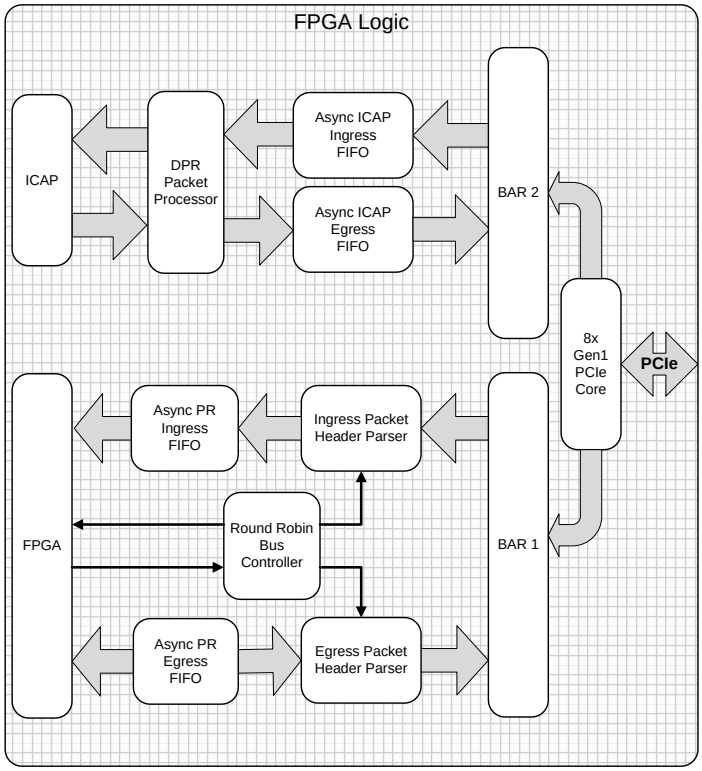

Figure 11. Static Core Logic

The static core depicted in Figure 11 consists of a PCIe core that is used to transfer data packets to and from to a Host PC. As these data packets enter the static video processing core via the PCIe bus, if the BAR address is set to BAR2 DPR processing cores are moved to the DPR engine (shown in the top of Figure 11). Otherwise, if the address is set to BAR1 the video packets are transferred across the framework to the processing cores (shown at the bottom of Figure 11) and data packets are sent to the video Packet Processing Engine (PPE).

Each of the configurable thirty two regions contain eight bit-wide read and write busses that are separately addressable from the static framework logic, each with separate read enables and write enables. This provides the ability for the video processing core to operate in full-duplex mode or write to one core while reading from another.

The logic described in the (notional) static framework that provides the ability to operate in full-duplex mode requires individual ingress and egress asynchronous FIFOs. These FIFOs alternately could be configured as a dual port memory and consequently accessed as an addressed memory buffer. The FIFOs enable decoupling of the ingress and egress data flows which can be asynchronous to each other or where data flows are non-contiguous or bursty. Moreover, FIFO queuing is advantageous for enabling rapid development and testing of new video processing cores, with individual clock regions relative to the rest of the system and for the system to operate in an efficient manner.

3.3.1.1. BAR Address 1 – Packet Processors

This section describes the FPGA static core region integrated into the overall PPE that is accessible when PCIe BAR Address 1 is selected. This address is used to read and write data to and from the PR algorithms in the FPGA fabric. The core only processes data packets after they have been separated from reconfiguration packets. If reconfiguration packets were to be accidently sent to the core, those packets will be dropped and an error is written to an error register readable by the host PC. The block diagram Figure 11 illustrates how the data and control are connected to the core.

3.3.1.2. Reconfigurable Round-Robin-Controller

This section describes the Reconfigurable Round Robin Controller (RRRC). The RRRC regions are the regions that store information that the static region uses to parse

the incoming and reconstruct the outgoing data. The top Reconfigurable Round-Robin-Controller region in the diagram is the Ingress RRRC. A functional block diagram of the Ingress RRRC is given in Figure 10.

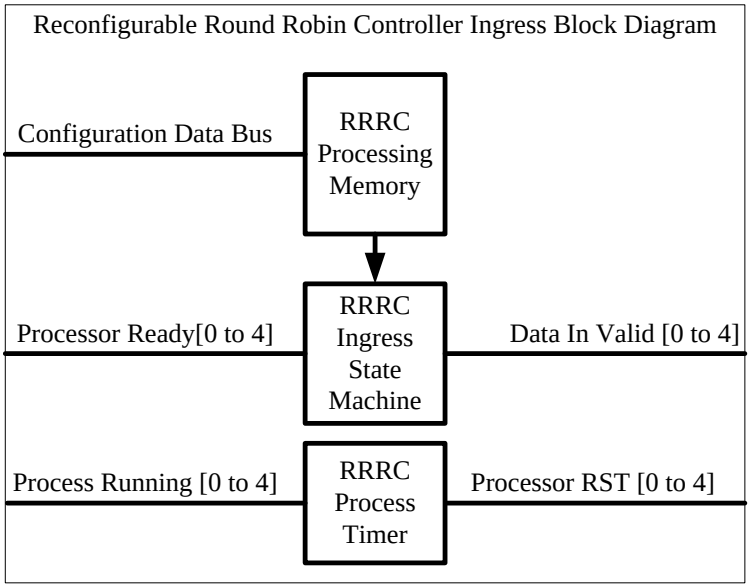

Figure 12. RRRC Ingress Block Diagram

The bottom part is the Egress RRRC. A functional block diagram of the Egress RRRC is given in Figure 10.

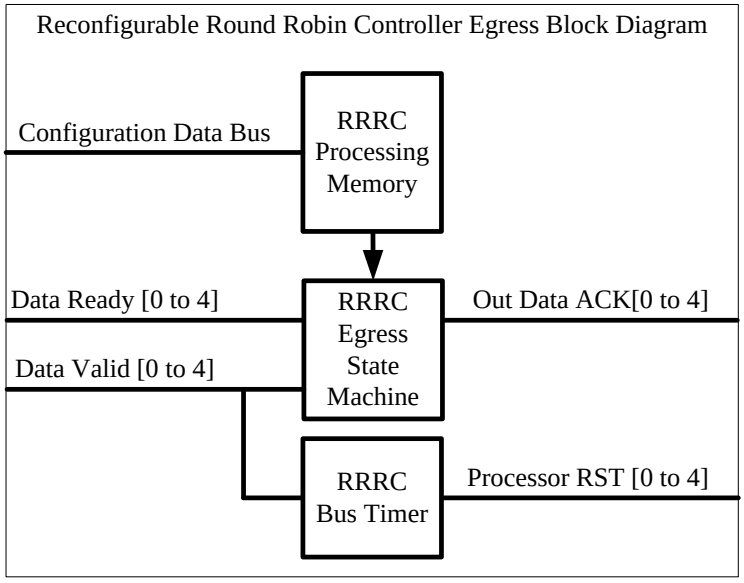

Figure 13. RRRC Egress Block Diagram

The primary blocks which make up the Ingress and Egress RRRC are the RRRC Ingress Sate Machine, Processing memory, Egress State Machine and Process and Bus Timer. A description of each block and operating mode follows.

3.3.1.2.1. RRRC State Machines

The RRRC consists of two state machines which are used to control ingress data to each of the processing elements and egress data from each of the processing elements. This function is critical to ensuring the framework and processing elements operate in an efficient manner. This is because if a processing element is idle with or without data the processing effectiveness of the entire system will suffer. The following is a description of the two modes the state machines can be configured to operate in.

3.3.1.2.2. Non Weighed Round Robin Mode

In non-Weighed Round Robin mode, the RRRC Egress and Ingress State Machines operate in a sequential order where the current service pointer is incremented by the Data_Valid signal associated with each of the processing elements. First the RRRC loads all of the processing elements sequentially one after another. Once loaded, the state machine waits until the first processing element de-asserts the Process_Running signal and asserts the Data_Ready signal. Then the RRRC asserts the processing element's Data_ACK their individual Data_Valid output signals, thus indicating that valid output data is transferring to the PCIe Egress FIFO. Once these transfers are complete, each processing element de-asserts its Data_Valid signal and then re-asserts its corresponding Processor_Ready signal, indicating that the processing element is again ready to receive the next data set to be processed.

Once this service event sequent completes, the RRRC Output State Machine determines the next processing task available for service following the service determination protocol and the Input State Machine loads the processing element that just finished moving its data to the PCIe Egress FIFO.

3.3.1.2.3. Weighed Round Robin Mode

The RRRC memory is used to store the adaptive core latency information for each of the thirty two cores or weights (or skips) for a video stream. The memory is a single FPGA Block RAM configured with a nine bit data path and 2K memory depth. If the RRRC Processing Memory is not enabled no skips are preformed and the processors are read and written in sequential order. When enabled, the memory is configured by the host VPCS on initialization with a specific skip pattern for each full processor write and read with each bit of data representing the skip pattern for a given full processor write. The high bit is used to indicate if the pattern is a write or read sequence. The second highest bit is used to signal a reset memory address counter this bit is used to restart the skip pattern. The Table 3 shows the memory bit to processor mapping and the Skip_RST and Read/Write bit.

Table 3. RRRC Skip Table

Processor	bit 0	bit 1	bit 2	bit 3	bit 4	bit N	Skip RST	High bit 8 Read/ Write
0	1	0	0	0	0	0	X	X
1	0	1	0	0	0	0	X	X
2	0	0	1	0	0	0	X	X
3	0	0	0	1	0	0	X	X
4	0	0	0	0	1	0	X	X
N	0	0	0	0	0	X	X	X

3.3.1.2.4. RRRC Process Timer

To ensure a robust operating environment, each of the processing elements in the framework is associated with a watchdog timer to ensure that no processing node is in an unknown state. The watchdog timer is configured on device initialization with the number of cycles expected for a given processing node. During operation the timer is started once all of the data for the process is moved across the framework from the PCIe FIFO to the processing element. Once the processing is complete the processing unit stops the watchdog timer by de-asserting the process running signal. If the watchdog timer expires before the processor de-asserts the process running signal the watchdog timer will reset the processing element. The data being processed by this node is then dropped. The VPCS will fill in the dropped data with 0bs.

3.3.1.2.5. Framework Bus Timer

The Framework Bus Timer is also used to ensure that the data egress process from each processing element does not hang up the fabric this can be caused by the process asserting its Data_Valid signal (which indicates the process is driving the framework egress bus) and hanging up hence not de-asserting the signal. The way this prevented is when the RRRC asserts a process Out_DATA_ACK signal the bus watchdog timer is started. If the process does not complete before the timer expires the watchdog timer will reset the processing element. The un-transferred data being processed by this node is then dropped. The VPCS will fill in the dropped data with 0bs.

3.3.1.2.6. Moving Data To and From Cores

To load a video processing core (e.g., the FIR core) the Ingress Data PPE checks the data in the buffer, by reading the header, to determine if the packet is 64x64 or 32x32. Once the determination is made it will cycle to the next available 64x64 or 32x32 processing engine. If a processing engine is not available the PPE will back pressure the PCIe core and wait until a processing core is available.

The Egress Data PPE operates with a clock frequency of 250 MHz with a bus width of sixteen bits. This equates to a total bandwidth of 8.2 Gbps. The Egress Data PPE is used to transfer the processed data back to the PCIe core. By reading the status of each core the Egress Data PPE determines which core has finished processing and when data is available. It will then move the available data from the cores to the PCIe. The PCIe core will then send it back to the Host PC.

3.3.1.3. BAR Address 2 – DPR

This section describes the FPGA static core region integrated into the overall PPE that is accessible when PCIe BAR Address two is selected. This region is used to read and write the reconfigurable configurable fabric in the FPGA. The core only works with the FPGA packets after they have been filtered from the video packets. If non-reconfiguration packets are sent to the reconfiguration core, the packets will be dropped and an error will be written to an error register readable by the host PC. The block diagram Figure 14 shows how the data and control are connected in the core.

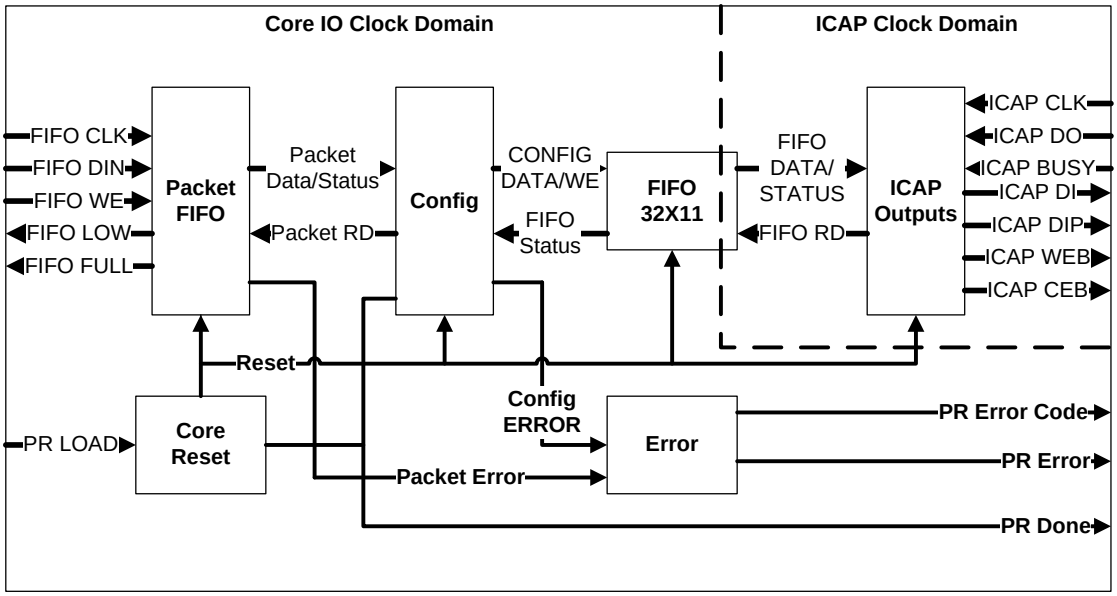

Figure 14. DPR Packet FIFO

3.3.1.3.1. DPR Core PACKET FIFO

This module ensures that incoming packets pass error checking prior to being forwarded onto other modules in the core. The PACKET FIFO module (Figure 14) receives the packets through a FIFO interface and uses the packet information (length, sequence number, and CRC value) to perform error checking.

After PR_LOAD assertion, the first received packet must contain a sequence number equal to 0x00h. Each subsequent packet contains a sequence number one greater than the prior packet. At 0xFFh, the sequence number should roll back over to 0x00h. If the CRC32 value or the sequence number does not match the expected value, the PACKET_ERROR signal is asserted and the packet is not forwarded.

The data is stored in a FIFO until the last word of the packet is received. The last word is identified based on the length received in the first word. After the last word is received, the count for the number of packets stored in the FIFO is incremented if the

sequence number and CRC check pass. The FIFO is determined to be full if the number of stored words equals 1,023. The FIFO size is 4 KB which can contain up to two maximum-size (512 words) packets. The packet information (length, sequence number, and CRC value) provides error checking up to the packet FIFO. Parity calculation is used to maintain data integrity throughout the rest of the core. The read data from the FIFO is forwarded to the next module with back pressure control.

The PACKET FIFO module asserts the error signal, PACKET_ERROR, when the error checking fails, data is written into a full packet FIFO (overflow), or data is read from an empty packet FIFO (underflow). The error signal persists until PR_LOAD is removed, resulting in an RC/EPRC-wide reset. Any remaining packets received after an error is reported will not be forwarded to other modules. Thus, it is not required to complete sending all remaining packets after an error [67] [66].

3.3.1.3.2. DPR Core CONFIG

The CONFIG module monitors configuration data forwarded from the PACKET FIFO module for configuration commands. It also descrambles the IDCODE into its original value (refer to Figure 14 to see the relationship of the CONFIG Module to other components associated with the PACKET FIFO).

The EPRC configuration logic then divides the data into bytes and sends the data to the ICAP interface through the ICAP FIFO. The full signal from the FIFO applies back pressure towards CONFIG. The empty signal is used by the ICAP OUTPUTS module to know when to read data from the FIFO. The CONFIG module asserts the error signal, CONFIG_ERROR, when the original partial bitstream contains invalid commands or information. The error is determined internally and the faulty data is never passed to the

ICAP. The ICAP output (ICAP_DO) is not used for determining error conditions. The CONFIG module always asserts PR_DONE, even in case of an error. It is expected to always wait for PR_DONE assertion before de-asserting PR_LOAD even if PR_ERROR is asserted first. This allows for any clean-up that needs to be done by the core (e.g. sending an ABORT to ICAP). PR_DONE assertion always requires EXT_PR_DONE input to be high (Logic 1).

