NANYANG TECHNOLOGICAL UNIVERSITY

Architecture Centric Coarse-Grained FPGA Overlays

Abhishek Kumar Jain

School of Computer Science and Engineering

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

January 2017

THESIS ABSTRACT

Architecture Centric Coarse-Grained FPGA Overlays

by

Abhishek Kumar Jain

Doctor of Philosophy

School of Computer Science and Engineering

Nanyang Technological University, Singapore

Coarse-grained FPGA overlays have emerged as one possible solution to make FPGAs more accessible to application developers who are accustomed to software API abstractions and fast development cycles. Existing overlay architectures offer a number of advantages for general purpose hardware acceleration because of software-like programmability, fast compilation, application portability, and improved design productivity, but at the cost of area and performance overheads due to limited consideration for the underlying FPGA architecture. This thesis explores coarse grained overlays designed using the flexible DSP48E1 primitive on Xilinx FPGAs, allowing pipelined execution of compute kernels at significantly higher throughput. We first evaluate an open source overlay architecture, DySER, mapped on the Xilinx Zynq device and show that DySER suffers from a significant area and performance overhead due to limited consideration for the underlying FPGA architecture. Next, we design and implement a more FPGA targeted overlay architecture that maximizes the peak performance and reduces the interconnect area overhead through the use of an array of DSP block based fully pipelined functional units and an island-style coarse-grained routing network. As the interconnect of the island-style overlay is still excessive, we next explore novel interconnect architectures to further reduce the interconnect area. We next develop DeCO, a cone shaped cluster of FUs, which shows 87% savings in LUT requirements compared to our island-style overlay, for a set of compute kernels. Our experimental evaluation shows that the proposed overlays exhibit frequencies close to the DSP theoretical limit and achieve high performance with significantly reduced area overheads. We also present a methodology for compiling high level language (C/OpenCL) descriptions of compute kernels onto DSP block based coarse-grained overlays. Our mapping flow provides a rapid, vendor independent mapping to the overlay, raising the abstraction level while also reducing compilation time significantly, hence addressing the design productivity issue.

Acknowledgements

First and foremost, I take this opportunity to thank my supervisor Associate Professor Douglas Leslie Maskell for giving me the opportunity to work in the area of reconfigurable computing, and then, expertly guiding me through this journey. His enthusiastic support and strong encouragement helped me carry out my research with composure and confidence. He has been appreciative of my ideas and encouraged critical thinking and creative writing, which have helped me to improve my skills. I would have not been able to complete my research work successfully without his support and directions.

I would also like to thank Associate Professor Suhaib A Fahmy, my co-supervisor, for his professional guidance, continuous support, effective suggestions, constructive criticism and timely help. He has been ever generous in taking out time to help me correct my mistakes and improve my technical writing. My grateful thanks are also extended to Dr. Kyle Rupnow for his constructive criticism, encouragement and endless support. I appreciate the suggestions of my mentors and especially the time they have taken to help me correct my mistakes and improve my writing skills.

I would also like to express appreciation for my friends and colleagues at Hardware and Embedded Systems Lab (HESL), especially Ronak Bajaj, Shreejith Shanker, Vipin Kizheppat, Abhishek Ambede, Sumedh Dhabu, Rakesh Varier, Sanjeev Kumar Das, Liwei Yang, Xiangwei Li, Vikram Shenoy Handiru, Supriya Sathyanarayana, Kratika Garg, Saru Vig and Tushar Chouhan for their invaluable support and encouragement. I am also thankful to our laboratory executive, Chua Ngee Tat, for his priceless technical assistance.

Finally, I am indebted to my family and my parents for their prayers and encouragement. I thank them for their understanding and their efforts to support me in pursuing higher studies.

Contents

List of Figures

List of Tables

List of Abbreviations

ALAP	As Late As Possible
ALM	Adaptive Logic Module
ALU	Arithmetic and Logic Unit
ALUT	Adaptive Look Up Table
API	Application Programming Interface
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
CB	Connection Box
CBCR	Connection Box Configuration Register
CGRA	Coarse Grained Reconfigurable Architecture
CLB	Configurable Logic Block
CPU	Central Processing Unit
CU	Compute Unit
CW	Channel Width
DeCO	DSP enabled Cone-shaped Overlay
DF	Data Forwarding
DFG	Data Flow Graph
DISO	DSP block based Island-Style Overlay
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processing
Dual-DISO	Dual-DSP block based Island-Style Overlay
DySER	Dynamically Specialized Execution Resources
FF	Flip Flop
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit

FU	Functional Unit
FUCR	Functional Unit Configuration Register
GOPS	Giga Operations Per Second
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HLL	High Level Language
HLS	High Level Synthesis
IF	Intermediate Fabric
II	Initiation Interval
ILP	Integer Linear Programming
IoT	Internet of Things
JIT	Just In Time
LUT	Look Up Table
MIMD	Multiple Instruction, Multiple Data
MOPS	Million Operations Per Second
MPPA	Massively Parallel Processor Array
NN	Nearest Neighbor
NRE	Non Recurring Engineering
OS	Operating System
PAR	Placement And Routing
PE	Processing Element
POB	Programmability Overhead per Bit
PR	Partial Reconfiguration
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
SB	Switch Box
SBCR	Switch Box Configuration Register
SC	Spatially Configured
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
SoPC	System on Programmable Chip
SRL	Shift Register LUT
TM	Time Multiplexed
VLIW	Very Long Instruction Word
VPR	Versatile Place and Route

1

Introduction

With the advancements in technology, parallel processing platforms such as graphics processing units (GPUs) [1, 2] and massively parallel processor arrays (MPPAs) [3, 4, 5, 6] are gaining popularity for accelerated execution of compute-intensive applications. However these platforms are unable to meet the ever increasing demand for computing power within a tight power budget. The performance gains achieved by adding more cores inside a computing platform come at the cost of rapidly scaling complexities to the inter-core communication, memory coherency and, most importantly, the power consumption. Application specific accelerators used to be preferred due to area, speed and energy efficiency [7, 8] and were deployed as an Application Specific Integrated Circuit (ASIC) block alongside a general purpose processor (GPP) [9, 10]. However, developing dedicated ASIC accelerators has become less practical due to the long turnaround time and high

cost associated with ASIC development. Field Programmable Gate Arrays (FPGAs), which allow the implementation to be modified post-deployment [11, 12, 13], are now more commonly used for rapid-prototyping of application specific accelerators. Some of the key advantages of FPGAs over other available platforms include reprogrammability compared to ASICs, lower power consumption than multicore processors and GPUs, real-time execution, and most importantly, the high spatial parallelism which can be used to significantly accelerate compute-intensive algorithms [14]. For more than a decade, researchers have shown that FPGAs can accelerate a wide variety of applications, in some cases by several orders of magnitude compared to state-of-the-art GPPs [15, 16, 17, 18, 19].

The gap between FPGAs and ASICs is shrinking with each new generation of FPGAs [16] due to advancements in process technologies, evolutions in the FPGA architecture, and the rising cost and complexity of ASIC design. As a result, FPGAs are now used for implementing large complex circuits, and are being deployed in production systems, replacing ASICs in areas like networking, where algorithms and protocols change quickly making them less feasible for ASIC implementation [20, 21]. The high non-recurring engineering (NRE) costs, high manufacturing costs, and long design time for ASICs are also motivating designers to use more FPGAs, for which the turn-around time is much reduced. The geometric growth in the logic density in FPGAs and the inclusion of coarse grained hard macros, such as Block RAMs and digital signal processing (DSP) Blocks [22], now makes FPGA a viable alternative to ASIC implementations for many compute-intensive circuits [23, 24, 25, 26, 27, 28].

This rapidly increasing logic density and the more capable hard resources in modern FPGA devices should make them more widely deployable. However, FPGAs remain constrained within specialist application domains, such as digital signal processing and communications. This is because accelerator design is a complex process, requiring low-level hardware device expertise and specialist knowledge of both hardware and software systems, resulting in major design productivity issues. High level synthesis (HLS) [29, 30, 31, 32, 33] has been proposed to address the

design productivity issue and has helped to simplify accelerator design by raising the level of programming abstraction from RTL to high level languages, such as C/C++/OpenCL. These tools allow the functionality of an accelerator to be described at a higher level to reduce developer effort, enable design portability, enable rapid design space exploration, thus improving productivity, verifiability, and flexibility. Even though HLS tools have improved in efficiency, allowing designers to focus on high level functionality instead of low-level implementation details, the prohibitive compilation times (specifically the place and route times in the back-end flow) have largely been ignored and are now a major productivity bottleneck that prevents designers from using mainstream design and debug methodologies based on rapid compilation. As such, HLS techniques are generally limited to static reconfigurable systems [34].

Another major stumbling block is the lack of a suitable abstraction at the hardware computing level, to eliminate the reliance on platform-specific detail, as has been achieved with server and desktop virtualization [35, 36, 37]. A key example of virtualization in a modern paradigm is cloud computing [38], where virtual resources are available on demand, with runtime mapping to physical systems abstracted from the user. So far, virtualization has focused primarily on conventional processor-based computing systems where high level management of computing tasks is supported by having a number of abstraction layers at different abstraction levels. However there is no agreed upon abstraction for FPGA fabrics.

One of the major benefits of FPGA as a rapidly reconfigurable hardware accelerator is to utilize the ability to partially and dynamically reconfigure the functionality of the FPGA fabric. Initial implementations of dynamic reconfiguration [39, 40] required the reconfiguration of the complete hardware fabric. This resulted in significant configuration overhead, which severely limited its usefulness. Xilinx then introduced the concept of dynamic partial reconfiguration (DPR) which reduced the configuration time by allowing a smaller region of the fabric to be dynamically reconfigured at runtime. The concept of DPR on FPGA is to use it as a virtual hardware element, implementing applications which are larger than the available

hardware resources. While DPR significantly improved reconfiguration performance [41], the efficiency of the traditional design approach for DPR is heavily impacted by how a design is partitioned and floorplanned [42, 43], tasks that require FPGA expertise. Furthermore, the commonly used configuration mechanism is highly sub-optimal in terms of throughput [44]. Unfortunately, the potential to use dynamic reconfiguration to adapt FPGA operation to changing application requirements has been hampered by slow reconfiguration times, poor CAD tool support, and large configuration file sizes, making dynamic reconfiguration difficult and inefficient.

In an accelerator context, this concept of being able to modify the behaviour at runtime by swapping in and out different accelerators and reusing the FPGA resources for multiple tasks [45, 46, 47, 48, 49, 50] is an advantage. A large application may need to be decomposed into several smaller hardware tasks, each of which is mapped to a temporal FPGA partition [51, 52] and executed in a time multiplexed manner under the control of an operating system (OS) which normally runs on a host processor [51, 52, 53, 54, 55]. While such ideas have been explored in the past, modern hybrid FPGAs are the first commercial platforms to enable this move to a more software oriented view. The combination of a GPP tightly coupled with high performance reconfigurable FPGA fabric, referred to as system on programmable chip (SoPC) [56], represents a possible solution to both high-end server-based computing [57, 58, 59] (where the performance curve of traditional server processors has begun to plateau) and high performance front-end processing for applications in the Internet-of-Things (IoT) domain (where performance and power are critical). A recent example of this from the server domain is Microsoft's Catapult, which is a server augmented with FPGAs to accelerate the Bing search engine [59]. The age old quest of building a computing platform [60] which can allow hardware resources to be tailored at runtime to perform a specific task in an efficient manner is now becoming part of the mainstream narrative.

However, despite the many accelerator success stories, there is a growing need to make FPGA hardware resources in a heterogeneous platform more accessible to application developers who are accustomed to software API abstractions and

fast development cycles [61]. The lack of platform abstraction and application portability prevents design reuse and adoption of these platforms for mainstream computing. Hence there is a need to relook at how to exploit the key advantages of the FPGA resources while abstracting implementation details within a software-centric processor-based system to achieve improved design productivity beyond what is achievable using HLS Tools. One possible solution is to treat the execution and management of software and hardware tasks in the same way, using an OS such that the hardware fabric is viewed as just another software-managed task [62, 63]. This enables more shared use, while ensuring better isolation and predictability. This run-time management, including FPGA configuration and interprocess data communication, was recently demonstrated for the Linux OS [62] using a coarse grained FPGA overlay. The use of an overlay, a programmable coarse-grained hardware abstraction layer on top of the FPGA, resulted in better application management, and has the potential for allowing portability across devices, software-like programmability through mapping from high-level descriptions, better design reuse, fast compilation by avoiding the complex FPGA implementation flow, and hence, improved design productivity. Another main advantage is rapid reconfiguration since the overlay architectures have a smaller configuration data size due to the coarse granularity.

Coarse-grained overlays have developed from ASIC-based coarse-grained reconfigurable architectures (CGRAs) [64, 65, 66, 51, 67, 68] in both the architectural choices and execution style. The key features that enabled CGRAs to address signal processing and high performance computing problems more efficiently include: energy efficiency, the ease of programming and application mapping [69, 70, 71], fast compilation and reconfiguration. CGRAs have not been successfully adopted as general purpose accelerators and instead are mainly seen as a component in SoCs for efficiently implementing a specific range of DSP functions as part of a larger system. This is because these functional units (FUs) are often too application specific making it difficult to find a particular configuration that suits a wide range of applications for the approach to be viable as a stand-alone product.

On the other hand, coarse-grained FPGA overlays are basically pre-implemented programmable components built on top of the FPGA hardware resources and can take many forms, including, soft-core processors, an array of programmable ALUs/processors [72], networks etc. Generally, these overlays consist of a regular arrangement of coarse grained routing and compute resources and in many cases an array of programmable FUs interconnected using a programmable interconnect architecture, similar to CGRA. The concept of a coarse-grained overlay is to use the FPGA as a programmable accelerator, instead of just as a hard-wired application specific accelerator. Coarse-grained overlays also attempt to replace the accelerator design problem with a problem of programming an array of ALUs/processors.

1.1 Motivation

Even though research into FPGA overlay architectures has increased over the last decade, the field is still in its infancy with only a relatively few overlay architectures demonstrated in prototype form [63, 73, 74]. Area and performance overheads have, however, prevented the realistic use of most of the overlays in practical FPGA-based systems. One of the reasons for this poor performance is that overlays are typically designed without serious consideration of the underlying FPGA architecture.

Embedded hard macros such as DSP blocks have been added to FPGAs in recent years. By building often used functions into optimised compact primitives, area, performance, and power advantages are achieved over equivalent "soft" implementations in the logic fabric. Many existing overlay architectures [63, 73, 74, 75] do not specifically use these macros, except insofar as they are inferred by the synthesis tools. However, it is well known that inference of hard macros by synthesis tools does not result in optimal throughput [76]. This thesis explores how the Xilinx DSP48E1 primitive can be used, at near to its theoretical limits, as a

programmable processing element (PE) to build efficient overlay architectures for pipelined execution of compute kernels.

For FPGA overlays to play a full-featured role alongside general purpose processors, it is essential that their key advantages be available in an abstracted manner that enables scaling, and makes them accessible to a wider audience. Our work explores how area and performance efficient overlays and automated mapping techniques can be used to improve design productivity. This is done by abstracting the complexity away from the designer, enabling overlay architectures to be exploited as a programmable accelerator alongside general purpose processors, within a software-centric runtime framework. This new design methodology approaches the fabric from the perspective of software-managed hardware tasks, enabling more shared use, while ensuring high performance and improved design productivity.

In this work, we aim to investigate novel techniques to design area and performance efficient FUs and interconnect architectures leading to high performance coarse-grained overlays. This will require the development of various types of coarse-grained overlays and automated mapping techniques. We have identified several challenges associated with the development of high performance FPGA overlays which can only be overcome by thorough investigation of associated overheads (mainly area and performance overheads) and efficient automated mapping techniques. We will then see how we can expose and effectively exploit the massively parallel architecture provided by the programmable overlay built on top of the FPGA fabric.

1.2 Objectives

The high level objective of this thesis is to develop area and performance efficient FPGA overlays to be deployed in a heterogeneous computing platform as a high performance programmable accelerator which can allow rapid compilation and fast context switching while improving accelerator design productivity. The main objectives are as follows:

- Classify FPGA overlays (existing in the literature) based on their architecture and execution style and identify performance metrics.
- Evaluate open source FPGA overlays in terms of area and performance overheads and develop techniques to improve the performance metrics.
- Demonstrate how the Xilinx DSP48E1 primitive can be used, at near to its theoretical limits, as a programmable PE to build an efficient overlay architecture for pipelined execution of compute kernels.
- Design a flexible and scalable interconnect architecture for an array of DSP block based fully pipelined FUs and evaluate the area overheads of the interconnect on a commercial FPGA device.
- Explore the possibility of developing larger, more efficient, overlays using multiple DSP blocks within each FU, maximising utilisation by mapping multiple instances of kernels simultaneously onto the overlay to exploit kernel level parallelism.
- Develop novel interconnect architectures for exploring area overheads versus flexibility trade-offs, while still allowing maximum kernel throughput.
- Develop methodologies for fast compilation of high level language (HLL) description of compute kernels onto developed overlays to raise the level of programming abstraction and improving design productivity.

Since the focus of the thesis is to improve the performance of the overlays while reducing the area overheads, the scope of this thesis does not include issues such as the design of high speed communication interfaces for the overlays.

1.3 Contributions

The main contributions of this thesis are novel architectures and tools with focus on developing area and performance efficient overlays while reducing interconnect area overheads and improving peak performance. The contributions are as follows:

- Design and implementation of an improved FU architecture using the flexibility of the DSP48E1 primitive which results in a 2.5 times frequency improvement and 25% area reduction compared to the original FU architecture. Our adapted version of a 6x6 16-bit DySER was implemented on a Xilinx Zynq by using a DSP block as the compute logic, referred to as DSP-DySER, providing a peak performance of 6.3 GOPS with an interconnect area overhead of 7.6K LUTs/GOPS.

- Design and implementation of a more FPGA targeted overlay architecture that maximizes the peak performance and reduces the interconnect area overhead. This is achieved through the use of an array of DSP block based fully pipelined FUs and an island-style coarse-grained routing network, resulting in a peak performance of 65 GOPS (10x better than DSP-DySER) with an interconnect area overhead of 430 LUTs/GOPS (18x better than DSP-DySER).

- Experiments to explore the possibility of developing larger, more efficient, overlays using multiple DSP blocks within each FU, maximising utilisation by mapping multiple instances of kernels simultaneously onto the overlay to exploit kernel level parallelism. By utilizing two DSP blocks within each FU, we show a significant improvement in achievable overlay size and overlay utilisation, with a reduction of almost 70% in the overlay tile requirement compared to existing overlay architectures, an operating frequency in excess of 300 MHz, and a peak performance of 115 GOPS with an interconnect area overhead of 320 LUTs/GOPS.

- DeCO, a cone shaped cluster of FUs utilizing a simple linear interconnect which reduces the area overheads for implementing compute kernels extracted from compute-intensive applications represented as directed acyclic dataflow graphs, while still allowing high data throughput. The proposed overlay exhibits an achievable frequency of 395 MHz, close to the DSP theoretical limit on the Xilinx Zynq, achieving a peak performance of 260 GOPS with an interconnect area overhead of just 60 LUTs/GOPS.

- A methodology for compiling high level language (C/OpenCL) descriptions of compute kernels onto DSP block based coarse-grained overlays, rather than directly to the FPGA fabric. Our mapping flow provides a rapid, vendor independent mapping to the overlay, raising the abstraction level while also reducing compile times significantly, hence addressing the design productivity issue.

1.4 Thesis Organization

The thesis is organised as follows. Chapter 2 discusses programmable accelerators such as GPUs, MPPAs and FPGAs, followed by a detailed description of FPGA hardware abstraction in the context of reconfigurable computing and research performed in academia and industry on abstraction techniques for reconfigurable computing. It then reviews various architectures and tools proposed in the area of FPGA Overlays by presenting a classification based on the architecture and execution style of overlays. Chapter 3 presentes the evaluation of an open source overlay architecture, DySER, mapped on the Xilinx Zynq device and shows that DySER suffers from a significant area and performance overhead due to the limited consideration for the underlying FPGA architecture. Chapter 4 presents a more FPGA targeted overlay architecture that maximizes the peak performance and reduces the interconnect area overhead through the use of an array of DSP block based fully pipelined FUs and an island-style coarse-grained routing network. Chapter 5 presents DeCO, a cone shaped cluster of FUs utilizing a simple linear interconnect, to reduce the area overheads for implementing compute kernels extracted from compute-intensive applications represented as directed acyclic dataflow graphs, while still allowing high data throughput. Chapter 6 presents a methodology for compiling high level language (C/OpenCL) descriptions of compute kernels onto DSP block based coarse-grained overlays, rather than directly to the FPGA fabric. Finally chapter 7 concludes our contributions and outlines future research directions.

1.5 Publications

The work presented in this thesis serves as a basis for manuscripts that have been published in peer-reviewed journals and conference proceedings.

- 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*, in ACM SIGARCH Computer Architecture News (CAN), vol. 43, no. 4, pp. 28-33, September 2015. [77]

- A. K. Jain, S. A. Fahmy, and D. L. Maskell. *Efficient Overlay Architecture Based on DSP Blocks*, in Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM), Vancouver, Canada, May 2015. [78]

- A. K. Jain, D. L. Maskell, and S. A. Fahmy. *Throughput Oriented FPGA Overlays Using DSP Blocks*, in Proceedings of the Design, Automation and Test in Europe Conference (DATE), Dresden, Germany, March 2016. [79]

- 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 IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM), Washington DC, USA, May 2016. [80]

- A. K. Jain, D. L. Maskell, and S. A. Fahmy. *Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs?*, in Proceedings of the IEEE International Conference on Pervasive Intelligence and Computing, Auckland, New Zealand, August 2016. [81]

- 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 Proceedings of the Second International Workshop on Overlay Architectures for FPGAs (OLAF), Monterey, CA, USA, Feb 2016. [82]

- K. D. Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell. *Microkernel Hypervisor for a Hybrid ARM-FPGA Platform*, in Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), Washington, DC, June 2013, pp. 219-226. [83]

- 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*, in Journal of Signal Processing Systems (JSPS), vol. 77, no. 1-2, pp. 6176, October 2014, Springer. [84]

2

Background and Literature Review

In this chapter, we first discuss the role of FPGAs in heterogeneous computing platforms followed by FPGA-based accelerator design process. We then discuss several efforts from academia and industry to simplify the accelerator design process, specifically by raising the level of programming abstraction and by providing efficient communication interfaces and runtime management techniques for FPGA-based accelerators. We then discuss some of the key barriers to mainstream use of FPGAs as a rapidly reconfigurable accelerator, mainly reconfiguration latency and fine granularity. Finally, we discuss about coarse-grained reconfigurable devices and coarse-grained FPGA overlays.

2.1 FPGAs in Heterogeneous Computing Platforms

Traditional homogeneous computing platforms perform all their computations using general purpose processors that typically follow a Von Neumann organisation. Hence, they spend most of the time on non-computational tasks (handling data movements, fetching and decoding instructions), with significant power spent on non-computational units [85]. As technology limits have prevented further increases in clock frequency, the performance gains achieved by adding more cores come at the cost of rapidly scaling complexity in inter-core communication, memory coherency and, hence, power consumption. The strong need for increased computational performance and energy efficiency has led to the use of heterogeneous computing platforms that combine traditional CPUs with additional compute architectures better able to accelerate computationally intensive tasks [86]. Examples are to use GPUs, MPPAs, floating-point units (FPUs), and cryptographic-processing units as co-processors to the host processor. These co-processing components usually incorporate specialized processing capabilities to handle particular tasks. Complex functionality is also sacrificed, disabling their ability to run operating systems, and they are typically managed by traditional CPUs to offload compute-intensive parts of applications.

The use of accelerator architectures in heterogeneous computing platforms offers a promising path towards improved performance and energy efficiency. One class of solution includes programmable accelerators such as GPUs and MPPAs. Applications on MPPAs [3, 4, 5, 6] can achieve better energy efficiency relative to conventional processors and GPUs. Another class of solution dedicates highly efficient custom-designed application-specific accelerator for computing tasks. This approach was preferable due to area, speed, and energy efficiency [7, 8] and these were deployed as ASIC blocks alongside a GPP [9, 10]. However, developing dedicated ASIC accelerators has become less practical due to the long turnaround time and high cost associated with ASIC development, as well as the rapidly evolving application space requiring flexibility.

FPGAs, which allow the hosted accelerator to be modified post-deployment [11, 12, 13], are now more commonly used for rapid-prototyping of application specific accelerators in heterogeneous computing platforms. FPGAs offer significant advantages in terms of sharing hardware between distinct isolated tasks, under tight time constraints. Historically, reconfigurable resources available in FPGA fabrics have been used to build high performance accelerators in specific domains, such as communications and signal processing. These application domains have driven the development of reconfigurable resources, from relatively simple modules to highly parametrizable and configurable subsystems. While FPGAs started out as a matrix of programmable processing elements, called configurable logic blocks (CLBs) connected by programmable interconnect to configurable I/O, they have evolved to also include a variety of processing macros, such as reconfigurable embedded memories and DSP blocks to improve the efficiency of FPGA based accelerators.

One major concern in using FPGAs or FPGA-like reconfigurable fabrics within a heterogeneous computing platform is data transfer bandwidth. The efficiency of data transport depends on the level (rank) of coupling between the processor and the reconfigurable fabric as discussed in [19] and shown in Figure 2.1.

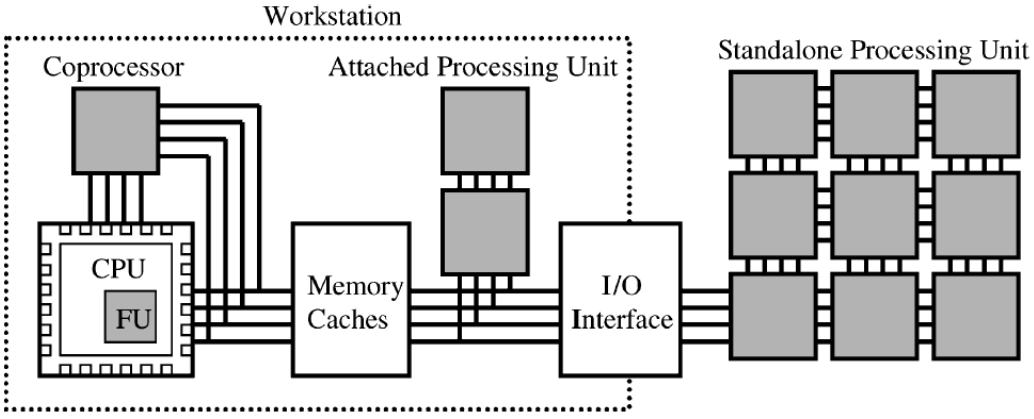


Figure 2.1: Different levels of coupling for reconfigurable fabric in a heterogeneous platform.

CHIMEARA [87] is an example of rank one coupling in which the reconfigurable fabric is integrated within the processor micro-architecture. Garp [69] is an examples of rank two coupling in which the reconfigurable fabric acts as a co-processor

which is able to perform computations without intervention of the main processor. PCI-PipeRench [88] is an example of rank three coupling, similar to multiprocessor systems, where the reconfigurable fabric is not connected directly with the Cache. Finally Virtual Wires [89] is an example of rank four coupling in which emulation is the main purpose. Efficient integration of the reconfigurable fabric to the processor (lower the rank of the coupling) allows more frequent use due to low latency and high bandwidth data transport. To address possible bottleneck problems, particularly in providing high bandwidth transfers between the CPU and the commercial FPGA fabrics, a number of vendor specific systems with integrated hard processors [56] have been proposed, referred to as hybrid FPGA platforms. Such approaches mean FPGAs can be used in a much wider range of applications. However, in order to make better use of FPGA resources, virtualization is an important issue to address.

In order to virtualize FPGA resources in hybrid FPGA platforms, it is necessary to first understand how an application is typically implemented. The process starts with deciding which parts should execute on the processor and the FPGA. This is called hardware-software partitioning. The part that executes on the FPGA is referred to as the hardware task and the one that executes on the processor is the software task. The software task is then described in a high level language (HLL) such as, C/C++ and the hardware task in a hardware description language (HDL) like Verilog or VHDL. After compiling the software and hardware tasks the software machine code must be loaded into processor memory and the hardware configuration bitstream must be loaded into the configuration memory of the FPGA. The configuration memory controls the low level features of the FPGA fabric, determining the logic contents of the LUTs, how primitives are connected, and which features are used. Within the software code, specific calls must be made to the interfaces that couple the hardware portion with the processor. And the hardware system must understand how to communicate using the same protocol. Both data transfer and control occur over this hardware-software interface. As more and more complex systems are being implemented on reconfigurable

hardware, a significant amount of research effort is being focused on the following aspects:

- How to abstract FPGA resources so that users do not need to deal with low level details.
- How to describe hardware tasks at a higher level of abstraction to improve design and verification.
- How to manage multiple tasks running simultaneously contending for access to FPGA resources.

In order to answer these research questions and to develop a virtualized platform framework, researchers are exploring the following key components:

- Abstracted computation and communication architectures that support time multiplexing of hardware and fast context switching so that resources can be shared.
- Tools that support the fast compilation of a high-level description of an application onto hardware to support programming abstraction.
- Runtime systems which support task management in an isolated manner to support management abstraction.

2.1.1 Raising the Level of Programming Abstraction

Virtualization of FPGA resources in hybrid FPGA platforms is motivated by the fact that there is a dramatic growth in the functionality and performance of these platforms. The rich functionality of these modern platforms, together with the integration of reconfigurable logic, is creating a demand for efficient use of reconfigurable hardware by employing virtualization techniques. However, the use of FPGAs in modern platforms remains constrained within specialist application domains, such as digital signal processing and communications. FPGA accelerators are normally designed at a low level of abstraction (typically RTL) in order to

obtain an efficient implementation, and this can consume more time and make reuse difficult when compared to a similar software design. To build an FPGA accelerator, designers typically start by manually converting the compute kernel into an fully pipelined datapath, specified using a hardware description language (HDL) such as Verilog or VHDL. The designer must specify the detailed structure of the datapath and must also define control for reading inputs from memories into buffers, stalling the datapath when buffers are full or empty, writing outputs to memory, and so on. This description is at the level of individual bits and clock cycles.

For a typical FPGA device, a fully pipelined datapath implementing just several lines of C code may require 2–3 orders of magnitude more lines of HDL code, but can achieve significantly better performance by pipelining and exploiting parallelism. However this performance comes at the cost of significant design effort. Hence accelerator design is a complex process, requiring low-level hardware device expertise and specialist knowledge of both hardware and software systems, resulting in major design productivity issues. HLS tools [29, 30, 31, 32, 33] have helped simplify accelerator design by raising the level of programming abstraction from RTL to high level languages, such as C or C++. These tools allow the functionality of an accelerator to be described at a higher level to reduce developer effort, enable design portability, and enable rapid design space exploration, thereby improving productivity, verifiability, and flexibility.

Raising the level of programming abstraction reduces the amount of information required to describe the functionality of an accelerator, typically with a marginal area and performance cost. For example, Bluespec [90] abstracts interface methods for control and concurrency, although it still describes cycle-level behavior of resources. Higher level tools use languages such as C or C++ where timing is no longer explicit. Vivado HLS is a commercial HLS tool, from Xilinx, which is normally used to generate application specific IP from algorithmic C specifications. It not only generates RTL code but also provides an environment for automated functional verification. It allows rapid design space exploration by rapidly generating different designs for different design choices.

LegUp is an open source academic HLS framework (built on top of LLVM compiler infrastructure) which can identify parts of the application suitable for running on FPGA hardware from a high level description (C) and then generate RTL for them. The LegUp framework consists of an HLS tool flow, a MIPS processor, a hardware profiler, an environment for automated functional verification and some benchmarks. It doesn't support recursion, dynamic memory, or floating point arithmetic. It uses as soon as possible (ASAP) scheduling and weighted bipartite matching for binding. ROCCC is a C to synthesizable VHDL compiler, developed at the University of California, Riverside, and specifically focused on FPGA based code acceleration from a subset of the C language. In order to make use of generated hardware, this compiler exports C functions which can be included in application program. These C functions handle all communications with the generated hardware.

2.1.2 Communication Interfaces and Runtime Management

Some of the latest HLS tools, such as Xilinx SDSoC, abstract away the details of communication between the accelerator and the host processor by automatically generating system level hardware and software drivers. Xillybus is another solution for data transport between processor and the accelerator which can be used to abstract the communication interface. It is designed to work with a number of interfaces: the PCIe interface (in a typical x86 based system) and the AXI interface (in an ARM based system), as the underlying transport mechanism by providing Standard FIFOs as interfaces to the application logic. Each FIFO stream is mapped to a device file by Xillybus driver. It provides most of the necessary communication interfaces such as memory mapped to stream interface, memory mapped to memory mapped interface and memory mapped register interface.

SIRC [91] was proposed as an open source abstract interface (a software API and hardware interface) for communication between host processor and the FPGA accelerator. It works on basic principle of hardware-software communication which is as follow: first SW sends data to local buffer in FPGA and triggers the user

logic to get this data from local buffer and put it in another local buffer after processing. After finishing its processing user logic will notify the SW to get the processed data from local buffers. SIRC uses Ethernet to communicate between a Windows based workstation and Xilinx FPGA board. User does not need the knowledge of communication protocol, OS or proprietary drivers. RIFFA was proposed as an open source reusable framework to integrate FPGA based IP cores with the host processor over PCIe interface [92]. This framework requires a PCIe bus enabled workstation and a FPGA board with a PCIe peripheral. On the SW side, it provides a PCIe Linux device driver and SW libraries and on the FPGA side, PCIe endpoint translates requests, coming from the host processor, to PLB requests via address translation.

CoRAM was proposed as a data-transport mechanism using a shared and scalable memory architecture[93]. It assumes that the FPGA is connected directly to L2 cache or memory interconnects via memory interfaces at the boundaries of the recongurable fabric. The requirement for using CoRAM architecture is that user logic is not allowed to access off-chip I/O pins and memory interfaces in order to provide application portability. Application can only interact with the external environment through a collection of specialized distributed SRAMs called CoRAMs that provide on-chip storage for application data. Control thread (a state machine) is used to inform user logic when the data within specific CoRAMs are ready to be accessed through locally addressed SRAM interfaces. Control thread can be expressed in a high level language such as C which can be translated to a state machine to provide high level of management abstraction or it can be compiled and executed on multithreaded processing core. In short control thread can be soft or hard. In order to evaluate a hypothetical FPGA with CoRAM architecture support, authors have used a cycle accurate software simulator written in bluspec system verilog (BSV). LEAP[94] (logic-based environment for application programming) scratchpad was proposed as an automatic memory management system to make reconfigurable memory hierarchy invisible which is very similar to the concept of CoRAM. Both of these projects share the objective of providing a

standard memory abstraction by virtualizing the FPGA memory and I/O interfaces. LEAP abstracts away the details of memory management by exporting a set of interfaces to local client address spaces.

A number of researchers have focused on providing operating system (OS) support for FPGA hardware so as to provide a simple programming model to the user and effective run-time scheduling of hardware and software tasks [45, 53, 95, 55]. Several operating systems have been developed for FPGA hardware [53, 96, 45, 97, 54, 98, 99, 100, 101, 102]. Several Linux extensions have also been proposed to support FPGA hardware [54, 99, 102, 100]. ReconOS [99] provides an execution environment by extending a multi-threaded programming model from software to reconfigurable hardware. RAMPSoCVM [102] provides runtime support and hardware virtualization for the MPSoC through APIs added to Embedded Linux to provide a standard message passing interface.

Even though efforts, such as Xilinx SDSoC, SIRC, RIFFA, Xillybus, CoRAM, LEAP and ReconOS, have abstracted the communication interfaces and memory management, allowing designers to focus on high level functionality instead of low-level implementation details, the prohibitive compilation times (specifically the place and route times in the backend flow for generating the FPGA implementation of the accelerator) have largely been ignored. Place and route time is now a major productivity bottleneck that prevents designers from using mainstream design and debug methodologies based on rapid compilation. As such, most of the existing techniques are generally limited to static reconfigurable systems [34].

2.2 Key Barriers to Mainstream Use of FPGAs

To better understand why FPGA devices have not achieved mainstream adoption as a rapidly reconfigurable hardware accelerator among the wider computing community, we must first understand how FPGAs differ from alternative solutions, specifically traditional GPPs. The most fundamental difference relates to how an application is mapped to the these platforms. A GPP provides functionality to

execute a compute kernel as a list of sequential instructions, whereas an FPGA architecture implements compute kernels by mapping them to fine grained resources, such as configurable logic blocks, and medium grained hard DSP blocks, Block RAMs, etc. These resources are interconnected via a fine-grained programmable island-style routing network to create a specialized datapath which implements the compute kernel. By exploiting parallelism in the algorithm, significant performance gains are possible. The following describes the key barriers to mainstream use of FPGAs for hardware acceleration.

The FPGA fabric, being programmable, is able to adapt to changing processing requirements, thus better utilising FPGA resources, while providing a more software centric approach to hardware design. This allows software applications to be profiled and partitioned, with the resulting hardware accelerator running on the FPGA fabric and the remaining software running on the GPP, with significant performance improvements. These accelerators can also be rapidly reconfigured by utilizing the ability to partially and dynamically reconfigure the functionality of the FPGA fabric. However, despite the popularity and inherent capability of FPGAs for partial reconfiguration, this feature is not well supported by FPGA vendors and is hampered by slow reconfiguration times, poor CAD tool support, and large configuration file sizes. These issues make dynamic reconfiguration difficult and inefficient, resulting in most FPGAs being used with just a single configuration.

Initial implementations of dynamic reconfiguration [39, 40] required the reconfiguration of the whole hardware fabric. This resulted in significant configuration overhead, which severely limited its usefulness. Xilinx introduced the concept of dynamic partial reconfiguration (DPR) which reduced configuration time by allowing a smaller region of the fabric to be dynamically reconfigured at runtime. The concept of DPR on FPGA is one way of virtualizing hardware to allow implementation of applications that are larger than the FPGA. DPR significantly improved reconfiguration performance [41], however the efficiency of the traditional design approach for DPR is heavily impacted by how a design is partitioned and floorplanned [42], tasks that require FPGA expertise. Furthermore, the commonly used configuration mechanism is highly sub-optimal in terms of throughput [44].

Despite numerous efforts in reducing reconfiguration times and improving CAD tool support for dynamic reconfiguration of FPGA fabric [103, 104], the implementation of rapidly reconfigurable hardware accelerators is still difficult.

In addition, a design for a reconfigurable device does not necessarily port well to the next hardware generation, making reconfigurable systems more difficult to work with. The designer must make a number of decisions, such as how to best fit the application to the device, including the datapath structure and the amount of parallelism. Applications are normally optimized for a specific target device, and are unable to execute on a smaller device or cannot take full advantage of the additional resources on a larger device. Once the designer has a working design it must be implemented on the FPGA. The FPGA tool flow typically takes an RTL description of the design and first performs technology mapping to convert it into the fine-grained device resources, followed by placement and routing (PAR). Due to the fine granularity of the FPGA resources, this process is complex and for large designs results in very lengthy place and route times.

The complexity in the FPGA tool flow due to the use of fine-grained FPGA resources can be easily demonstrated by an example. Figure 2.2(a) shows the placed and routed design of a simple 4 input 16-bit adder. Here, the FPGA design tools divide the design into basic circuit elements and map them to the fine-grained configurable logic blocks (CLBs). On the other hand, Figure 2.2(b) shows the placed and routed design of the same application on a coarse-grained architecture where compute blocks (or functional units (FU)) and interconnect have a 16-bit width, compared to single-bit tracks in the fine-grained FPGA implementation. It is clear that the PAR complexity is significantly reduced by using coarse-grained architectures, thus reducing compilation time. Another benefit of using a coarse-grained architecture is the reduced configuration data size and hence reduced reconfiguration latency which can allow faster context switching.

Because of this apparent advantage, researchers have explored a number of ASIC implementations of coarse-grained reconfigurable architectures (CGRAs) [105, 106,

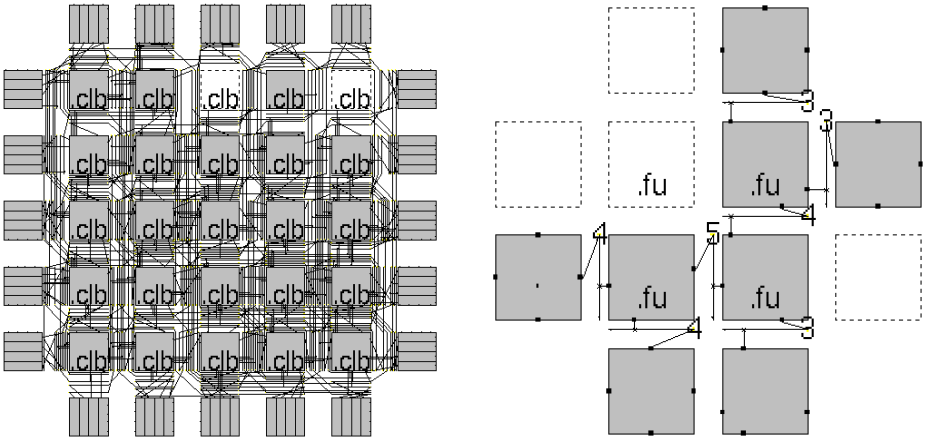


Figure 2.2: Placement and routing on (a) fine-grained (b) coarse-grained architecture.

107, 69, 64, 65, 66, 51, 70, 67, 68, 71]. Some key features that enabled these architectures to address signal processing and high performance computing problems more efficiently include: energy efficiency, ease of programming, fast compilation and reconfiguration. A review of these architectures is given in the next section.

