NANYANG TECHNOLOGICAL UNIVERSITY

ANALYSIS OF COMPUTE KERNELS AND IMPLEMENTATION ON FPGA OVERLAY ARCHITECTURES

by

SINGHAI PRANJUL
(G1402224H)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2015

Contents

List of Figures

List of Tables

Abbreviations

ALAP As Late As Possible

ASAP As Soon As Possible

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

COTS Commercial Off-the-shelf

DFG Data Flow Graph

FPGA Field Programmable Gate Array

FPU Floating Point Unit

FU Functional Unit

GPP General Purpose Processor

HDL Hardware description language

HLS High Level Synthesis

IF Intermediate Fabric

IR Intermediate Representation

MOPS Million Operations Per Second

NN Nearest Neighbor

PL Programmable Logic

PS Processing system

RTL Register Transfer Level

Abstract

Research efforts have shown strength of FPGA accelerators in a wide range of application domains where compute kernels can execute efficiently on an FPGA device. Despite these advantages, FPGAs have not yet been ready for mainstream computing. One reason is that design productivity remains a major challenge, restricting the effective use of FPGA accelerators to niche disciplines involving highly skilled hardware engineers. Coarse-grained FPGA overlay architectures have been shown to be effective when paired with general purpose processors, offering software-like programmability, fast compilation, application portability and improved design productivity. These architectures enable general purpose hardware accelerators, allowing hardware design at a higher level of abstraction, but at the cost of area and performance overheads. This report presents an analysis of compute kernels (extracted from compute-intensive applications) and effect of DSP-aware composition on DFG characteristics. We observe up-to 65% reduction in the number of nodes, up-to 32% reduction in the number of edges and up-to 42% reduction in the graph depth using DSP aware-composition. We perform design space exploration of linear dataflow overlay architectures by modeling programmability cost as a function of overlay design parameters. For the composite graphs, we present experiments to compare the programmability cost of island-style overlay with linear dataflow overlay. We observe upto 77% reduction in cost for sets using ASAP scheduling based approach and upto 81% reduction using proposed approach compared to island-style overlays. Finally, we evaluate the performance of overlay architecture against a set of commercial devices, such as 16-core EPIPHANY device and dual-core ARM cortex-A9, for a set of compute kernels.

Acknowledgment

First and foremost, I would like to thank Assoc Prof Dr Douglas Leslie Maskell for his guidance, enthusiastic support and strong encouragement. I would have not been able to complete my project successfully without his support and directions.

Moreover I would also like to thank Abhishek Kumar Jain for his professional guidance, continuous support, effective suggestions, constructive criticism and timely help. I am also thankful to him for carefully reading and commenting on countless revisions of this report.

Thanks to Mr. Jeremiah Chua in Centre for High Performance Embedded Systems (CHiPES) for his technical support and the facilities.

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.

Chapter 1

Introduction

1.1 Motivation

While the performance benefits of reconfigurable computing over processor based systems have been well established [8, 9, 10], such platforms have not seen wide use beyond specialist application domains such as digital signal processing and communications. Poor design productivity has been a key limiting factor, restricting their effective use to experts in hardware design [11]. Even as High Level Synthesis (HLS) tools improve in efficiency [12, 13], prohibitive compilation time (specifically place and route time) still limits productivity and mainstream adoption of reconfigurable platforms. Coarse-grained FPGA overlay architectures [14, 15, 3, 4, 1, 6, 5, 16] have been shown to be effective when paired with general purpose processors, offering software-like programmability, fast compilation, application portability and improved design productivity. These architectures enable general purpose hardware accelerators, allowing hardware design at a higher level of abstraction, but at the cost of area and performance overheads. In our work, we aim to conduct an analysis of compute kernels (extracted from compute-intensive applications) and effect of DSP-aware composition on Data Flow Graph (DFG) characteristics. We also aim to compare the programmability cost of island-style overlay with linear dataflow overlay by modeling programmability cost of linear dataflow overlay as a function of overlay design parameters.

1.2 Contribution

We first present analysis of compute kernels by extracting DFG using a C to DFG generator and then we present the effect of DSP-aware composition on DFG characteristics. Python graph library have been used for analyzing DFGs and DSP-aware composition. We perform design space exploration of linear data-flow style overlay architectures by modeling programmability cost as a function of overlay design parameters. In this, we first describe an approach for minimizing programmability cost by finding overlay design parameters for a set of compute kernels using sequenced DFGs by As Soon As Possible (ASAP) scheduling. We then present a novel approach for further reduction in programmability cost for the same set of compute kernels using DFGs sequenced by List scheduling algorithm where different DFGs can be scheduled using different resource constraints. Finally we compare the programmability cost of island-style and linear-data flow style overlays for a set of compute kernels. We also present an automated tool flow to provide a mechanism of finding optimal overlay design parameters for a set of compute kernels. We also evaluate the performance of overlay architecture against a set of commercial devices, such as 16-core EPIPHANY device and dual-core ARM cortex-A9 for a set of compute kernels. The main contributions can be summarized as follows:

- The analysis of compute kernels using a data flow graph based approach and the effect of DSP-aware composition on DFG characteristics
- Design space exploration of linear data-flow style overlay architectures by modeling programmability cost as a function of overlay design parameters
- Programmability cost comparison of island-style and linear-data flow style overlays for a set of compute kernels
- An automated tool flow that takes C description of a compute kernels and provide optimal overlay design parameters for reducing programmability cost
- Performance evaluation of overlay architecture against a set of commercial devices

1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information on computer kernels and overlay architectures. Chapter 3 studies current state of the art overlays and techniques for application mapping. In chapter 4, we present the analysis of compute kernels and the effect of DSP-aware composition on DFG characteristics In chapter 5, we present design methodology of linear dataflow overlay architectures for a set of compute kernels and compare the programmability cost with island-style overlays. Chapter 6, presents performance evaluation of overlay architecture against a set of commercial devices, such as 16-core EPIPHANY device and dual-core ARM cortex-A9 for a set of compute kernels. We conclude in chapter 7 and discuss future work.

Chapter 2

Background

2.1 Execution of Compute Kernels

In a typical signal processing application, 20% of the program code consumes 80% of the application execution time. This short section of code generally contains compute intensive arithmetic operations which we refer to as compute kernel. Fig. 2.1 illustrates this concept.

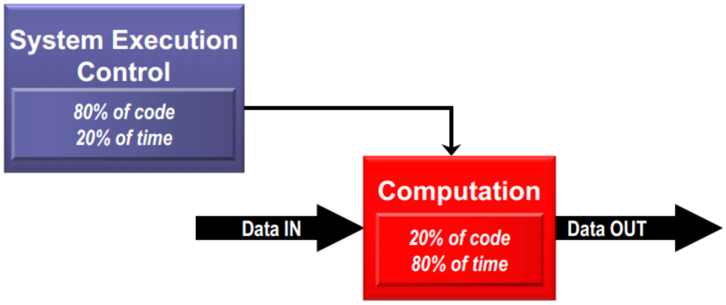


Figure 2.1: Typical Signal Processing Application

A General Purpose Processor (GPP) can be used for the execution of compute kernels by describing their functionality using C or C like programming languages. With the advancements in technology, parallel processing architectures such as multi-cores CPUs and DSPs, GPUs, Massively parallel processor arrays, FPGA based accelerators (as shown in Fig. 2.2) are gaining popularity for accelerated execution of kernels.

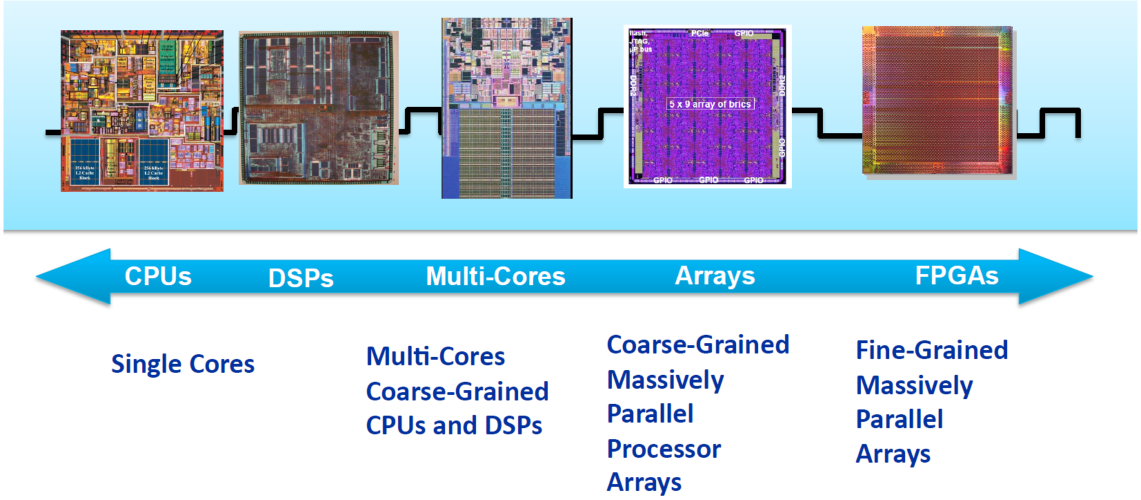


Figure 2.2: Execution Platforms for Compute Kernels

Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge that conventional processor designs will not be able to meet. On the other hand, Field Programmable Gate Array (FPGA) devices provide a substrate for implementing kernels as high performance fully parallel and pipelined designs [17]. Just to provide the clear understanding of the concept, we explain the execution of kernels on general purpose processors and FPGA based specialized accelerators in the following sections.

2.1.1 Execution on General Purpose Processors

As mentioned earlier, a general purpose processor can be used for the execution of compute kernels by describing their functionality using C or C like programming languages. A compiler then generates a list of instructions for the processor to execute sequentially. For example, C description of a compute kernel is shown in Table 2.1 which is then converted to a list of operations (GIMPLE description) using *GCC* compiler. Since the processor executes the list of operations sequentially, the execution time of the kernels increases on increasing the complexity of the kernel. Fig. 2.3 shows the high level view of kernel program execution on a general purpose processor.

Table 2.1: Compute Kernel Code Descriptions

(a) C description

```
#include<math.h>
#define SIZE 1000

#ifdef KERNEL
int kernel(int x){
  int temp = 16*x;
  return (x*(x*(temp*x-20)*x+5));
}
#endif

#ifndef KERNEL
int main(void){
  int i;
  int in[SIZE];
  int out[SIZE];
  for (i=0; i<SIZE; i++){
    out[i] = kernel(in[i]);
  }
  return 0;
}
#endif
```

(b) GIMPLE description

```
kernel (int x)
gimple_bind <
int D.2404;
int D.2405;
int D.2406;
int D.2407;
int D.2408;
int D.2409;
int temp;

gimple_assign <mult_expr, temp, x, 16>
gimple_assign <mult_expr, D.2405, temp, x>
gimple_assign <plus_expr, D.2406, D.2405, -20>
gimple_assign <mult_expr, D.2407, D.2406, x>
gimple_assign <mult_expr, D.2408, D.2407, x>
gimple_assign <plus_expr, D.2409, D.2408, 5>
gimple_assign <mult_expr, D.2404, D.2409, x>
gimple_return <D.2404>
>
```

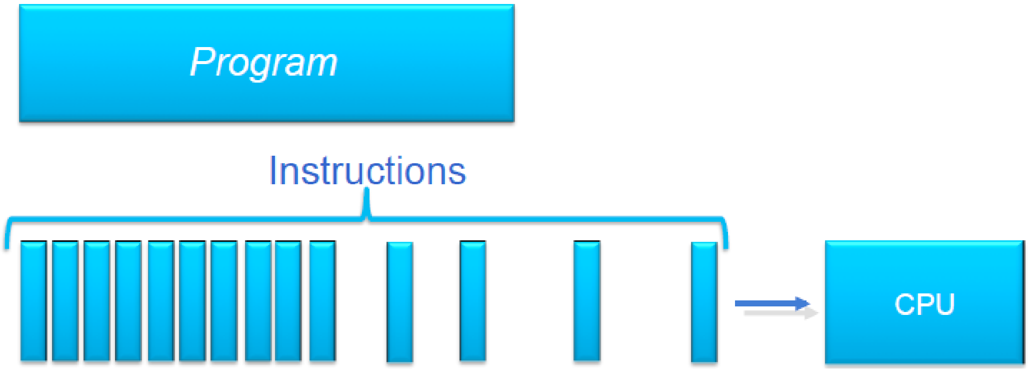


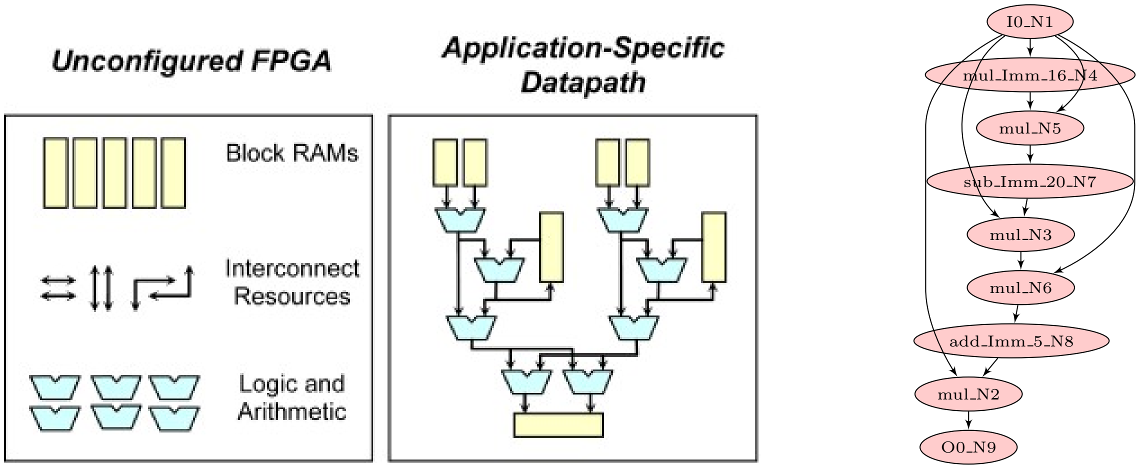
Figure 2.3: Kernel Execution on General Purpose Processor

2.1.2 Execution on FPGA based Specialized Accelerators

Usage of specialized accelerators is becoming significantly important in accelerating compute kernels. These accelerators are normally deployed as an Application Specific Integrated Circuit (ASIC) block alongside a general purpose processor. This limits the flexibility and increases time to market since developing an ASIC is still a complex and time consuming process. On the other hand, FPGAs are becoming popular for rapid-prototyping of accelerators.

For more than a decade, researchers have shown that FPGAs can accelerate a wide variety of software, in some cases by several orders of magnitude compared to state-of-the-art general purpose processors [18, 19]. To understand the execution of kernels on FPGAs, we must first understand how FPGA architectures differ from general purpose processor architectures. The most fundamental difference is that general-purpose processors provide functionality to execute a list of instructions sequentially, whereas FPGA architectures implement compute kernels by providing numerous resources such as configurable logic blocks, DSP blocks for logic and arithmetic and on-chip Block RAMs for storage. These resources are generally interconnected via a programmable island-style routing network which can be programmed to create specialized datapaths as shown in Fig. 2.4(a).

To use an FPGA for accelerating compute kernels, designers typically start by manually converting the compute kernel into an fully pipelined datapath as shown in Fig. 2.4, specified using Hardware description language (HDL). Fig. 2.4(b) shows the datapath of the kernel previously shown in Table 2.1. A fully pipelined datapath on FPGA results in maximum performance by producing output data at every clock cycle. However this performance comes at the cost of designer effort.



(a) Datapath Generation

(b) Datapath of the compute kernel

Figure 2.4: FPGA based Specialized Accelerators

FPGA accelerators are normally designed at a low level of abstraction (typically Register Transfer Level (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. As such, design productivity remains a major challenge, restricting the effective use of FPGA accelerators to niche disciplines involving highly skilled hardware engineers. Designers must specify the entire structure of the data path and must also define control for reading inputs from memories into buffers, stalling the data path when buffers are full or empty, writing outputs to memory, and so on. For a typical FPGA board, a fully pipelined datapath implementation of the several lines of C code may require more than 1,000 lines of HDL code. Such complexity leads to the frequent question: if compilers can extract parallelism from high-level code for multicores and GPUs, why cant they map a compute kernel spatially onto FPGAs in an automated fashion? For more than a decade, researchers have worked toward this goal with HLS tools.

High-level synthesis (HLS) has been proposed as a way of addressing the limited design productivity and manpower capabilities associated with hardware design [12, 13]. Advancements in HLS tools have helped raise the level of programming abstraction from RTL to high level languages, such as C or C++. However, achieving desired performance often still requires detailed low-level design engineering effort that is hard for non-experts. Even as HLS tools improve in efficiency, prohibitive compilation times (specifically the place and route times in the backend flow) still limit productivity and mainstream adoption [11]. Hence, there is a growing need to make FPGAs more accessible to application developers who are accustomed to software API abstractions and fast development cycles [20].

Coarse grained configurable overlay architectures have been proposed as a method to overcome some of these issues [14, 15, 3, 4, 1, 16]. Overlays can be used for reducing the prohibitive compilation time required to map an application to the conventional fine-grained FPGA fabric. Overlays have also been shown to be effective when paired with general purpose processors [2, 15] as this allows the hardware fabric to be viewed as a software-managed hardware task, enabling more shared use. We describe FPGA Overlay architectures in the next section.

