Framework for Translation of C/C++ Applications on Reconfigurable Computing Systems

THESIS

Submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

ASHISH MISHRA

ID No. 2009PHXF038P

Under the Supervision of

Dr. Kota Solomon Raju

Dr. Abhijit Rameshwar Asati

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
PILANI - 333031 (RAJASTHAN) INDIA
2016

Framework for Translation of C/C++ Applications on Reconfigurable Computing Systems

THESIS

Submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

ASHISH MISHRA

ID No. 2009PHXF038P

Under the Supervision of

Dr. Kota Solomon Raju

Dr. Abhijit Rameshwar Asati

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
PILANI - 333031 (RAJASTHAN) INDIA
2016

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisor, Dr. Kota Solomon Raju, for giving me a valuable problem definition and laying the basic foundation for my research work. He has inspired me throughout these years and pooled in valuable ideas whenever I was struck. His immense knowledge, insights and indispensible suggestions always inspired me to reach a new level. Without his support and guidance, my thesis work would not have been possible. I am very grateful for his guidance and for being a phenomenal advisor.

Secondly, I would like to express my gratitude to my co-supervisor Dr. Abhijit R. Asati for his valuable discussions in the domain of EDA VLSI design algorithms. His patience to understand the problem crux makes him special. His continued support and mentoring led me to re-structure my writing and presentation ability.

I take this opportunity to thank Prof. Souvik Bhattacharyya, Vice-Chancellor BITS, Pilani and Prof A. K. Sarkar, Director, BITS, Pilani (Pilani Campus) for providing me the necessary infrastructure, facilities and constant inspiration. I also acknowledge the kind support from R. N. Saha (Director, Dubai campus), Prof. G. Sundar (Director, Hyderabad Campus), Prof. G Raghurama (Director, Goa Campus), and Prof. S. K. Verma Dean (Academic Research and Development). I would also like to extend my sincere thanks to my doctoral advisory committee (DAC) members, Prof. S. Gurunarayanan and Prof. Sudeept Mohan for carefully going through my thesis drafts and helping me learn the various skills of academic research. I also thank DRC convener Dr. Abhijeet Asati and other DRC committee members for their time and insightful comments. I am also thankful to Dr. Navneet Gupta (HOD, Department of Electrical and Electronics Engineering), Prof. V. K. Chaubey, Prof. Anu Gupta, Prof. Sundar Balasubramaniam, Dr. Hari Om Bansal, Dr. Hitesh Datt Mathur, Dr. Dheerendra Singh, Dr. Anantha K Chintanpalli and Prof. Surekha Bhanot. A big thanks to the rest of the people that are or have been with BITS-Pilani during my time here so far, you all make this a great place to work at.

I am also fortunate to have the blessings of my Guru Babuji Maharaj and My father Shri. Hari Mohan Mishra, which I gratefully acknowledge. My deepest gratitude goes to my extended family: my sisters Roli di and Jaya di, my brother Anurag da, my brother-in-laws Mr.

Ameetabh Tyagi and Mr. Vinay Kumar Pandey. I have no words to thank my wife, Mrs. Lucky Mishra, who has always stood beside me like a rock whenever I had a difficult time. She has been a constant source of motivation and at times even sacrificed her professional time to help me grow further. My son Adyut never complained about not being able to spend enough time with him. Lastly, I express my thanks to all those who directly or indirectly contributed to the completion of this work.

Date: Signature:

Place: Name: Ashish Mishra

ABSTRACT

Embedded Systems are an integral part of the stupendous technological advances as they meet the automation, monitoring and computational demands of the electronic industry. Most of the times, these systems are transparent to the user and do their defined work eternally such as telecommunication systems. Further advancements in these systems have led to the migration of the applications from manual operation to fully automated behavior; major examples of such domains include automotive industries, artificial intelligence applications, remote seasoning, surveillance etc. The stringent system requirements of such systems have pushed the designers to explore innovative design methodologies that deliver higher performance, occupy lesser area and consume minimum power. Modern embedded systems are expected to perform extensive computations on a streaming data set with various degrees of constraint. The design exploration space for such systems is huge; starting from a complete software (SW) implementation on various platforms (general purpose processor/digital signaling processor (DSP) processor/superscalar processor/very long instruction word (VLIW) processor/multiprocessor systems) and culminating as a complete hardware (HW).

The HW implementation includes application specific integrated circuits (ASIC) or field programmable gate arrays (FPGA) based design flow that delivers better performance as compared to the SW implementation due to a dedicated datapath and is thus used in many critical applications. In exploration of design space, the system specifications may further be optimized by partitioning a design into HW and SW. To exploit the potential of FPGA based design, the research community is also targeting different methodologies such as HW-HW partitioning and usage of partial reconfiguration concept.

FPGAs are HW programmable chips on which any digital function can be synthesized, tested and prototyped. They are an attractive choice for the designer due to less non-recurring cost and less time-to-market as compared to the ASIC. These chips contain configurable resources that are used to implement any application in HW. But as they contain a limited amount of resources on the chip, hence the amount of resources consumed when implementing any digital function on a chip, should not exceed the area constraint.

The concept of system design using partial reconfiguration design poses a number of challenges including optimization of partial reconfiguration method, reducing reconfiguration latency, scheduling of HW, SW and reconfigurable HW. To solve these problems, design space automation is one of the key challenges. A general objective of this thesis is efficient

deployment of embedded applications on reconfigurable computing systems. This thesis work targets one of the design space automation requirement which is to map C/C++/HDL/DFG application code onto general purpose hardware such as FPGA using static and partial reconfiguration approach. The thesis proposes a novel design methodology to automate the mapping of C/C++/HDL/Dataflow graphs application code by converting it into system level blocks. This includes design and development of an algorithm that generates coarse-grain functional blocks from the fine grain instruction level i.e. the proposed algorithm clusters an instruction level specification to high-level abstraction, while optimizing the latency and data communication among the combined functional blocks. To achieve an optimized algorithm for the above purpose, various design flows exist and improving them to suit our proposed algorithm is another important contribution of this thesis work.

To map the application on the system level architecture, (assuming general purpose hardware in FPGA with built-in processor within it), there are traditionally **two methods:** HW-SW co-design **(method-1)** and intellectual property (IP) core as HW implementation **(method-2).** Both the methods have been extensively investigated, enhanced and compared throughout the dissertation. The co-design method achieves the required system parameters by implementing the system partly in HW and partly in SW. Though the first method is well established, automation of HW-SW co-design of an application is limited to manual design flow. This work proposes an automated co-design methodology of an application using genetic algorithm. The proposed design flow supports HW as functional sub-blocks, where ready-made IP blocks of critical part of the application are used. The proposed design flow uses time profiling and synthesis data to guide a genetic algorithm to generate good solutions.

In the co-design flow, a commercial high level synthesis tool has been used in this work for estimating the amount of resources consumed by the program. In addition, in this research a new approach has been proposed to estimate the amount of resources without synthesizing the program. For this purpose, the SW specification in C/C++ and a compiler has been used to generate control flow graph (CFG) which is a commonly used format for hardware generation. This process of mapping the logic from high level languages (HLL) to HW requires resource estimation at the granularity level of instruction. A resource table depicting the resource consumed by each operator found in the low level virtual machine (LLVM) compiler has been computed by coding in Verilog for Virtex-5 series. Using this library, a resource estimation algorithm based on CFG operator has been proposed and verified. In many cases if the entire specification is migrated to HW in the form of accelerator, it may consume significant

resources. Hence, optimization techniques applied at the HW specification are required to satisfy system design parameters (area-delay product). Various optimization techniques have applied and compared for area and latency trade-offs.

The second method for system design is to partition the specification in HW sub-modules and execute them as intellectual property (IP) core. Another major contribution of this work is proposing and testing the second methodology extensively. A dataflow graph specification has been used for verification and development of algorithms in this methodology.

The second method can further be divided into sub-methods: static **(sub-method-1)** and dynamic **(sub-method-2)** scheduling of HW blocks on FPGA. **In sub-method-1,** a partitioning algorithm based on graph isomorphism has been proposed, which takes dataflow graph as the input and returns partitions based on constraints. These clusters are interfaced as static IP cores for reusability. Multiple clusters are possible which are similar in nature and must be interfaced as separate HW blocks in a system-on-chip (SOC) design flow and hence we have named this flow as HW-HW design flow.

In sub-method-2, dynamic scheduling of HW blocks based on partial reconfiguration flow has been investigated. A genetic algorithm based approach for optimizing the partitioning process and generating the best partitions based on defined objective function has been proposed. Using partial reconfiguration design flow the partitions are bound to a specific area on the chip known as partial reconfiguration (PR) region using floor planning software Xilinx PlanAhead. The results highlight the pros and cons of this technique by comparing the time required in SW and PR flow. The flow re-uses the same Silicon area at different execution times so that the application fits into the minimum area possible. This method provides a robust technique for implementing any application on FPGA irrespective of the quantity of resources it consumes. This opens new channels for exploiting PR design in future products where the reusability of HW would be possible. This will allow the development of new algorithms at run time at the user level. The design flow will further boost the automation of development and facility offered by electronic design automation (EDA) tools.

In order to assess both the methodologies, discrete cosine transform which is a computationally intensive algorithm has been used for comparing both the approaches on ML507 board. The results clearly show the design flow of isomorphic clustering is much better than any other flow.

The thesis proposes novel methodologies for system level integration in simulation as well as in experimentation. This research gives us an insight to HW-SW design space exploration of

an application and provides a foundation for future research in this domain. It emphasizes on the anticipated constraints and challenges in system design methodologies by presenting various optimal solutions. The results of this work offer a wide spectrum of design space implementations to the designer with area-delay parameter as the criteria to choose among them.

TABLE OF CONTENTS

ABBREVIATIONS

ASIC	Application specific integrated circuits
AXI	Advanced Microcontroller Bus Architecture
BRAM	Random access block memory
CDFG	Control dataflow graphs
CFG	Control flow graph
CLB	Configurable logic block
DAG	Directed acyclic graph
DCT	Discrete cosine transform
DFG	Dataflow graph
DDR	Double data rate
DFG	Dataflow graphs
DMA	Direct memory access
DSP	Digital signal processing
EBB	Extended basic block
EDA	Electronic design automation
FIFO	First-in-first-out
FM	Fiduccia-Mattheyses
FPGA	Field Programmable Gate Arrays
GA	Generic algorithm
HAL	Hardware abstraction layer
HDL	Hardware description language
HLL	High level languages
HLS	High Level Synthesis
HW	Hardware
I/O	Input Output
ICAP	Internal configuration access port
IP	Intellectual Property
IR	Intermediate representations
KHz	Kilo Hertz
LLVM	Low level virtual machine
LUT	Look-up-table
MRTG	Modular random task graph generator

OS	Operating Systems
Pc	Crossover probability
PLB	processor local bus (PLB)
Pm	Mutation probability
PR	Partial reconfiguration
PRM	Partially reconfigurable modules
PRR	Partial reconfigurable region
RCS	Reconfigurable computing systems
RTL	Register transfer level
SA	Simulated annealing
SDK	Software Development Kit
SOC	System-On-Chip
SRAM	Static random access memory
SUIF	Stanford universal intermediate format
SW	Software
UART	Universal asynchronous receiver transceiver
UCF	user constraints file
VLSI	Very large scale integration
XPS	Xilinx Platform Studio
XPS	Xilinx platform studio

LIST OF UNITS

MHz	Mega Hertz
ns	Nano seconds
µsec	Microseconds
KB	Kilobyte
MB	Megabyte

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

This chapter presents an introduction to embedded system design, FPGA based System-On-Chip design, reconfigurable systems, and partial reconfigurable systems. After presenting the design flow for such systems, the research gaps are identified and the objectives are outlined. The chapter ends with the discussion on the thesis organization in the form of a flow graph.

1.1. Embedded System Design

Electronic systems have become ubiquitous and pervasive because of the possibilities of smart system designs which encompasses computation, communication and sensing. Such systems are built with high speed processing components, complex interfaces such as a camera and wired/non-wired communication. Embedded systems, which are designed to achieve application specific requirements, have further boosted the lifespan of such systems. These systems are designed with miscellanea of components such as Microcontrollers /Microprocessors/System-on-chip (SOC), sensors/actuators, input/output devices, storage elements and accelerators. In designing such systems, the characteristics which are of prime concern are good real-time performance, low monetary cost, low power design and less time-to-market. These systems are programmed with applications written in software (SW) for a chosen hardware (HW), making the design cycle rigid in terms of hardware parameter variations. Fig. 1.1 shows that a conventional design cycle of embedded systems, in which selection of HW and then writing the required SW [1.1] is sequential in nature.

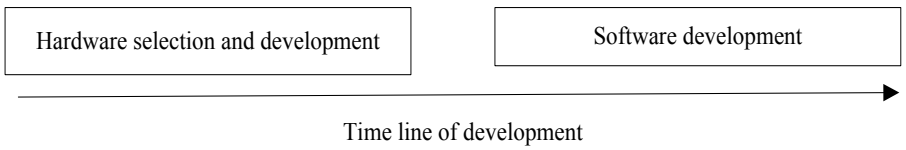


Figure 1.1: Embedded development cycle

The modern embedded systems [1.2, 1.3] which are used to design complex systems such as surveillance, object tracking, machine learning, etc. require more processing demand for intensive computational part of the application. The conventional embedded design flow starts by selecting the processing element, writing the SW for application, cross-compiling and burning the image created. Such a SW implementation for intensive applications offers a high degree of flexibility, but exhibits poor performance due to sequential execution on embedded

platforms like microcontroller. High-performance computing and parallel computing paradigms of computer science aim at achieving the performance by exploiting parallelism as multithreaded application which is bound to run on multi-core systems. Such computing is more applicable to general purpose machines where operating system support, OpenGL like library compilation and costly platforms are the backbone.

Signal processing in SW has been further strengthened by usage of digital signaling processors (DSP) which have a tailored data path for better computation. Traditionally, digital signal processors have been used in many Digital Signal Processing applications [1.4], mainly due to the short development time, lower power consumption, and lower cost. Such processors are still the choice for applications requiring computation. However, over the decade, many algorithms which were implemented in SW were migrated to HW because of the higher performance gain as compared to DSP processors.

The HW implementation, which requires the development in a hardware description language (HDL) and code verification, has the lowest degree of flexibility, but shows the best performance due to the possibility of extraction of parallelism. All those operations which are independent in SW can be executed at the same time in HW, allowing the design of a concurrent structure. Such applications which have been migrated from SW to HW for achieving performance are known as accelerators [1.5, 1.6]. Such accelerators are conventionally designed by two different approaches. The first approach is to design the HW as chip using ASIC methodology. The common examples of such chips are graphics cards, network cards, router, gaming gazettes etc. The performance gain in this process is enormous, but the high non-recurring engineering cost and long development cycle makes it attractive only for voluminous production.

The second design approach is based on designing the system using Field Programmable Gate Arrays (FPGA) which allows the accelerator to be designed, tested and interfaced to the processor at the user level. The FPGAs do not deliver as much performance gain as compared to the ASIC based design flow [1.7], but chip fabrication is bypassed which saves considerable design cycle time and thus are a favored choice [1.8]. The FPGAs chips are designed with configurable cells and can be used to implement any digital function. The user level HW programmability makes these chips an attractive choice for a large class of applications. The FPGA based design offers an embedded development environment platform, which has a processor around which the systems is designed and any HW intellectual property (IP) core can be interfaced to an underlying bus.

This platform allows the design to be implemented in SW, HW, HW-SW (HW-SW co-design) or HW-HW [1.9] thus providing a designer to discover various HW/SW trade-off. The comparison of HW and SW on various parameters such as execution, resources, etc. is shown in Table 1.1. Both HW and SW consume space on the FPGA chipset as configurable cells and memory.

Table 1.1: HW and SW comparison

Parameter	**HW**	**SW**
Execution	Concurrent	Sequential
Development	Flexible/Rigid	Very Flexible
Resources	Uses spaces	Uses Memory
Performance	Fast	Slow
Time to Market	More	Less

HW-SW co-design [1.10] flow requires the identification of candidate functions suitable for HW implementation and its interface to the remaining part of the application. Whereas, HW-HW design flow requires the design to be partitioned into sub-designs and executed in the predefined areas. The HW partitions can be interfaced and executed statically or dynamically, giving rise to two different approaches. When a part of the chip can only be used by an application at any point of time, we call it static approach. In the case of dynamic, the same area can be used by multiple applications at different point of times. This feature has now become feasible with the advent of partial reconfiguration, which allows a part of the chip to be reconfigured even when the system is running. HW-HW design flow implemented with partial reconfiguration support is thus a new emerging trend and has not been extensively tested on HW [1.7]. The increasing trends towards high performance, reusability and low power design have encouraged the researchers to add new dimensions to system design under various degrees of constraint. The topic of this research work is thus inspired from comparing the SW and HW design flows.

1.2. FPGA Based System Design

1.2.1. Introduction to FPGAs

The programmable chips came into existence due to the advent of general purpose processing [1.7] in which a HW code is burned into a programmable ROM chip referred as basic input output systems (BIOS). Programmable logic devices which include programmable logic array, programmable array logic and generic array logic [1.11] are used for developing glue logic in prototyping designs such as address decoder, error detection and correction, etc. With the

advancement in very large scale integration (VLSI) design flow, the density of logic gates increased, making it possible to bring more complex design into HW, giving rise to the generation of FPGAs.

The FPGAs are programmable VLSI chips that can be used to implement any digital function without changing the on chip HW resources. FPGA chips contain configurable memory cells, which are either anti-fuse or memory based. An anti-fuse based bit cell uses irreversible thin (gate) oxide breakdown mechanism to program a bit and it is one time programmable in nature. The examples of such FPGA are Actel based chips memory based FPGAs use SRAM, FLASH, EEPROM based technology [1.7]. Memory based technology offers re-programmable dimension of these systems by allowing them to be programmed almost unlimited times. The SRAM based technology holds the configuration in the memory only when the power is up; however, in the case of FLASH based technology the configuration is permanently stored in the memory. The major market players in the FPGA domain are Xilinx [1.12], Altera [1.13], Lattice [1.14], Actel [1.15], with Xilinx capturing 45-50% of the market share and supporting various embedded platforms based on the processor like PowerPC, ARM etc. The remaining discussions in this thesis are with respect to FPGAs manufactured by Xilinx. Xilinx offers a plethora of electronic design automation (EDA) tools for complete systems design such as Xilinx-ISE, Xilinx Platform Studio (XPS) [1.16], Software Development Kit (SDK) [1.17], System Generator [1.18], PlanAhead Software [1.19], Chipscope Debugging tools [1.20] and Vivado-HLS [1.21].

1.2.2. FPGA Architecture

A generic FPGA architecture [1.22] can be classified into three parts: a programmable logic element, a programmable interconnection network and set of inputs/outputs. A logic element contains a combination of SRAM based Look-up-table (LUTs), multiplexers and flip-flops as shown in Fig. 1.2 [1.23, 1.24]. The functionality is implemented by writing the output in the LUT. Suppose we need to implement the function $f(i, j, k) = \sum (0,1,2,5,7)$. The table stored in the LUT will be *0,1,1,0,0,1,0,1*. The logic element can give a combinational or registered logic by using a flip flop. The output *y* is a combinational logic, while *q* is a registered output. The *d* input can be a direct output to *y* or *q*.

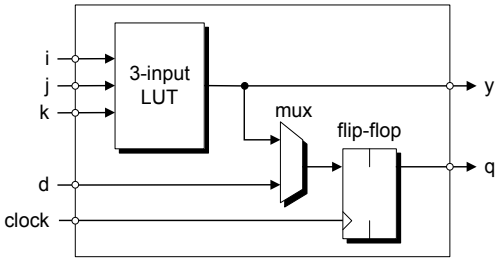


Figure 1.2: Logic element

The digital functions in FPGAs are realized using LUTs, which are a series of flip-flops and multiplexers that store the output function. These elements are put together to form a configurable logic block (CLB) that contains a hierarchical structure of LUTs. Fig. 1.3 shows the internal organization of 2-input LUT which has four flip-flops and three two input multiplexers.

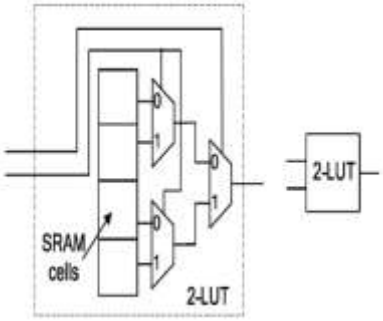


Figure 1.3: Look-Up-Table

The synthesized design in FPGA is mapped onto different logic blocks and connection is made by the interconnection network, which contains an array of switch matrix and wires. The wires are typically organized in wiring and routing channels. The channels provide wires of varying length such as short wire, medium length wire and global wires. Modern FPGAs contain a heterogeneous architecture consisting of fine grain and coarse grain components. The heterogeneous coarse grain components can be soft, firm or hard IP cores [1.25]. The soft-core IP is synthesizable RTL level designs which include embedded soft-core processors, bus controllers, memory controllers etc. that can be configured as per the requirement. Hard IPs are pre-placed components that can be used as black boxes and have an optimized layout. These include embedded hard core processors, hard macros such as ethernet, analog to digital converter, digital to analog converters, RF modules, digital clock manager, phase locked loop, high-speed transceivers etc. Firm IP cores are provided as parameterized RTL description, so that designers can optimize the cores for their specific design needs. Fig. 1.4 shows the organization of the Virtex-5 Xilinx FPGAs depicting LUT slices, processor, Block RAM,

FIFOs, Digital clock manager, etc. PowerPC processor is preplaced and is an example of hard IP whereas BRAMs are firm IP cores.