2.3 Coarse-Grained Reconfigurable Architectures

The Rapid [65] architecture was designed to implement computation-intensive and highly regular systolic streaming applications using an array of computing cells, where each cell consists of an multiplier, two ALUs, six general purpose registers and three small local memories. Performance of Rapid was measured as 1.6 GOPS, where an operation was a single MAC operation. Key features are small local memory, word based interconnect, NN inter-cell communication, suitability for deep pipelines, streaming FIFO based communication between external memory and array, custom array generation, static control and wide micro-instruction based dynamic control, Simulated annealing based placement and Pathfinder based routing. Automatic creation of custom arrays was explored using the Rapid framework in [108] by developing algorithms for reducing area overhead in custom array generation.

Morphosys[66] was proposed as a coarse-grain, integrated reconfigurable SoC targeted at high throughput applications such as multimedia and image processing. It consists of a processing unit called Tiny RISC processor core, a reconfigurable cell array (8×8 array), context memory, frame buffer and a DMA controller. The Tiny RISC processor core has an extended instruction set for effectively controlling the array operations. Key features are context memory, multi-context support, DMA support, streaming data buffers, hierarchical bus network, 2D mesh of functional units. Morphosys design flow is supported through a GUI called mView that takes user input for each application. mView generates assembly code for the array and has several built in features that allow visualization of array execution, interconnect usage patterns for different applications, and single-step simulation runs with backward, forward and continuous execution.

REMARC was proposed for multimedia applications and consists of a MIPS ISA based core and an 8×8 reconfigurable logic array[109]. Each processing element of the array consists of a 16-bit processor and execution of each processor was controlled by instructions stored in small local instruction memory. Programming abstraction was supported using REMARC assemblers. Authors explored this coarse grained array with VLIW control for multimedia applications using developed simulator and showed that it can be more compact than an FPGA accelerator achieving comparable performance.

Virtual Mobile Engine (VME) was proposed as a dynamically reconfigurable architecture deployed in consumer electronic products, Sony PSP and network walkman, for low power and programmability [110].

The concept of virtual embedded blocks (VEBs)[111] was proposed as a model to explore hard logic integration in an FPGA array. Authors proposed a methodology to study the effect of embedding floating point coarse grained units in FPGAs. Authors have shown that embedding coarse grained floating point units in FPGAs can result in 3.7× area reduction and 4.4× speedup.

[112] examined register file architectures for coarse-grained recongurable architectures. Issues and approaches to CGRA development were explored in [113] by implementing encryption-specialized CGRA. It was shown that flexibility and adaptability reduces as we increase the granularity of the functional units. Difficulty of finding optimum number of functional units for domain specialized architecture and problem of allocating hardware resources required by the algorithm was mentioned and techniques were proposed to allocate functional units to balance performance and area constraints.

The Chameleon system was proposed as heterogeneous processing tiles connected to each other by a network on chip. In CHAMELEON SoC, Montium architecture[70] was integrated with a processor and fine-grained reconfigurable logic in order to provide power efficient execution. The design methodology for the Chameleon SoC is based on the Kahn process network model. Montium consists of multiple processor tiles each consists 5 combinational ALUs, 10 local memories and a communication and configuration unit.

Zippy [114] approach aims at the investigation of a hybrid processor consisting of an embedded CPU and a coarse-grained reconfigurable array. Authors have explored temporal partitioning of netlists at coarse grain level and reported that resource requirement can be reduced and resource-constrained platform can be used to execute large applications using temporal partitioning followed by multi-context execution[115]. The Scalability problem was described as how to run a scaled version of an algorithm on a resource-constrained architecture. Cycle accurate CPU model written in C was extended with a coprocessor interface and was used to control multi-context homogeneous CGRA (4×4) specified in VHDL. Multi-context execution of FIR filters was shown using Modelsim based simulation framework. Authors have developed PAR tools for mapping applications (described in graph form and converted to a textual description in Zippy Netlist format (ZNF)) on reconfigurable array. A configuration generator was used to generate array configurations from placed and routed netlists. Authors also presented a brief survey on virtualization of hardware [116] and compared virtualized

and non-virtualized execution of ADPCM decoder on the developed simulation environment [114].

The Smartcell architecture was proposed for streaming applications to bridge the energy efficiency gap between FPGAs and customized ASICs. Key features are runtime partial reconfiguration, support for SIMD, MIMD and systolic style computations, cycle-by-cycle FU reconfiguration. It also supports three-level layered on-chip communication and two levels of pipelining, including instruction level pipelining and task level pipelining.

The key attraction of coarse-grained reconfigurable devices is their near ASIC-like computational and energy efficiency and software-like engineering efficiency. At least for commercial products, the main market has been as a component in SoCs for efficiently implementing a specific range of DSP functions as part of a larger system. CGRAs have not been successfully developed as stand-alone systems that designers can incorporate at the board level. This is because functional units are often too application specific to be efficient and useful for a wide range of applications. ASIC implementations of coarse-grain architectures also suffer from the design-time freeze of functional units and interconnect capabilities. It is hard to find a particular configuration that suits a wide enough set of applications for the approach to be viable as a stand-alone product. Hence there is a need for a mechanism where capabilities can be tailored to applications or adapted at runtime based on application needs.

2.4 Coarse-Grained FPGA Overlays

One solution that has been explored extensively by researchers is to implement a coarse-grained reconfigurable architecture on top a commercial FPGA device, referred to as a coarse-grained overlay. This allows the coarse-grained elements and structure, specifically the FU and interconnect to be changed at runtime as per the application requirements. Compared to a conventional coarse-grained reconfigurable architecture on an SoC device, using an FPGA to implement a

coarse-grained architecture has several potential advantages. These include: improved designer productivity, better design portability, software-like programmability, faster application switching and enhanced security. This is motivated by the fact that programs can be written at a higher level of abstraction with compilation to the overlay being several orders of magnitude faster than for the fine grained FPGA on which the overlay is implemented. That is, instead of the requirement for a full cycle through the FPGA vendor tools, overlay architectures present a simpler problem, that of programming an interconnected array of FUs. However, overlays are not intended to replace HLS tools and vendor-implementation tools. Instead intended to support FPGA usage models where programmability, abstraction, resource sharing, fast compilation and design productivity are critical issues.

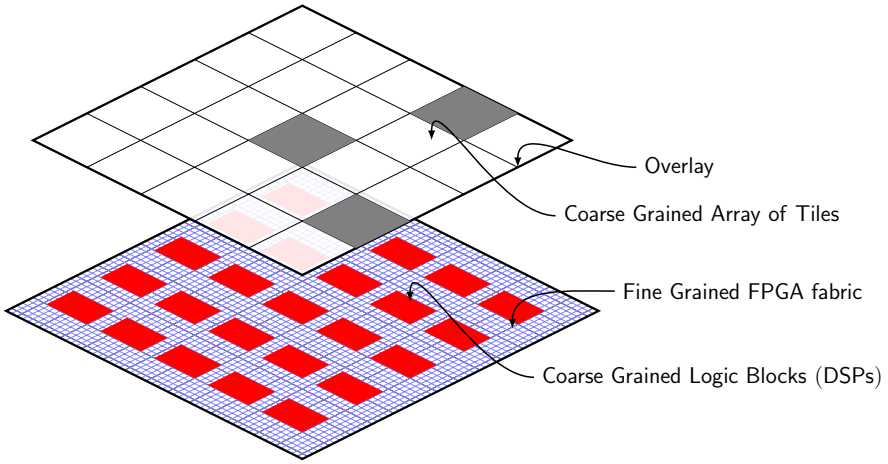


Figure 2.3: Coarse-grained FPGA overlay architecture.

FPGA overlays apply the concept of hardware abstraction to the domain of reconfigurable computing [117], simplifying application development by abstracting away details from the physical resources. FPGA hardware abstraction decouples system properties, such as scalability, reliability and isolation, from implementation details, thereby reducing developer effort and concern for device specific details. However, hiding device specific details and the underlying functionality of the resource can mean there is less control over optimization [75]. Within the reconfigurable computing community, there has been some research effort to abstract FPGA hardware from the user. CoRAM [93], LEAP [94], VirtualRC [118], SIRC[91] and RIFFA[119] are examples of approaches to abstract memory and communication interfaces in FPGA devices. Overlays such as QUKU [120],

DySER [75] and intermediate fabrics [73] are examples of approaches to abstract compute logic in FPGA devices. One motivation for using FPGA overlays for hardware abstraction aspect is to be able to execute several applications on limited hardware resources in a time multiplexed manner or to be able to execute an application of arbitrary size on a single device and allows the strategic reuse of resources used by the overlay. Both fine- and coarse-grained overlay architectures have been proposed to abstract FPGA fabric resources.

A fine-grained FPGA overlay (that is FPGA-on-an-FPGA) provides a non-vendor specific fine-grained architecture which is overlaid on top of an existing FPGA device [121, 122, 123]. This provides bitstream portability between different vendors and devices as well as compatibility with open-source FPGA tool flows. However, there are significant drawbacks due to the fine granularity of the virtual LUTs, switch boxes and connection boxes, which results in long compilation times, large configuration file sizes and large area and performance overheads. As such, fine-grained overlays are only viable if circuit portability is of paramount importance [121] due to the large overheads. Performance overheads showing a 100× increase in area and a 6× decrease in circuit performance was reported [121]. Subsequent architectures (ZUMA [122]) have reduced the area overhead to 40× through careful architectural choices, such as using LUTRAM to store the configuration data for the circuit to be mapped on the overlay. An analysis of a fine-grained overlay [122] embedded within the Xilinx Zynq SoPC platform was presented in [117].

Unlike fine-grained overlays which are programmable at the bit level, coarse-grained overlays are programmable at the data-word/operator level. This makes the design problem much simpler, resulting in significantly faster compilation. Programmable coarse-grained overlays range from simple soft-core processors [124] and multi-ALU vector processors [125], to arrays of simple ALU-based processing elements [75] and arrays of soft processors [126], where both the computational units (the FUs) and the interconnect are programmable.

Soft-processors are a coarse-grained overlay, as conceptually these are GPPs implemented on top of the FPGA fabric, making the system easier to program and more familiar to software designers [124]. While many soft processors have been proposed, they typically exhibit a significant performance gap compared to a traditional parallel and pipelined hardware implementation. Soft vector processors [127], such as VESPA [128], VEGAS [127], VIPERS [129], VENICE [130] and MXP [125] try to reduce the performance gap by utilizing multiple ALUs. While soft vector processors can exploit the parallelism in compute kernels having multilane patterns to achieve a significant speed-up, they do not perform well for compute kernels having reduction patterns. These patterns are explained in more detail in [131].

At the other end are the coarse-grained array-based overlays, similar to CGRAs, which consist of an array of FUs interconnected using a programmable network. Conceptually, an FU can be as simple as an arithmetic block (or simple ALU) or as complicated as a soft-processor. Most of the FU-array overlays are essentially dataflow machines, mainly used in systems as co-processors for task acceleration. One possibility for communicating among FUs (ALUs/soft-processors) is to use a bus-based architecture. However, this solution is unable to sustain the communication load among hundreds of FUs and eventually impacts system performance drastically. Hence most of the FU-array overlays adopt a scalable interconnect architectures built using FPGA hardware resources. These array-based overlays can be categorised depending on their architecture using the classification of [132], where 4 different categories are defined as: spatially configured, time multiplexed, packet switched, and circuit switched, as shown in Figure 2.4.

While examples of packet switched networks [132, 146, 147, 148] and circuit switched networks [149] in FPGA exist, they are generally very resource hungry, and are unsuitable for large FPGA-based overlay architectures. For example, in the case of packet switched networks, the CMU CONNECT [146] router consumes 1.5K LUTs with a critical path of 9.6 ns, while the Penn Split-merge [147] router consumes 1.7K LUTs with a critical path of 4.5 ns. Recently a light weight packet switched network using Hoplite switches [148, 150] was demonstrated which

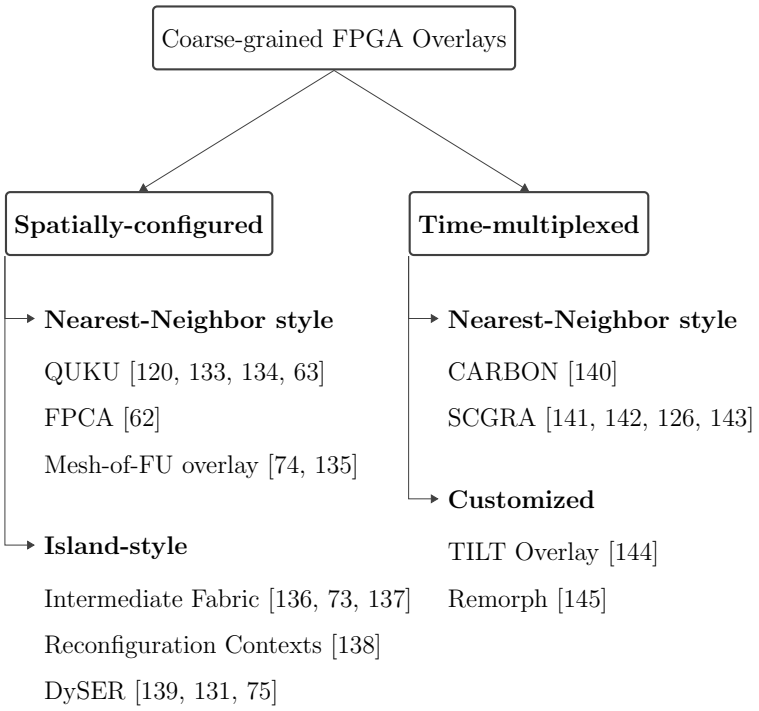


Figure 2.4: Categorization of coarse-grained FPGA overlays.

consumed just 60 LUTs with a critical path of 2.9 ns, opening the way for efficient packet-switched arrays of processing elements on FPGA.

As seen in Figure 2.4, most array-based overlays are restricted to just two classes: spatially configured overlays; and time-multiplexed overlays, where both the FU and the interconnect can fall within one of these two categories. These two categories of overlays will be described in more detail in the next two sections.

2.5 Time-multiplexed Coarse-Grained Overlays

A fully pipelined, spatially configured overlay can deliver maximum performance by executing one computation iteration every clock cycle (that is it has an II of one), but with a large FPGA resource overhead. Alternatively, 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 (TM) overlays have both a time-multiplexed FU and a time-multiplexed interconnect network [141, 144, 140, 145], which we refer to as TMFU-TMN. Time-multiplexing the overlay allows it to change its behavior on a cycle by cycle basis while the compute kernel is executing [141, 144, 140, 145], thus allowing the sharing of the limited FPGA resource for other purposes.

In a TM overlay, FUs generally behave like a conventional processor core, with instruction memory embedded within each FU. However, in many cases the storage requirements for the set of instructions is very large which results in a significant area overhead. This is mainly due to the choice of the scheduling strategy; the execution model; and the design of overlay architecture, which limits the scalability of the overlay and also impacts the kernel context switch time. One major benefit of using these type of overlays is that well established algorithms and tools can be used for application mapping. The algorithms commonly used to schedule the kernel operations to the array of FUs are List scheduling [151], Force Directed Scheduling [152] and Modulo Scheduling [153]. Again, as with SC overlays, many of the TM overlays (which we discuss next) are not architecture-focused and hence the FU operates at a relatively slow frequency.

2.5.1 Nearest-neighbor Style Interconnect Based

CARBON: One example is CARBON [140], which was implemented as a 2×2 array of tiles on an Altera Stratix III FPGA. Each tile contains an FU consisting of a programmable ALU and instruction memory, supporting a maximum of 256 instructions. An FU consumed 3K ALMs, 304 FFs, 15.6K BRAM bits and 4 DSP blocks, achieving an operating frequency of 90 MHz. Compared to the other TMFU-TMN overlays discussed here, CARBON has large resource requirements per FU with a relatively slow speed which limits the scalability of the architecture.

SCGRA: The SCGRA overlay [141] was proposed to address the FPGA design

productivity issue, demonstrating a 10-100× reduction in compilation time compared to the AutoESL HLS tool. An automatic nested loop acceleration framework, targeting a CPU-FPGA system using the SCGRA overlay was also developed [126]. Application specific SCGRA overlays were subsequently implemented on the Xilinx Zynq platform [143], achieving a speedup of up to 9× compared to the same application running on the Zynq ARM processor. The FU used in the Zynq based SCGRA overlay operates at 250 MHz and consists of an ALU, multiport data memory (256×32 bits) and a customizable depth instruction ROM (Supporting 72-bit wide instructions) which results in the excessive utilization of BRAMs. Authors have designed the FU in such a way that it can fetch configuration, from instruction ROM, every cycle. Configuration defines the behavior of the FU for a particular clock cycle. They have shown a way to generate DFG specific interconnect architecture, using genetic algorithm, instead of using a generalized and fixed interconnect architecture, such as Ring, Torus and fully connected, for all DFGs.

ArchSyn is a high level synthesis tool which extracts data flow graph from a C program and then map it to SCGRA architecture by scheduling the data flow operations. The scheduler not only determines the computation schedule of operations onto PEs but also determines the communication schedule of the intermediate data among PEs. Authors have formulated the problem of mapping DFG to the SCGRA into a precedence and resource constrained scheduling problem using a time indexed integer linear programming (ILP). The optimization goal of this tool is not only to minimize compute latency but also to minimize energy consumption in computation and communication.

Authors proposed accelerator design methodology that utilizes SCGRA as an intermediate compilation step. They proposed that new SCGRA design and generation of corresponding bitstream is needed per application domain basis using vendor tools. After that, proposed tool can schedule DFGs on SCGRA and generate instructions, per application basis, which can be merged with SCGRA bitstream to generate final downloadable bitstream for the target FPGA. As the full FPGA

bitstream needs to be reconfigured for a compute kernel change, very fast context switching between applications, in the order of a few microseconds, is not possible.

2.5.2 Customized Topology Based

reMORPH: The reMORPH [145] overlay was proposed as a 2D array of tile-based compute units, where each tile uses the hard macros available in the FPGA fabric to achieve a lower footprint, consuming 1 DSP Block, 3 block RAMs, 196 LUTs and 41 registers. However, the reMORPH FU does not use decoders to reduce the overhead caused by routing and muxes, and thus requires a 72-bit instruction memory (Supporting a maximum of 512 instructions) resulting in the over utilization of BRAMs. Tiles are interconnected using an NN style of non-programmable interconnect, which is adapted using partial reconfiguration at run-time, and hence, suffers from the same slow hardware context switch problem as SCGRA.

TILT: The TILT overlay [144, 154], being a floating point overlay, unlike the other overlays discussed here, results in high resource consumption. TILT overlay was proposed as a highly configurable compute engine for FPGAs with multiple, varied and deeply pipelined floating point FUs which can be scaled by instantiating multiple copies of the TILT core. All cores, consisting of data memory, crossbar switches and FUs, share a single instance of the instruction memory and execute in parallel in single-instruction-multiple-data (SIMD) fashion. TILT consists of a scratchpad which is not double-buffered and the off-chip memory transfers are interleaved into the compute schedule using ports that are shared with the FUs. 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 close to that of the HLS implementations, with an area overhead of less than 2x for the same throughput.

TILT was presented as an overlay to complement Altera OpenCL HLS by sacrificing application throughput in scenarios when limited hardware resources are available. Since OpenCL HLS maximizes throughput at the cost of more resources by generating a heavily pipelined, spatial design, authors suggested to use TILT overlay as TILT enables smaller implementations than OpenCL HLS when a lower throughput is adequate, allowing a larger range of design space to be explored. TILT uses a weaker form of application customization by varying the mix of preconfigured standard FUs and optionally generating application-dependent custom units to handle predication, loops and indirect addressing. However, as each application requires that TILT be recompiled, a hardware context switch (referred to as a kernel update in the paper) takes on average 38 seconds.

TILT and OpenCL HLS designs were generated using Altera's Quartus 13.1 and OpenCL SDK targeting the Stratix V 5SGSMD5H2F35C2 FPGA with 2 banks of 4 GB DDR3 memory on the Nallatech 385 D5 board. An 8-core TILT system (with each core having one multiply FU and one add/sub FU) was specifically designed to implement a 64-tap FIR filter application, resulting in a throughput of 30 M inputs/sec and consuming 12K eALMs. For the same application, Altera OpenCL HLS was used to generate a fully parallel and pipelined implementation, resulting in a throughput of 240 M inputs/sec (8× higher throughput than 8-core TILT) and consuming 51K eALMs (4× higher area than 8-core TILT).

A number soft processor designs for FPGAs have also been proposed, which because they overlay the FPGA fabric, can be considered as overlay architectures. In [72], a two dimensional array of soft-core processors was proposed to perform text analytic queries. A number of soft vector processor designs, including VESPA [128], VEGAS [127], VIPERS [129], VENICE [130] and MXP [125], have been proposed. While these have exhibited impressive performance improvements compared to other scalar soft processors, when compared against more capable processors, such as the ARM processor found in the Xilinx Zynq platform, more modest improvements of approximately 4× against a single ARM core have been reported [155]. Most of the time multiplexed overlays described above not only results in performance overheads due to FU sharing and reduced throughput but

also result in large area overheads due to instruction storage requirements, which also results in a long kernel context switch time.

2.6 Spatially-configured Coarse-Grained Overlays

In spatially configured overlays, the compute and interconnect logic of the overlay are unchanged while a compute kernel is executing to support maximum throughput by dedicating individual FUs to kernel operations. This is different to time multiplexed overlays, where the compute and interconnect logic of the overlay change on a cycle by cycle basis while a compute kernel is executing to support time-multiplexing of overlay resources among kernel operations.

By far the largest number of coarse grained overlays in the research literature consist of spatially configured FUs and spatially configured interconnect networks [73, 74, 75, 78], which we shall refer to as SCFU-SCN or in short SC overlay. In an SC overlay, an FU executes a single arithmetic operation and data is transferred over a dedicated point-to-point link between the FUs. That is, both the FU and the interconnect are unchanged while a compute kernel is executing. This results in a fully pipelined, throughput oriented programmable datapath executing one kernel iteration per clock cycle, thus having an initiation interval (II) between kernel data packets of one. A number of different spatially configured interconnect strategies have been proposed, with the most common being: island style [73, 78], nearest neighbor (NN) [74] and to a lesser extent linear interconnect [156, 157]. However, the island style and nearest neighbor connected overlays suffer from a high area overhead due to the resources required for the interconnect network and are unsuitable for large compute kernels due to the limited size of the overlay that can be mapped to the FPGA fabric.

Spatially configured overlay fits well in a scenario where performance in terms of throughput is a primary objective given the rich logic resources. With the exponential increase of logic density on FPGA devices, it is possible to accommodate a massive number of FUs on an FPGA which allows to map all of the operations in

a compute kernel spatially on the array of FUs to exploit the parallelism available. The primary target in such a scenario would no longer be hardware sharing given the limited area constraint, but rather achieving the highest performance in terms of throughput under the rich logic resources. The key feature of such an array is the ability to exploit large amount of physical hardware resources to deliver scalable performance for data-parallel and throughput oriented applications.

Spatially configured overlays have a number of other advantages as well, such as the possibility to maintain extremely high throughput by employing a very heavy pipelining within the architecture as well as drastically reduced compilation times and configuration data sizes due to just one instruction per functional unit. But this comes at a cost of area and performance overheads. Hence, significant recent research effort has aimed to reduce area overheads and improving performance. The primary metrics considered include: the F_{max} and peak throughput of the overlay [74], programmability cost [74], the configuration data size and configuration time [73]. With these in mind, we now discuss the key features, performance metrics and overheads for a number of spatially configured overlay architectures proposed in the literature.

2.6.1 Nearest-neighbor Style Interconnect Based

QUKU: QUKU was proposed as a rapidly reconfigurable coarse grained overlay architecture [133] to bridge the gap between soft processors and customized circuit implementations. QUKU consists of a dynamically reconfigurable, coarse-grain FU array with an associated soft-core processor providing system support. Sobel and Laplace kernels from Edge detection application were used to evaluate the performance of the QUKU architecture [63]. As shown in Figure 2.5, authors used the concept of datapath merging to generate the QUKU overlay datapath by merging the datapaths of both kernels (Sobel and Laplace).

A 4×4 array of FUs was also designed to support the compute kernels which was compared to QUKU overlay and customized FPGA circuit implementations of the

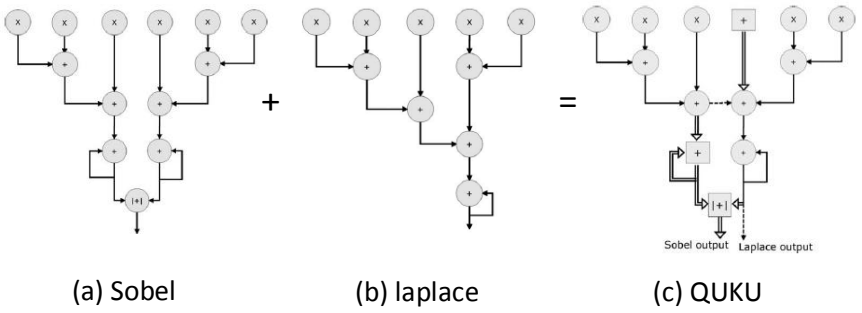


Figure 2.5: Datapath merging for QUKU overlay generation.

same compute kernels, designed using Xilinx System Generator for DSP. A point to point link was used to connect the FU with its four immediate neighbouring FUs. All of these designs were implemented on a Xilinx Virtex-4 LX25 device (ML401 development board). When compared to customized FPGA circuit implementations of the compute kernels, QUKU overlay and 4×4 FU array required approximately 1000× smaller configuration data sizes. It was shown that the customized FPGA circuit implementations of both kernels required 4% extra Slices than the device capacity, 4×4 FU array required 28% extra Slices than the device capacity while QUKU overlay (merged datapath) was able to fit on the device, consuming 89% of the Slices available on the device. Main point to note here is that both designs, QUKU overlay and 4×4 FU array, were evaluated for very small kernels. However, the concept of having an FU array or a merged datapath on top of FPGA fabric pave the way for fast context switching between kernels.

FPCA: FPCA [62] was proposed as an array of clusters, organized in a mesh with NN style interconnect, where each cluster consists of a set of PEs. PEs are connected by a permutation network with a high connectivity within a cluster, and then by a global NN style interconnect for more scalable connectivity. Authors used three image processing kernels (Gradient, Convolution and Sobel) as benchmarks to demonstrate the effectiveness of their overlay. Authors demonstrated the issue of mismatch between the processing throughput and the off-chip bandwidth. The *'gradient'* benchmark executing on a 32-bit overlay operating at 100 MHz has an off-chip bandwidth requirement of 2.4 GB/s (0.4 GB/s per I/O). Authors tried to replicate the kernel benchmark on the overlay to achieve higher throughput,

however due to the limited external DDR bandwidth, performance saturated at two copies of the kernel. On duplicating the kernel, the performance got doubled, however on further replication, performance improvement stops. Authors integrated the FPCA overlay within Xilinx Zynq device and used Linux OS for runtime management of the overlay. It was shown that the FPCA overlay results in 1.4 clock cycle per loop iteration which is close to the design target of 1 clock cycle per loop iteration. The gap is due to the overhead of page translation and bank switching in the main memory upon discontinuous data access. The approach used in the FPCA overlay project inspires research on high throughput communication interfaces and memory controllers for throughput oriented overlays.

Mesh-of-FU based Overlay: An overlay architecture optimized for high F_{max} and throughput was proposed in [74] for pipelined execution of data flow graphs. The prototyped 24×16 overlay is a nearest-neighbor-connected mesh of 214 routing cells and 170 heterogeneous functional units (FU) comprising 51 multipliers, 103 adders and 16 shift units. Authors used the concept of elastic pipelines to handle the latency imbalance at FU inputs. The overlay executes a given DFG by mapping the graph nodes to the FUs and configuring the routing logic to establish inter-FU connections that reflect the graph edges. A placer and router was developed by customizing VPlace [158] and PathFinder [159], respectively.

Authors mention that the direct synthesis of 24×16 overlay results in low F_{max} in the range of 100-150 MHz and a novel synthesis approach is necessary to achieve high F_{max}. Using the proposed approach of synthesis, an F_{max} of 355 MHz and a peak throughput of 60 GOPS was reported when implemented on an Altera Stratix IV FPGA (420K ALUTs), consuming 204 DSP blocks, 200K ALUTs and 250K FFs. 186K ALUTs and 230K FFs were required to implement the routing network for 170 FUs. Hence the proposed overlay results in a requirement of 3K ALUTs (routing network area) per GOPS (peak throughput). Thus the resource overhead associated with overlay architectures, particularly the routing network, is still a concern. For overlays to be compelling, they must have high performance, low area overhead and be capable of serving as general purpose compute-kernel accelerator.

2.6.2 Island Style Interconnect Based

Intermediate Fabrics: An overlay architecture, referred to as an intermediate fabric (IF) [136], was proposed to support near-instantaneous placement and routing (on average within a second). A 9×9 array of 16-bit FUs were interconnected using FPGA-like island-style coarse grained interconnect. The majority of FUs were mapped directly onto Xilinx DSP48 units, with some additional shift registers to handle realignment for pipelined routing. In the case of using fully flexible connection boxes, when implemented on to a Xilinx XC4VLX200 FPGA (178K LUTs), the array consumed 81 DSP Blocks and 32% of LUTs (57K LUTs) and 59% of the LUTs (105K LUTs) for a channel width of 2 and 4, respectively, resulting in an F_{max} of 195 MHz and LUT/FU count of 703 and 1296 for a channel width of 2 and 4, respectively. Authors evaluated customized IFs for twelve case studies, which they manually implemented as technology mapped IF-netlists. Instead of proposing a generic IF to support all of the netlists, authors focused on designing custom IFs for each netlist.

A generic IF (specialized for common image-processing kernels) in [73] was implemented on an Altera Stratix III FPGA (200K ALUTs) in order to evaluate area and performance overheads. It consists of 192 heterogeneous FUs comprising 64 multipliers, 64 subtractors, 63 adders, one square root unit, and five delay elements with a 16-bit datapath and supports fully parallel, pipelined implementation of compute kernels. The IF only achieves an F_{max} of approximately 125 MHz when implemented on a Altera Stratix III FPGA, resulting in a peak throughput of 24 GOPS, consuming 114K LUTs. When compared to a direct FPGA implementation of the biggest kernel that IF can support, IF consumes 4.5× LUTs, 4× FFs and 2× DSP blocks. It enabled a 700× improvement in compilation time compared to vendor tools at the cost of approximately 40% extra resources (approx 80K ALUTs) on the FPGA. Entire IF can be configured using 9234 bits of configuration data (stored in a 128-bit wide BRAM), enabling reconfiguration latency of 28 to 72 cycles. Actual speedup compared to software averaged 8.3× for the IF and 8.8× for the direct implementations. One major drawback of this IF was

the use of heterogeneous array of different types of FUs instead of homogeneous array of programmable FUs, resulting in increased requirement of programmable interconnect resources. For example, the kernels used in [73] required a minimum of 192 heterogeneous FUs. Using homogeneous array of programmable FUs, this requirement which could have reduced to 128 FUs and hence reduced interconnect. Further using DSP block like programmable FU where multiple operations can be merged onto single, FU requirement could have reduced to 64 FUs.

Another IF based on island-style interconnect architecture with a channel width (CW) of two, was mapped to a Xilinx XC5VLX330 FPGA (207K LUTs), along with a low overhead version of the interconnect [137]. To perform design space exploration of overlay interconnect architecture, authors used fixed-logic multipliers (mapped on DSP blocks by synthesis tool) as functional units (FUs) so that the device utilization represents the LUT and FF overhead of implementing the target application via an overlay rather than a direct HDL implementation. Both, the original and the optimized, overlay consumed 196 FUs (DSP blocks) for a size of 14×14. The original 14×14 overlay used 44% of LUTs (91K LUTs) with an F_{max} of 131 MHz while the optimized overlay used 24% of the LUTs (50K LUTs) with an F_{max} of 148 MHz. Authors were able to reduce LUT requirements by 48%-54% and flip-flop requirements by 46%-59%, while improving clock frequencies by an average of 24% at the cost of 16% routability overhead. Although the authors were able to reduce the LUTs/DSP count from 465 to 255, the overlay could not utilize DSP blocks to the full extent since each FU was mapped to a DSP block utilizing only multiplier.

Apparently, the IFs are beneficial in improving the design productivity and portability, though they do result in moderate area overhead and timing degradation. The area overhead is mainly due to the virtual routing resources consisting of multiplexer based coarse grained interconnect, implemented as switch boxes and connection boxes.

Reconfiguration Contexts: Different applications require different sized overlays, with an overlay large enough to satisfy the resource requirements of the

largest kernel being heavily underutilized when a small kernel is mapped to the overlay. Also according to [138], it is not possible to create an optimal fabric for all combination of compute kernels. To avoid above mentioned two issues, authors presented a design heuristic that analyzes kernel requirements from an application (or domain) and clusters them based on similarity into a set of fabrics referred to as reconfiguration contexts [138]. When a context does not support a kernel, authors proposed to reconfigure the FPGA fabric at runtime with the different overlays (reconfiguration contexts) [138]. The overall goal is to minimize individual context area by minimizing the number of resources required for the context to support all its assigned kernels. Although possibly more efficient than an IF, this overlay (designed for a specific set of kernels known at design time) provides no support for kernels not known at design time.

Authors designed 5 different overlays (2 floating point and 3 fixed point overlays), each specialized for a specific set of kernels, and implemented them on Xilinx Virtex-6 FPGA (XC6VCX130T) with an F_{max} ranging from 196 MHz to 256 MHz. The configuration data size for different overlays ranges from 429 Bytes to 1208 Bytes and the configuration time ranges from 13.4 us to 49.3 us.

DySER as a Dynamically Specialized Hardware: There are ever-increasing demands for performance without crossing power-demand thresholds in data centers and also without compromising battery life in mobile platforms. It will no longer always be possible to simply move to the next-generation general-purpose processor to meet tighter energy-performance constraints. An alternative that improves energy efficiency is the use of dynamically specialized hardware as an accelerator within heterogeneous computing platforms. The DySER architecture [131, 139] is one example of dynamically specialized hardware where the idea is to dynamically synthesize large compound FUs to match program regions, using a co-designed compiler and micro-architecture. The compiler enables automatic identification and specialization of compute intensive parts of an application. Intel's MicroOp-fusion is another example, where a specialized datapath is used for the execution of fused micro-ops by examining the instruction streams. Another

commercialized example is the SIMD accelerator, which extends a single instruction into multiple FUs and process parallel data streams at the same time.

DySER was originally designed as a heterogeneous array of 64 FUs (60% integer ALU, 10% integer multiply and 30% floating point units) interconnected with a circuit-switched mesh network [139]. Instead of using an array of homogeneous programmable FUs, with each FU capable of primitive operations like addition, multiplication, and logic operations, a heterogeneous array of FUs was used because of the area overheads associated with homogeneous programmable FUs. The DySER RTL was integrated with the OpenSPARC T1 RTL and synthesized as an ASIC, demonstrating up to 70% reduction in energy consumption and up to 10× speedup in application execution.

The DySER architecture was improved by using homogeneous programmable FUs and was then prototyped, along with the OpenSPARC T1 RTL, on a Xilinx Virtex-5 device (having 69K LUTs) [75]. The integrated system had a critical timing path of 12.7 ns, whereas OpenSPARC had a critical timing path of 10.1 ns. OpenSPARC consumed 31K LUTs while a 2×2 32-bit DySER consumed 27K LUTs. Due to excessive LUT consumption, it was only possible to fit a 2×2 32-bit DySER, a 4×4 8-bit DySER or an 8×8 2-bit DySER on the FPGA. The 2×2 32-bit DySER (supporting just 4 operations) is of limited value in performance evaluation, and instead a 4×4 DySER (supporting up-to 16 operations) or an 8×8 DySER (supporting up-to 64 operations) is required to provide meaningful performance comparisons. The possibility of building a 32-bit datapath out of a 2 bit datapath, but consuming 16 cycles across every link, was discussed as a FPGA specific design optimization, however this would degrade the performance significantly.

The DySER architecture, although relatively efficient from an application mapping perspective, suffered because it was implemented without much consideration for the underlying FPGA architecture. Considering the presence of hard macro blocks, and previous work that has demonstrated how these can be used for general processing at near to their theoretical limits [160], we propose enhancing DySER by

using the DSP48E1 found in all modern Xilinx FPGAs to take on most functions of the FU. The motivation is to use low level features of the FPGA architecture to develop architecture centric high performance computing blocks. In the next section, the Xilinx DSP48E1 hard macro is discussed, along with the dynamic mode control feature of the DSP blocks which allow cycle by cycle adaption of the DSP block functionality.

Xilinx DSP48E1 Block: As technology started growing rapidly, FPGAs started to get much more high performance with coarse grained processing elements which enabled them to address signal processing and high performance computing problems. Following the addition of memory blocks, hard multipliers were added to speed up common signal processing tasks. These later evolved into multiply-accumulate blocks, often used in filters. Modern Xilinx FPGAs provide DSP48E1 blocks for computations requiring 25×18 multipliers, 25-bit Add/sub operators, 48-bit ALU or a combination of these. They offer significantly reduced power consumption, superior logic density and speed, than equivalent LUT implementations of the same computations.

DSP48E1 blocks can be divided into three stages: pre-adder, multiplier, and ALU as shown in Figure 2.6. The pre-adder is a 25-bit two-input adder/subtractor which takes data from the 30-bit A port and the 25-bit D port and produces a . Its output is fed as one of the inputs of the multiplier which has asymmetric 18-bit and 25-bit inputs. The ALU is 48 bits wide and operates on the output of the multiplier and another input. Logic functions supported by the ALU are AND, OR, NOT, NAND, NOR, XOR, and XNOR. Three multiplexers X, Y, and Z, are used to select appropriate inputs for the ALU block.

DSP48E1 blocks can be configured in many different ways to perform different arithmetic operations and support internal pipelining. As we can see in Figure 2.6, the DSP48E1 block can be divided into a maximum of four pipeline stages. Two of the pipeline stages are available at inputs A and B, one at the output of the multiplier, and one at the output of the ALU. These pipeline stages are parameterised which means that the number of pipeline stages can be configured while

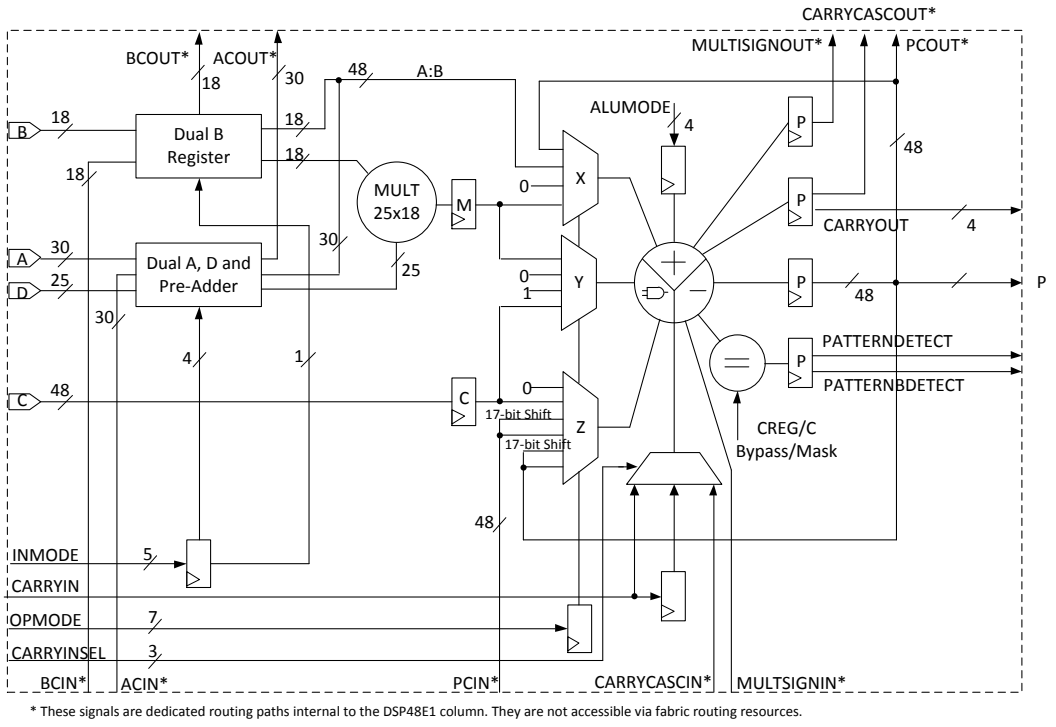


Figure 2.6: Internal architecture of the DSP block.

instantiating the primitive. This cannot be changed dynamically, unlike other functions of the DSP48E1. To achieve a high frequency, thus maximizing application throughput, all of the pipeline stages of the DSP48E1 primitive can be enabled.

The key feature of DSP blocks for the Xilinx family of FPGAs is the ability to dynamically change the functionality at runtime using the dynamic mode control feature of the DSP blocks. The functionality of these DSP blocks can be modified in every clock cycle, greatly enhancing the flexibility and usability of these blocks to implement programmable ALU within the FU of coarse-grained architectures. Four control inputs are used to control and configure the operations of DSP48E1. These are *INMODE*, *OPMODE*, *ALUMODE*, and *CARRYINSEL*.

INMODE is a 5-bit control input out of which least significant 4 bits selects the functionality of the pre-adder block, which serves as a 25-bit input to the multiplier block and input registers of A and D. The most significant bit of *INMODE* selects the input register of the multiplier B port. *OPMODE* is a 7-bit control

input, that controls the outputs of the X, Y, and Z multiplexers. *OPMODE*[1:0] selects the X multiplexer input, *OPMODE*[3:2] selects the Y multiplexer input, and *OPMODE*[6:4] selects the Z multiplexer input. *ALUMODE* is a 4-bit control input, that controls the behavior of the ALU block. *CARRYINSEL* is a 3-bit control input, which selects the appropriate source for CIN. In literature, the dynamic mode control feature has been used for:

- The iDEA Processor [160]. The iDEA processor shares the DSP block across multiple instructions, and achieves an F_{max} of 405 MHz using a 9-stage pipeline when implemented on a Virtex-6 (XC6VLX240T) device. iDEA consumes 1 DSP Block, 321 LUTs and 413 FFs. An RTL implementation of the processor, without instantiating the DSP primitive as an execution unit, occupies 38% more registers and 169% more LUTs compared to iDEA. The tool still synthesized a DSP block for the 16×16 multiplication, but only achieved a clock frequency of 173 MHz, just 42% of iDEAs frequency.