3.3.1.3.3. DPR Core ICAP OUTPUTS

The ICAP_OUTPUTS module (see right side of Figure 14) is responsible for reading the ICAP FIFO and output. It also monitors the PR_ERROR signals for status. The ICAP_OUTPUTS module generates an abort sequence and stops reading the ICAP FIFO if it detects an error condition.

3.3.1.3.4. DPR Core RESET

The DPR Core contains a RESET module (not shown in the block diagram for simplicity) which consists of a 3-bit down counter. The most significant bit (MSB) of the counter decrements the counter and is tied to the active-High RESET output. The counter is asynchronously loaded with 111b when PR_LOAD is de-asserted (logic 0). After PR_LOAD is asserted, the down counter counts down until the MSB is zero. At this point, the counter stops decrementing and them the RESET output is de-asserted. The final count value is 011b.

3.3.1.3.5. DPR Core ERROR

The ERROR module encodes the various error signals inside the core into PR_ERROR and PR_ERROR_CODE signals for external usage. Table 4 shows the encoding of the PR_ERROR_CODE signal.

Table 4: PR Error Code Encoding

Inputs				Outputs
EXT_PR_ERROR	**RESERVED**	**CONFIG_ERROR**	**CONFIG_ERROR**	**PR_ERROR_CODE**
1	X	X	X	1 and EXT_PR_ERROR_CODE
0	0	1	X	0010
0	0	0	1	0001
0	0	0	0	0000

3.3.1.3.6. DPR Core Procedure after Errors

The error signal persists until PR_LOAD is de-asserted which results in a PRC/EPRC-wide reset that allows for a new partial bitstream to be sent to the PRC/EPRC core. After the occurrence of any type of an error (PACKET/CONFIG), PR_ERROR is always followed by PR_DONE. Thus, it is required to wait for PR_DONE assertion before de-asserting PR_LOAD. This allows the core to complete any action that needs to be done after the error.

A PR_ERROR from the PRC/EPRC core should result in a system-level decision of either retrying with the same partial bitstream or first cleaning up with a blank partial bitstream. In case of a packet error, it is not possible to resend the packets from where the

error was encountered. Reconfiguration of any reconfigurable partition is allowed after an error.

3.3.1.3.7. DPR Core Interface Signals

Table 5 provides a list of the interface signals to the DPR core. These signals are what other logic elements within the system interface to in order to send packets into the configuration circuitry of the FPGA device.

Table 5. DPRPP Core Interface Signals

Signal	I/O	Description
PR_LOAD	Input	This port is asserted (active High) during the entire partial reconfiguration. Deassertion resets the PRC/EPRC core
FIFO_CLK	Input	This port is used to clock the incoming FIFO data and write strobe
FIFO_DIN[31:0]	Input	This is the incoming configuration data
FIFO_WE	Input	This is the write strobe for FIFO_DIN[31:0]
FIFO_FULL	Output	This port indicates that the FIFO is full and cannot receive more data
FIFO_LOW	Output	This port indicates that the FIFO has space for at least 2,048 bytes
ICAP_CLK	Input	This is used to clock the ICAP primitive
ICAP_DI[31:0]	Output	This is the incoming data for the ICAP primitive
ICAP_DIP	Output	This port provides odd parity on ICAP_DI [31:0] and does not connect to the ICAP primitive. It can be used by the user for additional error checking between the PRC/EPRC core and the ICAP primitive
ICAP_CEB	Output	This is the active-Low enable for the ICAP primitive
ICAP_WEB	Output	This is the active-Low write enable for the ICAP primitive

Signal	I/O	Description
ICAP_BUSY	Input	This is the busy signal from the ICAP primitive
ICAP_DO[31:0]	Input	This is the outgoing data from the ICAP primitive used for status
EXT_PR_ERROR	Input	TBD
EXT_PR_ERROR_CODE[2:0]	Input	TBD
EXT_PR_DONE	Input	TBD
PR_ERROR	Output	This port indicates that a partial reconfiguration (PR) error has occurred
PR_ERROR_CODE[3:0]	Output	This is the error code for the PR error
PR_DONE	Output	This port indicates that the PR operation has completed. It is dependent on multiple internal flags for a greater number of clock cycles to ensure that all configuration information has been given adequate time to be delivered to the ICAP

3.4. DPR Regions

The next section will discuss how the DPR designs were placed on the Xilinx Virtex 6 FPGA fabric. This step in the Partial Reconfiguration process is known as partitioning the design. This is true because the area of the die that will be used for the partial reconfiguration regions and the static area are defined. This process was done using the Xilinx PlanAhead tool. An image of the FPGA layout take from the PlanAhead tool is given in Figure 13.

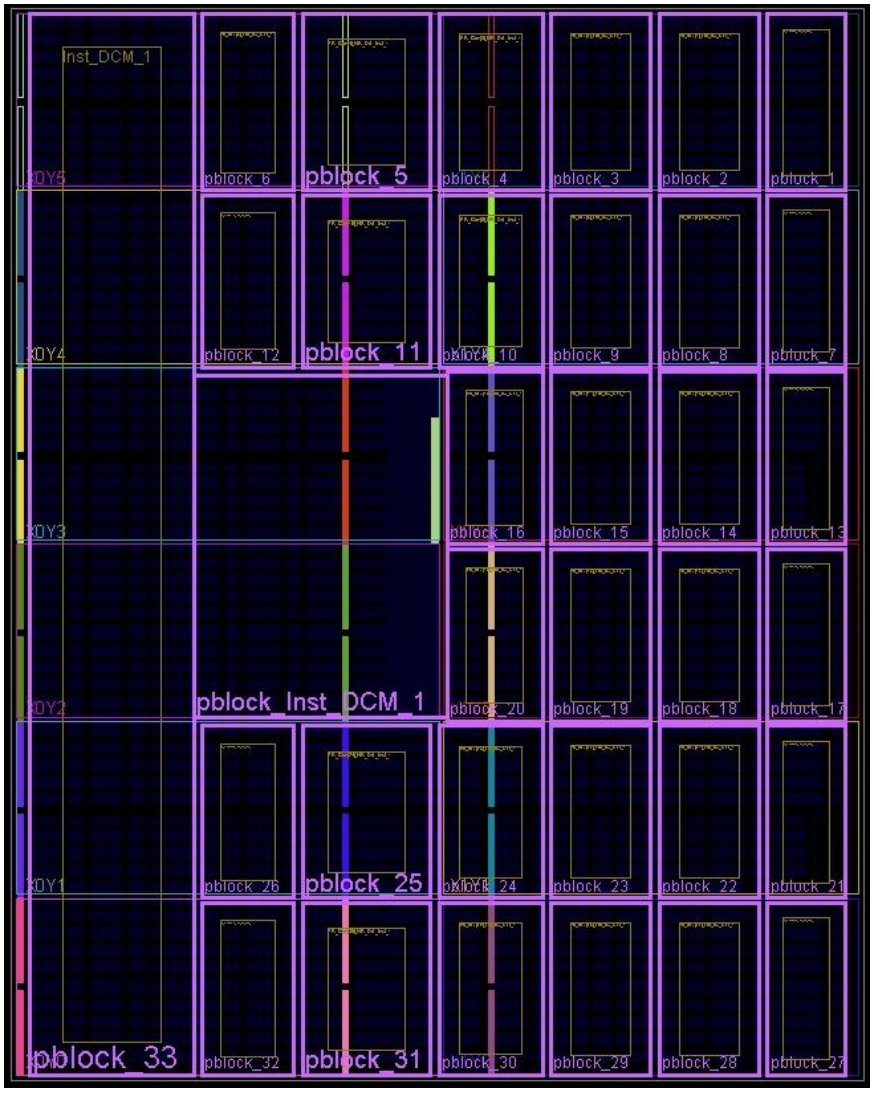

Figure 15. Dynamic Partial Reconfiguration Regions

The Dynamic Partial Reconfiguration Regions (DPRR) in the FPGA are placed on the right side of the FPGA fabric so that each of the thirty two regions contains approximately the same logic resources. This simplifies the development of processing blocks and gives a regular layout and roughing structure to minimize the need to change the static region when developing new functionality. An image of the thirty two regions after the Xilinx place and route tools finished one of the thirty two core test designs is shown in Figure 14.

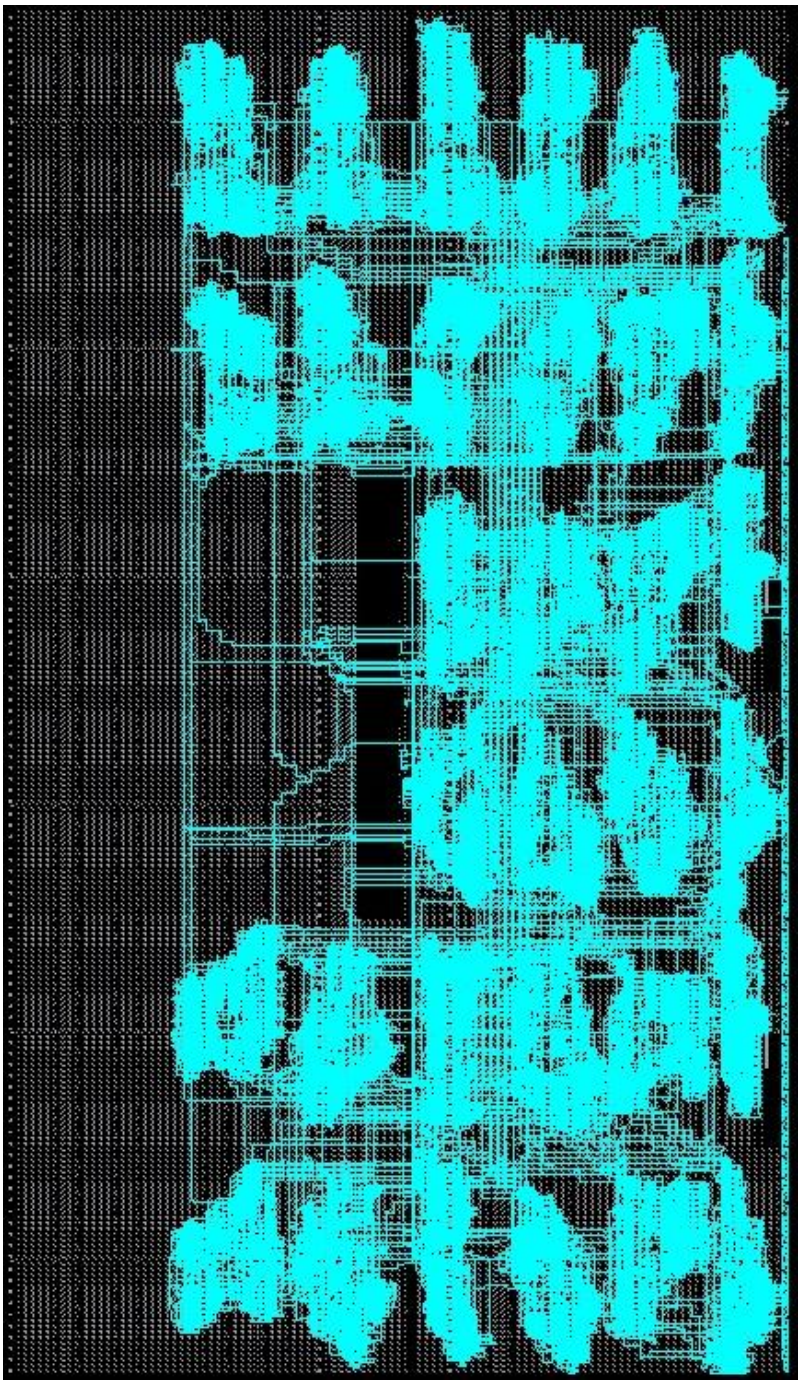

Figure 16. Dynamic Partial Reconfiguration Placement

Figure 14 shows how the Xilinx place and route tool placed each processing element into the area of the Virtex 6 defined in the PlanAhead tool. The image does not show the static region routing [68].

3.5. Base System Test Application

This section discusses the Base System Test Application which is used to perform low level verification that the static firmware including the RRRC and PR regions, software driver, and DPR process are operating as expected. The procedure to perform the test can be seen in Figure 17. To start the test first the PR controller bitstream is written to the FPGA development board. This process is done when the PC is turned on because the FPGA must be found on the PCIe bus during system enumeration otherwise the system will not load the PCIe into the list of available system PCIe devices.

Once the system is loaded, it is necessary to connect to the system using the custom driver and application to up load the selected partial bitstreams across the PCIe bus to the FPGA. After each of the partial bitstreams is loaded, the Software Test Application is loaded and started.

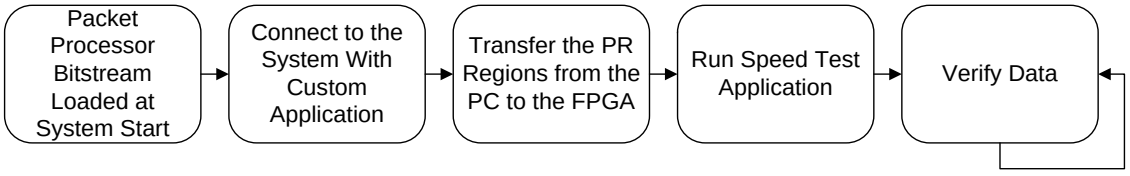

Figure 17. Test Application Flow

3.5.1. Test Application

The ivPCL Software Test Application GUI allows the user to actively monitor and test the base PR system and the FPGA data path speed for various sized packets. This is an important baseline when verifying the maximum speed and packet size at which the system can operate. The Figure 18 is the GUI interface for the speed controller.

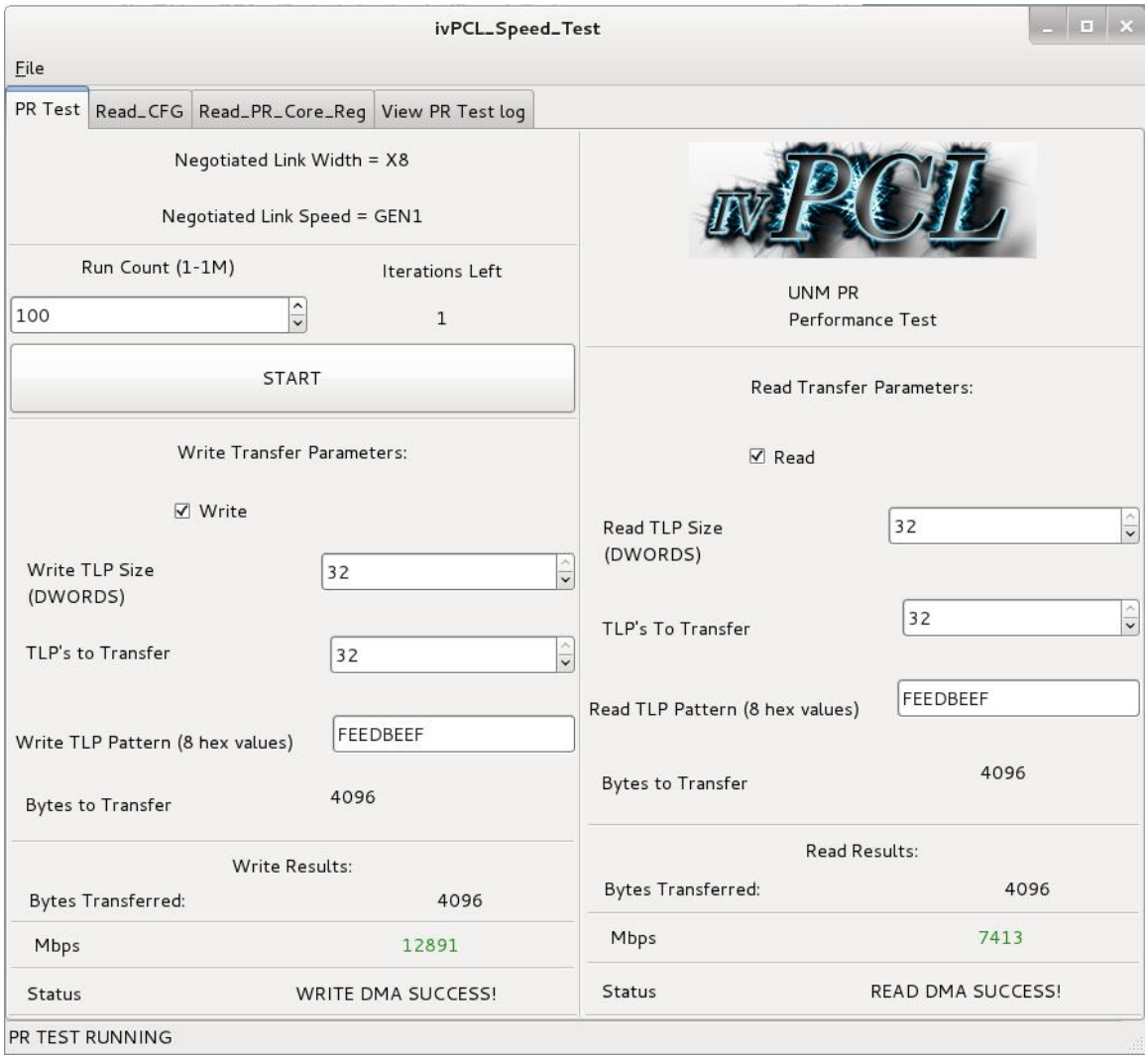

Figure 18. Speed Test GUI

The GUI allows the user to read and write various test patterns to the FPGA. It also allows the user to adjust the Transaction Layer Packets (TLPs) discussed in the next few sections.