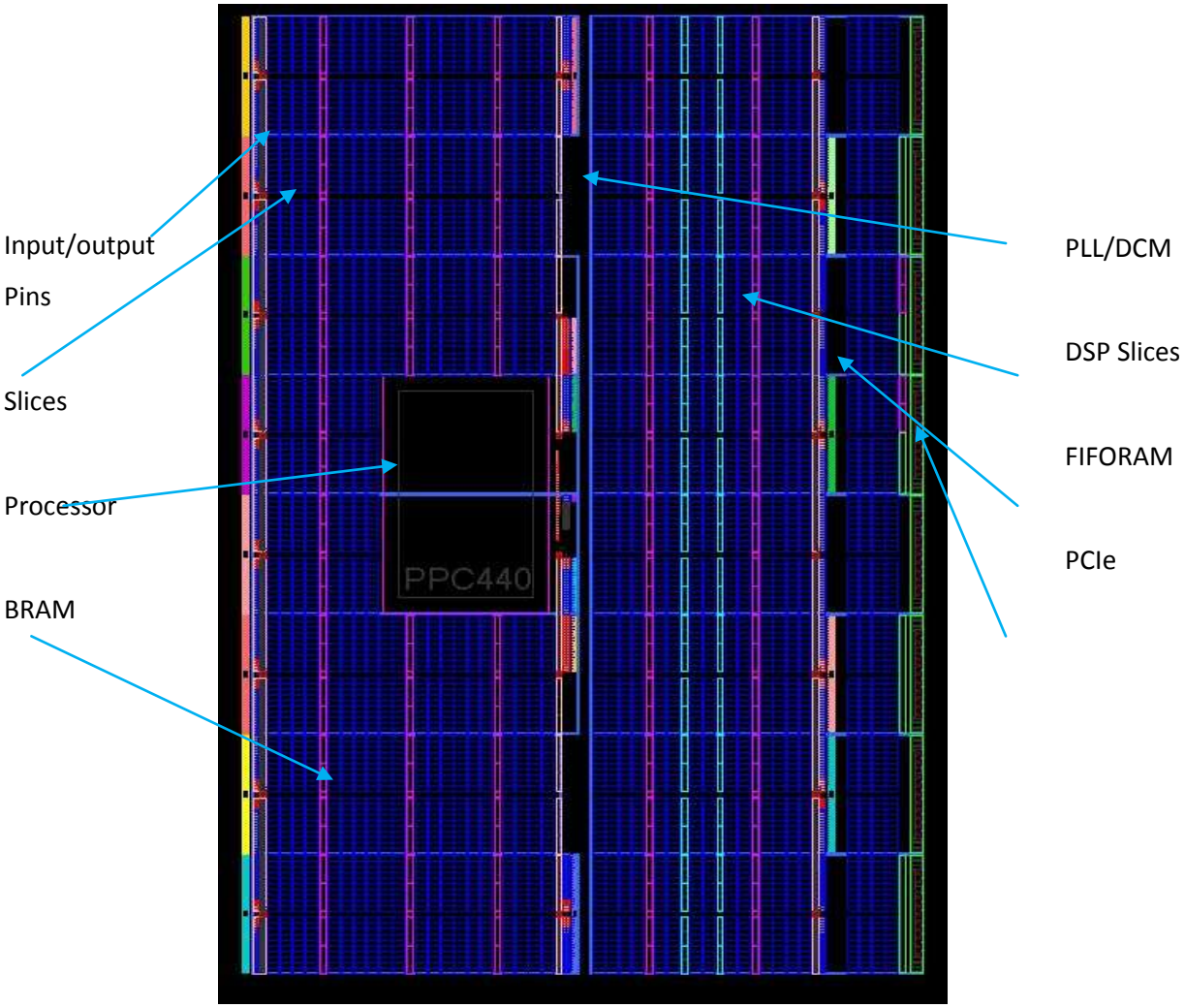


Figure 1.4: FPGA architecture

1.2.3. System-On-Chip Design

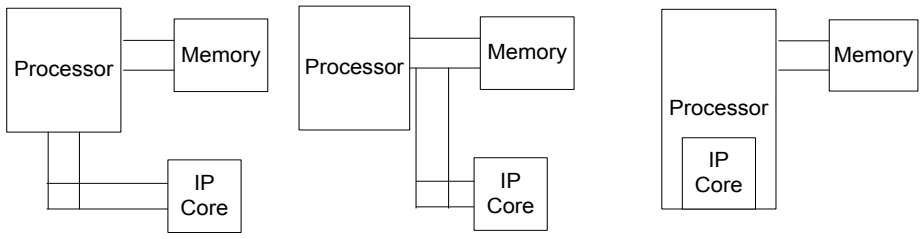
The integrated circuits had become increasingly complex and expensive, which has led to the emergence of new designs and reuse methodologies and it is collectively referred to as SOC. The SOC platforms [1.25] are not only a chip, but a combination of IP cores, software support and integrated platform. These platforms usually include embedded processor, ASIC logics, analog circuitry, embedded memory etc. Their SW includes: operating system, compiler, simulator, firmware, driver, protocol stack, integrated development environment (debugger, linker) and application interface (C/C++, assembly). There are several benefits of integrating a large digital system into a single integrated circuit. These include lower cost per gate, lower power consumption, faster circuit operation, more reliable implementation, smaller physical size, and greater design security. The principal drawbacks of SOC design are associated with the design pressures imposed on today's engineers, such as time-to-market demands,

exponential fabrication cost, increased system complexity and increased verification requirements.

1.2.4. FPGA Based System-On-Chip Design

Common architectures and supporting technologies are called platform based design. These platforms offer the designer IP libraries and tools support, bus support, mixed signal blocks, software component (e.g. Driver, OS). Examples of such platforms are Xilinx EDK, Altera SOPC builder, programmable system on chip (PSoC) [1.26] etc. The FPGA based system SOC design offers the designer a HW and SW development platform. Each of these platforms are tightly integrated, allowing the designer to create the HW and develop its SW, hence facilitating rapid system prototyping. An HW definition of an application developed by the designer is usually known as user defined intellectual property (IP), which FPGA based SOC usually supports.

In configurable SOC based design, different techniques are used for interfacing the IP depending on the speed requirement [1.27]. Examples of interfaces are buses, peer-to peer link to the processor and modified processor data-path. Fig.1.5 shows the IP interfaced in three ways: loosely coupled, medium-coupled and tightly coupled. In case of loosely coupled system, IP can be interfaced with a low speed bus. Medium coupled IP are interfaced with high speed bus. It is also possible to change the datapath of the processor and define a new operation as an instruction in tightly coupled systems. Such tightly bound systems come in the class of application specific integrated processor (ASIP) and is a popular way of implementation of accelerators. Designing such ASIP, requires the complete new definition of software toolchains requiring a dedicated team for designing, coding and verification of the application.



a) Loosely coupled IP (b) Medium coupled IP (c) Tightly coupled IP

Figure 1.5: Various IP interfacing techniques

Xilinx EDK design flow allows the development of user defined IP and generates its driver for usage in SW development in the SDK. The developed IP is wrapped around the bus wrapper

automatically which can communicate with the processor through slave registers, first-in-first-out (FIFO) memory or local memory. Medium coupling is allowed through the fast simplex link [1.28].

1.3. Introduction to Reconfigurable Computing Systems (RCS)

A common alias for FPGA based systems is reconfigurable computing systems. They can be defined as the study of computations involving reconfigurable devices, which includes architecture, algorithms and applications. They involve computation in space and time, using hardware that can adapt at the logic level to solve specific problems. Reconfiguration is the process of changing the behavior of the systems to execute various configurations. With the reprogrammable feature offered by FPGAs, it became possible to change the HW, giving rise to the emergence of the RC systems. This reconfiguration in FPGAs can be temporal or spatial which means the functionality can be changed by reconfiguring structure area with more than one functional block and scheduling in time. The reconfiguration of the device can be done statically or dynamically, allowing the design flow to be adaptable as per the requirement of the applications.

1.3.1. Dynamic Partial Reconfigurable Systems (DPRS)

A static random access memory (SRAM) based FPGA as discussed in section 1.2.2 is a two layer device consisting of configuration memory layer and logic layer. The lower layer is the configuration memory layer in which the configuration data is stored. The upper layer called the logic layer consists of the logic blocks and interconnection that forms the system architecture.

After choosing the required HW along with specific processor (either hardcore or soft core processor) system configuration file is prepared that is used to program the SRAM cells. The loading of configuration can be done either statically or dynamically [1.7] and hence are known as reconfigurable computing systems (RCS). The static approach known as compile time reconfiguration works by inactivating the running systems and loading a new configuration file. In the dynamic case which is also known as run-time reconfiguration, the configuration is loaded while the rest of the system blocks are running. This allows configuring a selected part of the chip by using specialized hardware. Hence, dynamic RCS systems are also referred as partial dynamically reconfigurable systems and are created by defining partial reconfigurable region (PRR) using a floor-planning software such as Xilinx PlanAhead. The concept of partial reconfiguration allows multiple configurations to be swapped in and out of the hardware independently. The applications to be reconfigured at run-

time are known as partially reconfigurable modules and are statically bound to the PR region, allowing them to run only in the defined partially reconfigurable modules region. Fig. 1.6 shows one such PRR region and two partial reconfigurable module (PRM) named as *add* and *mult* bound to this region.

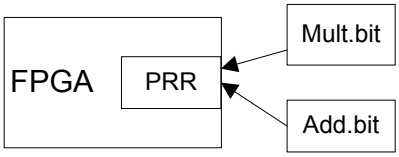


Figure 1.6: FPGA based system with one PR region with two PRM mapped

The PRM modules are compiled and partial bit files for each functional block configuration are moved to the memory available on the respective board such as compact flash, double data rate (DDR) or platform flash providing a database center for partial bit files. At run time, these files are loaded and executed by static scheduler, which calls the configuration in dependency order, stores the intermediate results and executes the complete application. The primary concern here is the total time taken starting from loading to executing the given configuration. Thus the idea of partial reconfiguration enables system flexibility in terms of providing more functionality to be performed in the same area by reducing the area and power requirement of the system. Runtime reconfiguration of programmable devices is the state of the art technology available, but is still not applied in the industrial applications due to lack of sophisticated and automated tools. To exploit the potentiality of RC systems, development of user friendly tools is required.

1.3.2. Advantages of Partial Reconfiguration

The advantages of dynamic partial reconfiguration are as follows:

i. Partial reconfiguration allows designing a self-adaptive and flexible system, where hardware changes can be rapidly migrated, depending on the applications.

ii. Partial reconfiguration allows designing an intelligent system that manages reconfiguration in order to save power.

iii. Partial reconfiguration enables hardware reuse that allows using more silicon than we pay for.

1.4. Motivation for Current Research

As described in the previous section, FPGAs can be used for both software and hardware design thereby offering a wide spectrum of choice of tradeoffs to the designer. The decision from a pure SW implementation to HW requires certain design parameters comparison based on performance, HW area, power consumption and suitable design flows for FPGAs. Further each layer from SW to HW, spans many optimization techniques, resulting in new dimensions of acceptable functional parameters. This research work has been inspired from such demand and comparative data interrogations from the designers.

The application which requires better performance can be partitioned into HW-SW or HW-HW parts [1.9]. For mapping such a candidate, two solutions are possible: 1) Migrating part of the design to HW and the remaining part to the SW, 2) Dividing the design into HW partitions by implementing and executing them sequentially as well as concurrently. HW-HW partitioning using partial reconfiguration has not been extensively tested on real hardware. Hence the objective of this thesis work is to achieve efficient HW-HW partitioning using static approach and partial reconfiguration flow and bring out the pros and cons of the proposed design flow.

Some of the high level synthesis tools such as Vivado-HLS convert a high level languages (HLL) specification like C/C++/SystemC etc. into HW automatically. But due to this, many of the functions normally consume more HW resources as compared to RTL design. In this scenario, the advantage of automation is nullified as most of the chip area is used. Using partial reconfiguration, more HW reusable functions may be divided into further small design. These functions can be executed temporally using partial reconfiguration concept.

In concise, we identify the following motivational gaps for the current work:

- Though HW-SW co-design flow is established, some of the practical design flows are not offered by the present tools. These practical flows are achieved by design methodology that can assist the designer.
- With the advent of high level synthesis tools like Vivado-HLS, the migration of SW to HW is now accelerated. These automated approaches have not been extensively explored for effectiveness in performance.
- The partial reconfiguration flow, has not been used for HW, HW partitioning problem, hence needs to be compared for its pros and cons.

1.5. Problem Definition and Objectives

Many applications fail to achieve the required performance on the processor as pure SW execution. This problem is solved by migrating a certain part of the code to HW. During this migration the decision making task is: which part of the code to migrate. This answer can only be addressed after an initial implementation of the design has been done on the target device, through which execution profile of the application can be accumulated. This makes the life of the design cycle longer and complex. Hence, what we need is the exact design method that can guide the designer prior to implementation. Such methods requires a proposal of an algorithmic approach to generate design place solutions and guide the designer for area/delay trade-offs.

The specification of an application can be done in various ways such as C language, Matlab, finite state machines, dataflow graphs etc. A dataflow model of computation aims at capturing the data flow and its computations among the various operations. The computation part of any design can easily be defined in such a model. They can be used as input specification and can be easily generated, verified and comprehended. Hence these models are a good choice for examining the proposed algorithms in terms of their correctness and effectiveness. These algorithms demand proposal of efficient partitioning and scheduling of operations in correct order. Such algorithms aim at creation of coarse grain clusters that can be mapped as static or dynamic modules. These clusters should be created in such a way that overall the area-delay product improves.

For estimating the area required for a DFG or a C program, we need to generate its control flow graph. Without the HW generation, making this possible will allow the SW programmer to bypass the HW learning and quickly generate functional parameters.

Hence the objective set for this work is defined as:

1. Design and implementation of automated system design flow for applications defined in high level language with HW-SW co-design concept to design optimal systems.
2. Design and development of an algorithm for estimation of resources consumed by functions described as control flow graphs generated from compiler.
3. Given a dataflow and control flow graph specification, design and implement efficient algorithms to convert fine grain graphs to coarse grain graphs. Create such coarse level graph by finding reusable patterns in the specification.

4. Identification of functional blocks for efficient mapping using a partial reconfiguration approach and schedule the application to exploit the strength of a RC system.

1.6. Thesis Organization

This thesis is organized into six chapters. Chapter 1 presents the introduction to reconfigurable systems and motivation for the thesis work. Chapter 2 discusses the literature survey of system design methodology, partitioning of DFGs, high level synthesis and partial reconfiguration.

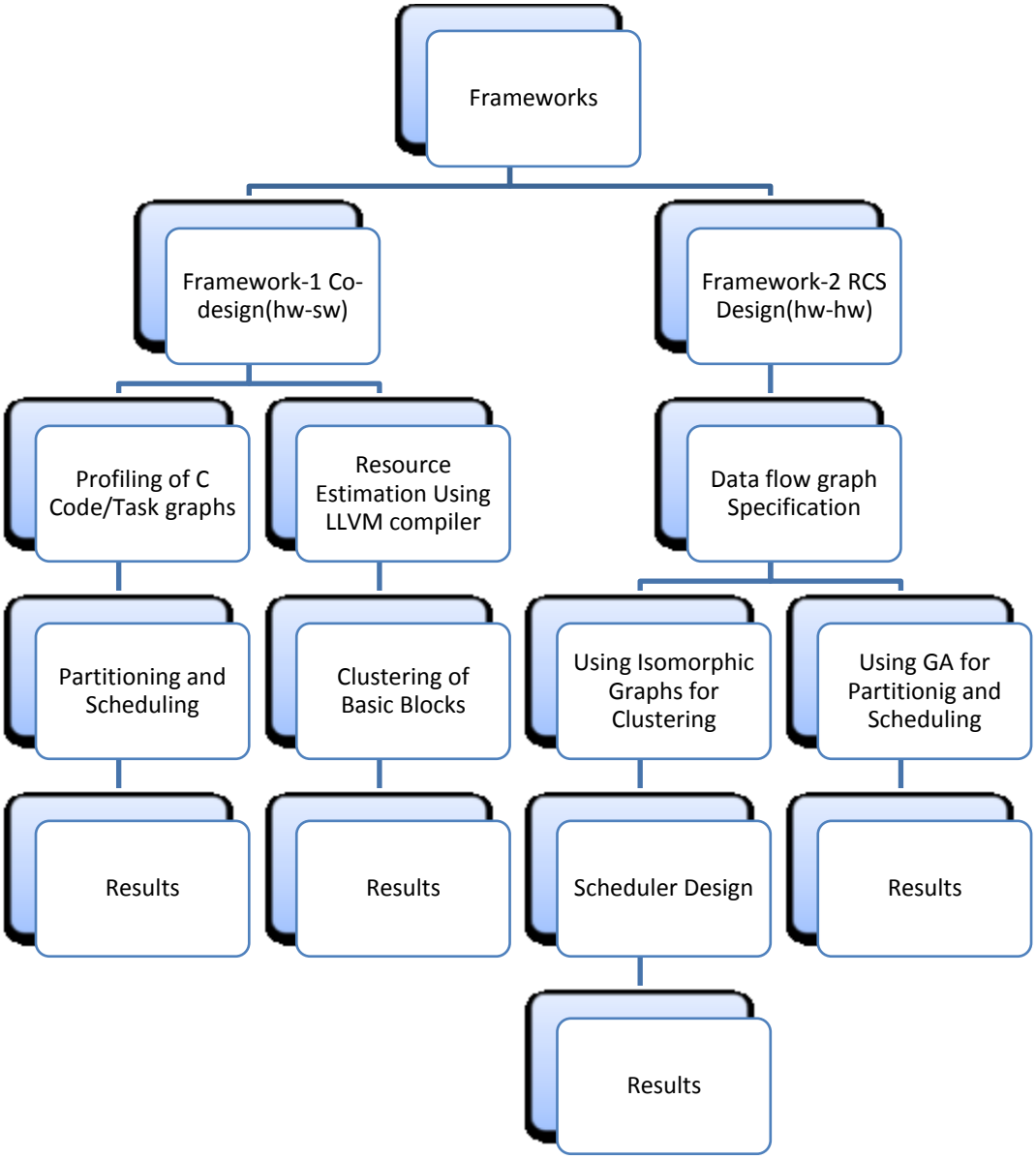


Figure 1.7: Organization of thesis work

Chapter 3 focuses on resource estimation of C/C++ programs and cluster creation. The work further demonstrates how clusters known as extended basic block can be created using an algorithmic approach. A framework for HW-SW co-design using genetic algorithm has been

proposed in chapter 4. Chapter 5 explains the partitioning problem and applies it for creating clusters to HW-HW design flow. Genetic Algorithm (GA) has been used for partitioning and scheduling of RC systems and is extensively covered in this thesis. It also highlights the process of dynamic partial reconfiguration and compares the results of the previous chapters. Chapter 6 gives a summary and draws conclusions on the basis of results obtained. It also presents the future work to be carried out with the existing platform. Fig. 1.7 shows the summary of work done and organization of thesis.

1.7. Conclusion

In this chapter we have discussed the principles of FPGAs, system-on-chip, reconfigurable systems and partial reconfigurable systems. These emerging systems are allowing searching for various HW and SW implementations to maximize the performance while meeting the constraints. This has motivated out research work and inspired us to define the research objectives.

REFERENCES

1.1 Peter Marwedel, Embedded system Design, Springer, Kluwer Academic Publishers, 2003. (chapter-1)

1.2 Peter Barry and Patrick Crowley, Modern Embedded System, Springer Publisher, 2012. (chapter-1)

1.3 Software Design Methods for Concurrent and Real-Time Systems. Hassan Gomaa, Addison-Wesley, 1993. (chapter-1)

1.4 Digital Signal Processors: Architecture, Programming and Applications, B. Venkataramani, M. Bhaskar, Tata McGraw-Hill Education, 2002.(chapter-6)

1.5 SOC W. Wolf, A Decade of Hardware/Software Co-Design, in Proc. of the 5th International Symposium on Multimedia Software Engineering (MSE), pp. 38–43, December, 2003.

1.6 R. A. Klein, A. Moona, Migrating software to hardware on fpgas, International conference on field programmable technology, pp. 217-224, December, 2004.