2.2 FPGA Overlay Architectures

Overlay architectures consist of a regular arrangement of coarse grained routing and compute resources. The key attraction of overlay architectures is software-like programmability through mapping from high level descriptions, application portability across devices, 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 smaller configuration data size due to the coarse granularity. Accelerators can be described at a higher level of abstraction and compiling it for overlays is several orders of magnitude faster than for the fine grained FPGAs. Researchers have proposed fine [21], [22] and coarse grained [14], [15], [3], [4], [16], [23] overlay architectures to abstract FPGA fabric resources.

2.2.1 Architecture

As shown in Fig. 2.5, coarse grained FPGA overlay architecture is a two-dimensional array of reconfigurable tiles, implemented on top of a commercial FPGA device. Coarse grained tiles contains programmable processing elements (PEs) which are

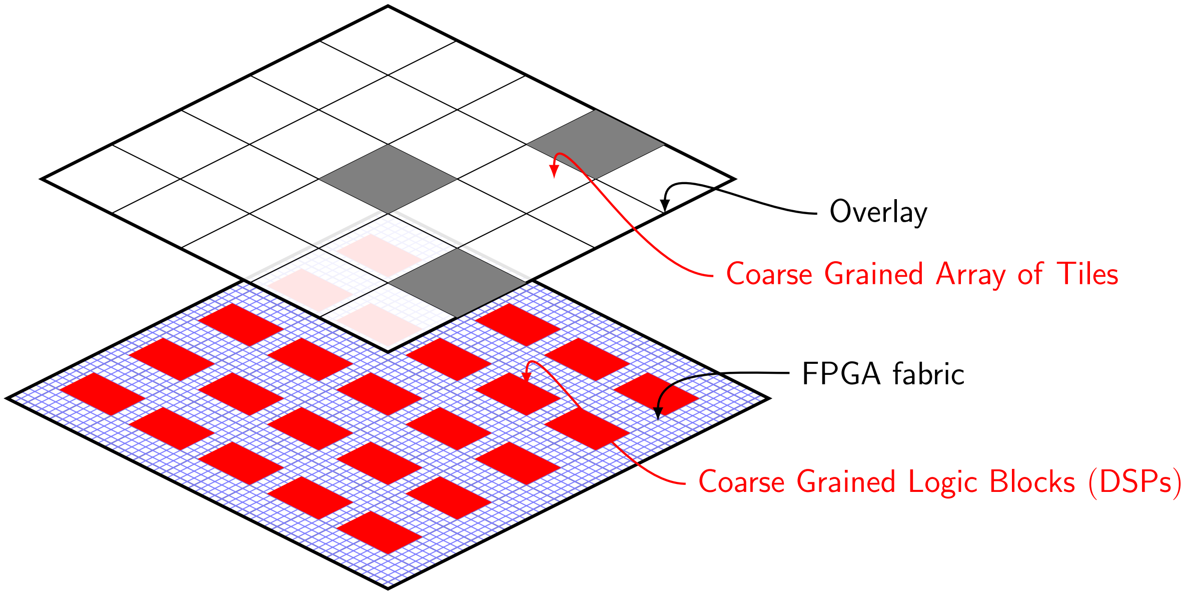

Overlay
Coarse Grained Array of Tiles
FPGA fabric
Coarse Grained Logic Blocks (DSPs)

Figure 2.5: FPGA Overlay Architecture

interconnected using programmable interconnect (PI) and the functions of the PE and the PI are controlled by configuration data. The overlay overcomes the need for a full cycle through the vendor implementation tools, instead presenting a much simpler problem of programming an interconnected array of processing elements.

The possible configuration space and reconfiguration data size is much smaller than for direct FPGA implementation of kernels because of the coarser granularity of the overlay. An overlay provides a leaner mechanism for hardware task management at runtime as there is no need to prepare distinct bitstreams in advance using vendor-specific compilation (synthesis, map, place and route) tools. Instead, the behaviour of the overlay can be modified using software defined overlay configurations.

2.2.2 Host Processor Interface

Despite having the implementation of the overlay architecture and its performance gain, there is no guarantee that it will surely provide reduction in kernel execution time. It depends heavily on how the overlay is interfaced to the host processor, communication mechanism between overlay, host processor and the external memory, communication bandwidth and latencies etc. Researchers have shown the effective use of coarse grained overlay architectures by pairing them with host processors as a coprocessor [24, 2] or as a part of the processor's pipeline [25]. Fig. 2.6 shows the integration of DySER [25, 26] overlay into the pipeline of a processor.

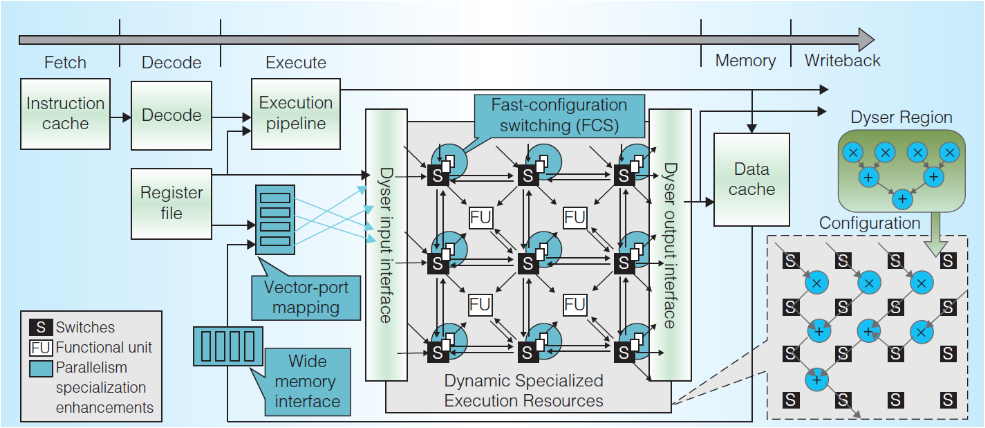


Figure 2.6: DySER Interfacing with Host Processor [1]

Integrating an overlay within a processor pipeline can provide huge performance and energy efficiency at the expense of complete redesign of processor micro-architecture. Another possible approach is to interface the overlay (as a co-processor) with the host processor via standard communication interfaces. To address possible bottleneck problems, particularly in providing high bandwidth transfers between the host procesor and the co-processor implemented on the FPGA fabric [27], it has been proposed to more tightly integrate the processor and the FPGA fabric. A number of tightly coupled architectures have resulted [28, 29], including vendor specific systems with integrated hard processors. One example of pairing the overlay (Intermediate Fabric (IF) Overlay [3]) with a high performance ARM processor via an Advanced eXtensible Interface (AXI) interface in a commercial computing platform (the Xilinx Zynq[30]) is shown in Fig. 2.7. Zynq platform partition the hardware into a Processing system (PS), containing one or more processors along with peripherals, bus and memory interfaces, and other infrastructure, and the Programmable Logic (PL) where custom hardware can be implemented. The two parts are coupled together with high throughput interconnect to maximize communication bandwidth. The Xilinx-Zynq consists of a dual-core ARM Cortex A9 processor equipped with a double-precision Floating Point Unit (FPU), commonly used peripherals and reconfigurable fabric.

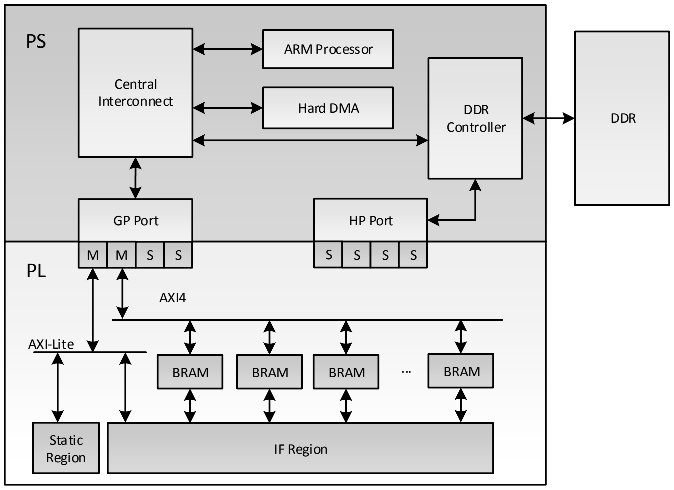


Figure 2.7: Intermediate Fabric (IF) Interfacing with Host Processor [2]

2.2.3 Runtime Management

When paired as a coprocessor, run-time management, including overlay configuration loading, data communication, can be carried out using an operating system (Linux) [24] and also using a commercial hypervisor [31]. Firstly, user needs to identify a kernel, as described in Table 2.2, to be implemented on top of overlay. Then DFG can be extracted after compiling this code using compiler front-end. After that a place and route tool can be used to map the DFG on top of overlay. After generating configurations based on the placement and routing, kernel code can be transformed in the code containing overlay APIs as shown in Table 2.2.

Table 2.2: Source Code Transformation

(a) Original C description

```c
#include<math.h>
#define SIZE 1000

#ifdef KERNEL
int kernel(int x)
{
    int temp = 16*x;
    return (x*(x*(temp*x-20)*x+5));
}
#endif

#ifndef KERNEL
int main(void)
{
    int i;
    int in[SIZE];
    int out[SIZE];
    for (i=0; i<SIZE; i++){
        out[i] = kernel(in[i]);
    }
    return 0;
}
#endif
```

(b) Modified C description

```c
#include <overlay.h>
#include<math.h>
#define SIZE 1000

void kernel(int *a, int *b, int length){
    //allocate BRAM as overlay memory
    memory_a = overlay_malloc(size_a);
    memory_b = overlay_malloc(size_b);
    //load overlay configuration for the task
    load_configuration();
    //copy inputs
    overlay_transfer_data(a, memory_a , size_a);
    //Trigger overlay to process data
    overlay_process();
    //copy_outputs
    overlay_transfer_data(memory_b , b, size_b);
}
int main(void){
    int in[SIZE];
    int out[SIZE];
    kernel(in, out , SIZE);
    return 0;
}
```

The working of modified C description is pretty straight-forward as shown in code listing 2.2. First it allocates input and output BRAMs as overlay memory. Then it loads the overlay configuration for the task. After that it transfers the input data to input BRAM and triggers the overlay. Finally, the Overlay starts processing the input data in a streaming fashion and transfer the processed data to output BRAMs.

2.3 Kernel Compilation on Overlays

Programming for Overlays can be done at a higher level (normally C/C++ or DFG level) and the high level description of kernel can be compiled onto overlay using an automated mapping tool-flow. There are normally two main steps associated with kernel compilation to overlays. Firstly, a compiler front-end extracts the DFG out of a high level description of the application [28],[32]. After that the DFG gets mapped to the overlay after going through a process of operation scheduling and placing and routing [33],[14],[4]. The work in this report uses HercuLeS front-end compiler for DFG generation and Python-graph library for further analysis on DFGs. We describe these here as follows.

2.3.1 HercuLeS Front-end Compiler

HercuLeS is a high-level synthesis tool that automatically transforms C description to RTL implementation of compute kernels. It translates *GIMPLE* (Intermediate Representation (IR) of *GCC*) to *NAC*. It has two main components: a frontend (*nac2cdfg*) and a backend (*cdfg2hdl*). We use front-end (shown in Fig. 2.8)to generate DFGs of compute kernel. C description is passed to *GCC* for *GIMPLE* dump generation, which is then processed by *gimple2nac* and finally *nac2cdfg* converts it to data flow graph (dot file format). We process the data flow graph using Python-graph Library.

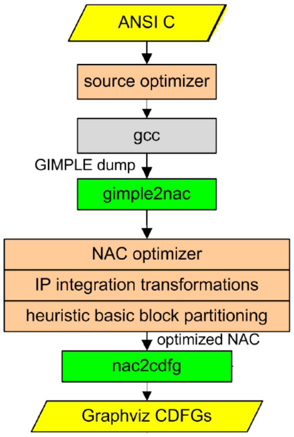
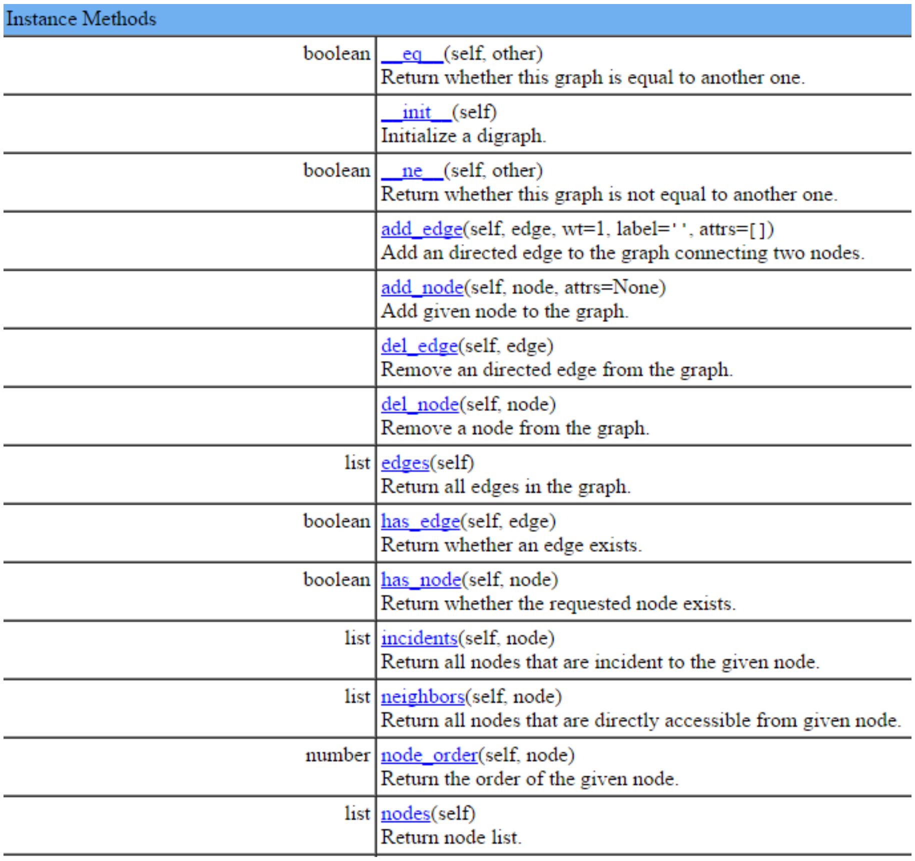


Figure 2.8: Front-end Compilation of Kernels

2.3.2 Python-graph Library

Python-graph is a library (containing a set of APIs) for working with graphs in Python. It provides suitable data structure for representing graphs and an implementation of important algorithms. It is a pretty useful development environment for developers interested in exploring graph algorithms for data flow graph (DFG) analysis. The most important feature which we have used is *digraph* class and the APIs are listed in Fig. 2.9.



Instance Methods	
boolean	__eq__(self, other) Return whether this graph is equal to another one.
	__init__(self) Initialize a digraph.
boolean	__ne__(self, other) Return whether this graph is not equal to another one.
	add_edge(self, edge, wt=1, label='', attrs=[]) Add an directed edge to the graph connecting two nodes.
	add_node(self, node, attrs=None) Add given node to the graph.
	del_edge(self, edge) Remove an directed edge from the graph.
	del_node(self, node) Remove a node from the graph.
list	edges(self) Return all edges in the graph.
boolean	has_edge(self, edge) Return whether an edge exists.
boolean	has_node(self, node) Return whether the requested node exists.
list	incidents(self, node) Return all nodes that are incident to the given node.
list	neighbors(self, node) Return all nodes that are directly accessible from given node.
number	node_order(self, node) Return the order of the given node.
list	nodes(self) Return node list.

Figure 2.9: Useful APIs for DFG processing

Using the set of APIs provided by this library, we developed a set of python modules containing implementation of frequently used graph scheduling algorithms. We implemented ASAP, As Late As Possible (ALAP) and List Scheduling algorithms, described later in this report.

Chapter 3

Literature Survey on FPGA Overlays

In the area of coarse grain overlay architectures, the compute the routing logic can either perform the same operation over the time, or can loop over a short list of instructions or can execute a fully fledged instruction stream. Based on this variety, researchers have proposed both spatially and temporally programmed overlays that are mapped to the fine grained fabrics of modern FPGAs. In spatially programmed overlays, the compute logic and routing of the overlay are unchanged while a compute kernel is executing while in temporally programmed overlays, the compute logic and routing of the overlay change on a cycle by cycle basis while a compute kernel is executing [16, 34, 35]. In this report, we focus on spatially programmed overlays.

Spatially programmed overlays normally have a single instruction register within each Functional Unit (FU) and hence FU behaves like a data flow processing element. An array of such a data flow processing element, interconnected via a island style or Nearest Neighbor (NN) style programmable interconnect, can be considered as a Spatially programmed overlay architecture. This type of 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 now 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 throughput under this mapping would be one kernel iteration per cycle since the initiation interval would be one. 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.

3.1 Intermediate Fabrics

An overlay architecture, referred to as an intermediate fabric (IF) [3], [36] was proposed to support near-instantaneous placement and routing. Standard VPR [37] algorithms were used for placement and routing of compute kernels. It consists of 192 heterogeneous functional units comprising 64 multipliers, 64 subtracters, 63 adders, one square root unit, and five delay elements with a 16-bit datapath and supported the fully parallel, pipelined implementation of compute kernels.

