NANYANG TECHNOLOGICAL UNIVERSITY

TIME-MULTIPLEXED FPGA OVERLAYS WITH LINEAR INTERCONNECT

LI XIANGWEI

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University

in partial fulfilment of the requirement for the degree of

Doctor of Philosophy

2018

Acknowledgements

Foremost, I would like to express my heartiest thanks to my advisor Associate Professor Douglas Maskell for providing me such a wonderful opportunity to explore and work in the field of reconfigurable computing. His rigorous research attitude and insightful guidance inspired me to realize the power of critical reasoning and helped me grow into an independent researcher. He has given me all the freedom to pursue my research, while unobtrusively ensuring that I do not deviate from the right direction throughout my journey towards this degree. To be honest, this thesis would not have been possible without his support and advice.

I gratefully acknowledge the contributions of Associate Professor Suhaib Fahmy who has given me many critical and instructive comments for the papers we have cooperated.

I would also like to express my gratitude to my senior fellow Dr. Abhishek Jain who has provided numerous help to me and whose great passion in this research field has deeply motivated me to move forward firmly.

I am profoundly grateful to work as a member of the Hardware and Embedded Systems Lab (HESL). Thanks to my colleagues and fellow research scholars, especially Dr. Jiang Lianlian, Dr. Cui Yingnan, Dr. Ye Deheng, Dr. Yang Liwei, Dr. Mao Fubing, Dr. Luo Tao, Dr. Gu Xiaozhe, Dr. Chen Hao, Miss Gao Yiyi, Mr. Shivaraman Nitin and Miss Rathore Vijeta for their constant support and encouragement. A special thank must go to our laboratory executive, Mr. Chua Ngee Tat, for his professional technical assistance.

My heartfelt thanks go to Miss Demi Zhang for her love and understanding during the second half stressful trip of my Ph.D study.

Last but not the least, I am forever indebted to my family for their unconditional love, trust and support. I thank them for their continuous patience and encouragement during the toughest time when I was struggling for the research work. I dedicate this thesis to them.

Abstract

The benefits of FPGAs over processor-based systems have been well established, however apart from specialist application domains, such as digital signal processing and communications, these platforms have not seen wide usage. Poor design productivity has been a key limiting factor, preventing the mainstream adoption of FPGAs and restricting their effective use to experts in hardware design. Coarse-grained overlay architectures have been proposed as a possible solution for improving design productivity by offering fast compilation and software-like programmability. These overlays can either be spatially configured (SC), with one complete functional unit (FU) allocated to each compute kernel operation and a routing network which is essentially static during computation, or, multiplexed, with the FUs and interconnect being shared between kernel operations.

This thesis examines an overlay architecture based on a simple linear interconnected array of time-multiplexed (TM) functional units. Sharing the FUs among kernel operations should significantly reduce the FPGA resource overhead compared to an SC overlay which requires one FU for each operation along with a fully functional routing network to support connections to neighboring FUs. The linear interconnected array of TM FUs should also result in reduced instruction storage and interconnect resource requirements compared to other TM overlays, again resulting in a more area efficient overlay.

In order to minimize the use of the fine-grained FPGA resource, we make use of the DSP block to design a fast, fully-pipelined, architecture-aware FU implementation, better targeting the capabilities of the FPGA. The results presented show a significant reduction of up to 85% in FPGA resource requirements compared to existing throughput oriented overlay architectures, with an operating frequency which approaches the theoretical limit for the FPGA device. A number of architectural enhancements are then proposed to improve the performance of the DSP block based FU.

The overlay subsystem is then integrated into complete hardware accelerator systems, along with memory interfaces, to an ARM processor or a host CPU. To achieve this, we investigate two different memory solutions based on AXI and PCIe interfaces, namely Xillybus and RIFFA. The performance of these hardware accelerators for a range of benchmarks is investigated and performance results are presented. The proposed AXI-Xillybus-V3 overlay system is also compared to a state-of-art TM overlay, namely VectorBlox MXP. The comparison results show the AXI-Xillybus-V3 achieves a very area efficient implementation at the expense of around half of the throughput (limited by AXI-Xillybus using a 32-bit bus compared to the 64-bit bus used by VectorBlox MXP). The proposed RIFFA-V3 overlay system shows a 3.6× better performance compared to the PCIe-Xillybus-V3, and a 5.7× better performance than AXI-Xillybus-V3, but at the cost of a larger BRAM consumption.

Contents

List of Figures

List of Tables

List of Abbreviations

ALAP	As Late As Possible
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
CGRA	Coarse Grained Reconfigurable Architecture
CLB	Configurable Logic Block
CPU	Central Processing Unit
DeCO	DSP enabled Cone-shaped Overlay
DFG	Data Flow Graph
DMA	Direct Memory Access
DSP	Digital Signal Processing
DySER	Dynamically Specialized Execution Resources
FF	Flip Flop
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FU	Functional Unit

GOPS	Giga Operations Per Second
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
IF	Intermediate Fabric
II	Initiation Interval
LE	Logic Element
LUT	Lookup Table
NN	Nearest Neighbor
PAR	Placement and Routing
PR	Partial Reconfiguration
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
SC	Spatially Configured
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
TM	Time Multiplexed
VLIW	Very Long Instruction Word

List of Publications

- **X. Li**, A. K. Jain, D. L. Maskell, and S. A. Fahmy, "A Time-Multiplexed FPGA Overlay with Linear Interconnect", *in Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, Dresden, Germany, March 2018.

- **X. Li**, C. F. Phung, D. L. Maskell, "FPGA Overlays: Hardware-based Computing for the Masses", *in Proceedings of 8th International Conference on Advances in Computing, Electronics and Electrical Technology (CEET)*, KL, Malaysia, February 2018.

- A. K. Jain, **X. Li**, P. Singhai, D. L. Maskell, and S. A. Fahmy, "DeCO: A DSP Block Based FPGA Accelerator Overlay With Low Overhead Interconnect", *in Proceedings of the International Symposium on Field Programmable Custom Computing Machines (FCCM)*, Washington, DC, June 2016.

- **X. Li**, A. K. Jain, D. L. Maskell, and S. A. Fahmy, "An Area-Efficient FPGA Overlay using DSP Block based Time-multiplexed Functional Units", *in 2nd International Workshop on Overlay Architectures for FPGAs (OLAF)*, Monterey, CA, February 2016.

- A. K. Jain, **X. Li**, S. A. Fahmy, and D. L. Maskell, "Adapting the DySER Architecture with DSP Blocks as an Overlay for the Xilinx Zynq", *ACM SIGARCH Computer Architecture News* vol. 43 no. 4, pp. 28-33, 2016.

Chapter 1

Introduction

Modern FPGAs have seen a rapid growth in logic density along with the integration of CPU, GPU, and other hard silicon modules. To achieve the best accelerator performance, these FPGAs are often custom designed, using conventional RTL hardware design techniques, and as such, have only found mainstream applicability in specific applications such as digital signal processing and communications. This is because design productivity issues, particularly the difficulty of hardware design and the long compilation times, are major stumbling blocks to the widespread adoption of FPGA based accelerators in general purpose computing [6, 7].

Traditionally, text-based hardware description languages (HDL) are used to define the behavior of the FPGA. However, getting the best performance from the HDL implementation still needs a good understanding of the target technology's capabilities and of basic hardware concepts such as pipelining and synchronization. Additionally, because of the fine granularity of the FPGA resource, the design compilation time is significant. It takes hours or even days to compile a very large design due to the fine-grained placement and routing used in the FPGA implementation. Even for the case where just a few lines of HDL code change, the traditional FPGA CAD tools have to go through the whole process (including synthesis, mapping, placement and routing) to generate a new bitstream to program the device. This design process greatly slows down the development progress of FPGA designs and to some extent, hinders the widespread adoption of FPGAs.

High-level synthesis (HLS) has been widely adopted by EDA vendors to address some of the design productivity issues by providing a higher level of abstraction for the hardware, hiding much of the low-level detail. Typical HLS tools such as Xilinx Vivado HLS [8], Altera SDK for OpenCL [9], and UoT LegUp [10] have been developed to interpret a high level language description of a user application and convert it into low-level RTL. Using HLS tools, there is less of a requirement for hardware specialization as custom digital logic circuits can be generated automatically with high performance. However, while HLS techniques alleviate the design productivity problem to some extent, the back-end flow requires very long compilation times, particularly for large designs, contributing to long design cycles and the lack of mainstream adoption of FPGAs by software designers who are used to rapid design iterations.

Because of these long design cycles, researchers have investigated other techniques for improving design productivity. One of these techniques is to use a virtual hardware representation which overlays the original FPGA fabric, referred to as an overlay architecture (or overlay).

An overlay is a virtual configurable architecture, implemented over the physical fine-grained FPGA fabric, thus enabling programmability at a higher level of abstraction [11]. Overlay architectures promise to tackle the "programmability wall" of FPGAs by avoiding the tedious fine-grained placement and routing process. Programming an overlay is similar to configuring an FPGA, except that configuration is also performed at a higher level, typically at the word or functional block level, rather than at the bit level. While overlays allow high-level programmability with a significantly reduced compilation time, these advantages are not available for free. They generally come at the cost of a lower performance with significantly more FPGA resource used than for an equivalent design mapped directly to FPGA. Even flexibility can be sacrificed as many overlays are specific to a set of applications [12, 13]. As such, a significant research effort has been applied to reducing the overlay area overhead and improving the throughput.

Overlays can be broadly classified based on the run-time configurability of their FUs. If an FU has a single fixed functionality at run-time, the overlay is referred to as spatially configured (SC), while if the FU changes its operation on a cycle-by-cycle[1] basis, the overlay is referred to as time-multiplexed (TM).

In an SC overlay, an FU executes a single arithmetic operation and data is transferred be-

[1]cycle-by-cycle means at each clock cycle

tween FUs over a programmable, but temporally dedicated, point-to-point link. That is, both the FU and the interconnect are unchanged while a compute kernel is executing. This results in a fully pipelined, throughput oriented datapath executing one kernel iteration per clock cycle, thus having an initiation interval (II) [14] between kernel data packets of one. The area overheads of SC overlays, in particular the large interconnect resource requirements, have limited the use of these overlays to very small compute kernels in practical FPGA-based systems [15]. This means that in a typical application a number of different kernels would need to be mapped to the overlay as the application executes to achieve the best application acceleration. Thus the kernel context switch time is also an important consideration in the efficient operation of an overlay [16]. Some of the current overlays utilize partial reconfiguration to reduce the overlay area, in particular the interconnect resources, by trading off runtime connection flexibility [17]. However, while faster than a complete FPGA reconfiguration, partial reconfiguration still results in a significant context switch overhead, which significantly impacts an application's runtime.

A number of TM overlays have been proposed which share the functional units among kernel operations in an attempt to reduce overlay resource requirements. Time-multiplexing the FU can significantly reduce the FU and interconnect resource requirements but at the cost of a higher II and hence a reduced throughput. The most common time-multiplexed overlays have both a TM FU and a TM interconnect network. Time-multiplexing the overlay allows it to change its behavior on a cycle-by-cycle basis while the compute kernel is executing, thus allowing sharing of the limited FPGA resources. However, in many cases the storage requirements for instructions are very large, resulting in a significant area overhead. This is mainly due to: the scheduling strategy; the execution model; and the design of the overlay architecture, which limits the scalability of the overlay while also impacting the kernel context switch time. Additionally, many of these overlays are not architecture-focused and hence the FU operates at a relatively slow frequency.

1.1 Motivation

While existing overlays allow high-level programming with a significantly reduced compilation time, these advantages are not available for free. They are achieved at the cost of a lower performance and the usage of more hardware resource. Even the flexibility is sacrificed as each overlay is specific to a set of applications. As such, research has focused on reducing the area overhead and improving the throughput. The most significant factor which affects the area overhead of an overlay is the virtual routing resources [18, 19], or interconnects between the different FUs. A number of interconnect strategies exist, with the most common being: island style [7, 15, 18, 20, 21], nearest neighbor (NN) [6, 22], network-on-chip (NoC) [23, 24] and to a lesser extent linear interconnect [16, 25]. Most (island style, NN, and NoC) are 2-D mesh structures which are quite similar to the architecture of FPGAs. However, these overlays require numerous multiplexers as routing resource for the FUs to communicate with each other. In contrast, a linear or feed-forward interconnect structure which only supports directed acyclic data flow graphs has a much simpler and more hardware efficient implementation. However this trades off implementation efficiency against generality.

The work in this thesis attempts to target the area overhead of existing TM overlays, by developing an area efficient overlay based on time-multiplexed FUs with linear interconnect, which significantly reduces the hardware requirement while adding just a modest performance penalty. We show how this overlay can be optimized with various architectural enhancements to achieve higher throughput without the proportional increment in resource consumption, thus improving the compute efficiency. To demonstrate the suitability of the overlay as an FPGA accelerator, the proposed overlays are integrated into complete hardware accelerator systems and quantitative results of throughput and area are presented. The proposed overlay based accelerator is then compared with a state-of-the-art TM overlay, VectorBlox MXP, in terms of throughput and resource usage.

1.2 Objectives

The main research goal of this thesis is to improve the FPGA design productivity by developing area and performance efficient overlay accelerator systems which can support rapid compilation and just-in-time context switching. We aim to achieve this goal by setting the objectives as follows:

- Categorize the existing TM overlays and evaluate their computational throughput and the relative FPGA resource consumed along with advantages and disadvantages.

- Design an area-efficient FPGA overlay with a linear connection of highly pipelined time-multiplexed FUs based on the Xilinx DSP48E1 macro.
- Explore improvements to the compute efficiency of the proposed overlay with a number of architectural enhancements and a new mapping technique.
- Develop a mapping tool to compile the compute intensive kernels from high level language onto the proposed overlays, thus raising the hardware design to a higher level of abstraction.
- Integrate the overlay subsystems into complete hardware accelerator systems based on two different memory interface, AXI and PCIe.

1.3 Contributions

This thesis proposes a number of TM overlays with linear interconnect, which target streaming-based data processing. A number of accelerator systems have been developed by integrating the overlays with both AXI and PCIe-based memory interfaces. The main contributions of this thesis are as follows:

- Present a comprehensive literature review of the TM CGRA-like medium-grained overlays. The study shows that most of the existing TM overlays suffer from relatively large area overheads, due to either their underlying processor-like architecture or, for CGRA-like overlays, due to the routing resources and instruction storage requirements. Reducing the area overhead for CGRA-like overlays, specifically for the routing network, and utilizing the fast context switch capabilities of these overlays are likely to result in better usability with corresponding improvements in design productivity.
- Design and implement an area efficient FPGA overlay which uses a linear connection of time-multiplexed FUs based on the Xilinx DSP48E1 macro. This architecture enables a significant reduction in the resource overheads compared to other TM overlays due to a reduction in the instruction storage and interconnect resource requirements, assisted by

a fully-pipelined, architecture-aware FU design. While the proposed overlay delivers a throughput which is 6× to 22× less than the SC DSP-based overlay [7] and Vivado [8] implementations due to the much larger II, it consumes 85% fewer equivalent slices (eSlices) than the DSP-based overlay and achieves 740× faster context switching compared to the Vivado implementations.

- Propose a number of architectural enhancements such as using a rotating register file (RRF) in the FU, replicating the stream datapath and adding write-back support to the FU. The architectural enhancements help reduce the II significantly with just a modest increase in the area overhead, thus improving the compute efficiency. Compared to the original version, the modified overlays can achieve up to 2.4× higher throughput in GOPS, 93.7% higher compute efficiency in MOPS/eSlice and a 43.7% lower latency in ns.

- Integrate the proposed overlays into complete hardware accelerator systems, along with the AXI or PCIe memory interface. The AXI-Xillybus-V3 represents a very area efficient implementation compared with VectorBlox MXP-V16 [26], at the expense of around half of the throughput (limited by AXI-Xillybus using a 32-bit bus compared to the 64-bit bus used by VectorBlox MXP). PCIe-Xillybus-V3 and RIFFA-V3 are proposed for more high-performance accelerator systems. RIFFA-V3 achieves a throughput of 3.6× higher than the PCIe-Xillybus-V3, and a throughput of 5.7× higher than AXI-Xillybus-V3, at the cost of a larger BRAM consumption.

1.4 Organization

The thesis is organized as follows. Chapter 2 presents some FPGA background and the traditional FPGA design flow, followed by an introduction to two of the emerging approaches to improve the FPGA design flow, i.e. high-level synthesis and overlay architectures. A comprehensive literature review of the existing TM overlays is presented which categorizes TM overlays into two groups, processor-based TM overlays and CGRA-like TM overlays. Chapter 3 proposes an area efficient FPGA overlay with linear interconnect. A basic automated mapping tool is developed, followed by a comparison of the area, throughput and context switch time of our proposed overlay with HLS implementations and existing overlays from the research literature. Chapter 4 examines the limitations of the proposed linear TM overlay and presents a number of architectural enhancements to improve the overall performance in terms of throughput, latency, and compute efficiency. Chapter 5 introduces a complete hardware accelerator system based on the improved linear TM overlay using two different memory interfaces, Xillybus and RIFFA. The performance of these hardware accelerators for a range of benchmarks is investigated and performance results are presented. Comparisons are also made to a state-of-the-art TM overlay, namely VectorBlox MXP. Finally, Chapter 6 draws conclusions for the thesis and explores potential directions for future work.

Chapter 2

Background

2.1 Introduction

Field-programmable gate arrays (FPGAs) are typically comprised of an array of programmable blocks, which include lookup table (LUT) based logic for implementing general logic functions as well programmable memories, multipliers, etc., all connected by a programmable routing network [27]. In current commercial FPGAs, the logic blocks are typically 6-input LUTs and the routing network is an island-style architecture which consists of channel wires, connection blocks and switch blocks.

The traditional FPGA design process (as shown in Figure 2.1), is typically based on a register-transfer level (RTL) description of the circuit, using hardware description languages (HDLs) such as Verilog and VHDL. The design entry phase may involve the use of intellectual property (IP) cores provided by the FPGA vendor or other design groups, thus enabling the designer to meet the design requirements with less effort. However, RTL design requires a good working knowledge of the target technology's capabilities and of basic hardware concepts such as pipelining and synchronization to achieve a close to optimal implementation on the FPGA device. The FPGA design flow (in Figure 2.1) requires a significant amount of simulation and validation at all stages in the process, which also includes synthesis, placement and routing, and bitstream generation. A problem identified in any of the simulation steps requires that the designer revert back to the design entry phase for correction, with changes to just a few lines of HDL code requiring a rerun of the whole process to generate a new bitstream. Additionally, the design cycles are relatively slow due to the fine granularity of the FPGA resources, which can result in the ”Place & Route” process requiring hours (or even days) for a very large application. This design process greatly slows down the development progress of FPGA designs and to some extent, even hinders the widespread adoption of FPGAs. Many solutions are emerging to improve the FPGA design flow, thus alleviating some of the design cycle time concerns. These ongoing developments are generally focusing on two approaches: high-level synthesis and overlay architectures.

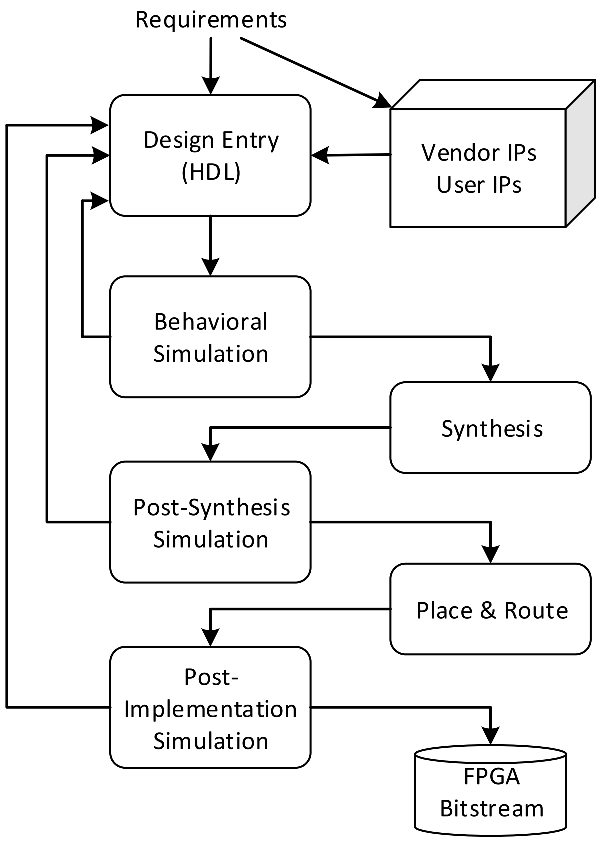


Figure 2.1: Traditional FPGA design flow.

2.2 High-level Synthesis

The evolution of high-level synthesis (HLS) traces back to the late 1970s, and it has gone through three generations from research to industry adoption during this time. The first generation (1980s to early 1990s) and second generation (mid-1990s to early 2000s) HLS tools had limited commercial success, mainly due to the choice of input languages and the inadequate quality of results [28]. However, the third generation HLS tools (early 2000s to now) have had better success, achieving higher levels of industry adoption, as they are able to model behavior at a higher level, while better targeting both dataflow and control domains.

The latest HLS techniques have been developed to translate a high level description of an application into custom logic, hiding much of the low-level FPGA detail. Typical HLS tools, such as Xilinx Vivado HLS [8], Altera SDK for OpenCL [9], and UoT LegUp [10] have been developed to interpret the high level language description of a user application and convert it into low-level RTL. Using HLS tools, there is less of a requirement for the designer to have a detailed hardware specialization as custom digital logic circuits can be generated automatically with high performance.

However, while HLS techniques alleviate the design productivity problem to some extent, it does not change the back-end flow (specifically the placement and routing process), which as mentioned before can require very long compilation times, contributing to long design cycles and the lack of mainstream adoption of FPGAs by software designers who are used to rapid design iterations [29]. Because of these long design cycles, researchers continue to examine other techniques for improving design productivity.

2.3 Overlay Architectures

Another way to improve productivity is to exploit the coarser granularity of modern digital circuits. The heterogeneous architecture of modern FPGAs provides opportunities to make use of the coarse-grained modules such as DSP blocks and block RAMs. One of these techniques is to use a virtual hardware representation which overlays the original FPGA fabric, referred to as an overlay architecture (or overlay).

An overlay is defined as a virtual configurable architecture, implemented over the physical fine-grained FPGA fabric, thus enabling programmability at a higher level of abstraction [11]. Overlay architectures promise to enhance the programmability of FPGAs by raising the design abstraction layer and avoiding the tedious fine-grained placement and routing process. Programming an overlay is similar to configuring an FPGA, except that configuration is also performed at a higher level, typically at the word and functional block level, rather than at the bit level. As such, the mapping tools for overlays can quickly generate an application bitstream in just a few seconds and configure the overlay in just a few microseconds, significantly faster than for fine-grained FPGA. Figure 2.2 shows a typical automatic mapping tool flow targeting an overlay. The overlay is firstly designed using the FPGA vendors design tools, and a bitstream for configuring the FPGA is generated, as shown in the RHS dashed box of Figure 2.2. The remainder of the tool chain generates an overlay configuration based on a user application. As the overlay is located at a layer between the user application and the underlying physical FPGAs, it is not necessary to regenerate the FPGA bitstream for different target applications. If an application changes, all that a designer needs to do is to regenerate the new configuration for the overlay using the mapping tool flow (shown on the LHS of Figure 2.2) and reprogram the overlay. This flow (which is more like a software programming flow) achieves thousands of times reduction in the design cycle time compared to a traditional FPGA CAD flow [16].

While overlays allow high-level programmability with a significantly reduced compilation time, these advantages are not available for free. They generally come at the cost of a lower performance with significantly more FPGA resource used than for an equivalent design mapped directly to FPGA. Even flexibility can be sacrificed as many overlays are specific to a set of applications [12, 13]. As such, a significant research effort has been applied to reducing the overlay area overhead and improving the throughput.

Overlays can be broadly classified based on the run-time configurability of their FUs. If an FU has a single fixed functionality at run-time, the overlay is referred to as spatially configured (SC), while if the FU changes its operation on a cycle-by-cycle basis, the overlay is referred to as time-multiplexed (TM). Table 2.1 lists some overlays categorized in terms of FU and interconnect configuration. From Table 2.1, it can be seen that overlays with SC FUs and SC interconnect networks [6,7,15,18–21,25,30] comprise a significant group. In an SC overlay, a single operation node is mapped to an individual FU and data is shifted between FUs over a programmable, but temporally dedicated, point-to-point link. That is, the FU and interconnect configuration are fixed while the kernel executes. The benefit of an SC overlay is that kernel execution achieves an initiation interval (II) [14] of one, with throughput just determined by the operating frequency of

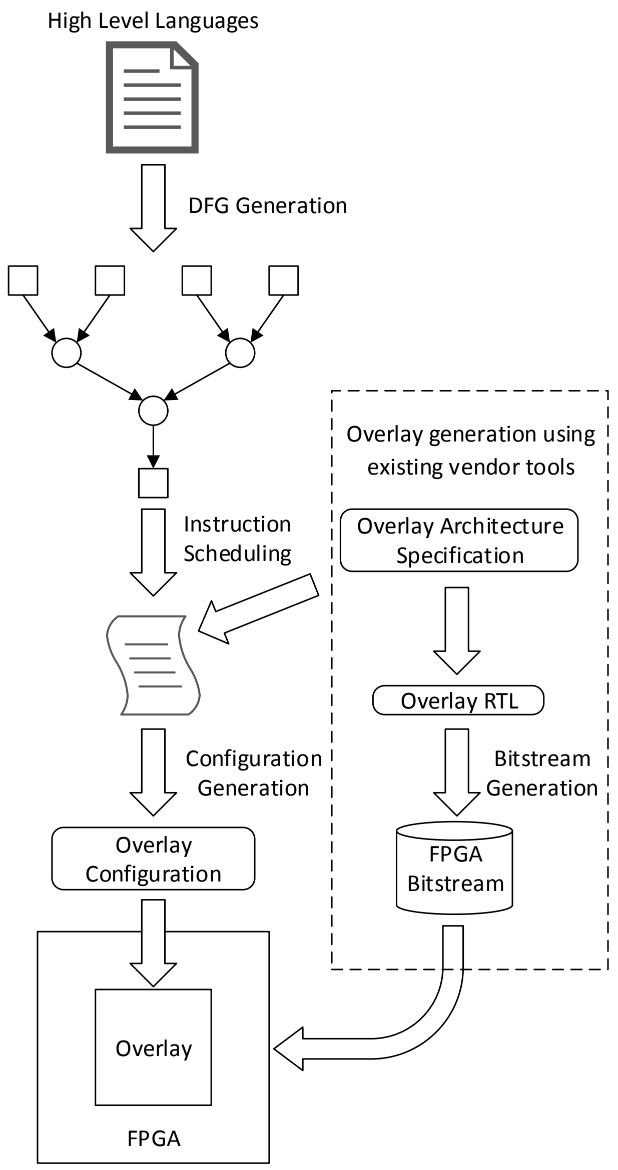


Figure 2.2: A typical overlay tool flow.

the overlay.

However, the area overheads of SC overlays, in particular their large interconnect resource requirements, have limited the practical use of these overlays in FPGA-based systems to very small compute kernels [15]. This means that a number of different kernels would need to be reconfigured to the overlay for a large application to achieve the best application acceleration. Thus the overlay context switch time (the time required to switch between executing kernels) is also an important consideration in the efficient operation of an overlay [16, 39]. Some of the current overlays utilize partial reconfiguration to reduce the overlay area, in particular the

Table 2.1: Selected overlay architectures.

Year	Overlay	FU		Interconnect		
	Name	SC	TM	SC	TM	NoC
2005	SPREE [31]		✓			
2006	QUKU [30]	✓		✓		
2010	IF [18]	✓		✓		
2011	VDR [25]	✓		✓		
2011	Heracles [32]		✓			✓
2012	ZUMA [33]	✓		✓		
2012	Octavo [34]		✓			
2012	reMORPH [17]		✓		✓	
2013	VCGRA [19]	✓		✓		
2013	CARBON [35]		✓		✓	
2013	MXP [26]		✓			
2013	SCGRA [36]		✓		✓	
2013	TILT [37]		✓		✓	
2015	DSP-based [7]	✓		✓		
2016	DeCO [38]	✓		✓		
2016	GRVI Phalanx [23]		✓			✓

interconnect resources, by trading off runtime connection flexibility [17]. However, while faster than a complete FPGA reconfiguration, partial reconfiguration still results in a significant context switch overhead, which will impact an application's runtime if multiple kernels are used.

As there is always a trade-off between area and speed in hardware design, a number of research groups have shifted their attention to overlays which share the functional units among kernel operations in an attempt to reduce overlay resource requirements. Sharing or time-multiplexing the FU can significantly reduce the FU and interconnect resource requirements but at the cost of a higher II and hence a reduced throughput. TM overlays can be generally divided into two categories: processor based overlays, and coarse-grained reconfigurable architecture (CGRA) like overlays. They are comprised of an array of interconnected FUs, which execute multiple operations on a cycle-by-cycle basis, similar to conventional processor cores [40]. Although the development of TM overlays is still at the primary stage, some of the existing works have shown great potential in tuning the compute density (throughput per area) and achieving rapid hardware context switching compared to the SC alternatives. In the next section, we review the current state-of-the-art relating to TM overlays.

2.4 Processor Based TM Overlays

Most successful TM FPGA overlays are based on processor implementations. These implementations range from simple single-issue processors, through multithreaded processors, to parallel processors and processor arrays. Overlays based on a processor implementation have the advantage of a well-known, well-designed instruction set architecture (ISA) which makes them easy to use, however, they tend to utilize a large amount of FPGA resource with a significant power consumption.

Table 2.2: Soft processors (32-bit).

Year	Name	Device	F_{max}	Area
2005	CUSTARD [41]	Virtex-2	30 MHz	2400 Slices
2005	UT Nios [42]	Stratix	77 MHz	3000 LEs
2005	SPREE [31]	Stratix II	82 MHz	1200 LEs
2007	Leon3 [43]	Virtex-2	125 MHz	3500 LUTs
2010	MB-LITE [44]	Virtex-5	65 MHz	1450 LUTs
2010	Leon4 [45]	RT4G150	150 MHz	4000 LUTs
2012	iDEA [46]	Virtex-6	453 MHz	335 LUTs
2012	Octavo [34]	Stratix IV	550 MHz	900 ALUTs
2016	GRVI [23]	UltraScale	375 MHz	320 LUTs

2.4.1 Soft Processors

A soft processor generally refers to a processor architecture which can be implemented on FPGA, which then allows the ISA to be customized to suit a specific application. FPGA vendors provide commercial soft processors such as Xilinx MicroBlaze [47] and Altera Nios II [48], implementing a conventional MIPS-like architecture for software portability. These industrial soft processors allow non-hardware experts to better target FPGAs with dedicated tools such as Xilinx EDK and Altera Eclipse. However, these implementations are not portable between different FPGA vendor devices and their RTL source code is not freely available. To overcome this, open source clones of these commercial soft processors have been developed, such as the performance centric UT Nios [42] and the area-efficient MB-LITE [44]. While these implementations are open source and can be customized to a specific application, their ISAs are not. To address this issue, a number of open source soft processors with free ISAs such as OpenSPARC [49], OpenRISC [50], Plasma [51], RISC-V [52], Leon3 [43] and Leon4 [45], were developed by industrial or independent groups. A recent survey of open source soft processors [53] showed that apart from Leon3, most had a larger area overhead and provided less performance compared to MicroBlaze and Nios II. Table 2.2 lists the latest versions of some typical soft processors in the last decade.

2.4.1.1 Single-issue Processors

Many of the earlier soft core processors were single-issue processors because of their simplicity and area efficiency. These processors were to some extent constrained by the limited resources available in earlier generations of FPGA devices. MicroBlaze [47], Nios II [48], OpenRISC [50] and Plasma [51] are all examples of single-issue processors. Single-issue processors also tend to have fewer pipeline stages than multi-issue (superscalar) processors [54]. Some other single-issue processors include:

SPREE: The Soft Processor Rapid Exploration Environment (SPREE), was developed to automatically generate synthesizable HDL implementations of soft processor architectures from textual descriptions of the ISA and datapath [31], facilitating the microarchitectural exploration of soft processors. The SPREE processor with a 3-stage pipeline demonstrates 9% less area and 11% speedup in wall-clock-time compared to the Nios II family of commercial soft processors. By customizing the microarchitecture to specific software applications, the tuned version of SPREE provides an average improvement of 11.4% over the fastest-on-average general purpose processor in terms of compute efficiency [55]. The complexity of SPREE can be reduced by using functional component abstractions, however, some practical issues such as combinational loops, false paths, and multi-cycle paths, which affect the functionality and performance of the soft processor, may arise due to the careless use of these components.

iDEA: iDEA [46,56] is a lightweight soft processor based on the Xilinx DSP48E1 primitive and was developed to address the resource consumption issue while better targeting the underlying

FPGA architecture. The 9-stage pipelined design with no data forwarding outperforms MicroBlaze in both resource consumption (a 59% reduction in LUTs with an 18% increase in FFs) and speed (a 92% increase in f_{max}). To reduce the execution time caused by NOP insertion due to data hazards, data forwarding approaches applicable to the DSP48E1 primitive, such as internal loopback and external forwarding, were explored, resulting in an improvement of up to 25% for a set of benchmarks [57].

While iDEA was designed as a soft processor to handle integer operations, it cannot fully support 32-bit multiplication because of the limited width of the multiplier inputs in the DSP48E1 (25×18 bits). Only a single DSP block is used to implement the soft processor, however, as there are hundreds of DSP blocks available in the modern FPGAs, making better use of these resources within a multi-processor system would significantly improve the performance for large compute kernels.

2.4.1.2 Multi-issue Processors

While most of the early generation of soft processors were single-issue cores, multi-issue or superscalar single processor implementations have also been developed. One of the best examples is the LEON3 processor [45], based on the 32-bit SPARC V8 processor architecture, which was developed for space applications and is available as a soft core for FPGAs. Another example is the Intel Nehalem soft processor core [58] which was developed for emulation purposes and uses five FPGAs while running at a frequency of just 520 kHz. Unfortunately, a superscalar architecture requires significant hardware complexity to dynamically extract the instruction parallelism which when implemented in FPGA results in very high hardware costs.

2.4.1.3 Multithreaded Processors

While single-issue processors are expected to run at a higher frequency with a pipelined architecture, their area-efficiency and instruction-per-cycle (IPC) count can be improved significantly with minimal extra complexity to support multithreading [59]. UTMT II [60] and MT-MB [61] are two typical soft processors which support multithreading on the Altera Nios II/e and Xilinx MicroBlaze core respectively. UTMT II achieved a 25% LE area reduction compared with Nios

II/e, while MT-MB achieved a peak performance of $5\times$ over that of MicroBlaze. Apart from the extension of commercial cores, there are a number of independent research efforts towards providing multithreading support on soft processors, such as CUSTARD [41] and Octavo [34].

CUSTARD: The Customizable Multithreaded Processor (CUSTARD), was one of the first customizable multithreaded soft processors, supporting a parameterizable number of threads, threading type, datapath bitwidths and custom instructions [41, 62]. CUSTARD is a RISC processor which has a fully bypassed architecture with a 4-stage pipeline. When implemented on an XC2V2000 FPGA and compared with MicroBlaze using five typical benchmarks, the CUSTARD processor achieved an average speedup of $2.41\times$ across all benchmarks with custom instructions. However, CUSTARD, and its extended version [62], only achieved a clock frequency of 30 MHz to 50 MHz, which is far less than the 100 MHz achieved by the MicroBlaze soft processor. Additionally, the custom instruction speedup came at a penalty of two times the area consumption and less I/O support compared to MicroBlaze.