1.7 Cristophe Bobda, Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications, Springer, 2007. (chapter-2)

1.8 Kuon, J. Rose, Measuring the gap between FPGAs and ASICs, IEEE transactions on computer-aided design of integrated circuits and systems, vol. 26, no. 2, February, 2007.

1.9 J. Straunstrup, W. Wolf, Hardware/Software Co-design Principles and Practices, Springer, Kluwer Academic Publishers, 1993. (chapter-4)

1.10 Patrick R. Schaumont, A practical introduction to hardware/software codesign, Springer international edition, 2011. (chapter-2)

1.11 Robert K. Dueck, Digital Design with CPLD Applications and VHDL, Delmar Cengage Learning, 2nd revised edition, July, 2004.

1.12 http://www.xilinx.com/,last accessed on 15 July 2016.

1.13 https://www.altera.com/, last accessed on 15 July 2016.

1.14 http://www.latticesemi.com/, last accessed on 15 July 2016.

1.15 http://www.microsemi.com/products/fpga-soc/fpga-and-soc, last accessed on 15 July 2016.

1.16 http://www.xilinx.com/products/design-tools/xps.html, last accessed on 15 July 2016.

1.17 http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/SDK_Doc/index.html, last accessed on 15 July 2016.

1.18 http://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html, last accessed on 15 July 2016.

1.19 http://www.xilinx.com/products/design-tools/planahead.html, last accessed on 15 July 2016.

1.20 http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_process_analyze_design_using_chipscope.htm, last accessed on 15 July 2016.

1.21 http://www.xilinx.com/products/design-tools/vivado.html, last accessed on 15 July 2016.

1.22 Wayne Wolf, FPGA based System Design, Prentice hall India, 2004. (chapter-3)

1.23 http://www.xilinx.com/support/documentation/data_sheets/3000.pdf, last accessed on 15 July 2016.

1.24 Clark N. Taylor , FPGA Implementation Details, 2008. (http://ece320web.groups.et.byu.net/CourseNotes/FPGA.pdf), last accessed on 15 July 2016.

1.25 Dimitris Gizopoulos, A. Paschalis, Yervant Zorian, Embedded Processor-Based Self-Test, Springer, Kluwer, 2004. (chapter-4)

1.26 http://www.cypress.com/products/32-bit-arm-cortex-m-psoc. , last accessed on 15 July 2016.

1.27 System-on-a-Programmable-Chip Development Platforms in the Classroom, users.ece.gatech.edu/~hamblen/papers/SOC_top.pdf. , last accessed on 15 July 2016.

1.28 http://www.xilinx.com/products/intellectual-property/fsl.html.

Chapter 2

Literature Survey

Reconfigurable systems are versatile platforms which allow the designer to conceive any kind of SW or HW optimizations and methodologies. It allows achieving the desired system performance by different approaches. These systems contain a processor, on which the SW can be executed and HW fabric on which any accelerator can be designed. Techniques like fixed point vs. floating point arithmetic, usage of simple vs. complex algorithm, language selection etc. allow bringing out the tradeoff in SW implementation. Similarly optimization in HW implementation can be achieved by various techniques like pipelining vs. parallelism, static vs. dynamic scheduling etc. The design of HW accelerators on RCS requires that the resources available on the chip are effectively used to deliver the best performance.

To gain an elaborate insight in the research progress made in the field of chosen problem statement and identify the existing gaps we undertook an extensive literature survey. This enabled us to highlight the problems to be focused on, formalize the solutions and explore the strategy to be adopted to achieve the objectives of the proposed work. This chapter is divided into four sections which cover the literature survey for HW-SW frameworks, high level synthesis, partitioning/scheduling, and partial reconfiguration process.

2.1. Frameworks and Design Methodology

The initial framework for HW-SW partitioning and mapping using manual time estimation and manual synthesis was proposed in [2.1], which demonstrates a complete working example of digital camera using a C specification. This design shows an image compression implementation on a soft core based 8051 complied on FPGAs. A co-simulation testing setup having a cross-compiler and HDL simulator was used for verification. This benchmark was downloaded and simulated in Xilinx ISIM simulator as shown in Fig. 2.1 for verifying the way in which co-simulation is performed. The 8051 VHDL model files and the C code stored in ROM VHDL model is shown in Fig. 2.1. The simulation setup includes the cross compiler and its conversion to a ROM image. The execution in the simulator was possible without a HW implementation allowing the co-simulation to accumulate the performance results.

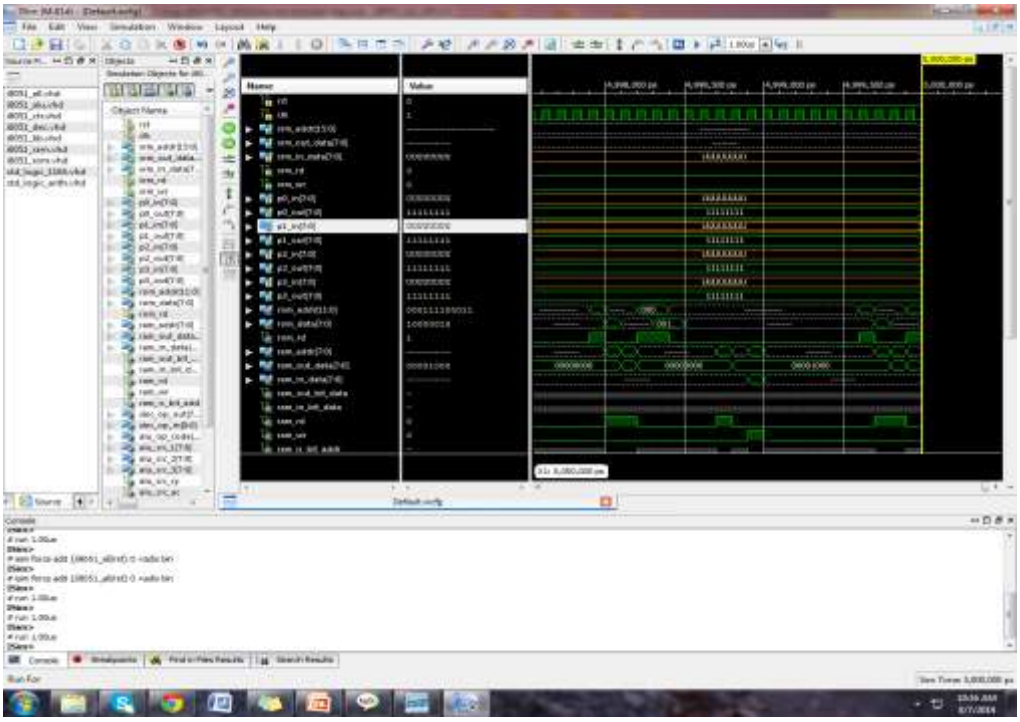


Figure 2.1: Co-simulation of digital camera case study in Xilinx ISIM

The timing constraint kept for the design was given as 1 second, since holding a camera for a prolonged time is frustrating to the user. Hence to achieve the required performance a co-design methodology was adopted and four implementations were written for achieving the target of 1 second. Implementation 1 is pure SW execution and fails to achieve the required time. In implementation 2, one of the modules is randomly chosen and is migrated to HW. In implementation 3 discrete cosine transform (DCT) is implemented in HW and 4 is an implementation of using fixed point arithmetic in DCT block. The Table 2.1 shows the time comparison, which is coming down from 9.1 seconds to 0.099 seconds.

Table 2.1: Three parameters for Digital Camera Case Study on 8051 [Source: 2.1]

	Implementation 2	Implementation 3	Implementation 4
Performance (second)	9.1	1.5	0.099
Power (watt)	0.033	0.033	0.040
Size (gate)	98,000	90,000	128,000
Energy (joule)	0.30	0.050	0.0040

The shortcomings of this design flow were manual partitioning, manual estimation of time and manual conversion of C to HDL, making it time consuming and resulting in a longer design cycle. These guidelines can be adopted only by an experienced designer who knows an in-depth understanding of the platform.

Similar work was done in Google summer of code 2006 [2.2] (a contest organized by Google) on the Leon3 platform, which is from Aeroflex Gaislor company [2.3]. The design flow uses

operating system based design and device driver for migrated IP to HW. This work shows the time profiling on general purpose computer for media application and uses the results for profiling process. The conclusion was that the function reconrefframes waste approximately 60% of CPU-time and should be migrated to HW. The flow does not use algorithmic approach and a C to HDL compiler to compute the resources consumed. Most of the work in this domain lacks the demonstration of time estimation on an embedded target, which can give better estimation results. The structural design is shown in Fig. 2.2 with Theora codec interfaced to APB bus. This work gave an idea of the time profiling on the host and demonstrated the difference in results.

The popularly available tools for co-design are COSYMA, Lycos, Polis, Chinook, Akka, CASTLE/SIR, CodeSign, CoWare and Symphony, The Ode System, COOL, PeaCE [2.4]. Most of these tools are either proposed or not available on-line. The only commercial tool among them is CoWare and its results have not been reported in literature. The common problems encountered in these tools are:

i. ANSI C language is not a standard input in most of them.

ii. The design space exploration for a given design requires manual intervention.

iii. An efficient conversion of SW to HW automatically is missing.

The generic steps required to convert a SW specification into HW [1.10] are:

Step-1: Convert the SW specification into control flow graph using a compiler

Step-2: Generate a data dependence graph from the control flow graph

Step-3: Convert each node in the a combinational circuit

Step-4: Use multiplexer and registers for the combinational circuit design

Step-5: Content the circuit node using the edges in the data dependence graph.

Step-6: Generate a finite state machine for each node in control flow graph.

These steps are applicable to sub-set of a SW language and may fail in cases such as dynamic pointers, file operations, recursion etc. The problems were addressed in [2.5] which present a framework called ASSET where profiling was used to find the critical parts in a C program and a C to HDL compiler was written to automate HW generation. The ASSET framework is based on Stanford universal intermediate format (SUIF) compiler for basic block profiling and HW generation. A HW generation from a C specification is only possible by converting it into a control flow graph. This CFG is parsed and control edges are converted into a finite state

machine and data edges are converted into datapath. SUIF compiler has many drawbacks such as: the development was stopped in early 2000, recursion is not available, lack of built-in pass for optimizations, no community support etc. The work was a first effort to perform the profiling of basic blocks in complier and find the critical part of the program. Though it has not shown the proposed area estimation technique, but it focuses on HDL generation and interfacing, thus lacking to show the framework on a benchmark for generality.

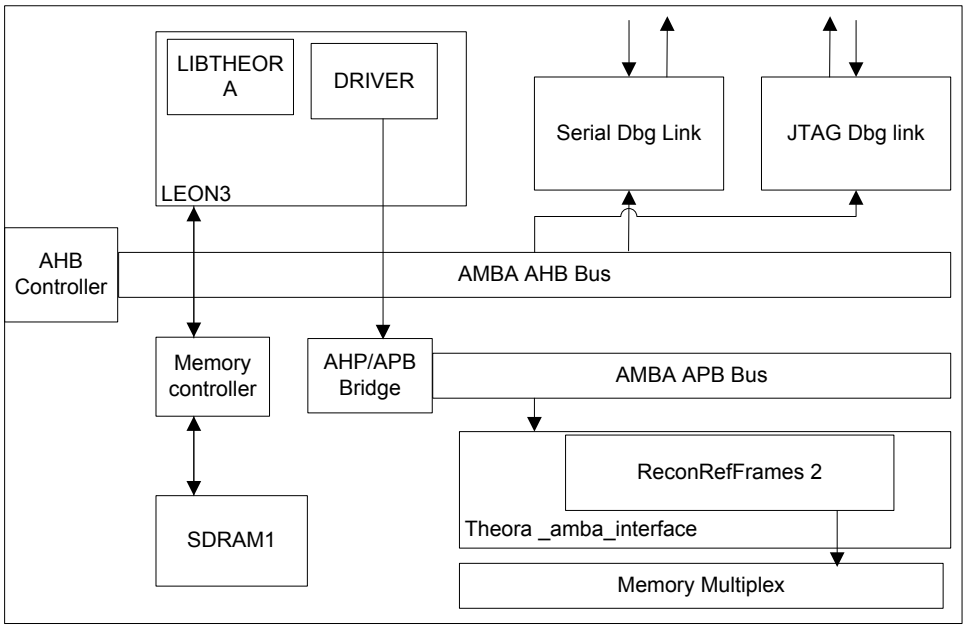


Figure 2.2: HW accelerators in Leon3 platform on Altera StratixII [Source: 2.2]

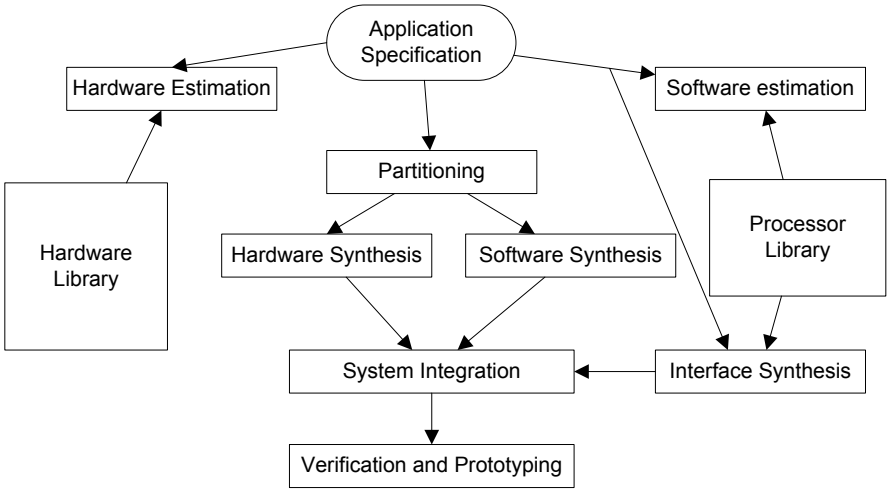


Figure 2.3: ASSET co-design methodology [Source: 2.5]

Fig. 2.3 shows the design flow of work in [2.6] with HW estimation and SW estimation being the building block for partitioning process, followed by system integration on a SOC platform.

The framework fails to show the results of a benchmark, hence cannot be used for comparison purposes.

From the literature survey done, it was identified that presently LegUp tool [2.6] is the only open source platform that allows co-design methodology to be implemented. It is an open source hardware software co-design tool which starts from a C specification and allows the designer to map applications as SW on a MIPS processor or as HW accelerator on FPGA fabric. The design flow is shown in Fig. 2.4, with a C program as input and implementation as HW or SW. It uses a profiling technique [2.7] by proposing a HW architecture which can count the real-time cycles and energy profiles of an FPGA-based soft processor. The tool uses low level virtual machine (LLVM) compiler for preprocessing of C programs and HW migration.

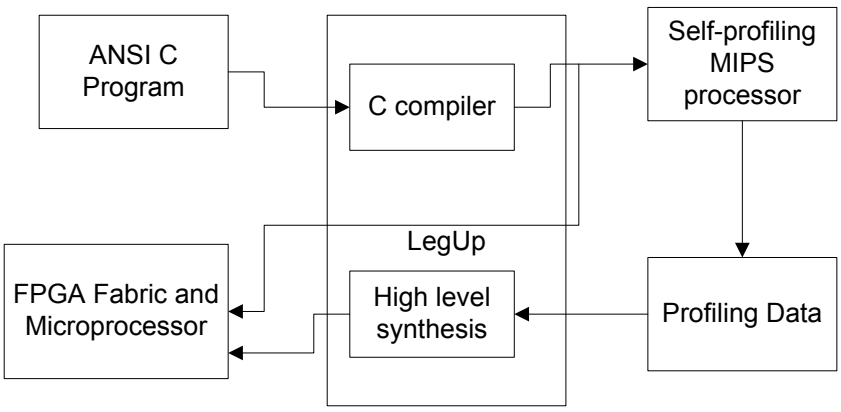


Figure 2.4: LegUp Co-design methodology [Source: 2.6]

The comparative analysis of various benchmarks problems is listed in Table 2.2 [2.7]. For the MIPS processor-SW flow, the processor runs at 74 MHz on the Cyclone II and the benchmarks take between 6.7K-29M cycles to complete their execution. LegUp-Hybrid2 provides a 47% (1.9×) speedup in program execution time vs. SW (MIPS-SW). In the LegUp-Hybrid1 flow, there is 73% reduction in program execution time vs. software (a 3.7×speed-up over software). Looking broadly at the data for MIPS-SW, LegUp-Hybrid1and LegUp-Hybrid2, it is observed that: execution time decreases substantially as more computations are mapped to hardware. LegUp produces heavily pipelined hardware implementations, whereas, we believe eXCite does more operation chaining, resulting in few computation cycles yet longer critical path delays. Considering total execution time of a benchmark, LegUp and eXCite offer similar results. Both of the pure hardware implementations are a significant win over software.

Table 2.2: Performance results of LegUp [Source: 2.7]

	MIPS-SW			LegUp-Hybrid2			LegUp-Hybrid1			LegUp-HW			eXCite-HW		
Benchmark	Cycles	Freq.	Time	Cycles	Freq.	Time	Cycles	Freq.	Time	Cycles	Freq.	Time	Cycles	Freq.	Time
adpcm	193607	74.26	2607	159883	61.61	2595	96948	57.19	1695	36795	45.79	804	21992	28.88	761
aes	73777	74.26	993	55014	54.97	1001	26878	49.52	543	14022	60.72	231	55679	50.96	1093
blowfish	954563	74.26	12854	680343	63.21	10763	319931	63.7	5022	209866	65.41	3208	209614	35.86	5845
dfadd	16496	74.26	222	14672	83.14	176	5649	83.65	68	2330	124.05	19	370	24.54	15
dfdiv	71507	74.26	963	15973	83.78	191	4538	65.92	69	2144	74.72	29	2029	43.95	46
dfmul	6796	74.26	92	10784	85.46	126	2471	83.53	30	347	85.62	4	223	49.17	5
dfsin	2993369	74.26	40309	293031	65.66	4463	80678	68.23	1182	67466	62.64	1077	49709	40.06	1241
gsm	39108	74.26	527	29500	61.46	480	18505	61.14	303	5656	58.93	113	5739	41.82	137
jpeg	29902639	74.26	401328	16072954	51.2	313925	15978127	46.65	342511	5861516	47.09	124475	3248488	22.66	143358
mips	43384	74.26	584	6463	84.5	76	6463	84.5	76	6443	90.09	72	4344	76.25	57
motion	36753	74.26	495	34859	73.34	475	17017	83.98	203	8578	91.79	93	2268	42.87	53
sha	1209523	74.26	16288	358405	84.52	4240	265221	81.89	3239	247738	86.93	2850	238009	62.48	3809
dhrystone	28855	74.26	389	25599	82.26	311	25509	83.58	305	10202	85.38	119	-	-	-
Geomean	173332.0	74.26	2334.1	86258.3	69.95	1232.6	42700.5	67.78	630.0	20853.8	71.56	291.7	14594.4	40.87	357.1
Ratio	1	1	1	0.50	0.94	0.53	0.25	0.91	0.27	0.12	0.96	0.12	0.08	0.55	0.15

The framework does not seem to be very promising from the results as the speedup achieved is very low. Hence is the need is to explore the reasons behind it. The design flows discussed above are manual in nature and are applicable to a very experienced designer. From this, it is concluded that for complete automation there are primarily two requirements.

- The amount of resources consumed by an IP which is migrated to HW should be known.
- A framework which can show the critical part of the design and optimize the code should be used.

2.2. Partitioning and Scheduling

The first phase of the design cycle is the specification of the design which includes its functional and non-functional requirement. The functional requirement describes the behavior of the design and non-functional requirement describe the constraints on the design metrics. The functional behavior of a design is captured in a model such as finite state machine/Task graph or in a language such C/C++ etc. These models/programs are usually converted to abstract data structures such as link list for syntax checking and executable generation. Such linked structures can be depicted as graphs in which each node corresponds to an operation and edge as data communication. Control flow and data flow are the two structures that are available in all languages. Control flow corresponds to the syntax such as loops, goto etc. and data flow corresponds to computation like addition in y = ax + b. For example for the code snippet given below, the CFG is shown in Fig. 2.5.

```
{
```

temp = 0;

for (i=0; i < 5; i++)

temp = temp + i ;

}

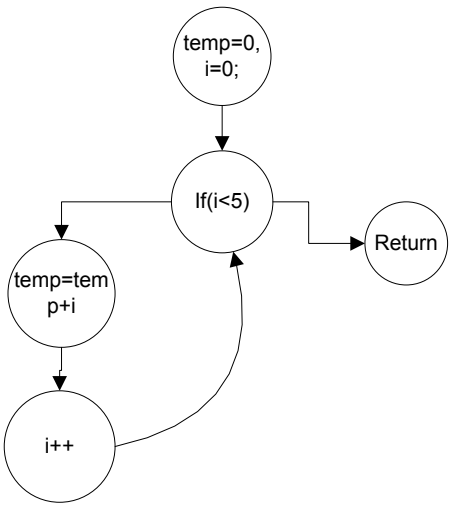


Figure 2.5: Control flow graph for the code snippet

Similarly a task/data flow graphs is modeled as non acyclic graph. For example for the equation $ax^2 + bx + c$, the DFG is shown in Fig. 2.6. It shows the various ways in which the nodes can be clubbed together to create a partition. Now for a given architecture which has multiple processing elements, where these clusters can be executed, finding the best possible solution is the task of partitioning and scheduling stage. The problem of partitioning and scheduling can be as applicable to CFG and task graphs and can be discussed separately.