3.5.2. Test Procedure for PCIe Performance

In this dissertation, the means of which the performance system performance is measured is by counting the number of TRN clocks until a DMA transfer is complete. (TRN_CLK is the interface clock provided by the PCIe core and used by DPR core to the

Test application.) For reads, the count stops when all completions have been returned, and due to writes being posted (no response), the count stops when all writes have been successfully sent into the core [69].

3.5.2.1. PCIe Performance

Excluding PCIe protocol and traffic overhead [70] the primary elements which impact performance in a PCIe system are the maximum payload size, maximum read request size, and request type.

The PCIe specification allows for a Maximum Payload Size (MPS) of up to 4,096 bytes. However, every device in the PCIe hierarchy must use the same MPS setting, and the setting must not exceed the capability of any device within the hierarchy. Therefore, the device with the smallest Maximum Payload Size (MPS) in a system determines the system MPS.

A system's MPS setting is determined during the overall system enumeration and configuration process. Each device in the hierarchy advertises its MPS capability in its Device Capability register, which is located in the device's configuration space. The VPCS probes every device to determine its MPS capability, determines the MPS setting, and programs every device by writing the MPS setting to its Device Control register. Therefore, in a PCIe packet based system, the VPCS must adapt the packet size depending on the MPS of the system.

The MPS of a system does impact the performance of a system. However, increasing payload size does not increase efficiency at the same rate. This is related to all the payloads on the PCIe bus have a Transaction Layer Packets (TLPs) overhead to transfer the data. The transaction layer, Data Link Layer (DLL), and physical layer

(PHY) add overhead to each TLP, thereby reducing the effective data transfer rate. The transaction layer attaches the packet's header and optional End-to-end Cyclic Redundancy Checksum (ECRC). The DLL adds the sequence number and Link Layer CRC (LCRC) to the packet to ensure successful transmission across the link. The PHY also adds information to mark the beginning and end of the packet.

The TPL overhead varies between 20 to 28 bytes depending on the use of 32-bit or 64-bit addressing and optional ECRC. For the purpose of this research, memory read or memory write TLPs are used to move data to and from the FPGA with 64-bit addressing. The 64-bit addressable TLP header is 16 bytes (as opposed to 12 bytes for 32-bit addressing) and requires an additional 4 bytes of information to be exchanged in the packet.

To transfer large amounts of data across the PCIe bus requires multiple TLPs. Although each TLP contains a given amount of overhead, larger multiple TLP transfers increase link efficiency. The maximum payload size (MPS) setting, assigned to the communicating devices, determines the maximum TLP size. Increasing the MPS does not necessarily lead to a corresponding increase in link efficiency because as individual TLPs become larger, other factors such as traffic overhead begin to impact link performance. We have that:

$$\text{Payload Efficiency} = \text{MPS}/(\text{MPS}+\text{TLPs}). \quad (5)$$

Therefore, the Theoretical Maximum Data Throughput (TMDT) in bytes is the packet efficiency as a percentage of the theoretical bandwidth, given by:

$$\text{TMDT} = (\text{Bus Speed} * 2 * (\text{PCIe Lane Width}))/(\text{PCIe Symbol Encoding}). \quad (6)$$

The PCIe Symbol Encoding for Gen 1 and Gen 2 is 8b-10b. Encoding is used to maintain a DC balance for the bus and for clock data recovery [69].

Chapter 4
Implementation Results

This chapter describes the research results developed as result of application of DPR FPGA processing to video frame processing. This section also describes the algorithms and mechanisms developed to test the Reconfigurable Video Packet Processing Framework (RVPPF). We begin by characterizing each of the basic system functions for each of the implementations on the RVPPF. In this process, three central questions are raised. First, what is the best mix of video algorithms to operate on the video frame targeted for processing with the RVPPF? Second, how is unprocessed video input distributed to the dynamic algorithms configured on the RVPPF? Third, how are processed frames interleaved into a coherent output video stream? Each of these questions is explored in turn. Lastly, the performance of the algorithm implemented on the RVPPF is evaluated.

4.0 RVPPF DPR Loopback

This section describes the RVPPF DPR Loopback implementation. A description of the static region RVPPF and associated FPGA firmware can be found in Chapter 4.

To baseline the performance of the RVPPF, test the ability for the RVPPF to reconfigure the FPGA fabric and provide a base to build new processing elements a Loopback DPR (L-DPR) core and associated host PC application was developed. The general approach for the Loopback core is to route packets from their originating facility back to the receiving end of the source without intentional processing or modification. This is primarily a means of testing the RVPPF transmission or transportation

infrastructure. The following is a description of the L-DPR base system and results from the implantation of thirty two asynchronous L-DPR cores implemented in the RVPPF.

4.1.1. RVPPF DPR Loopback FPGA Core

To test different possible functionalities, two L-DPR FPGA cores were developed. Both cores consisted of a state machine, configurable clock divider logic, and four Block RAM or Transfer Memories. The two cores are named as follows: (i) L-DPR Separate Transfer Memory L-DPR-STM with an Ingress FIFO (I-FIFO) and Egress FIFO (E-FIFO), and (ii) an I-FIFO E-FIFO Combined Transfer Memory (L-DPR-CTM).

In the L-DPR core, the Transfer Memory (Block RAM) can be configured to be in several different configurations. The memory stores up to 36K bits of data and can be configured as either two independent 18 Kb RAMs, or one 36 Kb RAM. Each 36 Kb block RAM can be configured as a 64K x 1 (when cascaded with an adjacent 36 Kb block RAM), 32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36, or 512 x 72 in simple dual-port mode. Each 18 Kb block RAM can be configured as a 16K x 1, 8K x2 , 4K x 4, 2K x 9, 1K x 18 or 512 x 36 in simple dual-port mode.

In the Virtex-6 architecture, dedicated logic in the block RAM enables users to implement synchronous or dual-clock (asynchronous) FIFOs. This eliminates the need for additional CLB logic for counter, comparator, or status flag generation, and uses just one block RAM resource per FIFO. The FIFO can be configured as an 18 Kb or as a 36 Kb memory. For the 18 Kb modes, the supported configurations are 4K x 4, 2K x 9, 1K x 18, and 512 x 36. The supported configurations for the 36 Kb FIFO are: 8K x 4, 4K x 9, 2K x 18, 1K x 36, and 512 x 72 [71].

When configured in the L-DPR-STM half-duplex (HDX) mode, the system provides communication in both directions, but only one direction at a time (not simultaneously). In this configuration, once a core begins receiving a signal, the internal state machine must wait for the transmitter to stop transmitting, before starting to process the data and return the data to the host. In this configuration the processing units have double the amount of local memory to use for data storage compared to L-DPR-CTM. A functional description follows and a depiction of the L-DPR-STM core can be seen in the Figure 19.

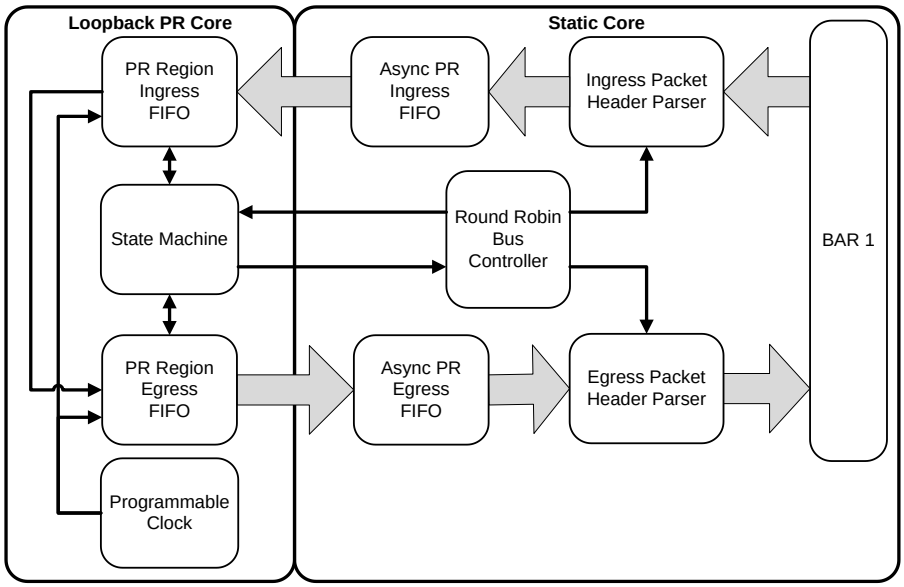

Figure 19. L-DPR-STM Core Block Diagram

For the L-DPR-CTM, when data enters the FPGA static region via the PCIe bus it is passed to the next available DPR region through an Ingress Asynchronous FIFO (IA-FIFO). The IA-FIFO is used to buffer the incoming data and move the data between the FPGAs static region clock domain and the FPGA PR regions clock domain. The ingress data packet is then moved to the PR regions Ingress FIFO (I-FIFO). Once finished, the Reconfigurable Round Robin state machine sends the data done signal to the PR region

the state machine which starts the loopback of the data to the HOST by moving the data packet to the Egress FIFO (E-FIFO). When the transfer is complete, the PR region state machine asserts data ready to the RRRC. For the final step in the date flow through the FPGA, the RRRC sends the current data into the E-FIFO and then back to the system via the PCIe bus.

When configured in the L-DPR-CTM full-duplex (FDX) mode the system allows communication in both directions, and, unlike when in the half-duplex mode, allows this to happen simultaneously. In this configuration the processing units have half the amount of local memory to use for data storage compared to L-DPR-STM, but can send and receive simultaneously. A functional description follows and a depiction of the L-DPR-CTM core can be seen in the Figure 20.

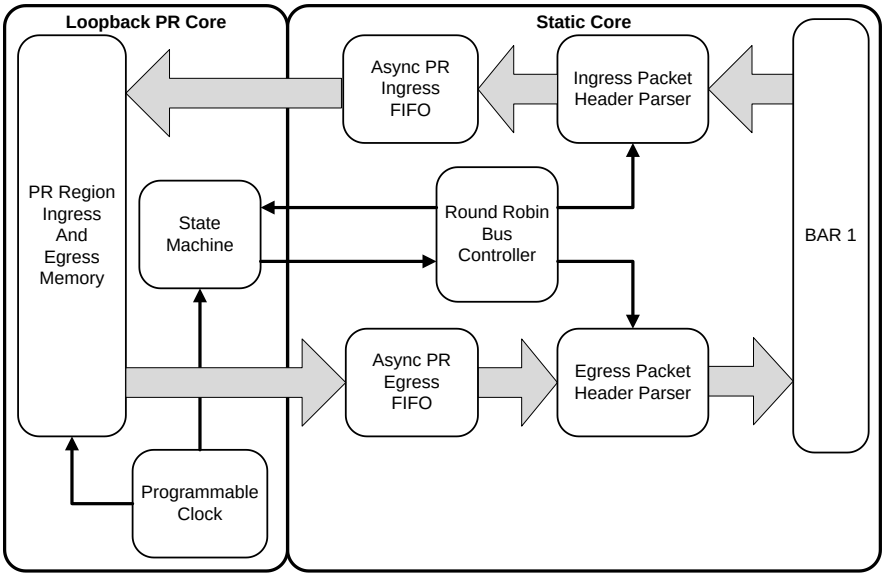

Figure 20. L-DPR-CTM Core Block Diagram

For the L-DPR-STM, when data enters the FPGA static region via the PCIe bus it is passed to the next available DPR region through an Ingress Asynchronous FIFO (IA-FIFO). The IA-FIFO is used to buffer the incoming data and move the data between the

FPGAs static region clock domain and the FPGA PR regions clock domain. The ingress data packet is then moved to the PR regions local memory. Once the Reconfigurable Round Robin state machine sends the data done signal to the PR region, the control state machine starts the loopback of the data to the HOST by asserting data ready. Once the RRRC responds with an acknowledge, the PR control state machine must start a counter to walk through the memory addresses in order to send the data back to the static regions Egress Asynchronous FIFO (EA-FIFO). For the final step in the date flow through the FPGA, the RRRC sends the data currently in the E-FIFO back to the system via the PCIe bus.

During the loopback operation when the packet is moved from the ingress memory to the egress memory the header tag (described in the software application section) placed on the data by the VPCS for data tracking must also be moved. The data tag header must always remain with the data packet or otherwise the VPCS will not know what to do with the data and will subsequently drop the packet.

The L-DPR core can be configured with several different clock speeds via the FPGA programmable clock that is a programmable division of the primary static core clock. This programmable clock allows for a more accurate model of the way different PR cores with different clock constraints and amounts of latency behave in the system. The clock speed is adjusted by the user via the host application writing to control registers in the loopback core.

4.1.2. RVPPF Loopback Speed Calculations

The RVPPF which includes the FIFO interface from the PCIe and the Reconfigurable Weighted Round Robin Controller (RWRRC) run off a single clock.

Therefore, the speed at which data moves between the PCIe and the cores for processing is simply the data in Bytes multiplied by the clock speed. Additional latency can come from the processing time that each core takes to perform processing on the data (is the RWRRC required to wait on a process to finish) and if the RWRRC is not configured to process the data in an efficient manner. For the Loopback operation none of the latencies from the RWRRC were observed. This was because the RWRRC was configured without weights as all cores had the same bandwidth needs.

4.1.3. RVPPF DPR Loopback Host Application

The primary purpose for the host application running on the PC for the Loopback Application (L-App) is to verify the system comprising of the host operating system driver, RVPPF and PR cores. DPR-L host application can do the following verification processes:

1. Generate packets of configurable size or read them from a file
2. Send data packets to the FPGA
3. Start the system timer to measure data speed
4. Send packets to the FPGA
5. Receive packets from the FPGA
6. Stop the system timer to measure data speed
7. Compare the data sent data to the received data
8. Return the status of the transfer to the user
9. Perform Dynamic Partial Reconfiguration of the FPGA Reconfigurable Fabric

In addition, to verify processor clock speed changes to the RVPPF the DPR-L can write to the 32 bit selectable packet delay register in the DPR-L cores. This can produce a slower or faster data write back from each of the cores. It also allows for verification of different clock speed cores and cores with different data processing capabilities running in the RVPPF.

4.1.4. RVPPF DPR Loopback Implementation

The primary purpose of the loopback Implementation is to detect dynamic system failures so that defects may be discovered and corrected and establish a base line for the way the system behaves with different configurations. Loopback testing cannot establish that the RVPPF functions properly under all conditions, but can only establish that it does not function properly under specific conditions. The scope of the testing includes configuring and testing with several different clock speeds to verify the ability of the RVPPF to perform DPR of the FPGA, operate with packets of different size, operate with various PR region frequencies and operate with both L-DPR-CTM and L-DPR-STM cores configured simultaneously in the RVPPF.

4.1.5. RVPPF DPR Loopback Results

This section presents performance measurements for both the L-DPR-CTM and L-DPR-STM cores in the current RVPPF implementation. The measurements were taken by creating various packet sizes that included processing tasks with different per frame processing latencies. In each case, frames from an input stream are received, decoded, held for a specified amount of time, encoded, and retransmitted as a processed output stream.

Figure 21 shows the results of processing a 640x480 test video stream with 16 bits per pixel. The RVPPF is configured with thirty two L-DPR-CTM and L-DPR-STM cores.