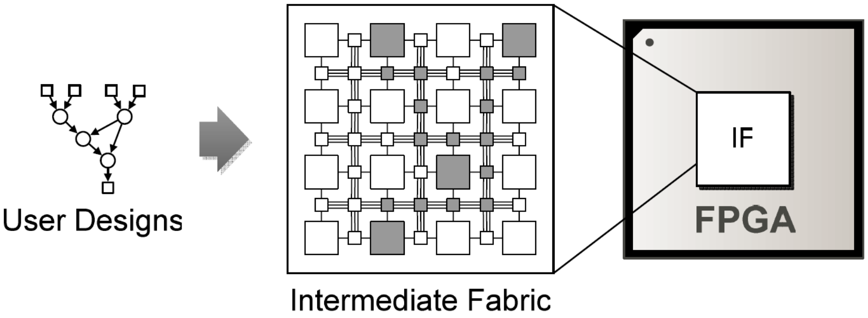


Figure 3.1: Intermediate Fabrics as Island-style Overlay [3].

Unlike a physical device, whose architecture must support many applications, IFs have been specialized for particular domains or even individual applications. Such specialization hides the complexity of fine-grained Commercial Off-the-shelf (COTS) devices, thus enabling fast place and route (700x speedup over vendor tools) at the cost of significant area (34% - 44%) and performance (7%) overhead when implemented on an Altera Stratix III FPGA [36]. However, the IF only achieved an F_{max}

of 125 MHz resulting in low throughput for the application benchmarks tested. Area overhead comes into picture mainly because of virtual interconnect logic which comprised of multiplexers based routing. This overhead was reduced by 48% - 54% by reducing flexibility of routing in [38], while improving speed by 24% with a modest routability overhead of 16%. Based on the above mentioned work on IFs, an end-to-end tool flow was presented for FPGA-accelerated scientific computing [39].

3.2 Mesh of FU based Overlay

This overlay executes a given DFG by mapping the graph nodes to the FUs and by configuring the routing logic to establish inter-FU connections that reflect the graph edges [4]. Multiple instances of the DFGs are then executed in a pipelined fashion on the overlay to achieve high performance. In addition to integer arithmetic, overlay also used floating point processing elements. It consisted of a 24×16 overlay with a nearest-neighbor-connected mesh of 214 routing cells and 170 heterogeneous functional units (FU) comprising 51 multipliers, 103 adders and 16 shift units. When implemented on an Altera Stratix IV FPGA, the overlay consumed 75% of the total device ALMs, with the routing network consuming 90% of the ALM resource used. An F_{max} of 355 MHz and a peak throughput of 60 GOPS was reported. A placer and router was also developed by customizing VPlace [40] and PathFinder [41], respectively.

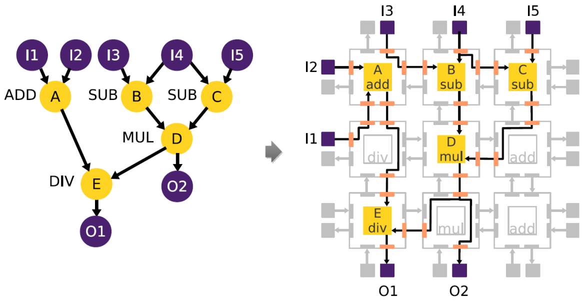


Figure 3.2: Nearest-neighbor connected Mesh of Functional units [4].

3.3 DySER Architecture

DySER [1, 26] was proposed as a coarse grained overlay architecture for improving the performance of general purpose processors. It was originally designed as a heterogeneous array of 64 functional units interconnected with a circuit-switched mesh network and implemented on ASIC. The DySER architecture was then improved and prototyped, along with the OpenSPARC T1 RTL, on a Xilinx XC5VLX110T FPGA [25]. However, due to excessive LUT consumption, it was only possible to fit a 2x2 32-bit DySER, a 4x4 8-bit DySER or an 8x8 2-bit DySER on the FPGA. An adapted version of a 6x6 16-bit DySER was implemented on a Xilinx Zynq-7020 [5]. The larger DySER array was achieved by using a DSP block as the compute logic, thus better targeting the architecture to the FPGA.

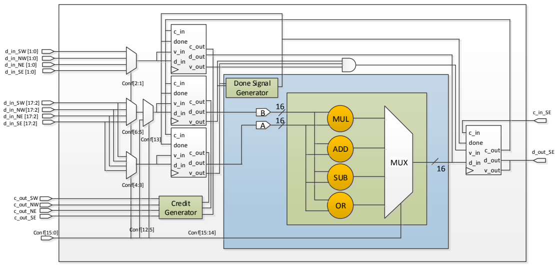
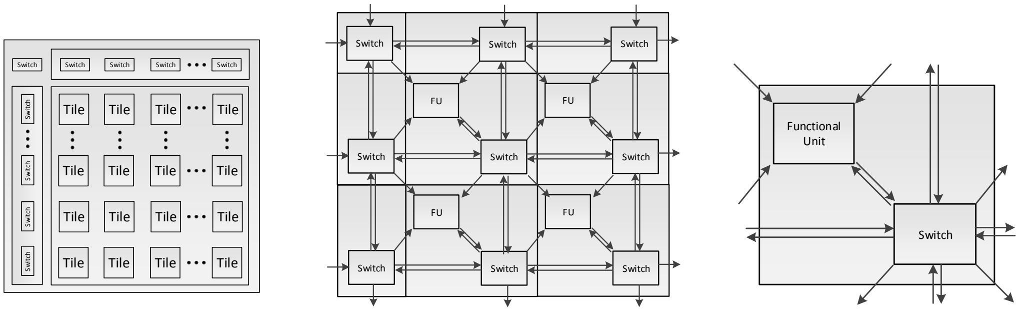


Figure 3.3: DySER functional unit [5].



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

Figure 3.4: DySER architecture as Island-style overlay.

3.4 DSP Block based Overlay Architecture

An overlay architecture with the FU based on the DSP blocks found in Xilinx FPGAs was recently proposed [6]. This overlay combines multiple operations in a compute kernel and maps them to the DSP block, resulting in a significant reduction in the number of processing nodes required. An F_{max} of 370 MHz with throughputs better than that achieved by directly implementing the benchmarks onto the fabric using Xilinx Vivado HLS were reported.

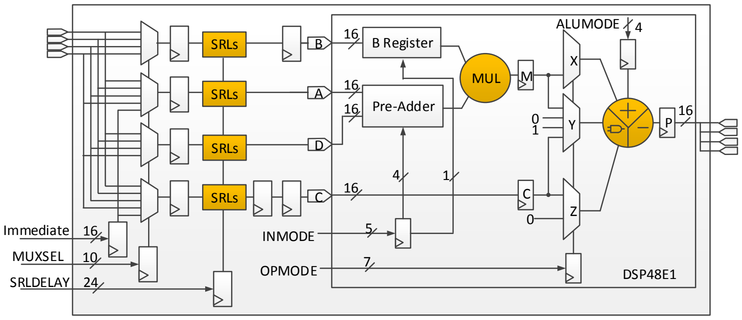
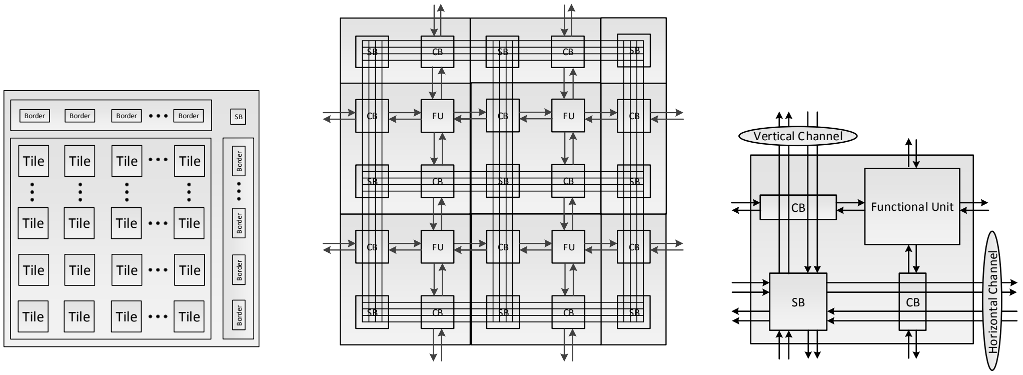


Figure 3.5: DSP block based functional unit [6].



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

Figure 3.6: DSP Block based architecture as Island-style overlay.

3.5 VDR Overlay

The VDR overlay [7] was proposed as an array of coarse-grained heterogeneous PEs interconnected by a set of programmable switches. The VDR overlay F_{max} was 172 MHz on an Altera Stratix III FPGA and achieved a 9× improvement in performance over a NIOS II soft processor. Primary objective was not only to reduce the compilation time, but also to improve the performance of soft processors.

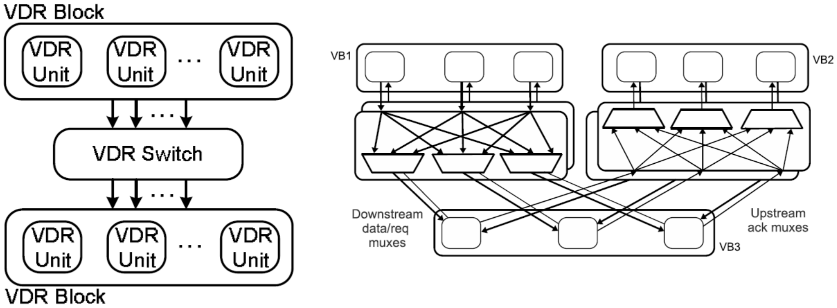


Figure 3.7: VDR Overlay as Linear Dataflow Machine [7].

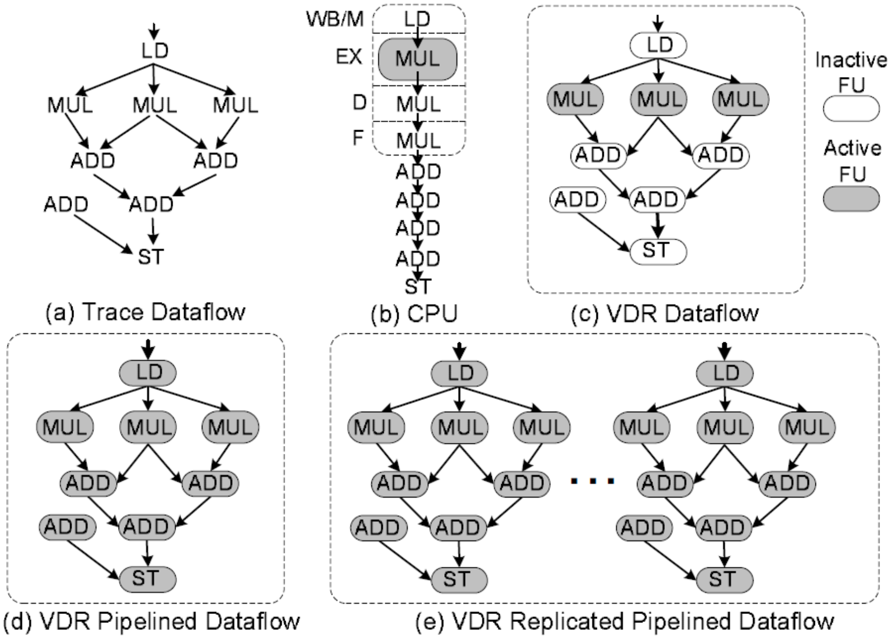


Figure 3.8: DFG execution on VDR Overlay [7].

Chapter 4

Analysis of Compute Kernels

In this chapter, we give the detailed analysis of compute kernels extracted from compute-intensive applications. We have compiled a benchmark set (shown in Table 4.2) containing a number of compute kernels from [3], [42], [43] and [44]. The idea is to find out the characteristics of kernels by extracting the data flow graphs (DFGs) using a C to DFG generator and a DFG analyzer. We have used frontend of HercuLeS HLS tool as C to DFG generator and a set of python modules for generating the characteristics. The DFG consists of nodes that represent operations and edges that represent the flow of data between operations. We perform DFG transformation from graph of operations to graph of composite operations where each composite operation can be mapped to a DSP block. Finally we generate and compare the characteristics of DFGs and DSP-aware DFGs using a set of python modules. The characteristics include number of arithmetic operations, number of edges, number of I/O interfaces, graph depth, graph width and average parallelism etc. The complexity of the graph can be reduced by composition since the it results in reducing the number of nodes and edges. The DSP-aware DFG can be used for efficient mapping on spatially programmed DSP block based overlays.

Table 4.1: Compute Kernel Code Descriptions

(a) C description

```c
#include<math.h>
#define SIZE 1000

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

int main(void){
  int i;
  int in[SIZE];
  int out[SIZE];
  for (i=0; i<SIZE; i++){
    out[i] = kernel(in[i]);
  }
  return 0;
}
```

(b) DFG description

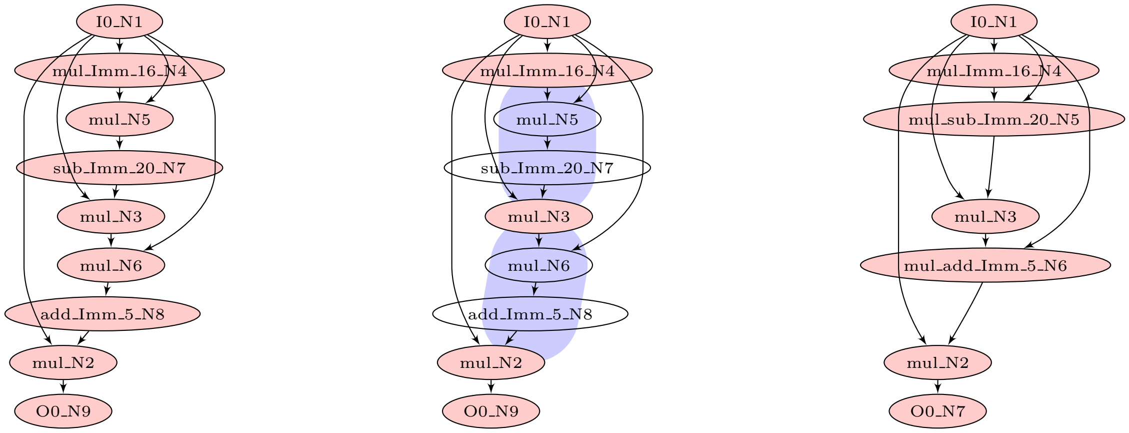
```
digraph kernel {
N8 [ntype="operation", label="add_Imm_5_N8"];
N9 [ntype="outvar", label="O0_N9"];
N1 [ntype="invar", label="I0_N1"];
N2 [ntype="operation", label="mul_N2"];
N3 [ntype="operation", label="mul_N3"];
N4 [ntype="operation", label="mul_Imm_16_N4"];
N5 [ntype="operation", label="mul_N5"];
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;
}
```

4.1 C to DFG Generation

We have used frontend of HercuLeS HLS tool as C to DFG generator. C description of compute kernels are taken from https://bitbucket.org/abhishekntu/kernelbench. DFG generator uses gcc compiler for generating gimple representation of kernel and then uses gimple2nac for generating a kind of intermediate representation and finally uses nac2cdfg for generating the DFG. For example, starting with a C description of the compute kernel, DFG generator transforms this to a DFG description, as shown in Table 4.1. Fig. 5.3(a) shows the nodes and edges in an example DFG.

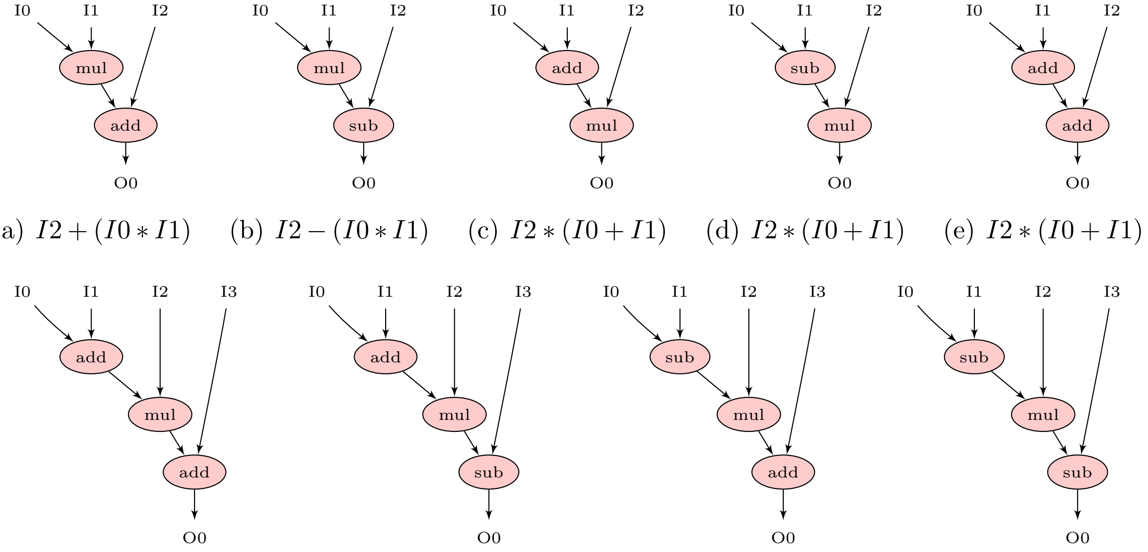
4.2 DSP-aware DFG Generation

The operations in the DFG can be executed on a arithmetic operator or on a functional unit having a combination of arithmetic operators. One such example is Xilinx DSP block which offer area, performance, and power advantages over the equivalent function implemented directly in the logic fabric. Since the DSP block consists of an



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

Figure 4.1: DFG and composite DFG generation.