Octavo: The Octavo soft processor [34] is a multithreaded 10-stage pipelined architecture designed to operate at the theoretical maximum BRAM frequency (550 MHz) on a Stratix IV device. A method of self-loop characterization was adopted to collapse the conventional register/cache/memory hierarchy into one unified entity, which is beneficial to absorb the propagation delays and simplify the ISA. To support fast multiplication, a fast multiplier which consists of two half-pumped DSP blocks was designed to overcome the hardware timing restriction of 480 MHz.

In summary, although single-core soft processors allow the benefits of software programmability and hardware re-usage, their performance is still significantly less than that of either hard processors or dedicated hardware accelerators, and cannot meet the requirements of very high speed applications. In order to improve the throughput, there is an increasing amount of research work exploring multi-core systems of soft processors with efficient routing technologies.

2.4.2 Parallel Processors

The sequential processing of single-issue soft processors has limited their use to specific lower performance applications. When large scale applications are considered, parallel computing, using single instruction, multiple data (SIMD) execution or other parallel processing techniques, may be required.

2.4.2.1 Multithreaded Parallel Processors

The Octavo soft processor [34] was further extended to support SIMD by duplicating the datapath with a shared instruction stream [63]. SIMD-Octavo was compared with VectorBlox MXP [26] (discussed in Subsection 2.4.2.3) and operates at about double the clock frequency of MXP and generally achieves better performance (for an equal number of lanes) in terms of execution time, area, and area-delay product. It has been claimed that the execution time of multi-lane SIMD-Octavo is better than hand-crafted Verilog HDL, but requires one to two orders of magnitude more hardware resource [63].

2.4.2.2 VLIW Processors

Very long instruction word (VLIW) processors have been proposed to exploit instruction level parallelism (ILP) by executing different operations on multiple FUs simultaneously [64].

TILT: The 32-bit floating point TILT overlay [12, 65], was proposed as an FPGA-based VLIW processor comprised of multiple floating point FUs with configurable pipeline depths. To enhance the throughput, multiple TILT cores can be instantiated, working in parallel with a single shared instruction memory. This architecture is referred to as TILT-SIMD. TILT has a separate 256-bit memory fetcher unit which allows for data transfer between up to 8 TILT cores and the off-chip DDR memory. The TILT overlay was evaluated for a set of five application benchmarks against Altera OpenCL HLS implementations. The TILT overlay was able to achieve an operating frequency over 200 MHz, which is close to that of the HLS implementations, with an area overhead of less than $2\times$ for the same throughput.

Currently, the TILT-System is not customized to a general class of kernel applications, and as such, a kernel update for a different application requires instruction rescheduling, with an associated FPGA reconfiguration, resulting in a context switch time of 38 seconds on average. Another drawback of the TILT overlay is that, even though TILT is more flexible than OpenCL HLS for implementing very small designs, it has less compute density compared to the OpenCL implementation. This problem can be solved by customizing the number of FUs and their functionality for specific applications.

2.4.2.3 Vector Processors

While it remains a problem for soft processors to scale their performance, soft vector processors (SVPs) are able to exploit data-level parallelism. They are able to explore the trade off between performance and area, with a hybrid approach which shares the benefits of traditional vector processing and modern SIMD mode. Most of the proposed SVPs have a similar architecture, with a scalar soft processor acting as the controller for multiple vector lanes executing custom instructions on a local memory [66]. SVPs can achieve a significant speedup over soft processors by effectively unrolling loops into vector operations. However, there are a number of obstacles limiting the widespread adoption of SVPs. These include, the centralized vector register file complexity, the difficulty in implementing precise exceptions for vector instructions, and the on-chip vector memory system cost [67].

A number of SVP designs, including VESPA [68], VIPERS [69], VEGAS [70], VENICE [71], and MXP [26], have been proposed. VESPA and VIPERS were developed in parallel as the first generation of FPGA-centric SVPs, with VEGAS, which better utilizes the on-chip FPGA memory, being the second generation. VENICE is the latest version, targeting high frequency and low area, and led to the first commercial SVP, referred to as VectorBlox MXP.

VESPA: VESPA was proposed as a MIPS-based processor with a VIRAM [72] compatible vector coprocessor, which results in a system combining the advantages of portability, scalability, and flexibility [68]. VESPA is portable across FPGA platforms, though the original design targeted Stratix III. The VESPA prototype achieved an average speedup from $1.8\times$ (2-lane) to $6.3\times$ (16-lane) over the scalar processor on EEMBC benchmarks [73]. The flexibility of VESPA makes it possible to trade off area savings (up to 70%) by adjusting the vector lane length and width. To better target the FPGA, an improved VESPA with support for vector chaining and heterogeneous lanes [74] was implemented on a Stratix III FPGA. The modified VESPA achieved up to 34% better compute efficiency relative to VESPA in terms of performance-per-area for the full set of EEMBC benchmarks.

VIPERS: Similar to VESPA, VIPERS consists of a single-threaded (Nios II-compatible) scalar core referred to as UTIIe, a memory interface unit, and a vector processing unit [69]. Three typical data-intensive applications were used as benchmarks to demonstrate the capabilities of VIPERS. The same three applications were also implemented as hardware accelerators for the Altera Nios II/s processor using the C-to-Hardware (C2H) Compiler [75]. Compared to Nios II, VIPERS demonstrated a scalable speedup ranging from $3\times$ to $29\times$, at the cost of a $6\times$ to $30\times$ area penalty. An improved version of VIPERS [76] offers double the vector registers and several new instructions (compared to VESPA), and is less strict about VIRAM compliance. Based on the benchmarks in [69], VIPERS with 16 lanes can achieve up to $25\times$ better performance with a modest $14\times$ area increase compared to the base Nios II processor. It is possible to achieve a further 30% area savings by customizing VIPERS to the benchmarks, equal to $6\times$ the logic area of the Nios II/s processor implementation.

Although both VESPA and VIPERS provide a wide range of granularity from 8-bit to 32-bit, the vector engine must be built to fit the largest width if mixed-width data processing is required. As a result, byte-sized data needs to be zero-extended or sign-extended to the full width, which unnecessarily adds overhead to the instruction memory and register files. Additionally, as the vector register file is connected to an on-chip memory (VIPERS) or on-chip data cache (VESPA), the memory/cache width must be large enough to support the traditional vector load/store operations. However, the amount of on-chip memory is limited by the capacity of a particular FPGA.

VEGAS: Though VESPA and VIPERS demonstrated the scalability and feasibility of SVPs, they were not specifically targeted to the underlying FPGA architecture. As such, a new SVP architecture, VEGAS, was presented as a vector core with a Nios II/f processor [70]. The most significant differences between VEGAS and the previous SVPs, is the use of a cacheless scratchpad memory and a fracturable ALU which can support byte, halfword or word operations efficiently, according to the data width. Instead of conventional vector load/store instructions, VEGAS adopted direct memory access (DMA) read/write commands to achieve better storage efficiency and less memory latency. VEGAS can achieve up to $2.8\times$ better performance than VESPA and $3.1\times$ better than VIPERS in terms of throughput-per-area, and outperforms a 2.66GHz Intel X5355 processor on the integer matrix multiply benchmark.

Despite the high performance VEGAS achieves, there are some drawbacks to the design which result in an area/ performance overhead. Firstly, it is cumbersome to track and spill values from the 8-entry vector address register file (VARF), which also consumes additional ALMs and FFs. Secondly, while the alignment network grows super-linearly with the number of vector lanes, only one single alignment network is implemented on VEGAS, which may introduce a performance penalty if the operands are unaligned.

VENICE: Based on the architecture of VEGAS, VENICE was proposed to maximize the throughput of SVPs with a small number of vector lanes [71]. While VEGAS achieved its best performance/area at 4-8 lanes, VENICE was tailored to 1-4 lanes without sacrificing performance. Removal of the vector address register file, adding a new conditional implementation, and streamlining the instructions, are the three major differences which reduce the area requirement and the complexity of programming, compared to VEGAS. 2D/3D vector instructions and operations on unaligned vectors were adopted to further improve the performance. VENICE can achieve over $2\times$ better throughput-per-area than VEGAS, and a speedup of $5.2\times$ higher than the fastest Nios II/f soft processor.

VENICE is much more area-efficient and easier to program compared with previous SVPs and further improves on the VEGAS ALU utilization. Since VENICE is designed as a small and fast SVP, the problem of efficiently integrating multiple VENICE components with high performance and interconnect simplicity remains a future problem.

MXP: The VectorBlox MXP was developed as a commercial IP core which can be connected with Altera or Xilinx FPGAs via Avalon or AXI interfaces [26]. It is similar in design to VENICE, but with added features such as fixed-point arithmetic, 2D-DMA support, and a C++ object based application programming interface (API) for higher level programming. MXP can operate at over 200 MHz on a Stratix IV device with less than 16 vector lanes. A 64-lane configuration demonstrated a speedup of up to $918\times$ that of a Nios II/f processor on matrix multiplication. Custom vector instructions (CVIs) were introduced for the latest SVPs to integrate streaming pipelines into the datapath with a minimum area overhead [77]. CVI-optimized SVPs achieved a $7200\times$ speedup and over $100\times$ improvement in terms of performance-per-ALM, compared to Nios II/f.

In general, SVPs achieve significant performance gains for data parallel applications. However, the scalability of SVPs is limited by the number of vector lanes, which is determined by the hardware resources on the FPGA. While increasing the number of vector lanes significantly increases the throughput, it also leads to clock frequency degradation. Additionally, compiler support for these processors is still at the primary stage as the repository of common operations and data types needs to be further improved.

2.4.2.4 Soft GPUs

Graphics processing units (GPUs) have a many-core architecture with considerable parallel processing capabilities. In general, GPUs and vector processors have many similarities with both supporting SIMD-style parallelism.

FlexGrip: FlexGrip [78] is a soft GPU based on the Nvidia G80 architecture targeting the Xilinx ML605 platform and provides direct CUDA compilation and execution. FlexGrip follows a single instruction multiple thread (SIMT) model with an instruction fetched and simultaneously mapped onto multiple scalar processors (SPs). FlexGrip with 32 SPs achieves a peak speedup of 30$\times$ compared to MicroBlaze, but with a significant area overhead, consuming 96% of the available LUTs.

MIAOW: MIAOW [79] is an open source RTL implementation of the AMD Southern Islands GPU ISA, which is compatible with OpenCL applications. The complete system was implemented on a Xilinx VC707 evaluation board requiring a considerable amount of FPGA resource (195K LUTs and 137 BRAMs). MIAOW was validated by comparing it with commercial GPUs in terms of area, power, and performance.

FGPU: A GPU-like SIMT soft processor, referred to as FGPU [80], was proposed as a flexible solution for software tasks. The VHDL implementation of FGPU did not use any FPGA specific IP cores or FPGA primitives, making it highly portable and customizable. It has a mixed ISA supporting both MIPS instructions and OpenCL functions. A speedup of 48.5$\times$ over MicroBlaze was achieved for a range of benchmarks on the Xilinx ZC706 FPGA board, with a 17.7$\times$ area overhead. To achieve high performance, FGPU is designed with an 18 stage pipeline. Due to the complexity of the compute units, an 8 compute unit version of FGPU consumes 124K LUTs on the ZC706, corresponding to 57% of the available resource.

SCRATCH: An application-aware soft GPU, referred to as the SCRATCH framework [81], was developed as an upgraded version of the MIAOW GPU architecture. The main contribution of the SCRATCH system is the MIAOW-based architecture optimization to support additional instructions and the SCRATCH trimming algorithm which removed unnecessary architectural functionality to improve performance. Similar to MIAOW, SCRATCH was evaluated on Xilinx Virtex 7 FPGAs. By applying architecture trimming along with multithread and multi-core parallelism, SCRATCH was able to achieve a peak speedup of 260× with a 250× better energy-efficiency compared to the original MIAOW system. In addition to the improvement in throughput and energy-efficiency, a significant reduction in FPGA resource was observed, specifically a 36% reduction in LUTs and a 41% reduction in FFs.

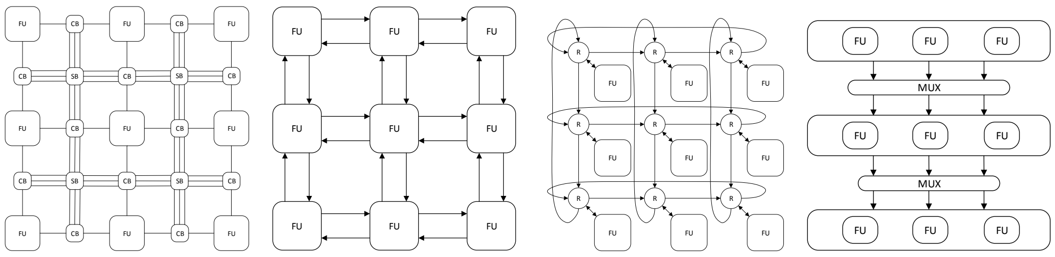
2.5 CGRA-like TM Overlays

Coarse-grained reconfigurable architectures (CGRAs) have been extensively researched due to their enhanced scalability, performance and power efficiency compared to CPUs. CGRAs typically fall within one of two classes: processor-centric arrays which are made up of individual processors connected via programmable interconnect; and CGRAs with coarse/medium-grained processing elements (also called medium-grained processing arrays).

2.5.1 Interconnect topology

Irrespective of the computational element (be it a processor or a dedicated processing element), CGRA-like overlays are characterized by an array structure of computational elements connected using programmable interconnect. A number of interconnect strategies exist, with the most common being: island style [7, 15, 18, 20, 21], nearest neighbor (NN) [6, 22], network-on-chip (NoC) [23, 24] and to a lesser extent linear interconnect [16, 25], as shown in Figure 2.3. Other interconnect strategies are possible, including circuit switched [82] networks, but these typically consume significant hardware resource and are less suited for FPGA-based overlays. There are also variations in the more common interconnect strategies. For example, for NN, alternative typologies include torus [22], mesh plus [83] and fully connected [84], while for NoC, many different typologies exist [24].

Island style and NN interconnects are a 2-D mesh structures which to some extent have a similar architecture to the interconnect on FPGAs. However, these interconnect strategies require a considerable amount of the FPGA routing to implement and as a result consume a significant amount of the FPGA resource [85]. In contrast, the resource requirement for a linear interconnect is significantly less because of its feed-forward array structure, which comes at the expense of reduced flexibility.



(a) Island style (b) Nearest neighbor (c) NoC torus (d) Linear

Figure 2.3: Typical overlay topologies.

2.5.2 CGRA-like Processor Arrays

Large CGRA-like processor arrays have seen a resurgence in recent years due to the higher capacity of modern FPGAs. This larger FPGA capacity, along with more efficient NoC implementations [24] has meant that they are able to accommodate more complex designs. These processor arrays have similarities to ASIC-based processor-centric CGRAs. Some examples include:

Heracles: Heracles [32] is an open-source integer-based 7-stage MIPS-III processor array with a 2D-mesh topology, which consists of a NoC architecture for data communication. Synthesis results showed that one processor element with cache memory consumed 5562 LUTs and 2695 FFs on a Virtex-5 LX330T, running at a frequency of 155 MHz. The Heracles virtual-channel router consumed 2058 LUTs, 2806FFs and operated at a frequency of 71 MHz. Compared to the classic unbalanced fat-tree [86] topology, the proposed virtual-channel router consumed only 1.7% of the fabric logic, with a 2.3× higher clock frequency. However, LUT consumption became the bottleneck when scaling due to the attached memory subsystem, thus Heracles was restricted to a 4×4 array on Virtex-5.

GRVI Phalanx: GRVI Phalanx [23] is a massively parallel overlay based on an FPGA-efficient implementation of the RISC-V [52] soft processor. The GRVI processor uses just 320 LUTs and runs at a frequency of up to 375MHz on a Kintex UltraScale FPGA. Multiple GRVI processors with shared memory and local interconnect, are formed as clusters, which efficiently communicate with each other via a Hoplite NoC [24]. Implementations with 400 and 1680 RISC-V cores on a Kintex UltraScale KU040 and a Virtex UltraScale+ VU9P have been reported. Currently there is minimum tool support for this platform with no application performance comparisons with other overlays.

120-core MIPS Overlay: A 120-core MIPS overlay [87] using on an optimized implementation of the μaptiv MIPSfpga [88], based on the silicon-tested microAptiv MIPS processor, was developed. The design achieved a significant FPGA resource reduction, requiring just 30% of the ALMs and 18% of the M20K RAM blocks compared to the original μaptiv MIPSfpga design [88]. This was achieved by replacing the complex instruction/data cache with dedicated scratchpads, adopting DSP blocks for multiplication and using a NoC-specific modification to the decoder. The improved MIPS processors with a Hoplite NoC [24] increased the maximum array size from 30 to 120 cores on a DE5-NET board, while achieving a higher frequency (94 MHz).

2.5.3 CGRA-like Medium-grained Overlays

CGRAs with medium-grained processing elements have a number of advantages compared to CPUs, including better scalability, performance and power efficiency [89]. Additionally, compared to fine-grained reconfigurable architectures, such as FPGAs, which typically consist of an array of logic blocks at the bit-level (or a small number of bits), CGRAs are reconfigurable at the word-level (8-bit, 16-bit, 32-bit, etc.). In CGRAs, the processing elements are typically much larger than the FPGA's fine-grained lookup tables (LUTs), and can be an arithmetic logic unit (ALU) or word-level multiplier, or even a DSP primitive. This coarse granularity results in a reduction in the configuration memory, the configuration time, and the placement and routing complexity, compared to fine-grained FPGAs [90]. Although there has been a significant amount of CGRA research over the last few decades, only a few CGRAs have been commercialized, mainly because they are less flexible compared to FPGAs and lack a well-defined design flow [30].

An alternative to an ASIC CGRA is the CGRA-like FPGA overlay, which implements a CGRA as a virtual configurable architecture on top of a reconfigurable FPGA. Initially, mapping CGRAs to FPGA was performed to demonstrate their functionality before ASIC implementation. More recently, specific dedicated CGRA-like FPGA overlays were developed mainly to improve the design productivity of FPGA. Many of these initial CGRA-like overlays were more throughput-oriented SC overlays which mapped each operation to a single FU to achieve an II of one. However, as mentioned earlier, these overlays were relatively small due to the limited hardware resources available in the underlying FPGA and were unable to accommodate larger compute kernels. Recently, researchers have shifted to more area-efficient overlay architectures which are able to time-multiplex the operations to an FU on a cycle-by-cycle basis. This makes it possible to map larger application kernels to the overlay, but at the cost of throughput. A summary of some of the TM CGRA-like overlays is given in Table 2.3.

Table 2.3: Selected CGRA-like overlays.

Year	Name	Granularity Arithmetic	Device	F_{max} Size	FPGA Resource
2010	Heracles [32]	32-bit Integer	Virtex 5	155 MHz 4×4	12K LUTs, 8.8K FFs
2011	MIN Overlay [91]	8/32/64-bit Integer & FP	Virtex 6	100 MHz 30	22K LUTs, 4.8K FFs, 40 DSPs
2011	CARBON [35]	32-bit Integer	Stratix III	150 MHz 2×2	3K ALMs, 517 FFs, 15Kb BRAM, 4 DSPs
2012	reMORPH [17]	32-bit Integer	Virtex 6	400 MHz[1] 40	196 LUTs, 41 FFs, 3 BRAMs, 1 DSP[2]
2013	SCGRA [36]	32-bit Integer	Zynq-7000	250 MHz 2×2	5K LUTs, 9K FFs, 50 BRAMs, 12 DSPs
2016	GRVI Phalanx [23]	32-bit Integer	UltraScale	375 MHz 10×5×8	177K LUTs, 1200 BRAMs
2017	MIPS Overlay [87]	32-bit Integer	Stratix V	94 MHz 60×2	2.4K ALMs, 2.1K FFs, 2 DSPs, 3 M20Ks[2]

[1] Reported F_{max} is only for an FU.
[2] Reported Resource is only for a single FU.

2.5.3.1 TM Overlays with Homogeneous FUs

Time-multiplexed CGRA-like overlays with Homogeneous FUs have the advantage that they can be more easily tiled to the FPGA architecture due to their regularity. Additionally, applications can be more easily scheduled as operations can be arbitrarily mapped to FUs. However, having only homogeneous FUs can restrict application flexibility. Some examples include:

CARBON: CARBON [35] is a CGRA-like overlay which was implemented as a 2×2 array of tiles on an Altera Stratix III FPGA. Each tile has an FU with a programmable ALU and instruction memory, supporting up to 256 instructions. An FU consumed 3K ALMs, 517 FFs, 15.6Kb BRAM and 4 DSP blocks, achieving an operating frequency of 150 MHz. Compared to the other TM overlays discussed here, CARBON has a large resource requirement with a relatively slow speed which limits the scalability of the architecture. Additionally, the BRAMs were not effectively used to read the instruction memory, which results in the need for an additional bypass register to avoid the extra latency.

reMORPH: The reMORPH overlay [17] better targeted the FPGA fabric, with an FU consuming 1 DSP Block, 3 block RAMs, 196 LUTs and 41 registers. This low footprint makes it possible to implement around 40 tiles on the Xilinx Spartan 6 LX45 FPGA. A reMORPH tile uses the Xilinx DSP primitive as a 5-stage pipelined ALU with a BRAM as its instruction memory, which ensures a high operating frequency (400MHz). To reduce the overhead due to routing and multiplexers, the reMORPH FU does not use decoders resulting in a 72-bit-wide instruction memory (supporting up to 512 instructions) which causes an over utilization of BRAMs, thereby limiting the possible size of this overlay. Tiles are interconnected using an NN style of non-programmable interconnect, which is adapted using partial reconfiguration at runtime, and hence changing between application kernels is relatively slow (that is the overlay has a large hardware context switch time).

SCGRA: The SCGRA overlay [36] was proposed to address FPGA design productivity, demonstrating a $10\times$ to $100\times$ reduction in compilation time compared to the AutoESL HLS tool. Application specific SCGRA overlays were subsequently implemented on the Xilinx Zynq platform [22], achieving a speedup of up to $9\times$ higher than the standalone Zynq ARM processor. The FU used in the Zynq-based SCGRA overlay operates at 250 MHz and consists of an ALU, multi-port data memory (256×32 bits) and a customizable depth instruction ROM (Supporting 72-bit wide instructions) which results in the excessive utilization of BRAMs. As the full FPGA bitstream needs to be reconfigured for a compute kernel change, very fast context switching between applications is not possible.

Although the SCGRA overlay allows for different size implementations, there is a significant performance drop for larger implementations due to the following reasons. Firstly, the higher BRAM requirement for instruction memory means that there needs to be a trade-off in the number of BRAMs for the I/O buffer, which has a negative effect on data reuse. Secondly, a larger SCGRA overlay will increase the routing cost between PEs, therefore reducing the compute performance. Finally, the operating frequency drops as the overlay size increases, resulting in a degradation of the overall performance.

2.5.3.2 TM Overlays with Heterogeneous FUs

Time-multiplexed CGRA-like overlays with Heterogeneous FUs have the advantage that they can support a wider range of applications, including mixed integer and floating point applications. An example is the MIN overlay:

MIN Overlay: The MIN overlay consists of heterogeneous FUs, which are connected by a global multi-stage interconnection network (MIN) [91]. The heterogeneous FUs can support up to 64-bit floating point computations. MIN uses a global interconnect network instead of the traditional 2-D array topology, which significantly reduces the routing resource requirements, resulting in better hardware resource utilization. Compared with the crossbar network in TILT, the proposed two parallel blocking MINs reduce the cost complexity from $O(n^2)$ to $O(n \log n)$. A number of different parameterized architectures, chosen to evaluate the impact of the number and types of FU, memory and I/O, were implemented on a Virtex 6 FPGA. The smallest architecture (called A1) has 30 FUs and a 64 I/O global network and consumes around 1% of registers, 4% of DSPs and 15% of LUTs, while running at a frequency of 100MHz. A heuristic scheduling, placement and routing algorithm was used and could achieve just-in-time compilation in less than 300ms. While MIN has relatively good FPGA resource utilization, the LUT usage due to the routing network in larger designs will eventually become the limiting factor.

Additionally, in some cases, routing fails due to missing registers which could be overcome by adding a register file in some of the FUs.

2.6 Summary

TM overlays for FPGA are reasonably mature with processor based TM overlays being better accepted compared to CGRA-like overlays. This is because the processor based overlays have the advantage of well understood ISAs and easily accessible compilation tool chains making application development much easier for non-hardware designers. Furthermore, processor based overlays using parallel processing techniques, such as multi-issue, multithreading, VLIW, and vector processing, have been developed and shown to improve overlay performance. However, these overlays suffer from similar problems to processors implemented directly in silicon, such as being complex with significant resource utilization and power consumption, which tends to negate some of the designer productivity advantages (such as their software programmability).

On the other hand, CGRA-like TM FPGA overlays have only really appeared within the last several years. These overlays are again targeted at improving FPGA designer productivity, and are better tailored towards area-efficient higher speed processing than processor-based overlays, although they still suffer from a lower speed and higher FPGA resource utilization than direct HDL- or HLS-based application implementation on FPGA. Recent CGRA-like overlays better utilize the coarse-grained modules present in modern FPGAs, such as DSP blocks and BRAMs. These overlays are particularly targeted towards the acceleration of small compute intensive loops [22].

A selection of the TM overlays from the literature (both processor-based and CGRA-like) are summarized in Table 2.4. Table 2.4 categorizes the different overlays based on the overlay type and provides an indication of the computational throughput and the relative FPGA resource consumed. So that the overlay's implementation technology does not overly impact the throughput and resource utilization, both these metrics have first been nominally normalized to that of a Virtex 7 implementation. The throughput is normalized by multiplying by the ratio of the maximum Virtex 7 BRAM frequency divided by the maximum BRAM frequency of the original target device, while the resource consumption is determined by considering the total system resource utilization (adjusted to account for technology changes, such as the transition from 4-LUTs to 6-LUTs) divided by the number of cores/FUs. The advantages and disadvantages of the different overlay types are also presented.

The review of the literature indicates that TM CGRA-like medium-grained overlays show potential benefits over other overlays. This is because time-multiplexing the overlay allows it to change its behavior, cycle by cycle, during the compute kernel execution, thus allowing better sharing of the limited FPGA resource. However, most of the TM overlays described previously suffer from relatively large area overheads, due to either their underlying processor-like architecture or, for CGRA-like overlays, due to the routing resources and instruction storage requirements. Reducing the area overhead for CGRA-like overlays, specifically for the routing network, and utilizing the fast context switch capabilities of these overlays are likely to result in better usability with corresponding improvements in design productivity. Additionally, there is scope for improving the FPGA resource utilization and overlay operating speed. These issues, along with the development of an overlay based coprocessor tightly coupled to a host processor for application acceleration will be examined in future chapters.

Table 2.4: A summary of selected TM overlays.

Name	Overlay Type	Throughput	Resource	Advantages	Disadvantages
CUSTARD [41]	Single-issue μp	Very low	Medium	Well understood processor ISA; good tool support; easy to use; area efficient; high flexibility when configuring the core and tuning to specific applications	Relatively low performance; relatively high power consumption; Some processor designs attempt to be general and do not make good use of a specific FPGA architecture
UT Nios [42]	Single-issue μp	Low	Medium		
SPREE [31]	Single-issue μp	Low	Low		
Leon3 [43]	Single-issue μp	Low	Medium		
MB-LITE [44]	Single-issue μp	Very low	Low		
Leon4 [45]	Single-issue μp	Medium	Medium		
iDEA [46]	Single-issue μp	Medium	Very low		
SIMD-Octavo [63]	MT Processor	High	Low	Well understood processor ISA; good/adequate tool support; high performance; supports parallelism; low power consumption	Resource hungry; higher code complexity; inefficient if the data cannot be executed in a highly parallel manner
TILT [12]	VLIW Processor	Medium	High		
MXP [26]	Vector Processor	High	Medium		
FGPU [80]	Soft GPU	High	Very high		
SCRATCH [81]	Soft GPU	Very high	Very high		
Heracles [32]	CGRA-like μp Array	Medium	Medium	Well understood processor ISA; high performance; high scalability; efficient NoC routing network	Limited tool support; resource hungry; high power consumption; lack of application benchmark evaluations
GRVI Phalanx [23]	CGRA-like μp Array	Very high	Very low		
MIPS Overlay [87]	CGRA-like μp Array	High	Medium		
MIN Overlay [91]	CGRA-like MG Overlay	Medium	Low	Moderate performance; low area consumption; low power consumption, makes good use of coarse-grained modules such as DSPs and BRAMs	Limited tool support; large routing area overhead; long context switching time; only suitable for acceleration of small kernels (except SCGRA); lack of a pipeline-aware scheduling strategy
CARBON [35]	CGRA-like MG Overlay	Low/Medium	Low		
reMORPH [17]	CGRA-like MG Overlay	Medium	Very low		
SCGRA [36]	CGRA-like MG Overlay	Medium	Low		

Chapter 3

Proposed Overlay

3.1 Introduction

As discussed in Chapter 2, a number of overlays have been proposed which share functional units among kernel operations in an attempt to reduce overlay resource requirements. This time-multiplexing (TM) of the overlay means that it can change its behavior on a cycle-by-cycle basis while the compute kernel is executing [12, 17, 35, 36], thus allowing sharing of the limited FPGA resources. However, in many other TM overlays, the storage requirements for instructions are very large, resulting in a significant area overhead. This is mainly due to: the scheduling strategy; the execution model; and the design of the overlay architecture, and limits overlay scalability while also impacting the kernel context switch time. Additionally, many of these overlays are not architecture-focused and hence the FU operates at a relatively slow frequency.

This chapter introduces our preliminary investigations into a resource efficient linear time-multiplexed overlay, acting as a processing pipeline to address the concerns identified above. An area efficient FPGA overlay architecture is developed that uses a linear connection of time-multiplexed FUs based on the Xilinx DSP48E1 macro. This architecture enables a significant reduction in the resource overheads compared to other TM overlays due to a reduction in the instruction storage and interconnect resource requirements, assisted by a fully-pipelined, architecture-aware FU design. The proposed 8-stage pipelined FU runs at a frequency of 325 MHz on a Xilinx Zynq device while consuming a very small resource footprint of 1 DSP block, 160 LUTs and 293 FFs. The FUs are then daisy-chained using a direct connection between FUs, thus minimizing the resource required by the connection network. The main contributions can be summarized as follows:

- An architecture using a linear connection of FUs to form a processing pipeline which is able to support the execution of feed-forward DFGs. This architecture results in a significant reduction in both the FU and interconnect compared to other overlays.
- A scheduling methodology which reduces the number of instructions that need to be stored in each FU thus enabling an FU with a very small memory footprint.
- A 32-bit area-efficient, time-multiplexed FU with a very small instruction memory built around a fully pipelined DSP block using just a few memory primitives.
- An architecture with a significantly reduced context switching time, thus enabling rapid application kernel changes.

The work presented in this chapter has been published in

- **X. Li**, A. K. Jain, D. L. Maskell, and S. A. Fahmy, "An Area-Efficient FPGA Overlay using DSP Block based Time-multiplexed Functional Units", *in 2nd International Workshop on Overlay Architectures for FPGAs (OLAF)*, Monterey, CA, February 2016.
- A. K. Jain, **X. Li**, S. A. Fahmy, and D. L. Maskell, "Adapting the DySER Architecture with DSP Blocks as an Overlay for the Xilinx Zynq", *ACM SIGARCH Computer Architecture News* vol. 43 no. 4, pp. 28-33, 2016.

3.2 Programmable Pipelines

Heterogeneous system on programmable chip (SoPC) platforms, such as the ARM-based Xilinx Zynq, have been successfully deployed in embedded systems which require significant computing performance within a tight power budget, such as when pre-processing at high data rates is required. While these applications have tended to use the FPGA fabric as a static accelerator, it does open up the possibility for including smaller compute engines, integrated into a larger accelerator system, such as the hypervisor-based hardware virtualized execution and management environment [39], where a rapidly configurable overlay could co-exist with both static and partially reconfigurable accelerators on the FPGA fabric. The advantage of such a system is that the FPGA hardware is able to support fast compilation, software-like programmability and run-time management, with a relatively small configuration data size and fast non-preemptive hardware context switching.

As discussed previously, an SC overlay can deliver maximum performance by replicating the compute kernel datapath and executing one iteration every clock cycle, but has a significant hardware resource overhead. Another problem is the mismatch between the hardware processing throughput and the off-chip bandwidth. For example, Figure 3.1(a) shows a small computational kernel, the ‘gradient’ benchmark from the medical imaging domain [92], while Figure 3.1(b) shows the resulting data flow graph. This simple benchmark executing on a 32-bit SC overlay operating at a modest 150 MHz has an off-chip bandwidth requirement of 4.8 GB/s (0.8 GB/s per I/O). This is greater than the Xilinx Zynq external DDR bandwidth of 4.2 GB/s, as detailed in Table 3.1 [93]. While this memory bottleneck could be solved using a streaming interface directly between the off-chip sensors and the FPGA fabric, or directly between an external Dynamic random-access memory (DRAM) and the FPGA fabric, it does show that time-sharing the FU among multiple operations of the kernel, using a CGRA-like TM overlay, may be a feasible alternative. This is particularly the case when an overlay is used withing a larger system with multiple cores all sharing the external I/O bandwidth. In this case, a small TM overlay, with its lower resource utilization, higher II and reduced throughput, is likely to be better matched to the limited off-chip bandwidth.

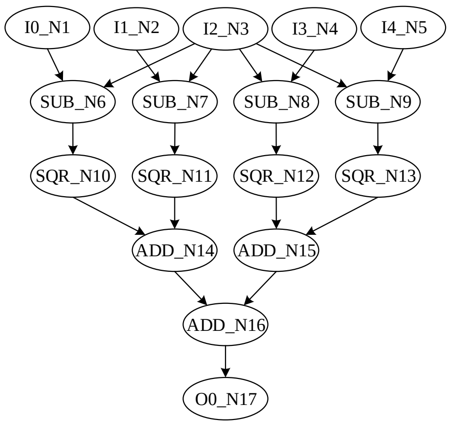
Table 3.1: Theoretical ZedBoard interface bandwidths.

Interface description	Ports	Bandwidth (GB/s)
AXI Accelerator Coherency Port (ACP)	1	2.4
AXI General Purpose (GP)	4	4.8
AXI High Performance (HP)	4	9.6
External DDR Memory	1	4.2
On-chip Memory (OCM)	1	3.6

However, as discussed in Chapter 2, existing CGRA-like TM overlays are still relatively inef-

```
#define SQR(x) ((x)*(x))
for(int i=0;i<64;i++)
  for(int j=1;j<63;j++)
    for(int k=1;k<63;k++)
      b[i][j][k] =
      SQR(a[i][j][k]-a[i][j][k-1])+
      SQR(a[i][j][k]-a[i][j][k+1])+
      SQR(a[i][j][k]-a[i][j-1][k])+
      SQR(a[i][j][k]-a[i][j+1][k]);
```

(a) C source code



(b) Data flow graph

Figure 3.1: The 'gradient' benchmark.

ficient, and hence there is scope for improvement, particularly with regards to resource utilization and speed. One way to improve the performance, as mentioned in Chapter 2, is to simplify the interconnect topology of the CGRA-like overlay. Thus, instead of the multiple FUs connected by a relatively heavyweight island style or NN interconnect network, as in many of the existing SC [6,7,15,21,85,94] and TM [12,17,35,36] overlays, it should be possible to utilize a simple linear interconnect structure, similar to that used in the highly efficient SC DeCO overlay [38]. If this linear interconnection could be further reduced to a simple direct connection between FUs, it would significantly reduce the hardware resource requirement, while still allowing application mapping by allocating DFG nodes from the same scheduling time step to the individual FUs. Additionally, using a simple low resource, but highly pipelined FU (similar to the iDEA soft core processor [56]) would better target the FPGA. This combination of linear interconnect and time multiplexed FU presents an interesting design space which will likely result in high throughput programmable architectures with reduced II and significantly reduced hardware resource requirements. In the remainder of this chapter we examine such an overlay.