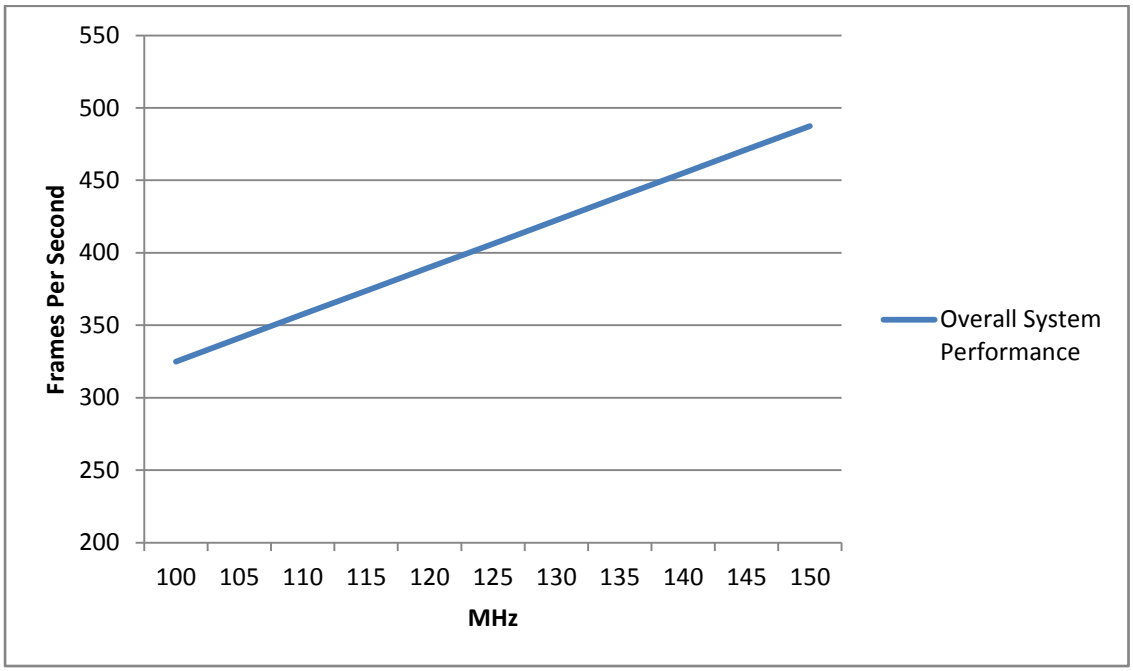

Figure 21. Frames per Second

Varying performance was emulated by processing latencies which were varied by decreasing and increasing the processors clocks from 100 MHz to 150 MHz. The graphs show a nearly linear increase in performance as the speed of the processing nodes is increased which indicates that the RVPPF implantation is highly scalable for video streams.

4.2. RVPPF DPR Video 1-D and Separable 2-D FIR Filter

The following section is a description and implantation of a video processing application using a 1-D FIR filter (V-FIR) core and 1-D FIR filter (V-FIR) core with a matrix transpose controller used to create a separable 2-D implantation on the RVPPF. The primary purpose of the FIR implementation is to answer the following question:

what is the best mix of FIR processing cores and algorithms for video processing with the RVPPF?. Second, *how is unprocessed video input distributed to the dynamic algorithms configured on the RVPPF?* Third, *how are processed frames interleaved into a coherent output video stream?* In answering the above questions, the RVPPF will demonstrate the ability to partially reconfigure the hardware, successfully deconstruct video frames into packets, send the packets across the PCIe bus to the RVPPF, process the packets in the RVPPF, and send them back to the host to be re assembled into processed video frames. We also present an example that demonstrates that the system can using 640x480 frames maintain a video packet throughput of 300 frames per second.

4.2.1. Removing Filter Edge Effects

When using a digital filter to process images the edge effects must be taken into account. This is particularly important when breaking the frame into blocks and processing them independently. If the edges are not taken into account the result will be lines on the processing boundaries of the block.

To remove the edge effects for the 1-D and subsequent separable 2-D filter, a zero padding value was added to the video packets. This was achieved by increasing the row size *M* and the column size *N* in each direction by the number of coefficients minus 1. For example if a row size 64 and column size 64 block is to be processed by a filter with eight coefficients the resultant image to be processed will have a row size of 78 and column size of 78. Figure 22 depicts the resulting structure.

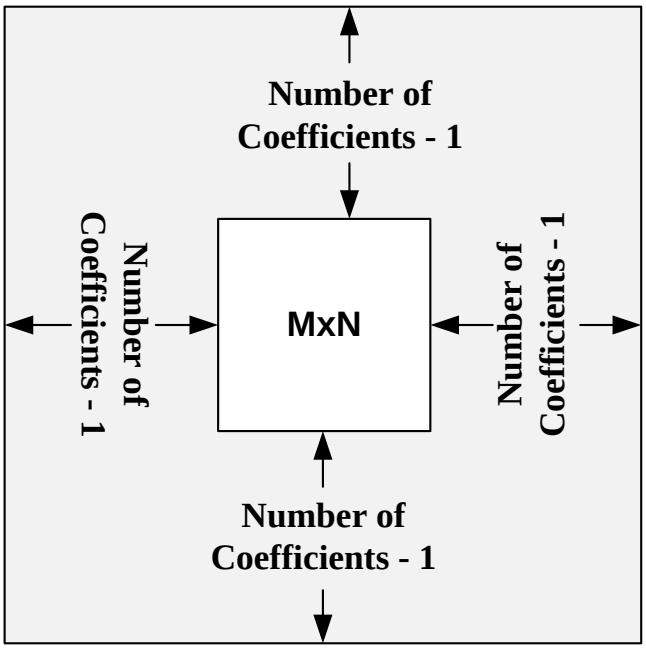

Figure 22. Filter Memory Structure.

The number of coefficients should be significantly less than either *M* or *N* so as to minimize the overhead. Thus, larger packet sizes are needed for FIR filters with larger numbers of coefficients.

The data padding for this and the subsequent separable 2-D filter application was done by the VPCS before sending the data to the FPGA. The data pad was also removed from the data with the VPCS after processing just prior to reassembly of the video frame.

For the purposes of clarity in the following sections, it should be noted that all of the FIR filters are using eight coefficients. Therefore, all references to 64x64 and 32x32 filters are actually zero-padded such that the video being processed is constructed of 78x78 and 46x46 blocks of data.

4.3. RVPPF DPR FIR Filter FPGA Core

The following section provides a description of the FIR Filter FPGA core that was used in this application [64]. For the purpose of the RVPPF, the macro core incorporated with a wrapper consisting of the PR control state machine, configurable clock divider machine, configurable clock divider logic, PR Region Memory Controller and four Block RAM or Transfer Memories. Details of the FIR core implementation are from Dally and Towles [57]. A block diagram and description of the FIR implantation as applied for the RVPPF is shown in Figure 23.

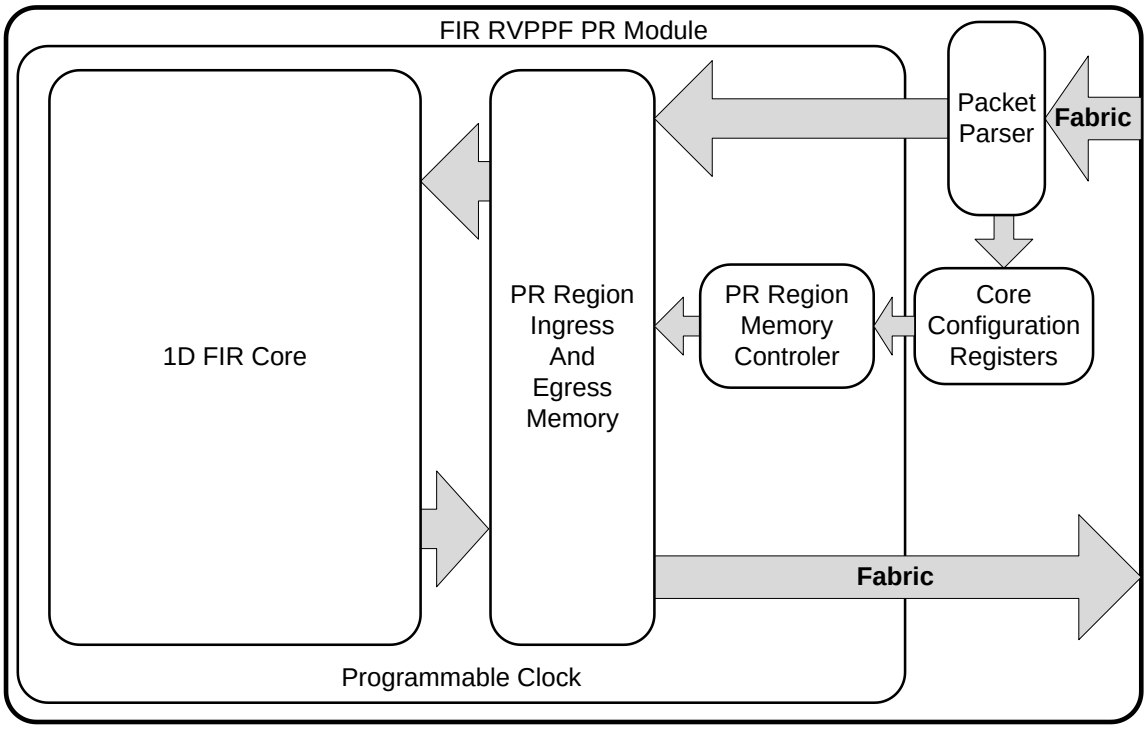

Figure 23. FIR implementation for the RVPPF

The following sections describe the two filter modes used: 1-D FIR Mode and 2-D Separable Filter Mode. The mode is selected when setting the Core Configuration Registers.

4.3.1. 1-D FIR Filter Mode

Once ingress data finishes transferring to the DPR-STM and the data complete signal is sent by the RRRC (not shown), the packet processor state machine begins to parse the data packet. For the first step in processing the packet the packet processor state machine skips the first eight bytes of data which contain the VPCS tag header (described in the software application section). The next eight bytes of data in the header contain the descriptor word which indicates whether the data in the memory is a FIR control sequence used to configure the configuration registers of the filter or a video frame to be processed by the FIR filter. Once the data is identified, the process of writing the configuration registers or processing the data with the FIR filter is performed. If the data is to be processed with the filter it is moved through the filter and the result is place back in the DPR-STM. When processing is complete the egress state machine indicates to the RRRC the data is finished processing by asserting the data ready signal. Once the AI-Receive FIFO (not shown) is ready to receive data an acknowledge handshake is sent by the RRRC to the egress state machine. The data is then moved into the AI-Receive FIFO and subsequently to the host via the PCIe bus.

4.3.2. 2-D Separable FIR Filter Mode

Once ingress data finishes transferring to the DPR-STM and the data complete signal is sent by the RRRC (not show) the packet processor state machine begins to parse the data packet. For the first step in processing the packet the packet processor state machine skips the first eight bytes of data which contain the VPCS tag header (described in the software application section). The next eight bytes of data in the header is the

descriptor word which is used to configure the Core Configuration Registers and setup the transpose memory controller, see Figure 23.

After the memory transpose is configured the data packet is read into the DPR-STM and then the data is sent to the 1-D FIR Core. As data exits the core the data is written to the DPR-STM. During this write process the data is transposed by manipulating the memory addresses the data from the 1-D FIR Core writes to.

Once the entire first pass data is written out to the DPR-STM (and now transposed) the data is sent back through the 1-D FIR Core to complete the final pass through the 1-D FIR Core. As data exits the core the data is once again transposed using address write manipulation.

Once all of the data has been processed by the 1-D FIR Core and transposed twice using memory write address manipulation it is ready to be sent back to the host PC. Once the AI-Receive FIFO (not shown) is ready to receive data an acknowledge handshake is sent by the RRRC to the egress state machine. The data is then moved into the AI-Receive FIFO and subsequently to the host via the PCIe bus.

4.3.3. RVPPF DPR FIR Filter Processing Core

Both the 64x64 and 32x32 FIR cores are comprised of several 8-bit registers, some SRAM buffers (used to store data in the data pipeline) forming a large pipeline and four Block RAMs configured as a 128 Kb dual port memory. With each clock cycle the FIR filter reads a new data word from the Block RAM and shifts it through the filter which results in a processing time for a single pixel (without considering any latency) of $t = 1/f_{clk}$ where f_{clk} is the FIR core clock frequency. This design for both filters was able to obtain a synthesis output clock frequency of 266 MHz using Xilinx XST Release

Version 14.2 [72] for synthesis and 250 MHz using Synopsys Simplify Precision G-2012.09-SP1 [73] for synthesis in the Virtex 6 device. However, once the design was constrained to the PR regions the final clock speed was reduced due to local timing restrictions to a programmable speed of 100 and 150 MHz per FIR core.

4.3.3.1. 1-D FIR Filter Speed Calculations

This section describes the processing speed calculations used to calculate the speed of the 1-D FIR filter. These calculations are when the filter is in 1-D mode or transpose is OFF.

As data enters the core, it is first moved into the DPR-STM seen in Figure 19 from the PCIe FIFO. Because the data path from the PCIe to the cores DPR-STM is configured to be 16 bits wide and the pixel data is 16 bits the number of D_{TX} cycles needed to perform the transfer is equal to the number of pixels in the *M* x *N* image plus the data header of 8 bytes or 4 x16 bit writes. The number of cycles is thus given by:

$$D_{TX} = \frac{(M*N)+4}{f_{CLK}}. \tag{7}$$

Once the data is stored in the DPR-STM the core takes six cycles (denoted by *S*) to write the configuration registers and clock divider before processing start. After setup the core start reading from the DPR-STM. Therefore, the latency T_{Lat} between the data written to DPR-STM and out of the core is equal to the pipe length *P* delay of eleven additional cycles with a clock speed f_{CLK} into the filter. The latency is thus given by:

$$T_{LAT} = \frac{P+S}{f_{CLK}}. \tag{8}$$

After processing the data, it must be sent back to the host across the framework to the PCIe FIFO. Like the receive data path this data path is 16 bits wide. Therefore the calculation for D_{TX} and D_{RX} for this step are the same as before.

Therefore, the total processing time T_{IMG} (per processing core) needed to process all the pixels of a row *M* and column *N* packet of image data with a clock frequency of f_{CLK} is given by:

$$T_{IMG} = T_{LAT} + D_{RX} + D_{TX} + \frac{M*N}{f_{CLK}}. \quad (9)$$

In Table 6, we have the size and speed calculations for the 64x64 and 32x32 video packet implementations.

Table 6. 1-D FIR Size and Speed Calculations for a Single Video Packet

Clock Speed	64x64	32x32
$\mathbf{f_{clk}}$	**100 MHz**	**100 MHz**
$\mathbf{T_{IMG}}$	**6425 ns**	**2329 ns**

4.3.3.2. 2-D Separable FIR Speed Calculations

This section describes the processing speed calculations used to calculate the speed of the 2-D Separable FIR filter. These calculations refer to the case are when the filter is in 2-D mode or transpose is ON. The calculations in this mode are the same as the 1-D implementation with the exception of the multiple passes required to perform the 2-D Separable FIR filter processing. The transpose function itself does not add any cycles to the data processing overhead because the transpose is implemented by manipulating the address writes from the to the filter core to the DPR-STM.

As data enters the core it is first moved into the DPR-STM from the PCIe FIFO. Because the data path from the PCIe to the cores DPR-STM is configured to be 16 bits wide and the pixel data is 16 bits the number of D_{TX} cycles needed to perform the transfer is equal to the number of pixels in the *M* x *N* image plus the data header of 8 bytes or 4 x16 bit writes. The number of cycles is thus given by:

$$D_{TX} = \frac{(M*N)+4}{f_{CLK}}. \tag{10}$$

Once the data is stored in the DPR-STM, the core takes six clock cycles to write the configuration registers and clock divider before start to process the first data pass. After setup, the core starts reading from the DPR-STM. Therefore, the latency T_{Lat} between the data written to DPR-STM and out of the core is equal to the pipe length *P* delay of eleven additional cycles with a clock speed f_{CLK} into the filter. The overall latency is thus given by:

$$T_{LAT} = \frac{P+S}{f_{CLK}}. \tag{11}$$

After the first pass, the data has been processed with one pass through the 1-D filter and transposed. The next step in this process is to resend the data to the 1-D filter. Unlike the first pass this process does not need six cycles, denoted previously by *S*, to write the configuration registers and clock divider before start to process data. However, the core does take fourteen cycles R_{Cycles} to reset and start receiving data. After the setup the core starts reading from the DPR-STM. Therefore, the latency T_{Lat} between the transpose data written to DPR-STM and out of the core is equal to the pipe length *P* delay of eleven additional cycles with a clock speed f_{CLK} into the filter. The latency is thus given by:

$$T_{LAT} = \frac{R_{Cycles} + \mathrm{S}}{f_{CLK}}. \tag{12}$$

Once the data is processed by the 1-D filter it is written back to the DPR-STM using the memory address transpose function. This process does not create and additional cycles.