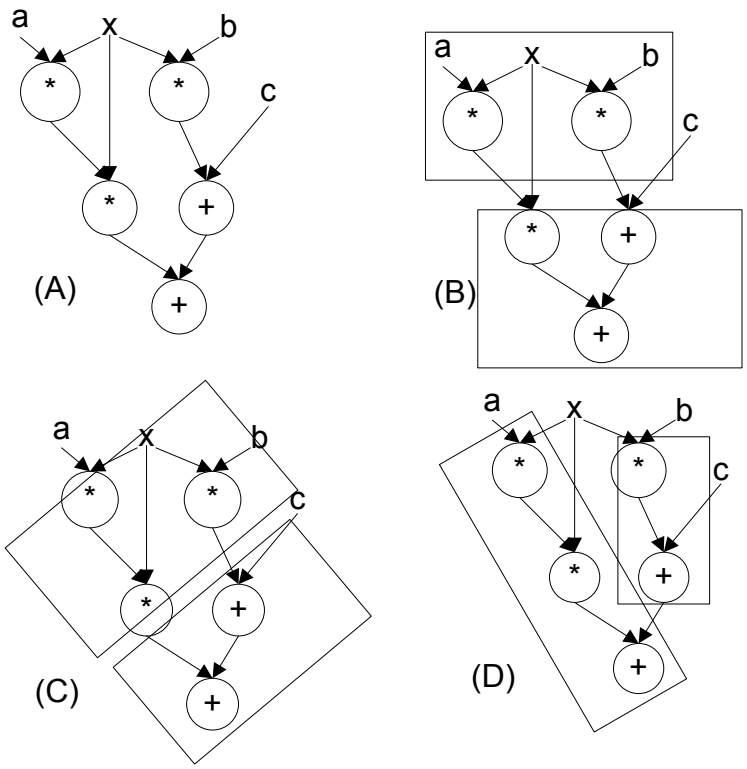


Figure 2.6: Partitioning of graph

The initial work in the development of frameworks for HW-SW partitioning and scheduling of the applications described in C was seen in [2.8]. The work shows the design process with a new language for describing the control flow structure of the program and used LCC compiler for generating the compiler intermediate representation (IR). The framework was applied to customized multi-FPGA systems and hence was not extensively used.

Similar research was done in [2.9], which shows temporal partitioning and scheduling for multi-FPGA systems and clearly demonstrates the mapping of the DFG to FPGAs and reports the reconfiguration time of the executable model. In this approach, partial reconfiguration design flow was not used, but the work was dedicated more to scheduling at the instruction level and a multi-FPGA dedicated architecture was used for experimentation. The work compares the level based vs. non level based scheduling. The algorithms were described to work on DFG model only and are not applicable to complete ANSI-C set. This problem of ANSI-C is addressed in [2.10] which follows a basic block generation using a compiler and proposes an algorithm for area based cluster creation, focusing on overlapping basic blocks for improving performance. The work proposes its own intermediate representation format for the generation of basic blocks and their scheduling. In [2.11] a compiler framework has been proposed to partition and schedule the instructions, but the performance has been shown for defined architectures, which is not general. For making the partitioning process efficient for co-design, a compiler intermediate format has been extended in [2.12] and called as

hierarchical control dataflow graphs (CDFG) which can be seen as clusters for performing parallelism and high level synthesis.

The initial work involving the creation of a library of operators and sharing them in the scheduling process is reported in [2.13], which shows the effect of sharing of operators with respect to time and area but lack the design flow starting from ANSI-C set. The work in [2.14] extends the CDFG model to accommodate control flow mapping to HW easily and shows that a library of operators has been created for mapping. The CDFGs have been converted to hierarchical control flow graphs to exploit the inherent parallelism. The results have not been shown for a benchmark program, hence cannot be used for comparison purposes. Chapter 4 of [1.7] discusses the partitioning and scheduling problem in detail. The LIST scheduling algorithm for RC systems where each node is assigned a priority and partitions are created based on a given area is discoursed. The work lacks to demonstrate a design running on PR region and the consequences of partitioning a design. This gap between the algorithm proposal and its verification inspires this research work to set up the experimental analysis.

The work in [2.15] uses integer level programming with scheduling to find similar patterns as a candidate for partial reconfiguration. The primary purpose of finding the isomorphic graphs is to reduce reconfiguration overhead. This work highlights only the software simulation result and does not compare the real time performance by interfacing an application to the processor bus. The design flow presented hence is not realistic. In [2.16] a design flow for mapping applications on Xilinx FPGA partial dynamic reconfiguration flow using PlanAhead has been extensively shown. The work compares the resource usage for encryption algorithm but lacks to show the timing analysis. The most recent work [2.17] proposes an algorithm for finding a pattern of the configuration to be placed in multiple PR regions. It also shows the theoretical time equation that can used to calculate the reconfiguration time.

Partitioning is a process by which we divide the input specification into sub-sets depending on the constraints like: the number of subsets, maximum vertices in a sub-set and maximum number of edges between the sub-sets. The partitioning technique can be broadly classified into constructive and iterative approaches. Constructive partitioning aims at identifying an initial partition and are greedy in nature. This means such algorithms fail to find the global minima and may stop at local minima. Examples are random selection, cluster growth, hierarchical clustering, Fiduccia-Mattheyses (FM) and min-cut [2.18, 2.19].

The literature in [2.20] discusses the various optimization problems in system level synthesis theoretically which are partitioning and scheduling. To achieve hardware and software performance trade-off, many optimization algorithms have been already proposed, such as

Tabu search and simulated annealing [2.21, 2,22] which applies partitioning to loops, blocks, subprograms and processes, Ant colony optimization [2.23], swarm optimization [2.24] and Genetic Algorithm (GA) [2.25, 2.26, 2.27] which is a stochastic optimization algorithm modelled on the theory of natural evolution. Since its first successful implementation, GA has been used to solve a wide range of problems, such as travelling salesman problem [2.28], real-time problems such as reconfiguration of evolvable hardware (EHW) [2.29], and other optimization problems that have complex constraints [2.30]. Hardware and software co-design using genetic algorithm has been implemented previously [2.31, 2.32, 2.33] on simulated data set which contains various area and delay parameters.

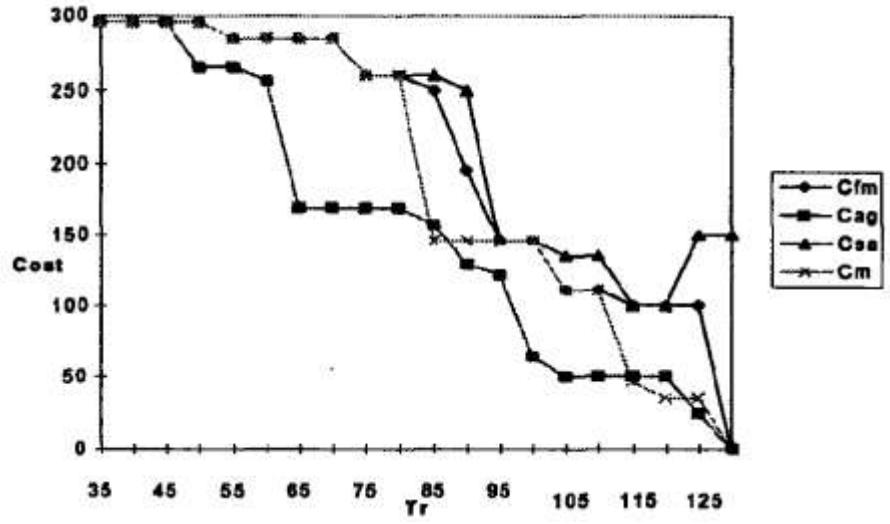


Figure 2.7: Comparative results among FM, GA, SA and MFM for cost vs. time constraint [source: 2.34]

A simulation of a theoretical task graph on HW and SW using GA is compared with approaches like FM and simulated annealing (SA) in [2.34]. The objective function (Objfct) is used to guide the GA and it includes time/cost along with predetermined deadline. Mapping is a process which determines which node will run on what component. The work fails to show the mapping on concurrent architecture (multiple CPU/ASIC). It misses to propose an algorithm which can find the execution time if nodes are concurrently executed on multiple CPUs and ASICs. Fig. 2.7 clearly shows the comparison of four partitioning algorithms which are Fiduccia Mathesys (FM), GA, simulated annealing (SA) and modified Fiduccia Mattheyes (MFM). In FM, the elements that improve the OF are selected and locked. This process is repeated until all the blocks are locked and there are no improvement of the objective function. This method only allows the movement of a block in each step. GA converges quickly as compared to other algorithms and hence is better algorithm as compared to others. MFM algorithm is a particular version of the FM that includes a special attention to

communication costs. Besides it permits the migration of more than an only block. SA works with one solution and moves towards a better solution, while GA starts from a set of solutions.

The running time and the optimal solution generated by GA in dependent on many parameters such as number of iterations, population size, mutation probability (p_m) and crossover probability (p_c). Adaptive probability [2.35] is a technique which is used to guide GA for values of mutation probability and crossover probability depending on the fitness value of the genes for better convergence of the GA. The crossover probability controls the rate at which solutions are subjected to crossover. The higher the value of crossover probability, the quicker are the new solutions introduced into the population. As crossover probability, increases, however, solutions can be disrupted faster than selection can exploit them. Typical values of p_c, are in the range 0.5-1.0. Mutation is only a secondary operator to restore genetic material. Nevertheless the choice of p_m, is critical to GA performance. Large values of p_m, transform the GA into a purely random search algorithm, while some mutation is required to prevent the premature convergence of the GA to suboptimal solutions. Typically p_m, is chosen in the range 0.005-0.05.

The next problem of system level synthesis is scheduling, which gives an exact order of the execution time of the nodes. Scheduling of the DFG starts from the assumption that unlimited resources are available, allowing any implementation to be feasible. Let us take a C code as shown in sample 2.1 and show its DFG for clarification (Fig. 2.8).

Sample 2.1: C Code

```
main()
{
float t, I1, o1, x1=0.0, x2=0.0;
while (1) {
in(I1);
t = I1 + a3*x2 + a1*x1; // line 6
o1=t+a4*x2 + a2*x1;      // line 7
x1 = t;                  // line 8
}}
```

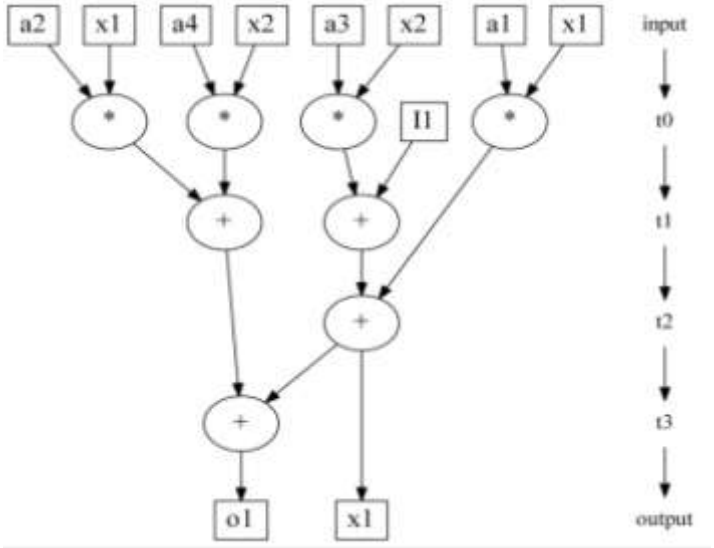


Figure 2.8: ASAP schedule

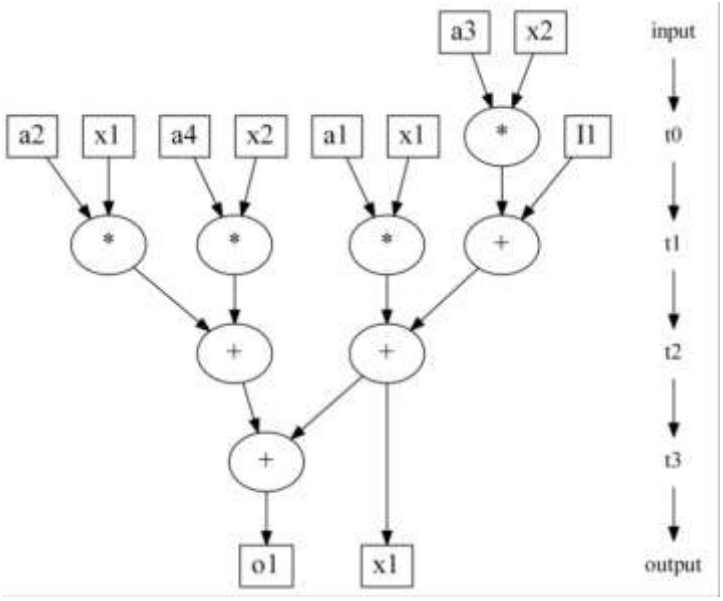


Figure 2.9: ALAP schedule

The commonly used algorithms for scheduling are:

A. As soon as possible (ASAP)

B. As late as possible (ALAP)

C. Constraint Scheduling

The ASAP/ALAP schedule is shown in Fig. 2.8 and Fig. 2.9 and is used as starting point to provide constraint parameters to scheduling algorithms. Assuming that unlimited resources are available, the DFG graphs are shown for line 6, 7, 8, 9 of the sample code. These algorithms assume that unlimited resources are available and allow maximum parallelism in the generated schedule. Maximum parallelism means that all the operators in the same level can

execute at the same time. For e.g. in Fig. 2.8, all the first four multipliers can run at the same time in *t0*. The pseudo code for the delay calculation is given below [1.7]. Each node is traversed and the delay of the parent is added to find the critical time of the DFG. We start from the nodes which have no predecessor and add the delay successively.

```
ASAP(G(V,E),d) {
    FOREACH ( vi without predecessor)
        s(vi) := 0;                          // starting node
    REPEAT {
        choose a node vi , whose predecessors are all planned;
        s(vi) := max_{j:(vj) ∈E} {s(vj)+ di};
            //predecessor delay = s(vj), current node delay = di
        }
        UNTIL (all nodes vi are planned);
            RETURN s;
}
```

Assume that an adder takes 10 clock cycles and multiplier takes 50 clock cycles. From the above algorithm, the output (*o1 in the graph)* will be available after 80 cycles (one multiplier and three adders). In case of ALAP scheduling, we start from the nodes without successors and traverse back as shown below:

```
ALAP(G(V,E),d, L) {
    FOREACH( vi without successor)
        s(vi) := L - di;              // L is the given latency
    REPEAT {
        Choose a node vi , which successors are all planned;
        s(vi) := min_{j:(vj) ∈E} {s(vj)} - di;
        }
        UNTIL (all nodes vi are planned);
            RETURN s
}
```

One of the parameter that is computed from ASAP and ALAP schedules is mobility. Mobility is the difference in the level of a node in the two schedules. For e.g. in Fig. 2.8 node 1(a2 and x1) has a mobility of 1. If the resources are restricted (i.e. lesser number of HW units for a given level), then the schedule changes accordingly. Practically limited amount of resources are available, hence we impose resource constraints to ASAP and ALAP schedules which is known as constraints scheduling [1.7]. Suppose that one adder and one multiplier are available

for allocation, and then the generated schedule is shown in Fig. 2.10. The output is this case will be available after 220 cycles since four multipliers and two adders are there for worst case path.

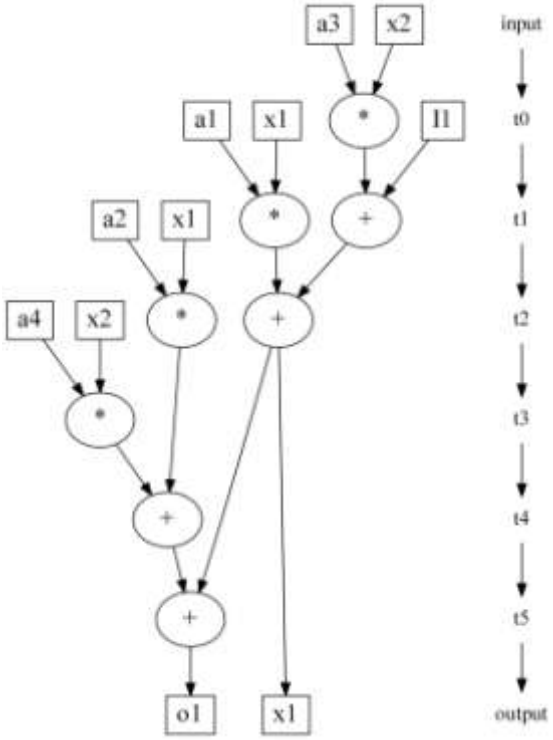


Figure 2.10: Constraint scheduling

The lecture notes in [2.36] show some scheduling examples on the dataflow model such. Task graph scheduling for parallel systems has been discussed in [2.37]. Since we requires a dataflow model of a specification, the Express benchmarks [2.38] describe the application written in C in the data flow format described as data flow in a dot model in Graphviz tool [2.39] from IBM. The sample of the cosine benchmark is given in Fig. 2.11 and can be used as an input for partitioning and scheduling phase.

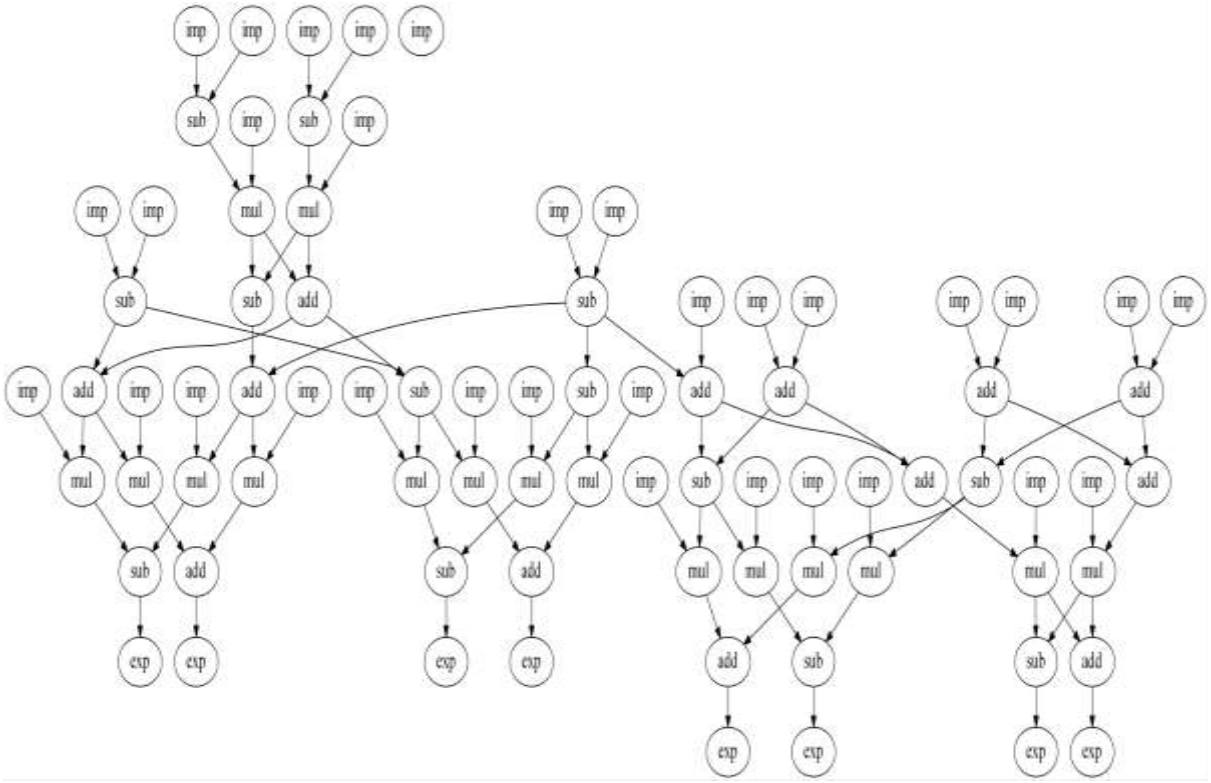


Figure 2.11: Express benchmark of Cosine-2 program [source: 2.59]

Integer linear programming [2.40, 2.41, 2.42] for scheduling has been used as a mathematical model and is based on equation used to solve the problem. Instruction/operator level which is very low level of abstraction may not produce desired optimized area and performance parameters, since such operator consume very less area as compared to the area available in RCS. Smaller nodes can be combined together to create coarse level nodes to reduce the overhead. This advantage also introduces interface design and intermediate data transmission design complexity. To create coarse level nodes one of the possibilities is finding the similar patterns in the application and uses them as reusable patterns. The scheduling of nodes clusters which are similar in nature extends the possibility of better implementation. The repetitive node patterns are known as isomorphic graphs as shown in Fig. 2.12.