3.2.1 Architecture Description

We propose to build a linear array of FUs as a programmable processing pipeline where each FU can be time multiplexed among operations present in a single scheduling stage of an ASAP sequenced data flow graph. The 32-bit pipeline consists of a streaming data interface made up of Distributed RAM acting as a FIFO, which feeds into a cascade of time-multiplexed FUs, with another Distributed RAM based FIFO at the pipeline output, as shown in Figure 3.2.

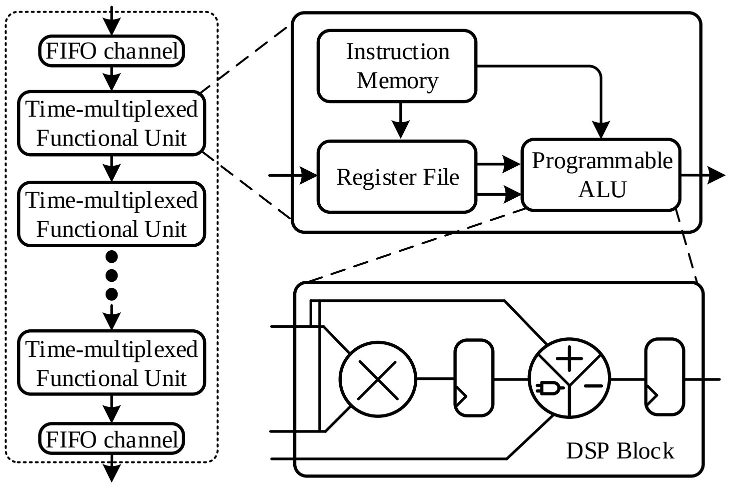

FIFO channel
Time-multiplexed Functional Unit
Time-multiplexed Functional Unit
Time-multiplexed Functional Unit
FIFO channel
Instruction Memory
Register File
Programmable ALU
DSP Block

Figure 3.2: A linear TM overlay.

As the Xilinx design tools provide four different methods to generate the FIFO IP core, we explore all possible implementations, and list the area requirement and maximum operating frequency in Table 3.2. From this table, each implementation consumes a similar amount of slice registers (≈2.0%) and LUTs (≈2.5%). However, the Block RAM based implementation and Built-in FIFO implementation require 1 extra RAMB36E1 primitive for the input and output datapaths. Since the Distributed RAM based FIFO has the highest operating frequency with a comparable area overhead, we choose this implementation as the FIFO channel.

Table 3.2: Area and frequency with different FIFO implementations.

FIFO Implementation	No. of FFs	No. of LUTs	No. of RAMB36E1s	F_{max}
Block RAM	2174 (2.0%)	1342 (2.5%)	2 (1.4%)	285MHz
Distributed RAM	2240 (2.1%)	1391 (2.6%)	1 (0.7%)	303MHz
Shift Register	2226 (2.1%)	1463 (2.7%)	1 (0.7%)	294MHz
Built-in FIFO	2154 (2.0%)	1292 (2.4%)	2 (1.4%)	241MHz

3.2.2 FU Microarchitecture

The proposed FU is a simplification of the iDEA soft processor [46], which is a DSP based fully functioning CPU optimized to achieve maximum speed on a Xilinx FPGA. However, our

FU does not require this full functionality as data is moved through the overlay in a quasi-streaming fashion. As such, there is no need for large instruction or data memory blocks based on BRAM primitives, as is seen in other CGRA-like overlays [17, 36]. Instead, a smaller instruction memory (IM) can be used. Examining the typical overlay benchmark examples from the literature [1, 2], it was determined that a depth 32 IM could be used as the identified benchmarks could all fit within this constraint. As such, a 32×32-bit IM can be implemented using LUTRAM based RAM32M primitives, resulting in a 5-bit instruction counter (IC) and a 5-bit program counter (PC). Similarly, a 32×32-bit register file (RF) is instantiated using RAM32M primitives. The ALU of an FU is based on a DSP block with three stage pipelining for maximum speed. Initially, a tag is required for each instruction to match with its corresponding FU as configuration data (e.g. instructions) is streamed through the array from FU to FU. It is customized as an 8-bit tag combined with the 32-bit instructions, which can support up to 256 FUs. This dedicated design results in some simplifications to the FU architecture, with a reduction in the number of pipeline stages compared to iDEA [46].

The proposed FU has an execution pipeline with 8 stages in total, with a latency of 1-clock cycle per stage. These are divided as follows: 2 stages for instruction fetch, 2 stages for operand indexing, 3 stages for instruction execute, and 1 stage for data transmission to the next FU. The lightweight FU is optimized for FPGA implementation, based on design macros, such as the DSP48E1 and the LUTRAM based RAM32M primitives, found in modern Xilinx FPGAs. The main components consist of an instruction memory (IM), register file (RF) and DSP-based ALU, shown as the three dashed blocks in Figure 3.3. As the FU operates as a feed-forward data flow processing element, data memory is not required in the functional unit.

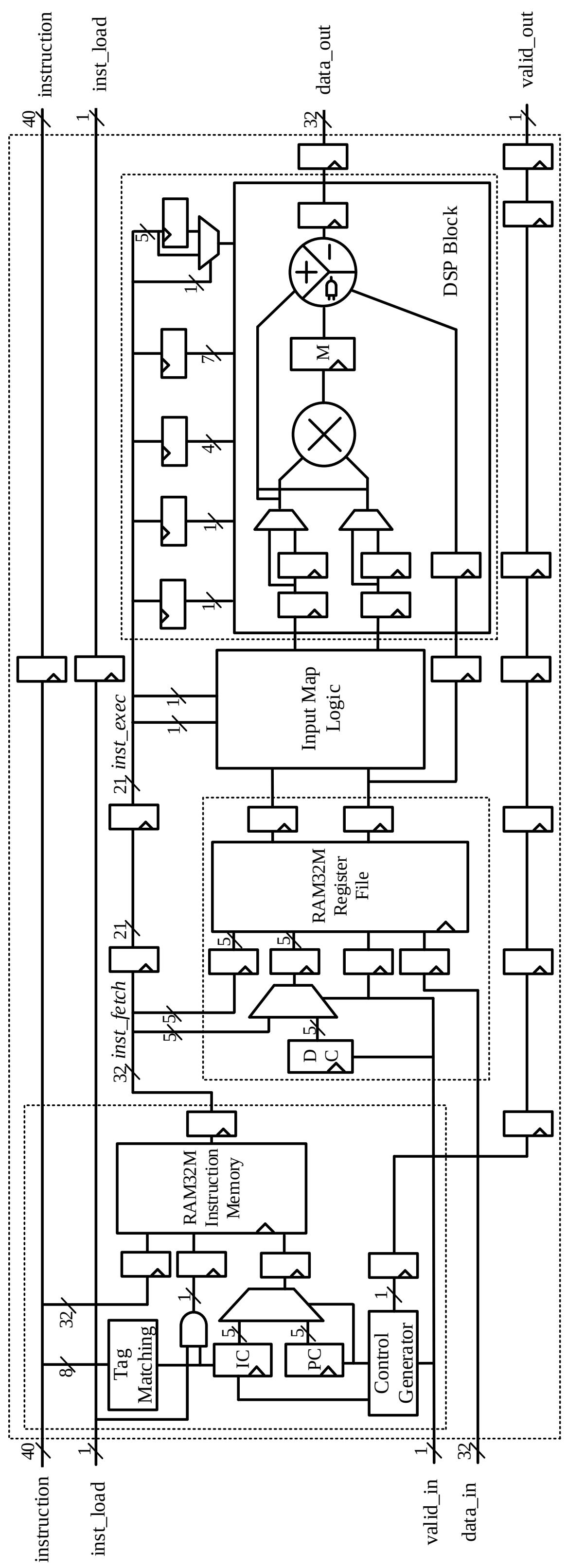


Figure 3.3: The proposed time-multiplexed functional unit.

(a) Instruction Memory

The instruction memory is used to store the list of instructions that are performed in the DFG scheduling stage and contains the arithmetic and any data bypass instructions needed to feed input data to the next scheduling stage. At initialization, or upon a hardware context switch, a 40-bit data word, made up of a 32-bit wide instruction (the kernel's context) and an 8-bit tag (used to match an instruction with its corresponding FU), is clocked to the FU instruction port from a separate 40-bit wide context memory implemented externally using BRAMs. The FU instruction ports are daisy-chained together, allowing for efficient configuration of the complete overlay. A 5-bit instruction counter (IC) is used to keep track of the instructions written to the FU. There are two types of instruction: arithmetic and data bypass (used to forward data to the next stage). The FU architecture supports a 32 entry IM implemented using RAM32M primitives. The RAM32M primitive is an efficient memory primitive implemented in LUTRAMs, which can be configured as a 32 deep 2-bit wide quad port (3 read, 1 read/write), 32 deep 4-bit wide dual port (1 read, 1 read/write) or a 32 deep 8-bit wide single port (1 read/write) memory. Since the IM writes occur only once at context initialization, we multiplex the read and write addresses enabling us to use the single port variant which uses just four RAM32M primitives to instantiate the 32×32 IM. An extra pipeline stage is added to the output of the RAM32M primitive to ensure that it operates at close to its maximum frequency. The *inst_fetch* signal (at the output of the IM block) represents the instruction which has been read from the IM block, and appears 2 clock cycles after the program counter (PC) initiates the instruction fetch, as shown in Figure 3.3. A description of the 32-bit internal instruction format, using an *ADD R3, R5* operation as an example, is shown in Table 3.3. From this table, it can be seen that a 32-bit instruction has four sections, the 19-bit DSP block configuration, the 2-bit input map multiplexing, two 5-bit source operand addresses, with the last bit reserved for future use. Upon the completion of the context write cycle, each FU contains the necessary instructions in its IM and the instruction count in its IC register.

Table 3.3: Instruction format.

FU part	ALU Control						Input Map Control		RF Control		Reserved
Signals	alumode	inmode	opmode	cea2	ceb2	usemult	split	immop	src-1	src-2	
No. of bits	4	5	7	1	1	1	1	1	5	5	1
Locations	[31:28]	[27:23]	[22:16]	15	14	13	12	11	[10:6]	[5:1]	0
ADD R3, R5	0000	00000	0110011	1	1	0	1	0	00011	00101	0

(b) Register File

The RF is used to hold the 32-bit data streamed from the input FIFO, or the previous FU stage. When valid data is available from the input FIFO (or previous stage), the data counter (DC) controls the data transfer to the RF using simulated load instructions. Data transfer is initiated by the *valid_in* signal going high, indicating that data is being streamed into the FU. For each transfer, the DC is incremented resulting in data being written into the RF in sequential locations. When the data transfer is complete and all data is available in the register file the *valid_in* signal goes low. The control generator then asserts a *control* signal which triggers the IM to start sending operand addresses to the RF and configuration data to the DSP block, with the specific instruction to be executed determined by the PC. The RF requires two read ports and one write port, but as reads and writes do not occur simultaneously we again multiplex one port enabling both read and write operations. Thus the 32 entry RF requires 8 RAM32M primitives configured as a 1 read/write and 1 read port memory (32-deep by 4-bit wide). The address bus is shared between the DC and the source-1 address using a multiplexer at the input of the address register. Similar to the IM, there is an extra pipeline register at the output of the RAM32M-based RF to ensure maximum operating frequency. Thus, it takes 2 cycles to source the operands from the RF for a specific instruction (*inst_fetch*). Hence, the *inst_exec* signal to the DSP block and the input map logic requires two additional register stages to ensure synchronization, as shown in Figure 3.3. After executing all the instructions related to the scheduling stage, including the data output to the next stage, or output FIFO, the DSP block flushes its internal pipeline and the program counter resets, allowing the same sequence of instructions to be issued over and over again, instead of needing to store a large set of instructions for each cycle. Thus, each FU only needs to execute a small number of instructions, allowing for a small IM and RF design.

(c) ALU

The other major block in the FU is the ALU, which consists of a DSP48E1 primitive, two 32-bit registers, one at the C input port for pipeline balancing and another at P output port, and an 19-bit register for holding the ALU configuration data. The DSP48E1 primitive consists of a pre-adder, a multiplier, an ALU, four ports for input data, and one port for output, as shown in Figure 3.4, and can be configured to support various operations such as multiply, add, sub, bitwise OR, etc. These functions are determined by a set of dynamic control inputs that are wired to the configuration registers. As instruction decoders are not used, the instruction format must explicitly specify the read addresses and the modes of operation of the DSP block directly, allowing us to achieve a relatively high frequency.

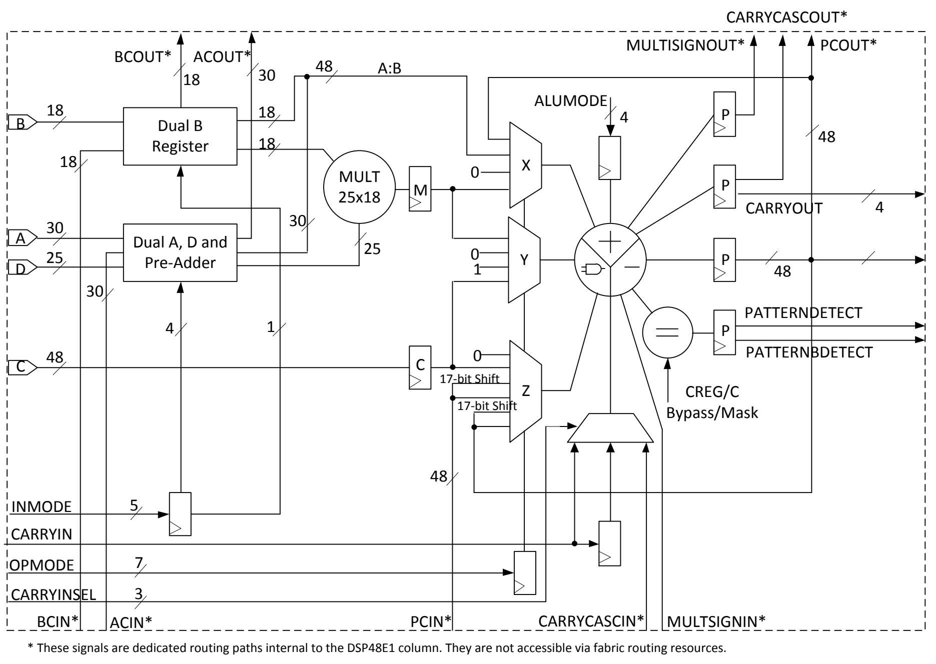


Figure 3.4: DSP48E1 primitive. [4]

The FU of Figure 3.3 was synthesized and mapped to a Xilinx Zynq XC7Z020-1CLG484C using Xilinx ISE 14.7. A frequency of 325 MHz was achieved while consuming 1 DSP block, 160 LUTs and 293 FFs (or 1 DSP block and 81 slices). A complete pipeline consisting of 8 FUs and the 2 I/O FIFOs, consumed 8 DSP blocks, 808 LUTs and 1077 FFs (that is 8 DSP blocks and 459 slices) representing less than 4% of the Zynq FPGA resources while operating at a slightly reduced frequency of 303 MHz. Figure 3.5(a) and 3.5(b) shows the resource consumption and F_{max} of the proposed processing pipeline as we increase the number of FUs in the pipeline. When a single FU is mapped to a more capable XC7VX485T Virtex 7 device, we achieved a frequency in excess of 600 MHz for the same resource utilization. The maximum configuration time for a single pipeline consisting of 8 FUs is 0.85 µs at 300 MHz. This assumes that all 8 IMs require the full 32 instructions and that the kernel contexts are already preloaded into the external context memory.

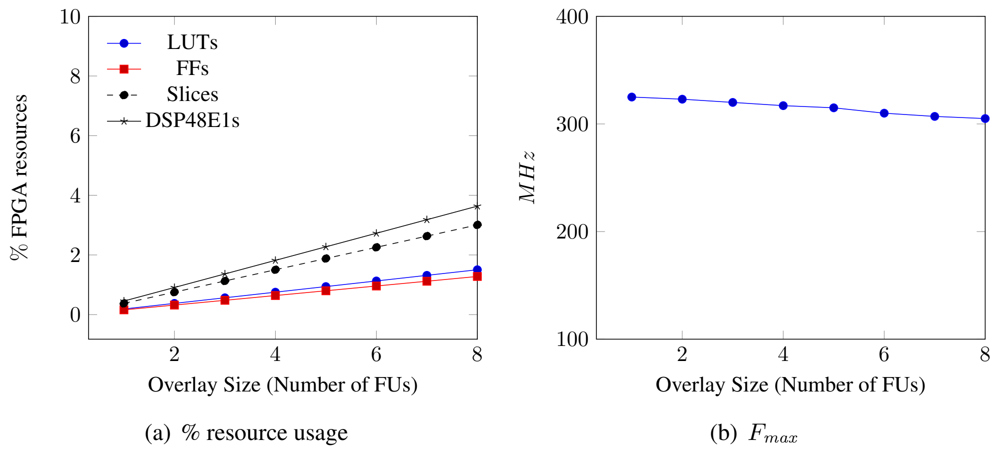


Figure 3.5: Overlay scalability results on Zynq platform.

3.2.3 Overlay Control and Functionality

A back-pressure control circuit is built around the input FIFO channel to manage the functionality of the proposed overlay, as shown in Figure 3.6. There are three control signals which indicate the duration for instruction load, overlay setup and data write respectively, referred to as *inst_load*, *reg_wren*, and *data_wren*. Initially, FU instructions are read from the memory and streamed through the daisy-chained FUs. During instruction load (when the *inst_load* is high), both the write enable port of the FIFO and the valid signal (*valid_out*) for data output are disabled. After instruction load, two integers are written to the back-pressure control circuit. The first represents the number of data elements that need to be loaded into the first FU for a specific compute kernel while the other is equal to the II minus one (II-1) and determines the interval between data loads. These values are written into the controller when the *reg_wren* signal is active (high). The process of instruction load and overlay setup represent the initialization of the overlay.

The dashed box on the left hand side of Figure 3.6 acts as the control module for the write enable port of the FIFO, while the other dashed box on the right hand side contains the logic to control the read enable port of the FIFO. The write enable signal (*wr_en*) is determined as shown in the truth table of Table 3.4. Data is written into the FIFO when *wr_en* is high. The read enable signal (*rd_en*) for the FIFO is generated from a counter which determines the number of cycles needed to load input data to the FU. The counter starts counting from 0 when the *empty* signal

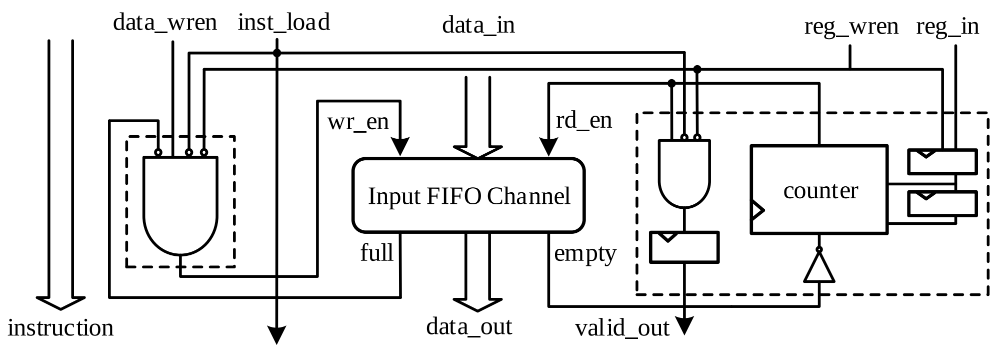


Figure 3.6: Back-pressure control circuit.

goes low (indicating that data is available in the FIFO). The counter counts up until the count value equals II-1, at which point it rolls over back to zero. The *rd_en* signal is valid only while the counter is less than the number of data elements that need to be loaded into the first FU. If the counter value is greater than or equal to the number of data elements that need to be loaded into the first FU, the *rd_en* signal is forced low and the input data is buffered in the FIFO.

Table 3.4: Truth table of the *wr_en* signal.

inst_load	*reg_wren*	*data_wren*	*full*	*wr_en*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

The working mechanism of the proposed overlay is shown in Figure 3.7. As can be seen from this diagram, the initialization of the overlay for the ‘gradient’ benchmark takes 22 clock cycles in total, consisting of 20 cycles for instruction load and 2 cycles for overlay back-pressure configuration. Then, input data is fed into the FIFO in a streaming fashion with an II of 11. The latency (47 cycles) is obtained by calculating the interval between the input of the first FU and the output of the last FU.

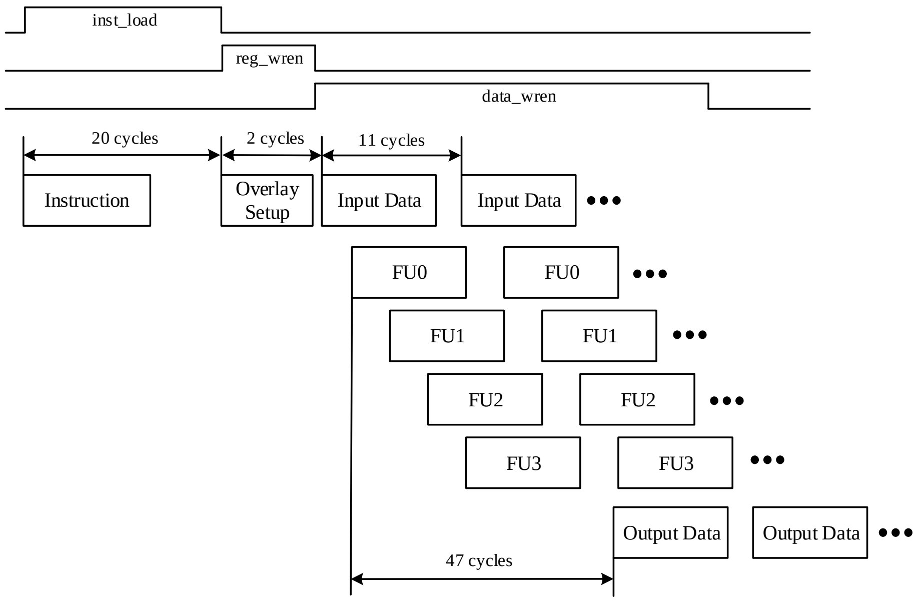


Figure 3.7: Flow chart of overlay functionality.

3.3 Compiling to the Overlay

An overlay has two separate design processes: the physical overlay implementation on the FPGA, and the application mapping to the overlay. While the design and implementation of the overlay relies on the conventional hardware design flow using vendor tools, this process is done offline, once only, and so does not impact the compute kernel implementation of an application. We then use an in-house automated compilation flow to provide a rapid, vendor independent, mapping to the overlay. The mapping process comprises DFG extraction from high level compute kernels, scheduling of the DFG nodes onto the overlay, and finally, the instruction generation for each FU. This is also done offline. Then at power-on, the bitstream for the overlay and any other unrelated hardware components are used to configure the FPGA. Subsequent to this, the ARM processor loads the kernel configuration into the overlay pipeline and initiates kernel execution. Our mapping flow is described below using the previous example.

HLL to DFG Conversion: The HercuLeS HLS tool [95] is used to transform a ‘C’ description of the compute kernel to a DFG text description, as shown in Table 3.5, where nodes represent operations and edges represent data flow between operations, as shown in Figure 3.1(b).

Table 3.5: DFG description of the compute kernel ‘gradient’ from Figure 3.1(b).

```
digraph graphname {
N1 [ntype="invar", label="I0_N1"];
N2 [ntype="invar", label="I1_N2"];
N3 [ntype="invar", label="I2_N3"];
N4 [ntype="invar", label="I3_N4"];
N5 [ntype="invar", label="I4_N5"];
N6 [ntype="operation", label="sub_N6"];
N7 [ntype="operation", label="sub_N7"];
N8 [ntype="operation", label="sub_N8"];
N9 [ntype="operation", label="sub_N9"];
N10 [ntype="operation", label="sqr_N10"];
N11 [ntype="operation", label="sqr_N11"];
N12 [ntype="operation", label="sqr_N12"];
N13 [ntype="operation", label="sqr_N13"];
N14 [ntype="operation", label="add_N14"];
N15 [ntype="operation", label="add_N15"];
N16 [ntype="operation", label="add_N16"];
N17 [ntype="outvar", label="O0_N17"];
N1 -> N6;
N2 -> N7;
N3 -> N6;
N3 -> N7;
N3 -> N8;
N3 -> N9;
N4 -> N8;
N5 -> N9;
N6 -> N10;
N7 -> N11;
N8 -> N12;
N9 -> N13;
N10 -> N14;
N11 -> N14;
N12 -> N15;
N13 -> N15;
N14 -> N16;
N15 -> N16;
N16 -> N17;
}
```

Operation Scheduling: Scheduling is used to generate a sequenced DFG, with nodes in each scheduling stage being allocated to a single FU for execution. Here, the set of instructions from the sequenced DFG is identified, then the cycle-by-cycle execution pattern is formed as shown in Table 3.6, and lastly the 32-bit FU instructions are generated.

Table 3.6: First 48 cycles of the schedule with II=11.

cycle	FU0	FU1	FU2	FU3	
1	Load R0[1]				
2	Load R1				
3	Load R2				
4	Load R3				
5	Load R4				
6					
7					
8	SUB (R2 R0)[2]				
9	SUB (R2 R1)				
10	SUB (R3 R2)				
11	SUB (R4 R2)				
12	Load R0				
13	Load R1				
14	Load R2	Load R0			
15	Load R3	Load R1			
16	Load R4	Load R2			
17		Load R3			
18					
19	SUB (R2 R0)				
20	SUB (R2 R1)	SQR (R0 R0)			
21	SUB (R3 R2)	SQR (R1 R1)			
22	SUB (R4 R2)	SQR (R2 R2)			
23	Load R0	SQR (R3 R3)			
24	Load R1				
25	Load R2	Load R0			
26	Load R3	Load R1	Load R0		
27	Load R4	Load R2	Load R1		
28		Load R3	Load R2		
29			Load R3		
30	SUB (R2 R0)				
31	SUB (R2 R1)	SQR (R0 R0)			
32	SUB (R3 R2)	SQR (R1 R1)	ADD (R0 R1)		
33	SUB (R4 R2)	SQR (R2 R2)	ADD (R2 R3)		
34	Load R0	SQR (R3 R3)			
35	Load R1				
36	Load R2	Load R0			
37	Load R3	Load R1	Load R0		
38	Load R4	Load R2	Load R1	Load R0	
39		Load R3	Load R2	Load R1	
40			Load R3		
41	SUB (R2 R0)				
42	SUB (R2 R1)	SQR (R0 R0)		ADD (R0 R1)	
43	SUB (R3 R2)	SQR (R1 R1)	ADD (R0 R1)		
44	SUB (R4 R2)	SQR (R2 R2)	ADD (R2 R3)		
45	Load R0	SQR (R3 R3)			
46	Load R1				
47	Load R2	Load R0			
48	Load R3	Load R1	Load R0		output

[1] The load operations are not real instructions from the IMs.
[2] The arithmetic instructions shown in this table are *inst_fetch* from Figure 3.9.

By using a simple ASAP schedule, we can allocate the nodes in each stage to a different FU, which in the 'gradient' example (as shown in Figure 3.1) results in 4 stages, with the FU in each stage being time-multiplexed among stage operations using a direct (non-programmable) connection between FUs. That is, the first stage would contain four subtract operations which would execute on the first FU, the second stage would contain four multiplication operations executing on the second FU, and so on.

As indicated in Table 3.6, FU0 waits for an initial 5 cycles (from clock cycle 1 to clock cycle 5) while the data is loaded into the RF. As soon as loading has completed, FU0 is triggered by the *control* signal (in the 6th clock cycle) and starts fetching the 4 SUB instructions from the IM (clock cycle 8 to clock cycle 11). After a delay of 2 cycles, the DSP block executes the 4 SUB instructions (clock cycle 10 to clock cycle 13), one every cycle, using the data from the RF. Note that as all operations are in the same scheduling stage there is no data dependency between operations. FU0 starts sending the resulting data to FU1 on the 14th clock cycle (due to the 3 stage internal pipeline in the DSP block and 1 cycle for the output register) and then waits for the cycle to repeat. Two extra cycles are needed for reading the instructions from the IM of FU0, and hence we use the back-pressure control circuit (Figure 3.6) at the input FIFO channel to pause further data input (from clock cycle 6 to clock cycle 11).

The operation of FU1 is similar to that of FU0. The output data from FU0 (the SUB instruction at cycle 10) is input to FU1 at cycle 14 due to the pipeline depth (2 cycles for instruction fetch, 3 cycles for the DSP block and 1 cycle for the output register). Data input into the RF of FU1 takes 4 cycles (clock cycle 14 to clock cycle 17), before FU1 is triggered (on the 18th clock cycle) and starts executing the 4 multiply (SQR) instructions on the data available in its RF. Once the first instruction is completed, the FU outputs the processed data to FU2 (for 4 cycles starting from the 26th clock cycle) and waits for the next set of data from FU0, upon which it repeats the previous operation. FU2 and FU3 also execute in a similar manner.

According to the above schedule, the II of the overlay is determined by the most compute intensive FU, i.e. the maximum of the number of data load operations plus the number of execution operations with 2 additional clock cycles needed to read the instructions from the IM, as in Equation 3.1. Thus the II for this example would be 11, consisting of 5 cycles for data entry, 2 cycles for the internal pipeline when reading instructions from the IM, and 4 cycles for the 4 subtract operations. Note that multiplexing the kernel operations of the DFG in Figure 3.1(b) to a single FU would result in an II of 17 (5 loads, 11 operations, and 1 store), assuming best case execution without NOP insertions, while a spatially configured overlay would require 11 FUs with an II of 1. Furthermore, this means that only a small subset of all possible instructions need to be stored in each FU, resulting in a very small memory requirement overcoming the problem of the large resource overheads due to the instruction storage requirements in the overlays described in Chapter 2.

$$II = \max_{FU}\{\#load + \#op + 2\} \tag{3.1}$$

The instructions for the 'gradient' kernel with ASAP scheduling are shown in Table 3.7. Since the output data of one FU is directly input into the register file of the next FU, no specific instruction is required for loading the data at each stage. During the initialization time, each FU checks the upper 8-bit tag of the external instructions, and only stores those where the tag corresponds to its internal tag. With this working mechanism, the proposed scheduling approach allows us to store just a small number of instructions for each FU, resulting in very small memories and hence an area efficient overlay. Other time multiplexed architectures, such as in [22] need large instruction memories as the FUs in those architectures need to store the full list of instructions executing every cycle, resulting in a much higher resource utilization.

Table 3.7: Instructions for kernel 'gradient'

```
00000000_0011000000110011110_10_00010_00000_0   //SUB R2, R0
00000000_0011000000110011110_10_00010_00001_0   //SUB R2, R1
00000000_0011000000110011110_10_00011_00010_0   //SUB R3, R2
00000000_0011000000110011110_10_00100_00010_0   //SUB R4, R2
00000001_0000100010000101001_00_00000_00000_0   //SQR R0, R0
00000001_0000100010000101001_00_00001_00001_0   //SQR R1, R1
00000001_0000100010000101001_00_00010_00010_0   //SQR R2, R2
00000001_0000100010000101001_00_00011_00011_0   //SQR R3, R3
00000010_0000000000110011110_10_00000_00001_0   //ADD R0, R1
00000010_0000000000110011110_10_00010_00011_0   //ADD R2, R3
00000011_0000000000110011110_10_00000_00001_0   //ADD R0, R1
```

To verify the functionality, a four FU version of the overlay for the 'gradient' benchmark was simulated using ModelSim SE-64 10.6. Figure 3.8 shows the initialization of the overlay. Initially, the overlay waits 120 ns for the global reset to finish, which in this simulation occurs within the first 6 clock cycles. The loading of the instructions is controlled by the *inst_load* signal, happening from cycle 7 to cycle 26. During this period, there is no data in the programmable pipeline and only instruction load is allowed. After instruction load (*inst_load* is low), the overlay setup phase occurs (*reg_wren* is high), and the two values which control the transfer of data through the overlay (the number of data elements that need to be loaded and the II-1) are written to the back-pressure control circuit. Lastly, external data is fed into the FIFO when *wr_en* is active, as shown in Figure 3.8. In this case, three packets of input data ("1, 2, 3, 4, 5", "1, 3, 5, 7, 9" and "0, 1, 0, 1, 0") are sent to the input FIFO. Any other data on *data_in* is ignored.

Once the initialization of the overlay is finished, the datapath from the input FIFO to the output FIFO functions as shown in Figure 3.9. Starting at cycle 31, the first five data elements are then read from the input FIFO and fed into FU0, corresponding to the five data loads from Table 3.6, as shown in Figure 3.9(a). The *control* signal for the IM is triggered once the *valid_in* signal becomes low. Two cycles of pipeline delay are required for the IM to access the instructions and present the source operand addresses to the RF, and another 2 cycles are needed to align the configuration (instruction execution) to the DSP block, as seen in the *inst_fetch* and *inst_exec* signals of Figure 3.9(a). The location of the *inst_fetch* and *inst_exec* signals in the FU pipeline are shown on Figure 3.3. Data is output from the FU four cycles after the instruction is executed, corresponding to a three-stage pipelined DSP block and an additional register at the FU output. The back-pressure control delays the second input data package, which starts transmitting at cycle 42, indicating the II equals to 11.