After processing the data it must be sent back to the host across the framework to the PCIe FIFO. Like the receive data path this data path is 16 bits wide. Therefore the calculation for D_{TX} and D_{RX} are the same.

Therefore, the total processing time T_{IMG} (per processing core) needed to process all the pixels of a row *M* and column *N* packet of image data with a clock frequency of f_{CLK} is given by:

$$T_{IMG} = T_{LAT} + D_{RX} + D_{TX} + \frac{M*N}{f_{CLK}}. \quad (13)$$

In Table 7, we have the size and speed calculations for the 64x64 and 32x32 implementations.

Table 7: 2-D Separable FIR Size and Speed Calculations

Clock Speed	64x64	32x32
f_{clk}	100 MHz	100 MHz
T_{IMG}	19.239 us	6.951 us

4.3.4. RVPPF DPR FIR Host Application

The data used to test the system is 16 bit 480x640 grayscale video. For both the FIR filter application the system has two sizes of prebuilt FIR filter cores available to use as any of the thirty two processing elements in the reconfigurable FPGA logic. One of the core types which can be configured is a 64x64 and the other is a 32x32 element FIR Filter.

The VPCS is also used to remove the filter edge artifacts. This process is described in section 5.2.1 “Removing Filter Edge Effects”. This is important to note because when the VPCS breaks the image data into processing blocks and reassembles it into processed video frames the data is in 64x64 and 32x32 blocks, and during processing it is 78x78 and 46x46 data packets.

As previously stated, in order to obtain the optimum processing performance for the system the VPCS must target the algorithm combination with the least number of transfers to and from the hardware engine. Therefore, the objective of the VPCS is to create more 64x64 blocks rather than 32x32 blocks to send to the FPGA firmware for processing. This approach reduces the number total transfers for a given amount of data.

First, the VPCS needs to determine the number of 64x64 blocks *R* in a row of the input video frame. This is found by dividing the number of rows *N* by the filter size. Hence *Fr* is given by:

$$Fr = \left(\frac{N}{F}\right) = \left(\frac{480}{64}\right) = 7.5. \quad (14)$$

The VPCS must then determine the number of video frame columns that will fit in the 64x64 blocks. This is found by dividing the number of columns *M* by the filter size *Fc*. Here we have:

$$Fc = \left(\frac{M}{F}\right) = \left(\frac{640}{64}\right) = 10. \quad (15)$$

Once the *Fr* and *Fc* are determine the remainder of the rows and columns are broken into 32x32 processing elements. The formula for the number of 32x32 elements is only applied to the calculations *Fr* and *Fc* which produce a remainder (or do not evenly fit into a 64x64 block) of 1/2 . The conditions of this calculation are as follows:

- If *Fr* or *Fc* produces a remainder other than ½ the VPCS will exit and inform the user that the video frame is not the correct size for the filters loaded into the framework.
- If the calculation of *Fr* produces a remainder the number of 32x32 or small filters in a column *Fsc* is found by dividing the number of columns *M* by the 32x32 or small filter size *Fs*. Thus

  $$Fsc = \left(\frac{M}{Fs}\right) = \left(\frac{640}{32}\right) = 20. \quad (16)$$

- If the calculation of *Fc* produces a remainder the number of 32x32 or small filters in a column *Fsr* is found by dividing the number of rows *N* by the 32x32 or small filter size *Fs*.

  $$Fsr = \left(\frac{N}{Fs}\right). \quad (17)$$

4.3.4.1. Host Video Buffer and RRRC Configuration

The video buffer coupled with the Reconfigurable Round Robin Controller (RRRC) in the FIR application is used to keep the data pipe full and hence improve overall system performance and efficacy. This section describes how both are used to buffer and schedule the frames and packets in the system.

As previously shown the framework in the FIR implantation is configured with twenty four 64x64 (or 78x78 to remove the processing edge effects) filters and eight 32x32 (or 46x46 to remove the processing edge effects). This configuration creates per frame processing inefficiencies which can be removed with the combination of a video buffer and scheduling the time the framework spends moving data between 64x64 and 32x32 elements with the RRRC.

To determine the optimal size of video buffer depth needed the VPCS runs a process to determine a video boundary which the processing elements can fit into evenly. This process starts by first finding the percentage of large filter processes needed to process a frame *Fp* across the largest processing blocks loaded into the framework *Pb* (In this application the largest processing blocks are the twenty four 64x64 filters) multiplying the number of large full framework loads *Fl* to process the frame, and divides the result by the number of large processing blocks needed to process a frame *Lb*. Thus the percentage of large filter processes required is given by:

$$Fp = \frac{Pb*Fl}{Lb} = \frac{24*1}{70} = 34.28\%. \quad (18)$$

This process is then repeated incrementing only the full framework loads *Fl* until the percentage is a whole number. The state diagram in Figure 24 depicts this process.

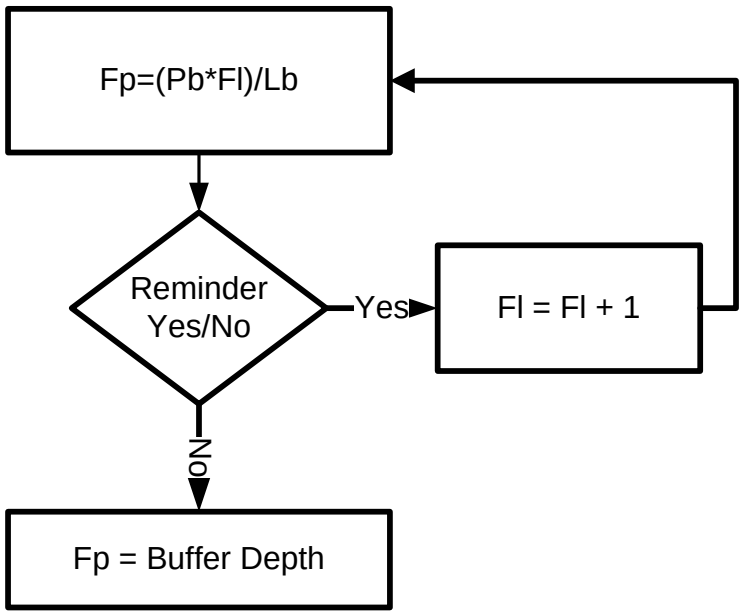

Figure 24. Buffer Depth Process

The result of this process for the 640x480 image, results in a frame buffer depth of twelve frames and thirty five full frame loads *Fl*.

The next step in the process is to set up the weights in the RRRC. This process starts by determining the number of frames the small filter set can process *Fsp* in the number of cycles calculated in the Buffer Depth Process. This is accomplished by first setting the number of *Fl* found in the Buffer Depth Process equal to the number of *Fs* small full frame loads and multiplying by the number of small processing blocks loaded into the framework *Ps*, then dividing the output by the number of small processing blocks needed to process a frame *Sb* (the number of small processing blocks needed to process a frame *Sb*). The equation is:

$$Fsp = \frac{Ps*Fs}{Sb} = \frac{8*35}{20} = 14. \quad (19)$$

This process is then repeated decrementing the full framework loads *Fl* until the number of *Fsp* is equal to the number found in the Buffer Depth Process *Fs*. The difference between the small filter processing block *Fs* and the *Fl* found in the Buffer Depth process multiplied by the number of small filters in the frame work is the number of frame weights *W* (or small filter processing skips) that must be added to the RRRC to balance the processing across the framework. Thus

$$W = Ps * (Fl - Fs) = 8 * (35 - 30) = 40. \quad (20)$$

The weights in the RRRC are distributed evenly across the processing cycles which make up the twelve frame buffer. Table 8 shows the read and write weights for the RWRRC as they are distributed across the twelve frames. The Skip Pattern column in the table depicts the number of 32x32 filter skips in a given framework load.

Table 8: RRRC Weight Calculations

Proc Cycle	64x64		No weight added32x32		Skip Pattern	Weight Added 32x32	
	Number of Processes Preformed	Percentage of Frame Processed	Number of Processes Preformed	Percentage of Frame Processed		Number of Processes Preformed	Percentage of Frame Processed
1	24	0.342857143	8	0.4	1	7	0.35
2	48	0.685714286	16	0.8	1	14	0.7
3	72	1.028571429	24	1.2	1	21	1.05
4	96	1.371428571	32	1.6	1	28	1.4
5	120	1.714285714	40	2	1	35	1.75
6	144	2.057142857	48	2.4	2	41	2.05
7	168	2.4	56	2.8	1	48	2.4
8	192	2.742857143	64	3.2	1	55	2.75
9	216	3.085714286	72	3.6	1	62	3.1
10	240	3.428571429	80	4	1	69	3.45
11	264	3.771428571	88	4.4	1	76	3.8
12	288	4.114285714	96	4.8	2	82	4.1
13	312	4.457142857	104	5.2	1	89	4.45
14	336	4.8	112	5.6	1	96	4.8
15	360	5.142857143	120	6	1	103	5.15
16	384	5.485714286	128	6.4	1	110	5.5
17	408	5.828571429	136	6.8	1	117	5.85
18	432	6.171428571	144	7.2	2	123	6.15
19	456	6.514285714	152	7.6	1	130	6.5
20	480	6.857142857	160	8	1	137	6.85
21	504	7.2	168	8.4	1	144	7.2
22	528	7.542857143	176	8.8	1	151	7.55
23	552	7.885714286	184	9.2	1	158	7.9
24	576	8.228571429	192	9.6	2	164	8.2
25	600	8.571428571	200	10	1	171	8.55
26	624	8.914285714	208	10.4	1	178	8.9
27	648	9.257142857	216	10.8	1	185	9.25
28	672	9.6	224	11.2	1	192	9.6
29	696	9.942857143	232	11.6	1	199	9.95
30	720	10.28571429	240	12	2	205	10.25
31	744	10.62857143	248	12.4	1	212	10.6
32	768	10.97142857	256	12.8	1	219	10.95
33	792	11.31428571	264	13.2	1	226	11.3
34	816	11.65714286	272	13.6	1	233	11.65
35	840	12	280	14	1	240	12

In summary, to keep the data frames aligned the VPCS application keeps a twelve frame buffer of 64x64 packets and 32x32 packets. This is to keep the data frames aligned when sending to the hardware and it keeps the data path at 100% utilization. Even though the VPCS maintains the buffer there are still more 64x64 (840) calculations that 32x32 (120) calculations on a twelve buffer. This requires loading the framework thirty five times to complete the 840 64x64 and thirty times for the 120 32x32 calculations. Therefore, the fabric would not be optimized without using the RWRRC.

4.3.5. Construction of the Video Packets

After the VPCS configures the FPGA firmware for processing the video data stream, the frame is parsed into packets and an identification header is attached to the packet. The packet header identification process is the same regardless of the size of the video frame. The following is a description of the process used to add the identification header to the packet and how the header is used by the system to process the data and reconstruct the video frame. The Packet identification process is handled per frame with an addressing scheme which can be seen in Figure 25.

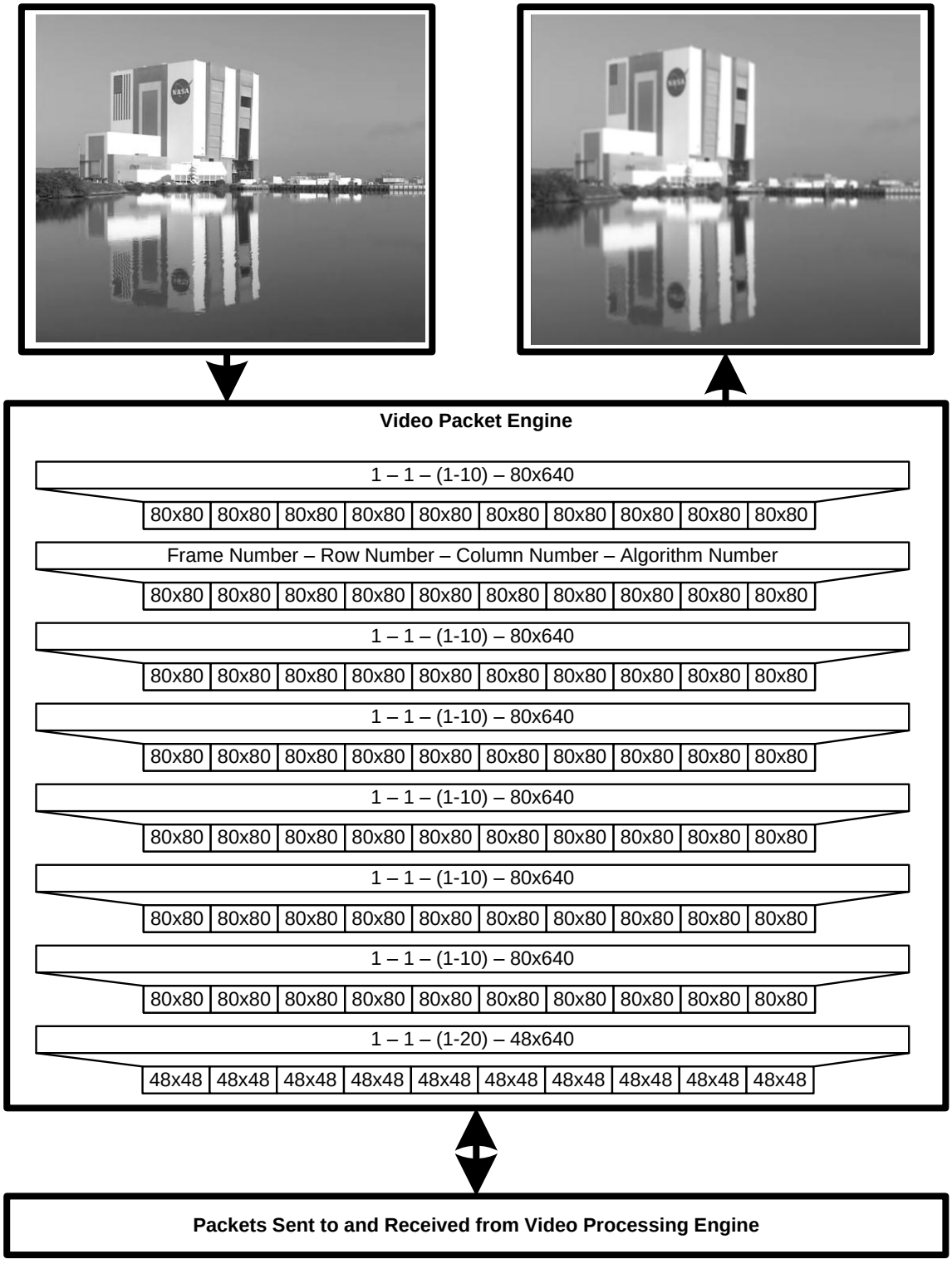

Figure 25. Packet Identification Process

To provide the ability to process video frames which may have frame-to-frame dependencies and to provide a video buffer, the hardware has a total configurable frame packet depth of 64 frames which can be processed at the same time. Therefore, the first thing the VPCS does before breaking frames up into packets is assign a frame number of 1-64 to the frames.

Once the frames in the buffer have been addressed the VPCS will break the video frames into packets. The packet sizes are directly related to the algorithms that the hardware is configured with. As previously discussed in the example the hardware is configured with twenty four 64x64 bit and eight 32x32 processing engines. To optimize the bus utilization the VPCS breaks the frames up starting with the largest data frame size moving to the smallest. In the example above the largest is algorithm available in the reconfigurable fabric is 64x64.

Starting at the top left corner of the VPCS will identify the largest row address from the data previously selected to send and tag that data with column address 1. The VPCS will then loop on the row 1 address data until all the packets have been built. In the example, row 1 address has a total of ten 64x64 column addresses. After all the column addresses have been assigned to a given row the VPCS will move to the next row. If the row is too small to be processed with the selected algorithm the VPCS will attempt to assign the next smallest algorithm to the row. In the example the next smallest row is 32x32 which has a total of a single row address and twenty column addresses. Once all the packet address headers are constructed in a block of video frames the VPCS sends the packet to the hardware via the custom hardware driver. The custom hardware driver must also add header logic to the packet so it can be roughed by the system.