- The Hoplite-DSP NoC [161]. Hoplite-DSP embeds the functionality of the Hoplite router into DSP blocks. A 32×16 Hoplite-DSP NoC mapped onto a VC707 board with a Xilinx XC7VX485T FPGA, consumes 1.5K DSP48E blocks, but reduces the LUT costs from 35K to 7K LUTs, representing a 5× saving, and reduces FF costs from 62K to 9K, representing a 6× saving.

- Application-specific hardware synthesis for DSP block architecture-aware high level synthesis [162]. Careful use of the internal architecture of the DSP block while synthesizing from HLL to RTL allows 1.2× throughput improvement over Vivado HLS generated RTL implementations, at the cost of up to 23% extra LUT resources.

2.7 Summary

In reviewing the literature, we find that many overlays are developed with little consideration for the underlying FPGA architecture. Previous work has demonstrated that the DSP-rich FPGA fabrics in modern devices can support general purpose processing at near to DSP block theoretical limits [124, 76]. In this thesis, we aim to explore coarse grained overlays designed using the flexible DSP48E1 primitive on Xilinx FPGAs, which can allow pipelined execution of compute kernels without adding significant area and performance overheads.

3

Adapting the DySER Architecture as an FPGA Overlay

3.1 Introduction

Recently the concept of using dynamically specialized hardware for accelerated computing is gaining traction for improving energy efficiency of computing platforms [163]. The DySER architecture [131, 139] is one example of dynamically specialized hardware which was proposed to improve the performance of general purpose processors by integrating a programmable array of FUs, referred to as dynamically specialized execution resources, into the processor pipeline. The DySER architecture exhibit similarities not only with other spatially configured coarse-grained architectures [63, 73, 74] but also with conventional tiled architectures

such as RAW [164], Wavescalar [165] and TRIPS [166]. However, integration of DySER within the pipeline of a processor would require a complete redesign of the processor micro-architecture. Another approach is to overlay the DySER architecture on top of a commercial FPGA fabric and use it as a bus-attached coprocessor within a heterogeneous computing platform.

This chapter examines the DySER overlay architecture, mapped on the Xilinx Zynq SoPC, to show that DySER suffers from a significant area and performance overhead due to limited consideration for the underlying FPGA architecture. The DySER architecture was chosen because it is the only spatially configured architecture available as open source. We then propose an improved FU architecture using the flexibility of the DSP48E1 primitive which results in a 2.5 times frequency improvement and 25% area reduction compared to the original FU architecture. The motivation is to use low level features of the FPGA architecture, specifically the dynamic mode control feature of DSP blocks, to develop architecture centric high performance computing blocks. We demonstrate that this improvement results in the routing architecture becoming the bottleneck in performance.

The main contribution of this chapter is to show how to configure and use the DSP block for an efficient and practical implementation of the DySER FU. The modified DySER architecture (referred to as DSP-DySER) is then targeted to the Xilinx Zynq. This DSP-DySER overlay can then be used to host accelerators to offload data-parallel compute kernels from compute-intensive applications running on the ARM processor. We demonstrate how adopting the Xilinx DSP48E1 primitive in the FU of the DySER architecture improves the performance and reduces the area overheads. The main contributions can be summarized as follows:

- An RTL implementation of an FU (compatible with the DySER architecture) using the DSP48E1 primitive, which can operate at near theoretical maximum frequency.
- A scalability analysis of DSP-DySER and an analysis of performance improvement on the Xilinx Zynq device, including the evaluation of peak

throughput (in terms of GOPS) and interconnect area overhead (in terms of LUTs/GOPS) of DSP-DySER.

- A quantitative analysis of the area overheads of the DSP-DySER architecture by mapping a set of benchmarks to DSP-DySER and to the FPGA fabric using Vivado HLS.
- A quantitative evaluation of the hardware performance penalty of DSP-DySER compared to HLS generated hardware implementations.

The work presented in this chapter is also discussed in

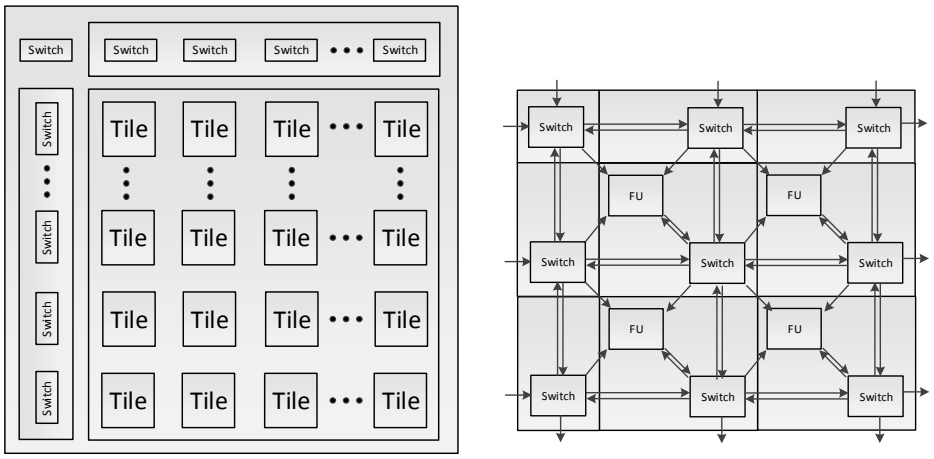
- A. K. Jain, X.Li, S. A. Fahmy, and D. L. Maskell. *Adapting the DySER Architecture with DSP Blocks as an Overlay for the Xilinx Zynq*, in ACM SIGARCH Computer Architecture News (CAN), vol. 43, no. 4, pp. 28-33, September 2015.

3.2 The DySER Architecture

As shown in Figure 3.1(a), The DySER architecture consists of two blocks, the tile fabric and the edge fabric, where each tile in the tile fabric instantiates a switch and an FU, while the edge fabric only instantiates a switch, forming the boundary at the top and left of the tile fabric. The resulting architecture contains I/O ports around the periphery of the fabric, which are connected to FIFOs. A simple 2×2 DySER overlay, consists of four tile instances and five switch instances along the North and West boundaries, resulting in 4 FUs and 9 switches, as shown in Figure 3.1(b). Extrapolating this to an $N \times N$ DySER architecture results in N^2 FUs and $(N+1)^2$ switches.

3.2.1 DySER Switch

The switches allow datapaths to be dynamically specialized. These switches form a network that creates paths from inputs to the FUs, between FUs, and from FUs



(a) DySER block diagram. (b) 2×2 DySER as an example.

Figure 3.1: DySER architecture.

to outputs. Switches in DySER have 5 inputs (4 from neighbour switches and 1 from the FU at the North-West direction) and 8 outputs (to all 8 directions). Hence, switches require a 5:1 multiplexer and a state machine for synchronization at each output.

3.2.2 DySER Functional Unit

The FU provides resources for the mathematical and logical operations, and synchronization logic. It receives its input values from the four neighbouring switches and outputs its result to the switch in the south-east direction. The FU consists of programmable computation logic and a state machine as synchronization logic at each input and output of the computation logic. The state machine implements a credit-based flow-control protocol to enable receiving of inputs asynchronously at arbitrary times from the FIFO interfaces.

The operators in the FU can be selected according to application requirements. We choose four operators: Add, Sub, Mul and OR in the FU, as shown in Figure 3.2, to map the benchmarks from [167]. The benchmark characteristics are given in Table 3.1. Figure 3.3 shows the mapping of these benchmarks to the DySER architecture. A 5×5 DySER can be used to implement all of the benchmarks considered in Table 3.1.

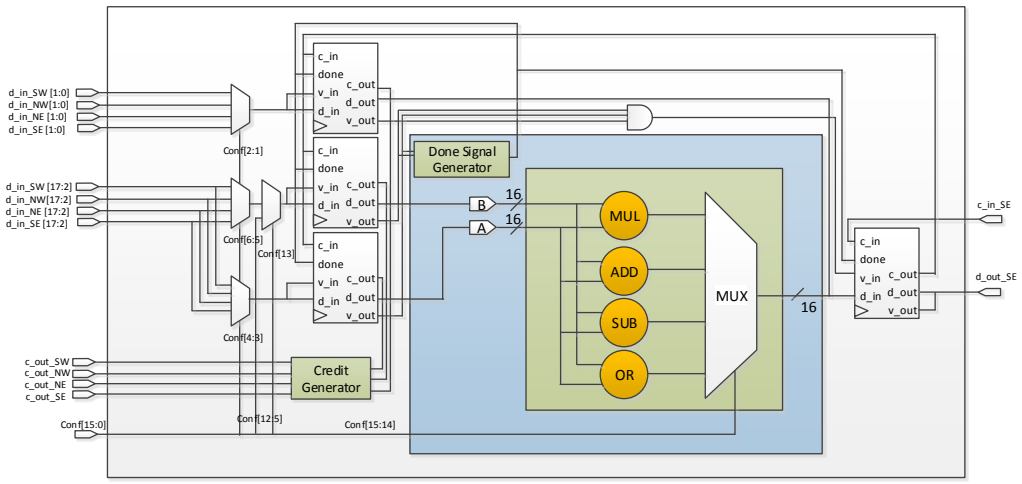


Figure 3.2: Functional unit architecture.

No.	Benchmark	Add	Sub	Mul	OR	Total
1.	fft	3	3	4		10
2.	kmeans	7	8	8		23
3.	mm	7		8		15
4.	mri-q	3		6	1	10
5.	spmv	6		8		14
6.	stencil	10	2	2		14
7.	conv	8		8		16
8.	radar	6		2		8

Table 3.1: Benchmark characteristics

Handling Latency Imbalance at FU inputs: A credit-based flow control (simplified for the statically-switched network) was used to handle latency imbalance at the FU inputs. Any stage (FU or switch) needs one credit to send data, and after sending the data it sends a credit signal to its predecessor. If a stage is processing, or delayed waiting for data, the valid bit is cleared and credit is not passed to the previous stage. This allows the handling of latency imbalance and prevents values from a new invocation overwriting values from a previous invocation, since the array receives inputs asynchronously at arbitrary times from the FIFO interfaces.

The original DySER FU, supporting a 16-bit datapath, was implemented using

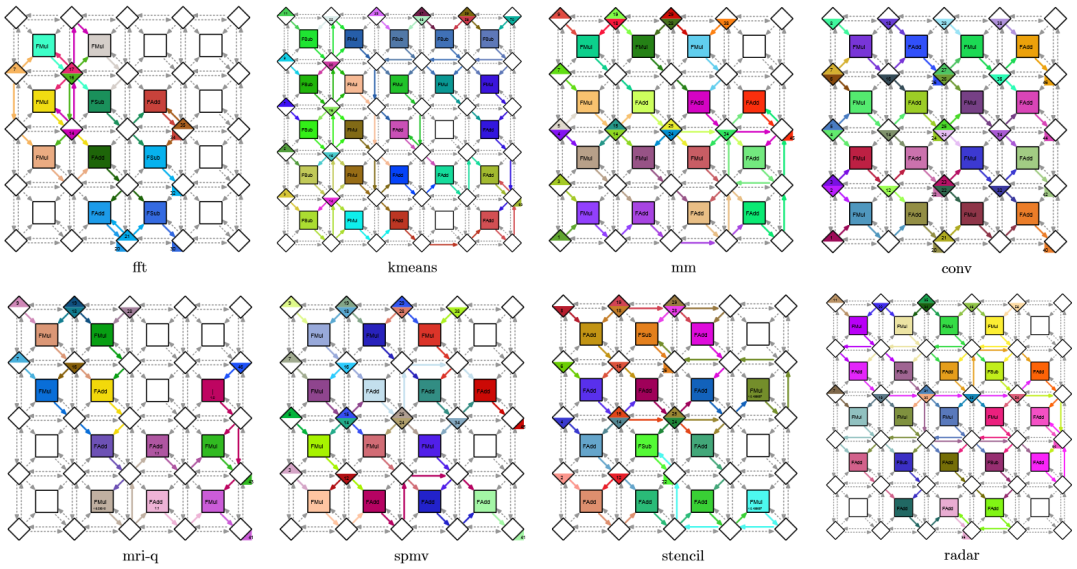


Figure 3.3: Mapping of kernels on DySER architecture.

Xilinx ISE 14.6 targeting a Xilinx Zynq XC7Z020. The FU consumes 49 Slices (148 LUTs, 66 FFs) and 1 DSP48E1 block, with a critical path of 6.7 ns. Hence the maximum operating frequency of the FU is 150 MHz. Figure 3.4 shows the physical mapping of the FU to the FPGA fabric [168]. While synthesizing, the tool infers a DSP block for multiplication. The remainder of the operations and the multiplexer in the compute logic are mapped to 17 Slices (57 LUTs). State machines and input selection multiplexers are mapped to 32 Slices (91 LUTs and 66 FFs). After integrating the FU into the DySER tile and implementing it on the FPGA fabric, we found that the critical path in the DySER Tile is the same as the critical path of the FU (6.7 ns), and hence the FU limits the performance of the DySER tile.

3.3 DSP Block Based DySER (DSP-DySER)

To reduce the DySER critical path, while building on the advantages of hard DSP macros for implementing high speed PEs, we examine the use of the Xilinx DSP48E1 primitive as a programmable FU in DySER. Despite the fact that the original FU uses a DSP block for multiplication, it does not fully exploit the

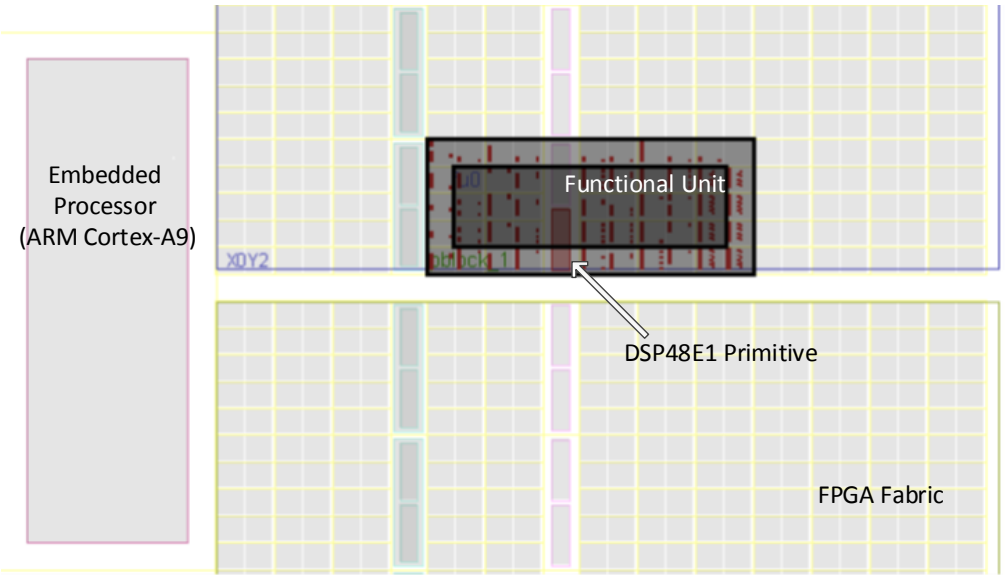


Figure 3.4: Physical mapping of the functional unit on FPGA.

performance advantage of the DSP block. Since the DSP48E1 can be dynamically configured and used for operations required by the FU, we show that an area and performance efficient FU can be built by making use of DSP block as an ALU, instead of just as a multiplier, and by enabling the internal pipeline registers of the DSP block.

3.3.1 DSP48E1 Based Functional Unit

We use the DSP48E1 primitive, as shown in Figure 2.6, to implement the computation logic in the modified FU, as shown in Figure 3.5. As mentioned earlier, The DSP48E1 primitive has a pre-adder, a multiplier, an ALU, four input ports for data, and one output port P, 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 configuration registers. The DSP48E1 primitive is directly instantiated providing total control of the configuration of the primitive. This allows us to maximize the compute kernel throughput

and achieve a high FU frequency by operating the DSP48E1 at its maximum frequency.

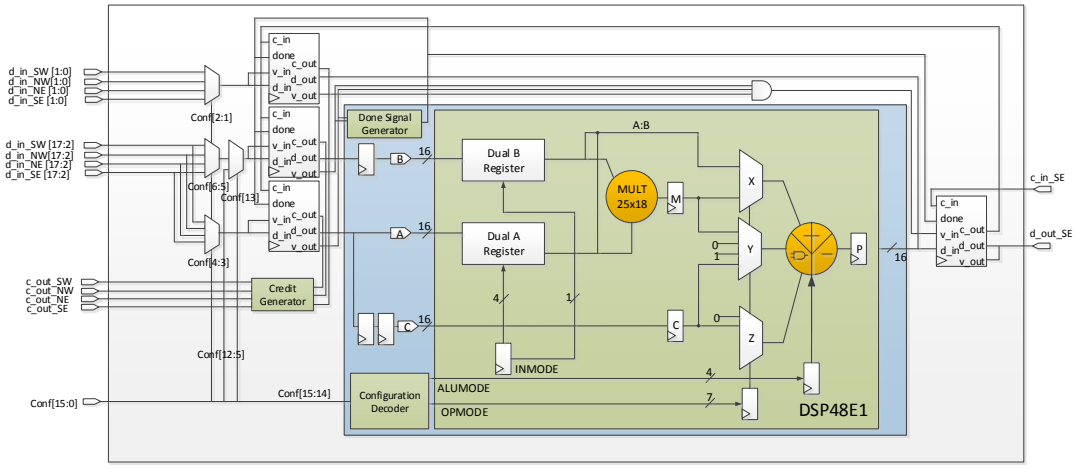


Figure 3.5: DSP48E1 based functional unit architecture.

We enable all of the pipeline stages of the DSP48E1 primitive. The redesign of the DySER FU replaces the original compute unit (CU), shown in Figure 3.2, with the fully pipelined DSP48E1 primitive, along with modifications to the done signal generation logic and configuration decoding logic, as shown in Figure 3.5. The two inputs from the FU (to the CU) are connected to the three ports of the DSP48E1 primitive, as shown in Figure 3.5. The FU configuration register includes 2 bits for operation selection with the other 14 bits for constant and input multiplexers. Additionally, we require three 16-bit registers at the DSP input ports (as shown in Figure 3.5), consuming 48 FFs to balance the internal pipeline stages of the DSP block. Table 3.2 shows the DSP48E1 configuration settings required for each operation. Inmode remains the same for all of the operations and hence we hard-code it to 00000.

3.3.2 Analysis of Performance Improvement

We analyze the performance improvement of the FU in terms of frequency and resource usage. The DSP48E1 based FU consumes 37 Slices (116 LUTs, 117 FFs) (25% less than the original FU) and 1 DSP block. Apart from obvious area savings,

Operation	ALUMODE	OPMODE	INMODE
ADD	0000	011 0011	00000
SUB	0011	011 0011	00000
MUL	0000	000 0101	00000
OR	1100	011 1011	00000

Table 3.2: DSP48E1 configuration for each operation

the strategy of using a fully pipelined DSP block as the computational part of the FU also improves overall timing performance. The FU has a critical path of just 2.7 ns, resulting in a maximum frequency of 370 MHz, which is 2.5× that of the original FU. Figure 3.6 shows the physical mapping of the FU onto the FPGA fabric.

Since a hard primitive is used for the implementation of CU operations, only minimal additional circuitry is implemented in the logic fabric which consists of configuration decoding logic, three 16-bit balancing registers and done signal generation logic. All of this additional circuitry is mapped to 10 Slices (25 LUTs and

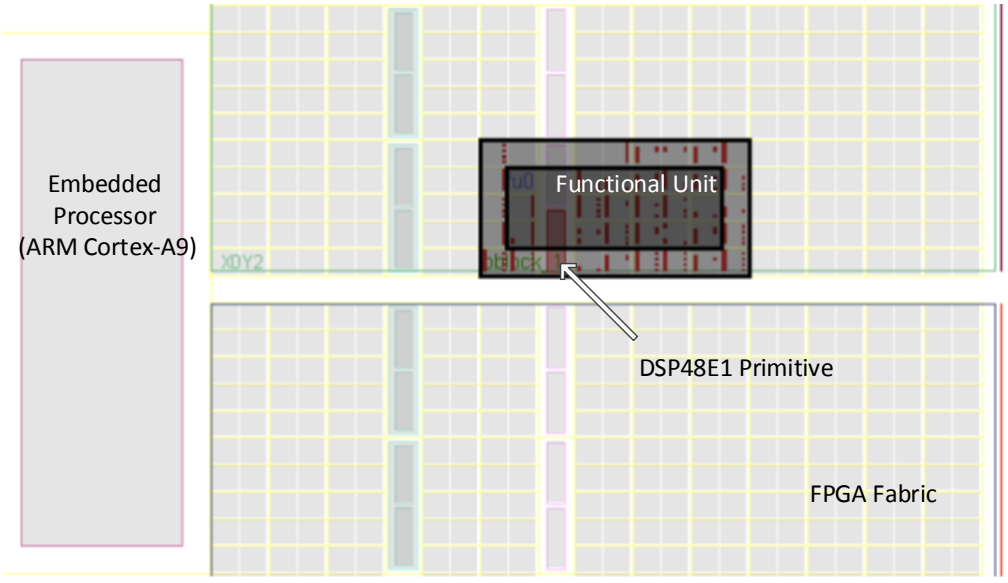


Figure 3.6: Physical mapping of the enhanced functional unit on FPGA.

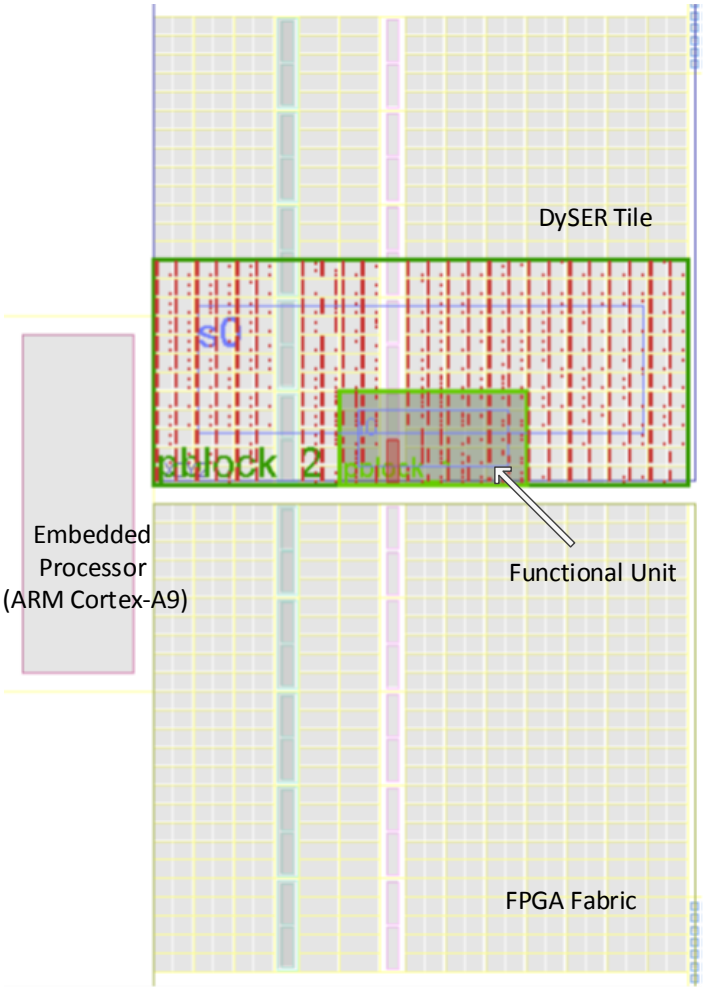


Figure 3.7: Physical mapping of the DSP-DySER tile on FPGA.

51 FFs). State machines and input selection multiplexers are mapped to 27 Slices (91 LUTs and 66 FFs).

By integrating the enhanced FU into the DSP-DySER tile and implementing it on the FPGA fabric, we found that the critical path of the switch, which is 5.3 ns, now limits the performance of the DSP-DySER tile. Figure 3.7 shows the physical mapping of the DSP-DySER tile to the FPGA fabric. It is clear that the major area overhead in DSP-DySER is due to significant resources consumed in the switch implementation. The switch consumes 251 Slices (995 LUTs and 325 FFs) and hence the whole tile consumes 288 Slices (1118 LUTs and 447 FFs). The largest source of area overhead comes from the multiplexing logic in the switch which could be minimized by using techniques mentioned in [137, 169, 170].

We have shown that a more architecture-oriented approach to designing the FU enables it to be small and fast. As a result the routing for the coarse grained array becomes the limiting factor which should be addressed.

3.4 Scalability Analysis

The Xilinx Zynq-7020 fabric consists of 220 DSP blocks, with a theoretical maximum frequency of 400 MHz, each of which can support up to 3 arithmetic operations, resulting in a peak throughput of 264 GOPS. We map the largest DSP-DySER array possible to the Zynq device and determine the resource utilization and peak throughput of the DSP-DySER overlay in GOPS to examine whether the DSP-DySER overlay can be used to exploit the raw performance of the available DSP blocks to the full extent.

The DSP-DySER overlay is implemented by replicating tiles and switches on the FPGA fabric. One tile consumes 2.16% of Slices and one switch consumes 1.88% of the Slices present in the fabric. As discussed previously, an $N \times N$ DSP-DySER overlay incorporates N^2 Tiles in the tile fabric and $2N + 1$ switches in the edge fabric. Hence, a 6×6 DSP-DySER overlay is the largest that can fit on the Zynq-7020. Table 3.3 shows the resource usage for different overlay sizes while Figure 3.8 shows the FPGA resource utilization.

Resource type	2x2	3x3	4x4	5x5	6x6
LUTs	5330	12785	22306	33875	48171
FFs	2781	5493	8950	13390	18728
Slices	2458	6538	9700	12284	13244
DSPs	4	9	16	25	36

Table 3.3: Resource usage for 16-bit DSP-DySER on Zynq-7020

It is clear from Table 3.3 and Figure 3.8 that excessive LUT usage ($\approx$ 1360 LUTs/DSP) becomes a limiting factor in exploiting the raw performance of all DSP blocks available on the Zynq device. It would only be possible to exploit the full

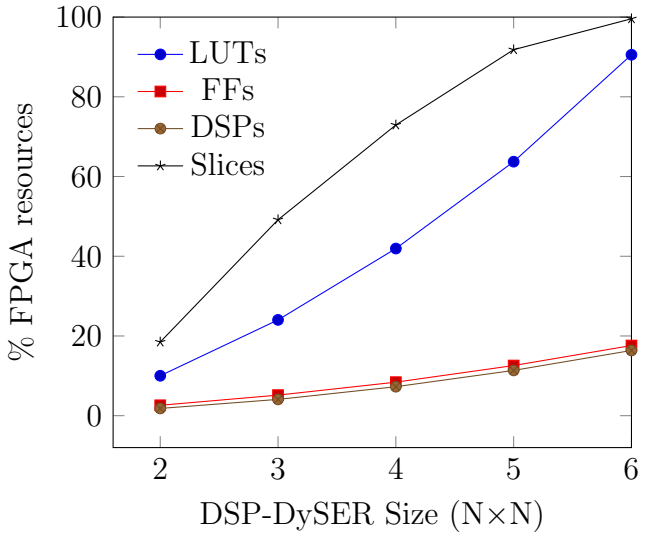


Figure 3.8: % Resource usage of Zynq-7020 for 16-bit DySER.

raw performance if the LUTs/DSP factor of the overlay implementation matches (or is less than) the LUTs/DSP factor of the Zynq device. The Xilinx Zynq-7020 fabric consists 53200 LUTs and 220 DSP blocks which corresponds to a LUTs/DSP factor of 240. This significant gap in the LUTs/DSP factor of the DSP-DySER overlay implementation and that of the Zynq device is mainly due to the significant resource overheads associated with the switch network, which needs to be optimized to exploit the raw performance of all DSP blocks to the full extent.

Since a DSP block in DSP-DySER is used to support one operation, an overlay of size $N \times N$ can support up to N^2 operations. Hence the peak throughput of an overlay of size $N \times N$ is equal to $N^2 * F_{max}$ GOPS. A 6×6 DSP-DySER achieves an F_{max} of 175 MHz on Zynq, hence providing a peak performance of 6.3 GOPS which is only 2% of the maximum achievable peak performance on Zynq.

A quantitative comparison of the DSP-DySER with the Intermediate Fabrics (IFs) overlay [137] is given in Table 3.4. The first and second rows of the table shows the device used to implement the overlay and resources available, respectively. We compare the overlay size, LUTs used, F_{max} in MHz, maximum number of operations supported in the overlay, and the peak throughput (in GOPS) for the three different overlays; IF, IF with low overhead version of the interconnect architecture,

referred to as IF(opt), and DSP-DySER. IF and IF(opt) use an FPGA-like island-style coarse-grained interconnect architecture. Fixed-logic multipliers (mapped to DSP blocks by the synthesis tool) were used as the FUs for IF and IF(opt), so that the device utilization just represents the interconnect area overhead.

Resource	IF [137]	IF (opt) [137]	DSP-DySER
Device	XC5VLX330	XC5VLX330	XC7Z020
Slices|LUTs	51.8K|207K	51.8K|207K	13.3K|53K
Overlay	14×14	14×14	6×6
LUTs used	91K	50K	48K
Fmax (MHz)	131	148	175
Max OPs	196	196	36
Peak GOPS	25.6	29	6.3
LUTs/GOPS	3550	1725	7620

Table 3.4: Quantitative comparison of overlays

However, because of the different FPGA fabrics and the different overlay architectures, it is difficult to make meaningful comparisons between the different overlays. Hence, we introduce a new comparison metric: the interconnect resource used per unit peak throughput (LUTs/GOPS), which allows us to quantify the area overhead of the overlay interconnect architectures irrespective of the FU implementation.

Although the efficient use of DSP block in DSP-DySER provides an improvement of 2.5× in frequency and a reduction of 25% in area compared to the original FU design, the significant LUT resource requirement for the switch becomes the bottleneck in the performance and scalability of DSP-DySER. It is clear from Table 3.4 that almost the same number of LUT resources are required to implement a 6×6 DSP-DySER and a 14×14 IF(opt). Although DSP-DySER uses a DSP block based programmable ALU as the FU, instead of a fixed-logic multiplier, the interconnect area overhead of DSP-DySER is 4.4× higher than the interconnected area overhead of IF(opt). This high interconnect area overhead (7.6K LUTs/GOPS) and the low peak performance (6.3 GOPS) prevents the realistic and practical use of DSP-DySER overlay as a programmable accelerator. For FPGA overlays to

become a possibility for general purpose on-demand application acceleration this interconnect area overhead needs to be significantly reduced.

3.5 Area Overhead Quantification

As a comparison, albeit an unfair one as we are comparing static implementations requiring a relatively long compile time with rapidly compiled dynamic implementations, we generate RTL of the compute kernels using Vivado HLS 2013.2 in order to perform a quantitative analysis of area overheads. We use the pipeline *pragma* with an II of one to generate fully parallel and pipelined RTL implementation of the compute kernels. Table 3.5 shows the results for the Vivado HLS implementations of the benchmarks from Table 3.1.

Benchmark	LUTs	FFs	Slices	DSPs	Freq. (MHz)
fft	218 (0.4%)	485 (0.4%)	117 (0.9%)	4 (1.8%)	324
kmeans	613 (1.1%)	1252(1.2%)	215 (1.6%)	8 (3.6%)	249
mm	315 (0.6%)	920 (0.8%)	205 (1.5%)	8 (3.6%)	295
mri-q	243 (0.4%)	588 (0.5%)	147 (1.1%)	6 (2.7%)	268
spmv	292 (0.5%)	842 (0.8%)	180 (1.3%)	8 (3.6%)	297
stencil	460 (0.8%)	870 (0.8%)	200 (1.5%)	2 (0.9%)	303
conv	353 (0.6%)	918 (0.8%)	222 (1.6%)	8 (3.6%)	272
radar	163 (0.3%)	457 (0.4%)	92 (0.7%)	6 (2.7%)	304
5×5 FU array	2900 (5.5%)	2925 (2.7%)	925 (6.9%)	25 (11.4%)	370
5×5 DSP-DySER	33875 (63.7%)	13390 (12.6%)	12284 (92.4%)	25 (11.4%)	175

Table 3.5: Experimental results for the Vivado-HLS implementations of the benchmark set

The compute kernels ranged from using 0.3-1.1% (on average 0.6%) of the total LUTs in the FPGA, 0.4-1.2% (on average 0.7%) of the total FFs in the FPGA, 0.7-1.6% (on average 1.3%) of the total Slices in the FPGA and 0.9-3.6% (on average 2.8%) of the total DSP blocks in the FPGA. Table 3.5 also shows the result of mapping a 5×5 DSP-DySER onto the Xilinx Zynq 7020 device. A 5×5 DSP-DySER achieves an F_{max} of 175 MHz, consuming 63.7% of the total LUTs, 12.6% of the total FFs, 92.4% of the total Slices and 11.4% of the total DSP blocks in the FPGA fabric. It clearly shows that DSP-DySER require almost all of the fine-grained resources of the FPGA fabric to support the benchmark set while the

largest HLS mapped benchmark requires only 1.6% of the fine-grained resources of the FPGA fabric. However, in an accelerator context, the concept of using DSP-DySER by swapping in and out different benchmark kernels and reusing the FPGA resources for multiple benchmark kernels is an advantage, but at the cost of huge area overheads.

The area overheads associated with the more flexible DSP-DySER architecture need to be reduced by exploring alternative FU and interconnect implementations for overlays. For example, the area overhead can be lowered by sacrificing some of the flexibility in the interconnect architecture. However, for overlays to be compelling, they must not just have high performance and low area overhead, but must also be capable of serving as a general purpose compute-kernel accelerator. A significant challenge in using FPGA overlays is identifying interconnect architectures that provide appropriate overhead vs. flexibility trade-offs for a particular application domain or a set of benchmark kernels.

As a further comparison, We assess the overhead of the flexible routing network in a similar way to [74] by designing a hardwired version of the 5×5 DSP-DySER, referred to as 5×5 FU array, in which the switches are replaced by direct wires. The routing network overhead is the ratio of the 5×5 DSP-DySER overlay resources to those of the 5×5 FU array. 5×5 FU array consumes 5.5% LUTs, 2.7% FFs, 6.9% Slices and 11.4% DSP blocks, while a fully functional 5×5 DSP-DySER overlay consumes 63.7% LUTs, 12.6% FFs, 92.4% Slices and 11.4% DSP blocks. Hence, a 5×5 DSP-DySER overlay can be used to implement all of the compute kernels with a routing network overhead of 11× more LUTs, 5× more FFs, and 13× more Slices.

It is difficult to get a good feel for the relative performance of each implementation from the results of Table 3.5, because they all have different resource utilisations and operating frequencies. This is also the case when comparing implementations targeted to different vendors FPGAs. To overcome this problem, we attempt to normalize the hardware resource utilization using a single equivalent slices (e-Slices) metric, where we assume that 1 DSP block is equivalent to 60 slices based

Benchmark	OP	Freq. (MHz)	Slices	DSPs	eSlices	MOPS	MOPS/eSlice
fft	10	324	117	4	357	3240	9.0
kmeans	23	249	215	8	695	5727	8.3
mm	15	295	205	8	685	4425	6.5
mri-q	10	268	147	6	507	2680	5.3
spmv	14	297	180	8	660	4158	6.3
stencil	14	303	200	2	320	4242	13.2
conv	16	272	222	8	702	4352	6.2
radar	8	304	92	6	452	2432	5.4

Table 3.6: Determining MOPS/eSlice for the Vivado-HLS implementations of the benchmark set

on the ratio of slices/DSP on the Zynq XC7Z02-1CLG484C (which is approximate 60). We then introduce a new performance metric, the throughput per unit area (in MOPS/eSlice), which is able, to some extent, to normalise performance between different implementations and architectures. We then compare the performance, in terms of throughput per unit area, of the Vivado-HLS generated RTL implementation of kernels with that of the DSP-DySER overlay. For each of the benchmarks in Table 3.1, we obtain the area in e-Slices, the throughput in MOPS and throughput per unit area in MOPS/eSlice, as shown in Table 3.6.

We observe that the average throughput per unit area for the HLS implementation of the benchmark set is ≈ 7.5 MOPS/eSlice. In comparison, a 5×5 DSP-DySER overlay achieves 0.3 MOPS/eSlice, which is around 4% of the HLS implementations. This 25× hardware performance penalty needs to be considered in context with the ability of the overlay to support runtime compilation and runtime configuration of the compute kernels. However, this penalty still needs to be reduced for there to be any hope of mainstream adoption of coarse-grained overlays as high performance programmable accelerators.

3.6 Summary

We have presented an enhancement to the DySER coarse-grained overlay that uses the Xilinx DSP48E1 primitive to implement most of the FU, improving the area and performance. We show an improvement of 2.5× in frequency and a reduction of 25% in area compared to the original FU design. We quantify the area overheads by mapping a set of benchmarks to the adapted version of a 6×6 16-bit DySER overlay, referred to as DSP-DySER, and directly to the FPGA fabric using Vivado HLS. DSP-DySER, when implemented on a Xilinx Zynq, provides a peak performance of 6.3 GOPS with an interconnect area overhead of 7.6K LUTs/GOPS, but with a 25× hardware performance penalty compared to the HLS generated hardware implementations.

We have demonstrated that an architecture-focused FU design exposes the significant overhead of the flexible routing. Hence optimizing the switch network or developing a light weight interconnect architecture to reduce this overhead is necessary. In the next chapter, we present a more FPGA targeted overlay architecture that maximizes the peak performance and reduces the interconnect area overhead through the use of an array of DSP block based fully pipelined FUs and an island-style coarse-grained routing network.

4

Throughput Oriented FPGA Overlays Using DSP Blocks

4.1 Introduction

In this chapter, we design and implement a more FPGA targeted overlay architecture that maximizes the peak performance and reduces the interconnect area overhead through the use of an array of DSP block based fully pipelined FUs and an island-style coarse-grained routing network (referred to as, DSP block based Island-Style Overlay (DISO)), achieving a peak performance of 65 GOPS (10x better than DSP-DySER) with an interconnect area overhead of 430 LUTs/GOPS (18x better than DSP-DySER). The overlay uses the dynamic programmability of the DSP block and maps up to three operations to each node (1 add/sub, 1 mul,

1 ALU op), resulting in a significant reduction in the number of processing nodes required. The motivation is to combine the benefits of an FPGA friendly interconnect architecture (having enough flexibility but with a low overhead compared to DySER) with the dynamic mode control feature of DSP blocks to develop architecture centric throughput oriented FPGA overlays. We demonstrate that this improvement results in better exploitation of the performance provided by the DSP blocks available on the FPGA fabric.

Next, we extend this work to improve the compute-to-interconnect resource usage ratio by including more compute nodes inside an FU, that is to use multiple DSP blocks. However, efficiently utilizing the DSP resources, both in terms of the optimum number of DSP blocks that can be used in an FU and being able to map applications to the new architecture, needs to be investigated. As such, we firstly analyse the characteristics of a number of compute kernels from the literature to ascertain the suitability of mapping multiple instances of kernels to the overlay.

We then prototype an enhanced version of DISO (referred to as Dual-DISO) which uses two DSP blocks within each FU and shows a significant improvement in performance and scalability, with a reduction of almost 70% in the overlay tile requirement compared to existing overlay architectures, an operating frequency in excess of 300 MHz and a peak performance of 115 GOPS (18x better than DSP-DySER) with an interconnect area overhead of 320 LUTs/GOPS (24x better than DSP-DySER).

We then map several benchmarks kernels onto the proposed overlays and show that the proposed overlays can deliver better throughput compared to Vivado HLS generated fully pipelined RTL implementations. We also present the approach of building a single large overlay and mapping multiple instances of kernels to the overlay to achieve effective utilization of overlay resources. The main contributions can be summarized as:

- The design and RTL implementation of an FPGA targeted overlay architecture using of an array of DSP block based fully pipelined FUs and an

island-style coarse-grained routing network (referred to as DISO), which can operate at near to the theoretical maximum frequency of the FPGA.

- A scalability analysis of DISO on the Xilinx Zynq device, including the evaluation of peak throughput (in terms of GOPS) and interconnect area overhead (in terms of LUTs/GOPS).
- An analysis of a wide variety of compute kernels using a DSP48E1 aware data flow graph based approach to ascertain the suitability of mapping multiple instances of kernels to the overlay.
- The design and parameterized RTL implementation of an enhanced FU which contains two DSP48E1 blocks for improving peak performance and reducing interconnect area overhead.
- A comprehensive comparison of the resource requirements of DISO and Dual-DISO with other overlays from the literature.
- A comprehensive analysis of the proposed overlays (DISO and Dual-DISO) by mapping a set of benchmarks (possibly with multiple instances of kernels) to the overlays and also directly to the FPGA fabric using Vivado HLS.

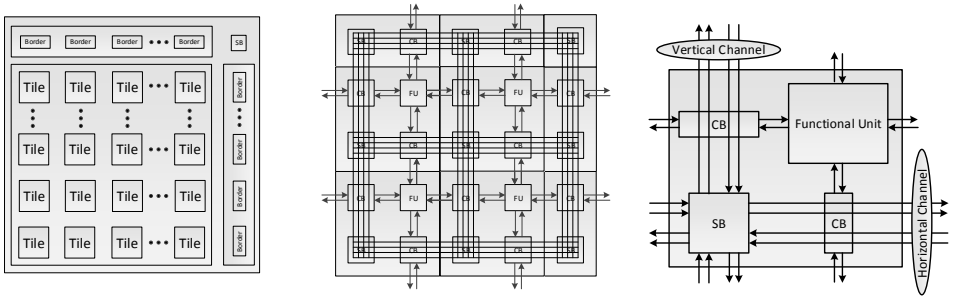
The work presented in this chapter is also discussed in

- A. K. Jain, S. A. Fahmy, and D. L. Maskell. *Efficient Overlay Architecture Based on DSP Blocks*, in Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM), Vancouver, Canada, May 2015.
- A. K. Jain, D. L. Maskell, and S. A. Fahmy. *Throughput Oriented FPGA Overlays Using DSP Blocks*, in Proceedings of the Design, Automation and Test in Europe Conference (DATE), Dresden, Germany, March 2016.