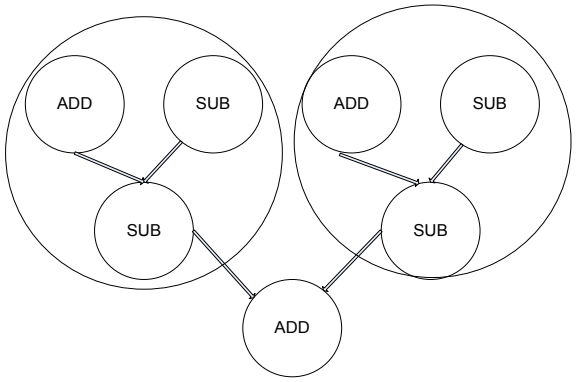


Figure 2.12: Node matching based isomorphic graphs

The problem of finding an isomorphic graph is a central position in complexity theory as a proposed occupant of the region that must exist between the polynomial-time and NP-complete problems. The easiest and the lengthiest way of finding such subgraphs can be a brute force method, which is not efficient since the order of complexity is $O(n! \times n^2)$. The time complexity for enumerate all bijective mapping will be $O(n!)$ and $O(n^2)$ will be complexity for checking whether each mapping is isomorphic. The problem of finding such graphs has been discussed extensively [2.43]. A weighted method has been used to guide the generation of isomorphic graphs [2.44]. We have used the method and applied it RCS systems after applying the area constraints and node matching as well.

2.3. Resource Estimation and High Level Synthesis

A FPGA chip has restricted resources in terms of LUTs, BRAMs, DSP slices etc. When an application written in high level language is migrated to HW, the primary concern is: The amount of resources consumed by the HW. Two methods can be discoursed for estimation of amount of resources. First is the analysis method which is based on a mathematical background and has an initial look-up-table for the resources consumed by each operator. Second method is the synthesis method which converts the specification into a HW using a HLL to HDL compiler.

Research works published in the **first method** domain can be cited firstly in [2.45] and have shown the estimation of input output pins and time required for the benchmarks. The work is outdated now, as the density of logic elements has increased many folds. The work done in the area estimation was first seen in [2.46]. The authors show the number of configurable logic blocks (CLB) consumed for a design based on number of operators, their bit-width and number of registers. The estimation is applicable to a dataflow model, where semantics is similar to the netlist. The work in [2.47] shows the estimation model for Matlab based system generator designs. The model is well developed for simulink based design and is not applicable to HLL. The work in [2.48] shows the compiler framework required for resource estimation of ANSI-C programs.

The resource estimation requires a mathematical model which gives results taking inputs as operators, bit-width, types of operators and return the number of LUTs/registers used. This model has been thoroughly proposed in [2.49]. The results have only been shown for one

benchmark which is not generic and estimation is not for ANSI-C programs. This work does not show the usage of resource estimation for ANSI-C program with a complete design flow.

The **second method** is based on high level synthesis (HLS) which is the process of converting a C/C++/SystemC based specification into synthesizable HW. Many tools have been developed over the last decade for this process to automate along with numerous optimizations applied is the respective tools. The SPARK [2.50] was the first open source tool produced and its primary objective was to apply scheduling of instructions for better synthesis. The following features were not supported in SPARK:

- Dynamic memory allocation
- Continue, break and goto
- Multidimensional arrays
- Function calls with parameter passing
- File operation

ROCCC [2.51] was another tool which was developed to remove some of the above restrictions, but required the changes in the syntax of the program. LegUp [2.52] from University of Toronto is still a active group in research in HLS and has developed a benchmark ChStone [2.53] written in HLS for comparison of different tools, Commercial tools include BlueSpec Compiler from Bluespec, Catapult C from Calypto Design Systems, eXCite from Y Explorations, Xilinx Vivado-HLS (formerly AutoPilot from AutoESL).

Control flow graphs have been frequently used for the automatic generation of HDL [2.54]. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. Basic blocks from the vertices or nodes in a control flow graph. LLVM [2.55] is a good candidate for the generation of CFGs. various kinds of optimization can be done on the C code for better conversion to HLS. The results in [2.56] show a comparative table of the low level optimizations and their effect on HLS. The work in [2.57] explores all the optimizations that can be done on the C code at various levels for HLSs. The work in [2.58] shows the generation of HW in a simplified manner. Most computational design implemented these days is image processing applications. The design flow in [2.59] explains the usage of FPGA for accelerator HW design. The lecture notes in [2.60] explain the generation of high level synthesis (HLS) from LLVM IR in a simple way with the use of datapath and finite state machine (FSM) design. The Xilinx Vivado HLS [2.61] tool provides options for carrying out

different types of optimizations on the behavioral description before synthesizing it which enables the user to bring the design closer to the given throughput or area specification.

2.4. Reconfigurable Computing Systems

From the discussion of the architecture of RC systems in chapter 1, various issues and challenges in the research work in this domain are identified. An RC system is developed around a specific HW and a defined architecture which can control reconfiguration. These systems have intrigued and inspired the designer to address the following questions:

1. Who controls the reconfiguration? *Processor (may be inside the FPGA)*
2. Where the configurator is located? *Dedicated physical component in the FPGA*
3. When the configurations are generated? S*ynthesizing the best static possible configuration*
4. Which is the granularity of the reconfiguration? *Smallbit based (CLB) or module based*

In order to further highlight this aspect, a 3 axis classification scheme described by John Williams [2.62], characterizes the diversity of the reconfigurable systems depending on

1. Reconfiguration Controller
2. Configuration generation
3. Level of reconfiguration granularity

These parameters are discussed as below:

2.4.1. Reconfiguration Controller

The systems in which reconfiguration is managed and controlled by some external device are called as externally reconfigurable and but those initiate and control their own reconfiguration are called as self-reconfigurable. So such a system can perform self read-back and reconfigure themselves by loading a new configuration stored in a external memory. Some systems can be a combination of both these which requests modules from a remote bitstream server controller.

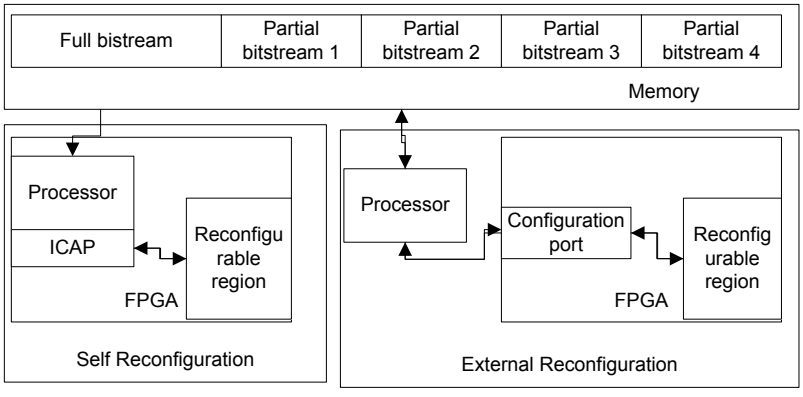


Figure 2.13: External vs. internal reconfiguration

2.4.2. External Reconfiguration

External reconfiguration implies that the FPGA resources can be reconfigured by an external device such as a personal computer or a microprocessor. In this case the external processor reads the partial bitstreams from external memory and sends the data through the standard reconfiguration part of the FPGA as shown in Fig. 2.13. A single FPGA configuration engine handles both full configuration and partial reconfiguration using the same programming mechanism. The external reconfiguration can be achieved using SelectMAP, JTAG, Serial ports [2.63].

2.4.3. Internal Reconfiguration or Self Reconfiguration

Internal reconfiguration or self-reconfiguration system uses an application running on a configuration controller, generally a processor inbuilt in the system to read partial bitstream from external memory and send the data to the ICAP (Internal configuration access port) which then reconfigures the portion of the FPGA indicated by the configuration frame address included in the partial bitstream. Partial bitstream contain all the necessary commands and the data necessary for partial reconfiguration [2.64]. The ICAP peripheral enables an embedded microprocessor to read and write the FPGA configuration memory at runtime. The ICAP peripheral provides control over FPGA resources with the granularity of a single configuration frame consisting of 41 32-bit words in Virtex-4 and Virtex-5 family, 81 32-bit words in Virtex-6 family and 65 16-bit words for Spartan-6 FPGA.

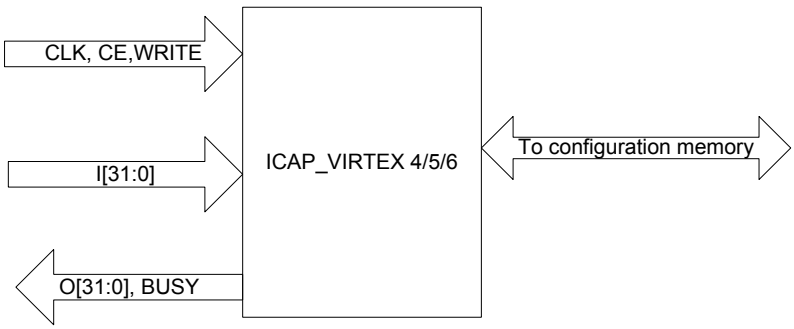


Figure 2.14: ICAP controller

The ICAP primitive for Vitex-4, Virtex-5, and Virtex-6 families is shown in Fig 2.14. The ICAP primitive [2.65, 2.66] has four input ports (CLK, CE/CSB, WRITE/RDWRB, I [31:0]) and two output ports (BUSY, O[31:0]). The partial reconfiguration time depends on the ICAP CLK frequency, configuration data size i.e. bitstream size and the data width of the ICAP port.

The configuration bandwidth for various families is shown in Table 2.3. The configuration time for the FPGA families for a single frame are also shown. Thus the data width and maximum clock frequency are limiting factors that impacts reconfiguration time.

Table 2.3: Configuration bandwidth Using ICAP primitive [Source: 2.65]

FPGA Family	Maximum ICAP CLK frequency	Data width	Bandwidth	Reconfiguration time for 1 frame (for virtex-4/5 41 32-bit words, for Virtex-6 81 32-bit words, for Spartan-6 65 16-bit words)
Virtex-4	100 MHz	32	3.2 Gbps	0.41 µs
Virtex-5	100 MHz	32	3.2 Gbps	0.41 µs
Virtex-6	100 MHz	32	3.2 Gbps	0.81 µs
Spartan-6	20 MHz	16	1.6 Gbps	3.25 µs

The reconfigurable systems can be characterized by the degree to which they manipulate the FPGA resources. A minimum of 1 frame can be reconfigured in a Xilinx FPGA. The Virtex-5 frame has a reconfiguration time of 0.41 microseconds. One frame of Virtex-5 contains 8 CLBs, one CLB has 2 slices and one slice has 4 LUTs.

2.4.4. Partial Reconfigurable System Design in Xilinx

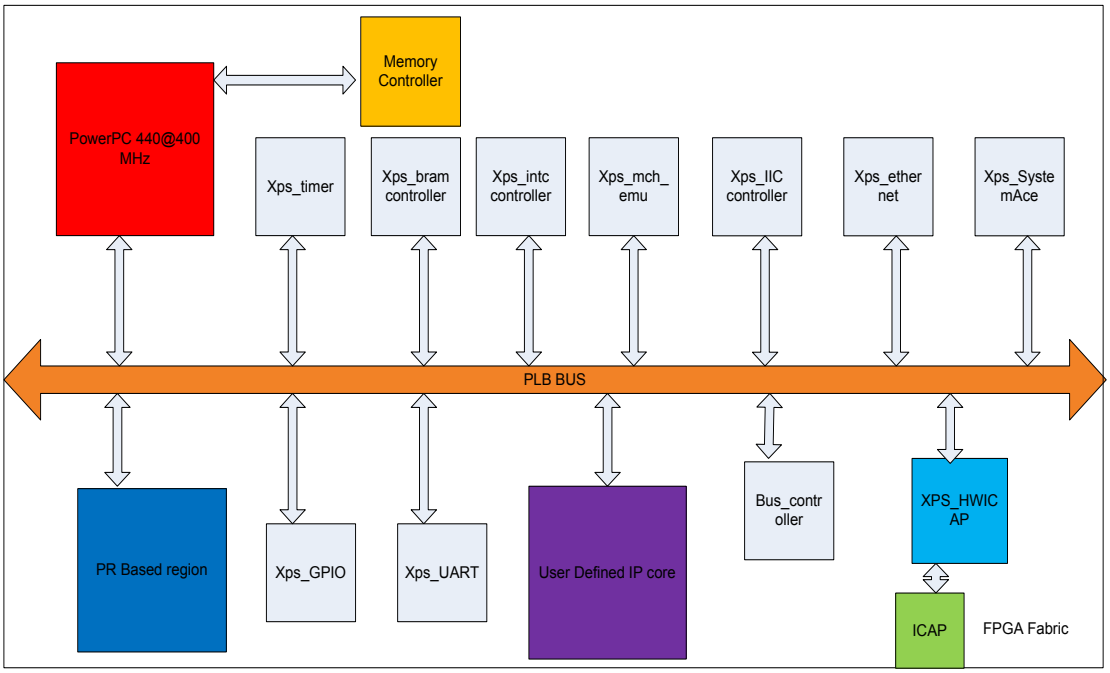


Figure 2.15: FPGA systems with one reconfigurable region

Fig. 2.15 shows a conventional partial reconfiguration (PR) design components using Xilinx EDK and PlanAhead tools depicting the one PR region in blue color. For example PRR can

execute integer operations or execute floating operations. The user defined IP core shown in purple color is a static logic defined by the user. The essential components used in PR design flow are memory where the partial bit files are stored and ICAP which loads the partial bit files on the configuration memory from external memory. The processor instructs the ICAP to load the bitstream from external memory to the buffer. ICAP then reconfigures the portion of FPGA indicated by the configuration frame address which is included in the partial bitstream through the configuration port.

2.5. Challenges of RCS

RCS design flow presents many degrees of concern in creating a successful design.

2.5.1. The Complex Design Flow

The biggest disadvantage in creating the partial reconfiguration flow is the complex steps the designer has to follow. The steps are given below

- Create an IP and verify the functionality.
- Interface the port of the IP to the bus wrapper as a black box.
- Write the scheduler in SW.

The problem found during PR design flow can sometimes be undesirable and can be resolved, leading to multiple time project creation.

PR can be explained with a analogy: consider a class room (black box) in a building which has fixed doors/windows (ports) in which various classes and masses (different designs) are involved and scheduled at different time (scheduler).

2.5.2. Restrictions in Design Flow

In-spite of the enormous advantages of partial reconfiguration technology, there are some limitations such as lack of complex applications that substantially benefit from a run-time reconfiguration, the design tools that permit quick development of PR systems and the cost of the programmable devices having PR feature which prevents its use in the commercial products. Some resources on the FPGA can be reconfigured and some cannot which poses substantial challenges in the system design. In the tools there are limitations in selection of resources during partial reconfiguration design that are [2.66] as follows.

1. Clocking logic such as global buffers, memory controller and digital clock manager should reside in the static region of the design. This means that if the synthesized IP uses any of these resources then it should in static region.

2. Multi-gigabit Transceiver, boundary scan, startup should not exist in the static region of the design.

3. No bidirectional interfaces are permitted between static and reconfigurable regions, except in case where there is a dedicated route. For example a bidirectional I/O buffer in the reconfigurable region routed to a top level I/O pad in the static logic can cross between the reconfigurable region and static logic via a bidirectional interface.

4. The number of global clocks that can be pre-routed to any clock region, and therefore to any reconfigurable partition, depends on the device family used. The number of clocks in reconfigurable region cannot exceed the defined limit such as 8 for Virtex-4, 10 for Virtex-5 and 12 for Virtex-6 & Virtex-7 respectively. These limits must account for both static & reconfigurable logic.

5. Active low resets & clock enables should be used in the design flow to allow the SW to reset and enable the configuration.

2.5.3. The Reconfiguration Overhead

Table 2.4: Comparison of Reconfiguration Throughput from 2003 to 2009 [source: 2.66]

Reference	Storage	Conf. Port	Cntlr	BS	RT	ARTP
[Fong et al. 2003]	PC/RS232	ICAP8	c-HW	34.8	6,200	0.005
[Griese et al. 2004]	PC/PCI	SMAP8@25	c-HW	57.0	14.328	3.88
[Gelado et al. 2006]	$BRAM_{PPC}$	ICAP8	v-OPB	110.0	25.99	4.13
[Delahaye et al. 2007]	SRAM	ICAP8	v-OPB	25.7	0.510	49.20
[Papadimitriou et al. 2007]	$BRAM_{PPC}$	ICAP8@100	v-OPB	2.5	1.67	1.46
[Papadimitriou et al. 2010]	CF	ICAP8@100	v-OPB	14.6	101.1	0.15
[Claus et al. 2007]	DDR	ICAP8@100	c-PLB	350.75	3.75	91.34
[Claus et al. 2007]	DDR	ICAP8@100	v-OPB	90.28	19.39	4.66
[Claus et al. 2008]	DDR	ICAP8@100	c-PLB	70.5	0.803	89.90
[Claus et al. 2008]	DDR	ICAP8@100	v-OPB	70.5	15.0	4.77
[Claus et al. 2008]	DDR2	ICAP32@100	c-PLB	1.125	0.004	295.40
[Claus et al. 2008]	DDR2	ICAP32@100	v-OPB	1.125	0.227	5.07
[French et al. 2008]	DDR2	ICAP32@66	v-OPB	514.0	112.0	4.48
[Manet et al. 2008]	DDR2/ZBT	ICAP32@100	c-OPB	166.0	0.47	353.20
[Liu et al. 2009]	DDR2	ICAP32@100	v-OPB	79.9	135.6	0.61
[Liu et al. 2009]	DDR2	ICAP32@100	v-XPS	80.0	99.7	0.82
[Liu et al. 2009]	DDR2	ICAP32@100	v-OPB	75.9	7.8	10.10
[Liu et al. 2009]	DDR2	ICAP32@100	v-XPS	74.6	4.2	19.10
[Liu et al. 2009]	DDR2	ICAP32@100	c-PLB	81.9	0.991	82.10
[Liu et al. 2009]	DDR2	ICAP32@100	c-PLB	76.0	0.323	234.50
[Liu et al. 2009]	$BRAM_{ICAP}$	ICAP32@100	c-PLB	45.2	0.121	332.10

The time required to place the bit file of an application which is stored in a memory on a partial reconfigurable region is known as reconfiguration time. When the reconfiguration is performed, the bit file has to move through various stages, which adds time and is referred as reconfiguration overhead. The characteristics that affect the reconfiguration overhead also depends on the system setup such as external memory, memory controller, reconfiguration controller and its interface with ICAP, and the user space to kernel space copy penalty when an operating system is running on the processor controlling the reconfiguration. The size of the local memory of the processor (cache) can significantly affect the reconfiguration time. The primary concern for partitioning and executing an application on one PR region is time overhead. Author in [2.67] gives exhaustive time model that can be used to calculate theoretical results.

For Table 2.4: BS: Bitstream size in bits, RT: Reconfiguration time in ms, ARTP: Actual Reconfiguration throughput. The last column shows that the maximum throughput is obtained is 353.20 Mbps using DD2/ZBT based memory. They have described the total reconfiguration time as the sum of times spent in transferring the data from the storage memory to processors local memory (phase 1), then to the ICAP configuration cache (phase 2) and then finally to the configuration memory of FPGA (phase 3). Thus, total reconfiguration time can be expressed as

Reconfiguration Time= RTSM PPC + RTPPC ICAP + RTICAP CM ... (2.1)

Where,

- RTSM PPC- time spent in transferring bitstream data from storage memory to processor local memory
- RTPPC ICAP - time spent in transferring bitstream data from PPC local memory to ICAP cache
- RTICAP CM - time spent in transferring bitstream data from ICAP cache to configuration memory.

An online calculator [2.67] for estimating time overhead given the bitstream size is also available. This calculator requires the bitstream size, which means that after the entire design flow is complete and project is created in PlanAhead.

The earliest work in this domain reports a throughput of 94.85 MB/s by using DMA and a new ICAP design [2.68]. Since this value was too low it required further improvement. Three designs as Master ICAP, BRAM ICAP and DMA ICAP [2.69] show a performance of 253.2 MB/s, 371.4 MB/s , 82.6 MB/s. Table 2.5 show the comparison of various methods. Although

the results have been promising, but the resource usage for BRAM is very high and scarcity of memory makes it unrealistic design. Very good results are cited in [2.70], where 1.2 GB/s is reported with the combination of full streaming direct memory access (DMA) and bitstream compression. Such a drastic improvement requires inspection as the factor is 1000 times. The answer for this is, simply instruction simulator (ISE) based project (non processor based) has much greater performance as compared to an EDK based project (processor based). Further improvement up to 2.2 GB/s is reported [2.71] by over clocking ICAP. The usage of partial reconfiguration for real systems by scheduling was first reported in [2.72]. The comparative summary is shown in Fig. 2.5.