When the data frames are finished processing and returned from the hardware the process is reversed the headers are removed and the frames are reconstructed.

4.3.6. RVPPF 1-D DPR FIR Results

Figure 26 shows the number of frames per second for the 1-D FIR filter implementation using twenty four 64x64 and eight 32x32 FIR filter processor cores running on the RVPPF as the latency decreases. The latency is decreased by increasing the clock speed across the processors from 100 to 150 MHz.

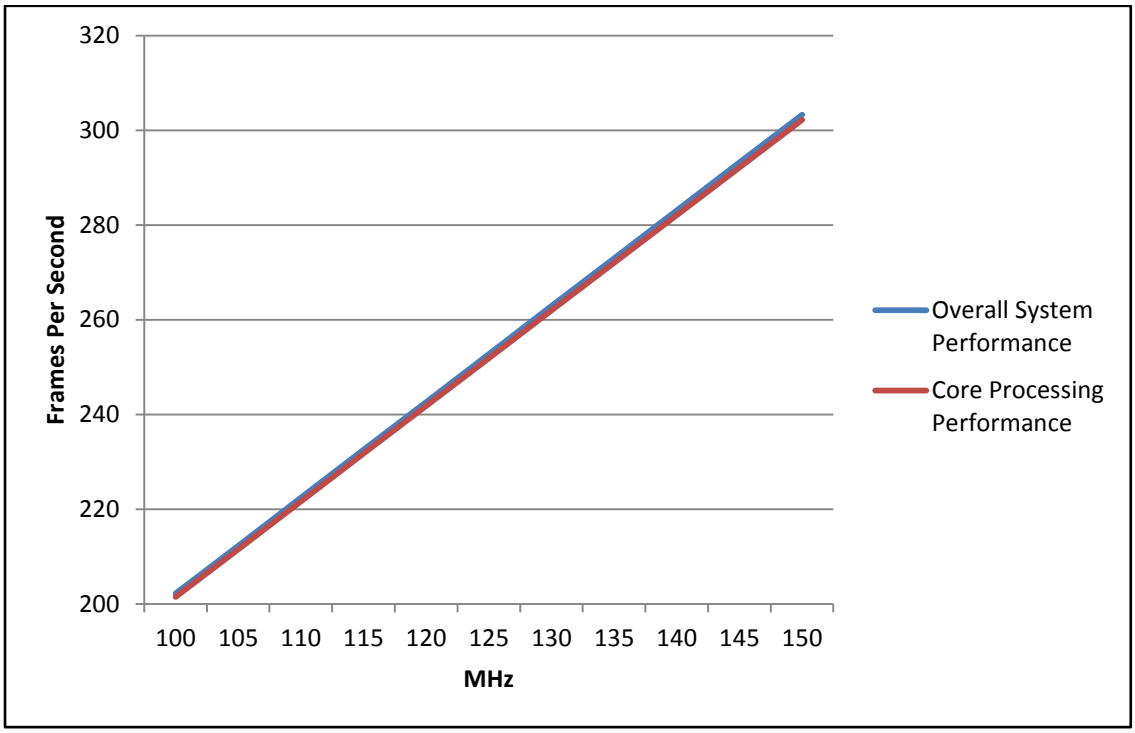

Figure 26. RVPPF 1-D DPR FIR Results

4.3.7. RVPPF 2-D Separable DPR FIR Results

Figure 27 shows performance measured in terms of the number of frames per second for the 2-D Separable FIR filter implementation using twenty four 64x64 and eight 32x32 FIR filter processor cores running on the RVPPF as the latency decreases.

The latency is decreased by increasing the clock speed across the processors from 100 to 150 MHz.

This graph also depicts how much of a processing impact the 2-D Separable FIR filter algorithm places on the individual processing nodes, and how this impact is masked by the processing framework using parallel processing and the RRRC.

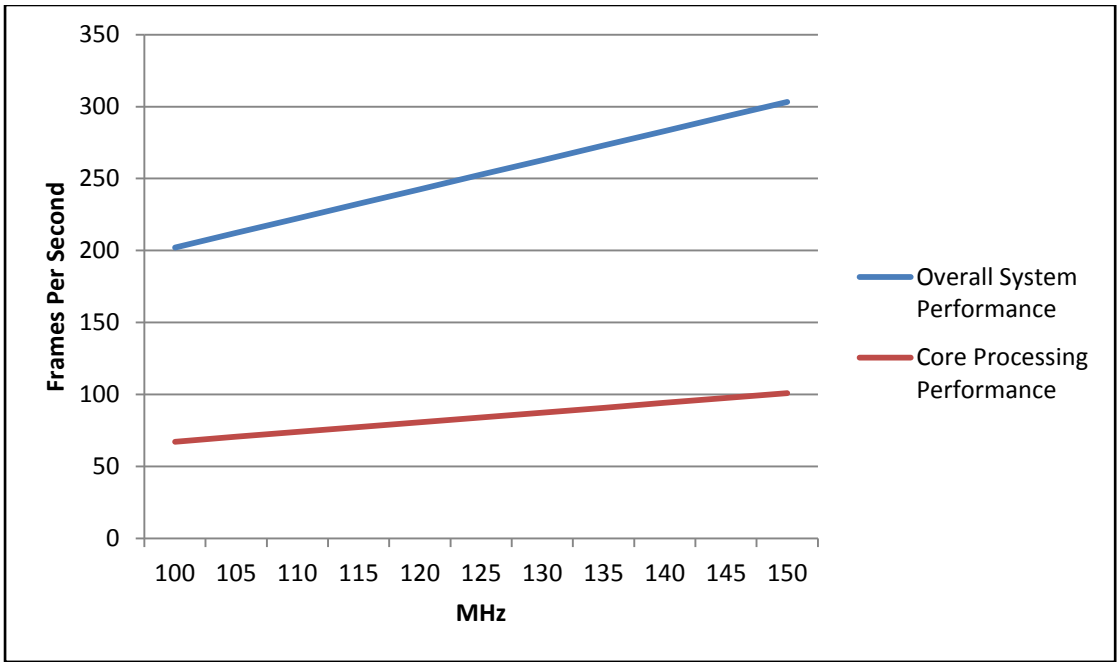

Figure 27. RVPPF 2-D Separable DPR FIR Results

4.4. RVPPF Pixel Processor

The following section is a description and implantation of a video processing application using a Pixel Processor core on the RVPPF. The primary purpose of the Pixel Processor implementation is to answer the following question: *what is the best mix of video algorithms to operate on the video frame when processing per pixel data targeted for processing with the RVPPF?*. Second, *how is unprocessed video input distributed to the dynamic algorithms configured on the RVPPF?* Third, *how are processed frames interleaved into a coherent output video stream?* In answering the above questions, the

RVPPF will demonstrate the ability to partially reconfigure the hardware, successfully deconstruct video frames into packets, send the packets across the PCIe bus to the RVPPF, process the packets in the RVPPF, and then send them back to the host to be re assembled into processed video frames. We also present an example that demonstrates that the system can reduce the clock speed of the Pixel Processor cores running on the RVPPF to dynamically optimize the power of the system.

4.4.1. RVPPF Pixel Processor FPGA Core

The following section is a description of the Pixel Processor FPGA core used to perform gamma correction this application. Gamma correction, also known as gamma compression or encoding, is used to encode linear luminance to match the non-linear characteristics of display devices. The Pixel Processor Gamma Correction is implemented using a Look-Up Table (LUT) structure that can be applied to each color or channel independently, or a single gamma correction curve can be applied to all three color channels simultaneously. For specifics on the Pixel Processor core see [52] [5].

For the purpose of the RVPPF, the core is used and a wrapper comprising of the PR control state machine, configurable clock divider logic and four Block RAM or Transfer Memory are added to the core. The low level details of the Pixel Processor core are given in Llamocca et al. [61]. A block diagram and description of the Pixel Processor implantation for the RVPPF is given in Figure 28.

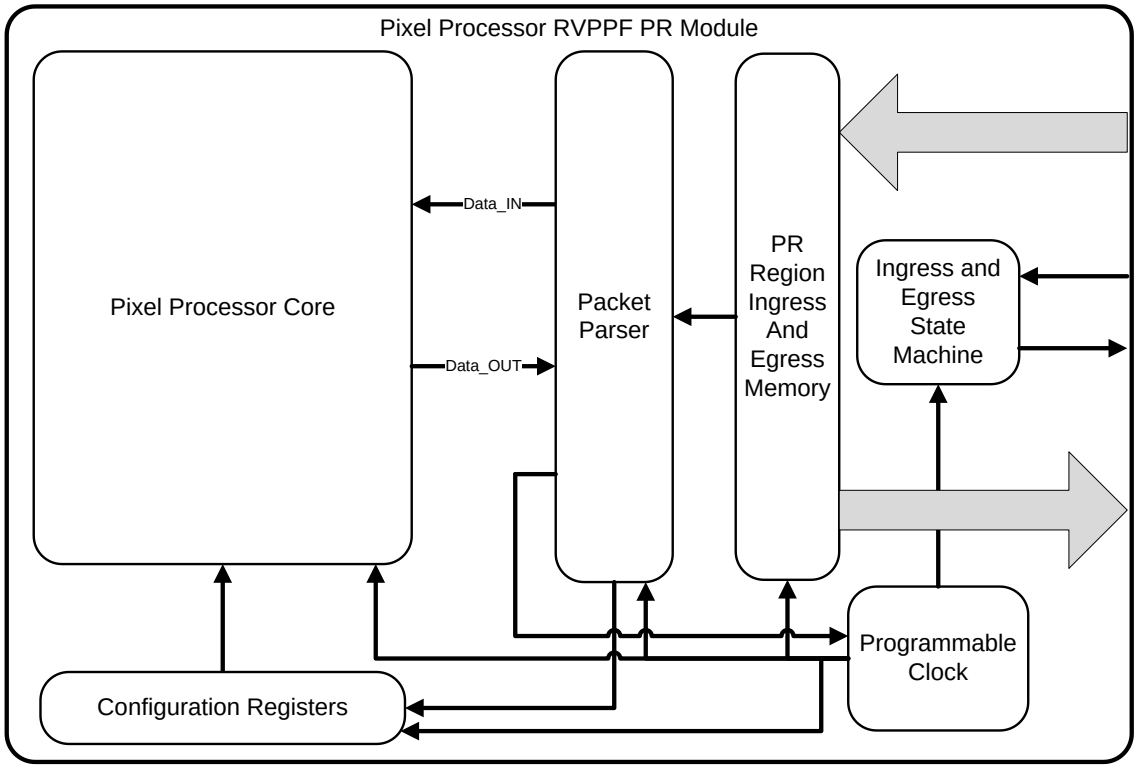

Figure 28. Pixel Processor PR Module

The following is a description of the data flow through the Pixel Processor filter module. Once, ingress data finishes transferring to the DPR-CTM and the data complete signal is sent by the RRRC (not show) the packet processor state machine begins to parse the data packet. For first step in processing the packet the packet processor state machine skips the first thirty two bits of data which contain the VPCS tag header (described in the software application section). The next thirty two bits of data in the header is the descriptor word which indicates whether the data in the memory is a Pixel Processor control sequence used to configure the configuration registers of the Pixel Processor or a video frame to be processed by the Pixel Processor. Once the data is identified the process of writing the configuration registers or processing the data with the Pixel Processor is performed. If the data is to be processed it is moved through the Pixel Processor and the result is place back in the DPR-CTM. When processing is complete the

egress state machine indicates to the RRRC the data is finished processing by asserting the data ready signal. Once the AI-Receive FIFO (not shown) is ready to receive data an acknowledge handshake is sent by the RRRC to the egress state machine. The data is then moved into the AI-Receive FIFO and subsequently to the host via the PCIe bus.

4.4.2. RVPPF Pixel Processor Host Application

The data used to test the system is 480x640 grayscale video. For the Pixel Processor, the system has one prebuilt core configured to perform gamma correction. Configuration registers are used to change the operation of the thirty two processing elements in the RVPPF. In order to obtain the optimum processing performance for the system, the video needs to be broken up into packet sizes that minimize the number of I/O transactions. The approach used for this was to break the video frames up on row boundaries which fill the DPR-CTM in each of the Pixel Processors on every transaction. The equation the host uses to break the video frame up is to first find the size of the Block RAM for each processing element, *B*, subtract the per packet header, *H*, which the Block RAM will need to store during processing and resend back the host, and divided by the sum of the video frame columns *M* multiplied by the bits per pixel, *b*. The equation the host application uses to return the height $\underline{Nn}$ of a given row size is

$$Nn = \frac{B-H}{M*b} = \frac{73728-64}{640*8} = 14.3875. \quad (21)$$

The output from this is rounded down so that the entire Nn x 640 frame packet will fit across the processing framework. The output from the row calculation is then used to find the total number of blocks size Nn x M (14 x M) the video frame will be broken up into Pn given by

$$Pn = \frac{N}{Nn} = \frac{480}{14} = 34.28. \quad (22)$$

The remaining frames *Rf* can be found by simply multiplying the height *Nn* of the processed row size by the total number of blocks size Nn x M (14x640) Pn and subtracting the total from the actual frame height *N*. The equation can be seen below:

$$Rf = N - (Nn * Pn) = 480 - (14 * 34) = 4. \quad (23)$$

Therefore, the 480x640 video frame used to test the pixel processor is broken up into thirty four 14x640 frames with a single 4x640 bit packet making up the remainder for the frame.

4.4.3. RVPPF DPR FIR Filter Block RAM

The transfer Block RAM located in each of the Pixel Processor cores is configured as DPR-CTM. In addition, when reading data from the dual port memory, the Pixel Processor algorithm does not read or write to first four data words from the memory which contains the data header added to the packet by the VPCS. Once processing is complete the unmodified header is then read back with the processed packet and used by the VPCS running on the host to reassemble the processed data.

The Block RAM usage per core can be calculated by multiplying the number of rows in a packet *Pr* times the column size of a frame *M* times the bits per pixel, *b*, then the header bits *H* must be added to the total, once this is complete the total Block RAM size available *B* is used to divide the total and give the total block ram utilization *Bu*. The algorithm RAM utilization is therefore given by:

$$Bu = \frac{Pr*M*b+H}{B} * 100. \quad (24)$$

which yields in our case the result

$$\frac{14*640*8+64}{73728} * 100 = 97.309\%. \quad (25)$$

Since a single block ram in a Xilinx Virtex 6 device is 36 Kb and a 14x640 image with 8 bit data contains 71680 bits, the size available to process in each region is limited to packets with14x640 images by the available Block RAM. This produces 97.3% block ram utilization for the thirty four packets making up the majority of the video frame and 28% block ram utilization for the remainder. Thus the achieved composite block RAM utilization for the frame is 95.3%.

4.4.4. Construction of the Video Packets for Pixel Processor

After the VPCS configures the FPGA firmware for processing the video data stream, the frame is parsed into packets and an identification header is attached to the packet. The packet header identification process is the same regardless of the size of the video frame.

The equation the VPCS uses to break up the video frames into packets for the Pixel Processor application is using the formulas above, reading the image data into a memory buffer on the host, adding the header which the VPCS uses to reconstruct the data and transferring the packet to the RVPPF.

The following is a description of the process used to add the identification header to the packet and how the header is used by the system to process the data and reconstruct the video frame. The packet identification process is handled per frame with an addressing scheme which can be seen in the Figure 29.