(a) $I2 + (I0 * I1)$ (b) $I2 - (I0 * I1)$ (c) $I2 * (I0 + I1)$ (d) $I2 * (I0 + I1)$ (e) $I2 * (I0 + I1)$

(f) $I3 + (I0 + I1) * I2$ (g) $I3 - (I0 + I1) * I2$ (h) $I3 + (I0 - I1) * I2$ (i) $I3 - (I0 - I1) * I2$

Figure 4.2: Examples of DFG segments mappable to DSP block

add/sub module, a multiplier and an ALU and can be used to consolidate up-to three operation, we explore the feasibility of mapping the combination of operations onto DSP blocks. We find the composite operations in the DFG as shown in Fig. 5.3(c) and replace them with a single node as shown in Fig. 5.3(e). We refer to the final graph as DSP-aware DFG. For example, we can use multiply-subtract and multiply-add to collapse N5-N7 and N6-N8 in Fig. 5.3(a) into N5 and N6 of Fig. 5.3(e), respectively. As a result, DSP-aware DFG contains only 5 nodes instead of the 7 nodes. Fig. 4.2 shows the composite operations supported by DSP block.

Table 4.2: DFG characteristics of Benchmark set

	Benchmark	Characteristics					
No.	Name	i/o	graph	op	graph	average	graph
		nodes	edges	nodes	depth	parallelism	width
1.	chebyshev	1/1	12	7	7	1.00	1
2.	sgfilter	2/1	27	18	9	2.00	4
3.	mibench	3/1	22	13	6	2.16	3
4.	qspline	7/1	50	26	8	3.25	7
5.	poly1	2/1	15	9	4	2.25	4
6.	poly2	2/1	14	9	5	1.80	3
7.	poly3	6/1	17	11	5	2.20	4
8.	poly4	5/1	13	6	4	1.50	2
9.	poly5	3/1	43	27	9	3.00	6
10.	poly6	3/1	72	44	11	4.00	11
11.	poly7	3/1	62	39	13	3.00	10
12.	poly8	3/1	51	32	11	2.90	8
13.	fft	6/4	24	10	3	3.33	4
14.	kmeans	16/1	39	23	9	2.55	8
15.	mm	16/1	31	15	8	1.88	8
16.	mri	11/2	24	11	6	1.83	4
17.	spmv	16/2	30	14	4	3.50	8
18.	stencil	15/2	30	14	5	2.80	6
19.	conv	24/8	40	16	2	8.00	8
20.	radar	10/2	18	8	3	2.66	4
21.	arf	26/2	58	28	8	3.50	8
22.	ewf	21/5	73	34	14	2.43	4
23.	fir2	17/1	47	23	9	2.55	8
24.	hornerbezier	12/4	32	14	4	3.50	5
25.	motionvenctor	25/4	52	24	4	6.00	12
26.	smoothtriangle	29/14	88	37	6	6.16	18

4.3 Characteristics of DFGs and DSP-aware DFGs

We use a set of python modules to process DFG and DSP-aware DFG to find out the number of operation nodes, I/O nodes, edges, graph depth and width, average parallelism etc. Table 4.2 and 4.3 shows the characteristics for DFGs and DSP-aware DFGs respectively. Graph depth is the critical path length of the graph and graph width is the maximum number of nodes that can execute concurrently. Average parallelism is the ratio of total number of op nodes and the graph depth.

Table 4.3: DSP-aware DFG characteristics of Benchmark set

	Benchmark	Characteristics					
No.	Name	i/o	graph	DSP	graph	average	graph
		nodes	edges	nodes	depth	parallelism	width
1.	chebyshev	1/1	10	5	5	1.00	1
2.	sgfilter	2/1	19	10	5	2.00	3
3.	mibench	3/1	14	6	4	1.50	3
4.	qspline	7/1	46	22	7	3.14	7
5.	poly1	2/1	12	6	3	2.00	4
6.	poly2	2/1	10	6	3	2.00	3
7.	poly3	6/1	13	7	3	2.30	4
8.	poly4	5/1	9	3	2	1.50	2
9.	poly5	3/1	28	14	6	2.3	6
10.	poly6	3/1	51	25	9	2.77	10
11.	poly7	3/1	44	21	8	2.62	7
12.	poly8	3/1	35	17	5	3.40	8
13.	fft	6/4	22	8	3	2.66	4
14.	kmeans	16/1	36	20	7	2.85	8
15.	mm	16/1	24	8	8	1.00	1
16.	mri	11/2	20	7	5	1.40	2
17.	spmv	16/2	24	8	4	2.00	2
18.	stencil	15/2	24	8	3	2.66	4
19.	conv	24/8	32	8	1	8.00	8
20.	radar	10/2	16	6	3	2.00	2
21.	arf	26/2	50	20	8	2.50	4
22.	ewf	21/5	56	18	8	2.25	5
23.	fir2	17/1	32	8	8	1.00	1
24.	hornerbezier	12/4	22	8	3	2.66	4
25.	motionvector	25/4	40	12	3	4.00	4
26.	smoothtriangle	29/14	76	25	5	5.00	14

We observe that average parallelism varies from 1 to 8 for DFGs and DSP-aware DFGs in benchmark set-I. Benchmark set-I DFGs contains up-to 44 op nodes and 88 edges while DSP-aware DFGs contains up-to 25 op nodes and 76 edges. Benchmark set-I DFGs exhibits a depth of up-to 14 and a width of up-to 18 while DSP-aware DFGs exhibits a depth of up-to 9 and a width of up-to 14. Number of I/Os remains same for DFGs and DSP-aware DFGs.

4.4 Effect of composition

The purpose of composition is to reduce the complexity of the graph so that it can be mapped to a light weight overlay architecture with low programmability overhead. Fig. 4.3 shows the effect of composition on the number of nodes, edges and also on the average parallelism for each benchmark.

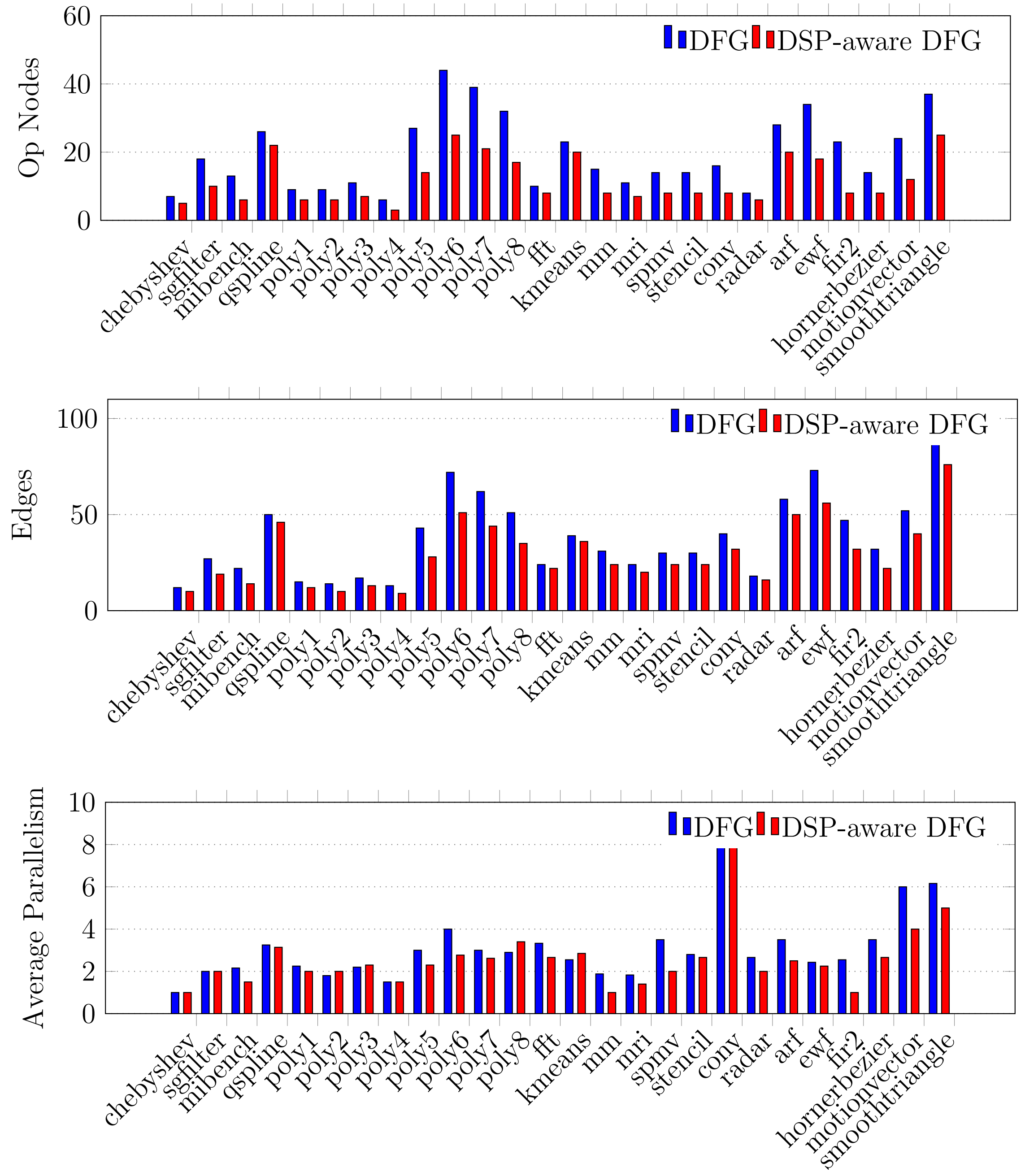


Figure 4.3: Effect on kernel characteristics using composite operations

We observe up-to 65% reduction in the number of nodes, up-to 32% reduction in the number of edges and up-to 42% reduction in the graph depth. It is clear from the results that the complexity of the graph can be reduced drastically using composite operations. We use DSP-aware DFGs in the next chapter to implement on top of overlay architectures.

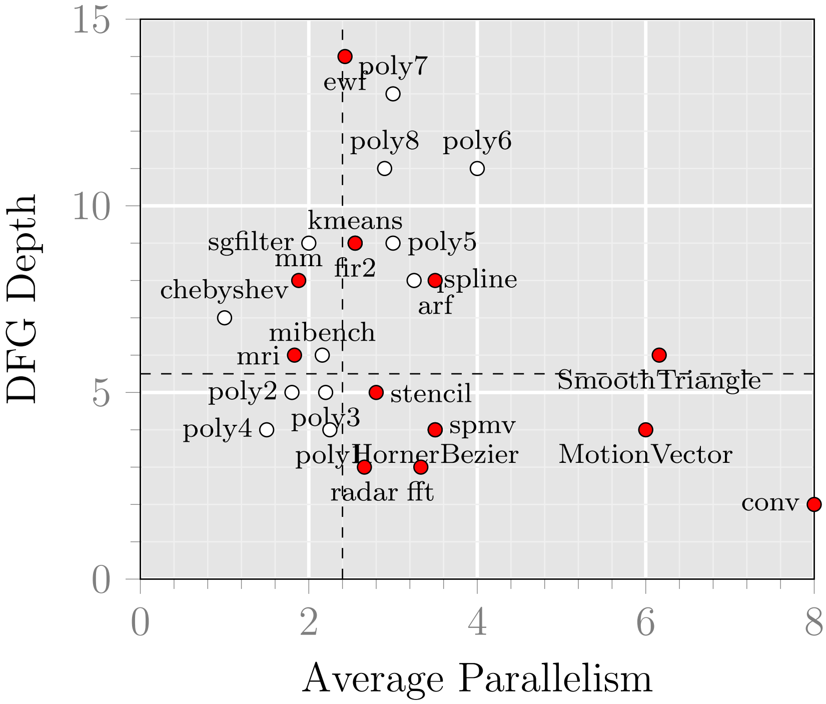


Figure 4.4: Distribution of Compute Kernels

Fig. 4.4 shows the distribution of kernels by plotting the depth and average parallelism of kernel on a two dimensional space. The kernels on bottom-left quadrant shows small depth and parallelism and we can assume that a low cost overlay can be used to support these kernels. However, the kernels on top right quadrant shows large depth and parallelism and we can assume that an overlay of high cost would be required for supporting them. We conduct experiments in the next chapter to validate our assumptions.

Chapter 5

Implementation on FPGA Overlays

In this chapter, we present the implementation methodology of compute kernels on FPGA overlay architectures. We map the compute kernels on two types of FPGA overlay architectures, linear dataflow overlay and island-style overlay. We make use of python-graph library to develop a set of tools for finding overlay design parameters for a set of compute kernels. We focus on efficient and cost-effective implementation of overlay architectures by performing a detailed analysis of kernels using proposed tools. The detailed discussion is shown in the following sections.

5.1 Linear Data Flow Overlay

As discussed in the previous section, DSP-aware DFGs contains DSP block as nodes which can be mapped spatially on a programmable architecture designed using DSP block as a functional unit. The array of functional units (DSP blocks) can be interconnected using different kind of programmable interconnect architectures such as island-style or linear dataflow-style (LDF-style). We use LDF-style for interconnecting functional units as shown in the Fig. 5.1. This LDF-style overlay facilitates an efficient and cost effective implementation of compute kernels. We calculate the programmability cost of LDF-style overlay for implementing a set of benchmarks and compare it with the cost of island-style overlay.

5.1.1 Architecture

As shown in Fig. 5.1, LDF-style overlay is a linear array of programmable tiles where each tile contains a programmable routing network and a cluster of DSP blocks and delay lines (DLs). Delay lines are required in each cluster for bypassing inputs of the tile to the outputs by delaying them equivalent to the latency of DSP blocks. The number of DSP blocks, delay lines and the complexity of routing network in each tile can be customized according to the set of benchmarks. Hence customized overlays can be designed with a low programmability cost for a set of benchmarks.

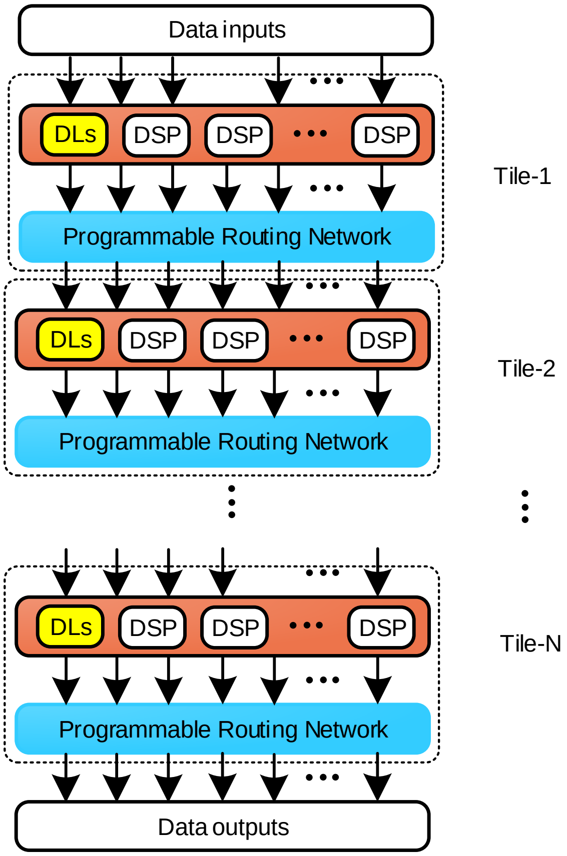

Data inputs
DLs
DSP
DSP
DSP
Programmable Routing Network
Tile-1
DLs
DSP
DSP
DSP
Programmable Routing Network
Tile-2
DLs
DSP
DSP
DSP
Programmable Routing Network
Tile-N
Data outputs

Figure 5.1: Block diagram of linear dataflow overlay.

5.1.2 Programmability-cost Modelling

For each tile, number of DSP blocks, delay lines and the routing network complexity can be decided based on a set of benchmarks. The complexity of routing network in n^{th} tile depends on the resources (number of DSP blocks and delay lines) of n^{th} tile and $(n+1)^{th}$ tile. If X_n and Y_n are the number of DSP blocks and delay lines in n^{th} tile, The routing network can be designed using $(X_n + Y_n)$ x $(4*X_{n+1} + Y_{n+1})$ crossbar switch, 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. If L_n is the number of LUTs/bit to design $(X_n + Y_n)$:1 multiplexer, the programmability cost of the overlay network would be equal to $\sum_{n=1}^{N-1}(L_n * (4 * X_{n+1} + Y_{n+1}))$ LUTs/bit, where N is the number of tiles in the overlay. If the overlay needs to support immediate data for operations as well, the DSP inputs needs to support an additional muxing input and the network complexity can be calculated as a $(X_n + Y_n + 1)$ x $(4*X_{n+1})$ crossbar switch and a $(X_n + Y_n)$ x (Y_{n+1}) crossbar switch. Assuming worst case scenario which is 1/2 LUT/bit increase on increasing additional muxing input, the $(X_n + Y_n + 1)$:1 multiplexer would consume $(L_n + 1/2)$ LUTs/bit and hence the programmability cost of the overlay network would be equal to $\sum_{n=1}^{N-1}(L_n * (4 * X_{n+1} + Y_{n+1}) + 2 * X_{n+1})$ LUTs/bit. Fig. 5.2 shows the LUT requirement on scaling the size of multiplexers.