4.2 DSP Block Based Island-Style Overlay (DISO)

Building on the ideas presented in the previous chapter and the advantages of hard DSP macros for implementing high speed FUs, we examine the use of the Xilinx DSP48E1 primitive for a programmable FU in an efficient overlay architecture targeting data-parallel compute kernels, commonly implemented as pipelined circuits on FPGAs. The architecture of the proposed overlay consists of a traditional, island-style topology, arranged as a virtual homogeneous two-dimensional array of tiles as shown in Figure 4.1(a), distributed across the fine grained FPGA fabric.

The overlay instantiates the tiles and borders, where each tile instantiates virtual routing resources and an FU and each border instantiates one switch box (SB) and one connection box (CB), forming the boundary at the top and right of the array, as shown in Figure 4.1(a). This results in an overlay architecture which contains I/O around the periphery of the overlay fabric. This I/O can be connected to a FIFO or BRAM I/O data port. Figure 4.1(b) shows the architecture of a 2×2 overlay having four tiles, an east boundary (two east borders), a north boundary (two north borders) and a switch box at the north east corner. It shows that a 2×2 overlay would consist of 4 FUs, 9 SBs and 8 CBs. Extrapolating, an $N \times N$ overlay would incorporate N^2 FUs, $(N+1)^2$ SBs and $N^2 + 2 * N$ CBs. Each tile contains an FU and interconnect resources, as shown in more detail in Figure 4.1(c).



(a) Overlay block diagram. (b) Architecture of a 2×2 overlay. (c) Tile architecture.

Figure 4.1: Overlay architecture.

4.2.1 Island-style Interconnect Architecture

Routing resources in the island-style interconnect architecture are switch boxes, connection boxes, and horizontal and vertical channels consisting of routing tracks. Unlike the single-wire tracks in fine grain FPGA fabrics, the overlay tracks comprise multiple wires; in this case, 16-bits to support a 16-bit datapath. Additionally, multiple tracks can exist in both the horizontal and vertical directions, forming channels within the overlay architecture. The number of tracks in a channel is referred to as the channel width (CW), and as this increases, application routing becomes easier but with a higher area overhead. The overlay tile shown in Figure 4.1(c) has four unidirectional tracks in each channel corresponding to a CW=4. For flexibility we split the channel so that there is an equal number of tracks in each direction.

Switch boxes (SBs) connect tracks to other tracks in intersecting channels. Connection boxes (CBs) connect FU inputs and outputs to routing tracks in adjacent channels. It is also possible to change the flexibility of the SBs and CBs depending on the routing requirements of the compute kernels. In this chapter, all SBs use f_s equal to one and all CBs use f_c equal to one. For the CBs and SBs, we use multiplexers to implement each possible connection. This means routing resources contribute significantly to area overhead.

4.2.2 DSP Block Based Functional Unit

The FUs provide the resources for the mathematical or logical operations of the application and consist of a programmable PE, MUX based reordering logic and variable length shift register based synchronization logic for balancing pipeline latencies, as shown in Figure 4.2. Variable length shift registers are implemented as SLICEM shift register LUTs (SRLs) to achieve maximum performance. The FU has 4 input and 4 output ports logically organised at the 4 cardinal points. The reordering logic is basically a 4×4 crossbar switch network which allows full connectivity between FU inputs and PE inputs.

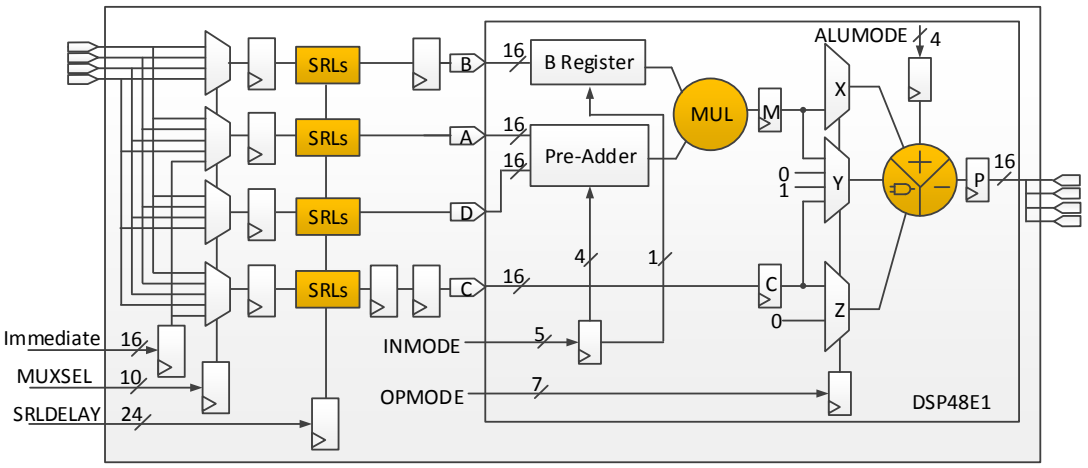


Figure 4.2: Functional unit architecture.

Instead of using the fine-grained FPGA resources to implement the programmable FU, we use the DSP48E1 primitive as a PE to implement functions in the programmable FU. As discussed in Chapter 3, the DSP48E1 primitive has a pre-adder, a multiplier, an ALU, four input ports for data, and one output port P, as shown in Figure 4.2, and can be configured to support various operations such as multiply-add, multiply-subtract, add-multiply, subtract-multiply, etc. These functions are determined by a set of control inputs that are wired to configuration registers. The DSP48E1 primitive is directly instantiated providing total control of the configuration of the primitive, and allowing us to achieve high frequency. Since the DSP48E1 can support three operations, an overlay of size $N \times N$ can support $3 * N^2$ operations. Hence the peak throughput of an overlay of size $N \times N$ is equal to $3 * N^2 * F_{max}$.

4.2.3 Architectural Optimization and Design Issues

This section describes some of the architectural optimizations to improve the performance of our overlay architecture. The overlay architecture proposed in this paper exploits very deep pipelining, and a number of issues including signal propagation delay arise in the design of the architecture. This is because the output of a pipeline stage (the FU output) can be connected to the inputs of multiple other

FUs, and multiple FUs may connect to the inputs of an FU, resulting in different signal propagation latencies.

Frequency and Throughput Optimization: To achieve a high F_{max}, thus maximizing application throughput, the FU must be operated at its maximum frequency. To achieve the highest F_{max} possible, we enable all three pipeline stages of the DSP48E1 primitive, add a register at the output of each reordering multiplexer, and register the outputs of the SRLs. As a result, the total latency of the FU is 7 clock cycles. In addition, to further increase the F_{max} and eliminate the possibility of combinational loops in the resulting HDL we use a 16-bit register at the output of each MUX in the CB.

Distinguishing PE Inputs: Any of the four inputs of the FU can connect to any of the four inputs of the PE. However, as the input pins of the DSP48E1 block are not logically equivalent, unlike that of the FPGA LUTs, we must implement reordering logic for each input pin using a multiplexer as shown in Figure 4.2. The four outputs of the FU are logically the same single output of the DSP block.

Latency Imbalance at FU Inputs: With a large pipeline latency in each node, and the need for signal timing to be correctly matched, balancing pipeline latencies at the different FUs is necessary. To maintain a full pipeline, delays are required at all paths to ensure that the inputs to each FU are correctly aligned. The most efficient way to achieve this is using variable-length shift registers, implemented using the LUT-based SRL32 primitives. The depth of the variable shift registers is set to introduce the right amount of delay for each path, and the maximum can be set to 16, 32, or 64 cycles, depending on the flexibility desired. We experimentally determine the optimal depth of the variable shift registers for our benchmark set. As long as the inputs at any node are not misaligned by more than the depth of the variable shift registers, the VPR place and route algorithms [171] can be used for placement and routing on the overlay. By doing this, we avoid the use of more complex place and route algorithms for pipelined interconnect [172]. In a later section we will describe the mapping process and the mapping tool chain used for mapping kernels to the overlay.

4.2.4 Mapping to the FPGA Fabric and Resource Usage

We synthesize and map the DISO overlay using Xilinx ISE 14.6 onto a Xilinx Zynq XC7Z020-1CLG484C. The FPGA resource usage and the overlay F_{max} is determined for the overlay with channel widths of 2 and 4.

As discussed previously, an $N \times N$ overlay would require N^2 FUs, $(N+1)^2$ SBs and $N^2 + 2 * N$ CBs. Table 4.1 shows the FPGA resource required for the FU, FU configuration registers (FUCR), SB, SB configuration registers (SBCR), CB and CB configuration registers (CBCR), for CW=2 and CW=4. The mapping of overlay components to the physical FPGA fabric and their micro-architectural resource usage is as follows.

Resource			CW=2				CW=4			
	FU	FUCR	SB	SBCR	CB	CBCR	SB	SBCR	CB	CBCR
LUTs	224	0	64	0	64	0	128	0	96	0
FFs	176	66	0	8	64	6	0	16	96	12
DSPs	1	0	0	0	0	0	0	0	0	0

Table 4.1: FPGA resource usage for DISO overlay components having CW=2 and CW=4

Resource Usage for the FU and FUCR: As mentioned in Section 4.2.2, the FU consists of a programmable PE, latency balancing logic and reordering logic. We use four 16-bit wide variable length shift registers, implemented as SLICEM shift register LUTs (SRLs) as the latency balancing logic, one on each of the four PE inputs. A LUT in a SLICEM can be configured as a variable 1 to 32 clock cycle shift register without using the flip-flops in the slice. The four LUTs in a SLICEM can then be cascaded to produce delays up to 128 clock cycles. Our overlay requires a shift register that can produce delays of up to 64 clock cycles. Hence, for each input, we cascade two registered SRLs to form a chain and use 16 chains of SRLs at each input of the PE to achieve a 16-bit variable delay of between 1 and 64 cycles. Thus each input consumes 32 LUTs and 16 FFs, resulting in 128 LUTs and 64 FFs for the complete latency balancing logic. The reordering logic requires 4 multiplexers, consuming 96 LUTs and 64 FFs. Additionally, we

require three 16-bit registers at the DSP input ports for pipeline balancing, (two at the C input and one at the B input as shown in Figure 4.2), consuming 48 FFs. Thus the total resource used by the FU is 224 LUTs, 176 FFs and 1 DSP block.

The FU configuration register includes 16 bits for DSP block configuration, 16 bits for immediate data, 10 bits for reordering multiplexer selection, 24 bits for depth selection of the 4 variable length shift registers. Hence, the FUCR consumes 66 FFs. The channel width does not impact the FU resource utilization. The resource utilization for the FU and FUCR are given in Table 4.1.

Resource Usage for the SB and SBCR: For CW=2, a SB requires four 16-bit 4:1 muxes, each consisting of 16 LUTs. The SB configuration register requires 8 bits as the selection inputs of the 4 muxes. Hence the SBCR consumes 8 FFs. For CW=4, a SB consists of eight 16-bit 4:1 muxes, requiring 32 LUTs at each cardinal point. Each mux consumes 16 LUTs. The SB configuration register requires 16 bits as the selection inputs of the 8 muxes. Hence the SBCR consumes 16 FFs. Total consumption of the SB and SBCR, for CW=2 and CW=4, is shown in Table 4.1.

Resource Usage for the CB and CBCR: For CW=2, a CB consists of two 2:1 and two 4:1 muxes (each 16-bits wide). As each mux is registered, the total resource usage is 64 LUTs and 64 FFs. The CB configuration register requires 6 bits for the selection inputs of the 4 muxes and hence the SBCR consumes 6 FFs. For CW=4, a CB consists of six 16-bit 4:1 muxes. As each mux is registered, the resource usage is 96 LUTs and 96 FFs. The CB configuration register requires 12 bits for the selection inputs of the six muxes and hence the SBCR consumes 12 FFs. Total consumption of the CB and SBCB, for CW=2 and CW=4, is shown in Table 4.1.

Resource Usage for the Overlay Tile: The overlay tile contains 1 FU, 1 SB, 2CBs and their configuration registers, while a border tile contains 1 SB, 1 CB and their configuration registers. Hence for CW=2, an overlay tile consumes 416 LUTs, 390 FFs and 1 DSP block and a border tile consumes 112 LUTs and 76

FFs. For CW=4, an overlay tile consumes 544 LUTs, 474 FFs and 1 DSP block while a border consumes 192 LUTs and 120 FFs.

Peak Throughput, Resource Usage and Frequency: The peak throughput and the resource consumption for different size overlays for a channel width CW=2 and CW=4 is presented in Tables 4.2 and 4.3, respectively.

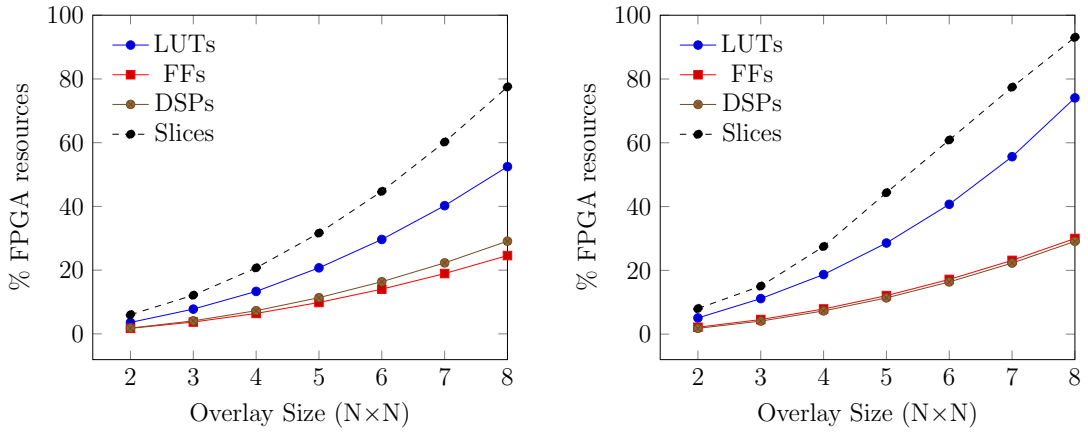
The resource consumption on Zynq, as a percentage of the total FPGA resources, for a channel width CW=2 and CW=4 is shown in Figs. 4.3(a) and 4.3(b), respectively. All of these results are post-place and route. Resource usage tracks our expectations, but it is worth noting that Slice usage becomes a limiting factor due to the reduced number of SLICEM primitives (required for the SRL latency balancing) on the lower-end Zynq fabric.

Resource type	2x2	3x3	4x4	5x5	6x6	7x7	8x8
LUTs	1910	4134	7087	11023	15761	21400	27918
FFs	1852	3950	6828	10486	14924	20142	26140
Slices	802	1614	2754	4207	5948	8007	10306
DSPs	4	9	16	25	36	49	64
f_{max}	392	382	380	370	356	350	338
T_{min}	2.55	2.62	2.63	2.7	2.81	2.85	2.96
Peak GOPS	4.70	10.31	18.24	27.75	38.44	51.45	64.89

Table 4.2: FPGA resource usage for DISO overlays with CW=2

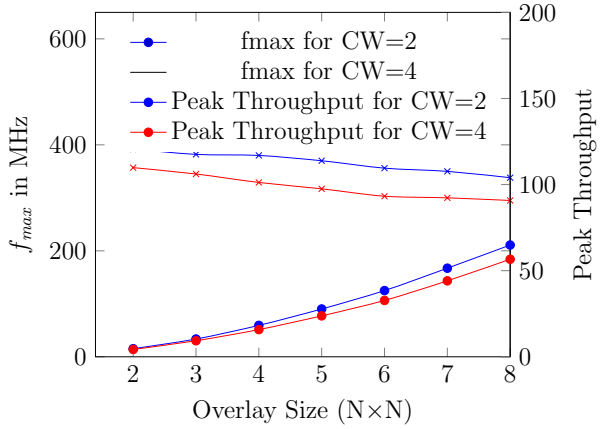
Resource type	2x2	3x3	4x4	5x5	6x6	7x7	8x8
LUTs	2702	5927	9941	15198	21658	29599	39410
FFs	2296	4866	8382	12844	18252	24606	31906
Slices	1067	2003	3655	5898	8097	10300	12379
DSPs	4	9	16	25	36	49	64
f_{max}	357	345	329	317	303	300	295
T_{min}	2.8	2.9	3.04	3.16	3.3	3.35	3.4
Peak GOPS	4.28	9.31	15.79	23.77	32.72	44.10	56.64

Table 4.3: FPGA resource usage for DISO overlays with CW=4



(a) Resources usage for CW=2.

(b) Resources usage for CW=4.



(c) Frequency trend.

Figure 4.3: Resource usage and frequency of DISO architecture on the Zynq.

Figure 4.3(c) shows the decrease in F_{max} as the size of the overlay increases for CW=2 and CW=4 on the Zynq device. A modest drop in frequency is observed, but even for large sized overlays, a frequency in excess of 300 MHZ is achieved. Figure 4.3(c) also shows the variation in peak throughput as the size of the overlay increases for CW=2 and CW=4. The Zynq fabric is able to accommodate an 8×8 DISO overlay (CW=2) which can provide a peak performance of 65 GOPS which is 24% of the maximum achievable peak performance if all the DSP blocks on Zynq were fully utilised. The 8×8 DISO overlay (CW=2) achieves an F_{max} of 338 MHz on Zynq, consuming 52% of the LUTs in the FPGA fabric, with a LUTs/GOPS count of 430. It is clear from Figure 4.3(a) and 4.3(b) that excessive LUT usage is still a limiting factor in exploiting the raw performance of all DSP blocks available on the Zynq device. Even though DISO has a significantly better

compute-to-interconnect resource usage ratio than the DSP-DySER overlay presented in chapter 3 (430 LUTs/GOPS versus 7.6K LUTs/GOPS for DSP-DySER), the compute-to-interconnect resource usage ratio of the overlay still needs improvement.

While techniques such as runtime LUT content manipulation [169] and interconnect multiplexing [132] can reduce routing network overheads and can improve the compute-to-interconnect resource usage ratio, they become unsuitable as the overlay frequency approaches the theoretical limit of the FPGA fabric.

One easy way to improve the compute-to-interconnect resource usage ratio is to include more PEs inside an FU, that is to use multiple DSP blocks. However, efficiently utilizing the DSP resources, both in terms of the optimum number of DSP blocks that can be used in an FU and being able to map applications to the new architecture, needs to be investigated. Hence in the next section, we analyse the characteristics of a number of compute kernels from the literature to ascertain the suitability of mapping multiple instances of kernels to the overlay.

4.3 Analysis of Compute Kernels

Most benchmarks used to analyse the performance of overlays are relatively small, limited by small overlay sizes. As FPGAs have increased in size, these benchmarks are no longer sufficient to fully test newer more efficient overlays. Thus, we have compiled a benchmark set (shown in Table 4.4) containing a number of small and medium sized compute kernels from the literature [162, 173, 167].

Table 4.5 and Table 4.6 presents the details of the benchmarks shown in Table 4.4. Table 4.4 shows the characteristics of the kernels after extracting the data flow graphs (DFGs), including the number of I/O nodes, graph edges, operation nodes, average parallelism, graph depth, and graph width. The graph depth is the critical path length of the graph, while the graph width is the maximum number of nodes that can execute concurrently, both of which impact the ability to efficiently map

a kernel to the overlay. The average parallelism is the ratio of the total number of operations and the graph depth. We observe that for the benchmarks selected, the average parallelism varies from 1 to 8. The DFGs contain up to 44 operation nodes, 88 edges and exhibit a depth of up to 14 and a width of up to 18.

No.	Benchmark Name	I/O nodes	DFG Characteristics (DSP-aware Characteristics)				
			graph edges	op nodes	graph depth	average parallelism	graph width
1.	chebyshev	1/1	12(10)	7(5)	7(5)	1.00(1.00)	1(1)
2.	sgfilter	2/1	27(19)	18(10)	9(5)	2.00(2.00)	4(3)
3.	mibench	3/1	22(14)	13(6)	6(4)	2.16(1.50)	3(3)
4.	qspline	7/1	50(46)	26(22)	8(7)	3.25(3.14)	7(7)
5.	poly1	2/1	15(12)	9(6)	4(3)	2.25(2.00)	4(4)
6.	poly2	2/1	14(10)	9(6)	5(3)	1.80(2.00)	3(3)
7.	poly3	6/1	17(13)	11(7)	5(3)	2.20(2.30)	4(4)
8.	poly4	5/1	13(9)	6(3)	4(2)	1.50(1.50)	2(2)
9.	poly5	3/1	43(28)	27(14)	9(6)	3.00(2.30)	6(6)
10.	poly6	3/1	72(51)	44(25)	11(9)	4.00(2.77)	11(10)
11.	poly7	3/1	62(44)	39(21)	13(8)	3.00(2.62)	10(7)
12.	poly8	3/1	51(35)	32(17)	11(5)	2.90(3.40)	8(8)
13.	fft	6/4	24(22)	10(8)	3(3)	3.33(2.66)	4(4)
14.	kmeans	16/1	39(36)	23(20)	9(7)	2.55(2.85)	8(8)
15.	mm	16/1	31(24)	15(8)	8(8)	1.88(1.00)	8(1)
16.	mri	11/2	24(20)	11(7)	6(5)	1.83(1.40)	4(2)
17.	spmv	16/2	30(24)	14(8)	4(4)	3.50(2.00)	8(2)
18.	stencil	15/2	30(24)	14(8)	5(3)	2.80(2.66)	6(4)
19.	conv	24/8	40(32)	16(8)	2(1)	8.00(8.00)	8(8)
20.	radar	10/2	18(16)	8(6)	3(3)	2.66(2.00)	4(2)
21.	arf	26/2	58(50)	28(20)	8(8)	3.50(2.50)	8(4)
22.	fir2	17/1	47(32)	23(8)	9(8)	2.55(1.00)	8(1)
23.	hornerbezier	12/4	32(22)	14(8)	4(3)	3.50(2.66)	5(4)
24.	motionvector	25/4	52(40)	24(12)	4(3)	6.00(4.00)	12(4)
25.	atax	12/3	123(99)	60(36)	6(6)	12.00(7.20)	27(9)
26.	bicg	15/6	66(54)	30(18)	3(3)	10.00(6.00)	18(6)
27.	trmm	18/9	108(90)	54(36)	4(4)	13.50(9.00)	27(9)
28.	syrk	18/9	126(99)	72(45)	5(4)	14.40(11.25)	36(18)

Table 4.4: The characteristics of the benchmarks

No.	Kernel	DAG Expressions
1.	chebyshev	$out = (x * (x * (16 * x^2 - 20) * x + 5))$
2.	sgfilter	$out = (x * (x * (7 * x - 76 * y + 7) + y * (92 * y - 39) + 7) - y * (y * (984 * y + 46) + 46) - 75)$
3.	mibench	$out = (x * (x + 2 * y + 6 * z + 43) + y * (y + 6 * z + 43) + z * (9 * z + 1))$
4.	qspline	$out = (z * u^4 + 4 * a * u^3 * v + 6 * b * u^2 * v^2 + 4 * w * v^3 * u + q * v^4)$
5.	poly1	$out = (x^2 * (x + y - 1) - y^2 * (x - y + 1))$
6.	poly2	$out = (x^2 * (2 * x^2 - 3 * y) + (y - 1)^2 * y^2)$
7.	poly3	$out = (15 * (x + 2 * t - 11 * v^2) + 25 * u * y - 80 * z)$
8.	poly4	$out = (c * (c + n) + h * (m + a * c))$
9.	poly5	$out = x * x * y * (y * (2 * z - x - 144) - z * (207 - z) - 3416)$ $+x * y * (y * (288 * z - 5184) + z * (78 * z - 9504) - c_2)$ $+x * z * (62208 + z * (z - 432)) - c_1$
10.	poly6	$out = t * (t * (z^2 * (864 - z) - c_3 * z + x^2 * (x * (32 - z) + z * (z - 72)) - x * z * (2592 + 87 * z)))$ $-t * ((x * (x * (z * (4 * z + 6 * x) - 432 * z) - z * (414 * z - 20736)) - (3456 * z^2 - c_4 * z)))$ $-(z * x^2 * (8 * x + 1728) + c_1 * x + c_2 * z)$
11.	poly7	$out = t^2 * y^2 * (4 * y + 96) + t^2 * (2304 * x + x * y * (8 * x - 12 * y - 160))$ $+t^3(x * (128 + y * (x - 2 * y - 24)) + y^2(y + 24)) + (t * x * (13824 + 96 * y + 20 * y(x - y)))$ $+(x * (y * (16 * x + 1152) + 27648))$
12.	poly8	$out = t^3 * (y * (y * (y - z) + 1728 + 71 * z) - 464 * z - 13824)$ $+t^2 * (y * (y * (4y + 288) + 6912 + 360 * z) + 55296 + z * (6 * z - 4312))$ $+(t * z * (432 * y - 13824)) + (z * (z - 13824))$
13.	fft	$out_0 = (in_0 - (in_1 * in_2 + in_3 * in_4))$ $out_1 = (in_0 + (in_1 * in_2 + in_3 * in_4))$ $out_2 = (in_5 - (in_1 * in_4 + in_3 * in_2))$ $out_3 = (in_5 + (in_1 * in_4 + in_3 * in_2))$
14.	kmeans	$out = (in_0 - in_1)^2 + (in_2 - in_3)^2 + (in_4 - in_5)^2 + (in_6 - in_7)^2$ $+(in_8 - in_9)^2 + (in_{10} - in_{11})^2 + (in_{12} - in_{13})^2 + (in_{14} - in_{15})^2$
15.	mm	$out = (in_0 * in_1) + (in_2 * in_3) + (in_4 * in_5) + (in_6 * in_7)$ $+(in_8 * in_9) + (in_{10} * in_{11}) + (in_{12} * in_{13}) + (in_{14} * in_{15})$
16.	mri	$out_0 = (in_6 * (in_0 * in_1 + in_2 * in_3 + in_4 * in_5) + in_7) * (in_9 \| in_{10})$ $out_1 = (in_6 * (in_0 * in_1 + in_2 * in_3 + in_4 * in_5) + in_8) * (in_9 \| in_{10})$
17.	spmv	$out_0 = (in_0 * in_1) + (in_2 * in_3) + (in_4 * in_5) + (in_6 * in_7)$ $out_1 = (in_8 * in_9) + (in_{10} * in_{11}) + (in_{12} * in_{13}) + (in_{14} * in_{15})$
18.	stencil	$out_0 = (in_0 + in_1 + in_2 + in_3 + in_4 + in_5) * in_6 - in_7$ $out_1 = (in_8 + in_9 + in_{10} + in_{11} + in_{12} + in_{13}) * in_6 - in_{14}$
19.	conv	$out_0 = (in_0 + (in_1 * in_2))$ $out_1 = (in_3 + (in_4 * in_5))$ $out_2 = (in_6 + (in_7 * in_8))$ $out_3 = (in_9 + (in_{10} * in_{11}))$ $out_4 = (in_{12} + (in_{13} * in_{14}))$ $out_5 = (in_{15} + (in_{16} * in_{17}))$ $out_6 = (in_{18} + (in_{19} * in_{20}))$ $out_7 = (in_{21} + (in_{22} * in_{23}))$
20.	radar	$out_0 = (in_0 * in_1 + in_3 * in_4) * in_2$ $out_1 = (in_6 * in_7 + in_8 * in_9) * in_5$
21.	arf	$out_0 = (((in_1 * in_2 + in_3 * in_4) + in_9) * in_{11} + ((in_1 * in_2 + in_3 * in_4) + in_9) * in_{12}) * in_{19} + in_{15} * in_{16}$ $+in_{17} * in_{18} + (((in_5 * in_6 + in_7 * in_8) + in_{10}) * in_{13} + ((in_5 * in_6 + in_7 * in_8) + in_{10}) * in_{14}) * in_{20}$ $out_1 = (((in_1 * in_2 + in_3 * in_4) + in_9) * in_{11} + ((in_1 * in_2 + in_3 * in_4) + in_9) * in_{12}) * in_{25} + in_{21} * in_{22}$ $+in_{23} * in_{24} + (((in_5 * in_6 + in_7 * in_8) + in_{10}) * in_{13} + ((in_5 * in_6 + in_7 * in_8) + in_{10}) * in_{14}) * in_{26}$
22.	fir2	$out = (in_1 + in_2) * b_{01} + (in_3 + in_4) * b_{01} + (in_5 + in_6) * b_{01} + (in_7 + in_8) * b_{01} + (in_9 + in_{10}) * b_{01}$ $+(in_{11} + in_{12}) * b_{01} + (in_{13} + in_{14}) * b_{01} + (in_{15} + in_{16}) * b_{01}$
23.	hornerbezier	$out_0 = (b_1 + (b_1 * (datasize_0 * in_0 + b_1)))$ $out_1 = (b_1 + datasize_1 * in_1)$ $out_2 = (in_5 * mem1out + (in_3 * in_4) * mem2out)$ $out_3 = (b_1 + (b_1 * (datasize_2 * in_2 + b_1)))$
24.	motionvector	$out_0 = in_{11} * in_{12} + in_{13} * in_{14} + in_{15} * in_{16} + b_0$ $out_1 = in_{21} * in_{22} + in_{23} * in_{24} + in_{25} * in_{26} + b_0$ $out_2 = in_{31} * in_{32} + in_{33} * in_{34} + in_{35} * in_{36} + b_0$ $out_3 = in_{41} * in_{42} + in_{43} * in_{44} + in_{45} * in_{46} + b_0$

Table 4.5: Kernel benchmarks

atax

```
for (i = 0; i < nx; i++)
        y[i] = 0;
for (i = 0; i < ny; i++)
{
        tmp[i] = 0;
        for (j = 0; j < ny; j++)
                tmp[i] = tmp[i] + A[i][j] * x[j];
        for (j = 0; j < ny; j++)
                y[j] = y[j] + A[i][j] * tmp[i];
}
```

bicg

```
for (i = 0; i < ny; i++)
        s[i] = 0;
for (i = 0; i < nx; i++)
{
        q[i] = 0;
        for (j = 0; j < ny; j++)
        {
                s[j] = s[j] + r[i] * A[i][j];
                q[i] = q[i] + A[i][j] * p[j];
        }
}
```

trmm

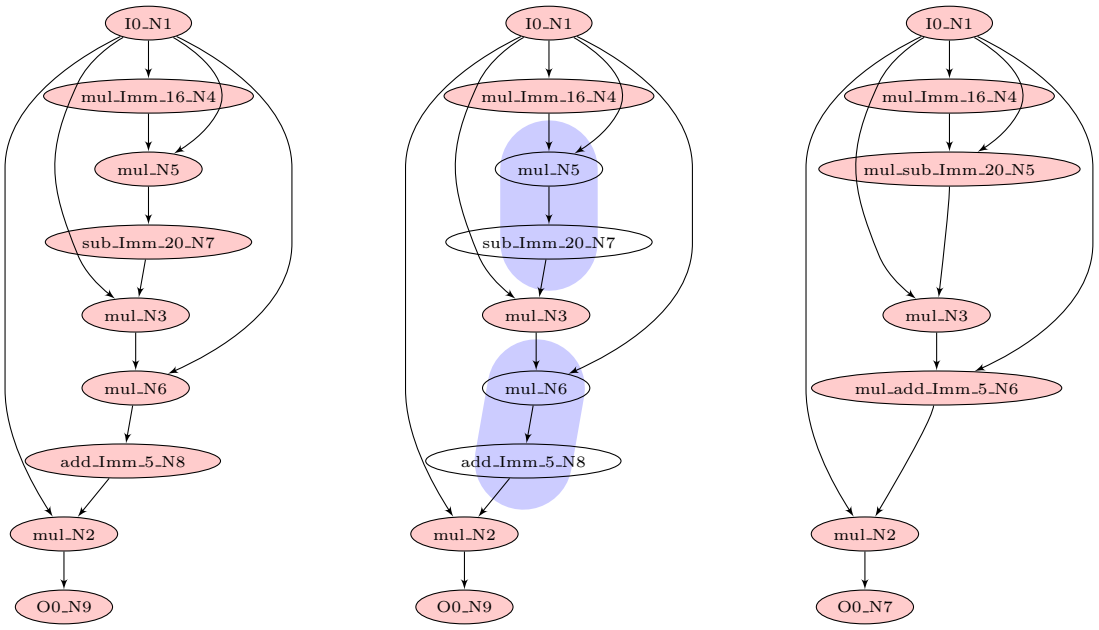
```
for (i = 1; i < ni; i++)
        for (j = 0; j < ni; j++)
                for (k = 0; k < i; k++)
                        B[i][j] += alpha * A[i][k] * B[j][k];
```

syrk

```
for (i = 0; i < ni; i++)
        for (j = 0; j < ni; j++)
                C[i][j] *= beta;
for (i = 0; i < ni; i++)
        for (j = 0; j < ni; j++)
                for (k = 0; k < nj; k++)
                        C[i][j] += alpha * A[i][k] * A[j][k];
```

Table 4.6: Linear algebra kernels

Mapping these kernels to DSP blocks allows us to reduce the number of FUs required by combining simple arithmetic operations into the more complex compound instructions supported by the DSP block, as shown in Figure 4.4 for the *chebyshev* benchmark. We perform this transformation on all of the kernels and re-analyse the benchmark characteristics for DSP-aware DFGs (shown in brackets in Table 4.4). It is clear from the *op nodes* column that an overlay with at least 45 DSP blocks is needed to accommodate all benchmarks, down from the 72 single operation nodes which would be needed if the DSP block capabilities were ignored.



(a) Input DFG (b) Node-merging (c) Mapped DFG

Figure 4.4: DSP48E1 aware DFG generation.

While DSP aware mapping does reduce the number of FUs required, the I/O requirements remain unchanged. Additionally, using a large overlay and mapping multiple instances of the smaller kernels to it impacts the availability of both compute and I/O resources. As the size of an island-style overlay increases, the number of I/O interfaces grows linearly while the number of compute tiles grow quadratically. Thus, an $N \times N$ overlay supports N^2 FUs, but as the I/O is determined by the overlay perimeter it is proportional to N (e.g. $4N$, $8N$, $12N$ depending on the number of I/O nodes per tile). The scalability curves for the three different architectures with different numbers of I/O nodes per tile, assuming a single DSP block based FU, are shown (as the grey dashed lines) in Figure 4.5. Here, an 8×8 $4N$ overlay has 64 DSP nodes and 32 I/O nodes, while an 8×8 $8N$ overlay has the same 64 DSP nodes but now has 64 I/O nodes. Figure 4.5 also shows plots of the I/O vs DSP nodes required for multiple replicated instances (represented as the symbols) of the compute kernels (represented as the solid lines) from Table 4.4. It can be seen that the replicated kernels towards the top left are I/O bound and require more I/O nodes, as provided by the $8N$ and $12N$ architectures, while the kernels with points below the architecture curves are compute bound and can make use of the available FUs for the given I/O

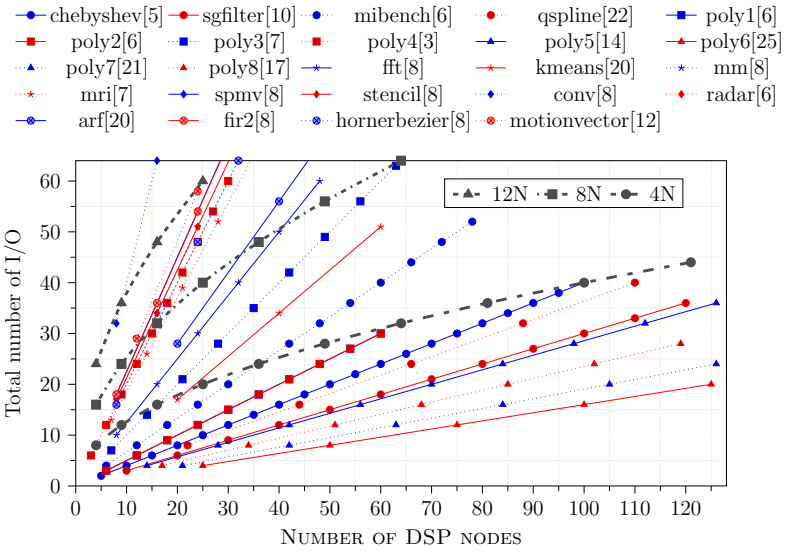


Figure 4.5: I/O scalability analysis.

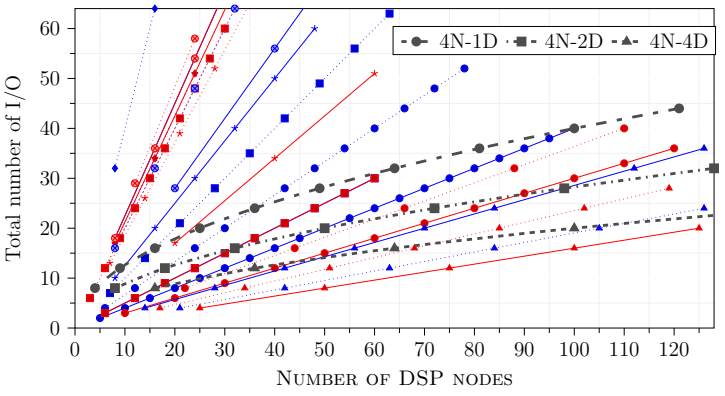


Figure 4.6: DSP scalability analysis (4N architecture).

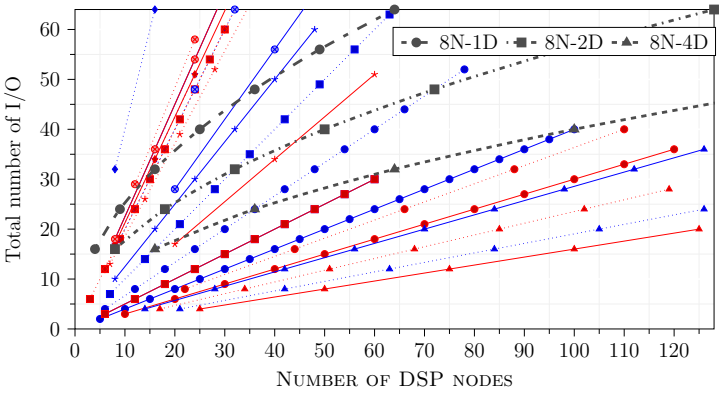


Figure 4.7: DSP scalability analysis (8N architecture).

architecture. For example, the replicated compute kernels towards the bottom right of Figure 4.5 have a limited I/O requirement and can consume the majority of DSP blocks in a $4N$ architecture.

To determine the impact of adding additional compute nodes into the FU we re-examine the scalability curves for the $4N$ architecture with an FU consisting of one, two and four DSP blocks, referred to as *4N-1D*, *4N-2D* and *4N-4D*, respectively. The resulting scalability curves, along with the I/O and DSP node requirements for replicated instances of the compute kernels, are shown in Figure 4.6. It can be seen that the *4N-4D* architecture is only suitable for a very small number of kernels (those below the *4N-4D* curve), with a significant underutilization of DSP blocks for the other kernels, and, as such, is not considered further. Similarly, the scalability curves for the $8N$ architecture with an FU consisting of one, two, and four DSP blocks, referred to as *8N-1D*, *8N-2D* and *8N-4D* in Figure 4.7, respectively, were also considered.

The benefit of using a *4N-2D* over a *4N-1D* architecture, or an *8N-4D* over a *4N-1D* architecture, is the reduced cost of the routing network per DSP block. The FPGA resource cost of the *4N-2D* architecture is 100 Slices per DSP block, compared to 160 Slices per DSP block for the *4N-1D* architecture. Thus, a 128 DSP block *4N-2D* overlay would consume 12.8K Slices while a 128 DSP block *4N-1D* overlay would consume 20.5K Slices. Due to the low cost of the *4N-2D* architecture, an overlay with 128 DSP blocks can easily fit onto a Xilinx Zynq device having 13K Slices. In the next section, we describe the detailed architecture of the dual-DSP block based overlay and its mapping to FPGA fabric.

4.4 Dual-DSP Block Based Island-Style Overlay (Dual-DISO)

We now examine the use of a cluster of DSP48E1 primitives as a programmable FU in an efficient overlay architecture targeting data-parallel compute kernels. We use a conventional tile-based island-style overlay architecture, similar to the DISO architecture presented earlier, where a tile consists of an FU and programmable routing resource, consisting of one switch box (SB), two connection boxes (CB))

and horizontal and vertical routing tracks, all 16-bits wide to support a 16-bit datapath. The number of tracks in a channel is referred to as the channel width (CW), and as this increases, application routing becomes easier but with a higher area overhead. Multiplexer-based connection boxes and switch boxes connect tracks to the FU and other tracks in intersecting channels, respectively.

4.4.1 Dual-DSP Block Based Functional Unit

To improve the compute-to-interconnect resource usage ratio and hence reduce the number of LUTs per DSP block in the implementation of the overlay, we improve on the FU of the DISO architecture by using two DSP48E1 blocks. The dual PE FU, shown in Figure 4.8, has the same 4-input, 4-output structure as DISO, allowing it to connect to any of the four adjacent channels. To ensure signal timing across the array is correctly matched, pipeline latencies at the different FUs must be balanced by introducing a delay into each path. This is again achieved by adding variable-length shift registers, implemented using LUT-based SRL32 primitives at each FU input, which can be set to a maximum of 16, 32, or 64 cycles, depending on the flexibility required. As long as the inputs at any node are not misaligned by more than the depth of the variable shift registers, the VPR place and route algorithms can be used for placement and routing on the overlay. Multiplexer-based reordering logic is used to connect the delayed inputs of the FU to the DSP blocks. This is required as any of the four inputs of the FU can connect to any of the four inputs of a DSP48E1 primitive which, unlike LUTs, are not logically equivalent.