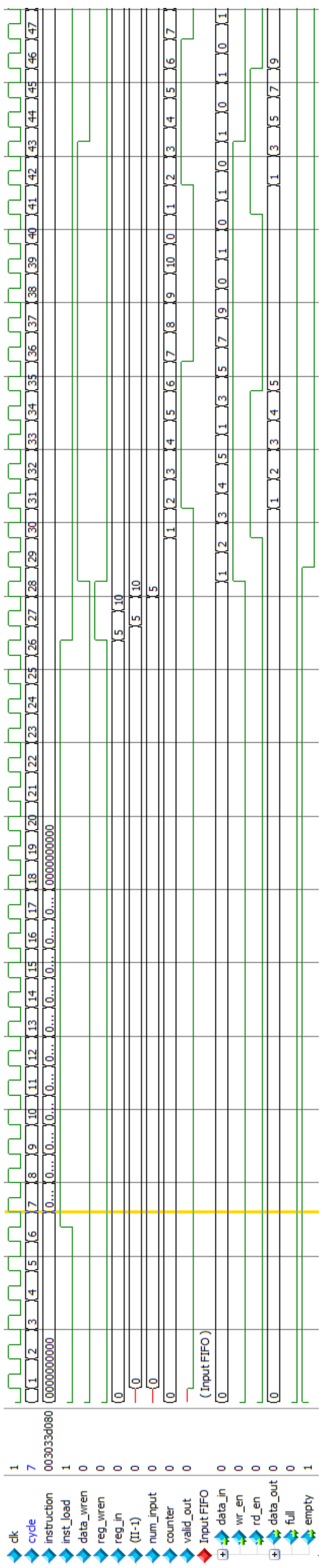
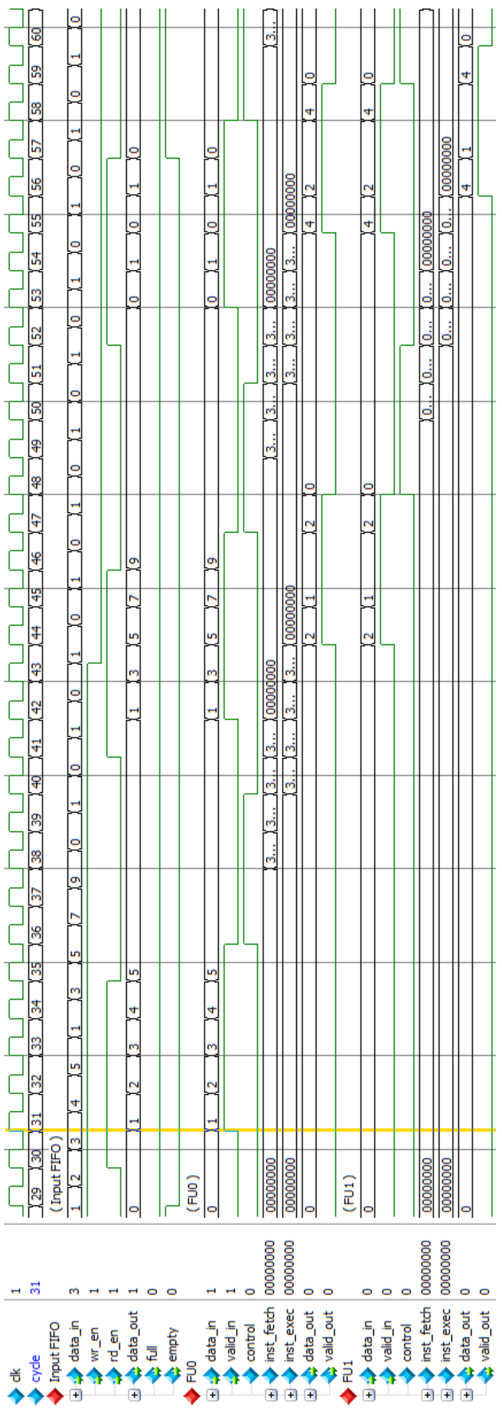
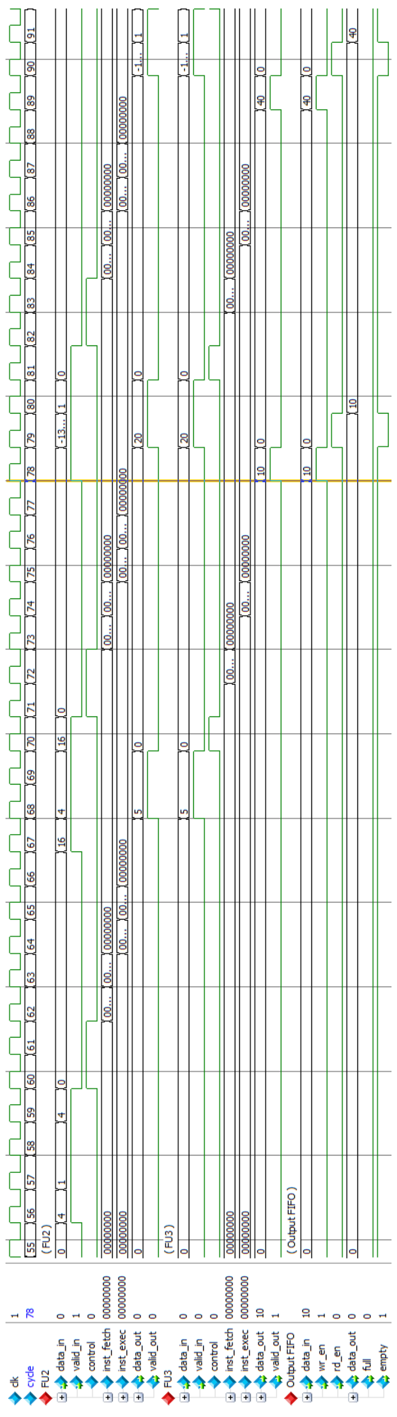


Figure 3.8: Initialization of the overlay.



(a) Data from input FIFO to FU1



(b) Data from FU2 to output FIFO

Figure 3.9: Simulation result of kernel ‘gradient’.

Figure 3.9(b) shows the functionality of FU2, FU3 and the output FIFO. The output of FU3 is valid at clock cycle 78 and clock cycle 89 for the first input data package ("1, 2, 3, 4, 5") and the second input data package ("1, 3, 5, 7, 9"), respectively. Thus the latency from data input of FU0 to the data output of FU3 is 47, which matches with the flow chart shown in Figure 3.7 and the schedule of Table 3.6.

3.4 Experimental Evaluations

We evaluate the performance of the proposed overlay and that of our compiler using a set of compute kernels extracted from compute intensive applications from the literature [1, 2] (the DFG kernels in graphical format are shown in Appendix A), as shown in Table 3.8. The graph depth corresponds to the number of FUs needed in the proposed overlay, while the effective operations per cycle (eOPC) is the ratio of DFG nodes (op nodes) and the II, which ranges from 0.9 to 1.8. The II is high for benchmarks with a large number of I/O nodes and high average parallelism.

Table 3.8: DFG characteristics of benchmark set.

No.	Benchmark Name	Characteristics						
		I/O nodes	graph edges	op nodes	graph depth	average parallelism	II	eOPC
1.	chebyshev	1/1	12	7	7	1.00	6	1.2
2.	mibench	3/1	22	13	6	2.16	14	0.9
3.	qspline	7/1	50	25	8	3.25	19	1.3
4.	sgfilter	2/1	27	18	9	2.00	13	1.4
5.	poly5	3/1	43	27	9	3.00	19	1.4
6.	poly6	3/1	72	44	11	4.00	25	1.8
7.	poly7	3/1	62	39	13	3.00	24	1.6
8.	poly8	3/1	51	32	11	2.90	21	1.5

To demonstrate the benefits of the proposed overlay, we compare it to two of the more architecture-focused SC overlays from the literatures [7, 85] and to RTL implementations of the same kernels using Vivado HLS 2016.1[1]. For all these implementations we use the minimal

[1]The RTL generated by Vivado HLS is a customized hardware design which generally has the best performance in terms of power, delay and area.

number of FUs/hardware for the benchmark implementations. This is to observe the effect of FU reduction on the area requirement. Figure 3.10 shows the number of FUs required for the proposed linear TM overlay compared to that of the SC DySER overlay [85] and DSP-based overlay in [7] for each of the benchmarks in Table 3.8, respectively. Since the DySER Overlay is limited in size (to a 5×5 overlay) on the Zynq-7020 device, it cannot fit the benchmarks which have more than 25 operation nodes. Figure 3.10 shows a significant reduction in the number of FUs required by the proposed overlay. However, as seen in Table 3.8 this reduction in the number of FUs comes at the expense of an increase in the II.

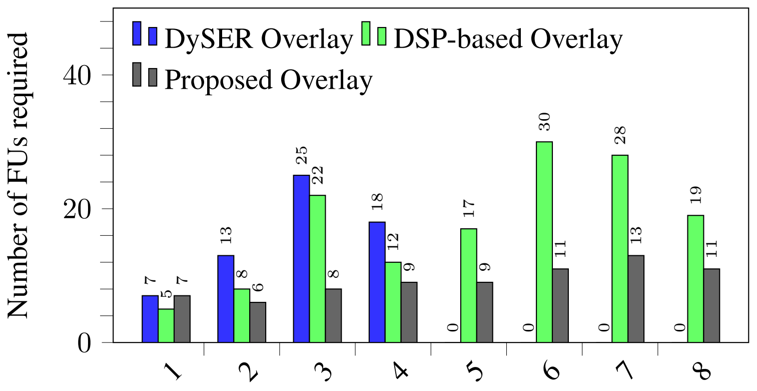


Figure 3.10: Number of FUs required for the benchmarks.

Because the different implementations we are attempting to compare use differing hardware resources, it is difficult to compare them directly. Instead we use a single equivalent slices (or eSlices) metric, where we assume that 1 DSP block is equivalent to 60 slices based on the ratio of slices/DSP on the Zynq XC7Z02-1CLG484C (which is approximately 60). That is, if the FU in the proposed overlay consumes 1 DSP block and 81 slices it uses 141 eSlices.

We also examine the resource area in eSlices and the throughput in Giga-operations/s (GOPS) for the 3 implementations on a Xilinx Zynq XC7Z02-1CLG484C for the benchmarks in Table 3.8, as shown in Table 3.9. The FPGA hardware resource consumption for the proposed overlay shows a significant reduction compared to the DSP-based overlay (up to 63% FUs and 85% eSlices), while using just 35% more resources than the Vivado implementations, as shown in Figure 3.11. The proposed linear TM overlay has a significant reduction in the throughput (ranging from 6× to 22×) compared to the DSP-based overlay and the Vivado implementations, due to the larger II, which is acceptable in cases when a low to moderate throughput is sufficient. The important point here is that compared to the Vivado implementations, the linear TM overlay has comparable area but with a reduced throughput, while the SC DSP-based overlay has comparable throughput but with an increased area. In terms of the throughput per unit area (in MOPS/eSlice) the three different implementations range from 0.26-0.35, 1.04-1.48, and 4.8-11.5 for the proposed TM overlay, the DSP-based SC overlay and a direct HLS implementation, respectively. Hence, even though the TM overlay is less efficient, in terms of MOPS/eSlice, then either the SC overlay or a direct implementation using Vivado, its small area consumption represents another useful design alternative for overlay design space exploration.

Table 3.9: Area and throughput comparison.

No.	Benchmark	Proposed Overlay		DSP-based Overlay [7]		Vivado HLS	
	Name	Throughput	Area	Throughput	Area	Throughput	Area
1.	chebyshev	0.35	987	2.35	1900	2.21	265
2.	mibench	0.29	846	4.36	3040	3.51	305
3.	qspline	0.40	1128	8.71	8360	6.11	1270
4.	sgfilter	0.42	1269	6.03	4560	4.59	645
5.	poly5	0.43	1269	9.05	6460	7.02	765
6.	poly6	0.53	1551	14.74	11400	11.88	1455
7.	poly7	0.49	1833	13.07	10640	10.92	1025
8.	poly8	0.46	1551	10.72	7220	8.32	1025

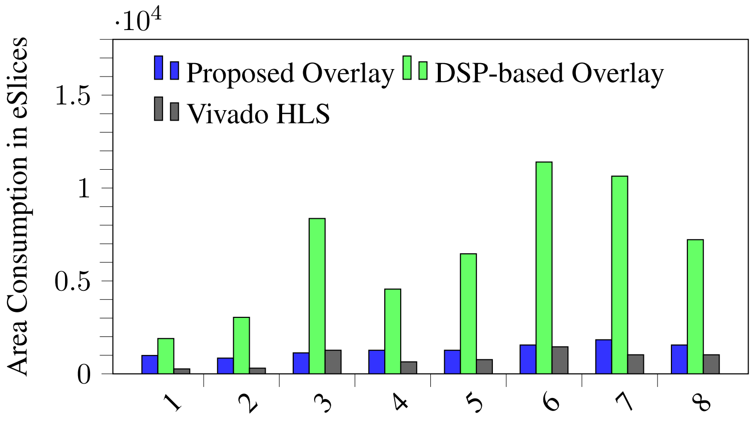


Figure 3.11: Area comparison for the benchmarks.

The context configuration data (CCD) required for the three different implementations is shown in Table 3.10. The CCD of the benchmark set for the proposed overlay ranges from 65 Bytes to 410 Bytes. Thus, the worst case context switch between kernels takes 82 cycles

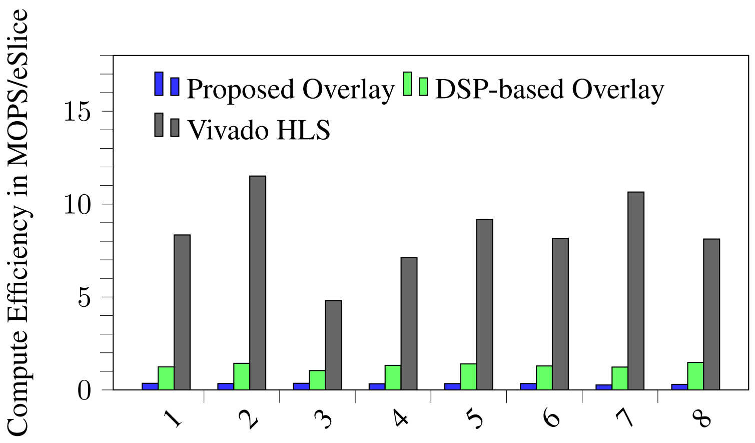


Figure 3.12: Compute efficiency comparison for the benchmarks.

(using a 40-bit wide context data word), which at 300 MHz is 0.27 us. For the SC DSP-based overlay [7], with a worst case of 323 Bytes of configuration data, configuration takes 13 us. This is 48× more than that of the proposed overlay as it does not use a local context memory, and instead configuration data must come from external memory, which is much slower. For the HLS implementation, we assume a partial reconfiguration (PR) bitstream size of 75 KBytes, which is just large enough for all the benchmarks, which range from 26 KBytes to 74 KBytes. On the Zynq platform this region requires a configuration time of 200 us, which is almost 740× slower than the proposed overlay. Thus the proposed overlay allows for much faster context switching between kernels, compared to the DSP-based overlay and the Vivado HLS implementation.

Table 3.10: Comparison on context configuration data.

No.	Benchmark	Proposed Overlay		DSP-based Overlay [7]		Vivado HLS	
	Name	Instructions	CCD	Tiles[1]	CCD	PR Frames[2]	CCD
1.	chebyshev	13	65Bytes	5	54Bytes	64	26KBytes
2.	mibench	26	130Bytes	8	86Bytes	92	37KBytes
3.	qspline	54	270Bytes	22	237Bytes	184	74KBytes
4.	sgfilter	30	150Bytes	12	129Bytes	92	37KBytes
5.	poly5	48	240Bytes	17	179Bytes	128	52KBytes
6.	poly6	79	395Bytes	30	323Bytes	164	66KBytes
7.	poly7	82	410Bytes	28	301Bytes	128	52KBytes
8.	poly8	65	325Bytes	19	204Bytes	128	52KBytes

[1] Proposed Overlay: 5 Bytes per instruction. DSP-based Overlay: 10.75 Bytes per tile.
[2] Vivado HLS: 36 frames in a DSP tile (10 DSP blocks), 28 frames in a CLB tile (50 CLBs), 1 frame size is equal to 404 Bytes.

3.5 Summary

In this chapter, we have presented an area efficient FPGA overlay that uses a linear connection of time-multiplexed FUs based on the Xilinx DSP48E1 macro. While the proposed overlay exhibits a lower throughput than spatially configured overlays due to the much larger II, it has a significantly reduced FPGA resource requirement and a much lower context switching time. Our experimental evaluation shows that for a range of benchmarks, the proposed overlay delivers a throughput which is 6× to 22× less than the SC DSP-based overlay and Vivado implementations, but which uses 85% fewer eSlices than the DSP-based overlay and just 35% more eSlices than the Vivado implementations.

We have demonstrated that the linear TM overlay can achieve significant area reduction and a faster context switch at the cost of a reduced throughput. In the next chapter, we examine the limitations of the proposed overlay, and present architectural enhancements along with a better scheduling strategy to reduce the II, which results in an improved throughput and hence an improved compute efficiency.

Chapter 4

Architectural Enhancements

4.1 Introduction

In this chapter, we examine the limitations of the linear TM overlay [96] presented in Chapter 3 and explore a number of architectural enhancements to improve the throughput. While the overlay of Chapter 3 is designed to be an area efficient architecture with fast context switching, the average compute efficiency is relatively low in terms of MOPS/eSlice compared to the DSP-based SC overlay [7] and implementations designed directly using Vivado HLS, as shown in Figure 4.1.

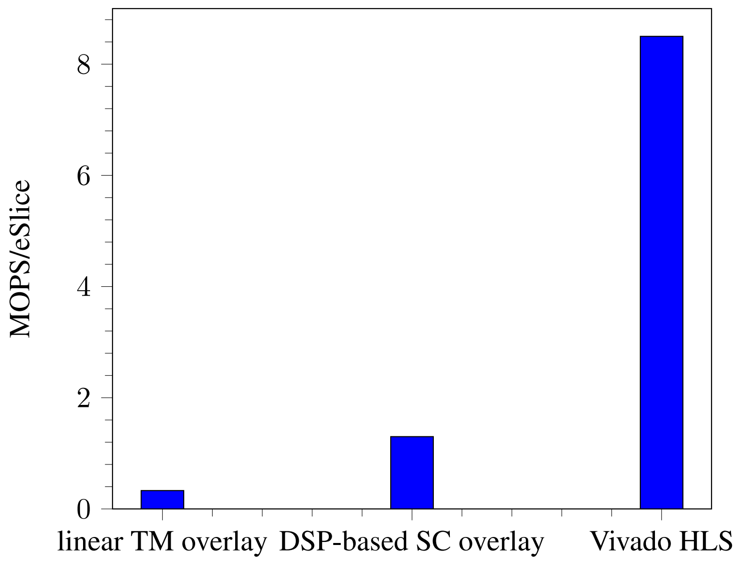
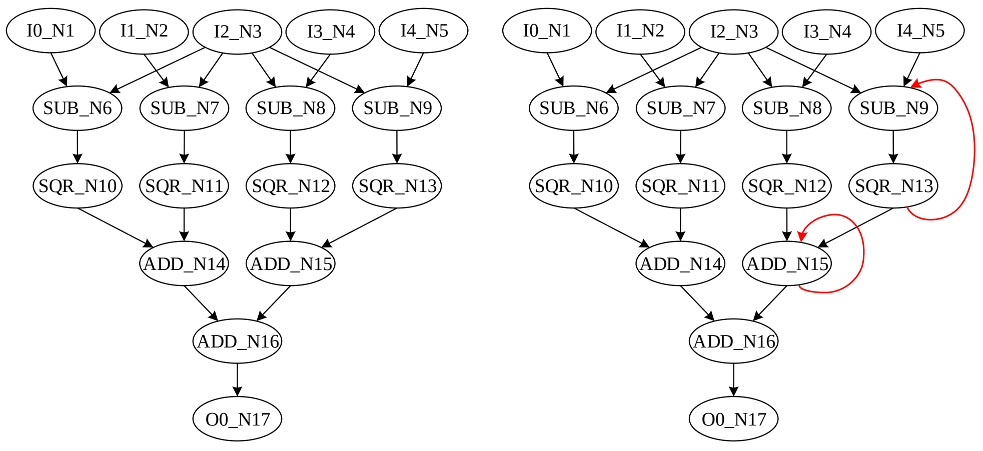


Figure 4.1: Average compute efficiency in MOPS/eSlice.

This low efficiency is mainly due to the low throughput caused by the large II for a given compute kernel, because of the non-overlap of data load and instruction fetch, as can be seen from the schedule of Table 3.6. The II of the overlay presented in Chapter 3 is obtained by the maximum of the number of data load operations plus the number of execution operations with 2 additional clock cycles needed to read the instructions from the IM. As a result, the data elements for the next iteration of the kernel cannot be fed into the RF until all the instructions have been fetched from the IM. To make it possible for the RF and IM of an FU to function simultaneously, the RFs are redesigned as rotating register files, realized by introducing a simple offset counter with two adders in addition to the original RF. This architectural enhancement leads to a significant reduction in II, with a very small increase in the area overhead. The II can be further reduced (by half) by replicating the data processing part of the FU and increasing the data I/O to 64-bit, while the IM and other control circuitry are reused.

Another disadvantage of the proposed overlays is that it can only handle feed-forward DFGs, as shown in Figure 4.2(a). When the DFG has inter-dependencies such as the red arrows in Figure 4.2(b), the corresponding intermediate outputs need to be written back to the FU for execution. To address this issue, we change the FU mapping by adding write-back support to the new FU design. Beside the benefit of supporting feedback DFGs, the FU write-back also provides an opportunity to significantly reduce the number of FUs by clustering DFGs with inter dependency, at the cost of a moderate increase in the II compared to the overlay with just the rotating register file. Additionally, without FU write-back, the size (depth) of the overlay is determined by the critical path of the DFG, and hence, the overlay has to be reconfigured whenever a new compute kernel requires a change in the depth of the DFG. FU write-back support is achieved by modifying the instruction format and adapting the RAM32M-based RF from a dual port configuration to a quad port configuration, along with some additional circuitry to control write-back into the datapath.

The resource utilization and operating frequency for the modified FU designs with a rotating register file, a replicated data path and with FU write-back are then examined. Additionally, the impact of the depth of the internal write-back path (IWP) for the version with FU write-back is analyzed. Compared to the original version (presented in Chapter 3), the modified overlays can achieve up to 2.4$\times$ higher throughput in GOPS, 93.7% higher compute efficiency in MOPS/eS-



(a) Feed-forward DFG (b) Feedback DFG

Figure 4.2: Two kinds of DFGs.

lice and a 43.7% lower latency in ns. The main contributions presented in this chapter can be summarized as follows:

- Replacing the original register file in the FU with a rotating register file. This is beneficial as it allows the concurrent execution of arithmetic operations with data load/store operations when there is no conflict.

- Replicating the data processing part of the FU and increasing the data I/O to 64-bit further reduces the II by half, while the IM and other control circuitry are reused.

- Adding write-back support to the FU, which makes it possible for the proposed overlay to handle feedback DFGs with a fixed architecture, hence also avoiding frequent overlay reconfiguration.

The work presented in this chapter has been published in

- **X. Li**, A. K. Jain, D. L. Maskell, and S. A. Fahmy, “A Time-Multiplexed FPGA Overlay with Linear Interconnect”, *in Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, Dresden, Germany, March 2018.

4.2 Architectural Enhancements

As discussed earlier, the II is one of the critical metrics to determine the throughput of an accelerator. The overlay presented in Chapter 3 has a relatively large II, especially for DFGs with a large number of inputs and operation nodes in the first scheduling stage. To reduce this II and to generally improve the performance of the overlay FU, a number of architectural enhancements are proposed. These will be discussed in the following subsections.

4.2.1 Rotating Register File

The most obvious way to reduce the II of Equation 3.1 is to overlap the loading of input data with instruction execution. Instead of adding additional complexity into the FU to support double-buffering, a rotating register file [97] is used to support the overlap of data written into the RF with subsequent instruction execution. As seen from Figure 4.4, the rotating register file is different from the normal RF in two aspects. First, the DC will never be reset so that the input data is written into successive locations of the RF in a rotating way, one data packet after another. Second, an offset counter (shown as the circuitry inside the dashed box) is added to update the read address index, eliminating the offset deviation caused by this rotating mode. The offset counter is triggered by the *control* signal, delayed by two clock cycles for synchronization with the instructions (*inst_fetch*) read from the IM. When the instruction fetch is finished, the *offset* value will be added with the original addresses of the RF to ensure that the corresponding data are used as operands. Besides the offset counter, the RAM32M-based RF needs to be adapted to support the situation when reads and writes happen simultaneously. The original design presented in Chapter 3 [96] used a RAM32M primitive with a dual port configuration (1 read, 1 read/write), whereas the rotating RF version requires a quad port configuration to support 2 reads and 1 write. We also examined the internal pipeline of the FU and found that some registers (the lower registers on the left of the RF and the two output registers at the very right of the FU in Figure 3.3) can be removed without sacrificing the performance.

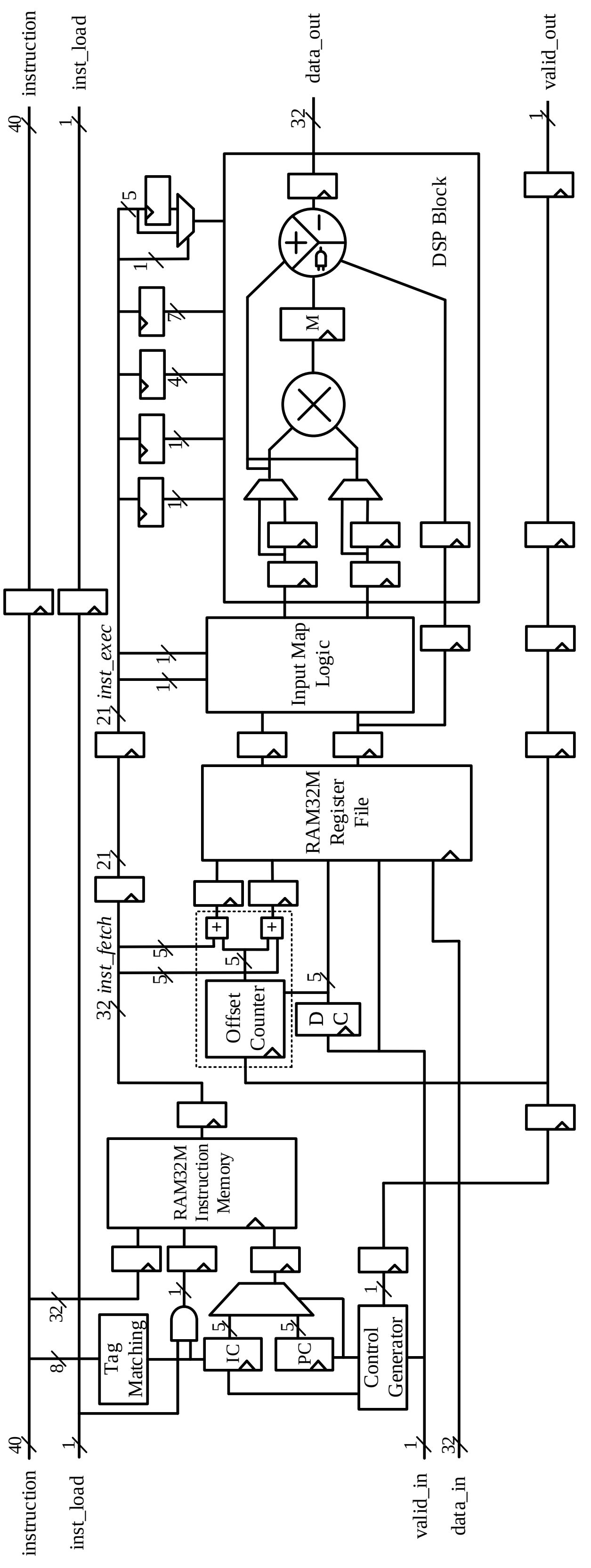


Figure 4.3: FU enhancement: V1.

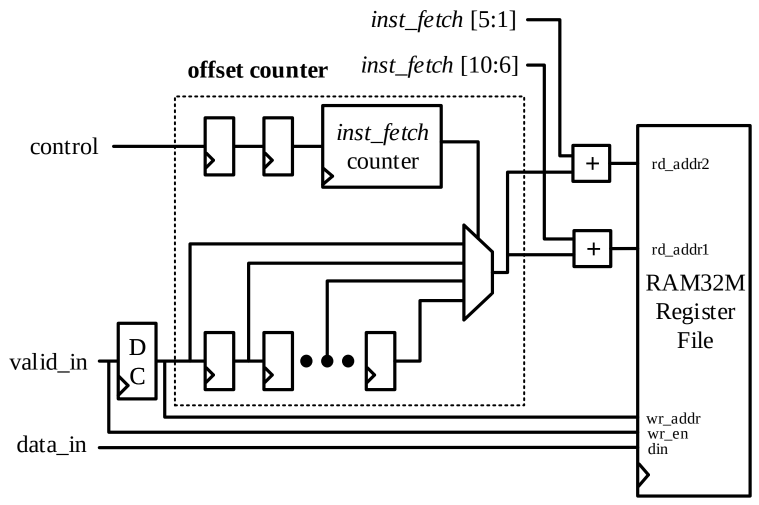


Figure 4.4: Rotating register file.

The new FU design, which we refer to as version 1 (V1), is shown in Figure 4.3. The V1 design requires 1 DSP block, 200 LUTs (25% more than [96]), 243 FFs (17% less than [96]) and has a frequency of 334 MHz (2.8% higher than [96]) on a Zynq XC7Z020 (610 MHz on a Virtex-7 VC707). An 8-FU V1 overlay is able to operate at 303 MHz on the Zynq-7020 device. The II of the V1 overlay is determined as:

$$II_{V1} = \max_{FU}\{\#load + 1, \#op + 2\} \tag{4.1}$$

where the extra cycle in the data load is required to separate data blocks. A complete timing analysis of the new V1 FU is presented later in the chapter (Section 4.3).

4.2.2 Replicating the Stream Datapath

The II can be further reduced, at the expense of an increased data bandwidth requirement, by increasing parallelism. Replicating the data processing part of the FU and increasing the data I/O to 64-bit can reduce the II by half, while the IM and other control circuitry are reused at runtime.

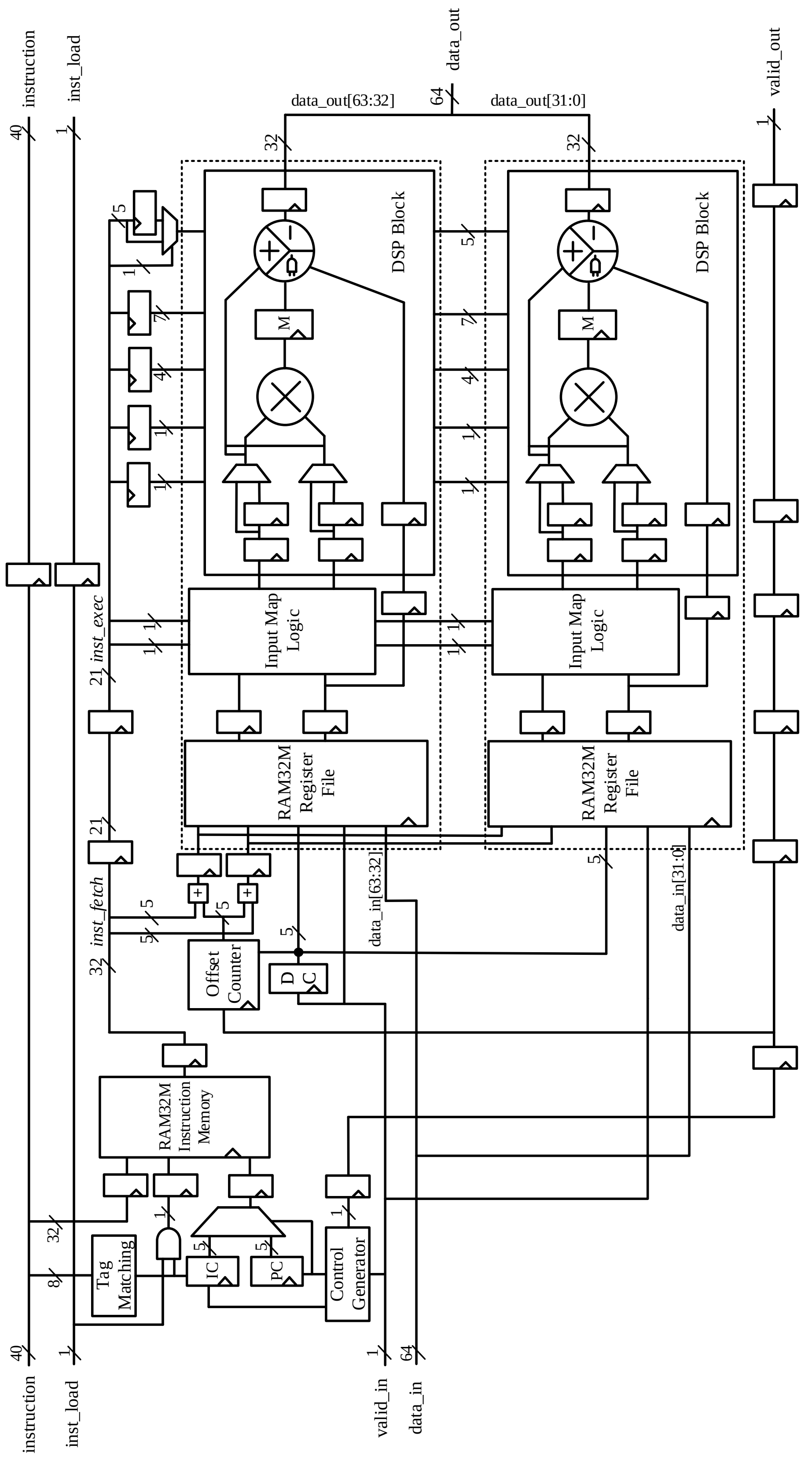


Figure 4.5: FU enhancement: V2.

This design, which we refer to as version 2 (V2), is based on the V1 design so as to take advantage of the rotating register file and the other enhancements in the V1 FU. In the V2 design, only the RF, the input map logic and the DSP block (shown within the dashed boxes) are replicated, as shown in Figure 4.5. The V2 design requires 2 DSP blocks, 292 LUTs and 333 FFs, and operates at a frequency of 335 MHz. This represents an FU resource consumption which has 46% more LUTs and 37% more FFs compared to the V1 design, with a frequency which is almost the same. The II of the V2 design is half that of the V1 design, as shown in Equation 4.2, resulting in a much higher compute efficiency. However, replicating the stream datapath to support SIMD style processing is not a general solution and is only applicable to a subset of all applications.

$$II_{V2} = \{\max_{FU}\{\#load + 1, \#op + 2\}\}/2 \tag{4.2}$$

4.2.3 FU Write-back

The main disadvantage with these overlays is that they are feed-forward only, and thus the overlay depth (the number of FUs) depends on the critical path of the DFG. However, if output data is able to be written back to the RF, multiple nodes on the DFG critical path could be combined within the same scheduling stage, thus reducing the overlay depth. Additionally, without write-back, when a new application kernel (with a different DFG depth) needs to execute, the overlay depth also needs to change. This requires an overlay reconfiguration between kernels which significantly impacts the kernel context switch time. Thus, a fixed architecture which is able to handle a range of more general kernels would improve execution time when multiple kernels need to be accelerated.

Table 4.1: New instruction format.

FU part	ALU Control								Input Map Control		RF Control		Reserved
Signals	*NDF*	*WB*	alumode	inmode	opmode	cea2	ceb2	usemult	split	immop	src-1	src-2	
No. of bits	1	1	4	2	7	1	1	1	1	1	5	5	2
Locations	[30]	[29]	[28:25]	[24:23]	[22:16]	15	14	13	12	11	[10:6]	[5:1]	[31][0]
ADD R3, R5 (*WB*)	0	1	0000	00	0110011	1	1	0	1	0	00011	00101	0 0
MUL R1, R0 (*WB*&*NDF*)	1	1	0000	11	0000101	0	0	1	0	0	00001	00000	0 0

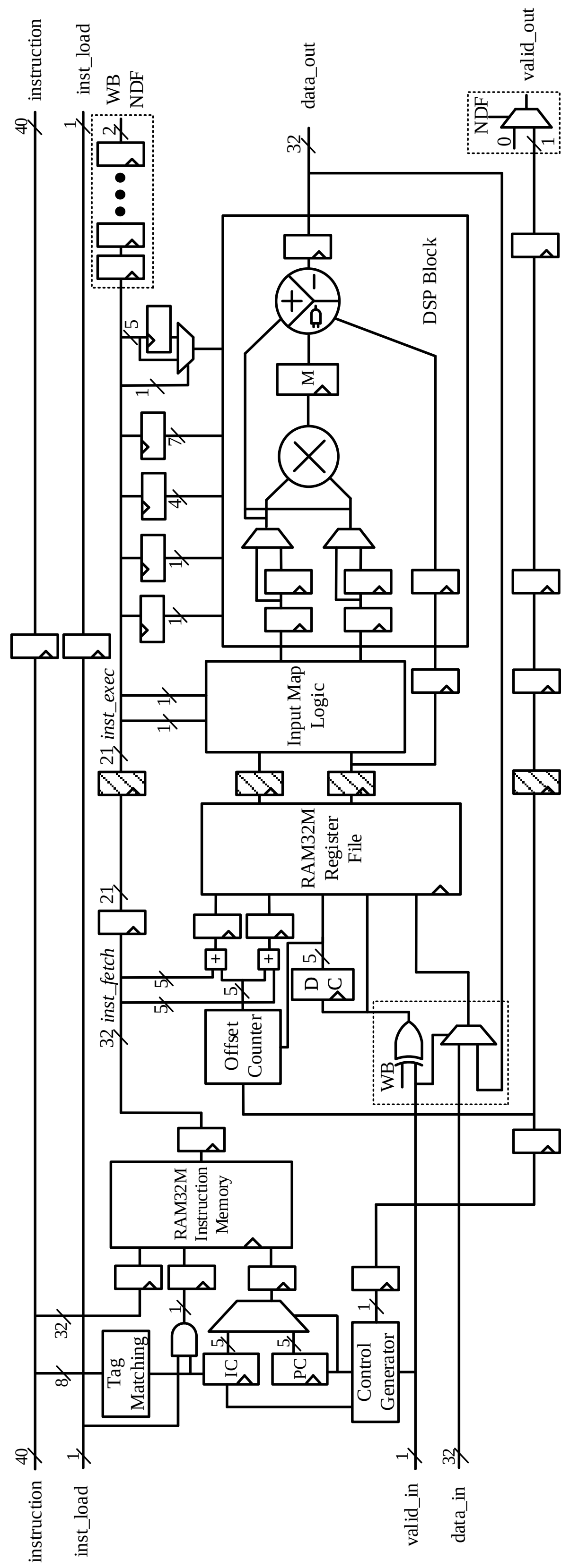


Figure 4.6: FU enhancement: V3-V5.