Finding overlay design parameters (N, X_n and Y_n) for a set of graphs: Given a set G of M graphs, each graph can be scheduled using well known scheduling techniques (ASAP, ALAP, LIST etc.) to generate sequenced graph which can be used to find out the number of nodes and crossing edges in each stage. We refer to the number of nodes in n^{th} stage as $g_m x_n$, total stages as Ng_m and crossing edges as $g_m y_n$ in m^{th} graph G_m. X_n and Y_n can be found out using equation 2.1 and 2.2.

$$X_n = \max(g_1 x_n, g_2 x_n, g_3 x_n, ...g_M x_n) \tag{5.1}$$

$$Y_n = \max(g_1 y_n, g_2 y_n, g_3 y_n, ...g_M y_n) \tag{5.2}$$

$$N = \max(Ng_1, Ng_2, Ng_3, ...Ng_M) \tag{5.3}$$

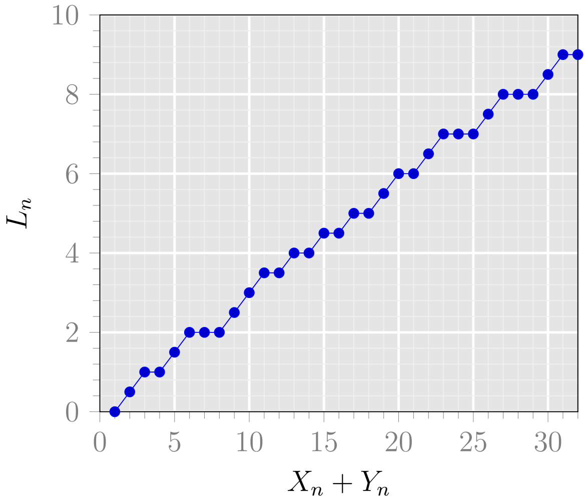


Figure 5.2: LUTs/bit for $X_n + Y_n$:1 Multiplexer

For generating the minimum value of N, X_n and Y_n, each graph have to be first scheduled optimally (each graph can be scheduled using different scheduling technique or even using different parameteres). Design space exploration is quite large in such a scenario and hence we started with some simple experiments. We decided to use ASAP scheduling for all graphs to generate overlay design parameters.

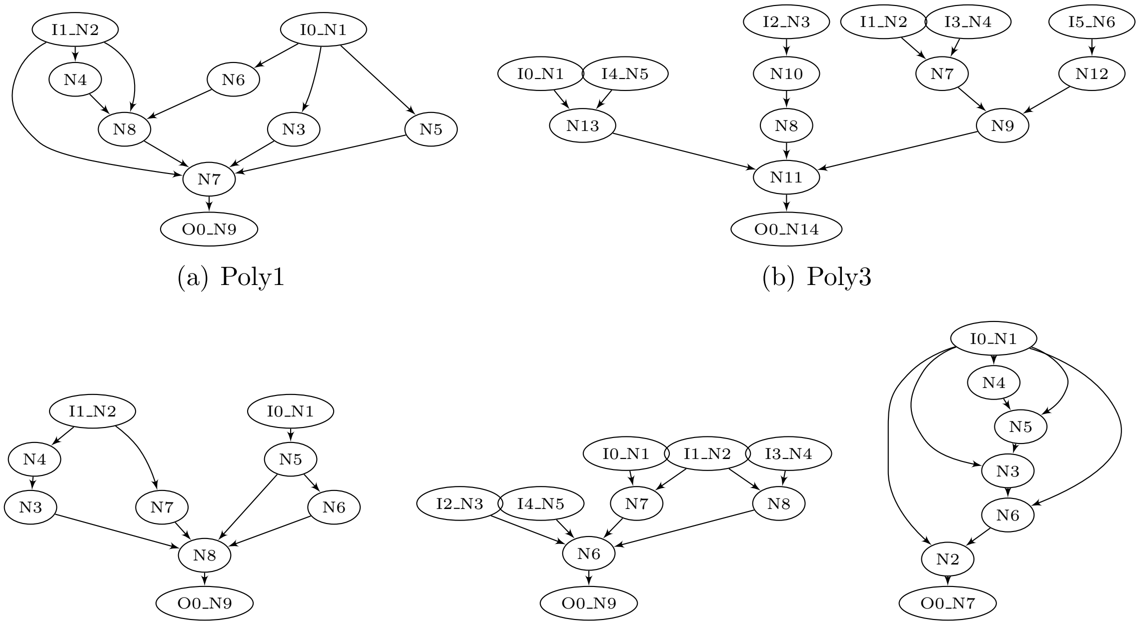
5.2 Set-specific cost calculation using ASAP scheduling

In this section, we calculate overlay design parameters and programmability cost using ASAP scheduling of a set of graphs We considered an example set of composite DFGs as shown in Table 5.1 to demonstrate the process of finding the parameters.

The example set contains five small polynomials requiring up-to 7 DSP blocks as shown in Fig. 5.3. So ideally a network of 7 DSP blocks should be able to map all of the kernels in the example set. Most obvious way of designing the overlay is to have similar tiles having same number of DSP blocks and delay lines. This technique would result in a wastage of resources since the number of operation nodes decreases on moving from one stage to next stage.

Table 5.1: Composite DFG characteristics of Benchmark set-I

No.	Benchmark Name	Characteristics					
		i/o nodes	graph edges	DSP nodes	graph depth	average parallelism	graph width
1.	poly1	2/1	12	6	3	2.00	4
2.	poly2	2/1	10	6	3	2.00	3
3.	poly3	6/1	13	7	3	2.30	4
4.	poly4	5/1	9	3	2	1.50	2
5.	chebyshev	1/1	10	5	5	1.00	1



(a) Poly1

(b) Poly3

(c) Poly2

(d) Poly4

(e) Chebyshev

Figure 5.3: Example set of graphs for generating overlay parameters.

In order to analyze the graphs and to find a valid schedule, we use a python module to implement ASAP scheduling algorithm as shown in the Fig. 5.4. It is important to note that only 40 lines of code is required to implement the ASAP scheduling algorithm. As shown in the code, we have used an API ($graph.incidents(node)$) from python-graph library to find the parents of the node. *asap_schedule* can be used as an API as part of the DFG scheduling library. The input to this API is a graph and the output of the API is a sequenced graph and the minimum latency. With the help of the sequenced graph, we can find out the maximum number of nodes in each stage as well as the number of bypass lines required.

```python
def asap_schedule(graph):
    #scheduling starts here...
    top_level_nodes = []
    scheduled_nodes = []
    unscheduled_nodes = graph.nodes()
    sequencing_graph = []
    l=1;

    for node in graph:   #for each node
        if graph.incidents(node) == []:   #if node is at the top level
            graph.set_level(node, l)  #schedule the node by setting its level l = 1
            top_level_nodes.append(node)  #push this node in the list of top level nodes
            scheduled_nodes.append(node)  #push this node in the list of scheduled nodes
            unscheduled_nodes.remove(node)  #remove this node from the list of unscheduled nodes

    sequencing_graph.append(top_level_nodes) #push all top level nodes in sequencing graph
    make_node_working = 1

    while unscheduled_nodes != []:     #repeat while the list of unscheduled nodes gets empty
        working_nodes = []    #initialize list of nodes which will contain the next level nodes
        l = l + 1
        make_node_working = 1

        for node in unscheduled_nodes:  #select a node from the list of unscheduled nodes
            predecesors = graph.incidents(node) #make a list of all precesessor nodes of this node
            make_node_working = 1
            for predecessor in predecesors: #check each predecessor node if it is already scheduled
                if predecessor in unscheduled_nodes:
                    make_node_working = 0

            if(make_node_working == 1):  #if yes, then
                working_nodes.append(node)  #push this node in the list of working nodes

        sequencing_graph.append(working_nodes) #push next level of nodes into sequencing graph
        for node in working_nodes:
            graph.set_level(node,l)
            scheduled_nodes.append(node)
            unscheduled_nodes.remove(node)

    return [sequencing_graph, l]
```

Figure 5.4: Code for ASAP Scheduling.

For example, the sequenced graph of the kernel ”Poly3” is:

- *Input Stage*: $[N1, N2, N3, N4, N5, N6]$
- $Stage1 : [N12, N10, N7, N13]$
- $Stage2 : [N8, N9]$

- $Stage3 : [N11]$
- $OutputStage : [N14]$

We can find out x_n and y_n for graphs using the information of the sequenced graph. For example, for $Poly3$, x_n is $(4, 2, 1, 1, 0)$ and y_n is $(0, 1, 0, 0, 0)$. We use ASAP scheduling for all the graphs in example set and generate the results as shown in Table 5.2. We find X_n as $(4, 2, 1, 1, 1)$ and Y_n as $(2, 3, 1, 1, 0)$ from the Table 5.2.

Table 5.2: Calculation of Overlay Parameters

(a) Calculation of X_n

X_n	g_1x_n	g_2x_n	g_3x_n	g_4x_n	g_5x_n	max
X_1	4	3	4	2	1	4
X_2	1	2	2	1	1	2
X_3	1	1	1	1	1	1
X_4	1	1	1	0	1	1
X_5	0	0	0	0	1	1

(b) Calculation of Y_n

Y_n	g_1y_n	g_2y_n	g_3y_n	g_4y_n	g_5y_n	max
Y_1	1	0	0	2	1	2
Y_2	3	2	1	0	1	3
Y_3	0	0	0	0	1	1
Y_4	0	0	0	0	1	1
Y_5	0	0	0	0	0	0

We use X_n and Y_n to calculate the cost of the overlay designed specifically for the example set of graphs. The cost can be calculated as $\sum_{n=1}^{N-1}(L_n * (4 * X_{n+1} + Y_{n+1}))$ LUTs/bit. By feeding the values of X_n and Y_n in the equation, we find the cost as 920 LUTs for a 16-bit overlay.

5.3 Resource-budget based set-determination using ASAP scheduling

Now since we can calculate set-specific cost, we determine the set of graphs which can be accommodated within a specified resource budget. We start with all of the graphs in benchmark set and calculate the cost for the same. We call the graph whose removal from the set results in maximum reduction in cost as dominant graph and plot the cost for the multiple sets as shown in Fig. 5.5. The Y axis of the graphs shows the programmability cost, the number of LUTs, and the X axis presents the dominant graph for the set containing graph lying on the right side of the dominant graph. For example, the programmability cost is 920 LUT for the set which is having mibench as dominant graph and the set as poly4, poly1, poly3, poly2, chebyshev. It is clear from the Fig. 5.5 that an overlay can be designed for all of the benchmarks with a programmability cost of 33480 LUTs.

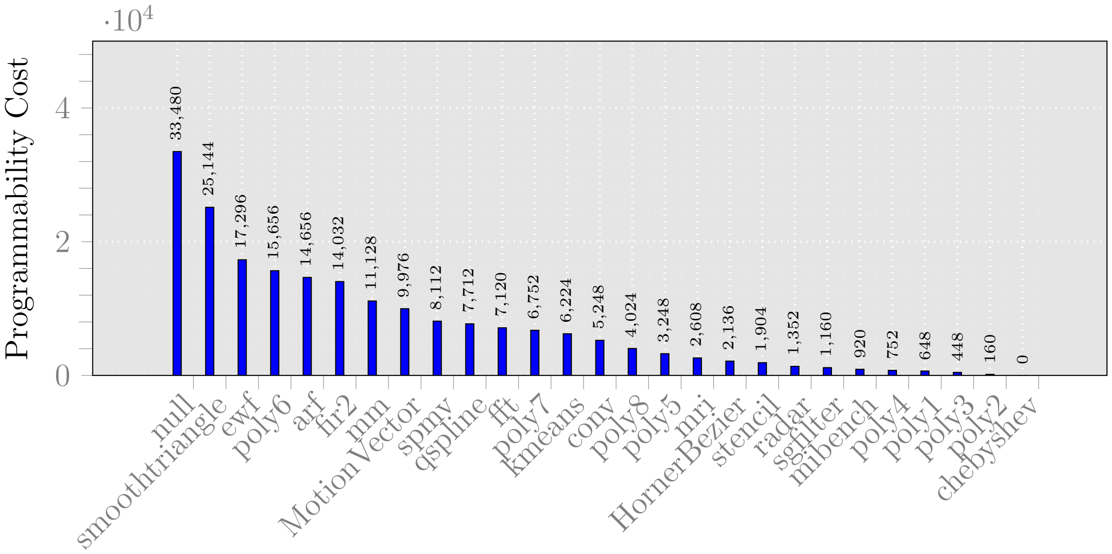


Figure 5.5: Programmability cost for multiple set of graphs

Now from the Fig. 5.5 we can find out the set for a given resource budget as shown in Table 5.3. For example, resource budget of 1000 LUTs can allow a set of 5 graphs which were shown in Fig. 5.3. We can further reduce the programmability cost of the sets using our proposed parameter finding approach.

Table 5.3: Resource-budget based set determination

Resource budget	Benchmark Set	Overlay Cost
1000	set 1 (poly1, poly2, poly3, poly4, chebyshev)	920
2000	set 2 (set 1, mibench, sgfilter, radar)	1904
4000	set 3 (set 2, stencil, HornerBezier, mri)	3248
8000	set 4 (set 3, poly5, poly8, conv, kmeans, poly7, fft)	7712
16000	set 5 (set 4, qspline, spmv, MotionVector, mm, fir2, arf)	15656
32000	set 6 (set 5, poly6, ewf)	25144

5.4 Finding optimal overlay design parameters for reducing cost

As we have seen from the previous section, the programmability cost of the overlay designed for the graphs in set-1 is 920 LUTs using ASAP scheduling for generating sequenced graph. We can further reduce this cost by using different scheduling for different graphs in the set and that can be done by efficiently moving nodes within stages by putting a constraint on maximum number of nodes in a stage.

```
def alap_schedule(graph):
   #scheduling starts here...
   bottom_level_nodes = []
   scheduled_nodes = []
   unscheduled_nodes = graph.nodes()
   sequencing_graph = []
   l=10;

   for node in graph: #for each node
      if graph.neighbors(node) == []: #if node is at the bottom level
      graph.set_level(node, l) #schedule the node by setting its level l = 10
      bottom_level_nodes.append(node) #push this node in the list of bottom level nodes
      scheduled_nodes.append(node) #push this node in the list of scheduled nodes
      unscheduled_nodes.remove(node) #remove this node from the list of unscheduled nodes

   sequencing_graph.append(bottom_level_nodes) #push all bottom level nodes in sequencing graph
   make_node_working = 1

   while unscheduled_nodes != []: #repeat while the list of unscheduled nodes gets empty
      working_nodes = [] #initialize an empty list of nodes which will contain the next level nodes
      l = l - 1
      make_node_working = 1

      for node in unscheduled_nodes: #select a node from the list of unscheduled nodes
         successors = graph.neighbors(node) #make a list of all successors nodes of this node
         make_node_working = 1
         for successor in successors: #check each successor node and check if all are scheduled
            if successor in unscheduled_nodes:
               make_node_working = 0

         if(make_node_working == 1): #if yes, then
            working_nodes.append(node) #push this node in the list of working nodes which contains nodes of level l-1
      sequencing_graph.append(working_nodes) #push next level of nodes into sequencing graph
      for node in working_nodes: #
         graph.set_level(node,l)
         scheduled_nodes.append(node)
         unscheduled_nodes.remove(node)
   return sequencing_graph
```

Figure 5.6: Code for ALAP Scheduling.

List scheduling actually does something similar in which we provide a resources constraint and then generate the sequenced graph. While doing the list scheduling, we use the sequencing graph of the ASAP scheduling and schedule the nodes in each stage based on the mobility of the node. If the number of nodes in a stage are less than the resources available then the sequencing graph is similar to that of an ASAP. But if the maximum number of nodes in a stage are greater than the resources available then the nodes are scheduled based on their mobility. Mobility of a node indicates the range of node in which it can be scheduled. Operations with smaller mobility are given higher priority as they have fewer stages in which the operation can be scheduled. To find the mobility, we initially perform the ASAP and ALAP schedule of the graph and find the difference of the stages for a node scheduled in ASAP and ALAP. This gives the mobility for a specific node and becomes a criteria for List scheduling. The code for ASAP and ALAP is shown in Fig. 5.4 and Fig. 5.6 respectively.

For example if we consider, the scheduling of Poly3 graph on a resource constraint architecture. We get the results as shown in the table 5.4. The API *list_schedule* is created which performs the list scheduling on the graph given as an input along with the scheduling parameter. Scheduling parameter can be described as the resource constraint in each stage. It is clear from the table that on constraining the resources in a stage the depth increases.