Table 2.5: Reconfiguration speed measurement of ICAP design for various sizes of partial bitstream [Source: 2.69]

ICAP design	Test 1 (Bit. Size/Reconf. Time)	Test 2 (Bit. Size/Reconf. Time)	Test 3 (Bit. Size/Reconf. Time)	Test 4 (Bit. Size/Reconf. Time)	Avg. reconfig. speed	Max. reconfig. speed
OPB_HWICAP (PowerPC cache disabled)	7.7 KB/12.1 ms	23.2 KB/36.5 ms	44.5 KB/75.6 ms	79.9 KB/135.6 ms	0.61 MB/s	0.64 MB/s
XPS_HWICAP (PowerPC cache disabled)	7.7 KB/9.2 ms	23.2 KB/27.9 ms	46.5 KB/57.9 ms	80.0 KB/99.7 ms	0.82 MB/s	0.84 MB/s
OPB_HWICAP (PowerPC cache enabled)	7.7 KB/694.8 µs	22.7 KB/2.3 ms	43.9 KB/4.5 ms	75.9 KB/7.8 ms	10.1 MB/s	11.1 MB/s
XPS_HWICAP (PowerPC cache enabled)	7.7 KB/336.9 µs	23.2 KB/1.3 ms	44.5 KB/2.5 ms	74.6 KB/4.2 ms	19.1 MB/s	22.9 MB/s
OPB_HWICAP (Microblaze cache enabled)	7.7 KB/1.3 ms	23.2 KB/3.9 ms	47.1 KB/7.9 ms	77.7 KB/13.0 ms	6.0 MB/s	6.0 MB/s
XPS_HWICAP (Microblaze cache enabled)	7.7 KB/532.6 µs	23.2 KB/1.6 ms	47.2 KB/3.3 ms	79.1 KB/5.4 ms	14.5 MB/s	14.6 MB/s
DMA_HWICAP	7.7 KB/95.1 µs	23.2 KB/282.3 µs	46.8 KB/566.3 µs	81.9 KB/991.1 µs	82.1 MB/s	82.6 MB/s
MST_HWICAP	7.7 KB/33.0 µs	23.2 KB/98.9 µs	44.8 KB/190.7 µs	76.0 KB/323.1 µs	234.5 MB/s	235.2 MB/s
BRAM_HWICAP	7.7 KB/28.0 µs	23.2 KB/66.3 µs	45.2 KB/121.7 µs	none	332.1 MB/s	371.4 MB/s

Most of the work has been concentrated on reducing reconfiguration time by designing an efficient ICAP controller, using DMA, placing bistream in BRAM and compressing the bitstream file. The theoretical speed of the ICAP is 100MHz, and its width can be programmed as an 8-bit or a 32 bit port allowing a bandwidth of 0.75 or 2.98GB/s.

Authors have shown one PR region that implements different protocol to read data from sensors. The paper also proposes a simple equation to calculate reconfiguration. The reconfiguration time mainly depends upon the following features [2.73]:

1. Configuration clock speed (CCLK)

2. Bus width

3. Bitstream size

The reconfiguration time of FPGA is estimated as

$$\text{Reconfiguration time(s)} = \text{Bitstream size} / (\text{Cclk} * \text{Buswidth}) \quad \ldots(2.2)$$

But Eq. 2.2 does not take system setup parameters into consideration. The only work with multiprocessor design for PR design is seen in [2.74]. From this we conclude that the conventional ICAP controller available in EDK flow has low throughput of 10 Mbps.

This leads us to summarize the research horizons as:

- Propose an automated HW-SW co-design approach and compare the performance.
- Propose partitioning approaches for converting dataflow graphs into clusters that accelerate the performance further.
- Design clusters as IP cores and map them to partial reconfigurable region. Compare the performance the design flow.
- Use resource estimation technique for making these clusters based on a given area.
- Perform the resource estimation using a compiler based flow.

2.6. Conclusions

- LegUp tool is the only open source tool available for co-design and results shown are not very promising in a SOC based design. Low speedups were reported in the survey which of the order of 2x in HW.
- The designer should be guided easily for adopting a particular design flow. Such a flow should generate various HW-SW solution of an application.
- Time profiling and HLS tools can be used for generating time and area parameters consumed by an application.
- Control and dataflow are commonly used graphically representations as input to partitioning and scheduling algorithms. These intermediate structures are also used by the HLS tools to convert the SW into HW.
- Graph isomorphic concept can be used to find the similar patterns in the dataflow model.
- Resource estimation can be done using a mathematical approach.
- Genetic algorithm performs better as compared to SA, FM and MFM.

- Using a partial reconfiguration for a partitioning and running a single application on HW has not been reported.
- We conclude that maximum throughput of 2 Gbps is possible with reconfiguration process with specialized architecture.

REFERENCES

2.1 Frank Vahid and Tony Givargis, Embedded System Design: A Unified Hardware/Software Introduction, John Wiley & Sons; ISBN: 0471386782. Copyright (c), 2002. (chapter-7)

2.2 Hardware Implementation of Theora Decoding Integration with LEON3, http://people.xiph.org/~j/bzr/theora-fpga/doc/leon3_integration. , last accessed on 15 July 2016.

2.3 www.gaisler.com/products/grlib/grlib.pdf, last accessed on 15 July 2016.

2.4 https://www.cs.ccu.edu.tw/~pahsiung/courses/codesign/resources/codesign-tools.html

2.5 ASSET: Anshul Kumar & M Balakrishnan. Automated Synthesis of embedded systems. Technical report, Dept. of computer science and engineering, IIT-Delhi, June 2002.

2.6 A. Canis, J. Choi, M. A. V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, Jason H. Anderson, LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems, ACM Transactions on Embedded Computing Systems (TECS) - Special issue on application-specific processors, vol. 13 Issue 2, article No. 24, September, 2013.

2.7 M. Aldham, J. Anderson, S. Brown, A. Canis, Legup: Low-Cost Hardware Profiling of Run-Time and Energy in FPGA Embedded Processors, ASAP '11 Proceedings of the ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 61-68, 2011.

2.8 J. B. Peterson, O'Connor R.B and Athanas, P.M. Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures, in IEEE Symposium on FPGAs for Custom Computing Machines, pp. 178-187, April 17-19, 1996.

2.9 M. Karthikeya, G. Purna, and D. Bhatia, Member, Temporal partitioning and scheduling data flow graphs for reconfigurable computers, IEEE transactions on computers, vol. 48, no. 6, June 1999.

2.10 S. Ganesan and R. Vemuri, An integrated temporal partitioning and partial reconfiguration technique for design latency improvement. In DATE ’00: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 320–325, New York, NY, USA, 2000, ACM Press.

2.11 R. Maestre, F. J. Kurdahi, M. Fernandez, R. Hermida, A Framework for Reconfigurable Computing task scheduling and context management-a summary, Circuits and Systems Magazine, IEEE, pp. 48-51, December, 2001.

2.12 Qiang Wu,Yunfeng Wang, Jinian Bian,Weimin Wu, A hierarchical CDFG as Intermediate Representation for Hardware Software Codesign, IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions, pp. 1429-1432, July, 2002.

2.13 J. M. P. Cardoso, On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures, IEEE Transactions on Computers, pp. 1362-1375, October, 2003.

2.14 T. Kato, T. Miyauchi, Y, Osumi, H. Yamauchi, A CDFG Generating Method from C Program, IEEE Asia Pacific Conference on Circuits and Systems, pp. 936-939, December , 2008.

2.15 R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, Partitioning and Scheduling of Task Graphs on Partially Dynamically Reconfigurable computers, IEEE transactions on computer-aided design of integrated circuits and systems, vol. 28, no. 5, May 2009.

2.16 Xie Di , Shi Fazhuang , Deng Zhantao, He Wei Design Flow for FPGA Partial Dynamic Reconfiguration, Published in: Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2012 Second International Conference on Date of Conference, pp. 119-123, December, 2012.

2.17 K. Vipin, S. A. Fahmy, Automated Partitioning for Partial Reconfiguration design of adaptive systems Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International Date of Conference, pp. 172-181, May, 2013.

2.18 Giovanni De Micheli, Synthesis and optimization of Digital Circuits, Tata McGraw-Hill Edition, 2003.(Chapter-7)

2.19 D. D. Gajski, F. Vahid, *S.* Narayau, J. Gong. Specification and Design of Embedded Systems, Prentice-Hall 1994. (Chapte-4)

2.20 Z. A. Mann and A. Orban, Optimization problems in system-level synthesis, in Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications, 2003.

2.21 P. Elis, Z. Peng, K. Kuchcinski, and A Doboli, System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search, Design Automation for Embedded Systems, vol. 2, No. 1, pp. 5-32, January, 1997.

2.22 P. Elis, Z. Peng, K. Kuchcinski, and A Doboli, Hardware/software partitioning with iterative improvement heuristics, Proc. of 9th International Symposium on System Synthesis, pp. 71-76, November, 1996.

2.23 M. Koudil, K. Benatchba, S. Gharout, and N. Hamani, Solving Partitioning Problem in Codesign with Ant Colonies, Artificial Intelligence and Knowledge Engineering Applications: A Bio inspired Approach, Lecture Notes in Computer Science, Springer, vol. 3562/2005, pp. 324-337, June, 2005.

2.24 S. Zheng, Y. Zhang and T. He, The Application of Genetic Algorithm in Embedded System Hardware-Software Partitioning, Proc. of the 2009 International Conference on Electronic Computer Technology, pp. 219-222, February, 2009.

2.25 L. Li, and M. Shi, Software-Hardware Partitioning Strategy Using Hybrid Genetic and Tabu Search, Proc. of 2008 International Conference on Computer Science and Software Engineering, pp. 83-86, December, 2008.

2.26 J. H. Holland, Adaptation in Natural and Artificial Systems, Cambridge, MA: MIT Press, 1992. (chapter-4)

2.27 D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Boston, MA: Addison-Wesley, 1989.(Chapter 3)

2.28 J. J. Grefenstette, R. Gopal, B. J. Rosmaita, and D. Van Gucht, Genetic algorithms for the travelling salesman problem, in Proc. 1st Int. Conference on Genetic Algorithms, pp. 160–168, 1985.

2.29 M. Vellasco, R. Zebulum, and M. Pacheco, Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms, CRC Press, 2001.(chapter-4)

2.30 G. De Micheli. “Computer Aided Hardware Software Codesign”, IEEE Micro., IEEE Computer Society Press, pp. 10-16, August 1994.

2.31 S. D. Scott, A. Samal, and S. Seth, HGA: A hardware-based genetic algorithm, in Proc. ACM/SIGDA 3rd Int. Symp. Field Programmable Gate Array, pp. 53–59, 1995.

2.32 M. Tommiska and J. Vuori, Implementation of genetic algorithms with programmable logic devices, in Proc. 2nd Nordic Workshop Genetic Algorithm, pp. 71–78, 1996.

2.33 B. Shackleford, G. Snider, R. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura, A high-performance, pipelined, FPGA-based genetic algorithm machine, Genetic Algorithms Evolvable Mach., vol. 2, no. 1, pp. 33–60, March, 2001.

2.34 J.I. Hidalgo, J. Lanchares, Functional Partitioning for Hardware - Software Codesign Using Genetic Algorithms, IEEE 1997.

2.35 M. Srinivas and L. M. Patnak , Adaptive Probabilities of Crossover and Mutation in Genetic Algorithm, IEEE Trans. on SMC., Vol. 24, pp. 656 – 666, No.-4, April, 1994.

2.36 Jaap Hofstede, Example of Scheduling and Allocation, Lecture Notes, Version 2, September, 2000. web.cecs.pdx.edu/~mperkows/temp/0113.Scheduling-and-Allocation-example.pdf

2.37 Oliver Sinnen, Task scheduling for parallel systems, Wiley , September, 2006.(chapter-4)

2.38 Express benchmark, http://www2.imm.dtu.dk/SoC-Mobinet/modules/HLS/benchmarks/index.html, last accessed on May, 2013.

2.39 Graphviz tool IBM, http://www.graphviz.org/

2.40 C. A. Floudas, X. Lin, Mixed Integer Linear Programming in Process Scheduling: Modeling, Algorithms, and Applications, Annals of Operations Research, Springer, Volume 139, Issue 1, pp 131–162, October 2005,.

2.41 Ralf Niemann and Peter Marwedel, Hardware/Software Partitioning using Integer Programming, EDTC '96 Proceedings of the European conference on Design and Test, pp. 473, 1996.

2.42 Fakhreddine Ghaffar, Benoit Miramond and François Verdier, Run-Time HW/SW Scheduling of Data Flow Applications on Reconfigurable Architectures, EURASIP Journal on Embedded Systems, December, 2009.

2.43 S. Bachl and F. J. Brandenburg, Computing and drawing isomorphic subgraphs, in Graph Drawing, vol. 2528, S. G. Kobourov and M. T. Goodrich, Eds. Berlin, Germany: Springer-Verlag, pp. 74–85, November, 2002.

2.44 Gross, Jonathan L., and Jay Yellen, eds. Handbook of graph theory, CRC press, 2003.

2.45 Min Xu,. F. J Kurdahi, A Tool for Chip Level Area and Timing Estimation of Lookup Table Based FPGAs for High Level Applications , Design Automation Conference, pp. 435-440, January, 1997.

2.46 A. Nayak, M. Haldar, A. Choudhary, P. Banerjee, Accurate Area and Delay Estimators for FPGA, Design, Automation and Test in Europe Conference and Exhibition, pp. 862-869, March, 2002.

2.47 Changchun Shi S., J. Hwang , S. McMillan, A. Root, V. Singh, A System Level Resource Estimation Tool for FPGA , Field Programmable Logic and Application, Lecture Notes in Computer Science Volume 3203, pp .424-433, September, 2004.

2.48 Z. Kulkarni, W. Najjar, R. Rinker, F.J., Kurdahi, Compile-Time Area Estimation for LUT-Based FPGAs, ACM Transactions on Design Automation of Electronic Systems, Volume 11, Issue 1, January, 2006.

2.49 Arce-Nazario R. A., Juan S., Jimenez M., Rodreguez D., Architectural Model and Resource Estimation, Reconfigurable Computing and FPGAs International Conference, pp. 103-108, December, 2008.

2.50 S. Gupta, N. Dutt N., R. Gupta, and A. Nicolau, Spark: A high-level synthesis framework for applying parallelizing compiler transformation, Proceedings of the 16th international conference on VLSI design, pp. 461-466, January, 2003.

2.51 University of California, Riverside. River side optimizing compiler for configurable computing (ROCCC). http://www.cs.ucr.edu/~roccc/. Last accessed on March, 2012.

2.52 A. Canis, J. Choi, B. Fort, R. Lian , From Software to Accelerators with LegUp High-Level Synthesis, 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), pp. 1-9, September, 2013.

2.53 Y. Hara, H. Tomiyama, S. Honda, H. Takada, K. Ishii, CHStone: A benchmark program suite for practical C-based high-level synthesis, IEEE International Symposium on Circuits and Systems, ISCAS, pp. 1192-1195, May, 2008.

2.54 R. Rinker, J. Hammes, W. A. Najjar, W. Bohm, Compiling Image Pro cessing Applications to Reconfigurable Hardware, IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2000. Proceedings, pp. 56-65, July, 2000.

2.55 Low level vitrual machine, http://llvm.org/ , last accessed on June, 2015.

2.56 Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, Jason Anderson, The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs, IEEE Int'l Symposium on Field-Programmable Custom Computing Machines (FCCM), Seattle, WA, May, 2013.

2.57 Sameer Mohamed Thahir, Compiling for Reconfigurable Computing Seminar Report, Cochin University of Science and Technology, May, 2011.

2.58 J. Cong, B. Liu, S. Neuendorffer, N. Juanjo, V. Kees, Z. Zhang, High-level synthesis for fpgas: from protopying to deployment, IEEE transactions on computer-aided design of integrated circuits and systems, vol. 30, no. 4, April, 2011.

2.59 C. Pal, A. Kotal, A. Samanta, A. Chakrabarti, R. Ghosh, Design space exploration for image processing architectures on FPGA targets, Proc. International Doctoral Symposium on Applied Computation and Security Systems(ACSS) , pp. 1-19, April, 2014.

2.60 Prof. Dr. Christian Plessl, Architecture Synthesis, Lecture notes, 2012. homepages.uni-paderborn.de/plessl/lectures/2012-Codesign/

2.61 Vivado Design Suite User Guide, UG902 (v2013.2) July 19, 2013.. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

2.62 Williams, John A., Neil W. Bergmann, Embedded Linux as a Platform for Self Reconfiguring systems-on-chip, Ersa'04: the 2004 International Conference On Engineering of Reconfigurable Systems and Algorithms. CSREA Press, 2004.

2.63 David Dye. Partial Reconfiguration of Xilinx FPGAs using ISE design suite. White paper: virtex-4, virtex-5, virtex-6 and 7 series FPGA, Xilinx Inc., 2012.

2.64 Introduction to Partial reconfiguration, Xilinx, www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf

2.65 Logicore IP XPS HWICAP, v-5.0, www.xilinx.com/support/documentation/ip.../xps_hwicap/v5_01_a/xps_hwicap.pdf

2.66 Partial Reconfiguration Design Considerations Objectives, mit.bme.hu/~feher/Heterogen_szam_rendsz/Old/04_PR_Design_Considerations.pdf

2.67 K. Papadimitriou, A. Dollas, S. Hauck, Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model, ACM transaction on reconfigurable technology and Systems, Volume 4, Issue 4, December, 2011.

2.68 C. Claus, F. H. Muller, J. Zeppenfeld, W. Stechele, A new framework to accelerate Virtex-II Pro dynamic partial, 2007 IEEE International Parallel and Distributed Processing Symposium, pp. 1-7, March, 2007.

2.69 M. Liu, W. Kuehn, Z. Lu, A. Jantsch, Run-time partial reconfiguration speed investigation and architectural design space exploration, International Conference on Field Programmable Logic and Applications, pp. 498-502, September, 2009.

2.70 A. S. Liu, R. N. Pittman, A. Forin, Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller, Proceeding FPGA '10 Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, pp. 292-292, 2010.

2.71 S. G. Hansen, D. Koch, J. Torresen, High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro, 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 174-180, May, 2011.

2.72 K. Vipin, and S.A. Fahmy. A high speed open source controller for FPGA Partial Reconfiguration. International Conference on Field-Programmable Technology (FPT), pp. 61-66, December, 2012.

2.73 C. S. Ibala and K. Arshak, Using Dynamic Partial Reconfiguration approach to read, Sensor with different bus protocol, SAS 2009 IEEE Sensors Applications Symposium, 2009.

2.74 S. Di Carlo, P. di Torino, G. Gambardella, M. Indaco , P. Prinetto, Dependable dynamic partial reconfiguration with minimal area, 23rd International Conference on Field programmable Logic and Applications, pp. 1-4,September, 2013.

Chapter 3

Automated Migration of Applications in Hardware Software Co-design Paradigm

This chapter presents an elaborate introduction to hardware software co-design process and challenges involved. The methodology required for co-design needs the identification of the time consuming part of the application and its automated hardware generation. Time profiling using gprof has been demonstrated for finding the critical parts of the program. A basic introduction on how C to HDL converter is designed using a compiler (LLVM compiler) and how it can be used for estimating the resources consumed by a program have been discussed. A commercial C to HDL converter (Vivado-HLS) has been explored in the chapter for accurate resource estimation and it also allows to apply various optimizations and explores the latency-area trade-offs.

The chapter is organized as follows: In section 3.1 an introduction to HW and SW systems is presented. A co-design framework has been proposed in section 3.2. The two important aspects: time and area estimation are demonstrated in section 3.3 and 3.4. The area estimation using a commercial high level synthesis tool is showcased in section 3.5. Section 3.6 presents the IP core interfacing techniques and its challenges generated from the tool. For measuring the performance, a HW timer is required which is discussed in section 3.7. In section 3.8, the co-design flow has been applied on a Dfdiv program and results are presented. Section 3.9 shows the results obtained for Dfdiv program from LegUp tool which was discussed in the literature survey.

3.1. Hardware and Software Systems

A digital system can be visualized as a HW-SW layered architecture as shown in Table 3.1 with various components used in each layer [3.1].

Table 3.1: Digital system hardware and software layered architecture

Applications (e.g. Brower, jpeg)
Compilers(e.g. gcc, LLVM)
Operating Systems(e.g. Windows, RTOS)
Firmware(e.g. HAL, BSP)
Interfaces(e.g. ISA, AMBA)
Hardware(e.g. processor, memory, accelerators)

The bottom most layers comprise of HW components followed by firmware, OS and applications layers. At the top of the layer, any application can be described as pure SW in a programming language. The same application can also be described as pure HW in a HDL and interfaced to the processor as accelerators.

A system-on-chip based design allows the system to be implemented in SW, HW or a combination of both as discussed in chapter 1. This gives rise to various permutations for design space exploration and the trade-off of system parameters. The parameters which are of prime concern in the design flow are time, area and power. These parameters are orthogonal in nature and affect each other. For example parallel HW unit consumes more area but it can do more computation in parallel.