The two DSP blocks are connected in series, with four additional registers added to each input of the second DSP block for pipeline balancing. Lastly, the output from either DSP block can be selected as the FU output. To maintain a high F_{max}, all three pipeline stages in the DSP48E1 primitives are enabled. Additional registers are added at the output of each reordering multiplexer, at the input selector of the second DSP block and at the FU output. These registers, along with the

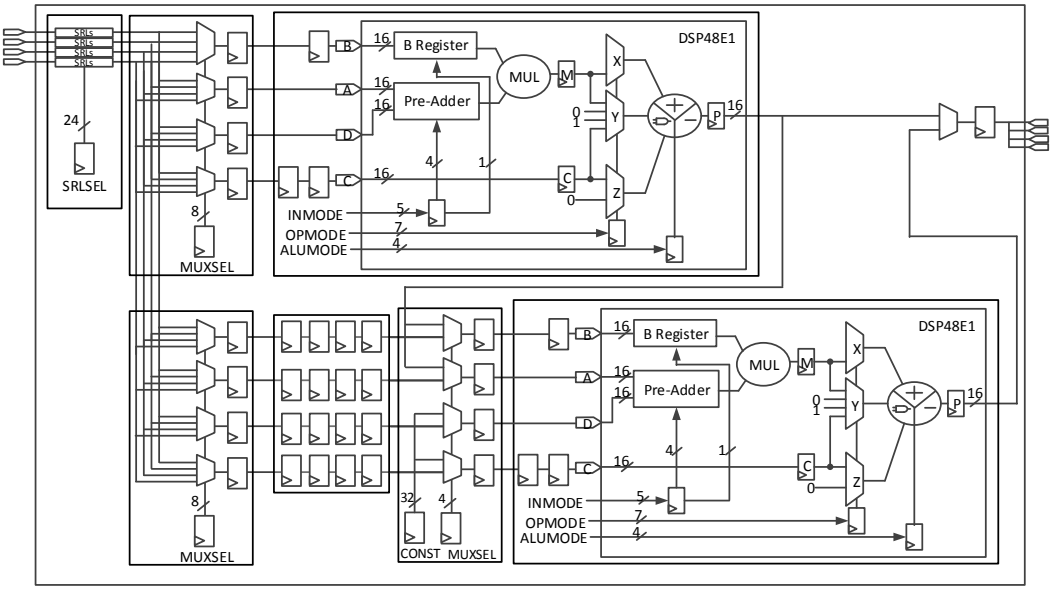


Figure 4.8: Architecture of Dual-DSP block based functional unit.

registered outputs of the SRLs result in a total FU latency of 8 clock cycles when using only one DSP block and 13 clock cycles when using both DSP blocks.

4.4.2 Resource Usage when Mapped to the FPGA Fabric

We synthesize and map an overlay with a CW=2, one I/O per row/column and an FU with 2 DSP blocks, referred to as the CW2-4N-2D overlay, along with an overlay with a CW=4, two I/O per row/column and an FU with 2 DSP blocks, referred to as the CW4-8N-2D overlay, using Vivado 2014.2 targeting the Xilinx Zynq XC7Z020.

An $N \times N$ CW2-4N-2D overlay would require N^2 FUs, $(N+1)^2$ SBs and $2 * N * (N+1)$ CBs. Table 4.7 shows the FPGA resources required for the FU, FU configuration registers (FUCR), SB, SB configuration registers (SBCR), CB and CB configuration registers (CBCR) for both the CW2-4N-2D and the CW4-8N-2D (in brackets) overlays. Note that there is no difference between the FU and FUCR for both overlays, the difference being restricted to just the routing in a tile. Next,

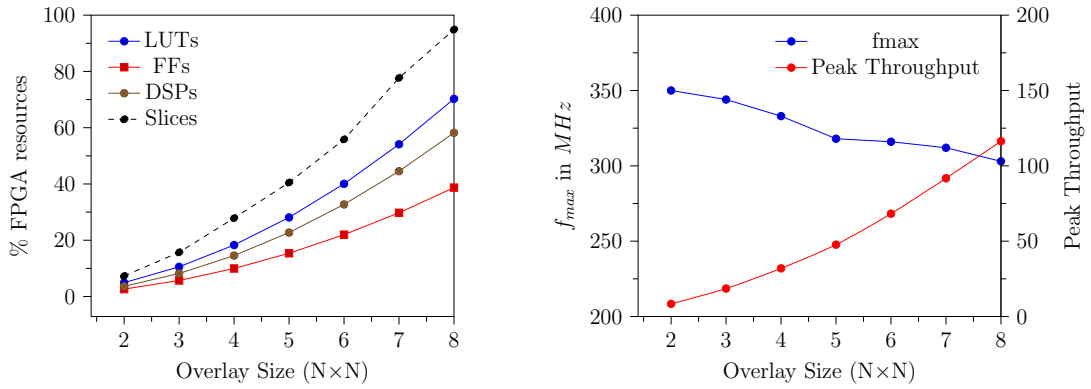
we describe the individual overlay component mapping onto the physical FPGA fabric and their resource usage.

As mentioned in Section 4.4.1, the FU consists of programmable PEs, latency balancing logic and reordering logic. We use four 16-bit wide variable length shift registers, implemented as SLICEM shift register LUTs (SRLs) as the latency balancing logic, one on each of the four PE inputs. As we require a maximum delay of 64 clock cycles for our benchmark set we use two cascaded SRLs to form a chain and use 16 chains at each input of the PE to achieve a 16-bit variable delay of between 1 and 64 cycles. Thus each input consumes 32 LUTs and 16 FFs, resulting in 128 LUTs and 64 FFs for the complete latency balancing logic. The reordering logic requires 4 multiplexers with registered outputs, at the input of each PE, consuming 128 LUTs and 128 FFs in total. Additionally, we require three 16-bit registers at the DSP input ports (as shown in Figure 4.8) for each PE, consuming 96 FFs. Delay lines at the four inputs of the second PE require 64 LUTs and 64 FFs and the second PE input selection logic requires 32 LUTs and 64 FFs. Finally at the output of the FU, we require a multiplexer with a registered output, consuming 8 LUTs and 16 FFs. Thus the total FU resource usage is 360 LUTs, 432 FFs and 2 DSP blocks. The FU configuration register includes 16 bits for each DSP block configuration, 16 bits for the two immediate operands, 8 bits for each reordering logic, 4 bits for the second PE input selection logic, 1 bit for the FU output selection logic and 24 bits for depth selection of the latency balancing logic. Hence, the FUCR consumes 109 FFs. The FU and FUCR resource utilization are given in Table 4.7.

Resource	FU	FUCR	SB	SBCR	CB	CBCR
LUTs	360	0	64 (128)	0	48 (96)	0
FFs	432	109	0	8 (16)	64 (128)	6 (12)
DSPs	2	0	0	0	0	0

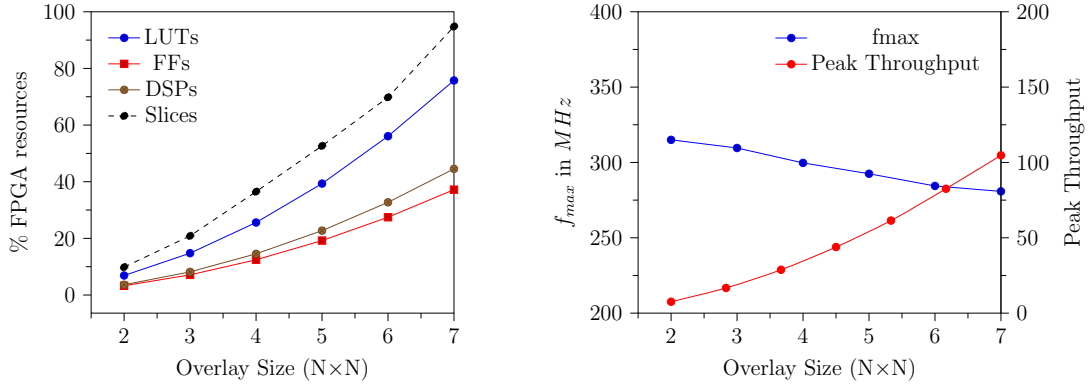
Table 4.7: FPGA resource usage for Dual-DISO overlay components

For the CW=2 overlay, a SB requires four 16-bit 4×1 muxes, each consisting of 16 LUTs. The SB configuration register needs 8 bits, 2 for each of the 4 muxes.



(a) % resource usage (b) F_{max} and Peak Throughput

Figure 4.9: Zynq-7020 CW2-4N-2D Dual-DISO overlay scalability results.



(a) % resource usage (b) F_{max} and Peak Throughput

Figure 4.10: Zynq-7020 CW4-4N-2D Dual-DISO overlay scalability results.

Hence the SBCR consumes 8 FFs. The total SB and SBCR resource usage, for CW=2, is shown in Table 4.7. A CB consists of two 2×1 and two 4×1 muxes (each 16-bits wide). As each mux is registered, the total resource usage is 48 LUTs and 64 FFs. The CB configuration register requires 6 bits for the selection inputs of the 4 muxes and hence the SBCR consumes 6 FFs. The total CB and CBCR resource usage, for CW=2, is shown in Table 4.7.

The CW2-4N-2D overlay tile contains 1 FU, 1 SB, 2CBs and their configuration registers, while a border tile contains 1 SB, 1 CB and the configuration registers. Thus, an overlay tile consumes 520 LUTs, 625 FFs and 2 DSP blocks while a border tile consumes 112 LUTs and 76 FFs. The post-place and route resource consumption on Zynq, as a percentage of total FPGA resources, is shown in Figure 4.9(a). Figure 4.10(a) shows the resource consumption results for CW4-4N-2D

overlay. It is clear that the Zynq fabric can accommodate 7×7 CW4-4N-2D overlay due to the high resource consumption of routing resources.

The overlay operating frequency approaches the DSP theoretical limit of 400 MHz on Zynq for small overlays, but as the overlay grows in size the frequency decreases slightly, as shown in Figure 4.9(b) and Figure 4.10(b). Since the DSP48E1 can support three operations, an $N \times N$ overlay can support a maximum of $6 \times N^2$ operations, and hence the peak throughput is $6 * N^2 \times F_{max}$ operations per second, as shown in Figure 4.9(b) and Figure 4.10(b) for the different overlay sizes.

The CW2-4N-2D overlay requires 109 configuration bits for the dual-DSP FU and 20 configuration bits for programming the routing network tile. Thus, an 8×8 overlay has a configuration size of 9100 bits (1137 Bytes).

The Zynq fabric is able to accommodate an 8×8 CW=2 Dual-DISO overlay and can provide a peak performance of 115 GOPS at an F_{max} of 300 MHz, consuming 70% of the LUTs in the FPGA fabric with a LUTs/GOPS ratio of 320. Figure 4.11 shows the mapping of the Dual-DISO overlay, for different array sizes (from a single tile up to an 8×8 array of tiles) onto the Zynq Fabric. We also mapped the overlay to a mid-sized Virtex-7 (XC7VX690T-2) device where we were able to implement a 20×20 CW=2 overlay, resulting in a frequency of 380 MHz and a peak throughput of 912 GOPS.

4.4.3 Discussion

A quantitative comparison of the proposed overlay architecture with some existing overlays from the research literature is given in Table 4.8. For the different FPGA devices and overlay sizes we compare the resource usage in terms of the LUTs used and the percentage of the total LUTs used in the target device (shown in brackets), the frequency, the maximum number of simultaneous arithmetic operations (Max OPs), the peak throughput of the arithmetic operations in GOPS and the interconnect area overhead in terms of LUTs/GOPS.

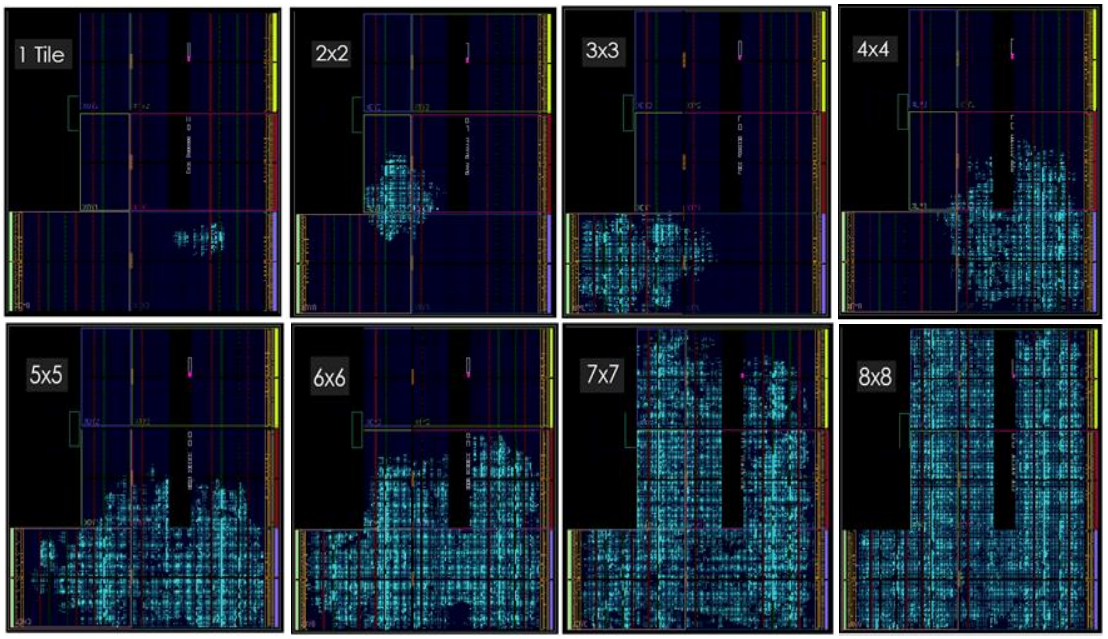


Figure 4.11: Physical mapping of Dual-DISO overlay on Zynq fabric.

Resource	IF [137]	IF (opt) [137]	DSP-DySER	DISO	Dual-DISO	Dual-DISO-V7
Device	XC5VLX330	XC5VLX330	XC7Z020	XC7Z020	XC7Z020	XC7VX690T
Slices|LUTs	51.8K|207K	51.8K|207K	13.3K|53K	13.3K|53K	13.3K|53K	108.3K|433.2K
Overlay	14×14	14×14	6×6	8×8	8×8	20×20
LUTs used	91K(44%)	50K(24%)	48K(90%)	28K(52%)	37K(70%)	228K(52%)
Fmax (MHz)	131	148	175	338	300	380
Max OPs	196	196	36	192	384	2400
GOPS	25.6	29	6.3	65	115	912
LUTs/GOPS	3550	1725	7620	430	320	250

Table 4.8: Quantitative comparison of overlays

The Xilinx Zynq-7020 fabric consists of 220 DSP blocks, with a theoretical maximum frequency of 400 MHz, and each of these can support up to 3 arithmetic operations, resulting in a peak throughput of 264 GOPS. We observe that for the DSP-DySER overlay, it is possible to fit an array of 36 DSP blocks (16% of the total DSP blocks), while for the DISO and Dual-DISO architectures it is possible to fit 64 (30%) and 128 (60%) DSP blocks, respectively. In terms of the Peak GOPS, DSP-DySER achieves 6.3 GOPS, while the DISO and Dual-DISO overlays achieve 65 GOPS and 115 GOPS, representing 2.4%, 25% and 44% of the maximum achievable GOPS, respectively.

Figure 4.12 shows the LUTs/GOPS, for the various overlays, and clearly shows that the Dual-DISO overlay has the lowest interconnect area overhead (approaching the ideal interconnect area overhead of 200 LUTs/GOPS for the Zynq device). A resource balanced overlay on Zynq would consist all of the 220 DSP blocks for computations (resulting in 264 GOPS) and all 53K LUTs for interconnect, resulting in the ideal interconnect area overhead of 200 LUTs/GOPS.

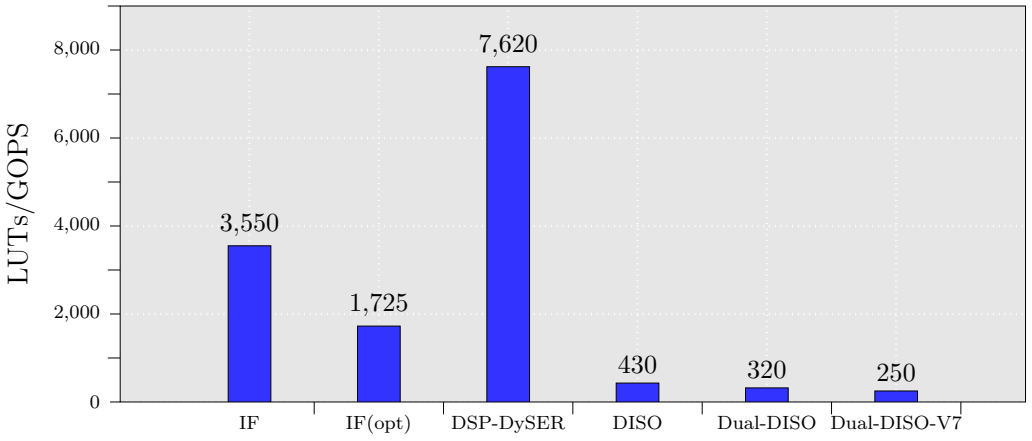


Figure 4.12: Comparison of interconnect area overhead.

4.5 Evaluating Kernel Mapping

Our mapping tool, which is described in detail in Chapter 6, takes a C description of the compute kernel and maps it to the DISO and Dual-DISO overlay using the steps described in Chapter 6. As a first step, we compare the number of overlay tiles needed for each of the benchmarks, when mapping on DSP-DySER, DISO and Dual-DISO overlay, as shown in Figure 4.13, where the numbers on the x-axis relate to the benchmark number from Table 4.4. We observe a reduction in the number of tiles required for DISO and Dual-DISO of up to 50% and 69%, respectively. The advantage of the DISO and Dual-DISO architectures becomes apparent by examining specific benchmarks, such as *poly7* (benchmark 11), where only a single DFG instance can fit onto an 8×8 DSP-DySER overlay (as a minimum it requires a 7×7 overlay). Since a 6×6 DSP-DySER is the largest which can fit on the Zynq fabric, it is not possible to support the *poly7* benchmark, which requires 39 DSP blocks on DSP-DySER. However, using the DISO architecture,

not only one but three separate instances of the *poly7* benchmark are able to fit onto an 8×8 DISO overlay, utilising 63 of the 64 tiles. Using the Dual-DISO architecture, 4 separate instances of the *poly7* benchmark are able to fit onto an 8×8 Dual-DISO overlay, utilising 56 of the 64 tiles. It is clear that architecture-focused overlays (DISO and Dual-DISO) can support larger compute kernels (even multiple instances of compute kernels) for a given resource budget, which other overlays, such as DSP-DySER, are unable to do.

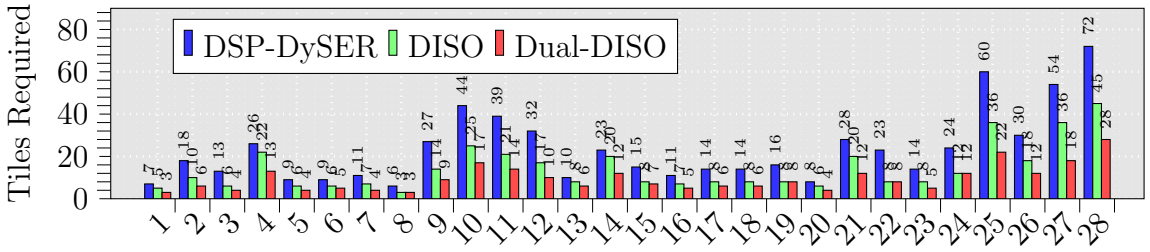


Figure 4.13: The number of tiles required for the kernels in table 4.4.

4.5.1 DISO

In this section, we first compare the DISO architecture with DSP-DySER in terms of resource usage for implementing a set of compute kernels. Next, we demonstrate the throughput advantages of the DISO architecture compared to Vivado-HLS generated RTL implementations.

Comparing with DSP-DySER: In chapter 3, a set of compute kernels were mapped onto a 5×5 DSP-DySER. In this section, we consider the same benchmark set (which is also shown as benchmarks 13-20 in Table 4.4) and try to map the compute kernels onto the DISO overlay so that we can compare the benefits of using DISO over DSP-DySER. We observe that a 5×5 DISO overlay can be used to map the all of kernels in the benchmark subset. While both DSP-DySER and DISO overlays have similar DSP usage, both consuming 25 DSP blocks (5×5 overlay), the LUT and FF requirement at (33875 and 11023) and (13390 and 10486), respectively, shows a significant reduction for the DISO overlay, as shown in Figure 4.14.

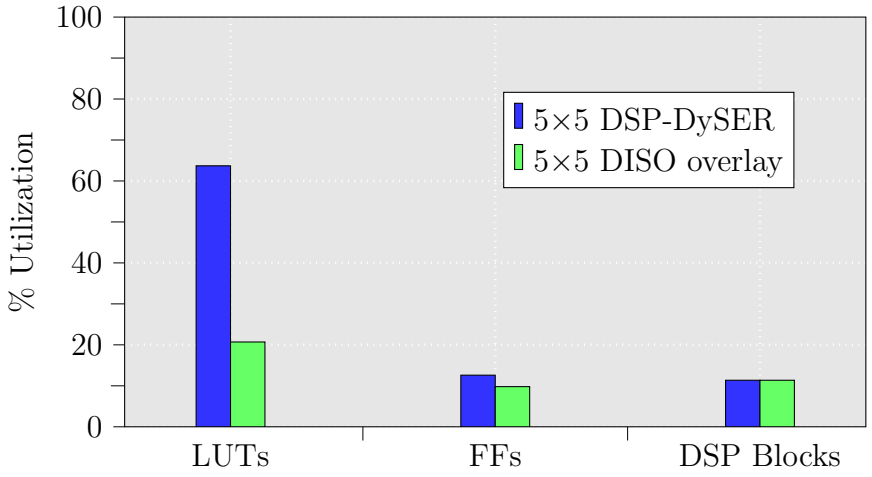


Figure 4.14: Comparison of overlay resources required for the benchmark set.

Comparing with Vivado-HLS generated RTL implementations: The resource usage of the overlay architecture depends on two main parameters: the overlay size and the channel width (CW). As it is desirable to minimize both of these while still maintaining the routability of the benchmarks, we conducted an experiment to find out the minimum DISO overlay size required for the benchmarks while at the same time ensuring routability. We consider two set of benchmarks from Table 4.4, set-I (benchmarks 1-8) and set-II (benchmarks 25-28). Set-I consists of small benchmarks having a relatively small graph width and average parallelism while set-II consists of larger benchmarks having a relatively higher graph width and average parallelism.

It is clear from Table 4.9 that DSP block aware node merging reduces the number of nodes in the DFG, and hence the number of tiles required (up to 53% for benchmark set-I and up to 40% for benchmark set-II). Table 4.9 also shows that the set-II benchmarks are not routable for an architecture with CW=2, whereas for CW=4, all benchmarks are routable. Hence, we prototype 2 overlay architectures: DISO-I (size=5×5) with CW=2 for benchmark set-I and DISO-II (size=7×7) with CW=4 for benchmark set-II. Figure 4.15 shows the mapping of set-I benchmarks on DISO-I (size=5×5) with CW=2 and Figure 4.16 shows the mapping of set-II benchmarks on DISO-II (size=7×7) with CW=4.

In order to determine the required depth for the variable length shift registers we

Benchmark	Benchmark Characteristics				Routability		DISO Overlay Results		
	i/o nodes	op nodes	merged nodes	savings	CW=2	CW=4	Latency	MLI	GOPS
chebyshev	1/1	7	5	28%	3×3	3×3	49	36	2.59
sgfilter	2/1	18	10	44%	4×4	4×4	54	31	6.66
mibench	3/1	13	6	53%	3×3	3×3	47	35	4.81
qspline	7/1	26	22	15%	5×5	5×5	76	64	9.62
poly1	2/1	9	6	33%	3×3	3×3	34	22	3.33
poly2	2/1	9	6	33%	3×3	3×3	29	7	3.33
poly3	6/1	11	7	36%	3×3	3×3	31	11	4.07
poly4	5/1	6	3	50%	2×2	2×2	24	12	2.22
atax	12/3	60	36	40%	—	6×6	72	58	18.0
bicg	15/6	30	18	40%	—	6×6	46	32	9.0
trmm	18/9	54	36	33%	—	7×7	58	30	16.2
syrk	18/9	72	45	37%	—	7×7	41	19	21.6

Table 4.9: Experimental results for DISO implementations of the benchmark set

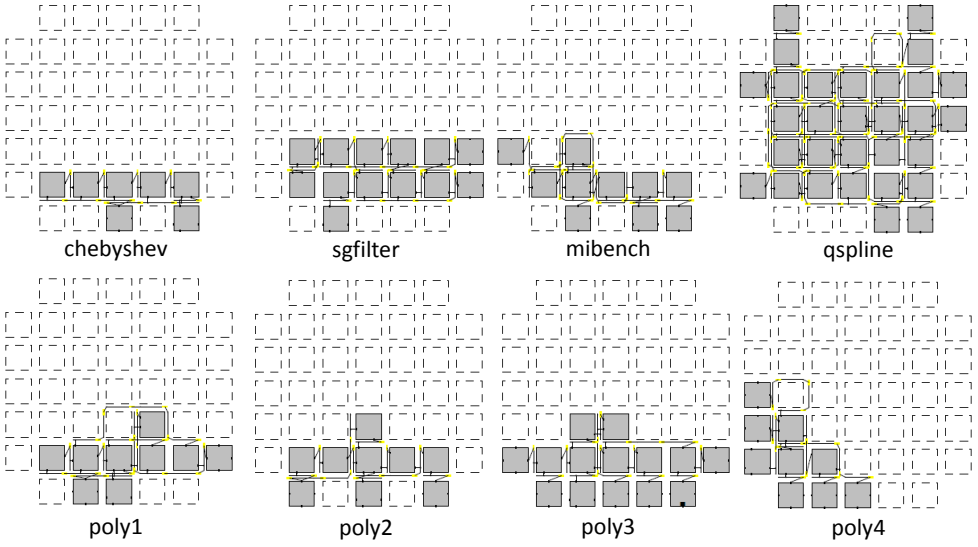


Figure 4.15: Benchmark set-I mapped on DISO-I.

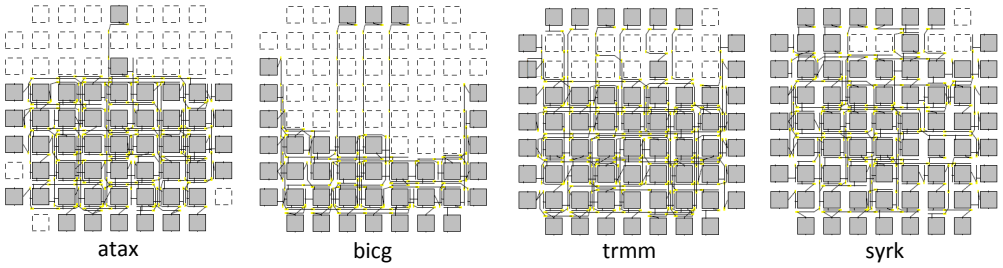


Figure 4.16: Benchmark set-II mapped on DISO-II.

conducted an experiment to find the maximum latency imbalance (MLI) for all nodes in the mapped DFGs. This is the largest difference between any two inputs for a node, labelled MLI in Table 4.9. Since the MLI $\leq$ 64 for the benchmarks, the depth we have chosen for the DISO overlay is sufficient.

DISO-I achieves an F_{max} of 370 MHz on Zynq, hence providing a peak performance of 27.75 GOPS while DISO-II achieves an F_{max} of 300 MHz on Zynq, hence providing a peak performance of 44.10 GOPS. For each benchmark we compute the throughput as the product of the number of compute nodes in the DFG and the overlay's frequency, in GOPS, as shown in Table 4.9. We observe kernel throughputs of up to 9.62 GOPS (33% of the DISO-I peak throughput) and 21.6 GOPS (49% of the DISO-I peak throughput) using DISO overlays for benchmark set-I and set-II, respectively.

We also generate RTL implementations of the compute kernels for benchmark set-I and set-II using Vivado HLS in order to compare the performance in terms of throughput. We use the pipeline *pragma* with an II of one to generate fully parallel and pipelined RTL implementation of the compute kernels. Example code used for generating RTL using Vivado HLS for *chebyshev* kernel benchmark is shown in Table 4.10.

```
#include "kernel.h"

using namespace hls;

void kernel(stream<ap_int<16> > &stream_in, stream<ap_int<16> > &stream_out)
{
        ap_int<16> x;
        if (!stream_in.empty())
        {
                x = stream_in.read();
                #pragma HLS pipeline II=1
                stream_out.write(x*(x*x*(16*x*x-20)+5));
        }
}
```

Table 4.10: Code used for generating RTL using Vivado HLS for *chebyshev*

Table 4.11 shows the results for the Vivado HLS implementations of the benchmark set and Figure 4.17 shows the normalized throughput of the overlay implementations compared to the Vivado HLS implementations (which are normalized to one). The DISO overlay is able to achieve an average throughput improvement

of 50% due to the highly pipelined architecture, something which Vivado HLS is currently unable to exploit. However, this better throughput comes at the cost of significantly more FPGA resource consumption and greater latency. For the DISO overlay implementations, we observe a resource consumption (in eSlices) which is 7× higher (on an average) and a latency which is 4× higher (on an average), compared to the HLS implementations. The Latency increase is due to the deep pipelining of the overlay, but does not impact the performance since the II is always one which ensures processing of input data samples every clock cycle without stall.

Benchmark	OP	Freq. (MHz)	Slices	DSPs	eSlices	MOPS	MOPS/eSlice	Latency
chebyshev	7	333	24	3	204	2331	11.4	13
sgfilter	18	278	40	8	520	5004	9.62	11
mibench	13	295	81	3	261	3835	14.6	9
qspline	26	244	126	14	966	6344	6.56	21
poly1	9	285	62	4	302	2565	8.49	12
poly2	9	295	45	4	285	2655	9.31	11
poly3	11	250	52	6	412	2750	6.67	12
poly4	6	312	36	3	216	1872	8.66	7
atax	60	263	78	18	1158	15780	13.6	13
bicg	30	270	91	18	1171	8100	6.91	7
trmm	54	222	105	36	2265	11988	5.29	8
syrk	72	250	237	24	1677	18000	10.7	10

Table 4.11: Determining MOPS/eSlice for the Vivado-HLS implementations of the benchmark set

From Table 4.11, the average throughput per unit area for the HLS implementation of the benchmark set is ≈ 9.2 MOPS/eSlice while a 5×5 DISO overlay achieves 1.8 MOPS/eSlice, which is around 20% of the HLS value. This 5× hardware performance penalty needs to be considered in context with the ability of the overlay to support runtime compilation and runtime configuration of the compute kernels.

DISO-I (size=5×5) and DISO-II (size=7×7) have a configuration sizes of 287 Bytes and 700 Bytes, respectively, and can be configured entirely in 11.5 us and 28 us,

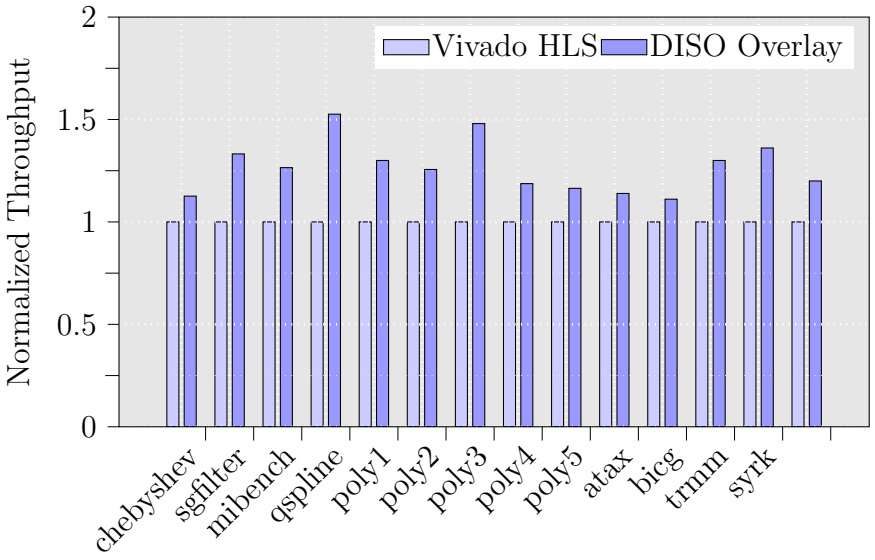


Figure 4.17: Normalized throughput of DISO overlay implementations over Vivado HLS implementations.

respectively, compared to 31.6 ms for the entire Zynq programmable fabric using the PCAP port, providing a 1000× improvement in reconfiguration time.

4.5.2 Dual-DISO

Even if we use an 8×8 DISO overlay (which is able to fit on a Zynq 7020 device) having a peak throughput of 65 GOPS, the achievable kernel throughput is 4.57 GOPS (on average) for the compute kernels of benchmark set-I, with a maximum throughput of 9.62 GOPS (for the *qspline* kernel). This gap is mainly due to the fact that different applications require different sized overlays, with an overlay large enough to satisfy the requirements of larger kernels being heavily underutilized when a small kernel is executing.

To avoid underutilization, other researchers have proposed using multiple instances of small overlays for smaller kernels and a large overlay for larger kernels, reconfiguring the FPGA fabric at runtime based on kernel requirements [138]. However, this approach negates a key advantage of overlay architectures, specifically rapid configuration to support fast switching of kernels. Reconfiguring the FPGA fabric takes orders of magnitude more time than a kernel context switch.

In this section, we take the approach of building a single large Dual-DISO overlay and mapping multiple instances of kernels to the overlay to achieve effective utilization of resources while maximizing throughput. We replicate multiple instances of the benchmarks from Table 4.4 and map them to an 8×8 CW2-4N-2D Dual-DISO overlay. The x-axis of Figure 4.18 indicates the benchmark number, followed by the number of replicated instances in brackets and shows that we are able to map multiple instances of kernels to the overlay. It should be noted that there are a number of benchmarks (benchmark 14-15, 17-19, 21-22, and 24) that are unable to map more than 1 instance due to the I/O limitations of the CW2-4N-2D overlay, as indicated in Figure 4.6.

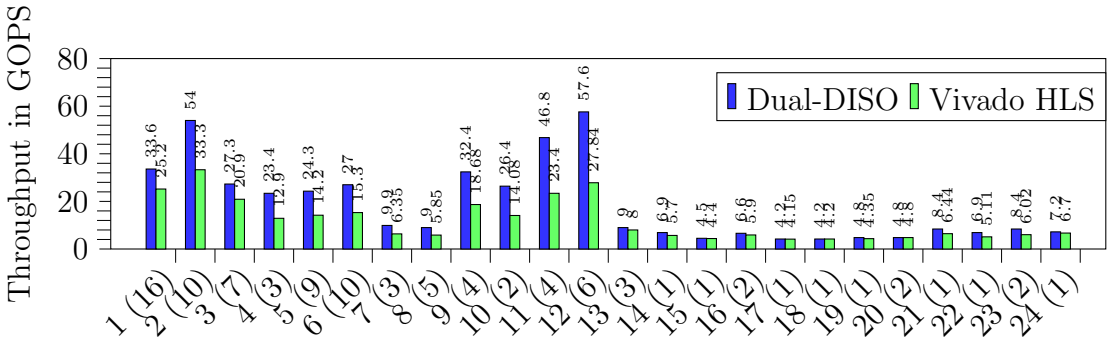


Figure 4.18: The performance comparisons of the CW2-4N-2D overlay and Vivado HLS implementations.

The actual throughput, in GOPS, for the replicated benchmark instances is shown by the left bar in Figure 4.18, calculated as the product of the DFG compute nodes and the implementation operating frequency. For example, an overlay throughput of 57.6 GOPS is achieved by instantiating 6 instances of the *poly8 (12)* benchmark. This is 50% of the absolute peak performance, of 115 GOPS, which could be hypothetically achieved by a synthetic kernel having 384 operations (128 Add/sub, 128 MUL, 128 ALU ops) which would fully utilise the DSP block resources of the 64 FUs in our 8 × 8 overlay. It is clear that the benchmarks with modest I/O requirements benefit from replication, while those with larger I/O requirements would benefit from the CW4-8N-2D overlay.

To compare the performance of our proposed overlay with application specific accelerator implementation, we generate RTL implementations of the same kernel

instances replicated an identical number of times using Vivado HLS. In this way, we are able to perform a quantitative comparison of performance between the two implementations. Figure 4.18 shows the performance comparison of the overlay implementations (left bar) and Vivado HLS implementations (right bar) in terms of throughput. Our overlay is able to achieve an average throughput improvement of 40% due to the highly pipelined architecture, something which Vivado HLS is currently unable to exploit. The Vivado HLS implementations of the replicated benchmarks require significantly less hardware resource (on average, our overlay requires 30× and 70× more slices, for benchmarks 1-12 and 13-24, respectively). However, this hardware penalty is the result of a general overlay architecture that can be effortlessly integrated into a virtualised hardware/software environment on the Zynq FPGA that may incorporate both static and PR accelerators as well as overlays for generality and performance. The key advantage of an overlay is the fast compilation, software-like programmability and run-time management, with a relatively small configuration data size and fast non-preemptive hardware context switching, all of which are missing in a static Vivado HLS accelerator design. As indicated in Section 4.4.2, our proposed overlay is able to perform a hardware context switch in just 45.5 us (1000× faster than reconfiguring the Zynq FPGA) using just 1137 Bytes of configuration data.

4.6 Summary

We have presented two FPGA targeted overlay architectures (DISO and Dual-DISO) that maximize the peak performance and reduce the interconnect area overhead through the use of an array of DSP block based fully pipelined FUs and an island-style coarse-grained routing network. A scalability analysis of DISO on the Xilinx Zynq device shows that the Zynq fabric can accommodate an 8×8 DISO overlay, achieving a peak performance of 65 GOPS (10× better than DSP-DySER) with an interconnect area overhead of 430 LUTs/GOPS (18× better than DSP-DySER). We then presented an analysis of a wide variety of compute kernels using a DSP48E1 aware data flow graph based approach to ascertain the suitability of

mapping multiple instances of kernels to the overlay and observed that the dual-DSP block based overlay is most suitable for our benchmark set.

We then presented a prototype of an enhanced version of DISO (referred to as Dual-DISO) which uses two DSP blocks within each FU and shows a significant improvement in performance and scalability, with a reduction of almost 70% in the overlay tile requirement compared to existing overlay architectures and an operating frequency in excess of 300 MHz. A scalability analysis of Dual-DISO on the Xilinx Zynq device shows that Zynq fabric can accommodate an 8×8 Dual-DISO overlay, achieving a peak performance of 115 GOPS (18× better than DSP-DySER) with an interconnect area overhead of 320 LUTs/GOPS (24× better than DSP-DySER). We demonstrate that this improvement results in better exploitation of the performance provided by the DSP blocks available on the FPGA fabric.

We then map several benchmarks kernels onto the proposed overlays and show that the proposed overlays can deliver better throughput compared to Vivado HLS generated fully pipelined RTL implementations. Our experimental evaluation shows that the DISO overlay delivers kernel throughputs of up to 21.6 GOPS (33% of the peak theoretical throughput of the DISO overlay) while the Dual-DISO overlay delivers kernel throughputs of up to 57.6 GOPS (50% of the peak theoretical throughput of the DISO overlay) and provides an average throughput improvement of 40% over Vivado HLS for the same implementations of our benchmark set. Using the Dual-DISO overlay, we show that we are able to map multiple instances of the benchmark kernels to the overlay automatically, resulting in more efficient utilization of overlay resources, without resorting to reconfiguring the FPGA fabric at runtime. We have demonstrated that architecture-focused FPGA overlays can better exploit the raw performance of the DSP blocks, with better resource utilization and significantly improved performance metrics, compared to other overlays.

However, when compared to HLS implementations, there is still a significant hardware resource penalty, even though a better throughput is achieved. This hardware resource requirement needs to be further reduced for overlays to be accepted as

a viable alternative to the standard HLS flow. In the next chapter, we present an overlay architecture with a novel interconnect architecture that maximizes the peak performance and reduces the interconnect area overhead through the use of a cone-shaped array of DSP blocks.

5

Low Overhead Interconnect for DSP Block Based Overlays

5.1 Introduction

DSP block based overlays (DISO and Dual-DISO) can greatly improve performance in terms of throughput by using the full capability of highly pipelined DSP blocks along with a flexible island-style interconnect architecture as shown in the Chapter 4. However, supporting the full flexibility of island-style interconnect contributes to a significant area overhead. Not only DISO and Dual-DISO, but many of the existing overlay architectures [136, 74] use general-purpose interconnect which allows all FUs to communicate with each other by exploiting the high flexibility of the interconnect architecture. This level of interconnect flexibility

is, in many cases, an over-provision and normally not required for implementing accelerators based on feed-forward pipelined datapaths.

A number of authors have instead proposed a linear array of interconnected FUs [157] [156] to improve resource utilization. The VDR overlay [157] was proposed as an array of tiles interconnected linearly by a set of programmable switches. However, the F_{max} was only 172 MHz when implemented on an Altera Stratix III FPGA. A domain specific reconfigurable array, with a linear feed forward interconnect structure and with array dimensions determined by merging the datapaths of all DFGs, was proposed in [174]. However this approach resulted in heavy underutilization of FUs, with not more than 40% utilization when mapping DFGs. Instead, it was observed that better utilization could be achieved if the FU architecture was customized to better match the shape of the DFGs [174], which in many cases have the general shape of an inverted cone.