Table 5.4: List Scheduling of Poly3 Graph

(a) *Resource Constraint* = 2

Stage	*SequencingGraph*
Input	['N1','N2','N3','N4','N5','N6']
1	['N12','N10']
2	['N7','N13']
3	['N8','N9']
4	['N11']
Output	['N14']

(b) *Resource Constraint* = 3

Stage	*SequencingGraph*
Input	['N1','N2','N3','N4','N5','N6']
1	['N12','N10','N7']
2	['N13','N8','N9']
3	['N11']
Output	['N14']

(c) *Resource Constraint* = 4

Stage	*SequencingGraph*
Input	['N1','N2','N3','N4','N5','N6']
1	['N12','N10','N7','N13']
2	['N8','N9']
3	['N11']
Output	['N14']

To explain our approach of finding optimal overlay parameters, let S equal to [maximum graph width - 1] among G number of graphs present in the set. We can set a resource constraint varying from 2 to $S+1$ while scheduling each graph using list scheduling. This means that S sequencing graphs can be generated for each graph present in the set. Now we can generate S^G programmability costs for all the possible combinations of graphs and their scheduling choices. For example, in case of set-1, $S = 3$ and $G = 5$. Thus there are 3^5 possible choices for overlay parameters. We find the cost for all the possible combinations and consider the minimum cost for the specific set of values of X_n and Y_n. These values of X and Y enable us to design an overlay which will fit all the the graphs in a set. We have considered an example for set - 1 and their parameters are shown in Table 5.5 We have not considered inputs while doing the calculation. The maximum number of scheduling that can be considered are 4 as maximum utilization of a graph in a stage is 4, therefore the sequencing graph will be same as ASAP if we consider more resources for this particular set.

Table 5.5: Calculation of Overlay Parameters

(a) *Chebyshev with* $RC = 2$

N	X_n	Y_n
1	1	1
2	1	1
3	1	1
4	1	1
5	1	0
6	1	0

(b) *Poly1 with* $RC = 2$

N	X_n	Y_n
1	2	2
2	2	3
3	1	3
4	1	0
5	1	0
6	0	0

(b) *Poly1 with* $RC = 3$

N	X_n	Y_n
1	3	2
2	2	2
3	1	0
4	1	0
5	0	0
6	0	0

(d) *Poly1 with* $RC = 4$

N	X_n	Y_n
1	4	1
2	1	3
3	1	0
4	1	0
5	0	0
6	0	0

(e) *Poly2 with* $RC = 2$

N	X_n	Y_n
1	2	1
2	2	1
3	1	3
4	1	0
5	1	0
6	0	0

(f) *Poly2 with* $RC = 3$

N	X_n	Y_n
1	3	0
2	2	2
3	1	0
4	1	0
5	0	0
6	0	0

(g) *Poly3 with* $RC = 2$

N	X_n	Y_n
1	2	4
2	2	2
3	2	1
4	1	0
5	1	0
6	0	0

(h) *Poly3 with* $RC = 3$

N	X_n	Y_n
1	3	2
2	3	0
3	1	0
4	1	0
5	0	0
6	0	0

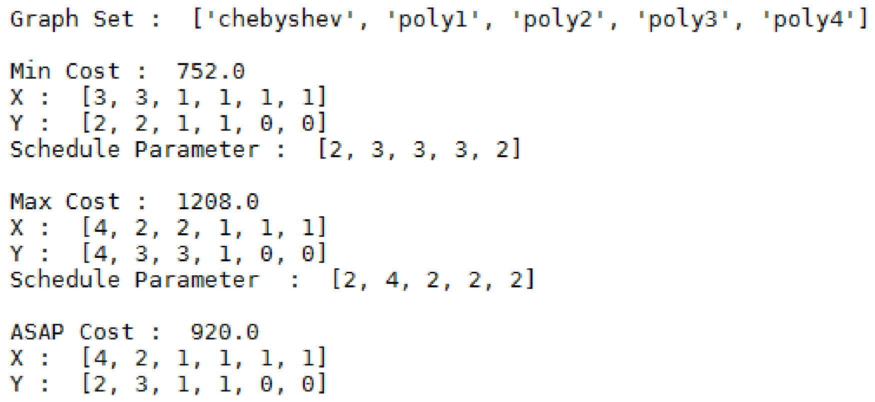
(i) *Poly3 with* $RC = 4$

N	X_n	Y_n
1	4	0
2	2	1
3	1	0
4	1	0
5	0	0
6	0	0

(j) *Poly4 with* $RC = 2$

N	X_n	Y_n
1	2	2
2	1	0
3	1	0
4	0	0
5	0	0
6	0	0

To find the minimum cost of this set we have to consider one scheduling of each graph and thus we will have 3^5 possible sets each giving a programmable cost for that specific set. In Table 5.5 we have the values of X_n and Y_n, thus by forming various sets of each schedule for a graph we can find the X (Max of X_n) and Y (Max of Y_n) for a specific set and further calculate the cost. There would be some repetitive cases while considering the scheduling for graphs. As in chebyshev the graph width is 1 so all the schedules in chebyshev will be similar to its ASAP schedules as it has only one node in each stage.



```
Graph Set :  ['chebyshev', 'poly1', 'poly2', 'poly3', 'poly4']

Min Cost :  752.0
X :  [3, 3, 1, 1, 1, 1]
Y :  [2, 2, 1, 1, 0, 0]
Schedule Parameter :  [2, 3, 3, 3, 2]

Max Cost :  1208.0
X :  [4, 2, 2, 1, 1, 1]
Y :  [4, 3, 3, 1, 0, 0]
Schedule Parameter  :  [2, 4, 2, 2, 2]

ASAP Cost :  920.0
X :  [4, 2, 1, 1, 1, 1]
Y :  [2, 3, 1, 1, 0, 0]
```

Figure 5.7: Output Screenshot for a set of graphs

For $set-1$ we will have 1 unique schedule for chebyshev, 3 unique schedules for poly1, 2 unique schedules for poly2, 3 unique schedules for poly3 and 1 unique schedule for poly4. Thus there would be $1\times3\times2\times3\times1 = 18$ unique costs for $set-1$. X and Y of these 10 schedules are shown in Table 5.5. All these combinations are handled by our tool and it gives the minimum cost for a set of graph. The input given to this tool are the number of graphs, Max graph width and the dot files of the graphs. The output of this tool is the Min cost, Max cost and ASAP cost along with the value of X, Y and the Schedule Paramater. This output can be seen in Fig 5.7 and the code for finding the Min, Max and ASAP cost can be seen in Fig 5.12. Fig 5.8 shows the cost of various possible sets of overlay parameters for set - 1. As explained earlier, there would be maximum of 18 unique cost for this set of graph which can be seen in Fig 5.8.

From the graph we see that Max cost is greater than the ASAP cost. There could be some combinations of graphs with certain schedulability such as one graph has more operations at the initial stages whereas the other graph in a set has most of its operations in the mid stage, then the max cost of such a set would definitely be more than ASAP. Max cost would be always be equal to or greater than ASAP cost. Thus the schedule corresponding the max cost is a worst way of designing an overlay that can fit the desired set of graph.

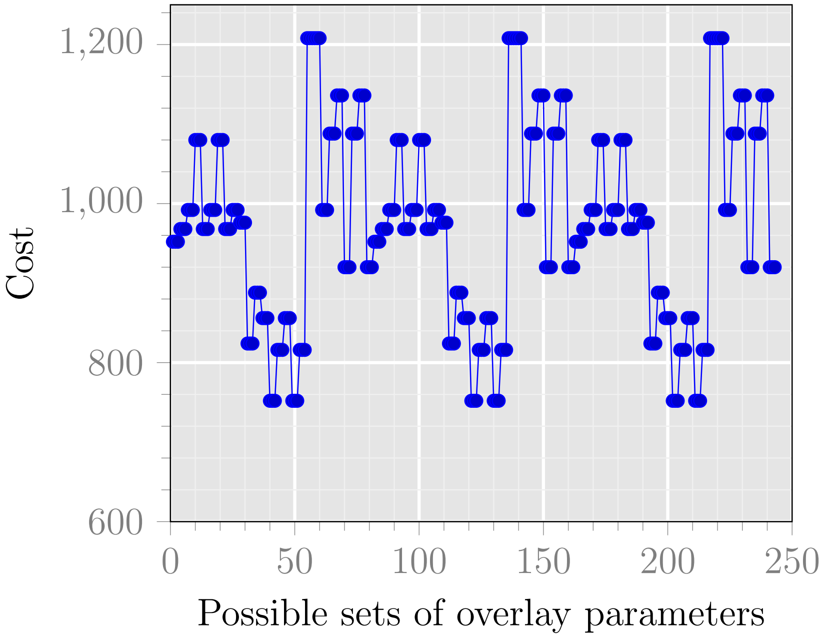


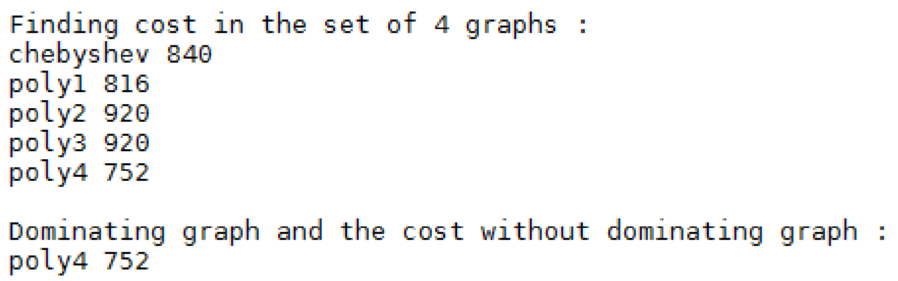
Figure 5.8: All possible costs for set - 1

After finding the minimum cost for a specific set of graph, we considered all the benchmarks and found out the most dominating graph among the lot and then the cost after removing the same. Dominating graph is the one whose removal from the set results in maximum decrease in the cost of the set. This helped us in finding out the most dominating graph in a set of benchmark which would be mostly responsible for the large overlay design relatively as compared to other graphs in a set. Thus if we have a resource constraint and our goal is to fit maximum number of graphs from a set of benchmarks then we would eliminate the most dominating graphs from the set until we meet the resource constraint. Thus elimination of dominating graphs from the set would result in effective utitlization of resources rather than randomly removing graphs from a set.

```
from scheduler import asap_schedule
from scheduler import alap_schedule
from scheduler import list_schedule
graphlist = ['chebyshev', 'poly1', 'poly2', 'poly3', 'poly4']
#graphlistcopy = graphlist
mylist = list(graphlist)
mydict = {}
dominantdict = {}
num = 4
while(num > 3):
        print '\nFinding cost in the set of', num, 'graphs : '
        for each in graphlist:
                mylist.remove(each)
                command = 'python app.py ' + str(num) + ' 2 12';
                for bench in range(0, num):
                        command = command + ' ' + mylist[bench]
                [status, cost] = commands.getstatusoutput(command)
                print ,each, cost
                mylist = [];
                mylist = list(graphlist)
                mydict[each] = int(cost)
        mincost = min(mydict.values())
        print "Dominating graph and the cost without dominating graph : "
        print mydict.keys()[mydict.values().index(mincost)], mincost,"\n\n\n\n\n"
        domi_graph = mydict.keys()[mydict.values().index(mincost)]
        dominantdict[domi_graph] = mincost
        graphlist.remove(domi_graph)
        num = num - 1;
```

Figure 5.9: Code for finding the dominating graph and cost for the set of graph.

Fig 5.9 shows the code for finding the dominating graph among the set of benchmarks and the cost after removing the same. The output of the code is shown in Fig 5.10.



```
Finding cost in the set of 4 graphs :
chebyshev 840
poly1 816
poly2 920
poly3 920
poly4 752

Dominating graph and the cost without dominating graph :
poly4 752
```

Figure 5.10: Output Screenshot showing the dominating graph along with the cost

We see that in a set of 5 graphs : chebyshev, poly1, poly2, poly3 and poly4 the most dominaing one is poly4 as there is maximum reduction in cost by removing the same from the set. Thus cost for set - 1 is 920 and cost after removing poly4 is 752. This is the minimum resource requirement if we want to fit 4 graphs from set - 1 by using minimum resources. Same approach has been used to create the Fig 5.11. We initially considered a set of 26 graphs and then kept on removing the dominating graph and found the cost after removing the same. Cost reduction is shown in Table 5.14

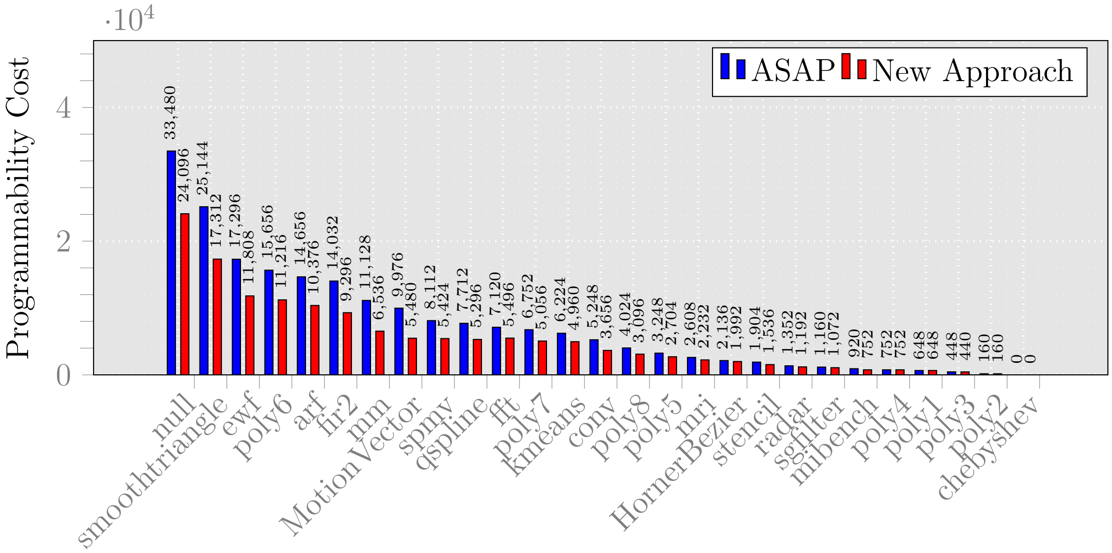


Figure 5.11: Programmability cost using new approach for multiple set of graphs

Table 5.6: Cost reduction using proposed approach

benchmark set	Overlay Cost (ASAP approach)	Overlay Cost (new approach)
set 1 (poly1, poly2, poly3, poly4, chebyshev)	920	752
set 2 (set 1, mibench, sgfilter, radar)	1904	1536
set 3 (set 2, stencil, HornerBezier, mri)	3248	2704
set 4 (set 3, poly5, poly8, conv, kmeans, poly7, fft)	7712	5296
set 5 (set 4, qspline, spmv, MotionVector, mm, fir2, arf)	15656	11216
set 6 (set 5, poly6, ewf)	25144	17312

```
# Read the dot file and create a graph
g_num = int(sys.argv[1]);
G = []; G = create_listof_graph()
s_num = int(sys.argv[2]); S = [];
for count in range(2, 1+s_num):
   S.append(count + int(sys.argv[3]))
s_num = s_num - 1;
A_x_3 = [[[]]]; A_x_2 = [[]]; A_x_1 = []; A_x_3.pop(0); B_x_3 = [[[]]]; B_x_2 = [[]]; B_x_1 = []; B_x_3.pop(0)
A_y_3 = [[[]]]; A_y_2 = [[]]; A_y_1 = []; A_y_3.pop(0); B_y_3 = [[[]]]; B_y_2 = [[]]; B_y_1 = []; B_y_3.pop(0)
N_list = [];
for g_iter in range(0, g_num):
   for s_iter in range(0,s_num):
      blocks = []; edges = []
      [blocks, edges, N] = list_schedule(G[g_iter], S[s_iter])
      A_x_2.append(blocks); A_y_2.append(edges);
      N_list.append(N)
   A_x_2.pop(0); A_y_2.pop(0)
   A_x_3.append(A_x_2); A_y_3.append(A_y_2);
   A_x_2 = [[]]; A_y_2 = [[]];
Nmax = max(N_list)
for g_iter in range(0, g_num):
   for s_iter in range(0, s_num):
      loop_count_x = max(N_list) - len(A_x_3[g_iter][s_iter])
      for count_x in range(0, loop_count_x):
         A_x_3[g_iter][s_iter].append(0)
         loop_count_y = max(N_list) - len(A_y_3[g_iter][s_iter])
         for count_y in range(0, loop_count_y):
A_y_3[g_iter][s_iter].append(0)
temp_list_1 = []; temp_list_2 = []
for final_iter in range(0, pow(s_num,g_num)):
   for g_iter in range(0, g_num):
      x = g_iter;
      y = final_iter/(pow(s_num, ((g_num - 1) - g_iter))) % s_num;
      temp_list_1.append(x); temp_list_2.append(y)
      B_x_2.append(A_x_3[x][y]); B_y_2.append(A_y_3[x][y])
   B_x_2.pop(0); B_y_2.pop(0)
   B_x_3.append(B_x_2); B_y_3.append(B_y_2);
   B_x_2 = [[]]; B_y_2 = [[]];
temp_list_x = []; temp_list_y = []
list_x = []; list_x_all = [[]]; list_x_all.pop(0); list_y = []; list_y_all = [[]]; list_y_all.pop(0)
for final_iter in range(0, pow(s_num,g_num)):
   for count in range(0, Nmax):
      for g_iter in range(0,g_num):
         temp_list_x.append(B_x_3[final_iter][g_iter][count])
         temp_list_y.append(B_y_3[final_iter][g_iter][count])
      list_x.append(max(temp_list_x));list_y.append(max(temp_list_y))
      temp_list_x = []; temp_list_y = []
   list_x_all.append(list_x);
   list_y_all.append(list_y)
   list_x = [[]]; list_y = [[]]
   list_x.pop(0); list_y.pop(0)
cost = []; val = 0;
for final_iter in range(0, pow(s_num,g_num)):
   val = 0;
   for count in range(0, Nmax):
      x = list_x_all[final_iter][count]
      y = list_y_all[final_iter][count]
      c = LutsRequired(x, y);
      val = val + c
   cost.append(val)
min_cost = min(cost); max_cost = max(cost); min_cost_idx_list = []; max_cost_idx_list = []
for iter in range(0, pow(s_num,g_num)):
   if min_cost == cost[iter]:
      min_cost_idx_list.append(iter)
   if max_cost == cost[iter]:
      max_cost_idx_list.append(iter)
min_cost_idx = min_cost_idx_list[0]; max_cost_idx = max_cost_idx_list[0]
scheduling_min_list = []; scheduling_max_list = []
for g_iter in range(0,g_num):
   val = (min_cost_idx/(pow(s_num,(g_num - 1 - g_iter)))) % s_num
scheduling_min_list.append(S[val])
   val = (max_cost_idx/(pow(s_num,(g_num - 1 - g_iter)))) % s_num
scheduling_max_list.append(S[val])
mincost = min(cost)
```

Figure 5.12: Code for minimum Cost Calculation for a set of graph.