The SW designer works at the application development layer and achieves the functionality in SW using a high level language [3.1]. The popular languages used for embedded system design are C, C++ and Java. Here, the developer requires minimal insight into the HW characteristics and focuses on achieving the desired functionality. Depending on the complexity of the design and performance requirements, these designs can either be operating system (OS) [3.2, 3.3] or non-OS [3.4] based. In OS based approach, a device driver is required for communicating with the accelerators, where as in non-OS based simple read /write protocol driver can be used. Multi-threaded applications can be easily be developed in OS based approach, hence is a preferred choice. The popularly used processor based software platforms in this domain are Beagle board, Beagle-bone, Raspberry-pi, Panda board, ARM LPC series and FPGA based platform design.

When the SW implementation does not meets the required performance gain primarily due to application demanding intensive computation, the designer looks for a total HW implementation. For e.g. SW implementation of a JPEG compression roughly takes xyz seconds on an ARM based platform, but it may take xyz/100 seconds on an ASIC IC based design or xyz/10 second based on FPGA based design flow [2.1]. As a result of this gain, frequently used applications such as multimedia, compression are being migrated to HW to improve performance. The migration of a SW implementation to HW implementation is a time consuming process as it requires the description of the application in the form of control and datapath structures in a HDL language. An HDL programmer aims at optimizing the resource usage, achieving the maximum clock frequency and limiting the power dissipation during this conversion. The popular languages used to describe the HW specification are Verilog, SystemVerilog, VHDL and SystemC.

3.2. Automated Approach to HW-SW Co-design

Now suppose we are given an application in high level language, which has certain functions like preprocessing, discrete cosine transform (DCT) and encoding as shown below.

main (){

Preprocessing ();

DCT ();

encoding (); }

A SW implementation of the above program may take xyz seconds and a HW implementation may take xyz/100 second, but more area will be consumed on the FPGA [2.1]. From an upper limit of xyz seconds to lower limit of xyz/100 seconds, many possible implementations may exist between the two boundaries. Such implementations are possible by migrating one or more SW functions to HW. The remaining functions are executed in SW depending on the resource constraints. For e.g. the DCT functions can be implemented in HW, while the remaining two functions can be in SW. Such an exploration in the design process gives rise to HW-SW co-design.

The aim of co-design approach is to meet the system-level objectives by exploiting the trade-offs between HW and SW in a system through their concurrent design.

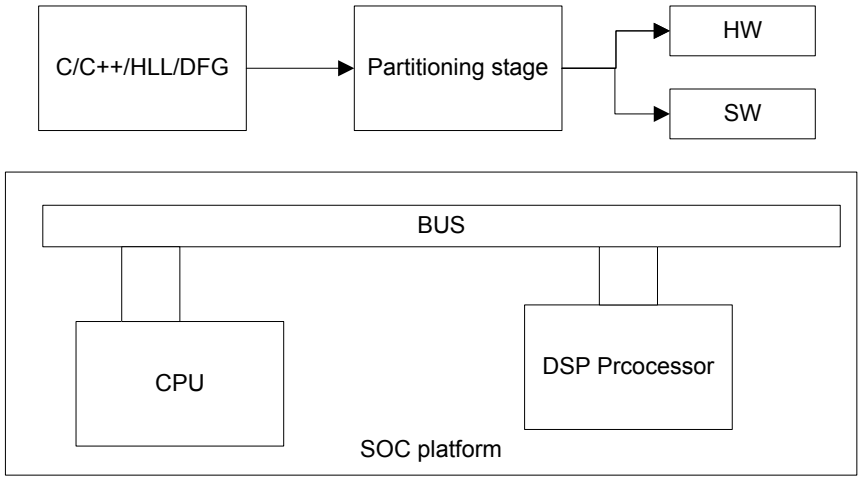


Figure 3.1: SOC platform on FPGA with a design in HW and SW

Fig. 3.1 shows that a single application is divided into two parts (HW and SW) using a partitioning approach. Co-design flow requires a SOC platform and a partitioning approach as shown in Fig. 3.2.

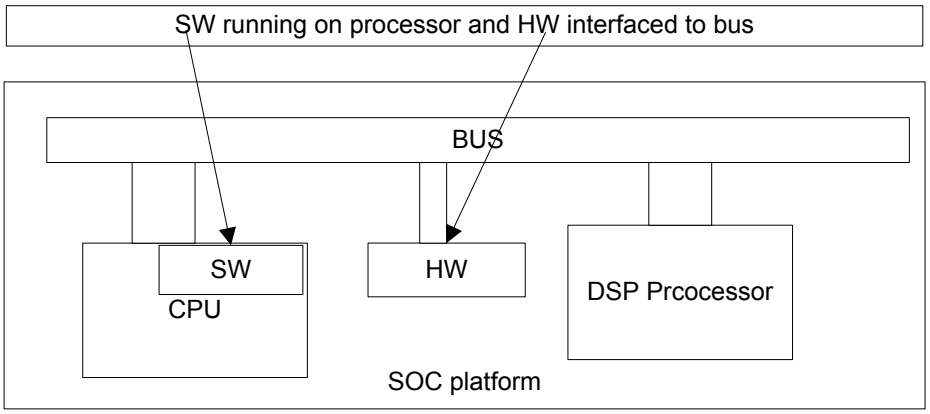


Figure 3.2: Design in HW-SW with bus interface

The HW part is mapped to programmable fabric and SW part is compiled on the processor. HW-SW co-design is a paradigm which aims at achieving system design requirements by migrating design components from top to bottom or bottom to top in a digital system design environment.

It encompasses several problems [3.5]:

- Co-specification: Creating specifications that describe both hardware and software elements (and the relationships between them).
- Co-synthesis: Automatic or semi-automatic design of hardware and software to meet a specification. The co-synthesis problem can be broken up into four principal phases:
    - Scheduling: choosing times at which computations occur;
    - Allocation: determining the processing elements (PEs) on which computations occur.
    - Partitioning: dividing up the functionality into units of computation.
    - Mapping: choosing particular component types for the allocated units. These phases are, of course, related. However, as in many design problems, it is often desirable to separate these problems to some extent to make the problem more tractable.
- Co-simulation: Simultaneous simulation of hardware and software elements, often at different levels of abstraction.

Hardware/software partitioning introduces a design methodology that makes use of several techniques that will become important in other styles of co-synthesis as well.

- The functional specification must be partitioned into processes, each of which denotes a

thread of execution. The best way to partition a specification depends in part on the characteristics of the underlying hardware platform, so it may make sense to try different functional partitioning.

- The performance of the function executing on the platform must be determined. Since exact performance depends on a great number of hardware and software details, we usually rely on approximations.
- The processes must be allocated onto various processing elements of the platform.

The design flow as shown in Fig. 3.3 describes the synthesis of a specification into HW and SW components, interface generation and output verification. It shows that an ideal co-design tool that takes a high level specification and outputs the HW part on ASIC/FPGA and the SW part which can be compiled on the processor. The tool can take specification at a high level of abstraction and should be able to decide the part which should be migrated to HW. It should then produce the required HW and SW drivers for easy interfacing. Such a fully automated approach is still missing in the current existing tools.

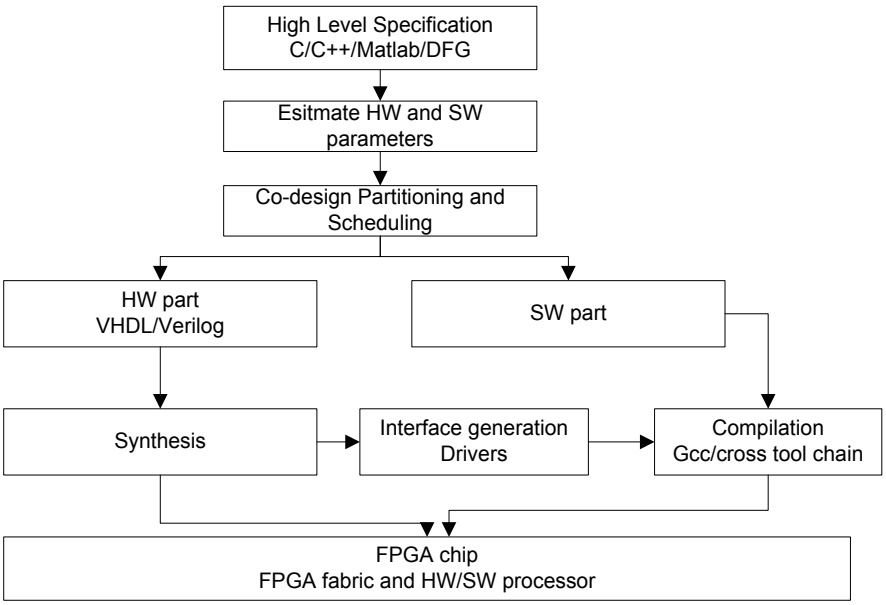


Figure 3.3: Co-design tool

The co-design aims at performance exploration without a real HW implementation by performing co-simulation and co-synthesis. This requires that both HW and SW are executed on a simulation tool and the system parameters are extracted for performance comparison. The two crucial parameters for system design are execution time and area consumed by the various parts of an application. The various parts of the application are defined by the level of abstraction chosen to describe the application such as functions, modules etc. A partitioning

stage requires execution time and area of each function declared in the specification. Based on the given constraint and these parameters, it generates the best HW-SW solution. Hence robust techniques are required to estimate these parameters.

The co-design flow is advantageous, but comes with following challenges [1.10]:

- Following a well defined design flow which can assist the designer in partitioning process.
- The HW development can be done manually in hardware description language for an efficient design and requires comprehensive knowledge of FPGA resource usage.
- Interface knowledge is required to interface the IP core to the communication architecture.
- Setting up the performance measuring and debugging tools to the test the performance of the overall design is required.

3.3. Estimation of SW Performance Using Time profiling

Profiling is a technique by which we can determine the behavior of a program in terms of the amount of time a function takes, the number of call invocations and memory references made. The co-design flow requires an appropriate profiler to detect the functions that contribute to a large percentage of program execution. There are different types of profilers which aim at one or a combination of these features. The examples of time profiler include UNIX prof, GNU gprof, callgrind, Intel VTune, IBM Quantify, Visual C++ profiler, Matlab profiler etc. [3.7]. Among them, gprof is available in most of the toolchains, hence it is popularly used.

The profiling can be classified into intrusive or non-intrusive nature. The intrusive profiling instruments the code during compilation, hence incurs some amount of overhead. Gprof [3.8, 3.9], an intrusive profiler, is widely accepted open source profiler and Xilinx EDK tool chain also renders same profiler in its tool chain, hence it has been chosen in this flow. Two kinds of profile are generated by gprof, one is known as flat profile and other one is known as callgraph profile. Table 3.2 and 3.3 show the two profiles for the sample program which has 7 functions which are named as f1 to f7. The commands used in Linux operating system to generate the profile are given below:

$ cc -g -c myprog.c utils.c -pg *// compile with pg option to create object file*

$ cc -o myprog myprog.o utils.o -pg *// create executable*

$ ./myprog //execute the program // execute the program to generate the gmon data file

$ gprof myprog *// generate the statistics*

Table 3.2: Flat profile

% time	Cumulative seconds	Self seconds	Calls	Self s/call	Total s/call	name
25.04	8.15	8.15	1	8.15	8.15	F7
21.41	15.12	6.97	1	6.97	6.97	F6
17.85	20.93	5.81	1	5.81	5.81	F5
14.29	25.58	4.65	1	4.65	4.65	F4
10.72	29.07	3.49	1	3.49	3.49	F3
7.13	31.39	2.32	1	2.32	2.32	F2
3.56	32.55	1.16	1	1.16	1.16	F1

The interpretation for flat profile as shown in Table 3.2 can be described as follows:

- Column 1: It shows the percentage of the total execution time program spent in this function. These should all add up to 100%.
- Column 2: It shows the cumulative total number of seconds spent in these functions, plus the time spent in all the functions above this one.
- Column 3: It shows number of seconds accounted for this function alone.
- Column 4: It shows number of times the function was invoked.
- Column 5: It shows average number of seconds per call spent in this function alone.
- Column 6: It shows average number of seconds spent in this function and its descendents per call.
- Column 7: It shows name of the functions.

The interpretation of call profile as shown in Table 3.3 can be described as follows: Each row describes the function's descendents and child.

- Column 1: It shows unique index of the function.
- Column 2: It shows percentage of the total time spent in this function and its children.
- Column 3: It shows total amount of time spent in this function
- Column 4: It shows total time propagated into this function by its children.
- Column 5: It shows number of times this parent called the function and total number of times the function was called.
- Column 6: It shows current function.

Table 3.3: Callgraph profile

index	%time	self	children	called	names
[1]	100.0	0.00	32.55	1/1	main [1]
		8.15	0.00	1/1	f7 [2]
		6.97	0.00	1/1	f6 [3]
		5.81	0.00	1/1	f5 [4]
		4.65	0.00	1/1	f4 [5]
		3.49	0.00	1/1	f3 [6]
		2.32	0.00	1/1	f2 [7]
		1.16	0.00	1/1	f1 [8]
[2]	25.0	8.15	0.00	1/1	main [1]
		8.15	0.00		f7 [2]
[3]	21.0	6.97	0.00	1/1	main [1]
		6.97	0.00		f6 [3]
[4]	17.8	5.81	0.00	1/1	main [1]
		5.81	0.00		f5 [4]
[5]	14.3	4.65	0.00	1/1	main [1]
		4.65	0.00		f4 [5]
[6]	10.7	8.15	0.00	1/1	main [1]
		8.15	0.00		f3 [6]
[7]	7.1	2.32	0.00	1/1	main [1]
		2.32	0.00		f2 [7]
[8]	3.6	1.16	0.00	1/1	main [1]
		1.16	0.00		f1 [8]

Gprof instruments program to count calls and samples the program counter every 0.01 seconds in general purpose operating systems such as Linux. For example, if the total samples are 10 then execution time is equal to 100 milliseconds. It creates a histogram of each function and if the program counter value corresponds to that function, an entry is made. Any function which runs faster than 10 ms may shows zero time. This is because samples of the program counter are taken at fixed intervals of run time. This attributes to the statistical inaccurate nature of gprof [3.10] so the run-length of the function should be long enough as compared to sampling period for better results. Secondly the output from gprof gives no indication of parts of program that are limited by inputs/outputs (I/O). The other information given by gprof is number-of-calls made to each functions. This is derived by counting, and not by sampling, hence this is completely accurate and does not varies from run to run.

Although gprof tool is commonly available, but there were certain problems encounters when we executed it on our desktop. During heavy load the accumulated time varied and many

functions showed zero time as no time accumulation was done. The execution time of many functions on general purpose machines is less than 10 ms, since these machines are running at few GHz.

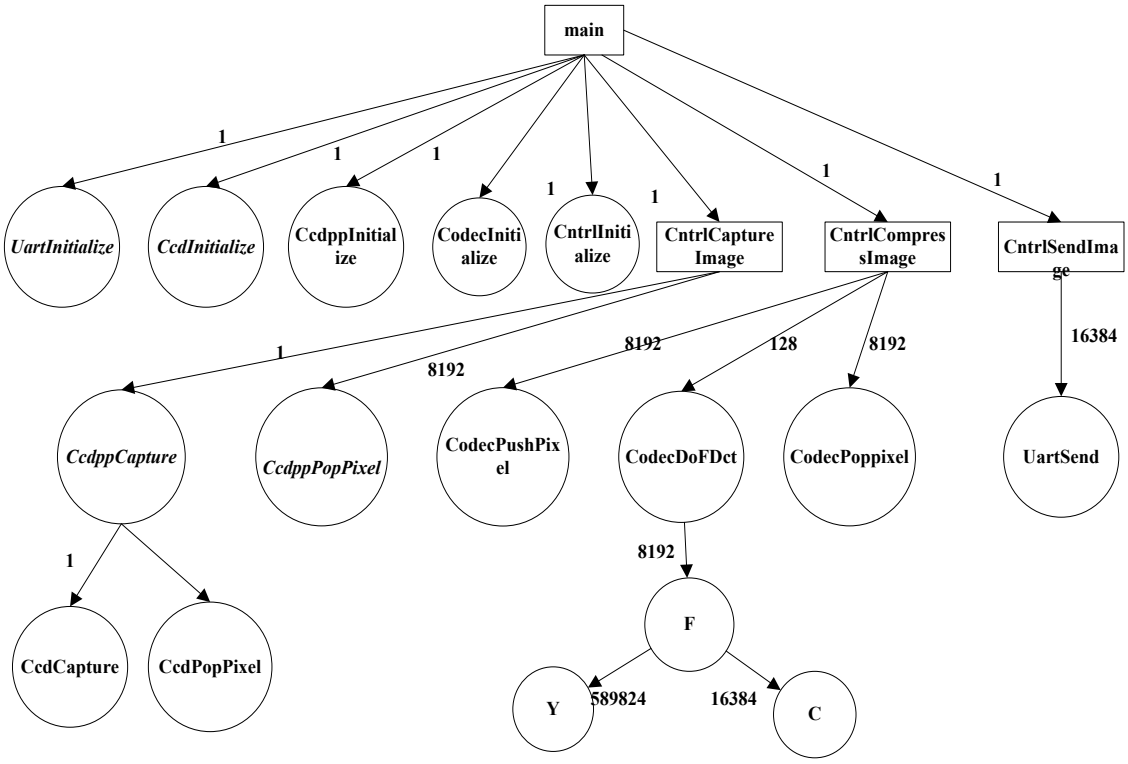


Figure 3.4: Callgraph of digital camera case study

The gprof does not display the call tree structure; hence we have used the pvtarce utility [2.18] for generating the visual diagram of callgraph. Fig. 3.4 shows the callgraph for the C language for digital camera case study [2.1] discussed in previous chapter. The Fig. 3.4 clearly shows the total number of functions present in the application, along with the number of calls at the edge. It also shows the hierarchical structure of the calls and is a good abstraction level for co-design implementation. The gprof cannot accurately show the time taken by each function, hence cannot be used for time estimation of each function on a general purpose machine. In order to capture the exact time, hardware timer has been proposed [3.11, 3.12] to do real time profiling which uses HW counter to trace the running address and store the values. There are other methods that be used for finding the time of each function on the general purpose desktop machine such as Time stamp counter TSC [3.13] which is a good choice for Intel based processors. The time.h C library [3.14] has functions that can be used for measuring the time of a piece of code. The drawback of using these functions is that the resolution is poor. For obtaining better results from gprof utility, we next migrate to time profiling on the target and explore more possibilities.

3.3.1 Time Profiling on Target

Time profiling on the target which usually run at low frequency, can give the better results as compared to profiling on general purpose machines. The Xilinx software development kit allows to parameterize the real time profiling [3.15] by changing the sampling frequency and bin size as shown in Fig. 3.5.