Introducing data write-back is relatively simple and involves feeding the *data_out* signal back into the FU and multiplexing it with the *data_in* signal, as shown in the lower left dashed box in Figure 4.6. This requires that the instruction format of [96] be modified with two extra bits added, a write-back (*WB*) bit and a no data forward (*NDF*) bit. Both bits are needed as there is a possibility that the output data will be both written back to the RF and bypassed to the next FU stage. Rather than adding two extra bits to the (already) 32-bit instruction, we note that the DSP primitive is only used to support operations with 2 or 3 operands (which means the D port is unused and can be disabled). This means that the three bits of the DSP *inmode* field can be hardwired, allowing the use of 1-bit as the *WB* flag, 1-bit as the *NDF* flag, with 1-bit reserved for future use. The *valid_in* signal and the delayed *WB* flag are then used to select between the two different data sources in the write-back logic. The new instruction format can be found in Table. 4.1.

4.2.4 Comparison of the New FU Architectures

The resource consumption and maximum frequency for the various FU designs, including the FU design of Chapter 3 (which we now refer to as version 0 (V0)), when implemented on a Zynq XC7Z020, are listed in Table 4.2. The V1 FU consumes around 25% more LUTs than that of V0, mainly due to the addition of the RAM32M primitive and the offset counter. However, there is an improvement in frequency and a reduction in the number of FFs due to the removal of the registers before the RF and after the DSP block. The resource consumption of V2 is less than twice that of V1, with a similar frequency to V1.

Table 4.2: Comparison of different FU designs.

	V0	V1	V2	V3	V4	V5
DSPs	1	1	2	1	1	1
LUTs	160	200	292	212	207	248
FFs	293	243	333	228	163	126
F_{max}	325	334	335	323	254	182
IWP	–	–	–	5	4	3

Table 4.2 also shows the resource utilization, operating frequency and internal write-back path (IWP) for three different implementations of the FU with write-back, referred to as version

3 (V3), version 4 (V4) and version 5 (V5). The V3 FU is identical to V1, except that the write-back logic is added, and includes all of the circuitry in Figure 4.6. The IWP for the V3 FU is five, consisting of one cycle in the RF, one at the register between the RF and the input map logic, and three in the DSP block. This overlay operates at a frequency close to that of the non-WB overlays. It should be noted that the IWP for DSP based implementations has a significant impact on instruction sequencing, due to data hazards, resulting in the need for NOP insertion to remove dependencies. Thus, other FU implementations with reduced IWP were also examined. To reduce the IWP, the registers between the RF RAM32M primitive and the input map logic (the four shaded registers in Figure 4.6) can be deleted, resulting in a frequency reduction caused by the reduced operating frequency of the RAM32M primitive. This FU, referred to as V4, is identical to V3 (except that the shaded registers in Figure 4.6 are removed) and has an IWP of 4 and a frequency of 254MHz. A further reduction in the IWP can be achieved by reducing the pipeline depth of the DSP block from three to two, resulting in an IWP of 3 and a frequency of 182MHz.

The V3-V5 FUs can then be implemented as a fixed depth overlay, as in Figure 3.2. We propose implementing two depth 8 overlays in a single tile. The two overlays in a tile could then either be connected in series (to form a single depth 16 overlay) or connected in parallel to produce a depth 8 overlay with dual datapaths, similar to the V2 based overlay.

As data forwarding within a DSP block is not possible, due to the inability to access the internal signals, it is important to understand the impact of a fixed depth overlay when write-back is used. When scheduling DFG nodes to the overlay, any dependency between nodes will require the insertion of NOPs, equal to IWP-1, unless other non-dependant nodes can be scheduled between the nodes with the dependency.

4.3 Compiling to the Overlay

Kernel Mapping: The HercuLeS HLS tool [95] is used to transform a ‘C’ description of the compute kernel to a DFG description, similar to the mapping tool as discussed in Chapter 3. Our mapping flow is described below using the previous ‘gradient’ example (shown in Figure 3.1).

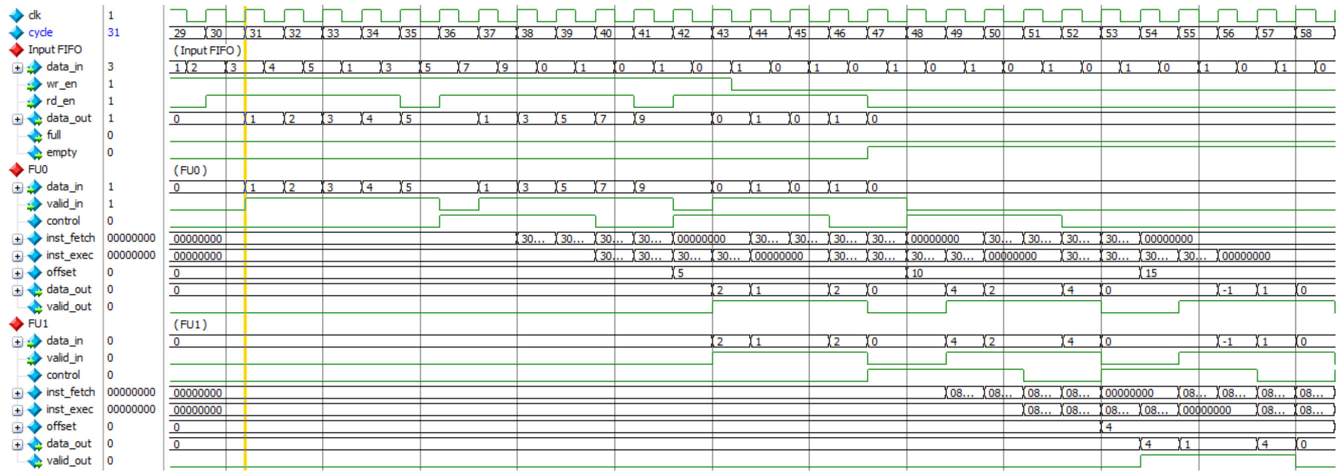
For the V1 and V2 based overlays, ASAP scheduling is used which results in no data dependencies between operations at the same scheduling stage, similar to that for the V0 based overlay [96], with nodes in each scheduling stage then being allocated to a single V1 or V2 FU for execution. The set of instructions from the sequenced DFG is identified, then the cycle-by-cycle execution pattern is formed, interleaving load/store and arithmetic/ALU operations, as shown in Table 4.3. As seen from this table, the FU utilization is improved significantly by overlapping the current instruction fetch with the data load for the next iteration. For the 'gradient' benchmark, the II is reduced from 11 for V0 [96] to 6 for V1 or 3 for V2 using the same ASAP scheduling. This translates to a throughput of 0.59 Giga-operations/s (GOPS) for the V1 based overlay with a latency of 142 ns (1.11 GOPS and 150.5 ns for V2), compared to 0.35 GOPS and a latency of 156 ns for the V0 based overlay. Lastly the 32-bit FU instructions are generated. This new instruction schedule for the 'gradient' benchmark (Table 4.3) is used with a 4 FU version of the V1 overlay in the simulation of Figure 4.7 to verify functionality.

The initialization and input data packets of the V1 overlay is identical to that of V0. The first five data elements are read from the input FIFO and fed into FU0 at clock cycle 31, corresponding to the five data loads from Table 4.3, as shown in Figure 4.7(a). The 2-cycle delay for the instruction fetch and another 2-cycle delay for the alignment of DSP block configuration are identical to those of V0. The *offset* value is updated with the accumulation of a new data packet when the last *inst_fetch* has been finished. This mechanism guarantees that current instruction fetch of FU0 can be overlapped with a new data packet transmission starting at cycle 31, indicating an II of 6. Since the two output registers at the very right of the FU have been removed, the output data of the FU can be obtained 3 cycles after the instruction is executed, reducing the latency by 1 cycle for each FU. Figure 4.7(b) shows the functionality of FU2, FU3 and the output FIFO. The output of FU3 is valid at clock cycle 74 and clock cycle 80 for the first input data package ("1, 2, 3, 4, 5") and the second input data package ("1, 3, 5, 7, 9"), respectively. Thus the latency from data input to FU0 to data output from FU3 is 43, which matches with the schedule of Table 4.3.

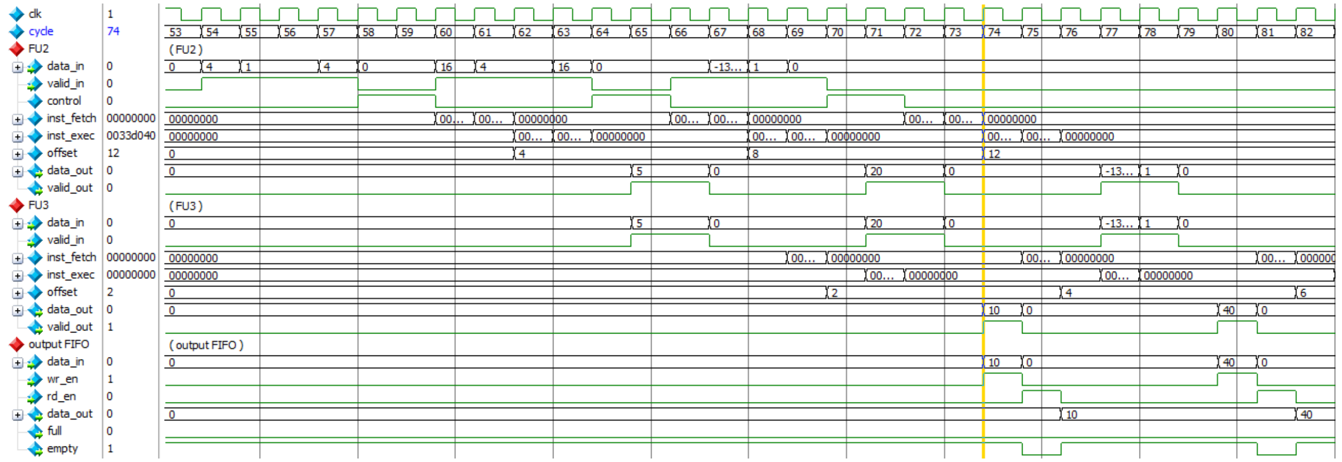
Table 4.3: First 44 cycles of the ‘gradient’ schedule (II=6).

cyc	FU0	FU1	FU2	FU3
1	Load R0[1]			
2	Load R1			
3	Load R2			
4	Load R3			
5	Load R4			
6				
7	Load R0			
8	Load R1; SUB (R2 R0)[2]			
9	Load R2; SUB (R2 R1)			
10	Load R3; SUB (R3 R2)			
11	Load R4; SUB (R4 R2)			
12				
13	Load R0	Load R0		
14	Load R1; SUB (R2 R0)	Load R1		
15	Load R2; SUB (R2 R1)	Load R2		
16	Load R3; SUB (R3 R2)	Load R3		
17	Load R4; SUB (R4 R2)			
18				
19	Load R0	Load R0; SQR (R0 R0)		
20	Load R1; SUB (R2 R0)	Load R1; SQR (R1 R1)		
21	Load R2; SUB (R2 R1)	Load R2; SQR (R2 R2)		
22	Load R3; SUB (R3 R2)	Load R3; SQR (R3 R3)		
23	Load R4; SUB (R4 R2)			
24			Load R0	
25	Load R0	Load R0; SQR (R0 R0)	Load R1	
26	Load R1; SUB (R2 R0)	Load R1; SQR (R1 R1)	Load R2	
27	Load R2; SUB (R2 R1)	Load R2; SQR (R2 R2)		
28	Load R3; SUB (R3 R2)	Load R3; SQR (R3 R3)		
29	Load R4; SUB (R4 R2)			
30			Load R0; ADD (R0 R1)	
31	Load R0	Load R0; SQR (R0 R0)	Load R1; ADD (R2 R3)	
32	Load R1; SUB (R2 R0)	Load R1; SQR (R1 R1)	Load R2	
33	Load R2; SUB (R2 R1)	Load R2; SQR (R2 R2)	Load R3	
34	Load R3; SUB (R3 R2)	Load R3; SQR (R3 R3)		
35	Load R4; SUB (R4 R2)			Load R0
36			Load R0; ADD (R0 R1)	Load R1
37	Load R0	Load R0; SQR (R0 R0)	Load R1; ADD (R2 R3)	
38	Load R1; SUB (R2 R0)	Load R1; SQR (R1 R1)	Load R2	
39	Load R2; SUB (R2 R1)	Load R2; SQR (R2 R2)	Load R3	ADD (R0 R1)
40	Load R3; SUB (R3 R2)	Load R3; SQR (R3 R3)		
41	Load R4; SUB (R4 R2)			Load R0
42			Load R0; ADD (R0 R1)	Load R1
43	Load R0	Load R0; SQR (R0 R0)	Load R1; ADD (R2 R3)	
44	Load R1; SUB (R2 R0)	Load R1; SQR (R1 R1)	Load R2	output

[1] The load operations are not real instructions from the IMs.
[2] The arithmetic instructions shown in this table are *inst_fetch* from Figure 4.7.



(a) Data from input FIFO to FU1



(b) Data from FU2 to output FIFO

Figure 4.7: Simulation result of kernel ‘gradient’.

Typically, most of the existing CGRA architectures adopt Modulo scheduling [98], or a derivative algorithm, to achieve a minimum II. However, Modulo scheduling is based on the assumption that each operation node is executed in 1 cycle and the transfer of data between two arbitrary FUs completes in 1 cycle, which is not realistic for highly pipelined architectures. Instead, for a fixed depth (V3-V5) overlay we use an iterative greedy scheduling strategy which groups DFG nodes at each scheduling step into clusters and then adds DFG nodes along the critical path from subsequent clusters, while balancing the II across all clusters. The final number of scheduling clusters must be less than or equal to the overlay depth. In order to minimize the II and latency as much as possible, instructions with feedback data (that is with write-back to the current FU) should be scheduled as soon as possible. Another prerequisite is that write-back and load operations cannot occur at the same time as only a single write into the RF is allowed. In that case, the instructions should be triggered as soon as enough operands are located in the RF, instead of waiting for the whole data packet to be stored. This requires an additional enable signal which tells when to generate a *control* signal to fetch instructions from the IM, and is achieved by using one of the reserved bits in the new instruction set (Table 4.1) to specify the starting point for the instruction fetch with a register variable generated by the instruction counter.

As an example of fixed depth overlay scheduling, consider a slightly more complicated benchmark, the 'qspline' benchmark of Figure 4.8, mapped to depth 4 (4 FU) V3-V5 overlays. The 'qspline' benchmark has a critical path of 8 and would require an 8 FU overlay if ASAP scheduling was used. The scheduling process targeting a 4 FU overlay produces the 4 instruction clusters shown in Figure 4.8 (within the red dashed regions). NOPs (equal to IWP-1) must be added between dependent instructions (dependent DFG nodes) unless other non-dependent instructions can be scheduled in between. For example, in the first (top) cluster, node 17 (MUL_4_N17) and node 15 (MUL_N15) have a dependency (indicated by the edge between the two nodes). The scheduling algorithm schedules node 17 first, followed by nodes 13, 25, 9, 20, and 12, before node 15 is scheduled. Hence, the dependency between nodes 17 and 15 is resolved and no NOPs need to be inserted. Similarly for the second cluster there are dependencies between nodes 14 and 11, nodes 26 and 27, and nodes 21 and 22. Scheduling as: nodes 14, 26, 21, 10, 16, 11, 27, 22, again resolves all dependencies, for all overlay versions. In cluster three, scheduling as: 28, 18, 24, 23, 8, 19, 30, resolves all dependencies for the V4 and V5 overlays, but not for the V3 overlay, which with an IWP of 5 requires 4 operations between dependent nodes. Hence, a single NOP must be added to the schedule between node 23 and node 19 for the V3 overlay. For the bottom cluster, graph balancing is performed enabling the addition of the outputs from node 23 and node 8 (in the 3rd scheduling stage) to be scheduled, followed by node 31. Lastly, IWP-1 NOPs are inserted before the final addition. An example schedule for a 4 FU version of the V3 overlay is shown in Table 4.4.

The consequences of a fixed depth overlay are an increase in the II with a corresponding reduction in the throughput, but with a significant reduction in both the resource utilization and the latency. For the ‘qspline’ benchmark, the 4 FU V3 overlay has an II of 17, with a throughput of 0.45 GOPS and a latency of 152 ns, while the V4 overlay has an II of 14, a throughput of 0.43 GOPS and a latency of 185 ns. The 4 FU V5 overlay has an II of 14, a throughput of 0.29 GOPS and a latency of 198 ns. In comparison, the 8 FU V1 overlay (with the minimum depth needed to implement the ‘qspline’ benchmark using ASAP scheduling) requires approximately twice the hardware resource and has an II of 11, a throughput of 0.69 GOPS and a latency of 340 ns. It should be noted, that in some cases, limiting the depth of the overlay and properly balancing the schedule between stages actually results in a reduction in the II, compared to the simple ASAP schedule used in the V1 overlay, as can be seen in Table 4.5. Because of the reduced operating frequency of the V5 FU (just 56% of the V3 version), the V5 overlay has a significantly reduced throughput compared to the V3 and V4 overlays and will not be considered further.

4.4 Experimental Evaluation

We compare the performance of our linear TM overlays using a set of compute kernels from [1, 2] (the DFG kernels in graphical format are shown in Appendix A), as shown in Table 4.5. The V1 (1 DSP, no WB), V2 (2 DSP, no WB), V3 (1 DSP, WB, IWP=5) and V4 (1 DSP, WB, IWP=4) overlays are compared to the V0 overlay from Chapter 3 [96]. The V1, V2 and V0 overlays have a depth equal to the critical path (*Depth*) of the DFG, and are configured on a kernel by kernel basis, while V3 and V4 both have a fixed depth of eight. All overlays are implemented on a Zynq XC7Z020.

Table 4.4: First 47 cycles of the ‘qspline’ schedule (II=17).

cyc	FU0	FU1	FU2	FU3	
1	Load R0[1]				
2	Load R1; Bypass R0[2]				
3	Load R2; Bypass R1				
4	Load R3; MUL (R2 4)*[3]				
5	Load R4; MUL (R3 R0)				
6	Load R5; MUL (R4 6)				
7	Load R6; MUL (R1 R0)	Load R0			
8	WB R7*; MUL (R5 4)	Load R1			
9	MUL(R6 R1)	NDF			
10	MUL(R7 R0)	Load R2; Bypass R0			
11		Load R3; Bypass R1			
12		Load R4; MUL(R2 R0)*			
13		Load R5; MUL(R4 R3)*			
14		Load R6; MUL(R5 R4)*			
15		Load R7; MUL(R6 R1)	Load R0		
16		WB R8*; MUL(R7 R0)	Load R1		
17		WB R9*; MUL(R8 R0)	NDF		
18	Load R0	WB R10*; MUL(R9 R4)	NDF		
19	Load R1; Bypass R0[2]	MUL(R10 R1)	NDF		
20	Load R2; Bypass R1		Load R2		
21	Load R3; MUL (R2 4)*[3]		Load R3; MUL (R2 R1)*		
22	Load R4; MUL (R3 R0)		Load R4; MUL (R3 R1)*		
23	Load R5; MUL (R4 6)		Load R5; MUL (R4 R0)*		
24	Load R6; MUL (R1 R0)	Load R0	Load R6; MUL (R6 R1)		
25	WB R7*; MUL (R5 4)	Load R1	WB R7*; NOP		
26	MUL(R6 R1)	NDF	WB R8*; MUL (R7 R1)	NDF	
27	MUL(R7 R0)	Load R2; Bypass R0	WB R9*; MUL (R8 R0)	NDF	
28		Load R3; Bypass R1	MUL (R9,R5)	NDF	
29		Load R4; MUL(R2 R0)*		Load R0	
30		Load R5; MUL(R4 R3)*		NDF	
31		Load R6; MUL(R5 R4)*		Load R1	
32		Load R7; MUL(R6 R1)	Load R0	Load R2; ADD (R1 R0)*	
33		WB R8*; MUL(R7 R0)	Load R1	Load R3; NOP	
34		WB R9*; MUL(R8 R0)	NDF	NOP	
35	Load R0	WB R10*; MUL(R9 R4)	NDF	NOP	
36	Load R1; Bypass R0[2]	MUL(R10 R1)	NDF	WB R4*; NOP	
37	Load R2; Bypass R1		Load R2	ADD (R4 R2)*	
38	Load R3; MUL (R2 4)*[3]		Load R3; MUL (R2 R1)*	NOP	
39	Load R4; MUL (R3 R0)		Load R4; MUL (R3 R1)*	NOP	
40	Load R5; MUL (R4 6)		Load R5; MUL (R4 R0)*	NOP	
41	Load R6; MUL (R1 R0)	Load R0	Load R6; MUL (R6 R1)	WB R5*; NOP	
42	WB R7*; MUL (R5 4)	Load R1	WB R7*; NOP	ADD (R5 R3)	
43	MUL(R6 R1)	NDF	WB R8*; MUL (R7 R1)	NDF	
44	MUL(R7 R0)	Load R2; Bypass R0	WB R9*; MUL (R8 R0)	NDF	
45		Load R3; Bypass R1	MUL (R9,R5)	NDF	
46		Load R4; MUL(R2 R0)*		Load R0	
47		Load R5; MUL(R4 R3)*		NDF	output

[1] The load operations are not real instructions from the IMs.
[2] The arithmetic instructions shown in this table are *inst_fetch* from Figure 4.6.
[3] * represents a write-back instruction where the output will be sent back to the RF after IWP-1 cycles.

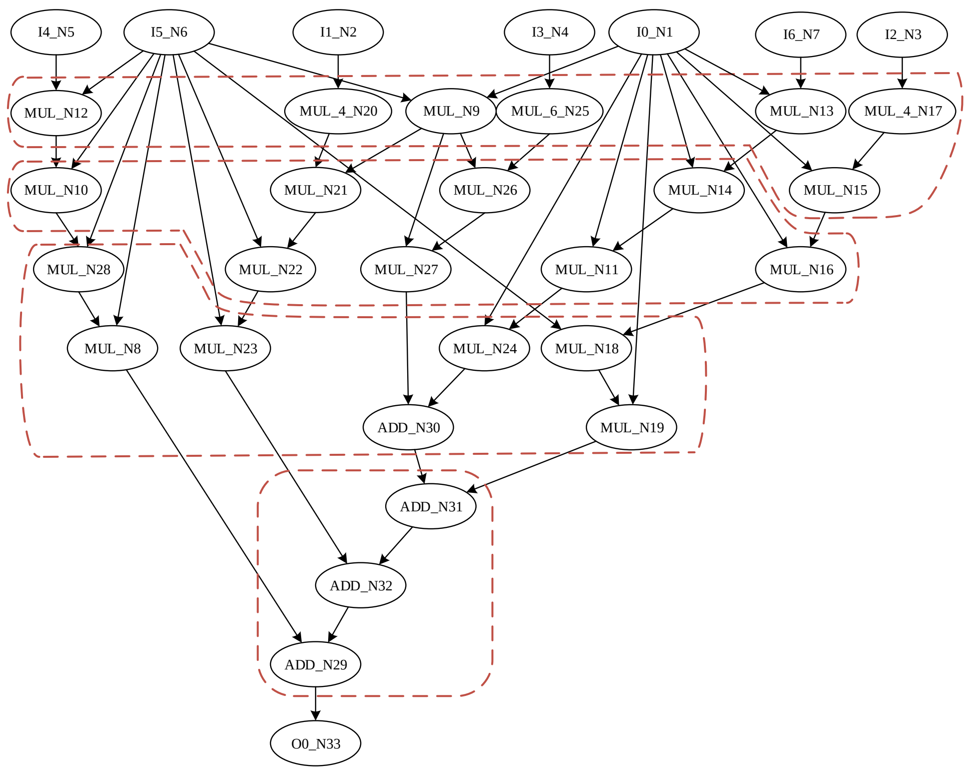


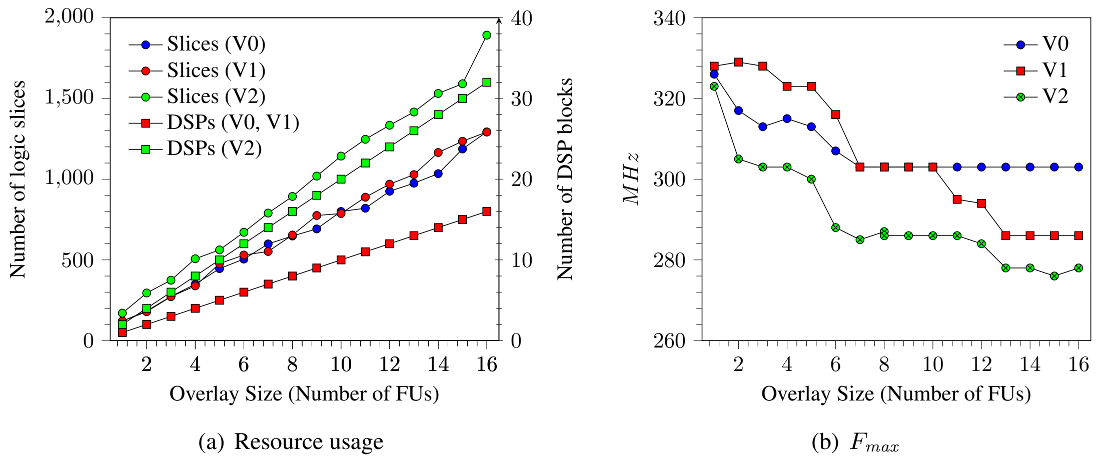
Figure 4.8: Data flow graph of the ‘qspline’ benchmark.

Table 4.5: DFG characteristics of benchmark set.

No.	Benchmark	I/O	#Ops	Depth	II_{V0}	II_{V1}	II_{V2}	II_{V3}[1]	II_{V4}[1]
1.	chebyshev	1/1	7	7	6	4	2	4	4
2.	mibench	3/1	13	6	14	8	4	8	8
3.	qspline	7/1	25	8	19	11	5.5	11	11
4.	sgfilter	2/1	18	9	13	8	4	8	8
5.	poly5	3/1	27	9	19	11	5.5	11	11
6.	poly6	3/1	44	11	25	14	7	13	12
7.	poly7	3/1	39	13	24	14	7	20	17
8.	poly8	3/1	32	11	21	12	6	16	14

[1] The II of the V3 and V4 overlays for the qspline benchmark reported here are different to that of Chapter 4.3, as here a depth 8 overlay is used, whereas in Chapter 4.3 the example used a depth 4 overlay.

The DSP, logic slice utilization and operating frequency for different depth V0, V1 and V2 overlays are shown in Figure 4.9. The V3 and V4 overlays are not included in this figure as they have a fixed depth (of 8). A depth 8 V1 overlay consumes 654 logic slices and 8 DSP slices which represents less than 5% of the logic and DSP resources on Zynq. The depth 8 V2 overlay consumes 893 logic slices and 16 DSP blocks or less than 8% of the Zynq resources. By comparison, the 8 FU V3 (V4) overlay consumes 814 (817) logic slices, 8 (8) DSP slices and operates at a frequency of 286MHz (233MHz).

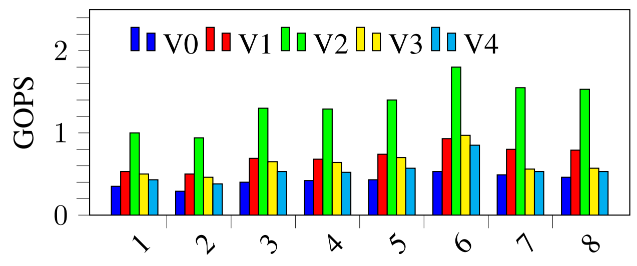


(a) Resource usage (b) F_{max}

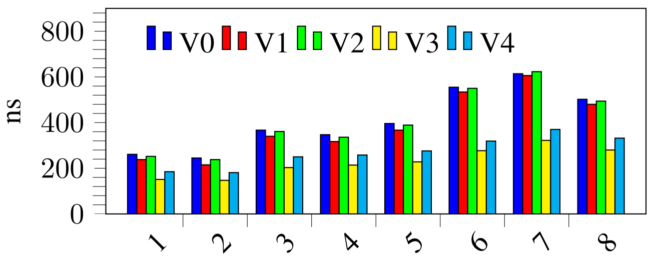
Figure 4.9: V1 and V2 overlay scalability on Zynq XC7Z020.

The DFG characteristics (number of I/O, number of arithmetic operations and graph depth) for the chosen benchmarks and the II achieved when mapped to the various overlays are shown in Table 4.5. For the first three benchmarks, which have a depth ≤ 8, ASAP scheduling is used to map to the V3 and V4 overlays, and thus, the II is the same as for the V1 overlay. The V1 (V2) overlay has an average 42% (71%) reduction in the II, while the V3 (V4) overlay has an average 34% (40%) reduction in the II for the depth > 8 benchmarks, compared to the V0 overlay.

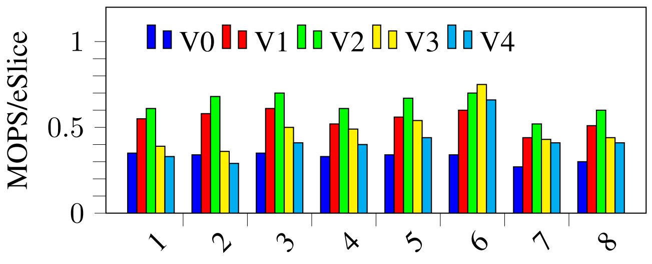
Figure 4.10 shows the throughput, latency and compute efficiency of the different overlays for the benchmarks given in Table 4.5. In terms of throughput, all overlays have a higher throughput than the V0 overlay. This is because interleaving data transfer with execution reduces the II and hence improves throughput. The two DSP V2 overlay has approximately twice the throughput as the V1 overlay, but also requires twice the data bandwidth. The size of both of these overlays is dependant on the depth (critical path) of the application kernel's DFG, and needs to be reconfigured when the application kernel changes. A depth 8 V1 (V2) overlay requires a minimum reconfigurable region of 7 (9) CLB tiles and 1 (2) DSP tile with a configuration time of 0.73 (1.02) ms using the processor configuration access port (PCAP). Additionally, the overlays require a further 0.29μs to load the configuration data for the largest benchmark.



(a) Throughput in GOPS



(b) Latency in ns



(c) Compute efficiency in MOPS/eSlice

Figure 4.10: Throughput, latency and compute efficiency for the benchmarks.

The single DSP V3 overlay has a throughput similar to the V1 overlay, with an average reduction of just 10%. The V4 overlay has a slightly reduced throughput as it operates at a lower frequency due to the removal of pipeline registers to reduce the IWP. The V3 and V4 overlays both have a fixed depth (in these experiments a depth of 8 is used). Adding write-back capabilities allows larger kernels to be mapped to a smaller number of FUs, removing the requirement that the overlay depth must be the same as the kernel critical path. This eliminates the need to reconfigure the overlay when the application kernel changes, making the overlay more general purpose (but requiring a different scheduling strategy). Thus, a hardware context switch on the V3 overlay requires just 0.25μs for the largest benchmark, representing a 2900$\times$ reduction compared to the V1 overlay.

The latency is heavily dependent on the depth of the overlay. For the V0, V1 and V2 overlays, the overlay depth is equal to the DFG depth due to the ASAP scheduling strategy used, and hence these overlays all have a larger latency. The V3 and V4 overlays generally show a significant reduction in the latency, particularly for larger depth DFGs, due to the fixed overlay depth.

As seen from Figure 4.10(c), all of the overlay designs proposed in this chapter (V1-V4) outperform the original V0 overlay in terms of compute efficiency (MOPS/eSlice). The V1, V2, V3 and V4 overlays achieve 66.7%, 93.7%, 48.5% and 27.3% better average compute efficiency, respectively, compared to the V0 overlay.

Of the two overlays with write back, the V3 overlay (with the higher IWP and the requirement for more NOP insertion) always outperforms the V4 overlay, across all metrics and all benchmarks. This is because the higher operating frequency negates the performance reduction due to additional NOP insertion.

4.5 Summary

We have presented an area efficient FPGA overlay with a linear connection of TM FUs based on the Xilinx DSP48E1. Interleaving data transfer with instruction execution significantly reduces the II with a resulting increase in throughput. Introducing write-back into the FU design and modifying the scheduling strategy allows the overlay depth to be fixed. This eliminates the need to reconfigure the overlay if the application kernel changes, making the overlay more general. These changes significantly reduce the latency with just a small decrease in throughput. The V3 overlay has the best performance, as for a fixed data bandwidth, it has comparable throughput with a significantly reduced latency.

Chapter 5

System Integration

5.1 Introduction

In Chapter 3 we proposed a linear TM overlay with potential for use as an area efficient FPGA accelerator. In Chapter 4, significant improvements to the architecture were described which further improved the overlay performance. However, to demonstrate the suitability of the overlay as an FPGA accelerator, it is important to develop a (memory) interface between the processor/memory subsystems and the overlay which is able to provide high-bandwidth (and large scale) data transmission. Developing an interface from scratch requires a significant amount of work, such as low level hardware design, device driver and software APIs, which impedes the fast development of a complete system design. An examination of existing FPGA solutions for memory interfaces, showed that most fall into one of two categories, AXI bus-based solutions for FPGA SoC systems (like in Xilinx Zynq) and PCIe-based solutions for stand-alone systems connected to the host CPU. Among the various frameworks, some of them abstract the interfaces with simple FIFO connections and provide streaming data transfers, which matches the requirements of our proposed linear TM overlay.

In this chapter, we examine two memory interfaces, namely Xillybus (which supports both AXI and PCIe-based systems) and RIFFA 2.2 (which supports just PCIe-based systems), and integrate the overlay subsystem into complete hardware accelerator systems. The performance of these hardware accelerators for a range of benchmarks is investigated and performance results are presented. The proposed overlay based accelerator is also compared to a state-of-art TM overlay, namely VectorBlox MXP, in terms of bandwidth and resource usage.