5.5 Comparison with Island-style FPGA Overlay

We analyze the mapping of kernels on DSP block based island-style overlay and compare the programmability cost with linear overlay. Table 5.7 shows the required size of island-style overlay for each benchmark.

Table 5.7: Analysis of kernels on island-style overlay

	Benchmark	Benchmark Characteristics				
No.	Name	i/o nodes	op nodes	DSP nodes	% Savings	Required size
1.	chebyshev	1/1	7	5	28%	3×3 (CW=2)
2.	sgfilter	2/1	18	10	44%	4×4 (CW=2)
3.	mibench	3/1	13	6	53%	3×3 (CW=2)
4.	qspline	7/1	26	22	15%	5×5 (CW=2)
5.	poly1	2/1	9	6	33%	3×3 (CW=2)
6.	poly2	2/1	9	6	33%	3×3 (CW=2)
7.	poly3	6/1	11	7	36%	3×3 (CW=2)
8.	poly4	5/1	6	3	50%	2×2 (CW=2)
9.	poly5	3/1	27	14	48%	4×4 (CW=2)
10.	poly6	3/1	44	25	43%	6×6 (CW=2)
11.	poly7	3/1	39	21	50%	5×5 (CW=2)
12.	poly8	3/1	32	17	46%	5×5 (CW=2)
13.	fft	6/4	10	8	20%	3×3 (CW=2)
14.	kmeans	16/1	23	20	13%	5×5 (CW=2)
15.	mm	16/1	15	8	46%	6×6 (CW=2)
16.	mri	11/2	11	7	36%	4×4 (CW=2)
17.	spmv	16/2	14	8	42%	5×5 (CW=2)
18.	stencil	15/2	14	8	42%	5×5 (CW=2)
19.	conv	24/8	16	8	50%	8×8 (CW=2)
20.	radar	10/2	8	6	25%	3×3 (CW=2)
21.	arf	26/2	28	20	28%	8×8 (CW=2)
22.	ewf	21/5	34	18	47%	10×10 (CW=2)
23.	fir2	17/1	23	8	65%	10×10 (CW=2)
24.	hornerbezier	12/4	14	8	42%	4×4 (CW=2)
25.	motionvenctor	25/4	24	12	50%	8×8 (CW=2)
26.	smoothtriangle	29/14	37	25	32%	11×11 (CW=2)

According to this table, we calculate the programmability cost of graphs sets described in the previous section. A programmability cost of 440 LUTs per tile was shown in [6] based on which we calculate the cost for the sets. Table 5.8 shows the required overlay size for the sets and programmability cost comparison.

Table 5.8: Programmability cost comparison

benchmark set	Required Size	Island-Cost	ASAP/proposed (linear cost)
set 1 (poly1, poly2, poly3, poly4, chebyshev)	3×3 (CW=2)	3960	920/752
set 2 (set 1, mibench, sgfilter, radar)	4×4 (CW=2)	7040	1904/1536
set 3 (set 2, stencil, HornerBezier, mri)	5×5 (CW=2)	11000	3248/2704
set 4 (set 3, poly5, poly8, conv, kmeans, poly7, fft)	8×8 (CW=2)	28160	7712/5296
set 5 (set 4, qspline, spmv, MotionVector, mm, fir2, arf)	10×10 (CW=2)	44000	15656/11216
set 6 (set 5, poly6, ewf)	10×10 (CW=2)	44000	25144/17312

The data in Table 5.8 is shown in Fig. 5.13 and it is clear from the table that the programmability cost can be reduced drastically using linear dataflow style overlays. We observe upto 77% reduction in cost for sets using ASAP based approach and upto 81% reduction using proposed approach as shown in Fig 5.14.

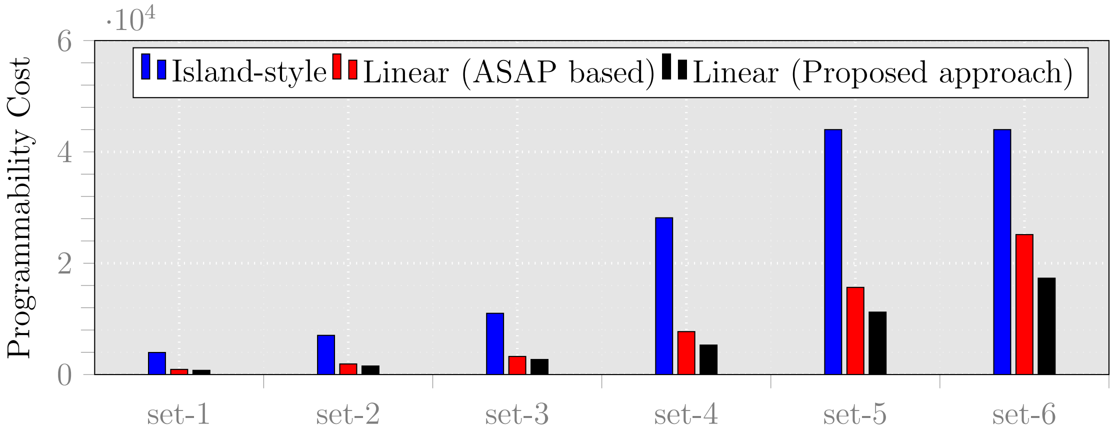


Figure 5.13: Programmability cost using new approach for multiple set of graphs

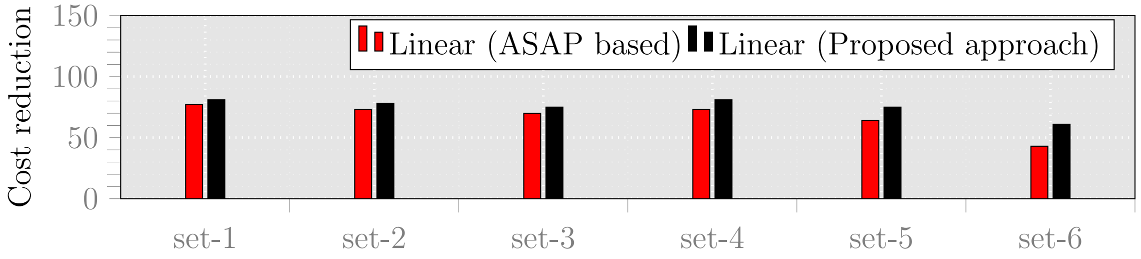


Figure 5.14: Cost reduction using new approach for multiple set of graphs

5.6 Summary

In this chapter, we presented linear dataflow overlay architecture, programmability cost modelling, cost calculation using sequenced graphs (ASAP scheduled), cost reduction by proposed approach, cost comparison with island-style overlays. We observe upto 77% reduction in cost for sets using ASAP based approach and upto 81% reduction using proposed approach compared to island-style overlays.

Chapter 6

Experiments

In this chapter, we present experiments to evaluate the performance of overlay architecture against a set of commercial devices, such as 16-core EPIPHANY device and dual-core ARM cortex-A9 for a set of compute kernels.

6.1 Performance Evaluation of Parallella Platform

6.1.1 16-core EPIPHANY device

The Parallela SoC houses the EPIPHANY co-processor, which is a many core, shared-memory, parallel computing fabric. It consists of a 2D array of compute nodes connected by a mesh network-on chip with dedicated Floating Point Unit and a local 32 KB memory on each node. The EPIPHANY chip works with an ARM Cortex A9 host which offloads computation to it. The EPIPHANY chip has either 16 or 64 cores but is scalable to a larger number of cores. All cores share a 1 GB of offchip shared RAM. These cores can communicate with each other using the on-chip mesh network. The Parallellas SDK provides direct read and write access to the local memory of a core from any other core using this network-on-chip.

Fig 6.1 and Fig 6.2 shows the EPIPHANY Architecture and eMesh Network-On-Chip Architecture respectively. We see that EPIPHANY has 16 cores running at 600 MHz so it would be interesting to see the performance of the benchmarks on

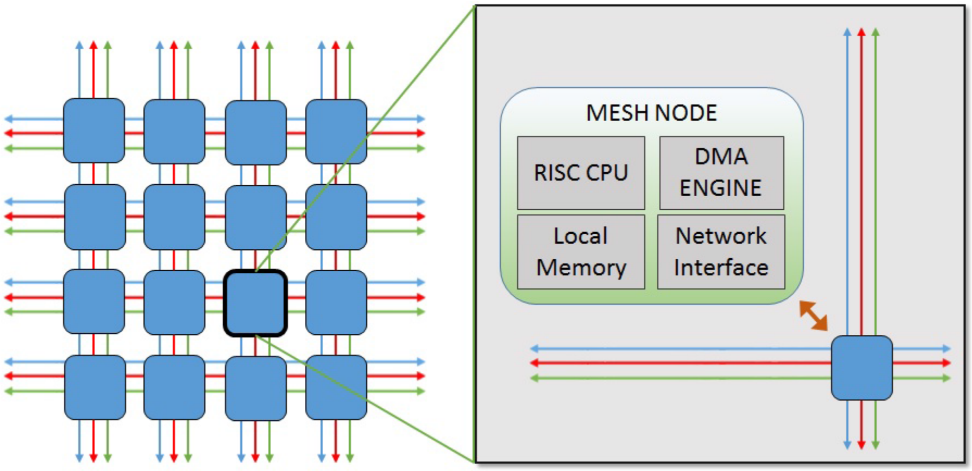


Figure 6.1: EPIPHANY Architecture

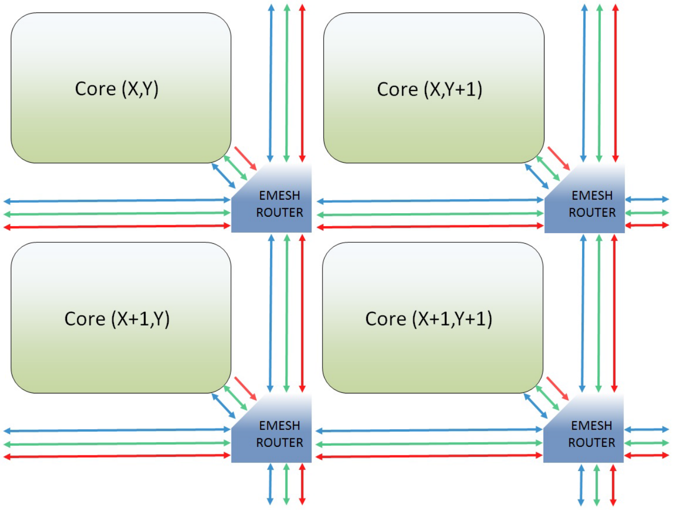


Figure 6.2: eMesh Network-On-Chip Architecture

Parallella. One noticeable thing about Parallella is that its local cores has only 32KB of On-Chip Memory. Thus we have to use the local memory very efficiently as 16 KB of the memory is used as Stack and Program memory (However this is flexible but in total we have only 32KB of local memory for each core). It provides 1 GB of Shared RAM as well but eCore to Shared RAM bandwidth is very less as compared to local bandwidth and thus we have considered only local memory for performing the experiments so as to achieve better performance.

Table 6.1: Performance of Benchmarks on EPIPHANY

No.	Name	Effective Time	Samples	Computation Time	M DFGs/s	operations	MOPs
1.	chebyshev	14	32000	480	66.67	7	467
2.	sgfilter	22	19200	424	45.28	18	815
3.	mibench	21.5	16000	341	46.92	13	610
4.	qspline	32	8000	258	31.01	26	806
5.	poly1	17	19200	332	57.83	9	520
6.	poly2	20	19200	356	53.93	9	485
7.	poly3	24	8000	185	43.24	11	476
8.	poly4	20	9600	185	51.89	6	311
9.	poly5	26	16000	424	37.74	27	1019
10.	poly6	36	16000	572	27.97	44	1231
11.	poly7	34	16000	553	28.93	39	1128
12.	poly8	30	16000	480	33.33	32	1067
13.	fft	31	6400	203	31.53	10	315
14.	kmeans	48.1	3840	185	20.76	23	477
15.	mm	38.2	3840	147	26.12	15	392
16.	spmv	67.2	3840	258	14.88	14	208
17.	stencil	38.2	3840	147	26.12	14	366
18.	conv	57.8	1920	111	17.30	16	277
19.	radar	28.7	5120	147	34.83	8	279
20.	arf	67.7	1920	130	14.77	28	414
21.	ewf	69.1	2400	166	14.46	34	492
22.	fir2	52.6	3520	185	19.03	23	438
23.	hornerbezier	38.5	4000	166	24.10	14	337
24.	motionvector	90.6	2240	203	11.03	24	265
25.	smoothtriangle	140	1440	203	7.09	37	262

We conducted the experiments by running the compute kernels on EPIPHANY device and results are shown in Table 6.1. Initially host sends data to the core which becomes the input for the graph. Same data is being written in the local memory of each core with the help of *e_write* API. If each data width is four bytes then maximum number of data that can be sent to the core is 4000 (16KB/4B) samples. For each graph the number of inputs and outputs may vary, thus if we have P number of inputs and Q number of outputs then the maximum number of samples that can run with the data in local memory is equal to $4000/(P + Q)$ iterations. We can run $4000/(P + Q)$ iterations of each graph on each core and total of $64000/(P + Q)$ iterations if we run the same graph on 16 cores. Max iterations that we can run for a unique set

```c
int main(void)
{
        // Local Variables
        e_platform_t platform;
        e_epiphany_t dev;
        e_mem_t emem;
        e_init(NULL);           //initialize device
        e_reset_system();
        e_get_platform_info(&platform);
        e_open(&dev, 0, 0, platform.rows, platform.cols);    // Open a workgroup
        //papi Declarations
        int events[2]={PAPI_TOT_CYC, PAPI_TOT_INS}, ret;
        long_long t0_dataLoad, t1_dataLoad, t0_programLoad, t1_programLoad, t0_compute, t1_compute, t0_total, t1_total, t0_read, t1_read;
        int i, j, k;
        float array[INPUT_SIZE], value;
        unsigned time = 0, doneFlag = 0;
        for (k = 0; k < INPUT_SIZE; k++)
        {
                array[k] =(float)(1*k);
        }
        t0_total = PAPI_get_virt_usec();
        t0_dataLoad = PAPI_get_virt_usec();
        for(i = 0; i < 4; i++)
        {
                for (j = 0; j < 4; j++)
                {
                        e_write(&dev, i, j, (off_t)INPUT_ADDR, (unsigned*)&array[0], INPUT_SIZE * sizeof(float));
                }
        }
        t1_dataLoad = PAPI_get_virt_usec();
        t0_programLoad = PAPI_get_virt_usec();
        // Load PE side and start SMVM
        e_load_group("pe.srec", &dev, 0, 0, platform.rows, platform.cols, E_FALSE);
        e_start_group(&dev);
        t1_programLoad = PAPI_get_virt_usec();
        t0_compute = PAPI_get_virt_usec();
        while(doneFlag != 1)
        {
                e_read(&dev, 0, 0, (off_t)0x5FFC , &doneFlag, sizeof(unsigned));
        }
        t1_compute = PAPI_get_virt_usec();
        t0_read = PAPI_get_virt_usec();
        for(i = 0; i < OUTPUT_SIZE; i++)
        {
                e_read(&dev, 0, 2, (off_t)(unsigned*)(OUTPUT_ADDR + 4*i), &value, sizeof(float));
        }
        t1_read = PAPI_get_virt_usec();
        t1_total = PAPI_get_virt_usec();
        printf("Samples : %d \n", SAMPLES);
        printf("Data Load Time : %lld uSec\n", t1_dataLoad-t0_dataLoad);
        printf("Program Load Time : %lld uSec\n", t1_programLoad-t0_programLoad);
        printf("Compute Time : %lld uSec\n", t1_compute-t0_compute);
        printf("Read Time : %lld uSec\n", t1_read-t0_read);
        printf("Total Time : %lld uSec\n", t1_total-t0_total);
        e_read(&dev, 0, 0, (off_t)0x5FF8 , &doneFlag, sizeof(unsigned));
        printf("Computation time using Core Timer : %f uSec\n", (float)doneFlag/600.0);
        e_close(&dev);    // Close the workgroup
        e_finalize();    //Finalize e-platform connection
        return 0;
}
```