Thus, to further reduce the overlay interconnect complexity, it appears feasible to customize the typical rectangular array of FUs to produce a cone shaped cluster of FUs utilizing a simple linear interconnect. As a first step, we perform an analysis of linear interconnect overlays from the perspective of both the DFG structure and interconnect flexibility, which we term the *programmability overhead*. This would allow a reduction in the area overheads when implementing compute kernels extracted from compute-intensive applications represented as directed acyclic data flow graphs by better targeting the set of FUs, while still allowing high data throughput. After determining the interconnect flexibility required for a set benchmarks, we design a cone shaped overlay with low overhead interconnect and observe significant savings in the LUT requirements compared to other overlays from the literature. The main contributions of this chapter can be summarized as:

- A design space exploration of linear overlay architectures by modeling programmability overhead as a function of the overlay design parameters
- An area efficient RTL implementation of a **D**SP-**e**nabled **C**one-shaped **O**verlay architecture, referred to as DeCO, which can operate at near to the DSP block theoretical maximum frequency

- An analysis of DeCO on the Xilinx Zynq device, including the evaluation of peak throughput (in terms of GOPS) and interconnect area overhead (in terms of LUTs/GOPS) of DeCO.
- A technique for mapping compute kernels to the proposed overlay by performing graph optimizations such as rebalancing, DSP48E1 aware node merging and architecture aware rescheduling.
- A quantitative performance comparison of the proposed DeCO architecture with other overlays from the literature
- A quantitative evaluation of the hardware performance penalty of DeCO compared to HLS generated hardware implementations.

The work presented in this chapter is also discussed in

- 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 IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM), Washington DC, USA, May 2016.

5.2 Interconnect Architecture Analysis

A linear interconnect architecture, consisting of a one-dimensional array of programmable homogeneous tiles, where each tile contains a programmable routing network and a cluster of DSP block based FUs and data forwarding (DF) links, is shown in Figure 5.1. Data forwarding is required in each cluster for forwarding input data to the next tile, thereby bypassing the current tile, and thus they require a latency equivalent to that of the DSP block based FUs. This significantly reduces the need to waste FUs for routing signals through the overlay, which is one of the main drawback with nearest-neighbor overlays. A linear interconnect based overlay has two main advantages. Firstly, the resource requirement is significantly reduced from that of an island-style or nearest-neighbor architecture as the routing network between FU stages is only required in one direction. Secondly,

the FU array complexity is reduced as delay balancing or reordering logic is not needed with a simple, strictly balanced pipelined structure with no feedback.

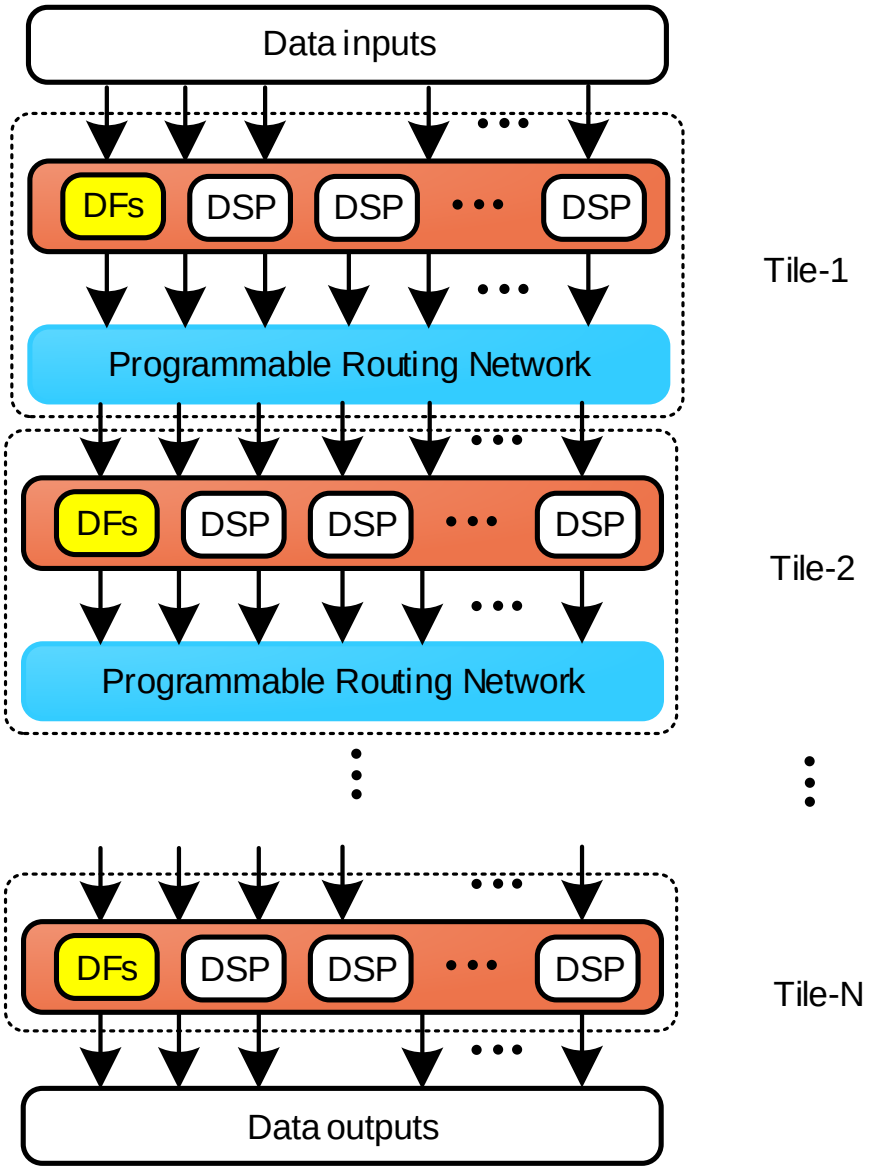


Figure 5.1: Block diagram of linear dataflow overlay.

As mentioned previously, a DSP-based overlay architecture should ideally have a programmability overhead less than the ratio of the LUTs/DSP in the target FPGA, so as to get the most out of the available FPGA resources, particularly the DSP blocks. For example, in the 16-bit overlay of [78], reordering logic, synchronization logic and routing network logic requires 6, 8 and 10 LUTs/bit per tile, respectively, to support interconnectivity between the DSP block based FUs,

resulting in a programmability overhead of 384 LUTs/FU. We term programmability overhead as the number of LUTs required per DSP block based FU to allow programmable connections between DSP blocks. However, this programmability overhead is significantly more than that of target FPGA (in this case a Xilinx Zynq XC7Z020), where there are 220 DSP blocks for 53K LUTs, resulting in a LUT/DSP ratio of 240. To achieve a programmability overhead approaching this value, we propose further customizing the number of FUs, data forwarding links and the complexity of the routing network in each tile based on the characteristics of the set of compute kernels. This is similar to datapath merging [174], except that we only merge computation blocks of sequenced DFGs in a stage-wise manner, leaving a fully flexible interconnect between stages, hence still retaining significant flexibility. This then allows us to handle unknown DFGs, so long as the DFG can be scheduled on the merged datapath.

5.2.1 Programmability Overhead Modeling

For each tile, the number of DSP blocks, delay lines and the routing network complexity can be decided based on a set of compute kernels. The complexity of the routing network in the n^{th} tile depends on the resources in the n^{th} tile and $(n+1)^{th}$ tile, that is the number of DSP blocks and delay lines. The routing network can then be designed using an $(X_n + Y_n) \times (I * X_{n+1} + Y_{n+1})$ crossbar switch, where I is the number of FU inputs, which because of the characteristics of the DSP block is 3 for a 32-bit FU and 4 for a 16-bit FU, and where X_n and Y_n are the number of DSP blocks and delay lines in n^{th} tile and X_{n+1} and Y_{n+1} are the number of DSP blocks and delay lines in $(n+1)^{th}$ tile. The reason for the different I between the two FUs is that the 16-bit FU is able to utilize the pre-adder in the DSP block, whereas the 32-bit version is not.

If L_n is the number of LUTs/bit required to design an $(X_n + Y_n)$:1 multiplexer, the programmability overhead per bit (POB) of the overlay network, in LUTs/bit, is:

$$POB = \sum_{n=1}^{N-1}(L_n * (I * X_{n+1} + Y_{n+1})) \tag{5.1}$$

where N is the number of tiles in the overlay.

Given a set G of M sequenced DFGs, we refer to the number of operation nodes in the n^{th} stage as $g_m x_n$, the total number of stages as Ng_m, and the crossing edges as $g_m y_n$ in the m^{th} DFG, G_m. We can then find X_n, Y_n and N using:

$$X_n = \max(g_1 x_n, g_2 x_n, ...g_M x_n) \tag{5.2}$$

$$Y_n = \max(g_1 x_n + g_1 y_n, g_2 x_n + g_2 y_n, ...g_M x_n + g_M y_n) - \max(g_1 x_n, g_2 x_n, ...g_M x_n) \tag{5.3}$$

$$N = \max(Ng_1, Ng_2, Ng_3, ...Ng_M) \tag{5.4}$$

5.2.2 Set-specific Overlay Design

We calculate the overlay design parameters and programmability overhead using a subset of DFGs from [75], given in Table 5.1. The graph depth is the critical path length of the graph, while the graph width is the maximum number of nodes that can execute concurrently, both of which impact the ability to efficiently map a kernel to the overlay. The average parallelism is the ratio of the total number of operations and the graph depth. We note that many of these DFGs are poorly balanced, so we firstly apply tree-balancing to all DFGs, which both reduces the graph depth and better shapes the DFG. Next we apply DSP aware node merging [78], which better targets the DSP block based FU, and lastly we apply rescheduling techniques, possibly increasing the graph depth, which results in different FU requirements at each stage. This results in a number of different DFGs for the same benchmark, with different values for the overlay design parameters (N, X_n, and Y_n), resulting in different programmability overhead values.

For example, the original *kmeans* benchmark [75], shown in Figure 5.2(a), has a depth of 9 requiring 9 stages in a linear overlay. The rebalanced graph is shown

No.	Benchmark Name	Characteristics (After transformation characteristics)				
		I/O nodes	graph edges	op nodes	graph depth	graph width
1.	fft	6/4	24(22)	10(8)	3(3)	4
2.	kmeans	16/1	39(35)	23(19)	9(5)	8
3.	mm	16/1	31(31)	15(15)	8(4)	8
4.	spmv	16/2	30(30)	14(14)	4(3)	8
5.	mri	11/2	24(21)	11(9)	6(5)	4
6.	stencil	15/2	30(23)	14(8)	5(3)	6

Table 5.1: DFG characteristics of benchmark set

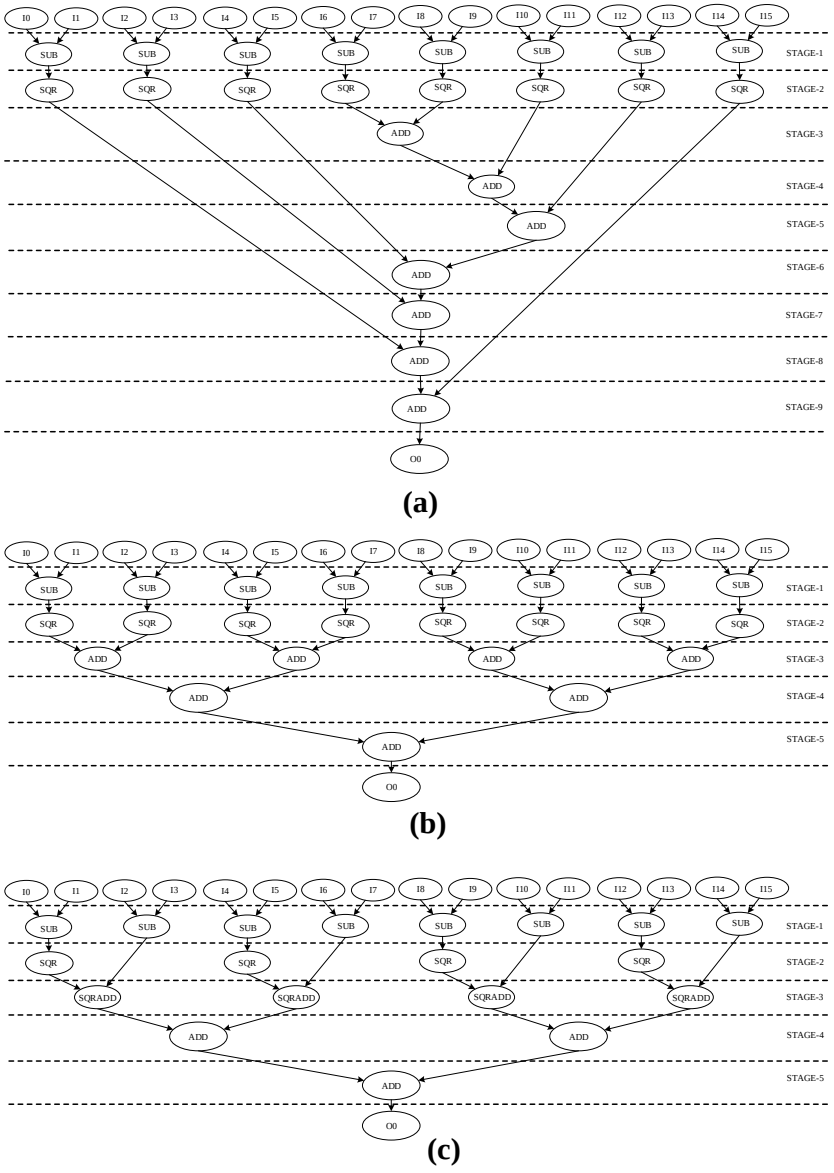


Figure 5.2: Applied transformations including graph balancing and DSP aware node merging.

in Figure 5.2(b), and requires just 5 stages, with a significantly reduced latency. One example of node merging and rescheduling using ASAP scheduling is shown in Figure 5.2(c) resulting in a reduced FU count. It should be noted that we observe a higher programmability overhead for the ALAP scheduled version, due to the additional data forwarding delay line needed to forward input data to the 2nd stage. The 3 implementations shown in Figure 5.2 result in a programmability overhead of 6.25, 4.75 and 4.3 LUTs/bit per FU, respectively. Thus, we would choose DFG 3, the one with the smallest programmability overhead, as the candidate DFG for the design process.

We repeat this process for the remaining DFGs in the benchmark set of Table 5.1, to determine the best FU and data forwarding link configuration to support all compute kernels in the benchmark set. This results in a structure with different resource requirements in each scheduling stage, which we refer to as a cone [175, 176]. This cone consists of 20 DSP block based FUs and four layers of connection network, as shown in Figure 5.3.

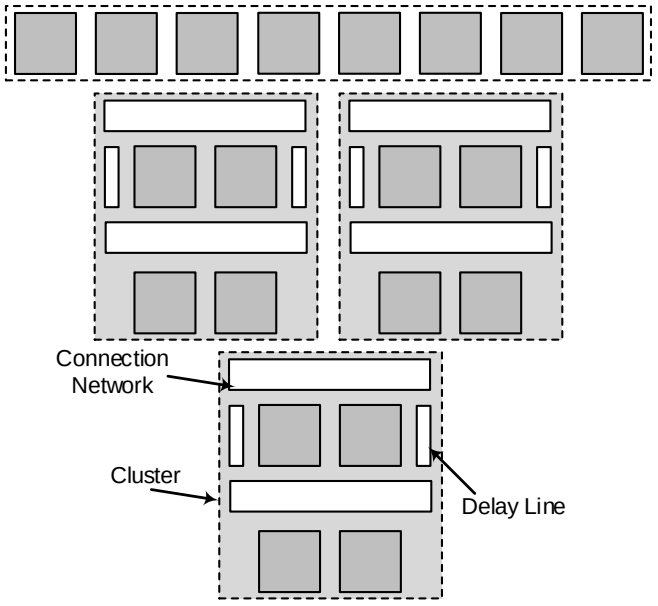


Figure 5.3: Design of the optimized cone.

Figure 5.4 shows the mapping of the largest benchmark (*"kmeans"*) on to the DeCO and onto the DISO[78]. When *"kmeans"* is mapped on DISO it has an

initial latency of 52 cycles while on DeCO the initial latency is just 24 cycles, more than half that of the DISO implementation.

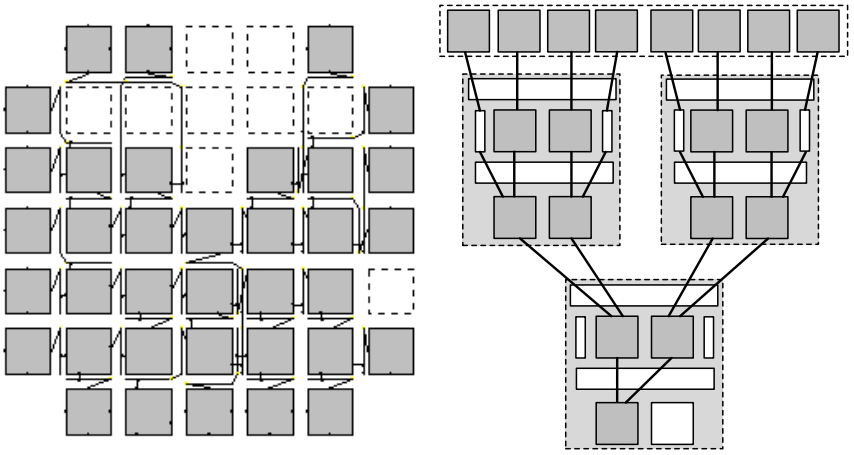


Figure 5.4: Mapping of *kmeans* on DISO vs DeCO.

5.3 The DeCO Architecture

We use the Xilinx DSP48E1 primitive as a programmable FU in the proposed DeCO architecture. However, unlike in other overlays from the literature [177, 78, 137] the interconnect network in the proposed overlay is very lightweight, enabling us to efficiently implement both 16-bit and 32-bit implementations. The overlay micro-architecture of the two implementations is described, which because of the characteristics of the DSP block, are slightly different. We then detail the physical mapping to the FPGA fabric and the resource usage. To achieve this we use Xilinx ISE 14.6 and a Verilog HDL description of the overlay targeting the Xilinx Zynq XC7Z020.

5.3.1 The 32-bit Architecture

The Xilinx DSP48E1 primitive does not provide a full 32-bit implementation, even though some internal signals are much larger than 32-bits. The problem arises, because the pre-adder is just 25-bit, while the multiplier is 25-bit $\times$ 18-bit. While this could be a problem for *long* integer multiplication, it is less likely if all

variables are restricted to the C language 32-bit *int* data type. This is because the default conversion rules for the *int* data type will truncate the result of an integer multiplication, discarding the most significant part. To avoid possible overflow conditions with the 25-bit pre-adder, we choose not to use it, and instead bypass the pre-adder. The resulting micro-architecture, showing the 32 bit connection network and the 32-bit FU, which is able to support the C language 32-bit *int* data type, is shown in Figure 5.5. In this case, the FU uses just the multiplier, the ALU, the three separate A, B and C ports for input data, and one output port P, as shown in Figure 5.5, and can be configured to support various operations such as multiply-add, multiply-subtract, etc. The actual DSP48E1 function is determined by a set of control inputs that are wired to the $ALUMODE$, $INMODE$ and $OPMODE$ configuration registers. The DSP48E1 primitive is directly instantiated providing total control of the configuration of the primitive, allowing us to achieve a high frequency.

The top most routing layer of the 20 FU overlay, shown in Figure 5.3, requires connections to four FUs (with three inputs each) and four data forwarding links (with a single input). Thus, for full routing flexibility, the top most routing network layer requires 16 8:1 multiplexers. However, during the design process we observed that there is no requirement for connectability between the left and right regions in the top part of the cone, and thus they do not need to be fully connected. Hence in the top connection layer, we use two separate smaller connection networks, each having 8 4:1 multiplexers. Similarly, in the second layer, we again isolate the left and right regions, allowing the use of 4:1 multiplexers. In the third connection layer, we combine the signals from the left and right regions, which now also only requires 4:1 multiplexers. Thus, the resulting 32-bit cone, shown in Figure 5.3, consists of a top layer of 8 FUs followed by three identical clusters of 4 FUs and 2 data forwarding links, and has a total latency of 24 clock cycles.

Each cluster consists of four FUs preceded by three individual 4:1 muxes at the input of each FU, and two data forwarding links preceded by a 4:1 mux, as shown for the 16-bit version in Figure 5.7. The 32-bit FU requires one DSP block and four additional registers at the DSP input ports (an 18-bit register for input B, a 25-bit

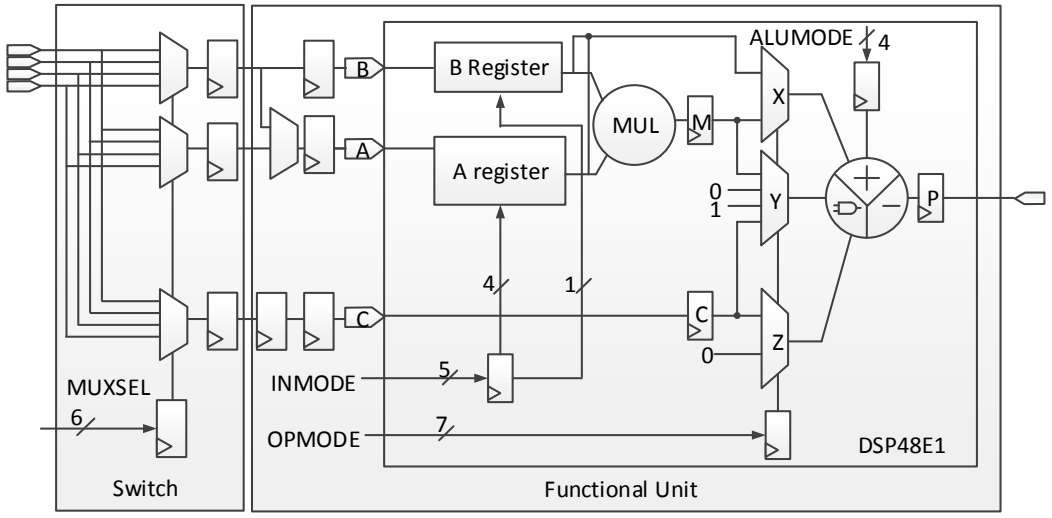


Figure 5.5: The 32-bit functional unit and interconnect switch.

register for input A, and two 32-bit registers for input C) for pipeline balancing (as shown in Figure 5.5), consuming 107 FFs. The 25-bit 2:1 multiplexer in front of the A input port consumes 13 LUTs and allows to choose between the 25 bits (for a multiplication operation) or the 14 most significant bits of the data going to the B input port (for an addition operation). Each FU connection network requires three 4:1 multiplexers with registered outputs, consuming 96 LUTs and 89 FFs. The delay line requires 32 LUTs (SRLs) and 32 FFs and the delay line input selection logic requires 32 LUTs and 32 FFs. This results in a total cluster resource consumption of 564 LUTs, 912 FFs and 4 DSP blocks. The cluster configuration register includes 16 bits for each DSP block configuration, 2 bits for each mux and 2 bits for delay line input selection logic. Hence, the cluster configuration register consumes 92 FFs. The entire routing network requires 84 bits for the 32-bit cone and hence the entire cone can be reconfigured using just 404 bits (50.5 Bytes) of configuration data.

The post-place and route resource consumption of the 32-bit DeCO is 2076 LUTs, 3984 FFs and 20 DSP blocks, with a programmability overhead of 103.8 LUTs/FU (3.24 LUTs/bit per FU), significantly less than original target of $\leq$ 240 LUTs/FU.

The 32-bit DeCO overlay achieves a frequency of 395 MHz, which is close to the DSP theoretical limit of 400 MHz on the Zynq device.

5.3.2 The 16-bit Architecture

The 16-bit FU is similar to the 32-bit FU, except that it can now make use of the DSP block A input and the pre-adder, allowing additional instructions such as add-multiply and subtract-multiply. This results in the four input, one output FU, shown in Figure 5.6. As with the 32-bit version, the DSP48E1 function is determined by a set of control inputs that are wired to the $ALUMODE$, $INMODE$ and $OPMODE$ configuration registers.

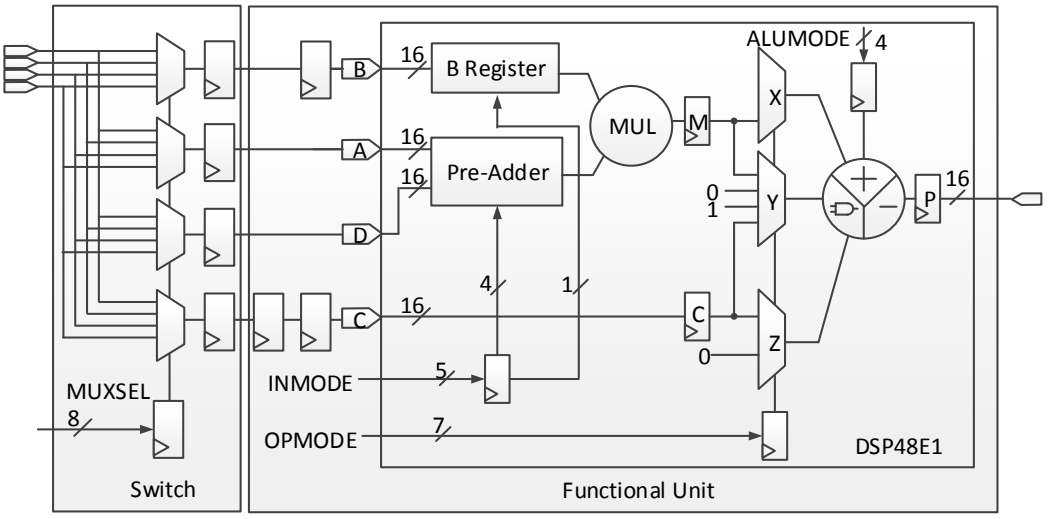


Figure 5.6: The 16-bit functional unit and interconnect switch.

Because of the extra DSP block input (used by the pre-adder), the top most routing layer requires connections to four FUs (with four inputs each) and four data forwarding delay lines (each with a single input). Again, because we do not need full routing flexibility, the top most connection layer requires two sets of 10 4:1 multiplexers. Similar to the 32-bit version, we use 4:1 multiplexers in the other connection layers. The 16-bit DeCO overlay also has a total latency of 24 clock cycles.

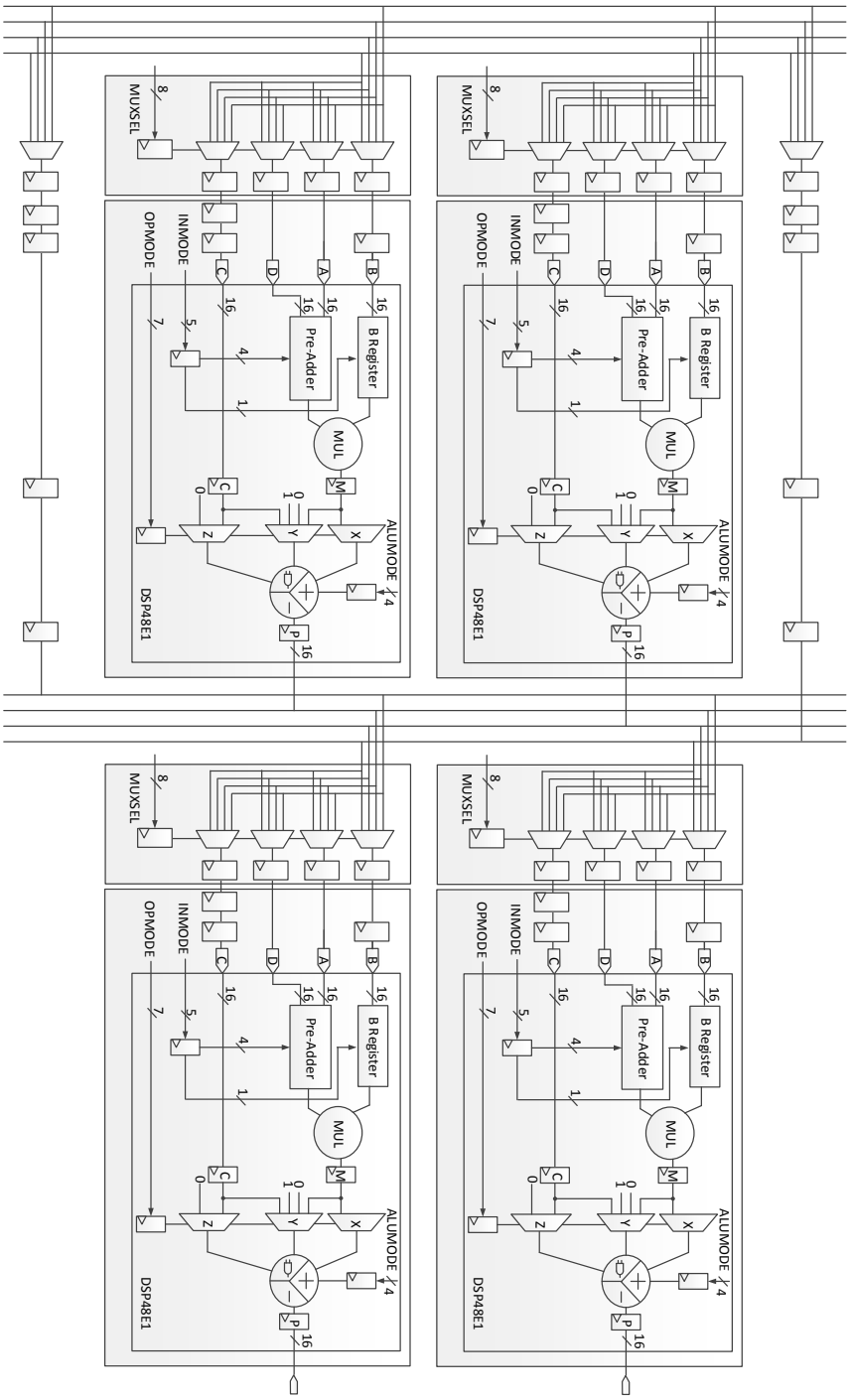


Figure 5.7: Micro-architectural design of the 16-bit cluster consisting of four functional units and two delay lines.

Resource Usage for the 16-bit Architecture: In the 16-bit version of the overlay, which also consists of three tiles with an additional eight FUs at the top layer, each tile consists of four FUs preceded by four individual 4:1 muxes at the input of each FU, and two data forwarding delay lines preceded by a 4:1

mux, as shown in Figure 5.7. The 16-bit FU requires one DSP block and three additional 16-bit registers at the DSP input ports for pipeline balancing (as shown in Figure 5.6), consuming 48 FFs. Each FU connection network requires four 4:1 multiplexers with registered outputs, consuming 64 LUTs and 64 FFs, while the delay line requires 16 LUTs (SRLs) and 16 FFs and the delay line input selection logic requires 16 LUTs and 16 FFs. This results in a total 16-bit cluster resource consumption of 320 LUTs, 512 FFs and 4 DSP blocks. The cluster configuration register includes 16 bits for each DSP block configuration, 2 bits for each mux and 2 bits for delay line input selection logic, resulting in a cluster configuration register size of 100 FFs. For the DSP block based FU, programming the FU settings requires 16 configuration bits while programming the routing network requires 2 configuration bits per 4:1 multiplexer. Thus, the entire routing network requires 108 bits for the 16-bit cone and hence the entire cone can be reconfigured using just 428 bits (53.5 Bytes) of configuration data.

The post-place and route resource consumption of the 16-bit DeCO is 1368 LUTs, 2348 FFs and 20 DSP blocks, with a programmability overhead of 68.4 LUTs/FU (2.14 LUTs/bit per FU), significantly less than original target of $\leq$ 240 LUTs/FU. The 16-bit DeCO achieves a frequency of 395 MHz, which is again very close to the DSP theoretical limit of 400 MHz on the Zynq device. The 16-bit DeCO was subsequently mapped to a Xilinx Virtex-7 XC7VX690T-2 and achieved a frequency of 645MHz. Compared to 32-bit version of the DeCO architecture, 16-bit version requires 35% less LUTs and 40% less FFs.

5.4 Experimental Evaluation

In this section, we first present a quantitative comparison of the DeCO architecture with some existing overlays from the research literature and then evaluate the performance of the DeCO architecture, using a benchmark set of compute kernels, which we then compare to other overlay implementations. Figure 5.8 shows the mapping of the 16-bit DeCO architecture onto the Zynq FPGA (the red area at

the bottom left of the fabric). The Zynq fabric is thus able to accommodate a number of multiple parallel instances of the DeCO architecture which allows the exploitation of all of the DSP blocks available on the fabric. In this section, we will only consider the 16-bit version of the DeCO architecture so that we can compare to the other overlays which are all 16-bit.

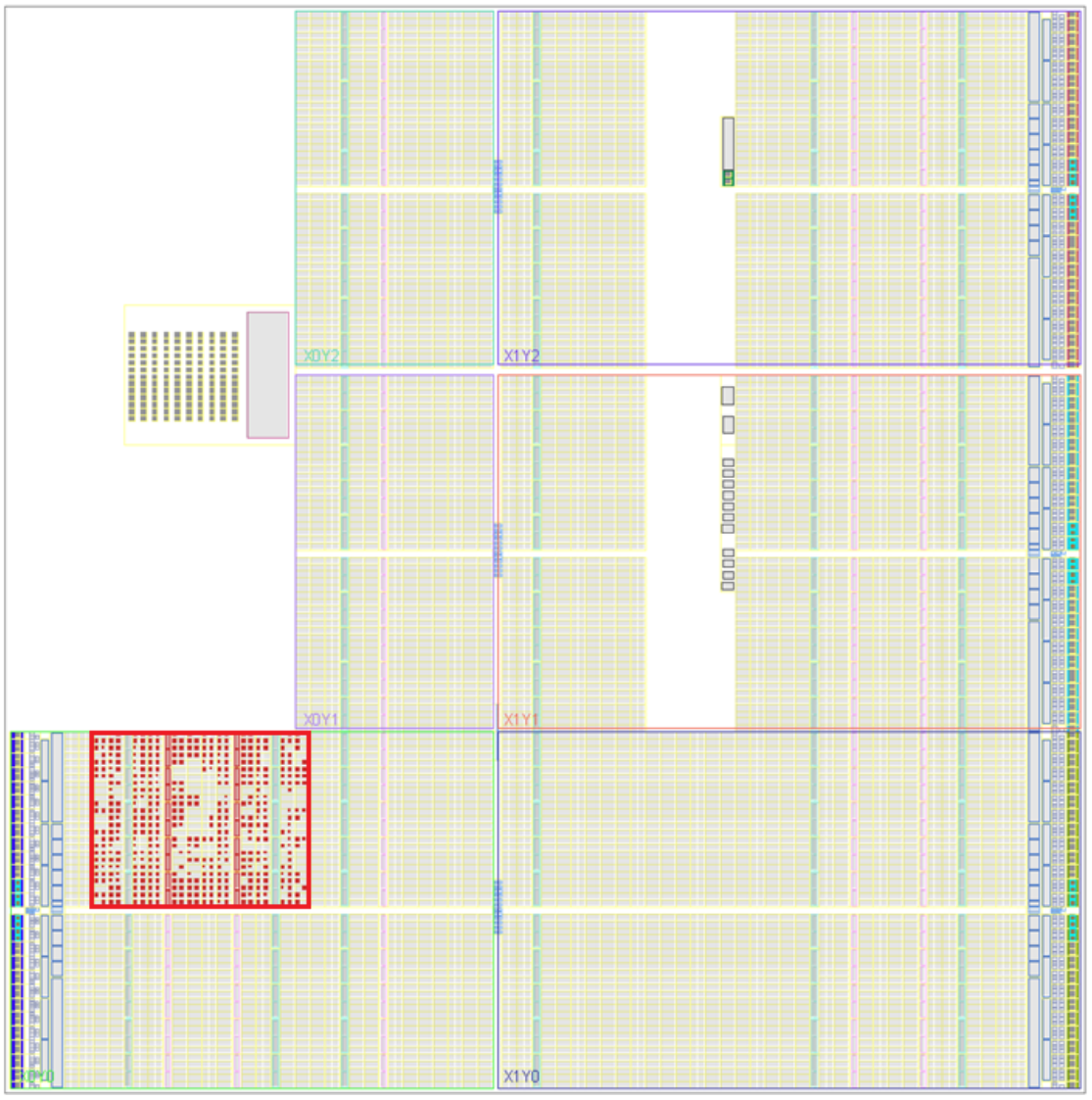


Figure 5.8: Mapping of DeCO on Zynq.

To show that the interconnect area overhead (in terms of LUTs/GOPS) has been reduced significantly, we compare DeCO with a number of different overlays, both from the literature and previous chapters as shown in Table 5.2. For the different overlays we compare the resource usage in terms of the LUTs used and the percentage of the total LUTs used in the target device (shown in brackets), the frequency,

Max OPs and the peak throughput of the arithmetic operations in GOPS. As the overlays are all different sizes, with peak throughputs (in GOPS) which are dependent on the number of FUs, it is important to normalize the characteristics of the different overlays, which we do by using the interconnect area overhead in terms of LUTs/GOPS. Table 5.2 also shows the LUTs/GOPS, for the various overlays, and clearly shows that the DeCO architecture has the lowest interconnect area overhead. The interesting point to note is that the smaller DeCO architecture achieves a peak performance of 23.7 GOPS (3.8× better than DSP-DySER) but with an interconnect area overhead of 58 LUTs/GOPS (130× better than DSP-DySER), consuming 45× fewer LUTs compared to DSP-DySER. This is mainly due to the considered use of the DSP blocks and the low overhead interconnect used in the DeCO architecture.

Resource	IF [137]	IF (opt) [137]	DSP-DySER	DISO	Dual-DISO	DeCO
Device	XC5VLX330	XC5VLX330	XC7Z020	XC7Z020	XC7Z020	XC7Z020
Slices|LUTs	51.8K|207K	51.8K|207K	13.3K|53K	13.3K|53K	13.3K|53K	13.3K|53K
Overlay FUs	14×14	14×14	6×6	8×8	8×8	20
LUTs used	91K(44%)	50K(24%)	48K(90%)	28K(52%)	37K(70%)	1368(2.5%)
Fmax (MHz)	131	148	175	338	300	395
Max OPs	196	196	36	192	384	60
GOPS	25.6	29	6.3	65	115	23.7
LUTs/GOPS	3550	1725	7620	430	320	58

Table 5.2: Quantitative comparison of DeCO with other overlays

We observe that for the DSP-DySER overlay, it is possible to fit an array of 36 DSP blocks (16% of the total DSP blocks on Zynq), while for the DISO and Dual-DISO architectures it is possible to fit 64 (30%) and 128 (60%) DSP blocks, respectively. For the DeCO architecture, it is possible to fit 11 parallel instances of DeCO resulting in the use of all of the 220 DSP blocks with a resource consumption of 15K LUTs only. It will result in a large overlay (can support up-to 660 arithmetic operations) operating at a frequency of 395 MHz and providing a peak throughput of 260 GOPS with an interconnect area overhead of 58 LUTs/GOPS. In terms of the Peak GOPS, DSP-DySER achieves 6.3 GOPS, DISO and Dual-DISO overlays achieve 65 GOPS and 115 GOPS, respectively, while 11 parallel instances of

DeCO achieve 260 GOPS, representing 2.4%, 25%, 44% and 98% of the maximum achievable GOPS, respectively.

5.4.1 Overlay Comparison and Analysis for Benchmark Set

For comparison purposes, we map the benchmark set onto the proposed 16-bit DeCO architecture, and onto on a 5×5 DSP-DySER [177] and a 5×5 DISO [78], both of which are 16-bit architectures. We do not consider IF [137] for the comparison, since it only uses fixed-logic multipliers (mapped to the DSP blocks by the synthesis tool) as functional units (FUs), and as such, it is not possible for IF to support the kernels in the benchmark set. We also do not compare with Dual-DISO as in this experiment we are only mapping single instances of the benchmark set.

While all overlays have comparable DSP usage (25, 25 and 20, respectively), the LUT and FF requirements at (33875, 11023 and 1368) and (13390, 10486 and 2348), respectively, show a significant reduction for the proposed overlay, as shown in Figure 5.9. This represents a savings in the LUT requirements of 96% and 87%, compared to DSP-DySER[177] and DISO[78], respectively.

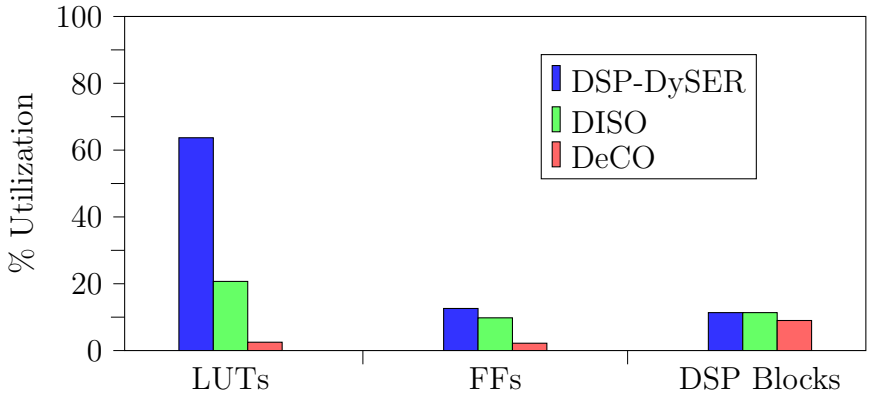


Figure 5.9: Comparison of overlay resources required for the benchmark set.