The main contributions can be summarized as follows:

- Examine memory interfaces for the hardware accelerators implemented on FPGAs, and provide a comprehensive analysis on two of the state-of-the-art integration frameworks, i.e. Xillybus and RIFFA.
- Present complete hardware accelerator systems by integrating the linear TM overlays with Xillybus and RIFFA interfaces, respectively, and present the performance of these hardware accelerators for a range of benchmarks.
- Make comparisons with a state-of-the-art TM overlay, namely VectorBlox MXP, which shows that the proposed overlay achieves approximately 50% of the throughput, but uses just half of the bandwidth and less than 20% hardware resource compared to MXP.

5.2 Available Solutions for Memory Interfaces

A number of emerging solutions have been developed for interfacing hardware accelerators with a processor in both the academic and industrial fields. The most common applications are AXI bus-based solutions [99–101] for FPGA SoC systems (like in Xilinx Zynq) and PCle-based solutions [5, 101–104] for stand-alone systems connected to a host CPU.

5.2.1 AXI bus-based Solutions

ZyCAP [99] was developed as an open source soft DMA controller for high performance partial reconfiguration on the Xilinx Zynq. It provides two interfaces, an AXI-Lite interface connected to general purpose (GP) port and another AXI4 interface connected to the high performance (HP) port, and achieves a maximum throughput of 382 MB/s between the external memory and the partial reconfigurable region (PRR).

Sadri *et al.* proposed an infrastructure to evaluate the energy consumption and data processing bandwidth between the ARM processor sub-system and the DRAM via the AXI HP/ACP interface [100]. The implementation achieved a maximum full-duplex data processing bandwidth of over 1.6 GB/s for a single HP/ACP port, running at 125 MHz on an XC7Z020-1C device.

Xillybus [101] is a commercial solution which provides a very simple abstraction of the AXI ACP interface for the Zynq device. It is consists of a loopback demo hardware design with a full Linux distribution running on the ARM processor. Xillybus can achieve a maximum data rate of around 300 MB/s running at 100 MHz on a Zynq-based ZedBoard.

5.2.2 PCIe-based Solutions

Northwest Logic [105] and Xillybus [101] are two representative commercial solutions which support PCIe interfaces for different generations while providing portability across different FPGA devices. Northwest Logic is closed source, with no evaluation kit available before a licensing payment. On the other hand, Xillybus is provided free of charge for research and teaching purposes. Users have full access to customize the complete Xillybus project, including the hardware design, the device driver and the software applications, with good documentation provided. To the best of our knowledge, the author of Xillybus has been shifting the focus from the AXI interface to the PCIe interface to support a higher bandwidth. The latest version of Xillybus (Revision XL) is able to achieve a maximum data rate of around 3.5 GB/s for Gen2x8 and Gen3x4 PCIe interfaces.

In addition to these commercial solutions, there are a number of academic PCIe-based solutions, such as DyRACT [102] and EPEE [103], which have been proposed for slightly different purposes. DyRACT mainly focuses on dynamic partial reconfiguration over the PCIe Gen2x4 interface, along with a configuration controller and clock management. EPEE was developed as a general purpose PCIe communication library, targeting a wide range of FPGA devices. However, there is no access to the EPEE project from the provided site.

ffLink [104] was proposed as the first open source solution for a PCIe Gen3x8 interface and achieved a throughput of 7.06 GB/s on a Xilinx VC709 platform. It was built based on the AXI infrastructure and heavily relied on the Xilinx CDMA engines, which hinders its widespread use beyond Xilinx devices.

JetStream [106] was presented as a novel PCIe Gen3 solution which provides both FPGA-to-Host and FPGA-to-FPGA communication. Apart from ffLink, JetStream is the only open-source solution which supports PCIe Gen3x8 and achieves a peak unidirectional bandwidth of 7.05 GB/s. However, JetStream was only tested for two specific Virtex-7 boards and it is not well documented for further development. Currently, the hardware project is unable to be used with a Vivado version above Vivado 2015.4 because of a compatibility issue.

RIFFA [5] is an open source reusable framework, supporting the integration of FPGA accelerators with workstations through both the second and third generation PCIe protocols. Similar to Xillybus, it aims at providing a very simple interface to user application logic via first word fall through (FWFT) FIFOs. A scatter-gather DMA-based design bridges the vendor-specific PCIe Endpoint core and multiple communication channels (up to 12) for the user defined IP cores. RIFFA supports a wide range of software APIs such as C/C++, Java, Python and MATLAB, and the hardware interface is easy to understand, with example timing diagrams provided. The latest version of RIFFA (version 2.2) achieves a unidirectional maximum bandwidth of 3.6 GB/s for the Gen2x8 configuration on the Xilinx VC707 platform and 3.5 GB/s for the Gen3x4 configuration on the Terasic DE5-Net, respectively.

Table 5.1 lists the theoretical bandwidth of the AXI and PCIe memory interfaces. Although the total theoretical bandwidth of an AXI interface is comparable to that of a PCIe interface, most of the AXI-based solutions are not able to make full use of all available high performance HP/ACP ports, while the PCIe-based counterparts can support up to 8 lanes and some of them achieve a bandwidth which is close to the theoretical maximum. Among the existing implementations, Xillybus and RIFFA appear to be the most promising solutions due to their availability, ease-of-use and portability. In subsequent sections we develop high performance overlay-based accelerators designs using Xillybus and RIFFA, mainly focused on PCIe-based solutions.

5.3 Overlay Integration

A block diagram of the proposed overlay accelerator system is shown in Figure 5.1. The memory subsystem provides a bridge between the overlay on the FPGA fabric and the ARM processor on a Zynq SoC via the AXI bus (or directly to the memory via the PCIe link). For the

Table 5.1: Theoretical bandwidth of typical memory interfaces.

Interface	Frequency	Upstream BW	Downstream BW	Ports/Lanes	Total BW[1]
AXI HP	150 MHz	1200 MB/s	1200 MB/s	4	9600 MB/s
AXI ACP	150 MHz	1200 MB/s	1200 MB/s	1	2400 MB/s
PCIe Gen2	250 MHz	500 MB/s	500 MB/s	8	8000 MB/s
PCIe Gen3	250 MHz	984 MB/s	984 MB/s	8	15800 MB/s

[1] In a streaming interface, the throughput is limited by either the upstream or downstream BW, depending on the number of inputs or outputs, and so the maximum theoretical throughput would be half of the total bandwidth reported here.

AXI bus-based overlay, two 32-bit FIFOs are used to connect a single 32-bit linear TM overlay with the memory subsystem. For the PCIe-based overlay, we propose replicating four 32-bit linear TM overlays to make full use of the 128-bit data bandwidth. The four overlay instances can implement multiple compute kernels at runtime and thus reduce the II to a quarter of that of a single overlay. Data transfers between the internal memory subsystem, the DDR SDRAM and the offchip DRAM are under the control of a scatter-gather DMA engine.

5.3.1 Xillybus

Xillybus is a portable, easy to use DMA-based data transfer solution which provides a simple abstraction of the AXI/PCIe interfaces. There is no prerequisite for knowledge of the AXI or PCIe protocols as all the low-level design is packaged into an IP core. One side of the Xillybus IP core is connected to the host processor, while the other side communicates with two standard FIFOs, providing six signals to control the data flow, as shown in Figure 5.2. The host processor can be either the ARM processor of the Xilinx Zynq SoC for the AXI bus-based solution (AXI-Xillybus) or a PC based x86 CPU for the PCIe-based solution (PCIe-Xillybus).

After the driver installation and a system reboot with the FPGA programmed by the Xillybus bitstream, several device files such as */dev/xillybus_write_32(128)* and */dev/xillybus_read_32(128)* are generated by the host driver like named pipes. These device files can be read from and written to like any file using basic Linux commands such as *write* and *read* on the Linux system. Xillybus also provides an IP core factory for users to customize multiple streaming device files and seekable device files which have access to the standard RAM of the FPGA.

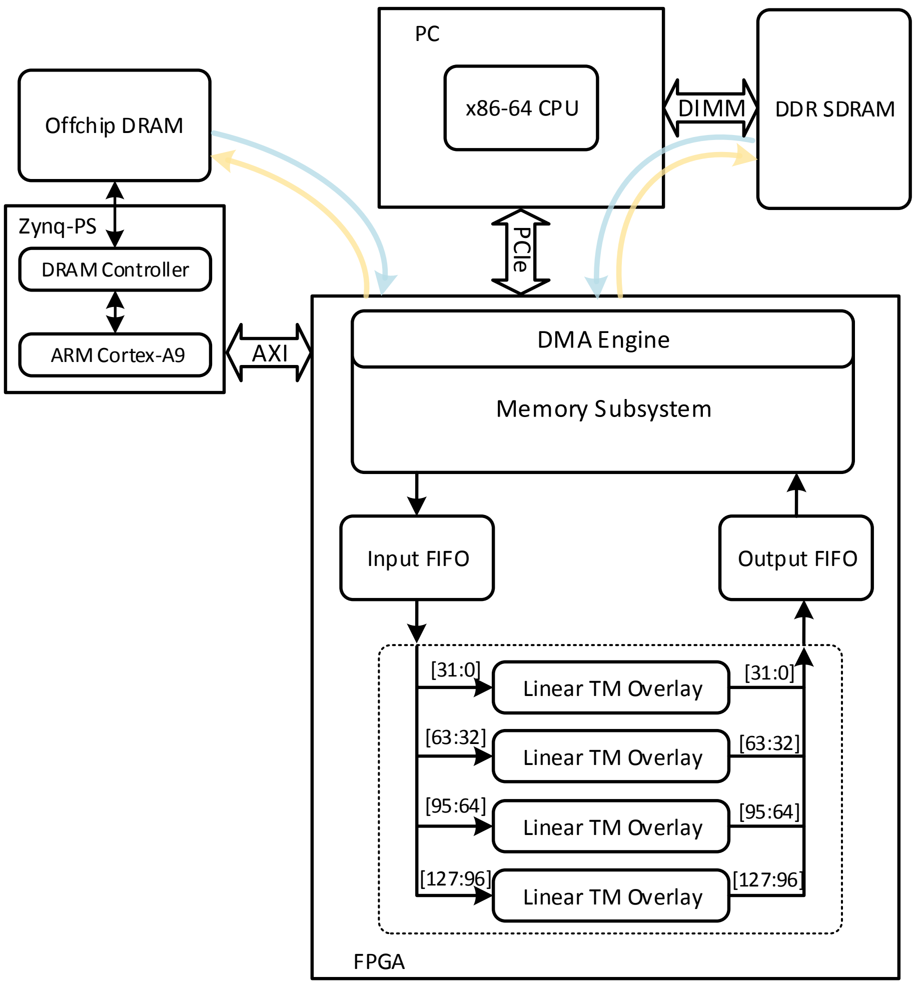


Figure 5.1: The proposed overlay accelerator system.

5.3.2 RIFFA

RIFFA has a relatively more complicated framework, as shown in Figure 5.3. It is mainly comprised of three layers. On the bottom layer, an RX engine and a TX engine are connected to the vendor-specific PCIe Endpoint IP core, which bridges the connection between RIFFA and the host CPU. The middle layer supports up to 12 communication channels per FPGA, and these channels send packets to the TX engine and receive packets from the RX engine through a scatter-gather DMA engine. The user IP core which connects to the RX/TX FIFOs from the communication channels, is located on the top layer.

Similar to Xillybus, an example design for data acquisition (*chnl_test.v*) is provided with

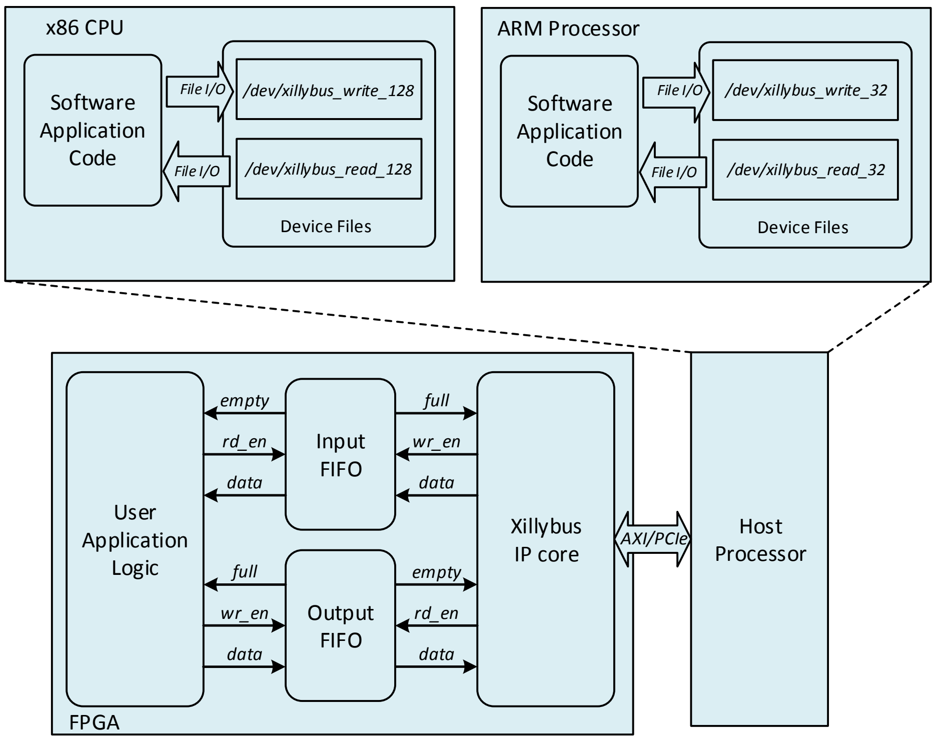


Figure 5.2: Xillybus framework.

driver and software code for testing (*testutil.c*). In this example, one sample channel is used to receive data from the host via the RX engine and send the same amount of data back to the host via the TX engine. For a more complicated design, multiple channels (up to 12 per FPGA) can be instantiated to integrate with different user cores independently.

5.4 Experimental Evaluations

We conducted experiments for both the AXI bus-based and PCIe-based overlay accelerators. For the AXI bus-based system, the experimental software runs on the ARM processor of the Xilinx ZedBoard. For the PCIe-based accelerators (implemented using both PCIe-Xillybus and RIFFA), the experiments are run on a Linux workstation (six 3.5 GHz Intel Xeon E5-1650 cores) with a Xilinx VC707 evaluation board plugged into the PCIe Gen2 slot of the motherboard.

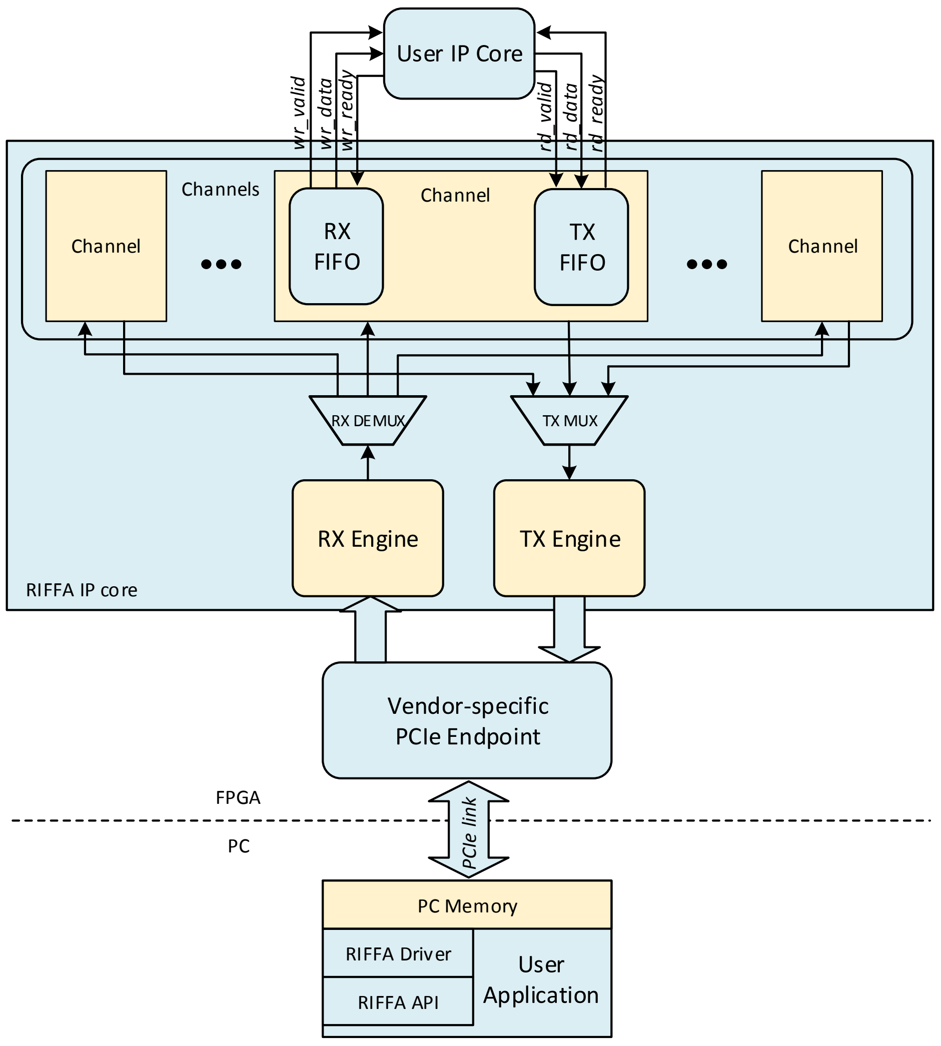


Figure 5.3: RIFFA framework.

5.4.1 AXI-Xillybus System

The AXI-Xillybus system is using one 32-bit wide AXI ACP port for data transmission, running at a frequency of 100 MHz. Hence the theoretical total bandwidth of this system is 800 MB/s, and the maximum streaming throughput is 400 MB/s in a full-duplex system.

5.4.1.1 AXI-Xillybus Loopback test

Before the accelerator system can be developed, we first ensure that the demonstration bundle provided by the Xillybus vendor operates correctly on our Zynq-based system. The demonstration bundle includes an example FPGA design containing just a single FIFO used to loop the data back, along with a device driver and sample software code. We evaluate the performance of Xillybus by testing the round trip data transmission throughput using the default settings. As shown in the code of Figure 5.4, the input (user_w_write) and output (user_r_read) pipes are connected to the same FIFO in a loopback fashion.

```
// 32-bit loopback
fifo_32x512 fifo_32
(
.clk(bus_clk),
.srst(!user_w_write_32_open && !user_r_read_32_open),
.din(user_w_write_32_data),
.wr_en(user_w_write_32_wren),
.rd_en(user_r_read_32_rden),
.dout(user_r_read_32_data),
.full(user_w_write_32_full),
.empty(user_r_read_32_empty)
)
```

Figure 5.4: AXI-Xillybus 32-bit loopback FIFO connection.

Figure 5.5 shows a simplified block diagram of the loopback structure. *Xillybus.v* provides the AXI bus-based Xillybus IP core connected to the ARM processor on a ZedBoard. And the top level *Xillydemo.v* wrapper file is used to integrate to the application logic. In this specific case, there is no application logic other than the single FIFO connected in loopback mode. After implementing the design and programming the bistream, the FPGA system is complete. We then use a simple high-level language program running on the host CPU (given in Figure 5.6) to measure the round-trip bandwidth.

In the host program, the write function and read function are executing sequentially, which means the read process has to wait until the write process has finished. It is inefficient for a full-duplex interface to work in half-duplex mode, and so as to make full use of the Xillybus IP core, we adopt a multi-threading technique (using Pthreads) to improve its performance by creating two different threads for the write and read functions respectively, as shown in the code of Figure 5.7.

5.4.1.2 AXI-Xillybus based Linear TM Overlay Accelerator

As previously discussed, Xillybus provides multiple streaming device files and seekable devices files which have access to the memory array. The accelerator and its data streams are shown

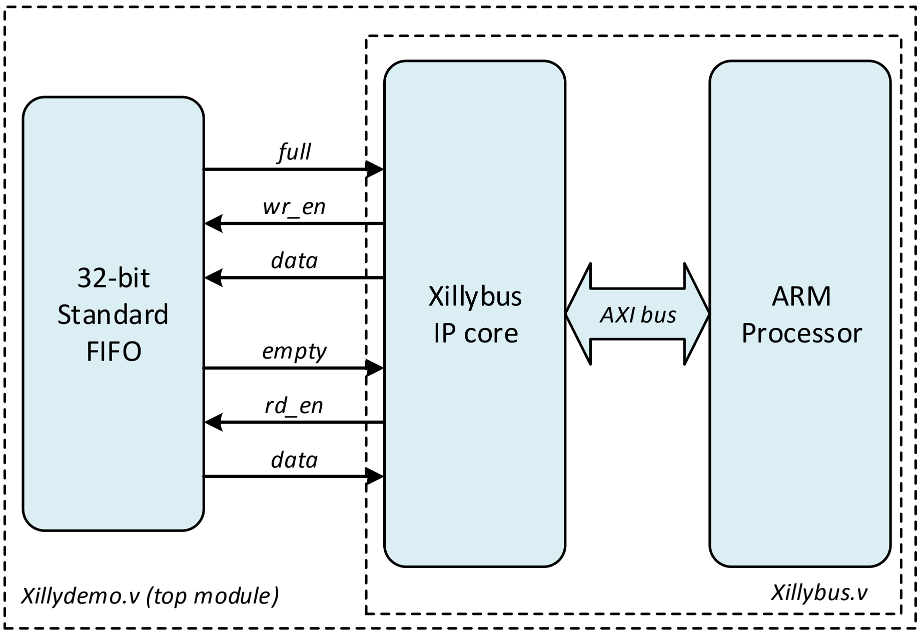


Figure 5.5: AXI-Xillybus loopback structure.

in Figure 5.8. In our design, two 32-bit streaming device files are instantiated for data acquisition and one 32-bit seekable device file is responsible for the instruction load and overlay setup.

5.4.1.3 Results and Comparisons

The bandwidth of the loopback test for AXI-Xillybus using different data block sizes is shown in Figure 5.9. As seen from this diagram, the single-threaded loopback test saturates at a throughput of around 150 MB/s (maximum of 160 MB/s) when the transfer size exceeds 8K words, while the multi-thread loopback test saturates (with a maximum throughput of 330 MB/s) when the transfer size exceeds 500K words. The two dashed lines indicate the theoretical throughput of the 32-bit AXI-ACP port running at 100 MHz. The throughput of the multi-thread test surpasses that of the single-thread test for data block sizes exceeding 64K words, and for large block sizes achieves approximately twice the throughput. Creating Pthreads introduces a significant overhead for smaller transfer sizes, thus the multi-threading version is only applicable for large data transfers.

We evaluate the performance of the proposed single DSP V3 overlay based accelerator (re-

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>

int fdr32 = 0;
int fdw32 = 0;
int N = 0;
int *array_input;
int *array_hardware;
struct timeval tv1, tv2;
int main(int argc, char *argv[]) {
fdr32 = open("/dev/xillybus_read_32", O_RDONLY);
fdw32 = open("/dev/xillybus_write_32", O_WRONLY);
N = atoi(argv[1]);
if (fdr32 < 0 || fdw32 < 0) {
perror("Failed to open devfiles");
exit(1);
}
//allocate memory
array_input = (int*)malloc(N*sizeof(int));
array_hardware = (int*)malloc(N*sizeof(int));
//initialize the inputs and outputs
for(int i = 0; i < N; i++){
array_input[i] = i;
array_hardware[i] = 0;
}
//Measure the excution time
gettimeofday(&tv1, NULL);
write(fdw32, array_input, sizeof(int)*N);
write(fdw32, NULL, 0); //an indicator at the end of writing data
read(fdr32, array_hardware, sizeof(int)*N);
gettimeofday(&tv2, NULL);
for(i = 0; i < N; i++){
if(array_input[i] != array_hardware[i]) {
  printf("recv[%d]: %d , expected %d \n\r", i, array_hardware[i], array_input[i
      ]);
  return;
  }
}
printf("round-trip bw: %f MB/s\n", N*4.0*2/1024/1024/((tv2.tv_sec-tv1.tv_sec)+(
    tv2.tv_usec-tv1.tv_usec)/1000000);
return;
}
```

Figure 5.6: Xillybus single-thread program in C.

ferred to as AXI-Xillybus-V3) and compare it to VectorBlox MXP [26] in terms of bandwidth and area consumption, using a set of kernels extracted from compute intensive applications from the literature [2, 3] (the DFG kernels in graphical format are shown in Appendix A), shown in Table 5.2. Both AXI-Xillybus-V3 and MXP are running on Xillinux-2.0, which is the latest Linux distribution released for ZedBoard (kernel 4.4), with AXI-Xillybus using the Pthread multi-threading technique with a data block transfer size of 1M words.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <pthread.h>

#define VP void*
#define NTH 2
int fdr32 = 0;
int fdw32 = 0;
int N = 0;
int *array_input;
int *array_hardware;
struct timeval tv1, tv2;
VP sample_1(VP arg) {
write(fdw32, array_input, sizeof(int)*N);
write(fdw32, NULL, 0);
pthread_exit(NULL);
}
VP sample_2(VP arg) {
read(fdr32, array_hardware, sizeof(int)*N);
pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
fdr32 = open("/dev/xillybus_read_32", O_RDONLY);
fdw32 = open("/dev/xillybus_write_32", O_WRONLY);
N = atoi(argv[1]);
if (fdr32 < 0 || fdw32 < 0) {
perror("Failed to open devfiles");
exit(1);
}
//allocate memory
array_input = (int*)malloc(N*sizeof(int));
array_hardware = (int*)malloc(N*sizeof(int));
//initialize the inputs and outputs
for(int i = 0; i < N; i++){
array_input[i] = i;
array_hardware[i] = 0;
}
// Creation of threads
pthread_t tid[NTH];
int value[NTH] = {1,2};
pthread_create(&tid[0], NULL, &sample_1, &value[0]);
pthread_create(&tid[1], NULL, &sample_2, &value[1]);
// Synch of threads in order to exit normally
gettimeofday(&tv1, NULL);
for(int loop = 0; loop < NTH; loop++) {
pthread_join(tid[loop], NULL);
}
gettimeofday(&tv2, NULL);
for(i=0; i<N; i++){
if(array_input[i] != array_hardware[i]) {
  printf("recv[%d]: %d , expected %d \n\r", i, array_hardware[i], array_input[i
      ]);
  return;
  }
}
printf("round-trip bw: %f MB/s\n", N*4.0*2/1024/1024/((tv2.tv_sec-tv1.tv_sec)+(
    tv2.tv_usec-tv1.tv_usec)/1000000);
return;
}
```

Figure 5.7: Xillybus multi-threading program in C.

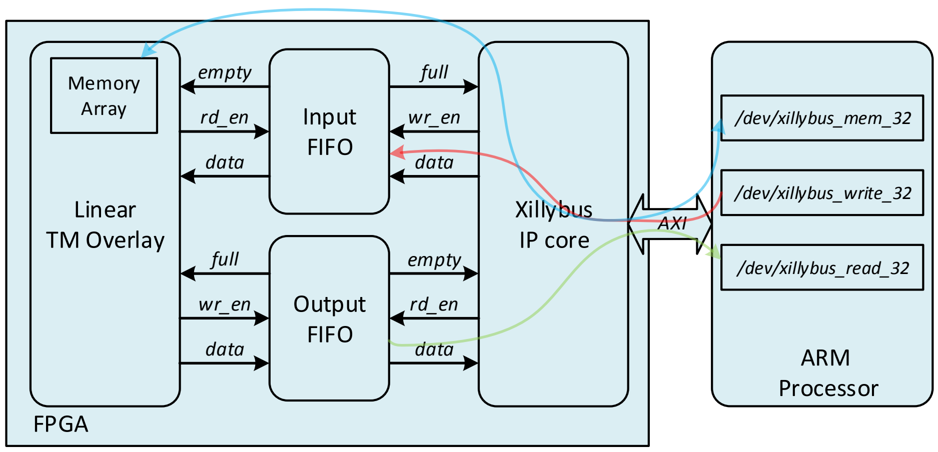


Figure 5.8: Linear overlay integrated with AXI-Xillybus.

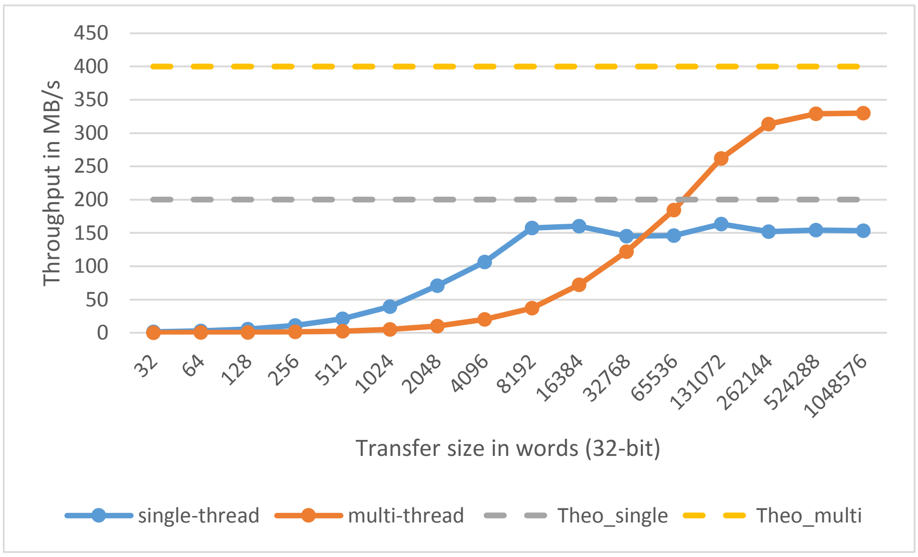


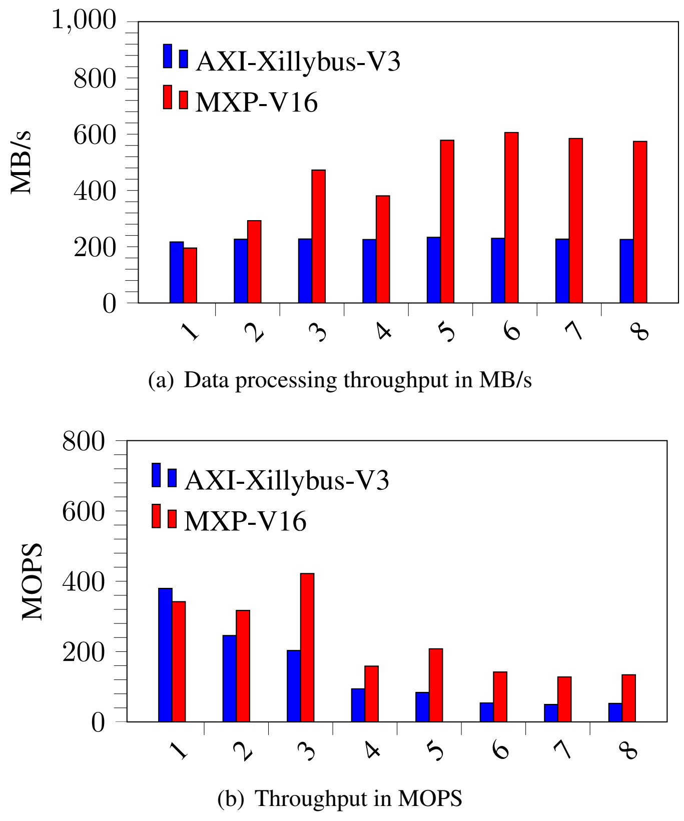
Figure 5.9: AXI-Xillybus loopback throughput.

Figure 5.10 shows the performance of a single 32-bit linear TM overlay (AXI-Xillybus-V3) and a 16-lane 32-bit MXP (MXP-V16) implemented on a ZedBoard for the benchmarks given in Table 5.2. As seen from the figure, MXP-V16 outperforms AXI-Xillybus-V3 over all benchmarks (with the exception of ‘chebyshev’). Table 5.3 shows the hardware resource usage of AXI-Xillybus-V3 and MXP-V16. As seen from this table, AXI-Xillybus-V3 is much more area efficient, and consumes only 19.8% LUTs, 26.1% FFs, 7.6% BRAMs and 7.1% DSP blocks,

Table 5.2: DFG characteristics of benchmark set.

No.	Benchmark Name	I/O nodes	graph edges	op nodes	graph depth	graph width
1.	chebyshev	1/1	12	7	7	1
2.	mibench	3/1	22	13	6	3
3.	qspline	7/1	50	25	8	6
4.	fft	6/4	24	10	3	4
5.	kmeans	16/1	39	23	5	8
6.	mm	16/1	31	15	4	8
7.	spmv	16/2	30	14	4	8
8.	stencil	15/2	30	14	3	6

compared to MXP-V16.



(a) Data processing throughput in MB/s

(b) Throughput in MOPS

Figure 5.10: Performance comparison for the benchmarks using a transfer size of 1M words.

The MXP memory system is based on one 64-bit AXI HP port running at a frequency of 110 MHz while AXI-Xillybus is based on a single 32-bit AXI ACP port running at a frequency of 100

MHz. This corresponds to a theoretical maximum throughput of 880 MB/s for MXP-V16 and 400 MB/s for AXI-Xillybus-V3. As seen from Figure 5.10, MXP-V16 achieves a throughput of 460.6 MB/s (52.3% of theoretical maximum) on average while AXI-Xillybus has a throughput of 226.6 MB/s (56.6% of theoretical maximum) on average. The throughput of AXI-Xillybus is almost half of that of MXP-V16, mainly due to AXI-Xillybus using only 32-bits of the available 64-bit ACP port while MXP-V16 is making full use of the 64-bit HP port.

Table 5.3: Area overhead of AXI-Xillybus-V3 and MXP.

System	Resource Consumption			
	LUTs	FFs	BRAMs	DSPs
AXI-Xillybus-V3	5,747	5,798	5	8
MXP-V16	28,974	22,174	66	112
Available	**53,200**	**106,400**	**140**	**220**

Although the AXI bus-based Xillybus provides an area efficient interface solution for the Xilinx Zynq SoC, the throughput is still far below the theoretical maximum shown in Table 5.1. This is mainly due to not making good use of the available HP/ACP ports (Xillybus uses only one 32-bit port which is operating at a frequency of 66.7% of the theoretical maximum that the AXI bus on Zynq can support).

5.4.2 PCIe System

Our PCIe system is based on the 8-lane Gen2 PCIe interface running at a frequency of 250 MHz, which corresponds to a maximum bandwidth of 8000 MB/s and hence a maximum streaming throughput of 4000 MB/s for a full-duplex system.

5.4.2.1 PCIe-Xillybus Loopback test

Xillybus has been upgraded to a new version (PCIe-Xillybus) to support high performance PCIe interface communication since 2015. The block diagram of PCIe-Xillybus for data loopback is almost the same as Figure 5.5, except that the 32-bit FIFO is replaced with a 128-bit FIFO and module *Xillybus.v* (including the Xillybus IP core and a Xilinx 7 series integrated block for PCIe) is connected to an HP Z420 workstation via the PCIe link, as shown in Figure 5.11. Similar to AXI-Xillybus, we developed two host programs to evaluate the round-trip bandwidth in both half-duplex and full-duplex mode.