Figure 6.3: Parallella Host code for Chebyshev Benchmark

of data is 32000 which is possible when we have $P = 1$ and $Q = 1$, which is the case for chebyshev. Thus the number of samples depend on the input and output of the graph as the memory is limited (32KB) and we can see in Table 6.1 that the graph with less number of samples have more I/O. Fig 6.3 shows the host code and Fig 6.4 shows the code for chebyshev benchmark which runs on each core.

```c
#include <stdint.h>
#include "e_lib.h"
#include "stdlib.h"
#include "string.h"
#include "e_lib.h"
#include <e_coreid.h>
#include "matrix.h"

// global memory management -- stack start address, giving 8KB of stack
asm(".global __stack_start__;");
asm(".set __stack_start__,0x6000;");
asm(".global __heap_start__;");
asm(".set __heap_start__,0x0000;");
asm(".global __heap_end__;");
asm(".set __heap_end__,0x1fff;");

volatile e_barrier_t barriers[16];
        e_barrier_t *tgt_bars[16];

float *data = (float*)INPUT_ADDR;
float *output = (float*)OUTPUT_ADDR;

int main()
{
       // Timer Variables
       e_irq_mask(E_TIMER0_INT, E_TRUE);
       e_ctimer_set(E_CTIMER_0, E_CTIMER_MAX);
       unsigned time_e, time_s;
       unsigned* clock = (unsigned*)0x5FF8;
       unsigned* doneFlag = (unsigned*)0x5FFC;
       *doneFlag = 0;
       *clock = 0;
       unsigned k=0,j=0,l=0;
       float result, input;

       //initialize barriers
       e_barrier_init(barriers, tgt_bars);
       e_barrier(barriers, tgt_bars);

       e_ctimer_start(E_CTIMER_0, E_CTIMER_CLK);
       time_s = e_ctimer_set(E_CTIMER_0, E_CTIMER_MAX);

       while(k<INPUT_SIZE)
       {
           input = *(data + k);
              *(output + j) = (input * (input * (16 * input * input - 20) * input + 5));
              k = k + INPUTS;
           j = j + OUTPUTS;
       }
       e_barrier(barriers, tgt_bars);
       time_e = e_ctimer_get(E_CTIMER_0);
       //e_barrier(barriers, tgt_bars);
       *clock = time_s - time_e;
       e_barrier(barriers, tgt_bars);
       *doneFlag = 1;
       return 0;
}
```

Figure 6.4: Parallella Core code for Chebyshev Benchmark

6.1.2 Dual-core ARM Cortex-A9

We conducted another experiment to observe the performance of dual-core ARM Cortex-A9 processor. Results are shown in Table 6.2. After performing the experiments on ARM and EPIPHANY, we calculated the MDFGs per second and Million Operations Per Second (MOPS) for the same set of benchmarks. Results are shown in Fig 6.5 and Fig 6.6. We observe MDFGs per second ranging from 7 to 67 and

MOPS ranging from 262 to 1231 for EPIHANY device. We observe MDFGs per second ranging from 3 to 18 and MOPS ranging from 73 to 250 for dual-core ARM Cortex-A9.

Table 6.2: Performance of Benchmarks on Dual-core ARM Cortex-A9

No.	Name	Effective Time	Samples	Computation Time	M DFGs/s	operations	MOPs
1.	chebyshev	70.78	100000	7078	14.13	7	99
2.	sgfilter	97.14	100000	9714	10.29	18	185
3.	mibench	74.65	100000	7465	13.40	13	174
4.	qspline	216.02	100000	21602	4.63	26	120
5.	poly1	76.5	100000	7650	13.07	9	118
6.	poly2	76.86	100000	7686	13.01	9	117
7.	poly3	104.7	100000	10470	9.55	11	105
8.	poly4	82.21	100000	8221	12.16	6	73
9.	poly5	196.5	100000	19650	5.09	27	137
10.	poly6	254	100000	25400	3.94	44	173
11.	poly7	246.25	100000	24625	4.06	29	118
12.	poly8	225.6	100000	22560	4.43	32	142
13.	fft	54.38	100000	5438	18.39	10	184
14.	kmeans	156.3	100000	15630	6.40	23	147
15.	mm	117.4	100000	11742	8.52	15	128
16.	spmv	87.18	100000	8718	11.47	14	161
17.	stencil	56.03	100000	5603	17.85	14	250
18.	conv	87.5	100000	8755	11.42	16	183
19.	radar	57.33	100000	5733	17.44	8	140
20.	arf	249.76	100000	24976	4.00	28	112
21.	ewf	266.34	100000	26634	3.75	34	128
22.	fir2	189.48	100000	18948	5.28	23	121
23.	hornerbezier	145.98	100000	14598	6.85	14	96
24.	motionvector	227.27	100000	22727	4.40	24	106
25.	smoothtriangle	324.77	100000	32477	3.08	37	114

6.2 Performance Evaluation of FPGA Overlays

Overlay architectures can execute one DFG iteration each cycle by using fully pipelined functional units. One such overlay was demonstrated in [6] which was able to execute DFGs at a frequency of above 300 MHz. In order to have a fair comparison of performance, we choose a practically feasible frequency of 250 MHz for overlays, resulting

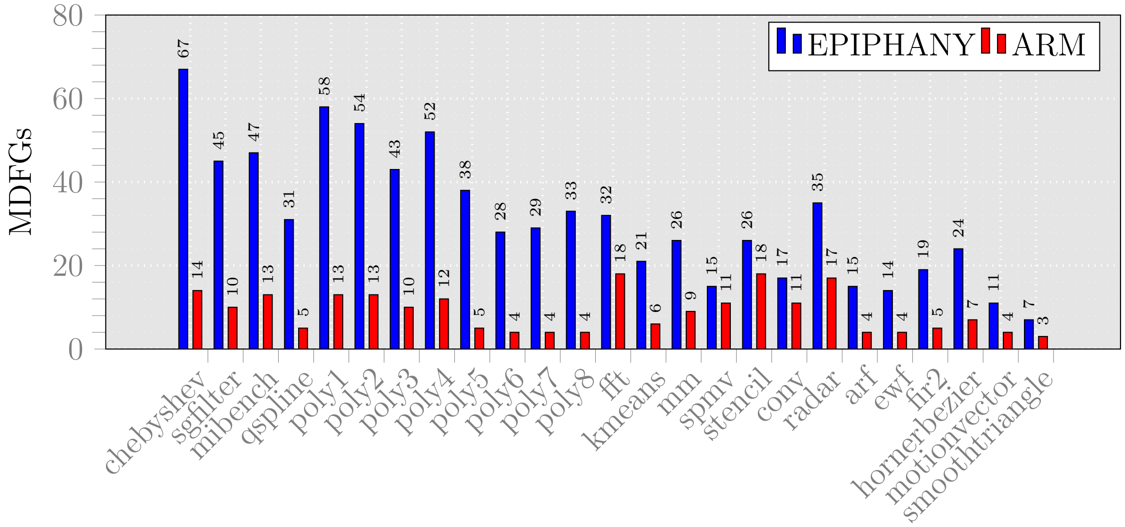


Figure 6.5: Comparison of MDFGs on EPIPHANY and ARM

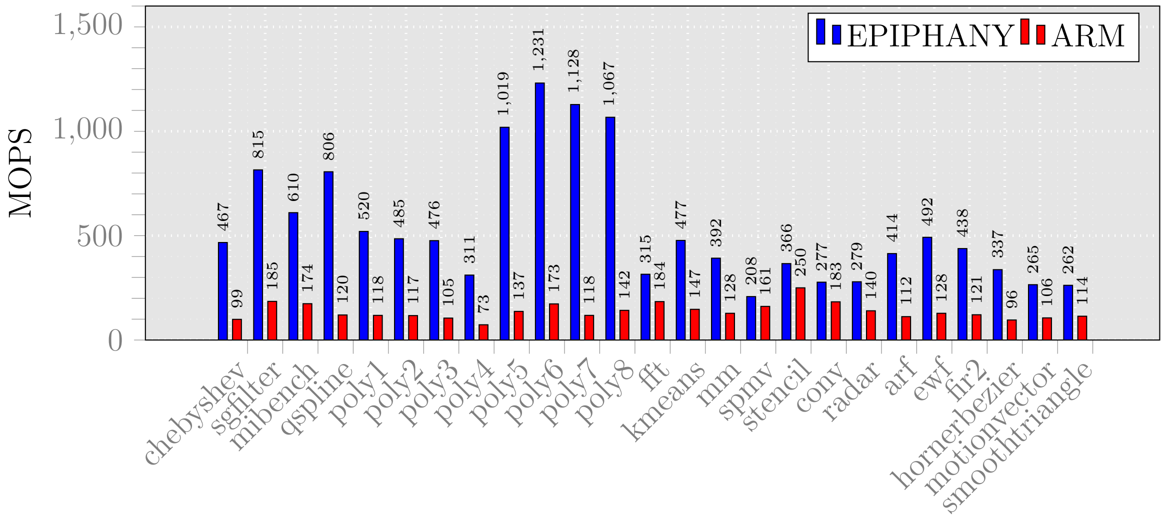


Figure 6.6: Comparison of MOPs on EPIPHANY and ARM

in 250 MDFGs per second, and estimate MOPS for the compute kernels ranging from 1500 to 11000 MOPS. We show the comparison of MOPS in Fig. 6.7. Y axis shows the natural log of MOPS. It is clear from the figure that overlays running at 250 MHz can provide an order of magnitude better performance assuming that the required I/O interfaces are present on the physical platform.

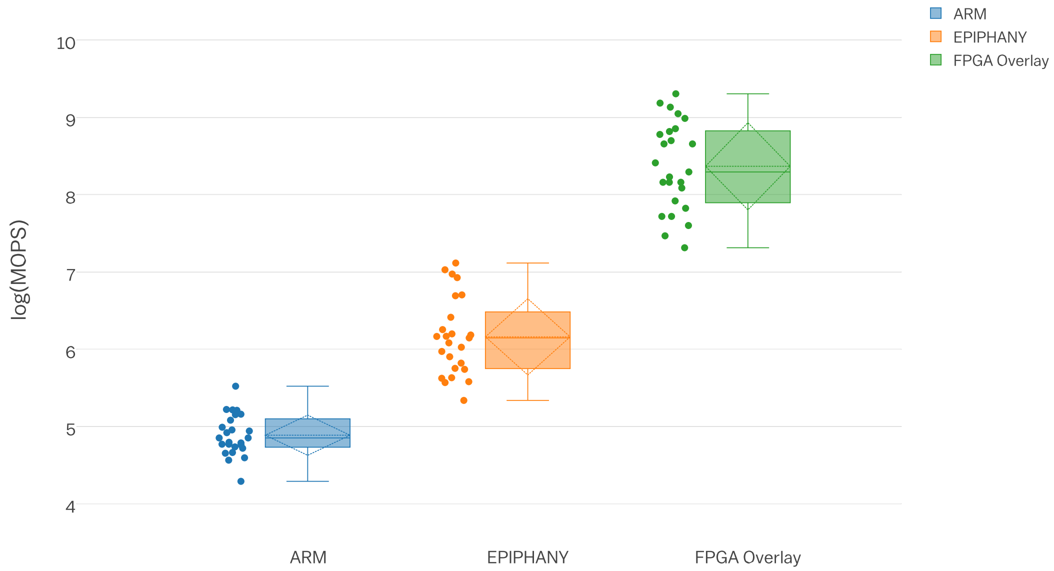


Figure 6.7: Performance Comparison with Overlays

6.3 Summary

In this chapter, we presented experiments to evaluate the performance of overlays against set of commercial devices and observed that ARM, EPIPHANY and FPGA overlays provide a performance of upto 250, 1231 and 11000 MOPS, respectively.

Chapter 7

Conclusions and Future Work

This chapter concludes and summarizes this report. Furthermore, in this chapter we discuss future research directions in detail.

7.1 Conclusions

This report proposed an approach for designing area efficient overlay architectures by reducing programmability cost for a set of compute kernels. Use of programmability cost modeling provided the base for finding optimal design parameters of overlay for a set of kernels. This work included developing an understanding of compute kernels, hardware acceleration concept, overlay architectures, terminologies and techniques, detailed performance evaluation of overlays and commercial multi-core devices. Experiments were designed to characterize the compute kernels, model the programmability cost, evaluate the cost for a set of graph, optimize the cost using proposed overlay parameter finding approach, compare the programmability cost of linear dataflow overlay with island-style overlay. A set of python modules were developed for different scheduling algorithms and were used to find the sequenced graphs which were then used to find the optimal overlay design parameters. Before we could begin finding the optimal overlay design parameters, an in-depth knowledge of the current trends and previous efforts in the field of overlay architectures were studied to compare and contrast their features.

An analysis of compute kernels and effect of DSP-aware composition were presented in chapter 4. We observe up-to 65% reduction in the number of nodes, up-to 32% reduction in the number of edges and up-to 42% reduction in the graph depth using DSP aware-composition. For the composite graphs, experiments were conducted to compare the programmability cost of island-style overlay with linear dataflow overlay and results were presented in chapter 5. We observe upto 77% reduction in cost for sets using ASAP based approach and upto 81% reduction using proposed approach compared to island-style overlays. The performance of commercial multi-core devices were evaluated for the compute kernels and were presented in 6. We estimated the performance of overlays and showed that it can achieve a performance of 11000 MOPS. Furthermore, the approach presented in this report facilitates high level application developers to use area-efficient overlays for hardware acceleration of compute kernels at significantly higher performance.

7.2 Future work

Some of the main future research directions are overlay designs for low programmability cost, automated RTL generation of overlays for kernel sets, overlay integration with host processor and a run time management system for overlays. We describe these directions in detail as follows:

- **Alternative routing network architectures for programmability cost reduction**: The programmability cost of the overlays can be further reduced by carefully designing the routing network architecture. Some of the possible choices are multistage switching networks, hierarchical routing network and omega network etc.
- **Automated RTL generation of overlays for different sets of kernels**: From the overlay parameters, it is possible to develop an RTL generator for overlay architectures.
- **Integration of overlay with a host processor on a heterogeneous computing platform**: Integration of overlay with a host processor is crucial for

effective runtime management of overlays

- **Mapping of a large kernel on overlays supporting cycle by cycle tile reconfiguration capability**: DSP blocks can be reconfigured on a cycle by cycle basis and hence overlays can be designed to support large kernels by temporally mapping operations on tiles.
- **A python library for graph scheduling algorithms**: Using Python-graph library, we developed basic graph scheduling algorithms which can be used further to develop python modules for advance graph scheduling techniques.

Finally, with these initiatives we hope to develop overlay architecture based accelerator design methodology where compute kernels can be compiled on area-efficient high performance overlays at runtime within a heterogeneous computing platform.

Bibliography

[1] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.

[2] 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.

[3] 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.

[4] Davor Capalija and Tarek 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.

[5] 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.

[6] 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.

[7] 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.

[8] Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

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

[10] Stephen 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.

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

[12] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz 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.

[13] Yun Liang, Kyle Rupnow, Yinan Li, and et. al. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012(649057):1–14, January 2012.

[14] 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.

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

[16] Cheng 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.

[17] 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.

[18] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[19] 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.

[20] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Queue*, 11(2):40:40–40:52, February 2013.

[21] 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.

[22] 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.

[23] Karel Heyse, Tom Davidson, Elias Vansteenkiste, Karel Bruneel, and Dirk 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.

[24] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of cgra. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 9–16, 2014.

[25] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[26] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 503–514, 2011.

[27] P.J. Bakkes, J.J. Du Plessis, and B.L. Hutchings. Mixing fixed and reconfigurable logic for array processing. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 118–125, 1996.

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

[29] 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.

[30] Xilinx Ltd. Zynq-7000 technical reference manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2013.

[31] Khoa Dang Pham, Abhishek Kumar Jain, Jin Cui, Suhaib A Fahmy, and Douglas L Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2013.

[32] Joo 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. January 2002.

[33] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 191–200, 2009.

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

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

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

[37] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for fpga research. In *Field-Programmable Logic and Applications*, pages 213–222, 1997.

[38] Aaron Landy and Greg 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.

[39] G. Stitt, A. George, H. Lam, C. Reardon, M. Smith, B. Holland, V. Aggarwal, Gongyu Wang, J. Coole, and S. Koehler. An end-to-end tool flow for FPGA-Accelerated scientific computing. *IEEE Design and Test of Computers*, 28(4):68–77, August 2011.

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

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

[42] Sivaram Gopalakrishnan, Priyank Kalla, M Brandon Meredith, and Florian Enescu. Finding linear building-blocks for rtl synthesis of polynomial datapaths with fixed-size bit-vectors. In *Proceedings of the International Conference on Computer-aided design*, pages 143–148, 2007.

[43] D Bini and B Mourrain. Polynomial test suite, 1996. *See http://www-sop. inria. fr/saga/POL.*

[44] Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, Zachary Marzec, Preeti Agarwal, Chris Frericks, Ryan Cofell, Jesse Benson, and Karthikeyan Sankaralingam. Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation. *Energy (mJ)*, 5(10):15, 2015.