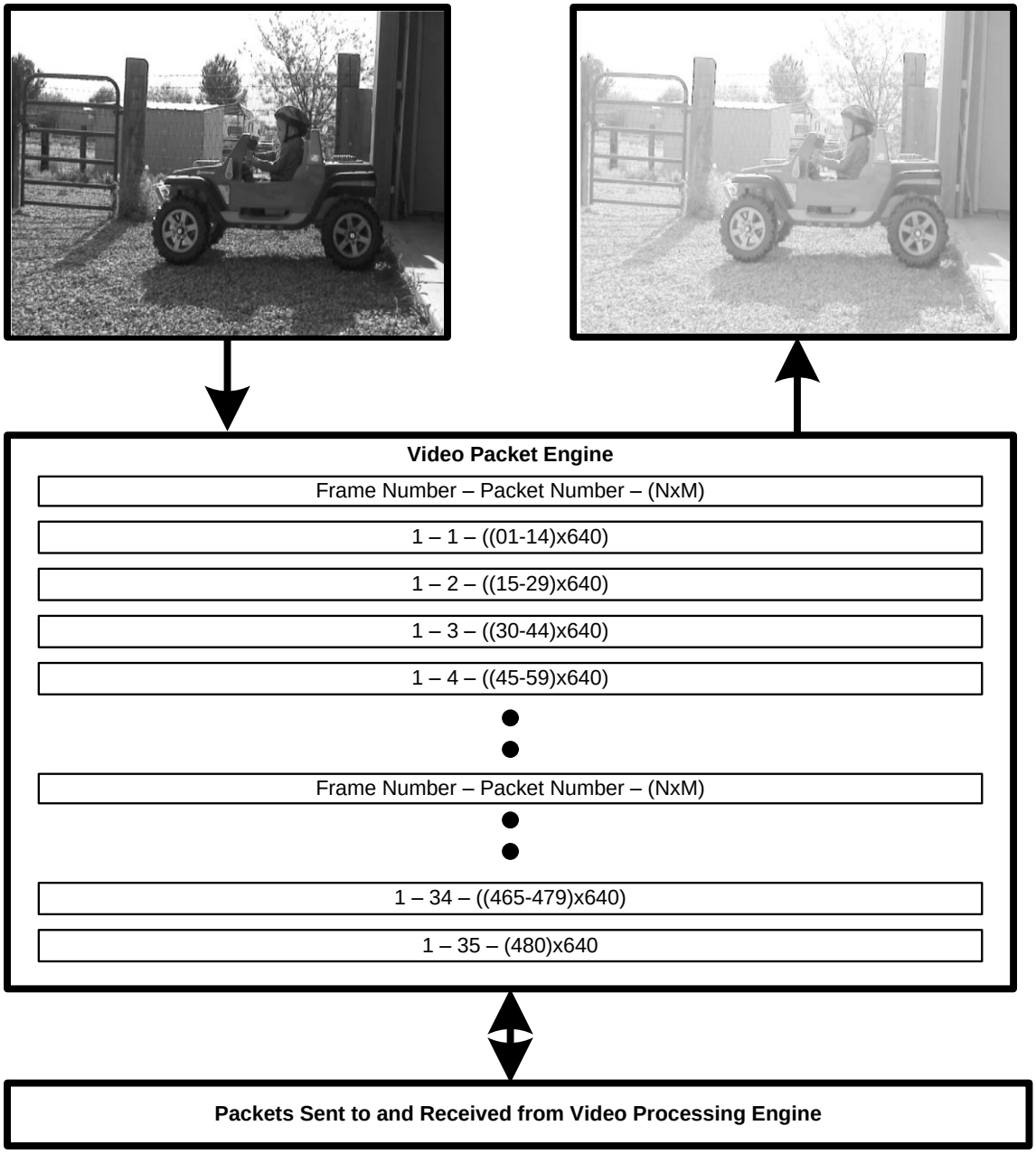

Figure 29. Packet Identification Process

To provide the ability to process video frames which may have frame-to-frame dependencies, the hardware has a total configurable frame packet depth of ten frames which can be processed at the same time. Therefore, the first thing the VPCS does before breaking frames up into packets is assign a frame number of 1-10 to the frames.

Once the frames in the buffer have been addressed the VPCS will break the video frames into packets. The packet sizes are directly related to the algorithms that the hardware is configured with. As previously discussed in the example the hardware is configured with thirty two Pixel Processing engines. To optimize the bus and DPR-CTM seen in Figure 20 utilization the VPCS breaks the frames up starting with the largest data frame size moving to the smallest. In the example above the largest packet frame size that will fit into DPR-CTM is 14x640.

Starting at the top left corner of the VPCS will identify the largest row address from the data previously selected to send and tag that data with address 1. The VPCS will then loop on the frame until all the packets have been built. In the example, packet 1 address space has a total of thirty four 14x640 addresses. After all the frames addresses have been assigned to a given packet the VPCS will move to the remaining data. The remained data is too small to max out a packet transfer however they are used to keep all the packets on video frame boundaries. In the example the remained data is 4x640. Once all the packet address headers are constructed in a block of video frames the VPCS sends the packet to the hardware via the custom hardware driver. The custom hardware driver must also add header logic to the packet so it can be roughed by the system.

When the data frames are finished processing and returned from the hardware the process is reversed the headers are removed and the frames are reconstructed.

4.4.5. RVPPF Pixel Processor Results

Figure 30 shows the performance of the Pixel Processor cores running on the RVPPF as the latency decreases. The latency is decreased by increasing the clock speed across the processors from 100 to 150 MHz.

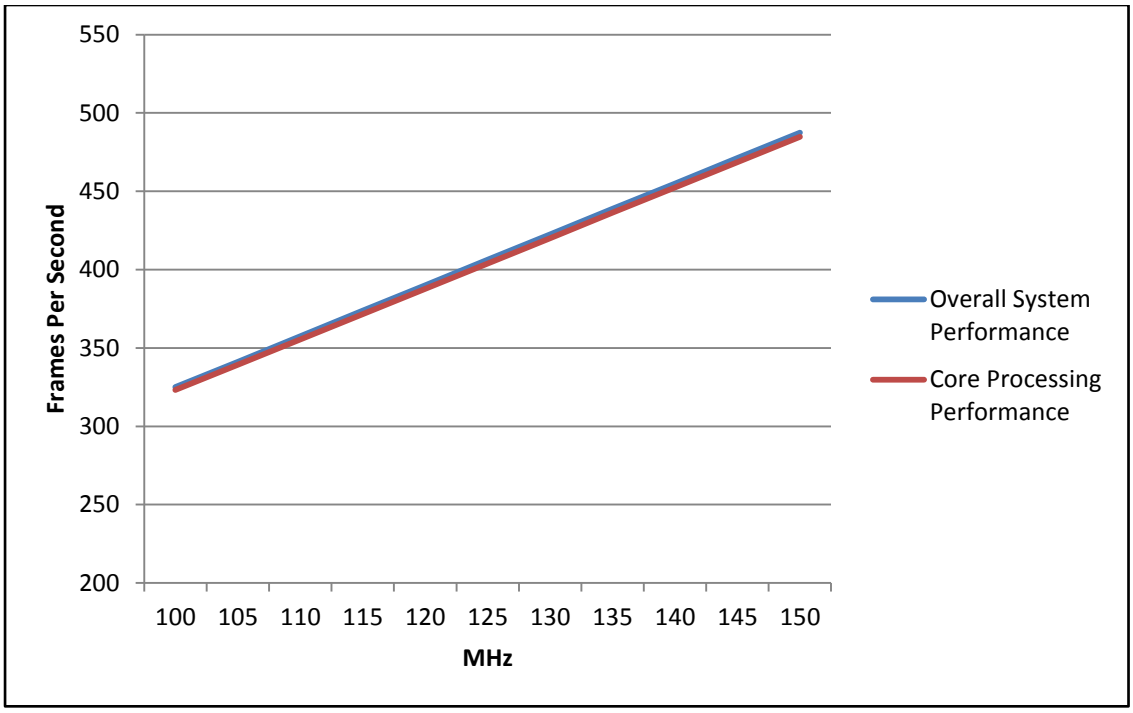

Figure 30. RVPPF Pixel Processor Results

4.5. Summary

This section described the RVPPF DPR Loopback, FIR Filter and Pixel Processor implementation developed to exploit the RVPPF. Three problems were addressed: It demonstrated how the best mix of video algorithms can be computed to operate on the video frame targeted for processing with the RVPPF. Second, how unprocessed video input can be distributed to the dynamic algorithms configured on the RVPPF. Third, how processed frames are interleaved into a coherent output video stream.

Chapter 5
Conclusions

This dissertation described design and implementation of a Reconfigurable Video Packet Processing Framework. The framework was developed and implemented on a system investigating how video packet processing can be used for applications that operate on packet video streams. The described approach exploits parallelism and re-programmability inherent in FPGAs to create and apply a distributed general-purpose computational environment applied to the video processing application. Evaluated instantiations of the proposed architectural approaches were developed incorporating several technologies including a Xilinx General Purpose Development board, FIR Filter and Pixel Processor IP soft macros, Fedora Linux, GLUnix libraries, and a custom host application. More generally, this chapter summarizes contributions of this work towards furthering the state of research into reconfigurable and adaptable computational architectures.

Section 5.0 reviews the motivations behind the direction of the research which led to the development of RVPPF as well as the overall architecture of researched system topologies. It additionally summarizes the rationale for selection of this approach to development of reconfigurable computational capabilities. Section 5.1 summarizes research contributions made by this dissertation during the design and development of the prototype system. Finally, Section 5.2 summarizes the research, general conclusions resulting from this research, implications towards the future of reconfigurable computer architectures, the role that FPGA technologies will play in development of those architectures, and finally discusses future research directions.

5.0 Review of Motivations and Design

This work demonstrated the potential for Dynamic Partial Reconfiguration (DPR) of Field Programmable Gate Array (FPGA) technology to provide a major disruptive solution to address high performance and real time processing requirements. This addresses known limitations of more traditional approaches to providing immediate and vital capabilities to increase system efficiency and resiliency. The motivation for initiating this research was to address unresolved questions regarding the degree to which DPR of FPGA technology can address known needs to improve effective computational density as well as creating new means for providing adaptive processing. To demonstrate the feasibility and utility of this new approach, the studied techniques are applied to study video processing applications which were selected as representative of high performance processing needs that can be addressed. The associated value proposition can be demonstrated for applications such as satellite-based image processing.

System hardware upgrades are typically quite complex, risky, and expensive. Current complex FPGA design flows often result in designs which are difficult, time consuming and expensive to change once they have been successfully deployed. Once these designs are operational with acceptable defects or deficiencies, there is great reluctance to consider change because change can result in nearly a complete redesign.

Furthermore, available resource provisioning is commonly critically bound by Size, Weight, and Power (SWAP). FPGA devices incorporated into these systems represent high-value, finitely-limited and hence scarce resources. Once deployed, the available FPGA logic is strictly finite. Yet even though the technology is

reprogrammable, there exist significant barriers to modification or adaptation of systems that are already are already critically bound by Size, Weight, and Power (SWAP).

The importance of this research in invested in creating realizable abilities to reconfigure the logic of such devices after deployment. This capability once developed enables revision of design logic, as necessary, to correct design deficiencies or to alter or enhance functionality. Dynamic reconfiguration enables adaptation of that same hardware to address changes to requirements or to balance or redistribute functions to different situational as needed.

Creating the means for development of cost-effective, easily-modified systems clearly are increasingly essential to address increasingly complex systems and distributed systems of systems. It is possible to address those needs but requires appropriate tools and techniques for DPR but moreover must be provided within a structured (well-posed) yet flexible architectural approach. A crucial underlying need addressed by this research was to evaluate suitable topologic architectural aspects of a deployment framework.

Successful realization will additionally require development of appropriate techniques and tools which together represent a development framework to create for complex infrastructures for FPGA Intellectual Property (IP) development and horizontal deployment.

Finally, development of the RVPPF was motivated by the increasing demand for real-time video processing with optimized Power Performance and Efficiency (PPE) which is characterized by packets of video with varying frame rates, image sizes, and jitter. The target environment for RVPPF is a general-purpose computer with an array of FPGA based processing boards and provided the basic infrastructure for enabling

effective exploration of the many open issues associated with reconfigurable computation.

5.1. Research Contributions

This section reviews the research contributions made by this dissertation. The major contributions are:

- A framework was developed to explore and implement packet based video processing and Dynamic Partial Reconfiguration. The software architecture of RVPPF provides an environment for exploring high-level issues such processor saturation and system bottle neck issues.
- Mechanisms developed to exploit inherent FPGA parallelism. Results demonstrated that to fully utilize FPGA parallelism, an algorithm must be sufficiently complex with associated processing latencies such that the algorithm speed does not exceed that which the device I/O can support.
- New results for utility and efficiency for a Partial Reconfiguration System (PRS) were demonstrated for packet-based processing. Specific performance results were developed demonstrating architectural advantages associated with a PRS implemented in the form of a Linux based Dynamically Reconfigurable System (DRS) for packet-based video processing. These results were demonstrated within a Matlab and Simulink development environment which facilitated DPR techniques to be leveraged for video processing applications and also facilitated straightforward creation of the associated test capabilities as well as efficient creation of software utilities required to measure and validate EPA system performance measures. Based

on the summary results that were developed, recommendations for further study were detailed.

- DPR techniques were used to demonstrate new advances in the state of the art in the parallel processing of multiple, digital video data streams across a single channel. This work demonstrated actual implementation of digital videos streamed passed into a FPGA fabric with a reconfigurable bus arbitration engine used to processes the digital video data streams meeting real time performance constraints necessary to sustain the stream data rates.

5.2. Future Research Directions

Creating reconfigurable highly parallelizable hardware applied to address high performance processing applications represents a significant advance in the state of art for reconfigurable computational architectures. In addition to exploring architectural topology tradeoffs, this research demonstrated the viability and utility of applying FPGA technologies in development of dynamically partial reconfiguration of computational capabilities.

Acronyms and Abbreviations

1-D	One dimensional
2-D	Two dimensional
8b-10b	Eight-bit / ten bit encoding
ACE	AXI Coherency Extensions
AHB	Advanced High-performance Bus
AHB-Lite	Lightweight version of AHB
AMBA	Advanced Microcontroller Bus Architecture
AMBA-2	Release 2 of the AMBA specification
AMBA-3	Release 3 of the AMBA specification
APB	Advanced Peripheral Bus
API	Application Programming Interface
ARM	Advanced RISC Machine
ASB	Advanced System Bus
ASIC	Application Specific Integrated Circuit
ATB	Advanced Trace Bus
ATM	Asynchronous Transfer Mode
AWRR	Adaptive Weighted Round Robin
AXI	Advanced eXtensible Interface
AXI-Lite	Lightweight version of AXI
BAR	
C	The C programming language
CDMA	Code Division Multiple Access
CLB	Combinational Logic Blocks
CLAY	Configurable Logic ArraY
CONFIG	Configuration
CRC	Cyclic Redundancy Checksum
CSC	Color Space Conversion
DC	Direct Current
DFC	Dynamic Frequency Control
DISC	Dynamic Instruction Set Computer
DLL	Data link layer
DPR	Dynamic Partial Reconfiguration
DPRR	Dynamic Partial Reconfiguration Regions
DRR	Deficit Round Robin
DSP	Digital Signal Processor
DWT	Discrete Wavelet Transforms
EA-FIFO	Egress Asynchronous FIFO
ECRC	End-to-end Cyclic Redundancy Checksum

E-FIFO	Egress FIFO
EKF	Extended Kalman Filters
EPA	Energy-Performance-Accuracy
EPRC	
EXT_PR_DONE	External PR (load) done signal
Fedora 16	Version of the Linux operating system
FDX	Full-duplex
FIFO	First In – First Out
FILO	First In – Last Out
FIR	
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GBytes	One billion bytes
Gen I	Generation 1
Gigabit	One billion bits
GPU	Graphics Processing Unit
Gbps	Giga-bits per second (billion bits per second)
GUI	Graphic User Interface
HDL	Hardware Description Language
HDX	Half-duplex
HSDPRC	High Speed Dynamic Partial Reconfiguration Controller
I/O	Input/Output
I-FIFO	Ingress FIFO
IA-FIFO	Ingress Asynchronous FIFO
ICAP	Internal Configuration Access Port
ICAP_DO	ICAP Data Out
ID	Identification
Ioctl	I/O control
ivPCL	
IP	Intellectual Property
L-App	Loopback Application
L-DPR	Loopback DPR core
L-DPR-CTM	L-DPR Combined Transfer Memory
L-DPR-STM	L-DPR Separate Transfer Memory
Linux	The Linux operating system
LSB	Least Significant Byte
LUT	Look Up Table
LUT8-to-1	Look Up Table 8-to-1 Multiplexer
MB	Mega-Byte (million bytes)
MHz	Mega-Hertz (million Hertz)

MIMD	Multiple Instruction Multiple Data
MPS	Maximum Payload Size
MSB	Most Significant Byte
Matlab	The Matlab simulation tool by MathWorks
NASA	North American Space Adminstration
NP-complete	Non-deterministic Polynomial Complete (complexity)
OS	Operating System
PC	Personal Computer
PCI	Peripheral Component Interface
PCIe	PCI Express
PGAS	Programming Model for Reconfigurable Supercomputing
PHY	Physical layer
PlanAhead	Xilinx floor planning tool
PowerPC	The PowerPC processor
PowerPC 750	Member of the PowerPC processor family
PPA	Power-Performance-Accuracy
PPE	
PR	Partial Reconfiguration
PRC	
RAM	Random Access Memory
RC	Reconfigurable Computing
RED	Random Early Discard
RRRC	Reconfigurable Round Robin Controller
RTL	Register Transfer Level
RVPPF	Reconfigurable Video Packet Processing Framework
RWRRC	Reconfigurable Weighted Round Robin Controller
s	seconds
SA	Systolic Array
SerDes	Serializer/Deserializer
SHMEM	
SIMD	Single Instruction Multiple Data
Simulink	Simulation Link interface tool for Matlab by MathWorks
SRAM	Static Random Access Memory
SWAP	Size, Weight, and Power
SystemC	The SystemC system description language
TDM	Time Division Multiplexing
TLP	Transaction Layer Packets
TMDT	Theoretical Maximum Data Throughput
V-FIR	Video FIR Fliter

Verilog	The Verilog hardware description language
Vivado	Xilinx FPGA tools environment
VHDL	VHSIC Hardware Description Language
Virtex II	2nd generation Xilinx FPGA product family
Virtex II Pro	Higher performance 2nd generation Xilinx FPGA product family
Virtex-4	4th generation Xilinx FPGA product family
Virtex-4 SX35	Member of the Xilinx Virtex 4 product family
Virtex-5	5th generation Xilinx FPGA product family
Virtex-6	6th generation Xilinx FPGA product family
Virtex 7	7th generation Xilinx FPGA product family
Virtex 7-2000T	Member of the Xilinx Virtex 7 product family
VPCS	Video Packet Configuration Software
WRED	Weighted Random Early Discard
WRR	Weighted Round Robin

References

[1] A. K. Brown and D. Reed, “Dynamic Reconfiguration in a GNSS Software Defined Radio for Multi-Constellation Operation,” in *ION GNSS*, Portland, September 2011.