As a second experiment, we compare the physical footprint in terms of configuration tiles, operating frequency and configuration time, of DeCO with the two

previous overlays and the minimum partial reconfiguration (PR) region which can contain the hardware resources of the RTL implementation (generated using Vivado HLS 2013.2) of the benchmark set, as shown in Table 5.3. The Zynq fabric consists of 22 DSP tiles (each containing 10 DSP blocks) and 133 CLB tiles (each containing 50 CLBs), and reconfiguration using PR must be done in a multiple of these tiles to be fast. DeCO requires 2 DSP tiles and 6 CLB tiles. DSP-DySER[177] requires 3 DSP tiles and 126 CLB tiles while DISO[78] requires 3 DSP tiles and 42 CLB tiles. We also include a PR region which is big enough to accommodate the largest benchmark (*"kmeans"*) in the set, which requires 1 DSP tile and 3 CLB tiles, half that of DeCO. In terms of area, all overlays have a higher resource utilization, as is expected, with DeCO, DSP-DySER[177] and DISO[78] requiring 2×, 21× and 7× the FPGA resources, compared to a PR-based direct FPGA implementation of the compute kernels, respectively.

	Area (DSP/CLB)	Tiles (DSP/CLB)	Freq. (MHz)	Config. data (Bytes)	Config. time (μs)	Peak GOPS
DSP-DySER	25/6142	3/126	175	194.0	7.2	4.37
DISO	25/2095	3/42	370	287.0	11.5	27.75
DeCO	20/258	2/6	395	53.5	2.0	23.70
PR region	10/150	1/3	249	49000.0	382.0	-

Table 5.3: Experimental results for the comparison of different implementations

Perhaps more notable is the time required to change the kernel context for the various implementations. This is important if we need to swap between different accelerator implementations as an application executes. The PR region can be reconfigured entirely using 49 KBytes of configuration data in 382 μs using PCAP, while DeCO can be reconfigured entirely using just 53.5 Bytes of configuration data in 2 μs, which represents a 190× improvement in configuration time. The other two overlays, while requiring a slightly longer configuration time than the proposed overlay, are also significantly better than for PR.

We also calculate the peak throughput of the overlays, in GOPS, as the product of the overlay frequency and the maximum number of arithmetic operations, as

show in Table 5.3. However, because of the different resources used by the HLS implementations, compared to that of the overlays, we cannot use the compute area overhead (LUTs/GOPS) metric. Instead, we use the throughput per unit area (in MOPS/eSlice), where the hardware resource utilization is normalised in terms of equivalent slices (e-Slices), assuming that 1 DSP block is equivalent to 60 slices based on the ratio of slices/DSP on the Zynq XC7Z02-1CLG484C (which is approximate 60), as discussed in Chapter 3.7. The average throughput per unit area for the HLS implementation of the benchmark set is $\approx$ 7.5 MOPS/eSlice. In comparison, the largest kernel (*kmeans*) in the benchmarks set, when mapped onto the DeCO architecture, achieves 5.3 MOPS/eSlice, which is around 70% of the HLS implementations. This 1.5$\times$ hardware performance penalty needs to be considered in context with the ability of the DeCO architecture to support runtime compilation and runtime configuration of the compute kernels.

5.4.2 Mapping Additional Compute Kernels on to the DeCO

To analyze the mapping flexibility and utilization of the proposed overlay, we map additional kernels, taken from [78] and [62], remembering that the DeCO architecture was originally designed for the kernels shown previously in Table 5.1. This demonstrates that the DeCO architecture is more general than the initial design set. Table 5.4 shows the required number of cones, the percentage utilization of the FUs in the DeCO architecture, and the achievable GOPS for each benchmark. It can be seen that the proposed DeCO architecture is able to efficiently map kernels which are unknown at overlay design time.

Table 5.4 shows FU utilizations of up to 95%. For small kernels, with a utilization of less than 50%, such as *gradient* and *Chebyshev*, we are able to improve utilization by mapping replicated versions of the kernels. For example, the FU utilization for *gradient* [62] on a single full cone is 45%, but by mapping two instances of this kernel onto the proposed cone, we can achieve an effective FU utilization of 90%. For some of the compute kernels, such as *FFT* and *Chebyshev*, the FU utilization

Benchmark	Required No. of Cones	% Utilization	GOPS
fft	1	40%	3.95
kmeans	1	95%	9.08
mm	1	75%	5.92
spmv	1	70%	5.53
mri	1	75%	4.34
stencil	1	80%	5.53
gradient [62]	0.5	90%	4.34
chebyshev	0.5	40%	2.76
sgfilter	1	50%	7.11
mibench	1	40%	5.13
bicg	3	50%	11.85
trmm	4.5	60%	21.33
syrk	4.5	80%	28.44

Table 5.4: Experimental results for mapping benchmarks

is less because these kernels are not cone shaped DFGs. Here, *FFT* is wide while *Chebyshev* is narrow. Using a single DeCO, we are able to achieve a throughput of up to 9.08 GOPS (that is 38% of the peak throughput).

To map larger kernels, multiple instances of the overlay cone are used. The last three rows of Table 5.4 show the mapping of larger benchmarks to replicated instances of the cone, allowing us to achieve high GOPS values. Cones are replicated in a multi-lane pattern to reflect the shape of the larger graph.

5.5 Summary

We have presented DeCO, a DSP-enabled cone-shaped overlay architecture that uses Xilinx DSP48E1 primitives as a programmable FU, resulting in an area-efficient overlay (due to the low overhead interconnect) for pipelined execution of compute kernels, with improved performance metrics. After determining the

interconnect flexibility required for a set benchmarks, we design and implement a 16-bit version and a 32-bit version of the DeCO architecture. The post-place and route resource consumption of the 32-bit DeCO overlay is 2076 LUTs, 3984 FFs and 20 DSP blocks, while the resource consumption of the 16-bit DeCO overlay is 1368 LUTs, 2348 FFs and 20 DSP blocks. Compared to the 32-bit version of DeCO, the 16-bit version requires 35% less LUTs and 40% less FFs.

The 16-bit DeCO, when implemented on a Xilinx Zynq, achieves savings in the LUT requirements of 96% and 87%, compared to 16-bit DSP-DySER[177] and 16-bit DISO[78], respectively. 16-bit DeCO achieves a frequency of 395 MHz and provides a peak performance of 23.7 GOPS (3.8× better than DSP-DySER) with an interconnect area overhead of 58 LUTs/GOPS (130× better than DSP-DySER), consuming 45× fewer LUTs compared to DSP-DySER, but with a 1.5× hardware performance penalty compared to the HLS generated hardware implementations. We have demonstrated that the use of an architecture-focused FU and low overhead interconnect can result in an efficient overlay architecture with significantly lower area and performance overheads compared to other overlays.

Although the proposed overlay architectures focus on Xilinx FPGA devices, the approach used to design these architectures is highly flexible when implementing on other platforms. For example, the proposed overlays can be implemented on an Altera FPGA device just by changing the DSP primitive instantiation. The mapping tool would need to incorporate FU templates based on the DSP block capability of the Altera FPGA device. The overlay interconnect is essentially vendor-independent and portable across all FPGA devices.

6

Mapping Tool for Compiling Kernels onto Overlays

6.1 Introduction

When FPGA accelerators are used to speed up a software application, these are normally designed at a low level of abstraction (typically RTL) in order to obtain an efficient implementation, and this can consume more time and make reuse difficult when compared with a similar software design. A designer must specify the structure of the datapath and control interfaces for reading inputs from memories into buffers, stalling the datapath when buffers are full or empty, writing outputs to memory, and so on. This makes the design process complex, requiring low-level device expertise and knowledge of both hardware and software systems. When

coding an accelerator, a fully pipelined datapath implementation of several lines of C code requires many lines of RTL code. Vendor implementation tools take the RTL description of an accelerator and generate the FPGA implementation, through logic synthesis, mapping, and placement and routing (PAR) processes, which are time-consuming compared to software compilation. The difficulty of accelerator design and the long, complex compilation flow are two key design productivity issues standing in the way of more mainstream adoption of FPGAs in general purpose computing [34].

High level synthesis (HLS) tools [30, 31] have helped simplify accelerator design by raising the level of programming abstraction from RTL to high level languages, such as C or C++. These tools allow the functionality of an accelerator to be described at a higher level to reduce developer effort, enable design portability, and enable rapid design space exploration, thereby improving productivity, verifiability, and flexibility. Raising the level of programming abstraction reduces the amount of information required to describe the functionality of an accelerator, typically with a marginal area and performance cost. For example, Bluespec [90] abstracts interface methods for control and concurrency, though it still describes cycle-level behavior of resources. Higher level tools use languages such as C or C++ where timing is no longer explicit. Most high level languages, like C/C++, are sequential programming languages with no standardized means to describe parallel execution. HLS tools extract parallelism in the datapath through multiple steps to build a hardware implementation.

At the same time, the significant amount of resources available on modern FPGAs could be exploited to replicate hardware to process more data in parallel. However, such datapath replication remains a manual design activity. Recently, FPGA vendors have been exploring explicitly parallel languages, such as OpenCL in order to bridge this gap between the expressiveness of sequential languages and the parallel capabilities of the hardware [178]. In OpenCL, parallelism is explicitly specified by the programmer, and compilers can use system information at runtime to scale the performance of the application by executing multiple replicated copies of the application kernel in hardware [179].

OpenCL also has the benefit of being portable across architectures, such as FPGAs, GPUs, and other parallel compute resources without requiring changes to algorithm source code [180]. This is a key capability of OpenCL that makes it a promising programming model for heterogeneous platforms. Specifically, the introduction of heterogeneous system on chip (SoC) platforms, or hybrid FPGAs, which tightly couple general purpose processors with high performance FPGA fabrics [56] provide a more energy efficient alternative to high performance CPUs and/or GPUs within the tight power budget required by high performance embedded systems. Hence techniques for mapping OpenCL kernels to FPGA hardware have attracted both academic and industrial attention in the last few years [181, 182, 183]. Altera OpenCL [178] and Xilinx SDAccel are new tools that allow compilation and offloading of OpenCL kernels onto FPGA fabric.

While these developments in HLS and OpenCL support have tackled design difficulty to a reasonable extent, allowing designers to focus on high level functionality instead of low-level implementation details, prohibitive compilation times (specifically PAR times) still limit design productivity significantly. Design iterations are extremely slow, and as FPGA devices grow in size, and designs occupy larger areas, this problem continues to worsen. Coarse-grained overlay architectures have emerged as a possible solution to this challenge [73, 74, 75]. Previous chapters have demonstrated ways in which more efficient overlays can be built [78, 79], and coupled with similar high-level design methods, they can address both aspects of the design productivity gap. More interestingly, the complexity of traditional FPGA implementation tools precludes their use in any scenario where accelerators are to be built online, even by a powerful host CPU. However, compile flows for overlays are multiple orders of magnitude faster, and can in fact be run on the embedded processors in hybrid FPGAs.

A method for compilation of OpenCL kernels to the TILT overlay was presented in [144]. Since the Altera OpenCL tool maximizes throughput at the cost of significant resource usage by generating a heavily pipelined, spatial design, the authors suggest the TILT overlay as an alternative target when a lower throughput is adequate, resulting in less area consumption. TILT uses a weaker form of application

customization by varying the mix of pre-configured standard FUs and optionally generating application-dependent custom units. However, as each application requires that the TILT overlay be recompiled, a hardware context switch (referred to as a kernel update in the paper) takes on average 38 seconds. An 8-core TILT system (with each core having one multiply FU and one add/sub FU) was designed specifically to implement a 64-tap FIR filter application, resulting in a throughput of 30 M inputs/sec and consuming 12K eALMs. For the same application, Altera OpenCL HLS was used to generate a fully parallel and pipelined implementation, resulting in a throughput of 240 M inputs/sec (8× higher) and consuming 51K eALMs (4× higher).

In [138], overlays having one dedicated functional unit were used for each OpenCL kernel operation. Five different relatively small sized overlays (2 floating point and 3 fixed point), each specialized for a specific set of kernels with no support for kernels not known at design time were implemented on a Xilinx Virtex-6 FPGA (XC6VCX130T). These overlays were able to achieve frequencies ranging from 196 MHz to 256 MHz. When the overlay residing on the FPGA fabric did not support a kernel, it was proposed to reconfigure the FPGA fabric at runtime with a different overlay which supported the new kernel. This is because different applications require different sized overlays, with an overlay large enough to satisfy the resource requirements of the largest kernel being heavily underutilized when a small kernel is mapped to the overlay. However, this requires a full reconfiguration of the FPGA fabric with a new overlay and takes 3.4 ms in the best case.

In this chapter, we present a methodology for compiling high level descriptions of compute kernels (C/OpenCL) onto efficient coarse-grained overlays, rather than directly to the FPGA fabric to improve design productivity. In the case of compiling OpenCL kernels, the methodology benefits from the high level of abstraction afforded by using the OpenCL programming model, while the mapping to overlays offers fast compilation. We use a custom mapping tool to provide a rapid, vendor independent, mapping to the overlay, demonstrating that the proposed approach raises the abstraction level while also reducing compilation time significantly.

The main contributions of this chapter can be summarized as follows:

- A mapping tool that takes a high level description of a compute kernel (C/OpenCL), bypasses the conventional FPGA compilation process, and maps directly to a coarse-grained overlay previously mapped to the FPGA
- A comparison of PAR times between the traditional PAR approach (for fine-grained FPGAs) and the proposed PAR approach (for overlays), for a set of benchmarks
- A demonstration of this flow running entirely on the embedded processor in a Xilinx Zynq, with low runtime

6.2 Compiling Kernels to the Overlays

In this section we demonstrate the compilation flow for the DISO, Dual-DISO and DeCO overlays. As mentioned previously in chapter 4, DISO and Dual-DISO, both consist of a spatially configured array of functional units interconnected using an island-style interconnect architecture. Each FU executes a single arithmetic operation and data is transferred over a dedicated point-to-point link between the FUs. That is, both the FU and the interconnect are unchanged while a compute kernel is executing. This results in a fully pipelined, throughput oriented programmable datapath executing one kernel iteration per clock cycle, thus having an initiation interval (II) between kernel data packets of one.

The design and implementation of the overlay itself still relies on the conventional hardware design flow using vendor tools. However, this process is done once offline and so does not impact the kernel implementation of an application. It is worth mentioning that some other overlay methodologies do indeed require the overlay itself to be adapted to the kernel being mapped, and hence lose the benefit of fast compilation [143]. Instead of compiling high level application kernels to RTL and then generating a bitstream using the vendor tools, we use an in-house mapping flow to provide a rapid, vendor independent, mapping to the overlay. The mapping

process comprises two main steps; DFG extraction from a kernel description and DFG mapping onto the overlay; and are described in detail in next sections.

6.2.1 DFG Extraction From a Kernel Description

Starting with a high level description of the compute kernel, in either C or OpenCL, we first extract a DFG representation of the kernel, where a node in the DFG represents an operation and an edge represents data dependency between nodes. DFG extraction uses existing HLS tools (HercuLeS [184] for C and LLVM [185] for OpenCL). We use two separate HLS tools as HercuLeS provides direct DFG generation, but unfortunately does not support OpenCL. While LLVM supports both C and OpenCL, it is unable to generate a DFG directly, and a separate tool needed to be written to produce the DFGs. We subsequently integrate with our custom mapping tool-flow to produce a seamless process for application kernel mapping targeting our overlay architectures.

DFG extraction from a C description: The HercuLeS front-end is used to extract the DFG from a C description of the kernel. The C description (shown in Table 6.1(a)) is first passed to GCC for GIMPLE dump generation (shown in Table 6.1(b)), which is then processed by gimple2nac to generate a NAC representation (shown in Table 6.1(c)) and finally it is converted into a DFG (in dot file format) using nac2cdfg, where a node can either be an operation node or an I/O node, as shown in Table 6.3.

GIMPLE is the intermediate representation (IR) used within the GCC compiler and NAC is the representation used by the HercuLeS tool which gets converted into a DFG. Figure 6.2(a) shows an example DFG, showing the nodes and edges.

DFG extraction from OpenCL description: The HercuLeS front-end currently does not support an OpenCL description of compute kernels. To support OpenCL, we use LLVM, which can extract an optimized LLVM intermediate representation (IR) from an OpenCL description, which we then convert to a DFG using our custom IR parser. DFG generation has the following steps: The Clang

(a) C description of the kernel

```
int example_kernel(int x)
{
        int temp = 16*x;
        return (x*(x*(temp*x-20)*x+5));
}
```

(b) GIMPLE representation of the kernel

```
example_kernel (int x)
gimple_bind <
int D.2534;
int D.2535;
int D.2536;
int D.2537;
int D.2538;
int D.2539;
int temp;

gimple_assign <mult_expr, temp, x, 16>
gimple_assign <mult_expr, D.2535, temp, x>
gimple_assign <plus_expr, D.2536, D.2535, -20>
gimple_assign <mult_expr, D.2537, D.2536, x>
gimple_assign <mult_expr, D.2538, D.2537, x>
gimple_assign <plus_expr, D.2539, D.2538, 5>
gimple_assign <mult_expr, D.2534, D.2539, x>
gimple_return <D.2534>
>
```

(c) NAC representation of the kernel

```
procedure example_kernel(in s32 x,out s32 D_2534)
{
localvar s32 D_2535;
localvar s32 D_2536;
localvar s32 D_2537;
localvar s32 D_2538;
localvar s32 D_2539;
localvar s32 temp;
L0005:
temp <= mul x,16;
D_2535 <= mul temp,x;
D_2536 <= sub D_2535,20;
D_2537 <= mul D_2536,x;
D_2538 <= mul D_2537,x;
D_2539 <= add D_2538,5;
D_2534 <= mul D_2539,x;
}
```

Table 6.1: Code descriptions for DFG extraction from C

front-end for the LLVM compiler, along with the LLVM disassembler generates an IR from the OpenCL Kernel. We generate the LLVM IR, after compiling the OpenCL kernel (shown in Table 6.2(a)) using Clang with the -O0 optimization flag, as shown in Table 6.2(b). LLVM optimization passes are then used to generate an optimized LLVM IR, as shown in Table 6.2(c). Lastly, our custom IR parser transforms the optimized IR description of the compute kernel into a DFG description, as shown in Table 6.3.

(a) OpenCL description of the kernel

```
__kernel void example_kernel(__global int *A, __global int *B)
{
        int idx = get_global_id(0);
        int x = A[idx];
        B[idx] = (x*(x*(16*x*x-20)*x+5)) ;
}
```

(b) LLVM intermediate representation (IR) of the kernel

```
%0:
%1 = alloca i32*, align 4
%2 = alloca i32*, align 4
%idx = alloca i32, align 4
%x = alloca i32, align 4
store i32* %A, i32** %1, align 4
store i32* %B, i32** %2, align 4
%3 = call i32 bitcast (i32 (...)* @get_global_id to i32 (i32)*)(i32 0)
store i32 %3, i32* %idx, align 4
%4 = load i32* %idx, align 4
%5 = load i32** %1, align 4
%6 = getelementptr inbounds i32* %5, i32 %4
%7 = load i32* %6
store i32 %7, i32* %x, align 4
%8 = load i32* %x, align 4
%9 = load i32* %x, align 4
%10 = load i32* %x, align 4
%11 = mul nsw i32 16, %10
%12 = load i32* %x, align 4
%13 = mul nsw i32 %11, %12
%14 = sub nsw i32 %13, 20
%15 = mul nsw i32 %9, %14
%16 = load i32* %x, align 4
%17 = mul nsw i32 %15, %16
%18 = add nsw i32 %17, 5
%19 = mul nsw i32 %8, %18
%20 = load i32* %idx, align 4
%21 = load i32** %2, align 4
%22 = getelementptr inbounds i32* %21, i32 %20
store i32 %19, i32* %22
ret void
```

(c) Optimized LLVM IR of the kernel

```
%0:
%1 = call i32 bitcast (i32 (...)* @get_global_id to i32 (i32)*)(i32 0)
%2 = getelementptr inbounds i32* %A, i32 %1
%3 = load i32* %2
%4 = mul nsw i32 16, %3
%5 = mul nsw i32 %4, %3
%6 = sub nsw i32 %5, 20
%7 = mul nsw i32 %3, %6
%8 = mul nsw i32 %7, %3
%9 = add nsw i32 %8, 5
%10 = mul nsw i32 %3, %9
%11 = getelementptr inbounds i32* %B, i32 %1
store i32 %10, i32* %11
ret void
```

Table 6.2: Code descriptions for DFG extraction from OpenCL

```
digraph example_kernel {
N8 [ntype="operation", label="add_Imm_5_N8"];
N9 [ntype="outvar", label="O0_N9"];
N1 [ntype="invar", label="I0_N1"];
N2 [ntype="operation", label="mul_N2"];
N3 [ntype="operation", label="mul_N3"];
N4 [ntype="operation", label="mul_Imm_16_N4"];
N5 [ntype="operation", label="mul_N5"];
N6 [ntype="operation", label="mul_N6"];
N7 [ntype="operation", label="sub_Imm_20_N7"];
N8 -> N2;
N1 -> N5;
N1 -> N6;
N1 -> N2;
N1 -> N3;
N1 -> N4;
N2 -> N9;
N3 -> N6;
N4 -> N5;
N5 -> N7;
N6 -> N8;
N7 -> N3;
}
```

Table 6.3: Compute kernel DFG description

6.2.2 DFG Mapping onto the Overlay

After extracting the DFG from the high level description of the compute kernel, the next step is to map the DFG onto the overlay. This process includes mapping of the DFG nodes onto the overlay FUs, FU netlist generation, placement and routing of the FU netlist onto the overlay, latency balancing and finally, configuration generation. The mapping process is shown in Figure 6.1, and is described in detail next.

6.2.2.1 DFG to FU-aware DFG Transformation

In this step, the DFG description is parsed and translated into an FU-aware DFG. This involves merging nodes that can be combined into a single FU, based on the capabilities of the DSP block primitive. For example, we can use multiply-subtract and multiply-add to collapse N5–N7 and N6–N8 in Figure 6.2(a) into N5 and N6 of Figure 6.2(b), respectively. As a result, the FU aware mapping requires only 5 FUs instead of the 7 required if each node were mapped to a single FU, as in

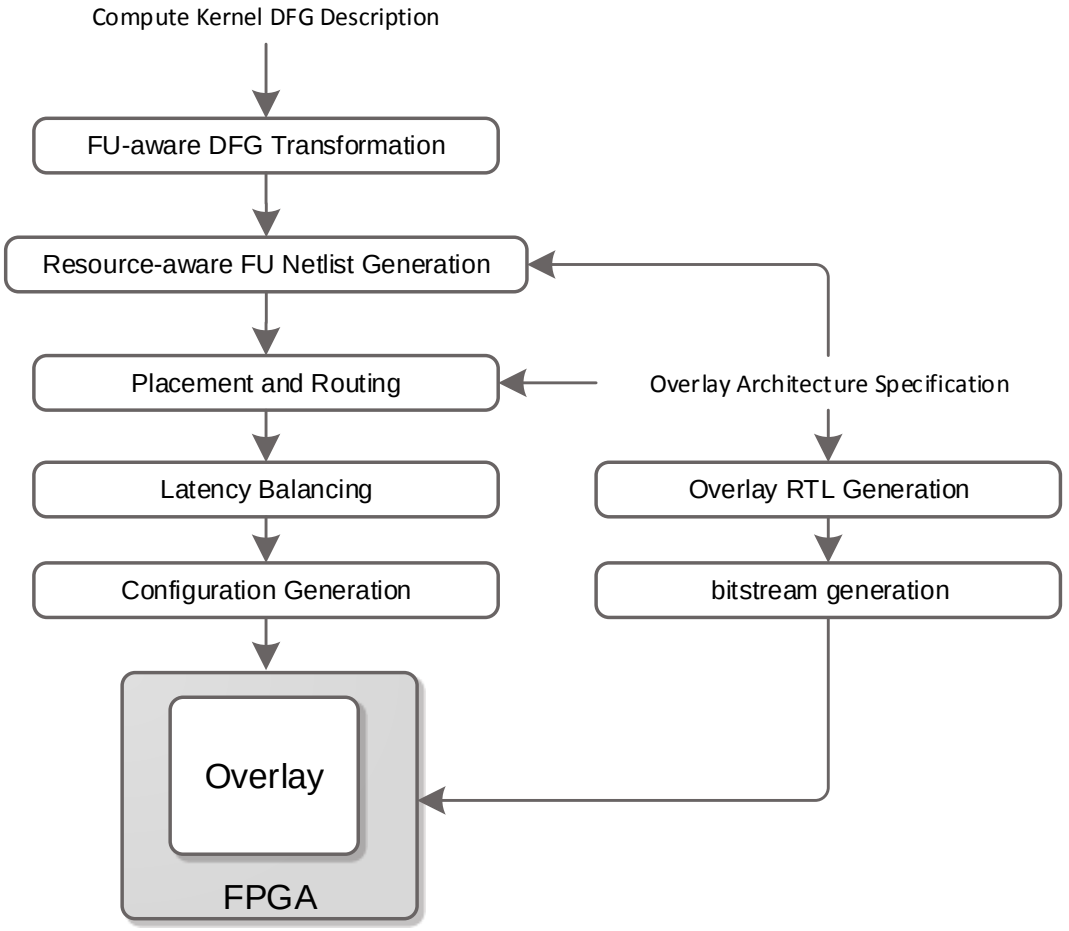


Figure 6.1: Mapping flow.

many other overlays. This results in the FU-aware DFG shown in Table 6.4 and Figure 6.2(b).

Furthermore, an FU can be made more complex by putting multiple DSP blocks within it so that it can further reduce the number of required FUs. For example using two DSP blocks within an FU, as with the Dual-DISO overlay, N4 and N5 can be combined together and similarly N3 and N6 can be combined together, resulting in another FU-aware graph as shown in Figure 6.2(c). In order to support the dual-DSP FU, we cluster two consecutive nodes in the single DSP-aware DFG if the fan-in of the resulting node, is ≤ 4. Dual-DSP based clustering results in a significant reduction in the number of FUs required compared to an FU with just a single DSP block, also meaning less global routing resources are needed. Our

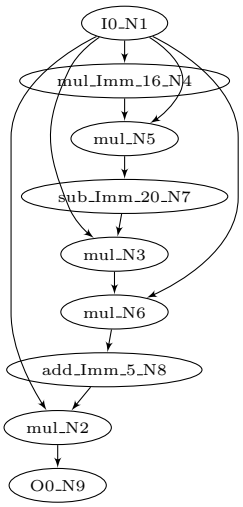
```
digraph example_kernel {
N7 [ntype="outvar", label="O0_N7"];
N1 [ntype="invar", label="I0_N1"];
N2 [ntype="operation", label="mul_N2"];
N3 [ntype="operation", label="mul_N3"];
N4 [ntype="operation", label="mul_Imm_16_N4"];
N5 [ntype="operation", label="mul_sub_Imm_20_N5"];
N6 [ntype="operation", label="mul_add_Imm_5_N6"];
N1 -> N5;
N1 -> N6;
N1 -> N2;
N1 -> N3;
N1 -> N4;
N2 -> N7;
N3 -> N6;
N4 -> N5;
N5 -> N3;
N6 -> N2;
}
```

Table 6.4: FU-aware DFG description for single-DSP FU

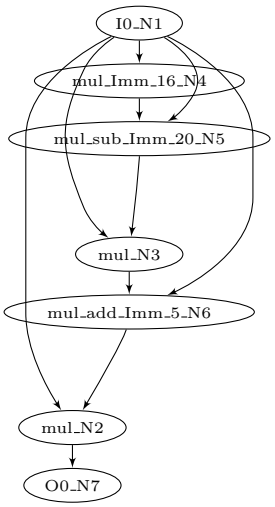
FU-aware DFG transformation currently only supports Xilinx DSP block based FUs, but could be easily modified to support any user-defined FU type.

6.2.2.2 Resource-aware FU Netlist Generation From FU-aware DFG

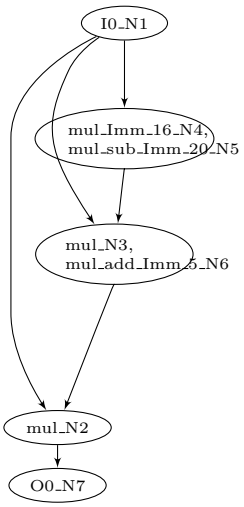
The FU-aware DFG for the kernel can be replicated the appropriate number of times to fit the available resources of the overlay architecture. This replicated DFG is used to generate the FU netlist. Table 6.5 shows the FU netlist for a single-DSP FU in VPR netlist format which can be used by VPR tool for the placement and routing on island-style overlay. The replication factor used in this example is one which means just one copy of kernel would be mapped onto the overlay. Figure 6.3 shows the concept of resource-aware replication of an FU-aware DFG, where we first map a single kernel of a DSP FU aware DFG onto the DISO overlay and single kernel of a dual-DSP FU aware DFG onto the Dual-DISO overlay. Since it is possible to fit 8 instances of the dual-DSP FU aware DFG onto the Dual-DISO overlay, the tool replicate the dual-DSP FU aware DFG 8 times and generates a resource-aware FU netlist (containing 8 kernel instances) which gets mapped to the Dual-DISO overlay. It is clear from Figure 6.3 that the proposed replication approach can significantly improve the utilization of the overlay resources.



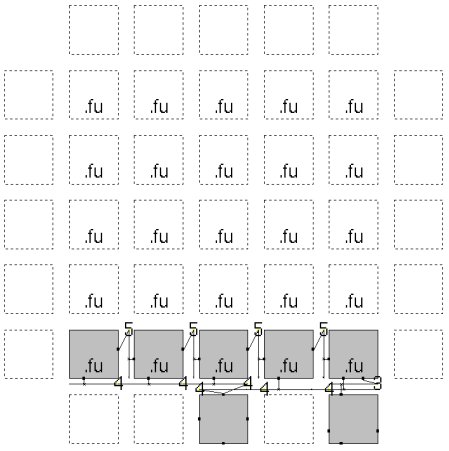
(a) DFG extracted from OpenCL Kernel



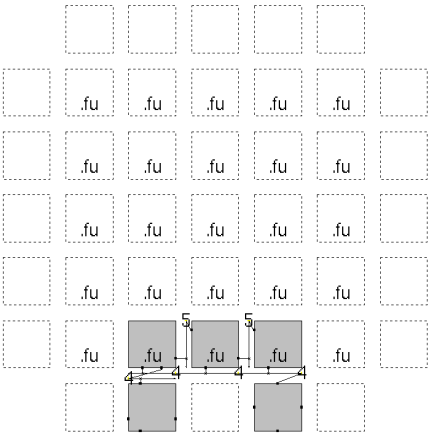
(b) FU-aware DFG where FU consists of one DSP block



(c) FU-aware DFG where FU consists of two DSP blocks



(d) FU-aware DFG placed and routed on 5x5 DISO overlay



(e) FU-aware DFG placed and routed on 5x5 Dual-DISO overlay

Figure 6.2: FU aware mapping, placement and routing on overlay.

To support mapping of the FU-aware DFG on the DeCO architecture, we apply tree-balancing and resource-aware replication technique to all FU-aware DFGs, which both reduces the graph depth and better shapes the DFG according to the DeCO architecture, and generate the FU netlist.

To enable ultra-fast resource-aware mapping at runtime, the process of converting the kernel to the FU-aware DFG can be bypassed by converting the kernel to an FU-aware DFG offline. The FU-aware DFG can then be used at runtime for resource-aware FU-netlist generation and PAR on the overlay. We demonstrate

```
.input N1
pinlist: N1

.output out:N7
pinlist: N7

.fu N2
pinlist: N1 N6 open open N7 open open open open
subblock: N2_blk 0 1 open open 4 open open open open

.fu N3
pinlist: N1 N5 open open N3 open open open open
subblock: N3_blk 0 1 open open 4 open open open open

.fu N4
pinlist: N1 open open open N4 open open open open
subblock: N4_blk 0 open open open 4 open open open open

.fu N5
pinlist: N1 N4 open open N5 open open open open
subblock: N5_blk 0 1 open open 4 open open open open

.fu N6
pinlist: N1 N3 open open N6 open open open open
subblock: N6_blk 0 1 open open 4 open open open open
```

Table 6.5: FU Netlist

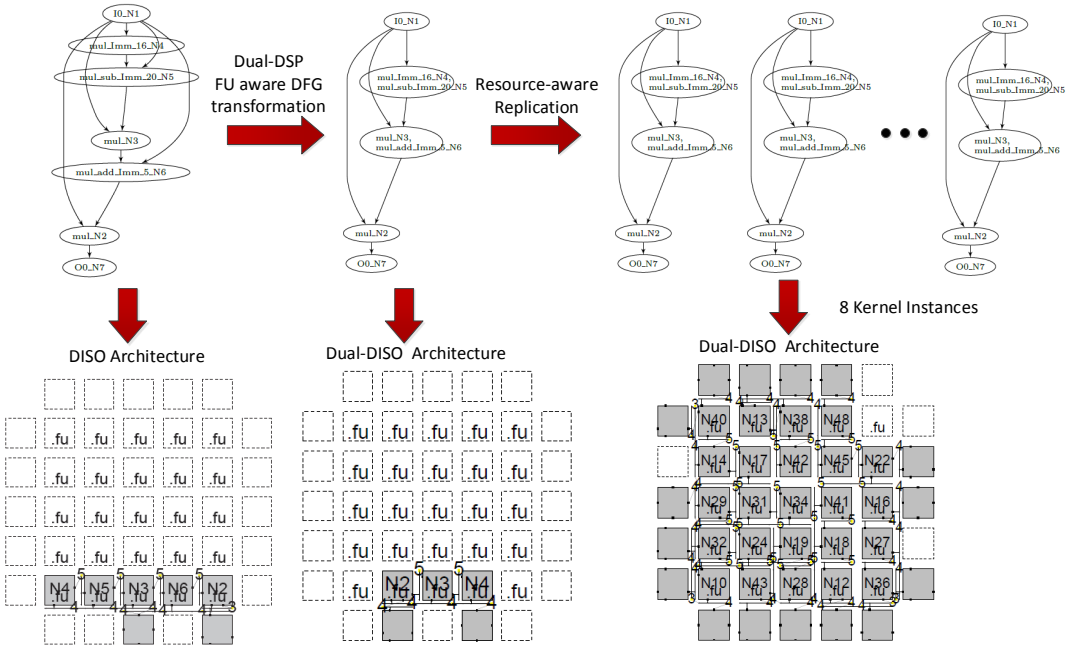


Figure 6.3: An example of resource-aware replication of an FU-aware DFG.

this in Section 6.3 by performing runtime PAR on the Xilinx Zynq Dual ARM embedded processor.

6.2.2.3 Placement and Routing of the FU Netlist to the Overlay

We now make use of VPR to place nodes onto the island-style overlay and route signals between them. In this manner, we are adopting VPR at a higher level of abstraction than its intended purpose. Rather than map logic functions to LUTs and single-bit wires to 1-bit channels, we map nodes in the graph to FUs, and 16-bit wires to 16-bit channels. At this level of granularity, a netlist can have 100s of nodes, making the problem much smaller than that of fine-grained FPGA placement and routing which deals with netlists of millions of nodes. The place and route algorithm maps DFG nodes onto homogeneous FUs and DFG edges to the overlay's routing paths to connect mapped FUs.

The architecture of the DISO and Dual-DISO overlays consists of a traditional island-style topology, arranged as a virtual homogeneous two-dimensional array of tiles. A VPR 5.0 architecture file is used to describe the architecture of overlay, as shown in Table. 6.6 for a 5×5 overlay. The field *layout* is used to define the width and height of the overlay. The switch box type, flexibility f_s of the switch box and channel width distribution are specified in the field *device*. The field *segmentlist* specifies that all the segments are unidirectional and span only one block, resulting in one connection box and two switch boxes per FU.

Figure 6.2(d) shows the DFG of Figure 6.2(b) mapped on a 5×5 DISO overlay using the VPR place and route tool. It shows the connections needed for data flow between the FUs and the routing resources. Similarly, Figure 6.2(e) shows the DFG of Figure 6.2(c) mapped on a 5×5 Dual-DISO overlay.

To support placement and routing of the FU netlist on the DeCO architecture, we perform scheduling on the FU netlist to generate a sequenced FU netlist which is used to place and route the FU netlist on the DeCO architecture in a stage-wise manner. Figure 5.4 in Chapter 5 shows the approach of placement and routing on DeCO architecture.

```
<architecture>
        <layout width="5" height="5" />
        <device>
                <sizing />
                <area />
                <chan_width_distr>
                        <io width="1"/>
                        <x distr="uniform" peak="1"/>
                        <y distr="uniform" peak="1"/>
                </chan_width_distr>
                <switch_block type="wilton" fs="3"/>
        </device>
        <switchlist>
                <switch type="mux" name="0" mux_trans_size="10" buf_size="1" />
        </switchlist>
        <segmentlist>
                <segment freq="1" length="1" type="unidir" >
                <mux name="0"/>
                <sb type="pattern">1 1</sb>
                <cb type="pattern">1</cb>
                </segment>
        </segmentlist>
        <typelist>
                <io capacity="1">
                        <fc_in type="frac">1</fc_in>
                        <fc_out type="frac">1</fc_out>
                </io>
                <type name=".fu">
                        <subblocks max_subblocks="1"
                        max_subblock_inputs="4" max_subblock_outputs="4" >
                        </subblocks>
                        <fc_in type="frac">1</fc_in>
                        <fc_out type="frac">1</fc_out>
                        <pinclasses>
                                <class type="in">0 1 2 3 </class>
                                <class type="out">4 5 6 7 </class>
                                <class type="global">8 </class>
                        </pinclasses>
                        <pinlocations>
                                <loc side="left">3 7 8 </loc>
                                <loc side="right">1 5 </loc>
                                <loc side="top">0 4 </loc>
                                <loc side="bottom">2 6 </loc>
                        </pinlocations>
                        <gridlocations>
                                <loc type="fill" />
                        </gridlocations>
                </type>
        </typelist>
</architecture>
```

Table 6.6: Architecture description of the overlay

6.2.2.4 Latency Balancing

As discussed previously, correct functioning of the mapped compute kernel is ensured only if it is latency balanced, which means that all FU inputs arrive at the same execution cycle. It is clear from Figure 6.2(d) that the inputs at a node might arrive at different clock cycles. For example, at N5 which is mapped onto the FU at (2,1) the inputs arrive in the 13th and 3rd clock cycles. The FUs have

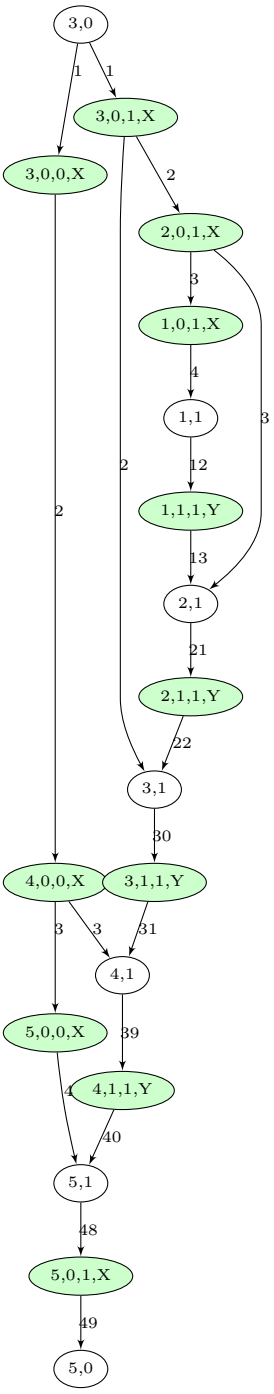


Figure 6.4: Overlay resource graph corresponding to Figure 6.2(d).

delay chains that must be configured to match these input latencies. To determine the latency imbalance at each node, we developed a tool to parse VPR output files and generate an overlay resource graph, as shown in Figure 6.4 for the mapping of Figure 6.2(d). An unshaded node in the graph shows the FU information and a shaded node shows the track information used to carry the data from one FU to another. For example, at N5 the data comes from N1 (placed at (3,0)) via two

tracks and from N4 (placed at (1,1)) via only a single track. The overlay resource graph is used to generate the configuration of the overlay (including the latency imbalance SRL configuration) which can be loaded onto the overlay at runtime by the host processor. Latency balancing step is not required while mapping the kernel onto the DeCO architecture, since the inputs at a node arrive at same clock cycles.

6.2.2.5 Configuration Generation

Next, we generate the configuration data for the FUs and the interconnect resources. This configuration data can then be used to set the programmable settings of the FU and the interconnect resources, implementing the kernel.

6.3 Experiments

Instead of compiling kernels onto relatively small kernel set-specific overlays, we compile replicated instances of kernels onto a large overlay to achieve effective utilization of resources. The size of the overlay on the fabric depends on the free resource after any other logic is mapped. In the case where there is minimal, or no, requirement for other logic we can completely fill the fabric with the largest possible overlay. Hence, the overlays we consider can have different sizes and FU types, with this information being exposed by the OpenCL runtime to the compiler, which can then replicate and map the kernel to utilize the maximum overlay resource.

We compile a set of benchmarks onto the overlay and measure the PAR time. Figure 6.5 shows the comparison of PAR times between three different scenarios. For each benchmark, the first (blue) bar shows the PAR time when Vivado 2014.2 is used targeting the Zynq FPGA fabric. The second (green) bar shows the PAR time when the proposed approach is used targeting the overlay. In both of these cases, we used a HP Z420 workstation with an Intel Xeon E5-1650 v2 CPU running

at 3.5 GHz with 16 GB of RAM, hence the *x86* suffix. The third (red) bar shows the PAR time when the PAR tool is running on the Zedboard consisting of Zynq device having a dual-core ARM Cortex-A9 CPU, running at 667 MHz with 512 MB of RAM. Xillinux-1.3 is used as an operating system running on the dual-core ARM with Portable Computing Language (pocl) infrastructure [186] installed.