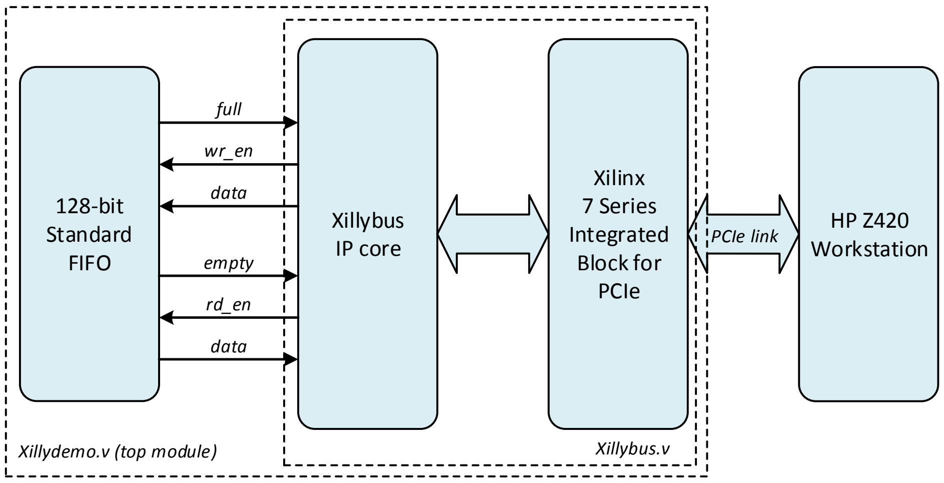


Figure 5.11: PCIe-Xillybus loopback structure.

5.4.2.2 PCIe-Xillybus based Linear TM Overlay Accelerator

Figure 5.12 shows the PCIe-Xillybus based overlay accelerator and its data streams. In this design, two 128-bit streaming device files are instantiated for data acquisition and one 128-bit seekable device file is responsible for instruction load and overlay setup.

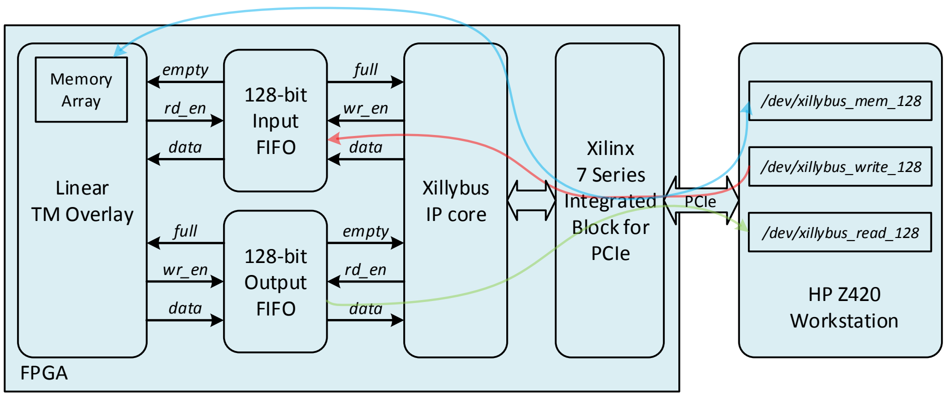


Figure 5.12: Linear overlay integrated with PCIe-Xillybus.

5.4.2.3 RIFFA Loopback test

Similarly, a loopback design using RIFFA was also implemented, similar to that of Xillybus, as shown in Figure 5.13. A first word fall through (FWFT) FIFO is used for data loop back, and the RIFFA IP core is acting as the bridge between the vendor specific PCIe Endpoint and the FWFT FIFO. Data transmission is under the control of a finite state machine (FSM), as shown in Figure 5.14. Since the upstream transfers and downstream transfers are functioning in different processing states, it is not possible to achieve full-duplex data transmission.

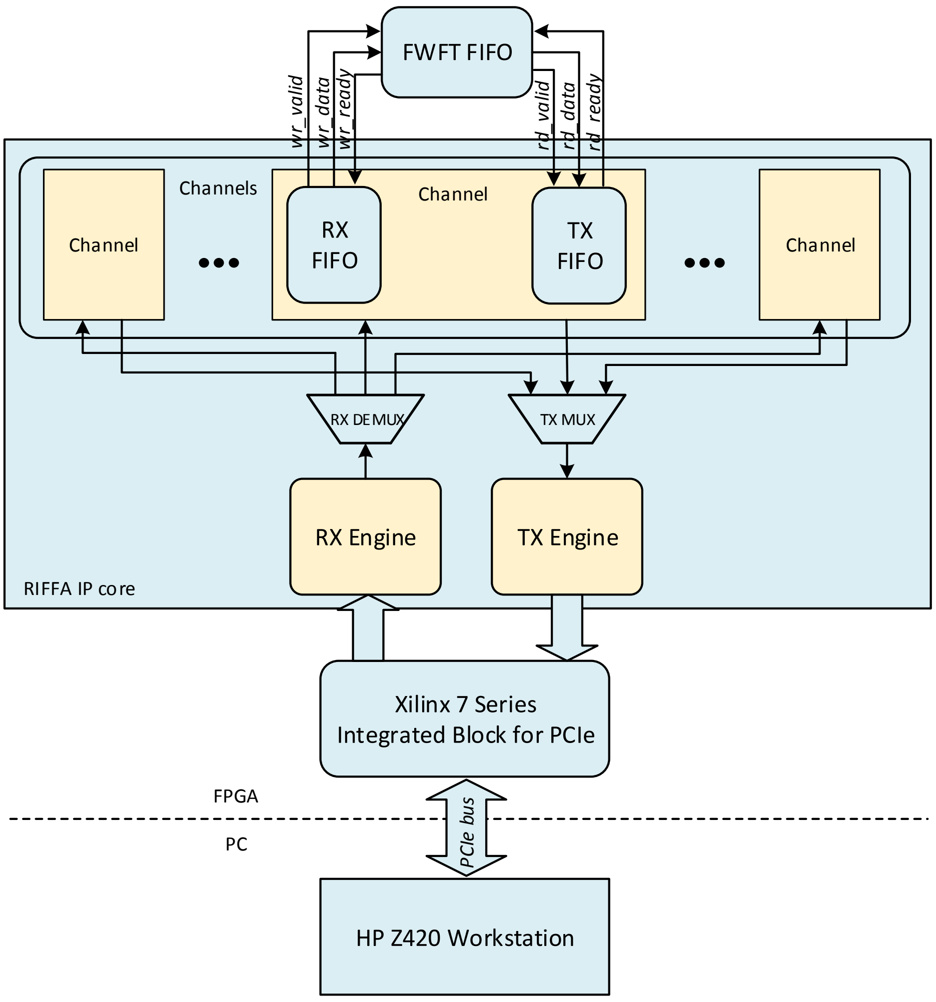


Figure 5.13: RIFFA loopback block diagram.

Unlike Xillybus, RIFFA has been developed along with its own software APIs, making software development relatively easy. Table 5.4 presents a detailed description of the major software

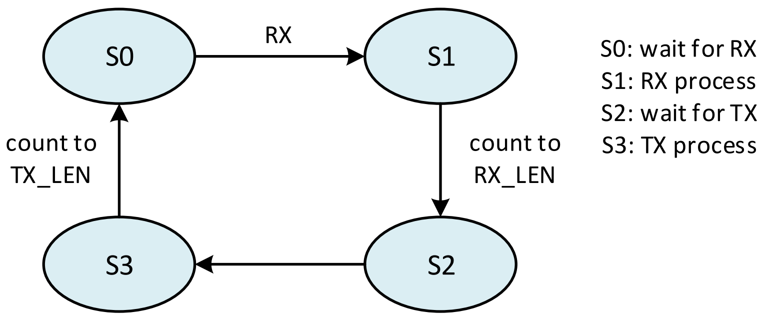


Figure 5.14: RIFFA FSM control for RX/TX.

APIs in C/C++. A simple Linux host program for bandwidth evaluation is shown in Figure 5.15. As mentioned previously, the RIFFA hardware was not developed for full-duplex data transmission. Hence the *fpga_send* function and *fpga_recv* function will execute sequentially and it is useless to use multi-threading techniques as was done for Xillybus.

Table 5.4: RIFFA software APIs in C/C++. [5]

Fuction Name and Description
*fpga_t *fpga_open (int id)* Initialize the FPGA with a specific ID and return a pointer to the *fpga_t* structure.
*void fpga_close (fpga_t *fpga)* Refresh the memory and resources for the specified FPGA.
*int fpga_send (fpga_t *fpga, int chnl, void *data, int len, int offset, int last, long timeout)* Send *len* 32-bit words from host CPU to FPGA channel *chnl*. The parameters *len*, *offset*, and *last* are sent to the FPGA channel. *timeout* defines an expiration time for the transfer.
*int fpga_recv (fpga_t *fpga, int chnl, void *data, int len, long timeout)* Receive a block transfer size of *len* 32-bit words from the FPGA channel *chnl* to the data buffer. The FPGA specifies an offset starting address to restore received data. *timeout* has the same meaning as before.

5.4.2.4 RIFFA based Linear TM Overlay Accelerator

Integration of the linear TM overlay with RIFFA is quite similar to that of Xillybus, except that the standard FIFOs are replaced with FWFT FIFOs. Four separate 32-bit depth-8 V3 overlays are connected (in parallel) with the 128-bit FWFT FIFOs to make full use of the data width. Although RIFFA has no access to the FPGA RAMs directly, it is possible to load the instructions and registers by customizing the FSM control, as shown in Figure 5.17. Compared with the original FSM, a new state for overlay initialization is added. The RX (view from the integrated

```c
#include <stdlib.h>
#include <stdio.h>
#include "timer.h"
#include "riffa.h"
#define NUM_TESTS 100

int main(int argc, char** argv) {
fpga_t * fpga;
int id;
int chnl;
size_t numWords;
int sent;
int recvd;
unsigned int * sendBuffer;
unsigned int * recvBuffer;
int err;
id = atoi(argv[2]);
chnl = atoi(argv[3]);
numWords = atoi(argv[4]);
GET_TIME_INIT(3);
// Get the device with id
fpga = fpga_open(id);
// Malloc the arrays
sendBuffer = (unsigned int *)malloc(numWords*4);
if (sendBuffer == NULL) {
  printf("Could not malloc memory for sendBuffer\n");
  fpga_close(fpga);
  return -1;
  }
recvBuffer = (unsigned int *)malloc(numWords*4 + 4);
recvBuffer +=1;
for (int j = 0; j < NUM_TESTS + 1; ++j) {
// Initialize the data
  for (int i = 0; i < numWords; i++) {
    sendBuffer[i] = i+1;
    recvBuffer[i] = 0;
    }
  GET_TIME_VAL(0);
  // Send the data
  sent = fpga_send(fpga, chnl, sendBuffer, numWords, 0, 1, 25000);
  printf("Test %d: words sent: %d\n", j, sent);
  GET_TIME_VAL(1);
  if (sent != 0) {
  // Recv the data
  recvd = fpga_recv(fpga, chnl, recvBuffer, numWords, 25000);
  printf("Test %d: words recv: %d\n", j, recvd);
  }
  GET_TIME_VAL(2);
  // Check the data
  for (i = 4; i < recvd; i++) {
  if (recvBuffer[i] != sendBuffer[i]) {
  printf("recvBuffer[%d]: %d, expected %d\n", i, recvBuffer[i], sendBuffer[i]);
  return;
  }
  }
  if (j > 0) {
  printf("send bw: %f\n", sent*4.0/1024/1024/((TIME_VAL_TO_MS(1) -
      TIME_VAL_TO_MS(0))/1000.0));
  printf("recv bw: %f\n", recvd*4.0/1024/1024/((TIME_VAL_TO_MS(2) -
      TIME_VAL_TO_MS(1))/1000.0));
  }
  }
// Done with device
fpga_close(fpga);
}
```

Figure 5.15: RIFFA host program in C.

overlay side) is divided into two processes: one for overlay initialization, and the other one for data transmission.

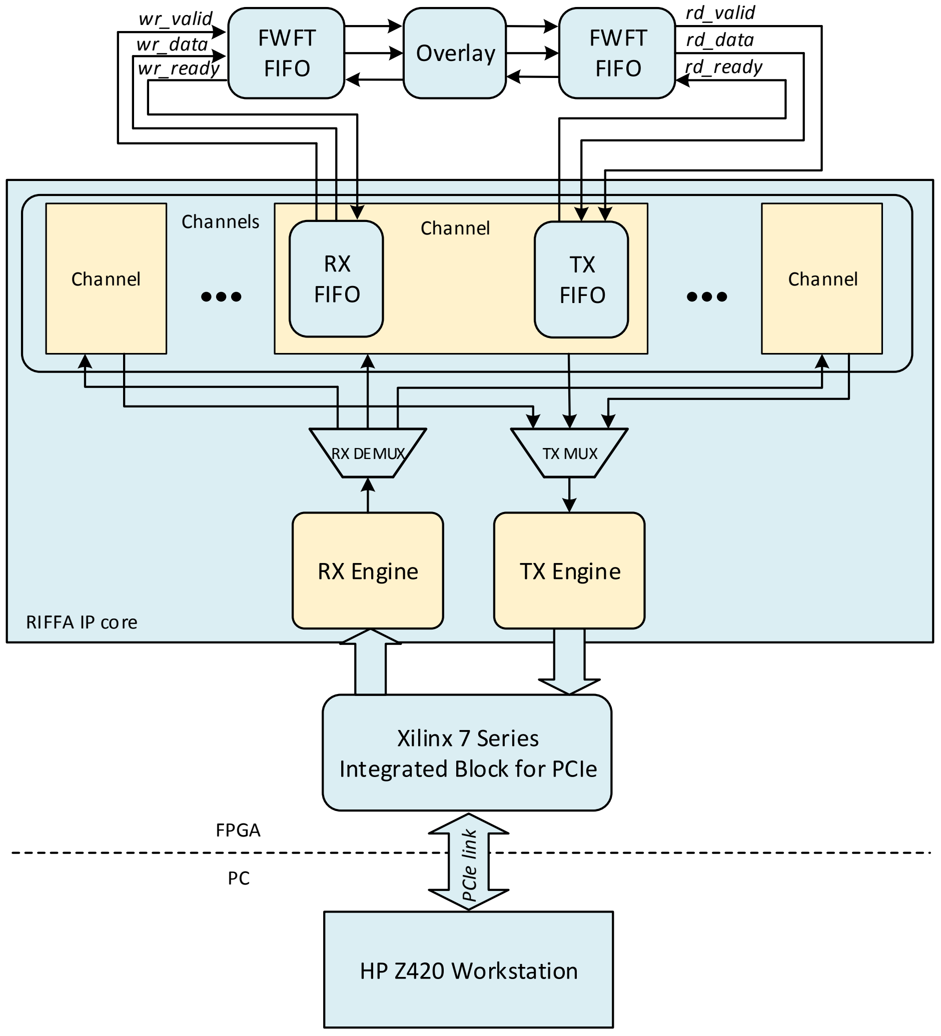


Figure 5.16: Linear overlay integrated with RIFFA.

5.4.2.5 Results and Comparisons

The bandwidth of the loopback test for PCIe-Xillybus using different data block sizes is shown in Figure 5.18. As seen from this diagram, the data transmission is very slow when the transfer size is less than 32K words. The single-threaded loopback test saturates at a throughput of 1900 MB/s when the transfer size exceeds 128K words, while the multi-thread loopback test saturates (with a maximum throughput of 2300 MB/s) when the transfer size exceeds 500K

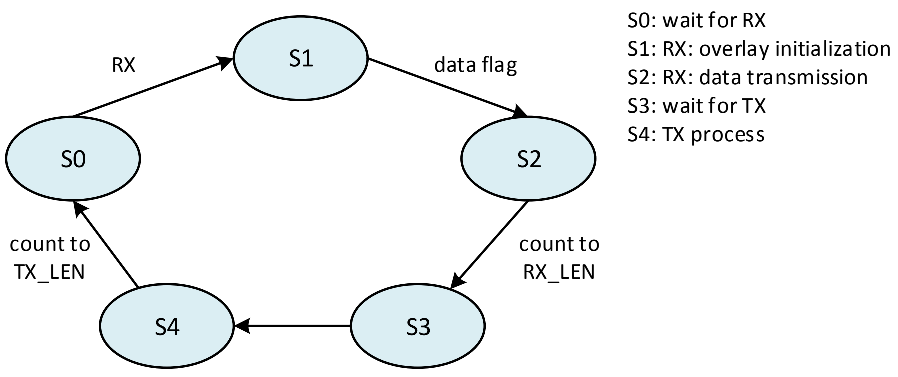


Figure 5.17: FSM control for RIFFA based overlay system.

words. The two dashed lines indicate the theoretical bandwidth of the Gen2x8 PCIe interface running at 250 MHz. The throughput of the multi-thread test surpasses that of the single-thread test for data block sizes exceeding 256K words, and achieves around 1.2× higher bandwidth when the transferring 1M words. Although the throughput of the multi-thread test is not as expected (twice that of the single-thread test), it shows better performance for the transmission of large data blocks.

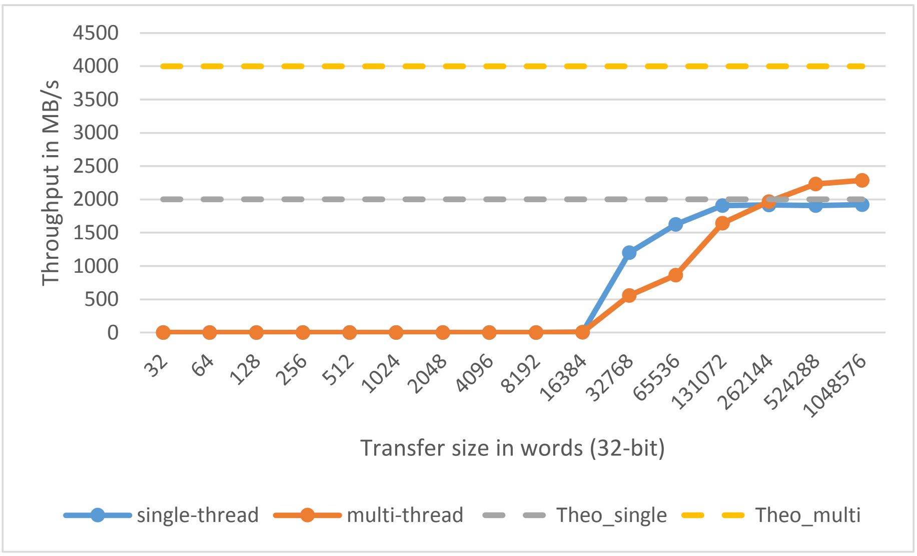


Figure 5.18: PCIe-Xillybus loopback throughput.

The bandwidth for RIFFA data loopback is shown in Figure 5.19. As seen from this diagram, both the upstream transfers (read) and downstream transfers (write) saturate at around 2.85 GB/s when the transfer size is equal to 128K sample words, corresponding a total bandwidth of 2.85 GB/s, as read and write cannot happen simultaneously. The bandwidth degrades slightly when the transfer size exceeds 128k words due to the overutilization of BRAMs.

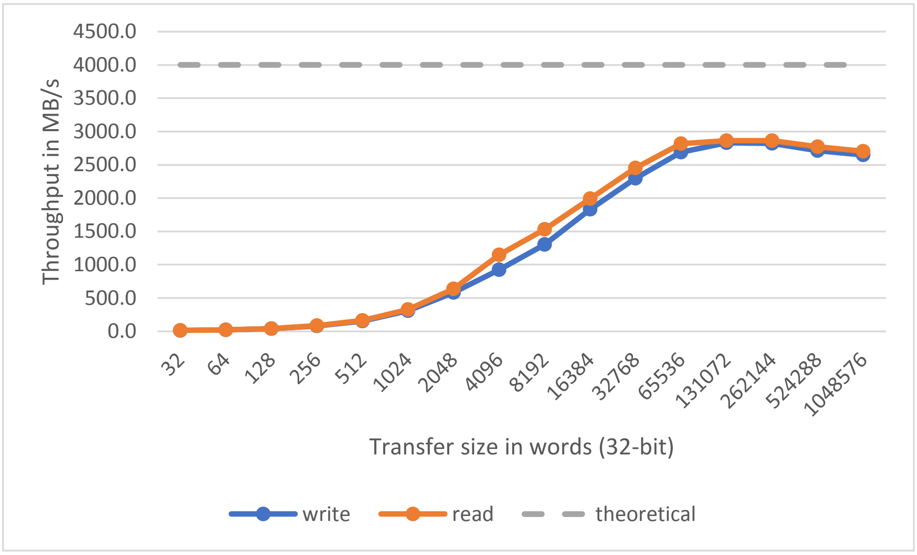
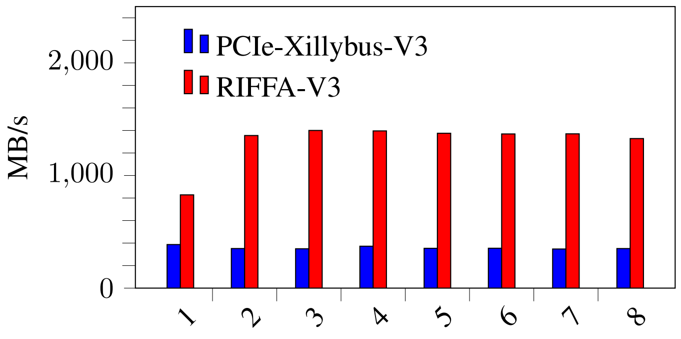


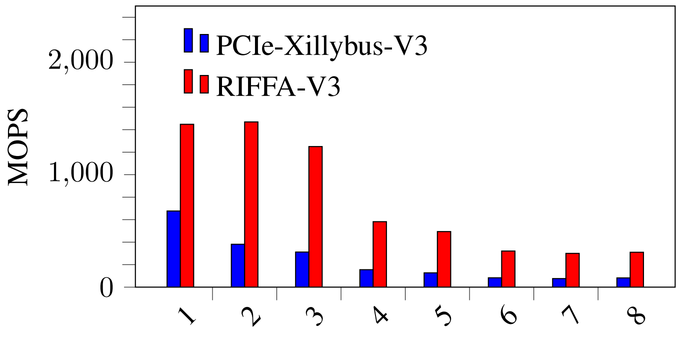
Figure 5.19: RIFFA loopback throughput.

Figure 5.20 shows the performance of four 32-bit linear TM V3 overlay integrated with PCIe-Xillybus (PCIe-Xillybus-V3) and RIFFA (RIFFA-V3), respectively. AXI-Xillybus uses a block size of 1M words with the Pthread multi-threading technique, while RIFFA uses a block size of 128K words. Both PCIe-Xillybus-V3 and RIFFA-V3 are running on an Ubuntu 14.04.5 workstation (six 3.5 GHz Intel Xeon E5-1650 cores) with a Xilinx VC707 evaluation board plugged into the PCIe Gen2 slot of the motherboard. As can be seen from Figure 5.20, RIFFA-V3 achieves an average throughput of 1300 MB/s (32.5% of theoretical maximum), which is around 3.6$\times$ better than PCIe-Xillybus-V3 with an average throughput of 358 MB/s (9% of theoretical maximum). The ratio of the throughput in MOPS is proportional to that of data processing throughput in MB/s.

Table 5.5 shows the hardware resource usage of RIFFA-V3 and PCIe-Xillybus-V3. As seen from this table, RIFFA-V3 consumes 15% fewer LUTs, 49% more FFs, 19$\times$ more BRAMs and same DSP blocks in comparison with PCIe-Xillybus-V3. There is no big difference among



(a) Data processing throughput in MB/s



(b) Throughput in MOPS

Figure 5.20: Performance comparison for the benchmarks.

the usage of logic slices and the DSP blocks except for the BRAMs. This is due to the half-duplex mode of RIFFA interface, resulting in large consumptions in BRAMs to implement two FWFT FIFOs with a large depth. Developing a full-duplex RIFFA based system would be very beneficial to minimize the resource usage, especially the BRAMs.

Table 5.5: Area overhead of RIFFA-V3 and PCIe-Xillybus-V3.

System	Resource Consumption			
	LUTs	FFs	BRAMs	DSPs
PCIe-Xillybus-V3	16,027	12,488	14.5	32
RIFFA-V3	13,657	18,581	289.5	32
Available	**303,600**	**607,200**	**1,030**	**2,800**

5.5 Discussion

We saw that AXI-Xillybus-V3 represents a very area efficient implementation compared with VectorBlox MXP-V16, at the expense of around half of the throughput. However, AXI-Xillybus has only half the theoretical bandwidth of MXP, as Xillybus uses just 32-bits of an ACP port while MXP uses the full 64-bits of an HP port. If Xillybus were modified to use a 64-bit port (something that will be left for future work), and two parallel V3 overlays were implemented, both implementations would then have similar throughput. The AXI-Xillybus dual V3 implementation would require significantly fewer hardware resources (approximately 20% of the LUTS and 8% of the hard macros (BRAM and DSP) required by MXP on Zynq. Thus, the linear TM overlay represents a relatively efficient implementation when FPGA resources are limited, as would be the case when an accelerator was used with other major subsystems in a design.

PCIe-Xillybus-V3 and RIFFA-V3 are proposed for more high-performance centric accelerator systems. The PCIe-Xillybus-V3 has a 1.6× better performance compared to the AXI-Xillybus-V3, at the expense of more than twice of resource consumption. However, the throughput of PCIe-Xillybus-V3 is slightly less than that of MXP-V16, as it achieves only 9% of the theoretical maximum of the PCIe Gen2x8 interface. Among all the 4 implementations, RIFFA-V3 shows the best performance, with a throughput of 1300 MB/s on average. However, the half-duplex mode of RIFFA-V3 makes it suffer from a large number of BRAMs.

5.6 Summary

In this chapter, we have proposed a number of overlay accelerator systems based on two different memory interfaces, AXI and PCIe. The AXI-Xillybus-V3 represents a very area efficient implementation compared with VectorBlox MXP-V16, at the expense of around half of the throughput. If Xillybus were modified to use a 64-bit ACP port and two parallel V3 overlays were implemented, it would be likely to achieve a throughput close to MXP-V16. The RIFFA-V3 has a 3.6× better performance compared to the PCIe-Xillybus-V3, and a 5.7× better performance than AXI-Xillybus-V3. However, the existing RIFFA implementation consumes a large number of BRAMs. RIFFA-V3 appears to be the most promising overlay accelerator for high computing purposes, however, there is a need to develop a full-duplex system, which will not only reduce the BRAMs utilization, but it will also double the theoretical throughput.

Chapter 6

Conclusions and Future Work

Coarse-grained overlay architectures have appeared to be a promising method to improve the design productivity of FPGAs by avoiding the tedious fine-grained placement and routing process. Time-multiplexing the overlay allows it to change its behavior on a cycle-by-cycle basis while the compute kernel is executing, thus allowing sharing of the limited FPGA resources, at the expense of a higher II and hence a reduced throughput. However, most of the existing CGRA-like TM overlays still suffer from relatively large area overheads due to the routing resources and instruction storage requirements. Reducing the area overhead for these overlays, specifically for the routing network, and utilizing the fast context switch capabilities of these overlays are likely to result in better usability with corresponding improvements in design productivity.

In this thesis, we propose an area efficient overlay based on a linear array of time-multiplexed FUs, which significantly reduces the FPGA hardware requirements while adding just a modest performance penalty, compared to the existing TM overlays. We explore the design space of this overlay, and present a number of architectural enhancements along with new instruction scheduling strategies to significantly improve the throughput and the latency compared to the original overlay. A toolchain is developed to compile high level language descriptions of compute intensive kernels onto the linear TM overlays, thus raising the hardware design to a higher level of abstraction, while also reducing compile times significantly. The overlay subsystems are integrated into complete hardware accelerator systems along with AXI or PCIe memory interfaces to an ARM processor on the Zynq SoC or a host CPU on the HP Z420 workstation.

The performance of these hardware accelerators for a range of benchmarks is investigated and performance results are presented and compared to a state-of-art TM overlay, VectorBlox MXP. This chapter draws conclusions by summarizing all the contributions presented in this thesis and outlines potential directions for future research.

6.1 Summary of Contributions

6.1.1 An Area-Efficient TM Overlay with Linear Interconnect

In Chapter 3, we proposed a resource efficient FPGA overlay that uses a linear connection of time-multiplexed FUs based on the Xilinx DSP48E1 macro. This architecture enables a significant reduction in the resource overheads compared to other TM overlays due to a reduction in the instruction storage and interconnect resource requirements, assisted by a fully-pipelined, architecture-aware FU design. While the proposed overlay exhibits a lower throughput than spatially configured overlays due to the much larger II, it has a significantly reduced FPGA resource requirement and a much lower context switching time. Our experimental evaluation shows that for a range of benchmarks, the proposed overlay delivers a throughput which is 6× to 22× less than the SC DSP-based overlay and Vivado implementations, but which uses 85% fewer eSlices than the DSP-based overlay and just 35% more eSlices than the Vivado implementations.

6.1.2 Architectural Enhancements for the Proposed Overlay

In Chapter 4, we examined the limitations of the linear TM overlay presented in Chapter 3 and explored a number of architectural enhancements to improve the throughput. Interleaving data transfer with instruction execution significantly reduces the II with a resulting increase in throughput. Introducing write-back into the FU design and modifying the scheduling strategy allows the overlay depth to be fixed. This eliminates the need to reconfigure the overlay if the application kernel changes, making the overlay more general. These changes significantly reduce the latency with just a small decrease in throughput. Compared to the original version, the modified overlays (V1 to V4) can achieve up to 2.4× higher throughput in GOPS, 93.7% higher compute efficiency in MOPS/eSlice and a 43.7% lower latency. The V3 overlay has the best performance among all the modified overlays, as for a fixed data bandwidth, it has comparable throughput with a significantly reduced latency.

6.1.3 Overlay Accelerator Systems based on AXI or PCIe Interface

In Chapter 5, we presented a number of overlay accelerator systems based on two different memory interfaces, AXI and PCIe. The AXI-Xillybus-V3 represents a very area efficient implementation compared with VectorBlox MXP-V16, at the expense of around half of the throughput. This is mainly because Xillybus only uses half of the 64-bit ACP port. If Xillybus were modified to use a 64-bit ACP port and two parallel V3 overlays were implemented, it would be likely to achieve a throughput close to MXP-V16. PCIe-Xillybus-V3 and RIFFA-V3 were proposed for more high-performance centric accelerator systems. The RIFFA-V3 has a 3.6$\times$ better performance compared to the PCIe-Xillybus-V3, and a 5.7$\times$ better performance than AXI-Xillybus-V3. However, the existing RIFFA implementation consumes a large number of BRAMs. RIFFA-V3 appears to be the most promising overlay accelerator for high computing purposes, however, there is a need to develop a full-duplex system, which will not only reduce the BRAMs utilization, but it will also double the theoretical throughput.

6.2 Future Work

We have proposed a number of linear TM overlay accelerators which demonstrate benefits such as area efficiency, high performance and fast context switching. Apart from these advantages, there are still some potential directions which can be further explored. We summarize the future work as follows:

6.2.1 Exploring More Applicable Interconnects

The proposed TM overlays were designed as a linear connection of highly pipelined time-multiplexed FUs to achieve data processing using a quasi-streaming method. However, these overlays are more applicable to the DFGs which have a small number of inputs and outputs. For example, when the DFG has a large number of input ports, the first FU is over-utilized while the rest of the FUs are in an idle state, resulting in an unbalanced FU utilization. A more customized routing network such as an inverse cone-shaped cluster of FUs inspired from DeCO [38], may overcome this problem and further reduce the II, compared to a direct linear interconnect of FUs.

6.2.2 Developing Better Instruction Schedules

Most of the existing CGRA architectures adopt Modulo scheduling [98], or a derivative algorithm, to achieve a minimum II. However, Modulo scheduling is based on the assumption that each operation node is executed in 1 cycle and the transfer of data between two arbitrary FUs completes in 1 cycle, which is not realistic for highly pipelined architectures. In the mapping toolflow, our current instruction scheduling strategy is a simple ASAP scheduling for the case that depth of the benchmarks is not larger than the number of FUs. However, the ASAP scheduling may not achieve the best nodes allocation for the FUs as it is not heuristic. A simplified version of list scheduling [107] referred to as Hu's algorithm [108] could be adopted to determine the minimum allocated nodes given a constraint to the number of FUs. By taking the pipeline stages into account, the ASAP scheduling and the iterative greedy scheduling in this thesis are reduced to that of multiprocessor scheduling as all the FUs are identical and hence have the same latencies for different operations. This needs to be further investigated.

6.2.3 Improving the Memory Interfaces

The current version of AXI-Xillybus only uses half the width of the 64-bit ACP port (e.g. 32-bits). If Xillybus were modified to use a full 64-bit ACP port and two parallel V3 overlays were implemented, it would achieve a throughput close to MXP-V16. Further, most of the existing memory solutions based on AXI HP interfaces do not use the full bandwidth available. Using just two of the four 64-bit ports available would allow four overlays to be replicated thus achieving a much higher performance. RIFFA-V3 appears to be the most promising memory interface for an overlay accelerator for high throughput computing purposes, however, there is a need to develop a full-duplex system, which will not only reduce the BRAMs utilization, but will also double the theoretical throughput.

6.2.4 Python Productivity for the Overlays

In recent years, the FPGA vendor, Xilinx, has been active in exploring the use of Python productivity for Zynq (PYNQ) [109] to improve design productivity and to allow an easier path to FPGA application development and use. PYNQ is an open-source project which provides a simple platform independent method to design high performance embedded applications on the Zynq FPGA. It consists of API accessible reconfigurable libraries (or overlays) which are programmable using Python in a browser based Jupyter Notebook. PYNQ has been quickly adopted by the reconfigurable computing research community such as the SPARK data analytics framework [110], deep recurrent neural network [111] and dynamic partial reconfiguration [112]. The proposed overlay accelerators would be of practical use if they were integrated within the PYNQ project.

Appendix A

Example Benchmark DFGs [1–3]

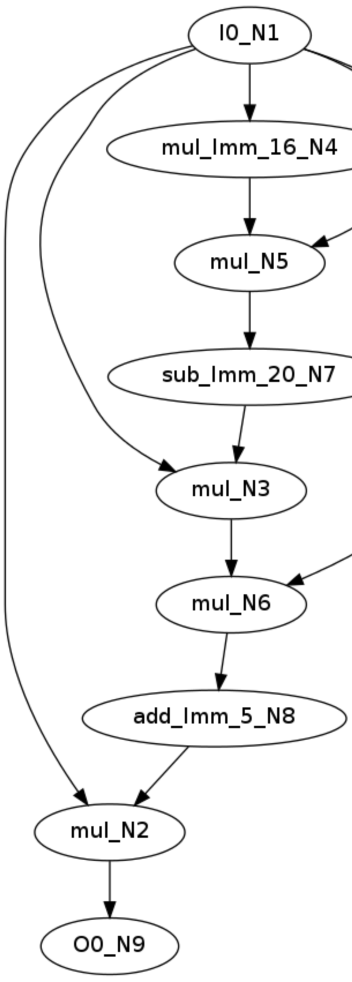


Figure A.1: DFG of ‘chebyshev’ benchmark.