[2] A. Paraboni, C. Capsoni, L. Nebuloni and M. Luccini, “A theoretical approach for the dynamic reconfiguration of an on-board antenna pattern and its performance assessment,” in *Antennas and Propagation (EUCAP)*, European Conference, April 2011.

[3] G. A. Vera, D. Llamocca, M. S. Pattichis and J. Like, “A dynamically reconfigurable computing model for video processing applications,” in *43rd Asilomar Conference on Signals*, Pacific Grove, Calif, November 2009.

[4] J. C. Hoffman and M. S. Pattichis, “A High-Speed Dynamic Partial Reconfiguration Controller using Direct Memory Access through a Multiport Memory Controller and Overclocking with Active Feedback,” *International Journal of Reconfigurable Computing*, vol. 2011, no. 2011, p. Article ID 439072, 2011.

[5] D. Llamocca, M. S. Pattichis and A. Vera, “A Dynamically Reconfigurable Parallel Pixel Processing System,” in *International Conference on Field Programmable Logic and Applications*, Prague, 2009.

[6] D. Llamocca, M. S. Pattichis and A. Vera, “A dynamically reconfigurable platform for fixed-point FIR filters,” in *International Conference on ReConFigurable Computing and FPGAs*, Cancun, 2009.

[7] D. Llamocca, M. Pattichis and A. Vera, “A Dynamically Reconfigurable Parallel Pixel Processing System,” in *International Conference on Field Programmable Logic and Applications FPL’2009*, Prague, 2009.

[8] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” in *International Symposium on Field Programmable Gate Arrays*, Monterey, CA., 2006.

[9] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203-215, 2007.

[10] W. Maogang, A. Ranjan and S. Raje, “Multi-million gate FPGA physical design

challenges," in *International Conference on Computer Aided Design*, 2003.

[11] F. Fons, "Embedded Electronic Systems Driven by Run-Time Reconfigurable Hardware," University Rovira i Virgili, Tarragona, 2012.

[12] G. S. A. G. a. C. Y. Vikas Aggarwal, SCF: A Framework for Task-Level Coordination in Reconfigurable, Heterogeneous Systems, ACM Transactions on Reconfigurable Technology and Systems, Vol. 5, No. 2, Article 7, pp. 1-23, June 2012.

[13] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez and C. A. Zeferino, "SPIN: a scalable, packet switched, on-chip micro-network," *IEEE Conference on Design, Automation and Test*, pp. 70-73, 2003.

[14] K. Papadimitriou, A. Anyfantis and A. Dollas, "An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems," *Instrumentation and Measurement*, pp. 1642-1651, June 2010.

[15] H. Tan and R. F. DeMara, "A Multilayer Framework Supporting Autonomous Run-Time Partial Reconfiguration," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 5, pp. 504-516, 2008.

[16] T. Marconi, "Efficient Runtime Management of Reconfigurable Runtime Resources," Delft, The Netherlands, 2011.

[17] S. Hauck and W. D. Wilson, "Runlength compression techniques for FPGA configuration," in *7th Annual IEEE Symp. on Field-Programmable Custom Computing Machines*, 1999.

[18] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam and A. Dasu, "Dynamically reconfigurable systolic array accelerators: a case study with extended Kalman filter and discrete wavelet transform algorithms," *IET Comput. Digit. Tech.*, vol. 4, no. 2, pp. 126-142, 2010.

[19] J. Carver, R. N. Pittman and A. Forin, "Relocation and automatic floor-planning of FPGA partial configuration bitstreams," Microsoft Research, WA, 2008.

[20] B. Krill, A. Amira, A. Ahmad and H. Rabah, "A New FPGA-Based Dynamic Partial Reconfiguration Design Flow and Environment for Image Processing Applications," *EUVIP*, pp. 226-231, 2010.

[21] J. A. Clemente, J. Resano, C. González and D. Mozos, "A Hardware Implementation

of a Run-Time Scheduler for Reconfigurable Systems," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems,* vol. 19, no. 7, pp. 1263-1276, 2011.

[22] P. Banerjee, S. Bhattacharjee and S. Sur-Kolay, "FAST FPGA PLACEMENT USING SPACE-FILLING CURVE," in *International Conference on Field Programmable Logic and Applications*, 24-26 Aug. 2005.

[23] A. Agarwal, J. Cong and B. Tagiku, "Fault tolerant placement and defect reconfiguration for nano-FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design,* 10-13 Nov 2008.

[24] J. Y. Hur, K. Goossens, L. Mhamdi and M. A. Wahlah, "Comparative analysis of soft and hard on-chip interconnects for field-programmable gate arrays," *IET Computers & Digital Techniques,* 14th July 2012.

[25] W. Wong, "Understanding FPGA Processor Interconnects," *Electronic Design*, June 17, 2012.

[26] P. ARM Holdings, "Licensing," [Online]. Available: http://www.arm.com/products/buying-guide/licensing/index.php. [Accessed April 2013].

[27] D. Galloway, D. Karchmer, D. Chow, D. Lewis and D. Rose, "The Transmogrifier: The University of Toronto Field-Programmable System," in *Second Canadian Workshop on Field-Programmable Devices*, Kingston, June 1994.

[28] D. Lewis, D. Galloway, M. Ierssel, J. Rose and P. Chow, "The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System," *IEEE Transactions on VLSI*, vol. 6, no. 2, pp. 188-198, June 1998.

[29] Berkeley, "BEE," Berkeley, 1 April 2012. [Online]. Available: http://bwrc.eecs.berkeley.edu/Research/BEE/. [Accessed 1 April 2012].

[30] G. Wigley and D. Kearney, "The Management of Applications for Reconfigurable Computing using an Operating System," in *Seventh Asia-Pacific Computer Systems Architecture Conference*, Melbourne, Australia, 2002.

[31] K. Bazargan, R. Kastner and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems," *IEEE design & Test of Computers,* vol. 17, pp. 68-83, 2000.

[32] A. Ferrucci, M. Martin, T. Geocaris, M. Schlag and P. K. Chan, "ACME: A Field-

Programmable Gate Array Implementation of a Self-Adapting and Scalable Connectionist Network," *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays,* 1994.

[33] P. K. Chan, M. Schlag and M. Martin, "A Reconfigurable Prototyping Board Using Field-Programmable Gate Arrays," *Proceedings of the 1st International ACM/SIGDA Workshop on Field-Programmable Gate Arrays,* vol. 1, no. 1, pp. 47-51, 1992.

[34] H. Hogl, A. Kugel, J. Ludvig, R. Manner, K. H. Noffz and R. Zoz, "A Second Generation FPGA-Processor for ATLAS," in *ATLAS internal note DQS-NO-026*, CERN, 1994.

[35] J. M. Arnold, "1993," in *The Splash 2 Processor and Applications*, Calif., Proc. Int'l Conf. Computer Design, 1993, pp. 482-485.

[36] T. EDU, "TM3," 2 April 2013. [Online]. Available: http://www.eecg.toronto.edu/~tm3. [Accessed 2013 April 2013].

[37] D. Smith and D. Bhatia, "Reconfigurable and Adaptive Computing Environment," in *In 6th International Workshop 117 on Field-Programmable Logic and Applications*, Darmstadt, Germany, September, 1996.

[38] W. Eatherton, T. Schiefelbein and H. Pottinger, "An FPGA-based Reconfigurable Coprocessor Board Utilizing a Mathematics of Arrays," Computer Science Department, University of Missouri-Rolla, 1995.

[39] G. M. Quenot, I. C. Kraljic, J. Serot and B. Zavidovique, "A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping," *In IEEE Workshop on FPGAs for Custom Computing Machines*, vol. 1994, no. 1, pp. 91-100, 1994.

[40] H. Hogl, A. Kugel, J. Ludvig, R. Manner, K. H. Noffz and R. Zoz, "A Second Generation FPGA-Processor for ATLAS," ATLAS internal note DQS-NO-026, CERN, 1994.

[41] T. M. Systems, "RAM SAN," Texas Memory Systems, Jan 2000. [Online]. Available: http://www.ramsan.com/files/f000034.pdf. [Accessed 1 May 2013].

[42] W. T. Jennings, "Switching Complexities," RIO Grande Semiconductor, Oct, 2004.

[43] G. V. M. Shreedhar, "Efficient Fair Queueing Using Deficit Round Robin," *ACM SIGCOMM Comp. Comm. Rev.*, vol. 25, no. 4, p. 231, Oct. 1995.

[44] S. Koehler, S. Greg and A. D. George, “Platform-aware bottleneck detection for reconfigurable computing applications,” *ACM Trans. Reconfigurable Technol.,* no. August 2011, p. 28, 2011.

[45] M. J. Wirthlin and B. L. Hutchings, “A dynamic instruction set computer,” *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 99-107, April 1995.

[46] J. Curreri, G. Stitt and A. George, “Communication Visualization for Bottleneck Detection of High-Level Synthesis Applications,” in *International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA)*, Monterey, CA, Feb. 22-24, 2012.

[47] M. Elteir, H. Lin and W. Feng, “Performance Characterization and Optimization of Atomic Operations on AMD GPUs,” in *IEEE Cluster conference,* Austin, TX, Sep. 26-30, 2011.

[48] A. Jara-Berrocal and A. Gordon-Ross, “An Integrated Development Toolset and Implementation Methodology for Partially Reconfigurable System-on-Chips,” in *IEEE International Conference on Application-specific Systems,* Santa Monica, CA., Sep. 11-14, 2011.

[49] M. Huang, K. Narayana, H. Simmler, O. Serres and T. El-Ghazawi, “Reconfiguration and Communication-Aware Task Scheduling for High-Performance Reconfigurable Computing,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS),* vol. 3, no. 4, pp. 20:1-20:25, Nov. 2010.

[50] B. Holland, A. George, H. Lam and M. Smith, “An Analytical Model for Multi-Level Performance Prediction of Multi-FPGA Systems,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS),* to appear.

[51] V. Aggarwal, A. George, C. Yoon, K. Yalamanchili and H. Lam, “SHMEM+: A Multilevel-PGAS Programming Model for Reconfigurable Supercomputing,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS),* to appear.

[52] V. Aggarwal, A. Georg, K. Yalamanchili, C. Yoon, H. Lam and G. Stitt, “Bridging Parallel and Reconfigurable Computing with Multilevel PGAS and SHMEM+,” in *High-Performance Reconfigurable Computing Technology and Applications Workshop (HPRCTA) at SC'09,* Portland, OR., Nov. 15, 2009.

[53] A. d. c. p. f. i. a. v. p. applications, “43rd Asilomar Conference on Signals,” in *Systems and Computers,* Pacific Grove, Nov., 2009.

[54] D. Llamocca, M. Pattichis and A. Vera, “Partial Reconfigurable FIR - Article ID 357978,” *International Journal of,* vol. 2010, p. 14, 2010.

[55] D. Llamocca and M. Pattichis, “Real-time dynamically reconfigurable 2-D,” in *2010 IEEE Southwest Symposium on Image*, Austion, Tx, 2010.

[56] D. Llamocca, M. S. Pattichis and A. Vera, “Dynamic Partial,” in *Infotech Conference at Aerospace*, Atlanta, GA, 2010.

[57] R. Kumar and A. Gordon-Ross, “MACS: A Minimal Adaptive Routing Circuit-Switched Architecture for Scalable and Parametric NoCs,” in *Intl. Conference on Field-Programmable Logic and Applications (FPL)*, Prague, Czech Republic, Aug. 31 - Sep. 2, 2009.

[58] J. Dally and B. Towles, Principles and Practices of Interconnection Networks, Morgan Kaufmann, 2004.

[59] P. Pratim Pande, C. Grecu, M. Jones, A. Ivanov and R. Saleh, “Performance evaluation and design trade-offs for network-on-chip interconnect architectures,” *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 1025-1040, 2005.

[60] K. D., M. Kim and G. E. Sobelman, “CDMA-based NoC architecture,” *IEEE Conference on Circuits and Systems*, vol. 1, pp. 137-140, 2004.

[61] R. F. Hodson, K. Somervill, J. Williams, N. Bergman and R. Jones III, “An Architecture for Reconfigurable Computing in Space,” MAPLD, 2005.

[62] J. Gause, P. Cheung and W. Luk, “Reconfigurable computing for shape-adaptive video processing,” *Computers and Digital Techniques,* vol. 151, no. 5, pp. 313-320, 17 Sept. 2004.

[63] F. Fons, M. Fons and E. Cantó, "Run-time self-reconfigurable 2D convolver for adaptive image processing," *Microelectronics Journal,* vol. 42, no. 1, pp. 204-217, 2011.

[64] D. Llamocca, Dynamically Reconfigurable Management of Energy, Performance, and Accuracy applied to Digital Signal, Image, and Video Processing Applications, Albuquerque: University of New Mexico, 2012.

[65] D. Llamocca, Dynamically Reconfigurable Management of Energy, Performance, and Accuracy applied to Digital Signal, Image, and Video Processing Applications, Albuquerque: University of New Mexico, 2012.

[66] D. Llamocca, M. Pattichis and A. G. Vera, “Partial Reconfigurable FIR Filtering System Using Distributed Arithmetic,” *International Journal of Reconfigurable Computing,* vol. 2010, p. 14, 2010.

[67] Xilinx, ML605 Hardware, San Jose: Xilinx, 2012.

[68] Xilinx, Virtex-6 FPGA Configuration, San Jose: Xilinx, 2012.

[69] Xilinx, PlanAhead User Guide - UG632 (v 11.4), San Jose: Xilinx, 2009.

[70] J. Wiltgen and J. Ayer, Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions, San Jose: Xilinx, 2011.

[71] PCISIG, "PCIe® Base 3.0 Specification," VTM Group, [Online]. Available: http://www.pcisig.com/specifications/pciexpress/base3/. [Accessed 26 4 2013].

[72] Xilinx, Virtex-6 FPGA Memory Resources - UG363 (v1.6), San Jose: Xilinx, 2011.

[73] Xilinx, “Xilinx ISE Software,” Xilinx, 31 March 2013. [Online]. Available: http://www.xilinx.com/support/download/index.htm. [Accessed 31 March 2013].

[74] Synopsys, “Synplify Premier,” Synopsys, 31 March 2013. [Online]. Available: http://www.synopsys.com/home.aspx. [Accessed 31 March 2013].

[75] Wikipedia, “Wikipedia,” [Online]. Available: http://en.wikipedia.org/wiki/Amdahl's_law. [Accessed 28 Jan. 2013].

[76] Wikipedia, “Gustafson's_law,” [Online]. Available: http://en.wikipedia.org/wiki/Gustafson's_law. [Accessed 28 Jan. 2013].

[77] D. Llamocca, Dissertation Dynamically Reconfigurable Management of Energy, Performance, and Accuracy applied to Digital Signal, Image, and Video Processing Applications, Albuquerque: University of New Mexico, 2011.