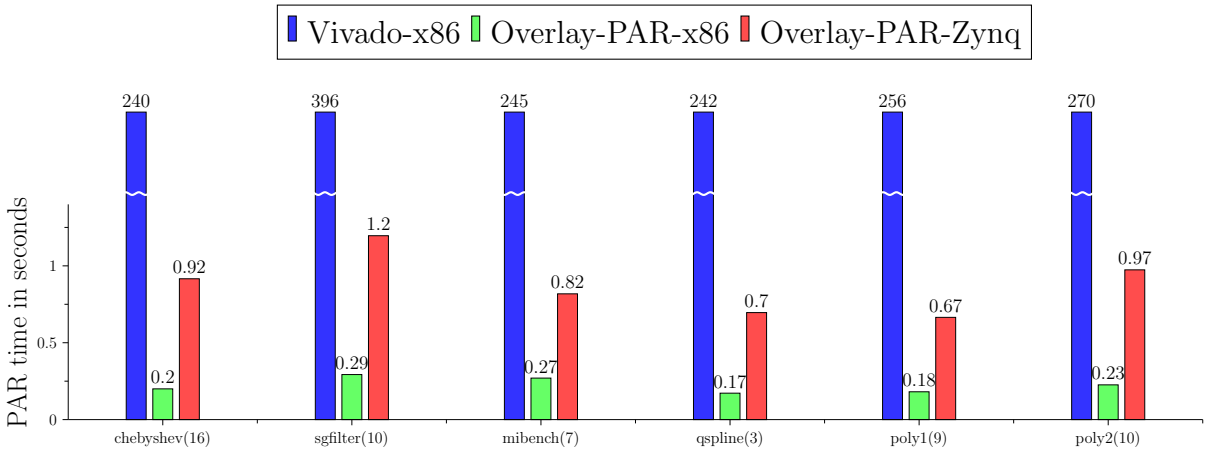


Figure 6.5: Comparison of PAR times (in seconds).

To place and route 16 copies of the *Chebyshev* benchmarks on an 8×8 Dual-DISO overlay, it takes 200 milliseconds on the workstation while on the Zynq ARM processor it takes 916 milliseconds. Vivado takes 240 seconds to place and route the same design, meaning that it is 1200× slower than when the application is mapped to the overlay. When using the Zynq ARM processor, the process is still approx. 250× faster than Vivado. For the set of 8 benchmarks, it takes on an average 200 ms for place and route on the workstation and 1 second on the Zynq. In Figure 6.6, the *Chebyshev* kernel is replicated multiple times to show the effect of kernel replication on PAR time. Note that these times exclude the initial compilation to the FU-aware DFG format which takes on average 3 seconds on a workstation or 15 seconds on the Zynq. We assume that this is acceptable as runtime loading would involve known kernels on a known architecture. And the introduction of new kernels at the cost of a few seconds still results in a significant performance improvement compared to the Vivado HLS approach.

Figure 6.7 shows the heterogeneous infrastructure which includes the overlay in the programmable logic region of the Zynq FPGA. Its size and FU type are exposed

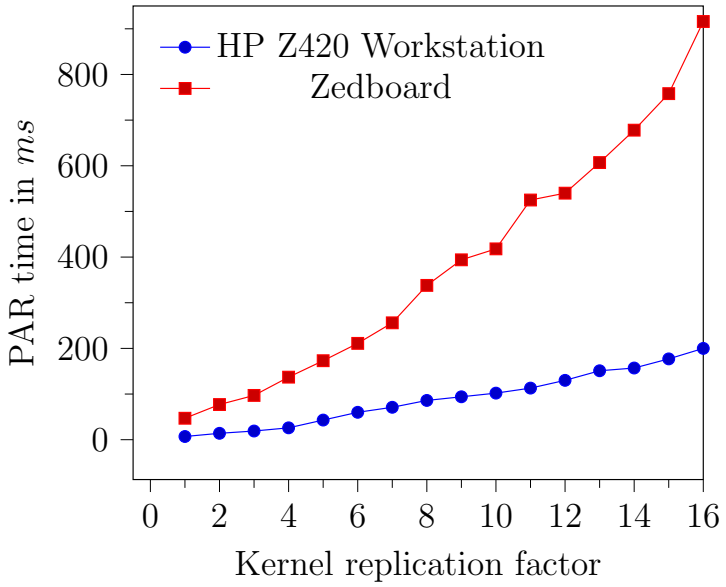


Figure 6.6: Effect of *Chebyshev* kernel replication on PAR time

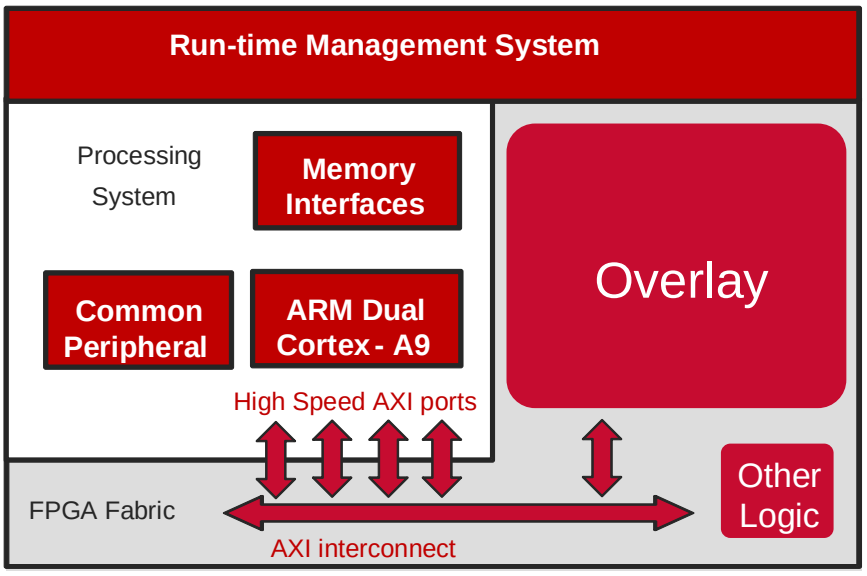


Figure 6.7: Architecture, implemented on the Zynq, consisting of an overlay whose size and FU type can be exposed by OpenCL runtime.

by the OpenCL runtime to the compiler so that it can replicate a suitable number of kernel copies to utilize available overlay resources. There may be situations in which other logic in the system consumes significant resources. In that case, the overlay size can be changed and the fabric can be reconfigured using a new overlay and the other logic. For example, in the case where the other logic is large, leaving only minimal resources for a 2×2 overlay, this information can be exposed by the

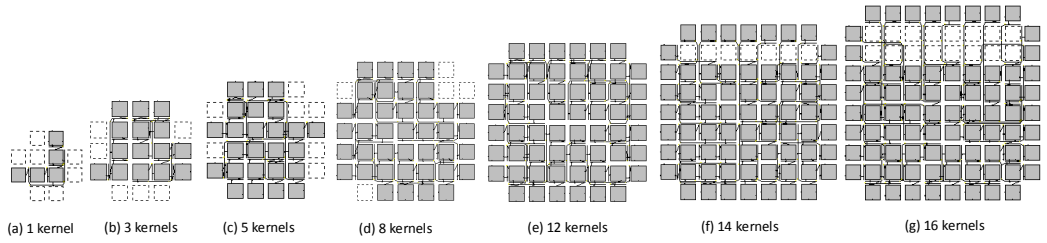


Figure 6.8: Performance scaling by the compiler using overlay size information provided by the OpenCL runtime.

OpenCL runtime to the compiler which can than choose to map only a single copy of the kernel as shown in Figure 6.8(a). In the case where other logic is minimal and most of the fabric resources can be used to map an overlay, we can fit an 8×8 overlay and the compiler can then choose to map 16 copies of the kernel as shown in Figure 6.8(g), limited by the available I/O. Figure 6.8(b)–Figure 6.8(f) show cases in between.

The proposed techniques in this thesis can be effectively used for compiling and offloading deep learning kernels onto overlays at runtime and also for FPGA based database query processing where hardware needs to be changed rapidly for each new query. One important point to note is that the kernel benchmarks used in this thesis are feed-forward compute kernels extracted from compute intensive applications and represented as directed acyclic graphs (DAGs). To support kernels with feedback paths on spatially configured overlays is an area for future research.

6.4 Summary

We have presented an approach for compiling a high level description of compute kernels onto coarse-grained overlays for improving accelerator design productivity. The methodology benefits from the high level of abstraction afforded by using the OpenCL programming model, while the mapping to overlays offers fast compilation, even on an embedded processor. We demonstrate an end-to-end compile flow

with resource aware mapping of kernels to the overlay. Using a typical workstation, the overlay place and route is ≈1200 times faster than the FPGA place and route using Vivado 2014.2. We successfully installed the pocl infrastructure on the Zedboard to support execution of OpenCL applications, and demonstrated place and route onto the overlay running entirely on the Zynq.

7

Conclusions and Future Research Directions

Coarse-grained FPGA overlays have emerged as one possible solution to virtualize FPGA resources, offering a number of advantages for general purpose hardware acceleration because of software-like programmability, fast compilation, application portability, and improved design productivity. These architectures allow rapid hardware design at a higher level of abstraction, but at the cost of area and performance overheads due to limited consideration for the underlying FPGA architecture. This thesis explores coarse grained overlays designed using the flexible DSP48E1 primitive on Xilinx FPGAs, allowing pipelined execution of compute kernels at significantly higher throughput without adding significant area overheads

to the functional unit. Our experimental evaluation shows that the proposed overlays exhibit an achievable frequency which is close to the DSP theoretical limit on the Xilinx Zynq. We also present a methodology for compiling high level language (C/OpenCL) descriptions of compute kernels onto DSP block based coarse-grained overlays, rather than directly to the FPGA fabric. Our mapping flow provides a rapid, vendor independent mapping to the overlay, raising the abstraction level while also reducing compile times significantly, hence addressing the design productivity issue. We map several benchmarks, using our mapping tool, and show that the proposed overlay architectures deliver better throughput compared to Vivado HLS generated fully pipelined RTL implementations. This chapter draws conclusions from the different contributions described in this thesis and outlines areas for future research.

7.1 Summary of Contributions

7.1.1 Adapting the DySER Architecture as an FPGA Overlay

In chapter 3, we evaluated an open source overlay architecture, DySER, mapped on the Xilinx Zynq SoPC and demonstrated that DySER suffers from a significant area and performance overhead due to limited consideration for the underlying FPGA architecture. We then proposed an improved functional unit architecture using the flexibility of the DSP48E1 primitive which results in a 2.5 times frequency improvement and 25% area reduction compared to the original functional unit architecture. We demonstrated that this improvement results in the routing architecture becoming the bottleneck in performance. Our adapted version of a 6x6 16-bit DySER was implemented on a Xilinx Zynq by using a DSP block as the compute logic, referred to as DSP-DySER, providing a peak performance of 6.3 GOPS with an interconnect area overhead of 7.6K LUTs/GOPS. We also quantified the area overheads by mapping a set of benchmarks to the DSP-DySER and

directly to the FPGA fabric using Vivado HLS. DSP-DySER suffers from 25× hardware performance penalty compared to the HLS generated hardware implementations.

7.1.2 Throughput Oriented FPGA Overlays Using DSP Blocks

In chapter 4, we designed and implemented FPGA targeted overlay architectures (DISO and Dual-DISO) that maximize the peak performance and reduce the interconnect area overhead through the use of an array of DSP block based fully pipelined FUs and an island-style coarse-grained routing network. A scalability analysis of DISO on the Xilinx Zynq device shows that the Zynq fabric can accommodate an 8×8 DISO overlay, achieving a peak performance of 65 GOPS (10× better than DSP-DySER) with an interconnect area overhead of 430 LUTs/GOPS (18× better than DSP-DySER).

We then presented an analysis of a wide variety of compute kernels using a DSP48E1 aware data flow graph based approach to ascertain the suitability of mapping multiple instances of kernels to the overlay and observed that the dual-DSP block based overlay is most suitable for our benchmark set. We then presented a prototype of an enhanced version of DISO (referred to as Dual-DISO) which uses two DSP blocks within each FU and shows a significant improvement in performance and scalability, with a reduction of almost 70% in the overlay tile requirement compared to existing overlay architectures and an operating frequency in excess of 300 MHz. A scalability analysis of Dual-DISO on the Xilinx Zynq device shows that Zynq fabric can accommodate an 8×8 Dual-DISO overlay, achieving a peak performance of 115 GOPS (18× better than DSP-DySER) with an interconnect area overhead of 320 LUTs/GOPS (24× better than DSP-DySER). We demonstrated that this improvement results in better exploitation of the performance provided by the DSP blocks available on the FPGA fabric.

We then mapped several benchmark kernels onto the proposed overlays and demonstrated that the proposed overlays can deliver better throughput compared to Vivado HLS generated fully pipelined RTL implementations. Our experimental evaluation demonstrated that the DISO overlay delivers kernel throughputs of up to 21.6 GOPS (33% of the peak theoretical throughput of the DISO overlay) while the Dual-DISO overlay delivers kernel throughputs of up to 57.6 GOPS (50% of the peak theoretical throughput of the Dual-DISO overlay) and provides an average throughput improvement of 40% over Vivado HLS for the same implementations of the benchmark set. Using the Dual-DISO overlay, we demonstrated that it is possible to map multiple instances of the benchmark kernels to the overlay automatically, resulting in more efficient utilization of overlay resources, without resorting to reconfiguring the FPGA fabric at runtime. We have demonstrated that architecture-focused FPGA overlays can better exploit the raw performance of the DSP blocks, with better resource utilization and significantly improved performance metrics, compared to other overlays.

7.1.3 Low Overhead Interconnect for DSP Block Based Overlays

As the island-style interconnect of the DISO overlay is still somewhat excessive, and negates many of the advantages of overlays, we next explored novel interconnect architectures to further reduce the interconnect area. Crucially, the interconnect flexibility provided by these overlay architectures is normally overprovisioned for accelerators based on feed-forward pipelined data paths, which in many cases have the general shape of inverted cones. Thus in chapter 5, we proposed DeCO, a cone shaped cluster of FUs utilizing a simple linear interconnect. The DeCO architecture reduces the area overheads for implementing compute kernels extracted from compute-intensive applications represented as directed acyclic dataflow graphs, while still allowing high data throughput. We performed design space exploration by modelling the programmability overhead as a function of the overlay design parameters, and compared to island-style overlays. The 16-bit

DeCO, when implemented on a Xilinx Zynq, achieves savings in the LUT requirements of 96% and 87%, compared to 16-bit DSP-DySER[177] and 16-bit DISO[78], respectively for our benchmark set of compute kernels. 16-bit DeCO achieves a frequency of 395 MHz and provides a peak performance of 23.7 GOPS (3.8× better than DSP-DySER) with an interconnect area overhead of 58 LUTs/GOPS (130× better than DSP-DySER), and with just a 1.5× hardware performance penalty compared to HLS generated hardware implementations. We have demonstrated that the use of an architecture-focused FU and low overhead interconnect can result in an efficient overlay architecture with significantly lower area and performance overheads compared to other overlays.

7.1.4 Mapping Tool for Compiling Kernels onto Overlays

In chapter 6, we presented a methodology for compiling high level language (C/OpenCL) descriptions of compute kernels onto DSP block based coarse-grained overlays, rather than directly to the FPGA fabric. Our mapping flow provides a rapid, vendor independent mapping to the overlay, raising the abstraction level while also reducing compile times significantly, hence addressing the design productivity issue. We demonstrate an end-to-end compilation flow with resource aware mapping of kernels to the overlay. Using a typical workstation, the overlay place and route time is 1200× faster than the FPGA place and route using Vivado design suite. The methodology benefits from the high level of abstraction afforded by using the OpenCL programming model, while the mapping to overlays offers fast compilation in the order of a few milliseconds, even on an embedded processor. We successfully installed the *pocl* infrastructure on the Zedboard to support execution of OpenCL applications, and demonstrated place and route onto the overlay running entirely on the embedded processor of the Zynq device.

7.2 Future Research Directions

We have shown that architecture centric FUs can be designed using the dynamic mode control feature of DSP Blocks for improving the performance of FPGA overlays. As a next step, we plan to release the overlays and tool-flow discussed in this thesis as open source for use by other researchers. In addition to this, we have identified a number of possible extensions to the different aspects of the work presented which can be explored in future research.

7.2.1 Using DSP Blocks for Building Time-multiplexed Overlays

The coarse-grained overlays proposed in this thesis can deliver maximum performance by executing one computation iteration every clock cycle (that is they have an II of one), but with the requirement of one FU for each operation in the compute kernel. Alternatively, a time-multiplexed overlay with its reduced FPGA resource requirements may be a feasible alternative allowing the remainder of the FPGA fabric to be utilized for other purposes. However, the exact architecture needs to be carefully analysed taking into account the characteristics of the application kernels and the underlying FPGA architecture.

Some of the area efficient overlays utilize a simple linear interconnect structure, which can reduce to a simple direct connection between FUs by allocating DFG nodes from the same scheduling time step to the individual FUs. For example, Figure 7.1 shows the medical imaging 'gradient' benchmark [62], while Figure 7.2 shows the resulting data flow graph (DFG). By using a simple ASAP schedule, we can allocate the nodes in each stage to a different FU which in this example 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. Thus, for the example shown in Figure 7.2, the

II would be 11, consisting of 5 cycles for data entry, 4 cycles for the 4 subtract operations, 1 cycle for data output and 1 cycle to flush the pipeline. Note that multiplexing the kernel operations of the DFG in Figure 7.2 to a single FU would result in an II of 17 (consisting of 5 load, 11 operation and 1 store, assuming best case execution without NOP insertions), while a spatially configured overlay would require 11 FUs with an II of 1.

```
#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]);
```

Figure 7.1: C code section for the 'gradient' benchmark

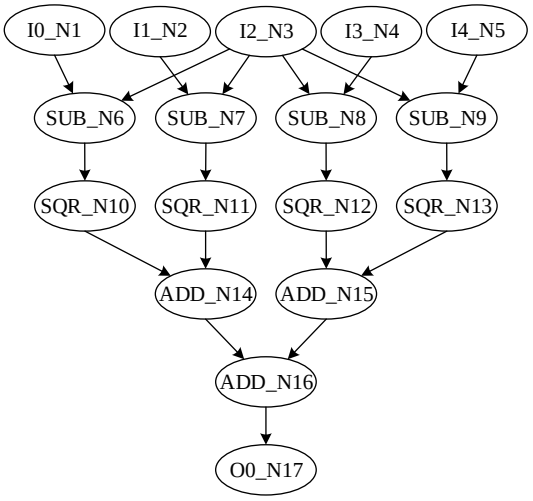


Figure 7.2: DFG for the 'gradient' benchmark

Furthermore, this means that only a small subset of all possible instructions needs to be stored in each FU, resulting in a very small memory requirement overcoming the problem of the large area overheads due to the instruction storage requirements in existing time-multiplexed overlays. This can be further assisted by using a simple low resource, but highly pipelined, FU (similar to the iDEA soft core processor [124]). We presented some of the preliminary work towards time-multiplexing the FU within an overlay in [82]. Additionally, apart from the DSP48E1 block used in this thesis for developing overlays, more advanced DSP

blocks, such as the DSP48E2 and the Altera floating point DSP, could be used for supporting overlays on more advanced FPGA fabrics.

7.2.2 Interfacing Overlays to a Host Processor

In a heterogeneous computing platform, where the overlays can be used to accelerate compute intensive tasks of an application, the major concern is the integration of the overlay with the host processor and the efficiency of the software-hardware communication. Managing software-hardware communication is normally one of the processor's many tasks, and hence this must be done in a way that does not degrade overall system performance. Low latency and high bandwidth are the key requirements to enhance the efficiency of the overall system. Overlays can be integrated with the host processor over many different types of communication interfaces, such as PCIe, Ethernet and AXI. However, it is necessary to develop a memory subsystem around the overlay to manage the transportation of data to and from the overlay via the communication interfaces. We presented some of the preliminary work towards the interfacing of an overlay with a host processor in [83, 84].

7.2.3 OpenCL Driver and Runtime for Overlays

Recently, FPGA vendors have been exploring explicitly parallel languages, such as OpenCL, in order to bridge the gap between the expressiveness of sequential languages and the parallel capabilities of the hardware [178]. In OpenCL, parallelism is explicitly specified by the programmer, and compilers can use system information at runtime to scale the performance of the application by executing multiple replicated copies of the application kernel in hardware [179]. OpenCL also has the benefit of being portable across architectures, such as FPGAs, GPUs, and other parallel compute resources without requiring changes to algorithm source code [180]. This is a key capability of OpenCL that makes it a promising programming model for heterogeneous platforms.

Specifically, the introduction of heterogeneous system on chip (SOC) platforms, or hybrid FPGAs, provide a more energy efficient alternative to high performance CPUs and/or GPUs within the tight power budget required by high performance embedded systems. Hence techniques for mapping OpenCL kernels to FPGA hardware have attracted both academic and industrial attention in the last few years [181, 182, 183]. Altera OpenCL [178] and Xilinx SDAccel are new tools that allow compilation and offloading of OpenCL kernels onto FPGA fabric. The tool-flow presented in Chapter 6 of this thesis can be used to compile OpenCL kernels onto the proposed overlays. However, it is necessary to develop an OpenCL driver and runtime for the offloading of OpenCL kernels to the proposed overlays.

Bibliography

[1] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE micro*, 30(2):56–69, 2010.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[3] B. D. de Dinechin, D. V. Amstel, M. Poulhies, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 97:1–97:6, 2014.

[4] B. D. de Dinechin, R. Ayrignac, P. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *Proceedings of the International Conference on High Performance Extreme Computing Conference (HPEC)*, 2013.

[5] L. Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 6(13):11–02, 2011.

[6] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell. Programming the Adapteva Epiphany 64-core network-on-chip coprocessor. In *Parallel Distributed Processing Symposium Workshops (IPDPSW)*, pages 984–992, May 2014.

[7] X. Zhang and K. K. Parhi. High-speed VLSI architectures for the AES algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(9):957–967, 2004.

[8] L. Liu, N. Chen, H. Meng, L. Zhang, Z. Wang, and H. Chen. A VLSI architecture of JPEG2000 encoder. *IEEE Journal of Solid-State Circuits*, 39(11):2032–2040, 2004.

[9] A. Hodjat, D. D. Hwang, B. Lai, K. Tiri, and I. Verbauwhede. A 3.84 gbits/s AES crypto coprocessor with modes of operation in a 0.18-μm CMOS technology. In *Proceedings of the ACM Great Lakes symposium on VLSI*, pages 60–63. ACM, 2005.

[10] L. Liu, H. Meng, L. Zhang, and Z. Wang. An ASIC implementation of JPEG2000 codec. In *Proceedings of the Custom Integrated Circuits Conference*, pages 691–694. IEEE, 2005.

[11] A. Hodjat and I. Verbauwhede. A 21.54 gbits/s fully pipelined AES processor on FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 308–309. IEEE, 2004.

[12] A. Descampe, F. Devaux, G. Rouvroy, B. Macq, and J. Legat. An efficient FPGA implementation of a flexible JPEG2000 decoder for digital cinema. In *European Signal Processing Conference*, pages 2019–2022. IEEE, 2004.

[13] O. T. Albaharna, P. Y. K. Cheung, and T. J. Clarke. On the viability of FPGA-based integrated coprocessors. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 206–215, 1996.

[14] S. Paul and S. Bhunia. A survey of computing architectures. In *Computing with Memory for Energy-Efficient Robust Systems*, pages 11–27. Springer, 2014.

[15] R. Tessier, K. Pocek, and A. DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

[16] S. M. Trimberger. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

[17] A. DeHon. Fundamental underpinnings of reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):355–378, 2015.

[18] A. George, H. Lam, and G. Stitt. Novo-G: at the forefront of scalable reconfigurable supercomputing. *Computing in Science Engineering*, 13(1):82–86, 2011.

[19] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Survey*, 34:171–210, June 2002.

[20] G. Brebner and W. Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, 2014.

[21] D. V. Schuehler and J. W. Lockwood. A modular system for FPGA-based TCP flow processing in high-speed networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 301–310. Springer, 2004.

[22] H. Parandeh-Afshar and P. Ienne. Highly versatile DSP blocks for improved FPGA arithmetic performance. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 229–236, 2010.

[23] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[24] K. Gaj and P. Chodowiec. FPGA and ASIC implementations of AES. In *Cryptographic engineering*, pages 235–294. Springer, 2009.

[25] B. Varma, K. Paul, and M. Balakrishnan. Accelerating 3D-FFT using hard embedded blocks in FPGAs. In *Proceedings of the International Conference on VLSI Design and Embedded Systems*, pages 92–97. IEEE, 2013.

[26] M. Jacobsen, P. Meng, S. Sampangi, and R. Kastner. FPGA accelerated online boosting for multi-target tracking. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 165–168. IEEE, 2014.

[27] B. Varma, K. Paul, M. Balakrishnan, and D. Lavenier. Fassem: FPGA based acceleration of de novo genome assembly. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 173–176. IEEE, 2013.

[28] S. Shreejith, S. A. Fahmy, and M. Lukasiewycz. Reconfigurable computing in next-generation automotive networks. *IEEE Embedded Systems Letters*, 5(1):12–15, 2013.

[29] S. Neuendorffer and F. Martinez-Vallina. Building Zynq accelerators with Vivado High Level Synthesis. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 1–2, 2013.

[30] 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 *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 33–36, 2011.

[31] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen. High-level synthesis: Productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012:1–14, January 2012.

[32] W. Najjar and J. Villarreal. FPGA code accelerators - the compiler perspective. In *Proceedings of the Design Automation Conference*, 2013.

[33] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–134, 2010.

[34] G. Stitt. Are field-programmable gate arrays ready for the mainstream? *IEEE Micro*, 31(6):58–63, 2011.

[35] J. Daniels. Server virtualization architecture and implementation. *Crossroads*, 16(1):8–12, September 2009.

[36] K. L. Kroeker. The evolution of virtualization. *Commun. ACM*, 52(3):18–20, March 2009.

[37] S. Nanda and T. Chiueh. A survey of virtualization technologies. Technical report, Stony Brook, NY, 2005.

[38] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *J Internet Serv Appl*, 1(1):7–18, May 2010.

[39] A. DeHon. DPGA utilization and application. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 115–121, 1996.

[40] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 22–28, 1997.

[41] M. Hubner, D. Gohringer, J. Noguera, and J. Becker. Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1–8, 2010.

[42] K. Vipin and S. A. Fahmy. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC)*, pages 13–25, 2012.

[43] K. Vipin and S. A. Fahmy. Automated partitioning for partial reconfiguration design of adaptive systems. In *Proceedings of IEEE International Symposium on Parallel Distributed Processing, Workshops (IPDPSW) – Reconfigurable Architectures Workshop (RAW)*, 2013.

[44] K. Vipin and S. A. Fahmy. A high speed open source controller for FPGA partial reconfiguration. In *Proceedings of International Conference on Field Programmable Technology (FPT)*, pages 61–66, 2012.

[45] G. Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 327–336. Springer-Verlag, 1996.

[46] G. Brebner. The swappable logic unit: a paradigm for virtual hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 77–86, April 1997.

[47] C. Huang and F. Vahid. Transmuting coprocessors: Dynamic loading of FPGA coprocessors. In *Proceedings of the Design Automation Conference*, 2009.

[48] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 223–228, 2005.

[49] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada. A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2010.

[50] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, and H. Takada. Comparison of preemption schemes for partially reconfigurable FPGAs. *IEEE Embedded Systems Letters*, 4(2):45–48, 2012.

[51] J. M. P. Cardoso and Markus Weinhardt. XPP-VC: a C compiler with temporal partitioning for the PACT-XPP architecture. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 864–874. Springer-Verlag, January 2002.

[52] K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, 48(6):579–590, 1999.

[53] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, November 2004.

[54] H. K. H. So, A. Tkachenko, and R. Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 259–264, 2006.

[55] K. Rupnow. Operating system management of reconfigurable hardware computing systems. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 477–478, 2009.

[56] S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig. A 16-nm multiprocessing system-on-chip field-programmable gate array platform. *IEEE Micro*, 36(2):48–62, 2016.

[57] C. Kachris, D. Soudris, G. Gaydadjiev, H. Nguyen, D. S. Nikolopoulos, A. Bilas, N. Morgan, C. Strydis, C. Tsalidis, J. Balafas, R. Jimenez-Peris, and A. Almeida. The VINEYARD approach: Versatile, integrated, accelerator-based, heterogeneous data centres. In *International Symposium on Applied Reconfigurable Computing*, pages 3–13. Springer, 2016.

[58] A. Madhavapeddy and S. Singh. Reconfigurable data processing for clouds. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 141–145. IEEE, 2011.

[59] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[60] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers*, pages 747–755, December 1963.

[61] M. Belwal, M. Purnaprajna, and T. S. B. Sudarshan. Enabling seamless execution on hybrid CPU/FPGA systems: Challenges & directions. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[62] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou. A fully pipelined and dynamically composable architecture of CGRA. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2014.

[63] N. W. Bergmann, S. Shukla, and J. Becker. QUKU: a dual-layer reconfigurable architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 12:63:1–63:26, March 2013.

[64] E. Mirsky and A. DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 157–166, April 1996.

[65] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 126–135. Springer-Verlag, 1996.

[66] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[67] C. Liang and X. Huang. SmartCell: an energy efficient coarse-grained reconfigurable architecture for stream-based applications. *EURASIP Journal on Embedded Systems*, 2009(1):518–659, June 2009.

[68] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*, pages 61–70, January 2003.

[69] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, April 2000.

[70] P. M. Heysters and G. J. M. Smit. Mapping of DSP algorithms on the MONTIUM architecture. In *Parallel and Distributed Processing Symposium*, 2003.

[71] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 191–200, 2009.

[72] R. Polig, H. Giefers, and W. Stechele. A soft-core processor array for relational operators. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 17–24, 2015.

[73] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters*, 3(3):81–84, September 2011.

[74] D. Capalija and T. S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[75] 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 *International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[76] B. Ronak and S. A. Fahmy. Efficient mapping of mathematical expressions into DSP blocks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[77] 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. *SIGARCH Computer Architecture News*, 43(4):28–33, April 2016.

[78] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient Overlay architecture based on DSP blocks. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2015.

[79] A. K. Jain, D. L. Maskell, and S. A. Fahmy. Throughput oriented FPGA overlays using DSP blocks. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2016.

[80] 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 *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2016.

[81] A. K. Jain, D. L. Maskell, and S. A. Fahmy. Are coarse-grained overlays ready for general purpose application acceleration on FPGAs? In *Proceedings of the International Conference on Pervasive Intelligence and Computing*. IEEE, 2016.

[82] X.i Li, A. K. Jain, D. L. Maskell, and S. A. Fahmy. An area-efficient FPGA overlay using DSP block based time-multiplexed functional units. *arXiv preprint arXiv:1606.06460*, 2016.

[83] K. D. Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2013.

[84] 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. *Journal of Signal Processing Systems*, 77(1–2):61–76, Oct. 2014.

[85] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *Computer*, 41(7):27–32, 2008.

[86] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[87] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 225–235, 2000.

[88] R. Laufer, R. R. Taylor, and H. Schmit. PCI-PipeRench and the SwordAPI: a system for stream-based reconfigurable computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 200–208. IEEE, 1999.

[89] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: overcoming pin limitations in FPGA-based logic emulators. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pages 142 –151, April 1993.

[90] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 69–70. IEEE, 2004.

[91] K. Eguro. SIRC: an extensible reconfigurable computing communication API. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 135–138, 2010.

[92] M. Jacobsen and R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, September 2013.

[93] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 97–106, 2011.

[94] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 25–28, 2011.

[95] M. Vuletic, L. Righetti, L. Pozzi, and P. Ienne. Operating system support for interface virtualisation of reconfigurable coprocessors. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 748–749, 2004.

[96] H. Walder and M. Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287, 2003.

[97] X. Changqing, W. Mei, W. Nan, Z. Chunyuan, and H. K. H. So. Extending BORPH for shared memory reconfigurable computers. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 563 –566, August 2012.

[98] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl. ReconOS: an operating system approach for reconfigurable computing. *IEEE Micro*, 2013.

[99] E. Lübbers and M. Platzner. ReconOS: multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8, October 2009.

[100] J. H. Kelm and S. S. Lumetta. HybridOS: runtime support for reconfigurable accelerators. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 212–221, 2008.

[101] X. Iturbe, K. Benkrid, A. T. Erdogan, T. Arslan, M. Azkarate, I. Martinez, and A. Perez. R3TOS: a reliable reconfigurable real-time operating system. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 99–104, 2010.

[102] D. Gohringer, S. Werner, M. Hubner, and J. Becker. RAMPSoCVM: runtime support and hardware virtualization for a runtime adaptive MPSoC. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[103] K. Vipin and S. A. Fahmy. ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq. *IEEE Embedded Systems Letters*, January 2014.

[104] K. Vipin and S. A. Fahmy. Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–8, June 2015.

[105] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pages 642–649, 2001.

[106] Zain-ul-Abdin and B. Svensson. Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. *Microprocessors and Microsystems*, 33(3):161–178, May 2009.

[107] B. D. Sutter, P. Raghavan, and A. Lambrechts. Coarse-grained reconfigurable array architectures. In *Handbook of Signal Processing Systems*, pages 553–592, January 2013.

[108] K. Compton and S. Hauck. Totem: Custom reconfigurable array generation. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119, 2001.

[109] T. Miyamori and K. Olukotun. REMARC: reconfigurable multimedia array coprocessor. *IEICE Transactions on Information and Systems*, 82(2):389–397, 1999.

[110] T. Makimoto and Y. Sakai. Evolution of low power electronics and its future applications. In *International symposium on Low power electronics and design*, pages 2–5. ACM, 2003.

[111] C. H. Ho, P. H. W. Leong, W. Luk, S. J. E. Wilton, and S. Lopez-Buedo. Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 35–44, 2006.

[112] Z. Kwok and S. J. E. Wilton. Register file architecture optimization in a coarse-grained reconfigurable architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 35–44, 2005.

[113] K. Eguro and S. Hauck. Issues and approaches to coarse-grain reconfigurable architecture development. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–120, 2003.

[114] C. Plessl and M. Platzner. Zippy - a coarse-grained reconfigurable array with support for hardware virtualization. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, pages 213–218, 2005.

[115] R. Enzler, C. Plessl, and M. Platzner. System-level performance evaluation of reconfigurable processors. *Microprocessors and Microsystems*, 29(23):63–73, April 2005.

[116] C. Plessl and M. Platzner. Virtualization of hardware - introduction and survey. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 63–69, 2004.

[117] T. Wiersema, A. Bockhorn, and M. Platzner. An architecture and design tool flow for embedding a virtual fpga into a reconfigurable system-on-chip. *Computers & Electrical Engineering*, 2016.

[118] R. Kirchgessner, G. Stitt, A. George, and H. Lam. VirtualRC: a virtual FPGA platform for applications and tools portability. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 205–208, 2012.

[119] M. Jacobsen, Y. Freund, and R. Kastner. RIFFA: a reusable integration framework for FPGA accelerators. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 216–219, May 2012.

[120] S. Shukla, N. W. Bergmann, and J. Becker. QUKU: a coarse grained paradigm for FPGA. In *Proc. Dagstuhl Seminar*, 2006.

[121] R. Lysecky, K. Miller, F. Vahid, and K. Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation (abstract only). In *Proceedings of the International Symposium on Field-programmable gate arrays*, pages 271–271, 2005.

[122] A. Brant and G. G. F. Lemieux. ZUMA: an open FPGA overlay architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, 2012.

[123] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker. A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture. In *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011.

[124] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell. iDEA: A DSP block based FPGA soft processor. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 151–158, 2012.

[125] A. Severance and G. G. F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013.

[126] C. Liu, H. C. Ng, and H. K. H. So. Automatic nested loop acceleration on fpgas using soft cgra overlay. In *Proceedings of the International Workshop on FPGAs for Software Programmers (FSP)*, 2015.

[127] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. F. Lemieux. VEGAS: soft vector processor with scratchpad memory. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 15–24. ACM, 2011.

[128] P. Yiannacouras, J. G. Steffan, and J. Rose. Vespa: Portable, scalable, and flexible fpga-based vector processors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2008.

[129] J. Yu, G. G. F. Lemieux, and C. Eagleston. Vector processing as a soft-core cpu accelerator. In *FPGA*, pages 222–232, 2008.

[130] A. Severance and G. G. F. Lemieux. VENICE: a compact vector processor for fpga applications. In *FPT*, pages 261–268, 2012.

[131] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.

[132] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. Packet switched vs. time multiplexed FPGA overlay networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, 2006.

[133] S. Shukla, N. W. Bergmann, and J. Becker. QUKU: a two-level reconfigurable architecture. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, 2006.

[134] S. Shukla, N. W. Bergmann, and J. Becker. QUKU: a FPGA based flexible coarse grain architecture design paradigm using process networks. In *Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–7, 2007.

[135] D. Capalija and T. S. Abdelrahman. Tile-based bottom-up compilation of custom mesh-of-functional-units FPGA overlays. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014.

[136] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, October 2010.

[137] A. Landy and G. Stitt. A low-overhead interconnect architecture for virtual reconfigurable fabrics. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 111–120, 2012.

[138] J. Coole and G. Stitt. Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts. *IEEE Micro*, 34(1), 2014.

[139] V. Govindaraju, C. H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[140] A. Brant. Coarse and fine grain programmable overlay architectures for FPGAs. Master's thesis, University of British Columbia, 2013.

[141] 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 *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 228–228, 2013.

[142] C. Liu and H. K. H. So. Automatic soft cgra overlay customization for high-productivity nested loop acceleration on fpgas. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 101–101, 2015.

[143] C. Liu, H. C. Ng, and H. K. H. So. QuickDough: a rapid fpga loop accelerator design framework using soft CGRA overlay. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, 2015.

[144] R. Rashid, J. G. Steffan, and V. Betz. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, 2014.

[145] K. Paul, C. Dash, and M.S. Moghaddam. reMORPH: a runtime reconfigurable architecture. In *Euromicro Conference on Digital System Design*, 2012.

[146] M. K. Papamichael and J. C. Hoe. CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 37–46, 2012.

[147] Y. Huan and A. DeHon. FPGA optimized packet-switched NoC using split and merge primitives. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 47–52, 2012.

[148] N. Kapre and J. Gray. Hoplite: building austere overlay NoCs for FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2015.

[149] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched noc for fpga-based systems. *IEE Proceedings-Computers and Digital Techniques*, 153(3):181–188, 2006.

[150] J. Gray. GRVI Phalanx: A massively parallel RISC-V FPGA accelerator accelerator. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–20, 2016.

[151] C. Y. Lin, N. Wong, and H. K. H. So. Operation scheduling for fpga-based reconfigurable computers. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 481–484. IEEE, 2009.

[152] A. Fell, Z. E. Rákossy, and A. Chattopadhyay. Force-directed scheduling for data flow graph mapping on coarse-grained reconfigurable architectures. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2014.

[153] L. Chen and T. Mitra. Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):21, 2014.

[154] R. Rashid. *A Dual-Engine Fetch/Compute Overlay Processor for FPGAs.* PhD thesis, University of Toronto, 2015.

[155] S. J. Jie and N. Kapre. Comparing soft and hard vector processing in fpga-based embedded systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[156] J. Coole and G. Stitt. Adjustable-cost overlays for runtime compilation. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 21–24, 2015.

[157] D. Capalija and T.S. Abdelrahman. Towards synthesis-free JIT compilation to commodity FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 202–205, 2011.

[158] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for fpgas. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 203–213, 2000.

[159] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 1995.

[160] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell. The iDEA DSP block-based soft processor for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):19:1–19:23, 2014.

[161] C. K. HB and N. Kapre. Hoplite-DSP: Harnessing the Xilinx DSP48 multiplexers to efficiently support NoCs on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016.

[162] B. Ronak and S. A. Fahmy. Mapping for maximum performance on FPGA DSP blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):573–585, 2016.

[163] T. Mudge. The specialization trend in computer hardware: Techincal perspective. *Commun. ACM*, 58(4):84–84, March 2015.

[164] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. W. Lee, W. Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, 2002.

[165] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Transactions on Computer Systems (TOCS)*, 25(2):4, 2007.

[166] D. Burger, S. W. Keckler, K. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, 2004.

[167] 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 *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 203–214. IEEE, 2015.

[168] Z. Marzec. Detailed performance evaluation of data-parallel workloads on the dyser prototype system, 2012.

[169] C. H. Hoo and A. Kumar. An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL*, 2012.

[170] K. Heyse, T. Davidson, E. Vansteenkiste, K. Bruneel, and D. Stroobandt. Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAS. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[171] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 213–222, 1997.

[172] K. Eguro and S. Hauck. Armada: timing-driven pipeline-aware routing for FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 169–178. ACM, 2006.

[173] L. N. Pouchet. Polybench: The polyhedral benchmark suite (2011), version 3.2, 2011.

[174] M. Stojilović, D. Novo, L. Saranovac, P. Brisk, and P. Ienne. Selective flexibility: Breaking the rigidity of datapath merging. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1543–1548, 2012.

[175] S. Govindarajan and R. Vemuri. Cone based clustering for list scheduling algorithms. In *Proceedings of the European Design and Test Conference*, pages 456–462, 1997.

[176] S. Govindarajan and R. Vemuri. Improving the schedule quality of static-list time-constrained scheduling. In *Design, Automation and Test in Europe Conference and Exhibition*, page 749, 2000.

[177] 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. In *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2015.

[178] D. P. Singh, T. S. Czajkowski, and A. Ling. Harnessing the power of FPGAs using altera's OpenCL compiler. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 5–6. ACM, 2013.

[179] S. Gao and J. Chritz. Characterization of OpenCL on a scalable FPGA architecture. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2014.

[180] D. Chen and D. Singh. Invited paper: Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 5–12. IEEE, 2012.

[181] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 186–193, May 2011.

[182] 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 *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, August 2012.

[183] K. Shagrithaya, K. Kepa, and P. Athanas. Enabling development of OpenCL applications on FPGA platforms. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, pages 26–30, June 2013.

[184] N. Kavvadias and K. Masselos. Automated synthesis of FSMD-based accelerators for hardware compilation. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, pages 157–160. IEEE, 2012.

[185] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

[186] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.