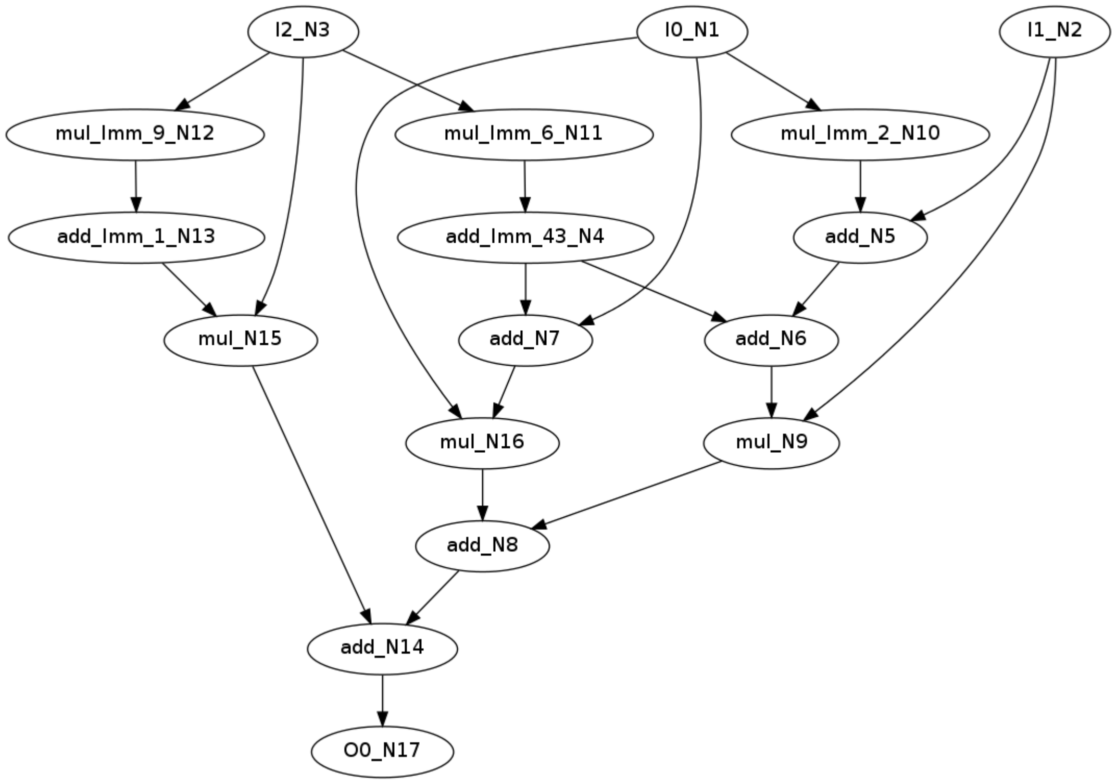


Figure A.2: DFG of ‘mibench’ benchmark.

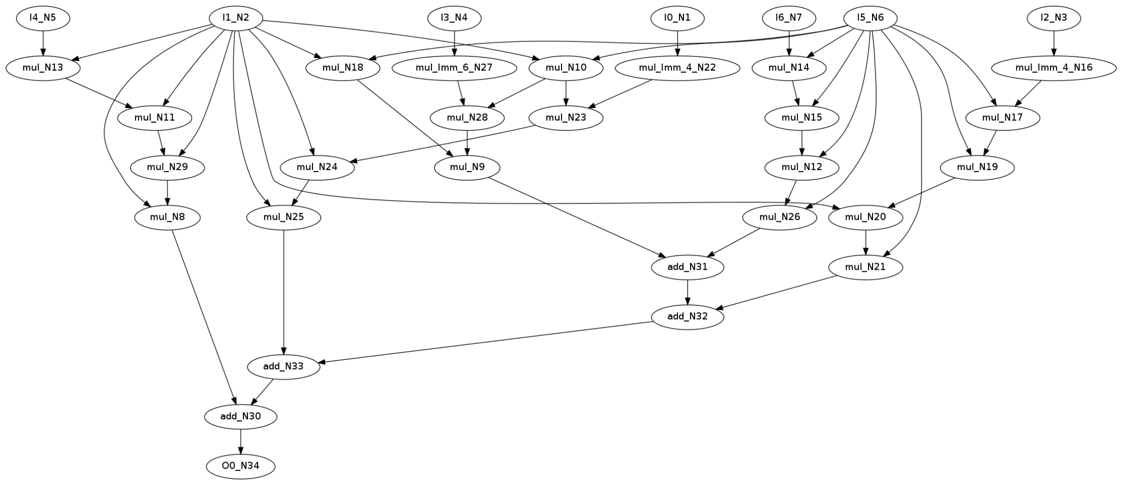


Figure A.3: DFG of ‘qspline’ benchmark.

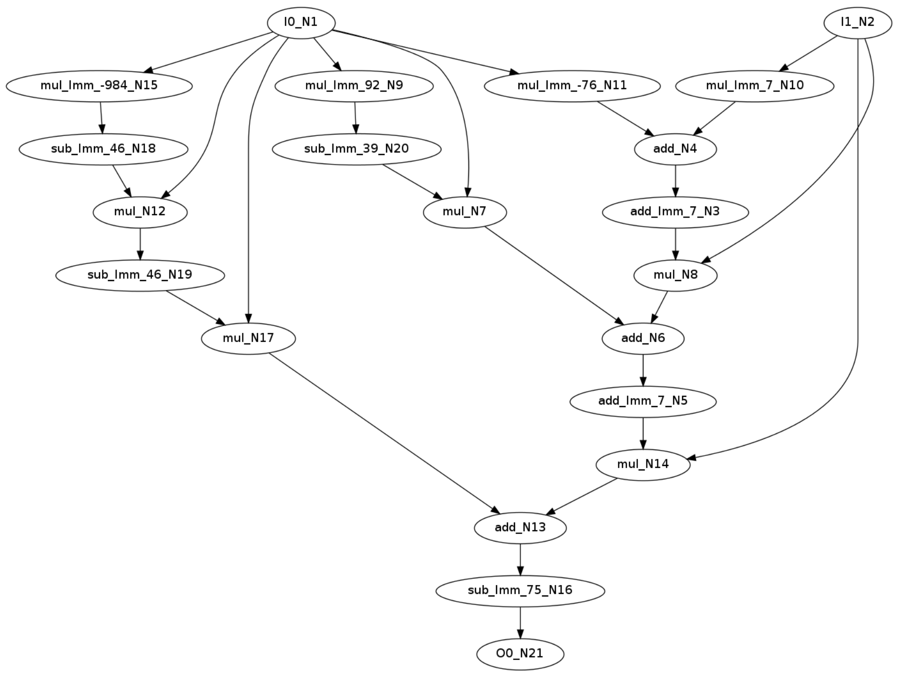


Figure A.4: DFG of ‘sgfilter’ benchmark.

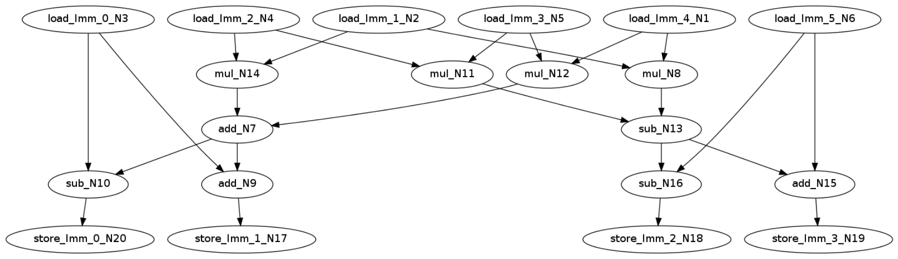


Figure A.5: DFG of ‘fft’ benchmark.

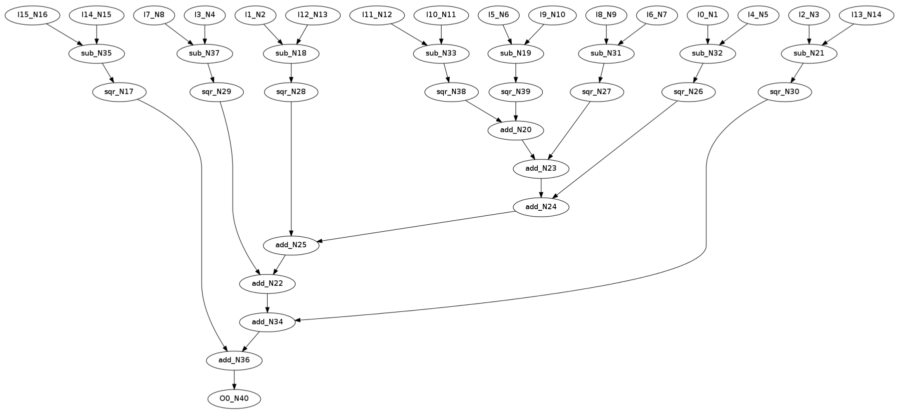


Figure A.6: DFG of ‘kmeans’ benchmark.

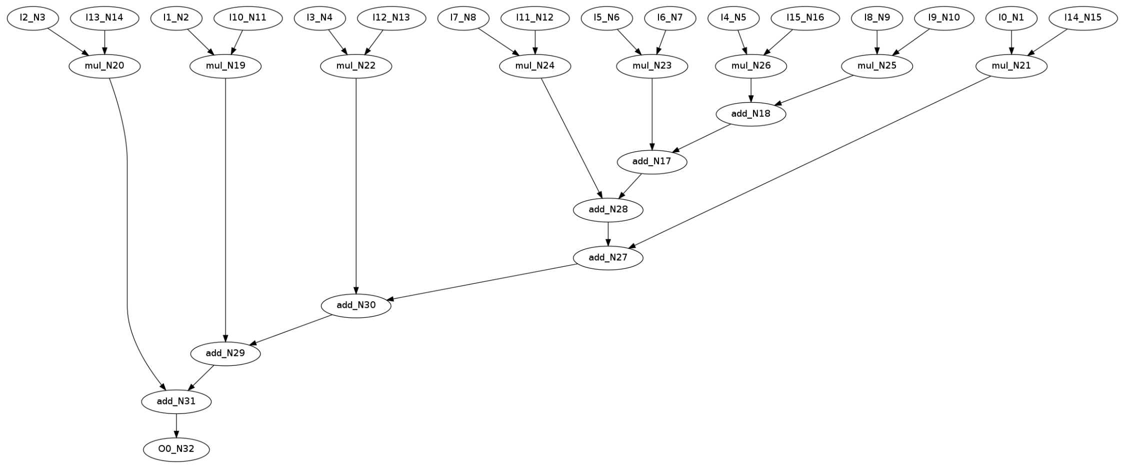


Figure A.7: DFG of ‘mm’ benchmark.

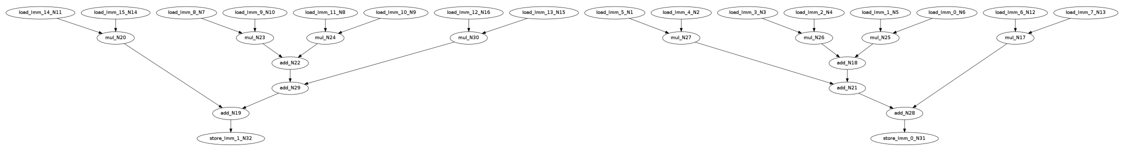


Figure A.8: DFG of ‘spmv’ benchmark.

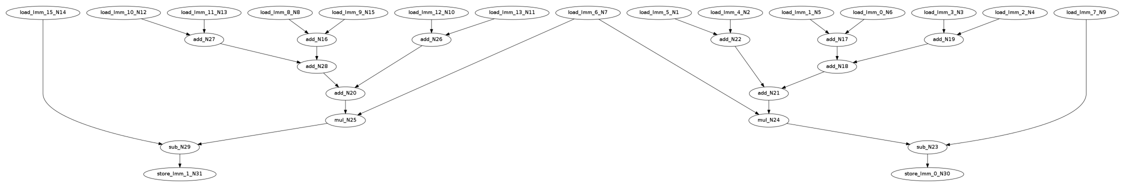


Figure A.9: DFG of ‘stencil’ benchmark.

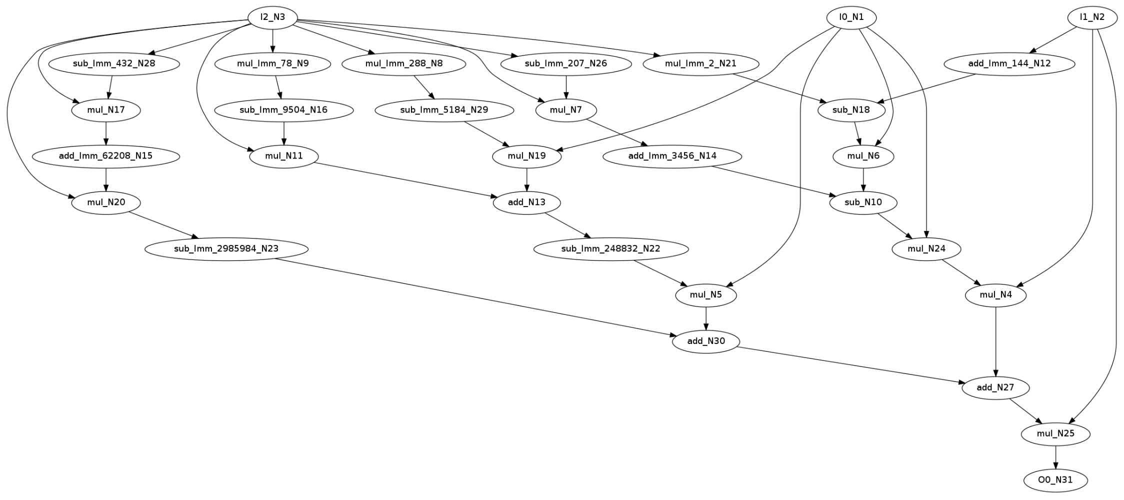


Figure A.10: DFG of ‘poly5’ benchmark.

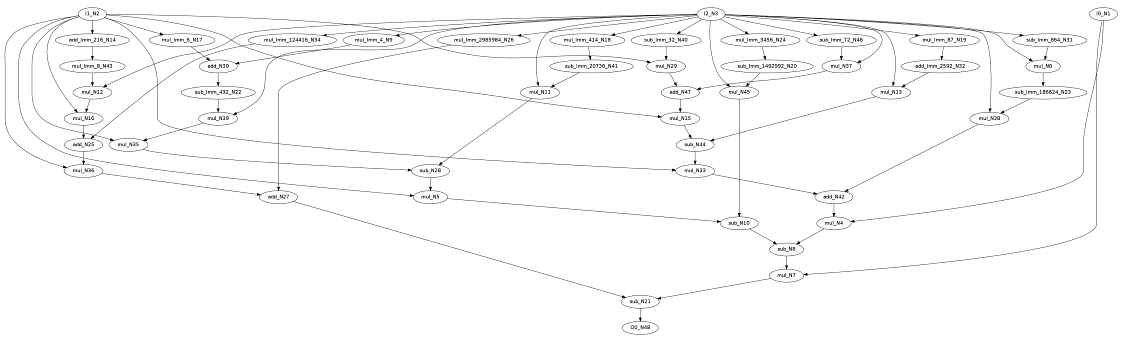


Figure A.11: DFG of ‘poly6’ benchmark.

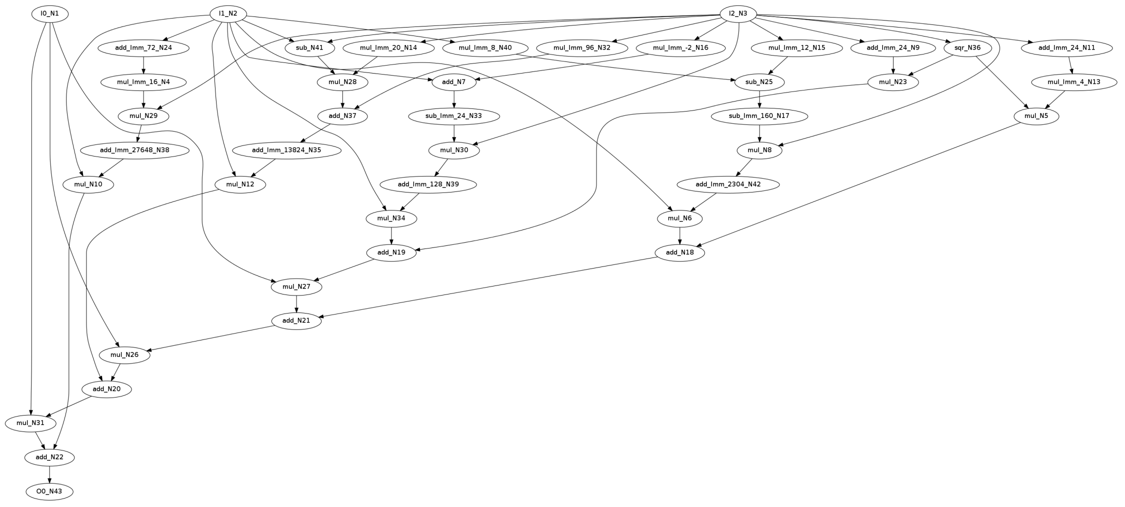


Figure A.12: DFG of ‘poly7’ benchmark.

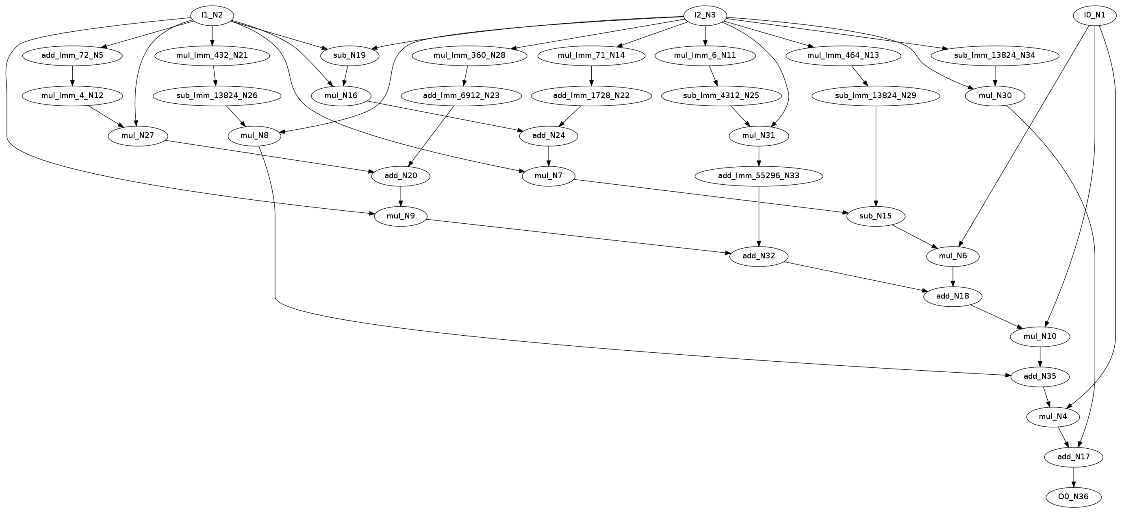


Figure A.13: DFG of ‘poly8’ benchmark.

References

[1] D. Bini and B. Mourrain, "Polynomial test suite, 1996." [Online]. Available: http://www-sop.inria.fr/saga/POL

[2] S. Gopalakrishnan, P. Kalla, M. B. Meredith, and F. Enescu, "Finding linear building-blocks for RTL synthesis of polynomial datapaths with fixed-size bit-vectors," in *Int. Conf. On Comput. Aided Design (ICCAD)*, 2007, pp. 143–148.

[3] C.-H. Hoy, V. Govindarajuz, T. Nowatzki, R. Nagaraju, Z. Marzecy, P. Agarwal, C. Frericks, R. Cofell, and K. Sankaralingam, "Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation," in *Int. Symp. on Performance Analysis of Syst. and Software (ISPASS)*, 2015, pp. 203–214.

[4] X. Ltd. 7 Series DSP48E1 Slice User Guide. [Online]. Available: www.xilinx.com

[5] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Trans. on Reconfigurable Technol. and Syst. (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[6] D. Capalija and T. S. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *Proc. 23rd Int. Conf. Field Program. Logic Appl. (FPL)*, 2013, pp. 1–8.

[7] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architecture based on DSP blocks," in *Proc. 23rd Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2015, pp. 25–28.

[8] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[9] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *Proc. 22nd Int. Conf. Field Program. Logic Appl. (FPL)*, 2012, pp. 531–534.

[10] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2011, pp. 33–36.

[11] D. Koch, F. Hannig, and D. Ziener, *FPGAs for Software Programmers*, 2016.

[12] R. Rashid, J. G. Steffan, and V. Betz, "Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS," in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, 2014, pp. 20–27.

[13] D. Koch, C. Beckhoff, and G. G. Lemieux, "An efficient FPGA overlay for portable custom instruction set extensions," in *Proc. 23rd Int. Conf. Field Program. Logic Appl. (FPL)*, 2013, pp. 1–8.

[14] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *ACM Sigplan Notices*, vol. 23, no. 7, 1988, pp. 318–328.

[15] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," in *Proc. 18th Int. Symp. High Performance Comput. Archit. (HPCA)*, 2012, pp. 1–12.

[16] J. Coole and G. Stitt, "Adjustable-cost overlays for runtime compilation," in *Proc. 23rd Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2015, pp. 21–24.

[17] K. Paul, C. Dash, and M. S. Moghaddam, "reMORPH: a runtime reconfigurable architecture," in *Proc. 15th Euromicro Conf. Digit. Syst. Design (DSD)*, 2012, pp. 26–33.

[18] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Proc. Int. Conf. Hardware/Software Codesign and Syst. Synthesis (CODES+ ISSS)*, 2010, pp. 13–22.

[19] K. Heyse, T. N. Davidson, E. Vansteenkiste, K. Bruneel, and D. Stroobandt, "Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAs," in *Proc. 23rd Int. Conf. Field Program. Logic Appl. (FPL)*, 2013, pp. 1–8.

[20] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proc. Int. Symp. High Performance Comput. Archit. (HPCA)*, 2011, pp. 503–514.

[21] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput oriented FPGA overlays using DSP blocks," in *Proc. Conf. Design, Autom. and Test in Europe (DATE)*, 2016, pp. 1628–1633.

[22] C. Liu, H.-C. Ng, and H. K.-H. So, "QuickDough: a rapid FPGA loop accelerator design framework using soft CGRA overlay," in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, 2015, pp. 56–63.

[23] J. Gray, "GRVI-Phalanx: A massively parallel RISC-V FPGA accelerator," in *Proc. 24th Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2016, pp. 17–20.

[24] N. Kapre and J. Gray, "Hoplite: Building austere overlay NoCs for FPGAs," in *Proc. 25th Int. Conf. Field Program. Logic Appl. (FPL)*, 2015, pp. 1–8.

[25] D. Capalija and T. S. Abdelrahman, "Towards synthesis-free JIT compilation to commodity FPGAs," in *Proc. 19th Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2011, pp. 202–205.

[26] A. Severance and G. G. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *Proc. Int. Conf. Hardware/Software Codesign and Syst. Synthesis (CODES+ ISSS)*, 2013, pp. 1–10.

[27] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008.

[28] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Comput.*, no. 4, pp. 18–25, 2009.

[29] G. Stitt, "Are field-programmable gate arrays ready for the mainstream?" *IEEE Micro*, vol. 31, no. 6, pp. 58–63, 2011.

[30] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A coarse grained paradigm for FPGAs," in *Proc. Dagstuhl Seminar*, 2006.

[31] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *Proc. Int. Conf. Compilers, Archit. and Synthesis for Embedded Syst. (CASES)*, 2005, pp. 202–212.

[32] M. A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: Fully synthesizable parameterized mips-based multicore system," in *Proc. 21st Int. Conf. Field Program. Logic Appl. (FPL)*, 2011, pp. 356–362.

[33] A. Brant and G. G. Lemieux, "ZUMA: An open FPGA overlay architecture," in *Proc. 20th Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2012, pp. 93–96.

[34] C. E. LaForest and J. G. Steffan, "Octavo: an FPGA-centric processor family," in *Proc. 20th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2012, pp. 219–228.

[35] A. D. Brant, "Coarse and fine grain programmable overlay architectures for FPGAs," Ph.D. dissertation, University of British Columbia, 2013.

[36] C. Liu, C. L. Yu, and H. K.-H. So, "A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme FPGA frequency," in *Proc. 21st Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2013, pp. 228–228.

[37] K. Ovtcharov, I. Tili, and J. G. Steffan, "TILT: a multithreaded VLIW soft processor family," in *Proc. 23rd Int. Conf. Field Program. Logic Appl. (FPL)*, 2013, pp. 1–4.

[38] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy, "DeCO: a DSP block based FPGA accelerator overlay with low overhead interconnect," in *Proc. 24th Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2016, pp. 1–8.

[39] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell, "Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform," *J. of Signal Process. Syst.*, vol. 77, no. 1-2, pp. 61–76, 2014.

[40] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on FPGAs?" in *Proc. 14th Int. Conf. on Pervasive, Intel. and Comput. (PICom)*, 2016, pp. 586–593.

[41] R. Dimond, O. Mencer, and W. Luk, "CUSTARD-a customisable threaded FPGA soft processor and tools," in *Proc. 15th Int. Conf. Field Program. Logic Appl. (FPL)*, 2005, pp. 1–6.

[42] F. Plavec, B. Fort, Z. G. Vranesic, and S. D. Brown, "Experiences with soft-core processor design," in *Proc. 19th Int. Parallel and Distrib. Process. Symp. (IPDPS)*, 2005, p. 167b.

[43] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, "GRLIB IP core user's manual," *Gaisler research*, 2007.

[44] T. Kranenburg and R. Van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture," in *Proc. Conf. Design, Autom. and Test in Europe (DATE)*, 2010, pp. 997–1000.

[45] C. G. AB, "GRLIB IP core user's manual," 2017.

[46] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: A DSP block based FPGA soft processor," in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, 2012, pp. 151–158.

[47] Xilinx, "MicroBlaze processor reference guide," 2017.

[48] Altera, "Nios II processor reference handbook," 2016.

[49] D. L. Weaver, *OpenSPARC Internals: OpenSPARC T1/T2 CMT Throughput Computing*. Sun Microsystems, 2008.

[50] D. Lampret, "OpenRISC 1200 IP core specification," 2001. [Online]. Available: https://opencores.org

[51] S. Rhoads, "Plasma-most MIPS I(TM) opcodes," 2009. [Online]. Available: https://opencores.org/project,plasma

[52] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[53] R. Jia, C. Y. Lin, Z. Guo, R. Chen, F. Wang, T. Gao, and H. Yang, "A survey of open source processors for FPGAs," in *Proc. 24th Int. Conf. Field Program. Logic Appl. (FPL)*, 2014, pp. 1–6.

[54] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *Proc. 16th Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2008, pp. 195–205.

[55] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *Proc. 14th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2006, pp. 201–210.

[56] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, "The iDEA DSP block-based soft processor for FPGAs," *ACM Trans. on Reconfigurable Technol. and Syst. (TRETS)*, vol. 7, no. 3, p. 19, 2014.

[57] C. Hui Yan, S. Fahmy, and N. Kapre, "On data forwarding in deeply pipelined soft processors," in *Proc. 23rd Int. Symp. Field Program. Gate Arrays (FPGA)*, 2015, pp. 181–189.

[58] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund *et al.*, "Intel nehalem processor core made FPGA synthesizable," in *Proc. 18th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2010, pp. 3–12.

[59] M. Labrecque and J. G. Steffan, "Improving pipelined soft processors with multithreading," in *Proc. 17th Int. Conf. Field Program. Logic Appl. (FPL)*, 2007, pp. 210–215.

[60] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proc. 14th Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2006, pp. 131–142.

[61] R. Moussali, N. Ghanem, and M. A. Saghir, "Supporting multithreading in configurable soft processor cores," in *Proc. Int. Conf. Compilers, Archit. and Synthesis for Embedded Syst. (CASES)*, 2007, pp. 155–159.

[62] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEE Proc.-Comput. and Digital Tech.*, vol. 153, no. 3, pp. 173–180, 2006.

[63] C. E. Laforest and J. H. Anderson, "Microarchitectural comparison of the MXP and Octavo soft-processor FPGA overlays," *ACM Trans. on Reconfigurable Technol. and Syst. (TRETS)*, vol. 10, no. 3, p. 19, 2017.

[64] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW processor with custom hardware execution," in *Proc. 13th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2005, pp. 107–117.

[65] R. Rashid, “A dual-engine fetch/compute overlay processor for fpgas,” Ph.D. dissertation, University of Toronto, 2015.

[66] C. E. LaForest, “High-speed soft-processor architecture for FPGA overlays,” Ph.D. dissertation, University of Toronto, 2015.

[67] C. Kozyrakis and D. Patterson, “Overcoming the limitations of conventional vector processors,” *ACM SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 399–409, 2003.

[68] P. Yiannacouras, J. G. Steffan, and J. Rose, “VESPA: portable, scalable, and flexible FPGA-based vector processors,” in *Proc. Int. Conf. Compilers, Archit. and Synthesis for Embedded Syst. (CASES)*, 2008, pp. 61–70.

[69] J. Yu, G. Lemieux, and C. Eagleston, “Vector processing as a soft-core CPU accelerator,” in *Proc. 16th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2008, pp. 222–232.

[70] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, “VEGAS: soft vector processor with scratchpad memory,” in *Proc. 19th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2011, pp. 15–24.

[71] A. Severance and G. Lemieux, “VENICE: a compact vector processor for FPGA applications,” in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, 2012, pp. 261–268.

[72] C. Kozyrakis and D. Patterson, “Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks,” in *Proc. 35th Int. Symp. Microarchitecture*, 2002, pp. 283–293.

[73] E. D. H. EEMBC, “The embedded microprocessor benchmark consortium.” [Online]. Available: https://www.eembc.org

[74] P. Yiannacouras, J. G. Steffan, and J. Rose, “Fine-grain performance scaling of soft vector processors,” in *Proc. Int. Conf. Compilers, Archit. and Synthesis for Embedded Syst. (CASES)*, 2009, pp. 97–106.

[75] D. Lau, O. Pritchard, and P. Molson, “Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions,” in *Proc. 14th Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2006, pp. 45–56.

[76] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, “Vector processing as a soft processor accelerator,” *ACM Trans. on Reconfigurable Technol. and Syst. (TRETS)*, vol. 2, no. 2, p. 12, 2009.

[77] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, “Soft vector processors with streaming pipelines,” in *Proc. 22nd Int. Symp. Field Program. Gate Arrays (FPGA)*, 2014, pp. 117–126.

[78] K. Andryc, M. Merchant, and R. Tessier, “FlexGrip: A soft GPGPU for FPGAs,” in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, 2013, pp. 230–237.

[79] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol *et al.*, “Enabling GPGPU low-level hardware explorations with MIAOW: an open-source RTL implementation of a GPGPU,” *ACM Trans. on Archit. and Code Optimization (TACO)*, vol. 12, no. 2, p. 21, 2015.

[80] M. Al Kadi, B. Janssen, and M. Huebner, “FGPU: An SIMT-architecture for FPGAs,” in *Proc. 24th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2016, pp. 254–263.

[81] P. Duarte, P. Tomas, and G. Falcao, “SCRATCH: an end-to-end application-aware soft-GPGPU architecture and trimming tool,” in *Proc. 50th Int. Symp. Microarchitecture*, 2017, pp. 165–177.

[82] C. Hilton and B. Nelson, “PNoC: a flexible circuit-switched NoC for FPGA-based systems,” *IEE Proc.-Comput. and Digital Tech.*, vol. 153, no. 3, pp. 181–188, 2006.

[83] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix,” in *Proc. 13rd Int. Conf. Field Program. Logic Appl. (FPL)*, 2003, pp. 61–70.

[84] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh *et al.*, “MorphoSys: A reconfigurable architecture for multimedia applications,” in *Proc. XI Brazilian Symp. Integr. Circuit Design*, 1998, pp. 134–139.

[85] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell, “Adapting the DySER architecture with DSP blocks as an Overlay for the Xilinx Zynq,” *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 28–33, 2016.

[86] C. E. Leiserson, “Fat-trees: universal networks for hardware-efficient supercomputing,” *IEEE Trans. on Comput.*, vol. 100, no. 10, pp. 892–901, 1985.

[87] C. Kumar HB, P. Ravi, G. Modi, and N. Kapre, “120-core microAptiv MIPS Overlay for the Terasic DE5-NET FPGA board,” in *Proc. 25th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2017, pp. 141–146.

[88] S. L. Harris, R. Owen, E. Sedano, and D. C. Martinez, “MIPSfpga: hands-on learning on a commercial soft-core,” in *Proc. 11th European Workshop on Microelectronics Education (EWME)*, 2016, pp. 1–5.

[89] R. Hartenstein, “Coarse grain reconfigurable architectures,” in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2001, pp. 564–569.

[90] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in *Proc. Conf. Design, Autom. and Test in Europe (DATE)*, 2001, pp. 642–649.

[91] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro, "An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Proc. Int. Conf. Compilers, Archit. and Synthesis for Embedded Syst. (CASES)*, 2011, pp. 195–204.

[92] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of CGRA," in *Proc. 22nd Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2014, pp. 9–16.

[93] N. Kapre, H. Jianglei, A. Bean, P. Moorthy *et al.*, "GraphMMU: Memory management unit for sparse graph accelerators," in *Proc. Int. Parallel and Distrib. Process. Symp. Workshop (IPDPSW)*, 2015, pp. 113–120.

[94] G. Stitt and J. Coole, "Intermediate fabrics: Virtual architectures for near-instant FPGA compilation," *IEEE Embedded Syst. Letters*, vol. 3, no. 3, pp. 81–84, 2011.

[95] N. Kavvadias and K. Masselos, "Hardware design space exploration using HercuLeS HLS," in *Proc. 17th Panhellenic Conf. on Informatics (PCI)*, 2013, pp. 195–202.

[96] X. Li, A. Jain, D. Maskell, and S. A. Fahmy, "An area-efficient FPGA overlay using DSP block based time-multiplexed functional units," in *Proc. 2nd Int. Workshop on Overlay Archit. for FPGAs (OLAF)*, 2016.

[97] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 283–299, 1992.

[98] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Int. Symp. on Microarchitecture (MICRO)*, 1994, pp. 63–74.

[99] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Syst. Letters*, vol. 6, no. 3, pp. 41–44, 2014.

[100] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ," in *Proc. 10th Conf. FPGAworld*, 2013, p. 5.

[101] Xillybus Ltd., "IP core product brief." [Online]. Available: https://www.xillybus.com

[102] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *Proc. 24th Int. Conf. Field Program. Logic Appl. (FPL)*, 2014, pp. 1–7.

[103] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, "An efficient and flexible host-FPGA PCIe communication library," in *Proc. 24th Int. Conf. Field Program. Logic Appl. (FPL)*, 2014, pp. 1–6.

[104] D. de la Chevallerie, J. Korinth, and A. Koch, “ffLink: A lightweight high-performance open-source PCI Express Gen3 interface for reconfigurable accelerators,” *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 34–39, 2016.

[105] Northwest Logic Inc., “PCI Express solution overview.” [Online]. Available: https://nwlogic.com/products/pci-express-solution/

[106] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy, “JetStream: An open-source high-performance PCI express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication,” in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, 2016, pp. 1–9.

[107] T. L. Adam, K. M. Chandy, and J. Dickson, “A comparison of list schedules for parallel processing systems,” *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[108] T. C. Hu, “Parallel sequencing and assembly line problems,” *Operations research*, vol. 9, no. 6, pp. 841–848, 1961.

[109] Xilinx Ltd., “PYNQ: Python productivity for Zynq.” [Online]. Available: http://www.pynq.io

[110] E. Koromilas, I. Stamelos, C. Kachris, and D. Soudris, “Spark acceleration on FPGAs: A use case on machine learning in Pynq,” in *Proc. 6th Int. Conf. Modern Circuits and Syst. Technol. (MOCAST)*, 2017, pp. 1–4.

[111] Y. Hao and S. Quigley, “The implementation of a deep recurrent neural network language model on a Xilinx FPGA,” *arXiv preprint arXiv:1710.10296*, 2017.

[112] B. Janßen, P. Zimprich, and M. Hübner, “A dynamic partial reconfigurable overlay concept for PYNQ,” in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, 2017, pp. 1–4.