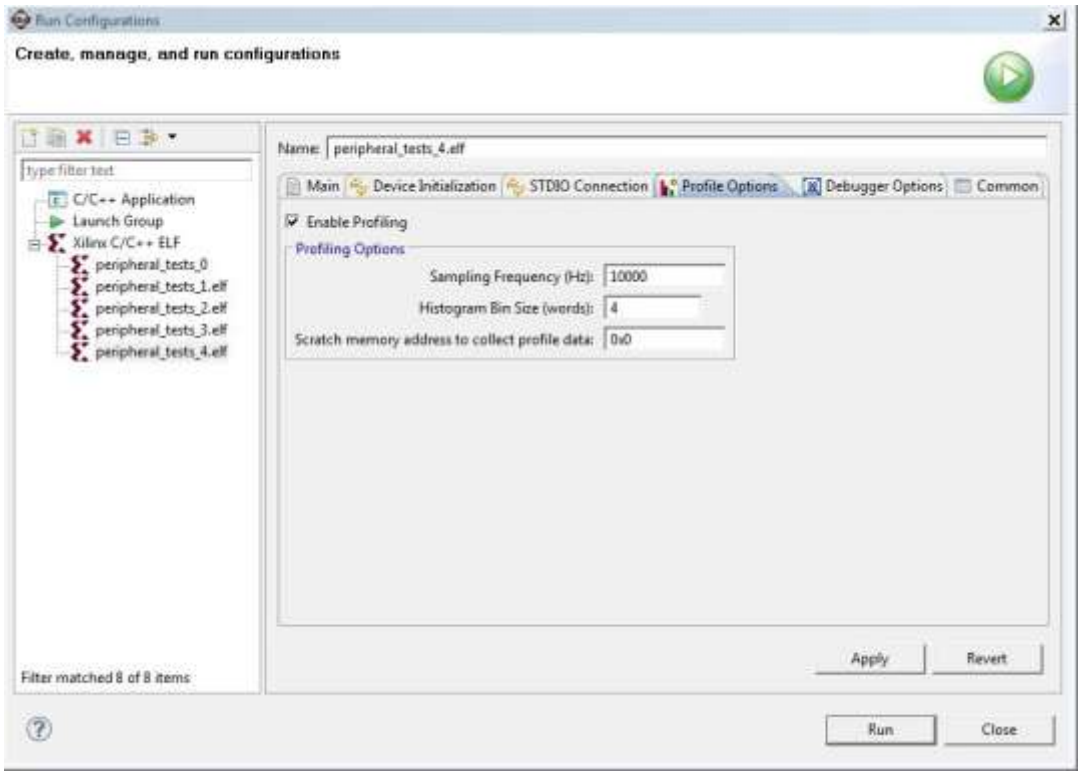
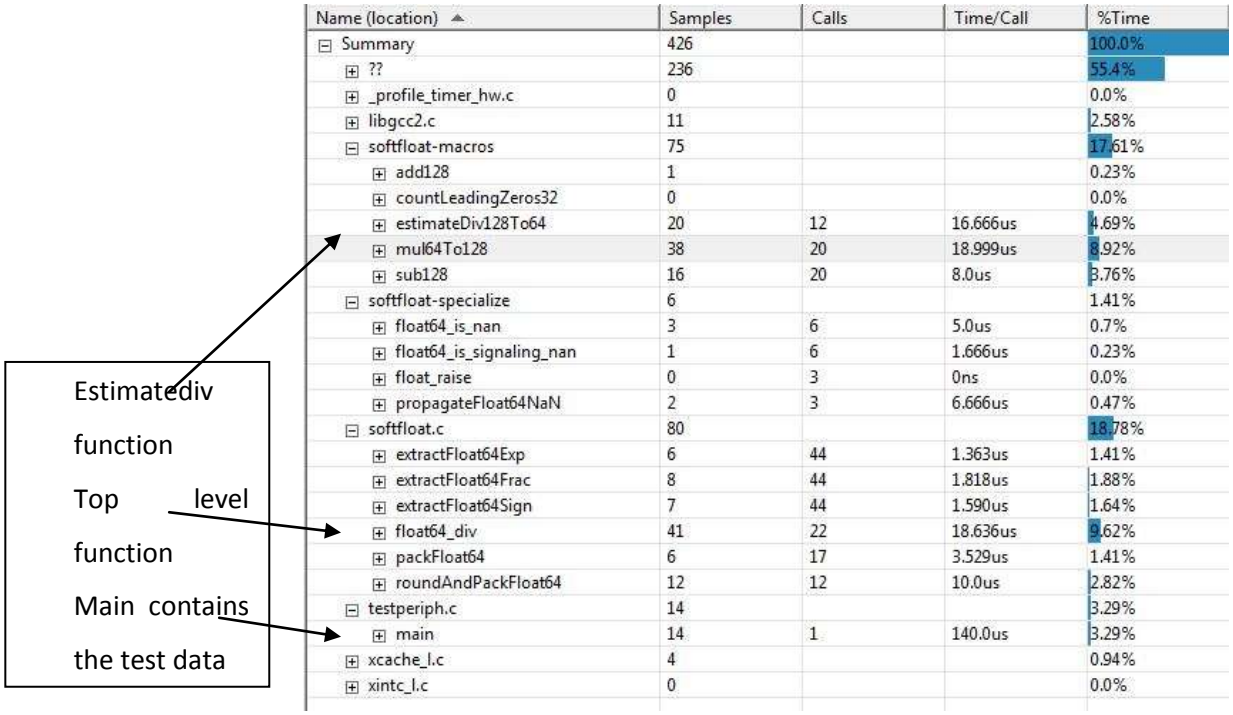


Figure 3.5: Profiling and bin options in SDK

A HW timer is used in the real time profiling to provide the time base and it is interrupt enabled. The sampling frequency determines the rate at which timer interrupt are given to the processor. Setting a higher value of frequency means more interrupts are generated and more samples are taken. This gives better results, but degrades the precision due to more calls being used in interrupt. The bin refers to program text segments and setting a smaller value of the bin allows the results to be better. For example, if we set the bin size to 10 bytes, we can only determine that the program was executing instructions between y and y+10 on each profile interrupt. The DfDiv program from ChStone benchmark [2.50] was profiled on the ML507 kit and results are shown in Fig. 3.6 with sampling rate of 100000 and bin size of 4. This means that the sampling time is 10 microseconds, which is good value for a program that takes milliseconds to run.



Name (location)	Samples	Calls	Time/Call	%Time
Summary	426			100.0%
??	236			55.4%
_profile_timer_hw.c	0			0.0%
libgcc2.c	11			2.58%
softfloat-macros	75			17.61%
add128	1			0.23%
countLeadingZeros32	0			0.0%
estimateDiv128To64	20	12	16.666us	4.69%
mul64To128	38	20	18.999us	8.92%
sub128	16	20	8.0us	3.76%
softfloat-specialize	6			1.41%
float64_is_nan	3	6	5.0us	0.7%
float64_is_signaling_nan	1	6	1.666us	0.23%
float_raise	0	3	0ns	0.0%
propagateFloat64NaN	2	3	6.666us	0.47%
softfloat.c	80			18.78%
extractFloat64Exp	6	44	1.363us	1.41%
extractFloat64Frac	8	44	1.818us	1.88%
extractFloat64Sign	7	44	1.590us	1.64%
float64_div	41	22	18.636us	9.62%
packFloat64	6	17	3.529us	1.41%
roundAndPackFloat64	12	12	10.0us	2.82%
testperiph.c	14			3.29%
main	14	1	140.0us	3.29%
xcache_l.c	4			0.94%
xintc_l.c	0			0.0%

Figure 3.6: Software profiling results of DFDIV on ML507 board on PowerPC@400MHz

All the functions used during the real time execution are shown in the profiling results. This includes all the functions running with elf file (BSP functions) which are shown as 55.4%. Float64_div is the top level function and calls all other functions. The best candidate for HW migration is estimateDiv128To64 function which calls mul64to128 and sub128 and as it consume 17% out of 44.6% of time. The total time taken by DfDiv using profiling is calculated using top function which is float64_div will be equal to 80 samples x 10 microseconds = 800 microseconds. Similarly 120 us is taken by roundAndpackFloat64, 60 us will be taken by propogateFloat64Nan, 60 us will be taken by estractflaot64Exp, 80 us will be taken by estractflaot64Frac, 70 us will be taken by estractflaot64Sig. Adding these values we get 390. Since float64_div calls estimateDic128To64, 800 - 390 = 410 is taken by estimateDic128To64. Hence we now know the time parameter for each function and these values will be used in partitioning algorithm in next chapter.

3.4. Estimation of HW Resources Using LLVM Compiler

This section presents a method to use LLVM compiler [2.52] to perform high level synthesis and estimate the resources required. High level synthesis is the process in which the sequential codes written in languages such as C/C++ is converted to hardware description design as discussed in chapter 2. Using C/C++ to develop and validate the algorithm prior to synthesis is more productive than developing the design at RTL level. Such a migration from a sequential to concurrent behavior descends with many challenges. A high level specification is

written for meeting the functional behavior of the application without concerning to the resources or performance constraints.

C/C++ constructs to RTL mapping is usually done by converting functions to modules, arguments to input/output port, operators to functional units, scalar to wire/reg., arrays to memory and control flow to finite state control logic. Many features of the high level specification such as dynamic memory allocation, continue, break/goto, multi-dimensional arrays, file operation are not the candidate for HW description, hence cannot be synthesized. System calls such as printf() or fprintf() cannot feature in the hardware design and so cannot be synthesized. Pointer casting or using the recursive functions is not supported. Any system calls which manage memory allocation within the system, for example malloc(), alloc() and free(), must be removed from the design code prior to synthesis. The reason for this is that they are implying resources which are created and released during runtime. To be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources. In this section, we address the following issues:

1. Generation of DFGs and CFGs in LLVM Compiler
2. Optimization of C code using LLVM compiler
3. Resource estimation using LLVM compiler
4. Generating extended basic blocks using LLVM compiler

3.4.1. Generating CFG and DFG from LLVM Compiler

Control flow graphs have been frequently used for the automatic generation of HDL from C/C++ specifications. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. Basic blocks form the vertices or nodes in such a graph. A control flow graph (CFG) in literature is a graphical representation, of all paths that might be traversed through a program during its execution. In a compiler, a control flow graph node represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets. The representation of a piece of code in CFG is essential to many compiler optimizations and static analysis tools. CFG is an intermediate representation which carries the control and data flow information. Any compiler can be used to generate CFG like SUIF, LLVM, GCC, and TRIMARAN. The one having the inbuilt pass for CFG generation would be the most optimum one. This pass is available in LLVM [2.52], hence it is a good candidate for the generation of CFGs. Using LLVM, we can generate the machine independent intermediate representation (IR) code and from that the CFG of basic blocks are generated. LLVM is a compiler infrastructure written in C++. It is designed for compile-time, link-time,

run-time, and idle-time optimization of programs written in arbitrary programming languages. Many HLS compilers have used LLVM such as Ctoverilog, PandA, LegUp etc. Clang [3.16] is the frontend of the LLVM complier that converts the C program into an IR [3.17] that is similar to assembly language and useful for performing processing in subsequent compiler stages. *Opt* utility in the LLVM compiler allows various passes to run on the code for doing optimizations such as dead code elimination etc.

For example, a program to find the gcd of two numbers is given below:

```
int gcd(int a , int b){
while(a!=b){
        if(a>b) a=a-b;
        else  b=b-a;  }
 return a;     }
```

The commands used to generate CFG in LLVM are:

- *Line 1: $ sudo apt-get install perl clang llvm*
- *Line 2: $ clang -S -emit-llvm filename.c -o filename.ll*
- *Line 3: $ opt –mem2reg –instsimplify –S filename.ll –o filename.opt.ll*
- *Line 4: $ opt -disable-opt -dot-callgraph -dot-cfg -S p11.ll I-o filename.opt.ll*
- *Line 5: $dot -Tjpg -o callgraph.jpg callgraph.dot*

The command shown in line 1 is used to install Perl, Clang and LLVM. Commands in line 2 emit the human readable IR representation of the C code. Line 3 commands does required optimizations like memory references that are converted to local registers. Line 4 produces callgraph and CFG of each function. Line 5 converts the graphs to jpg format. The generated CFG is given in Fig. 3.7. This graph can be used for scheduling and estimating the resources required for the program.

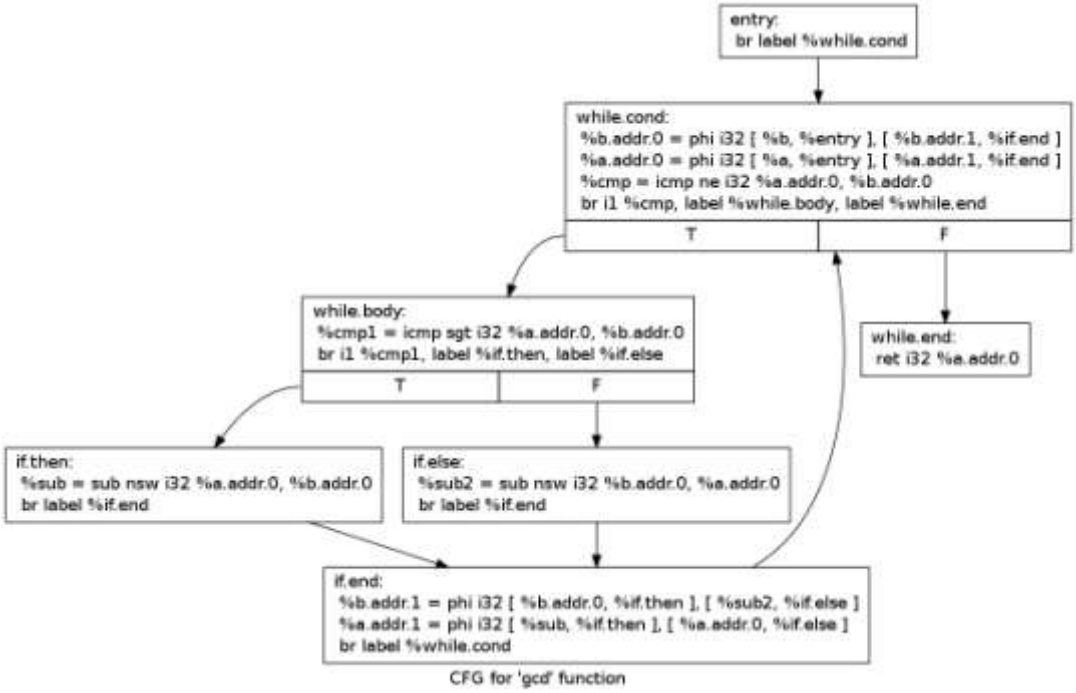


Figure 3.7: Control flow graph of GCD program

The generated graph shows the IR language [3.18], and defines the semantics of ANSI C in LLVM compiler. The control flow graph encapsulates the data edges as it defines the order of execution of the program. To generate instruction level CFG which shows the control edge and data edge, following commands are executed. We have used llvmpy [3.19] a python plugin.

- *$clang -emit-llvm filename.c -S -o filename.bc*
- *$opt -mem2reg filename.bc -o filename1.bc*
- *$llvm-dis -o filenameDFG.ll filename1.bc*
- *$./graph-llvm-ir ./filenameDFG.ll*

Graph-llvm-ir [6.3] is the python script used to generate DFG as shown in Fig. 3.8. The black arrow denotes data dependency and red once show the control dependency.

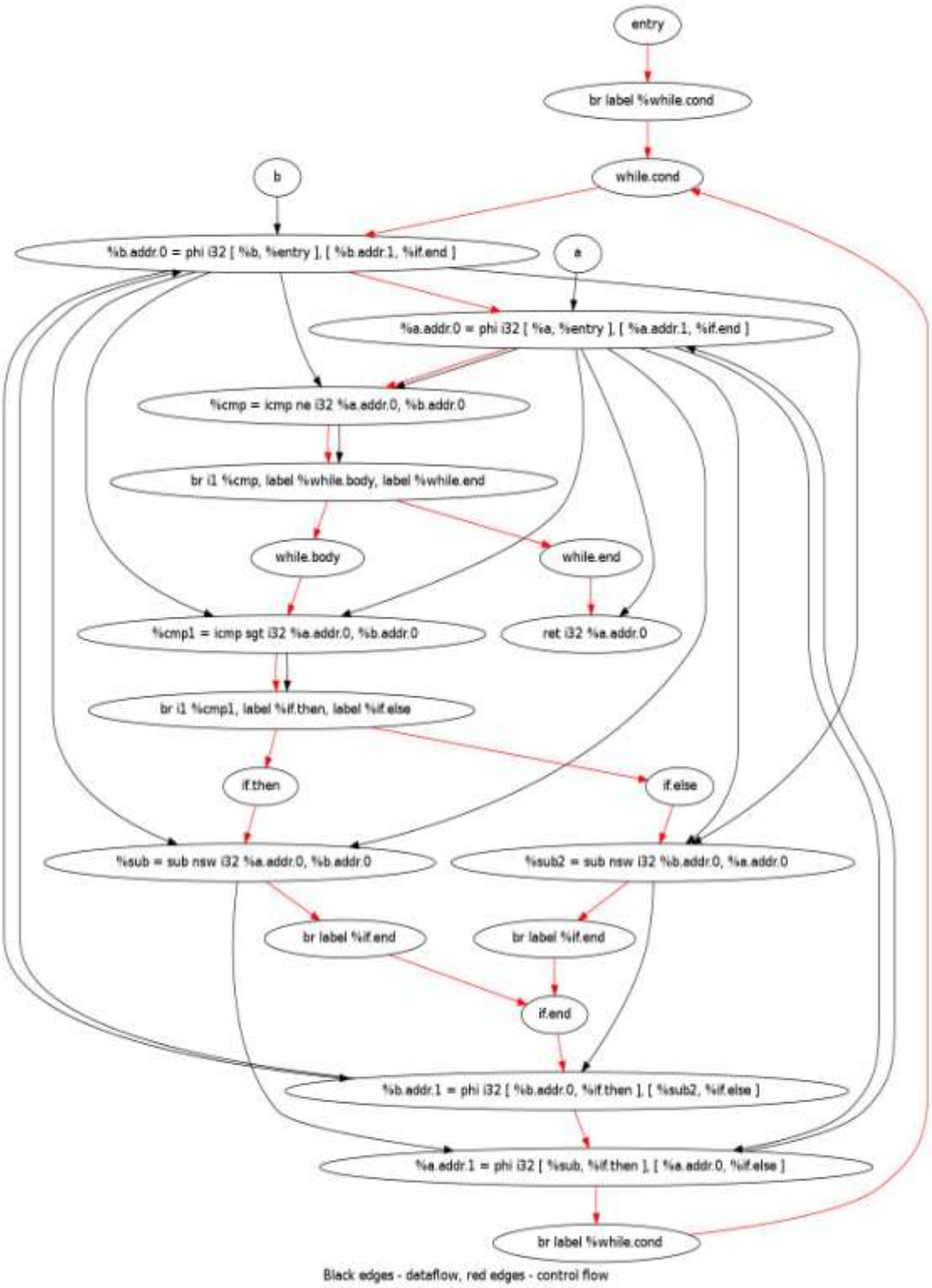


Figure 3.8: Instruction level CFG of GCD program

The above diagram can be divided into two parts to show the control flow and data flow separately:

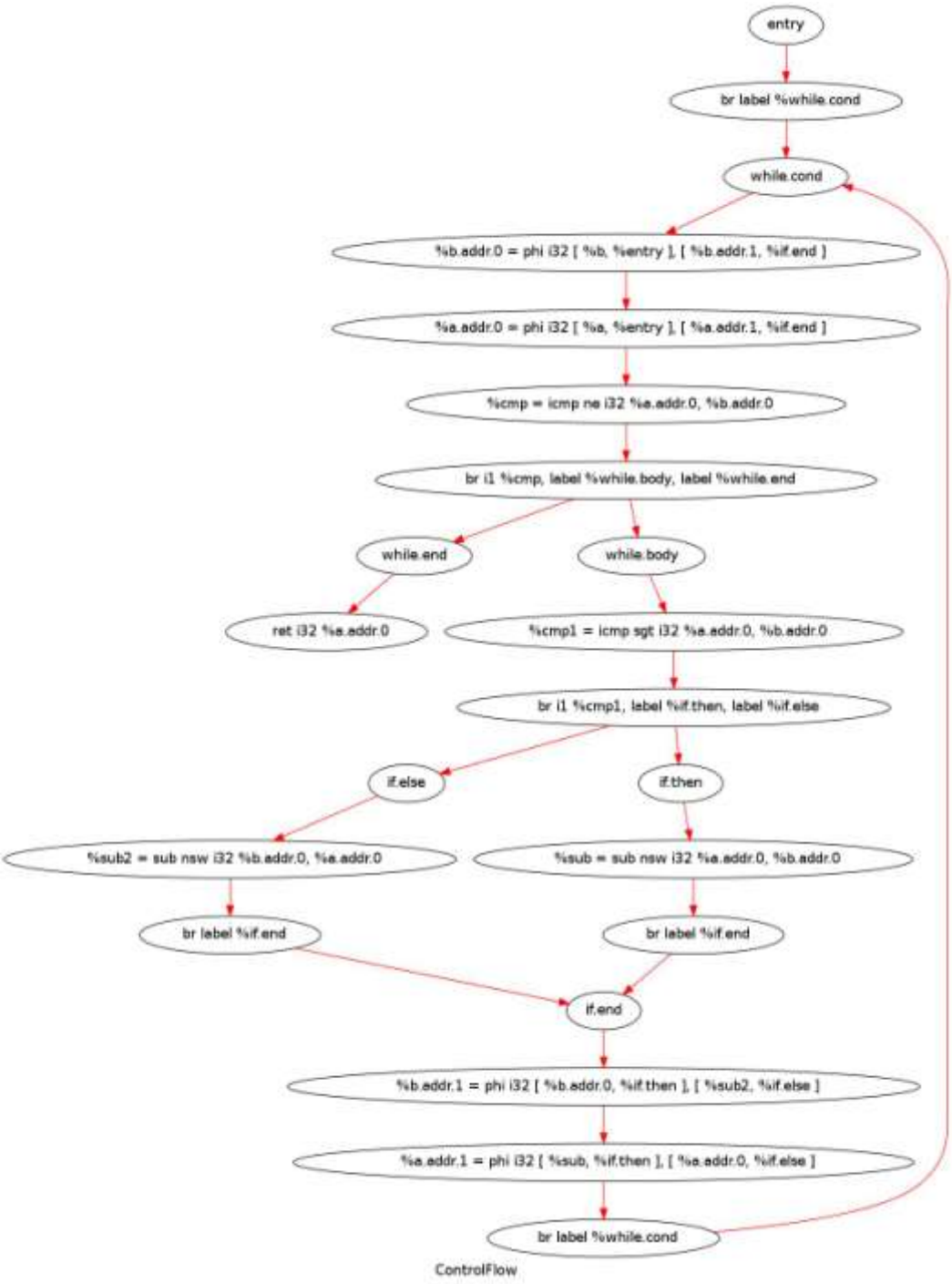


Figure 3.9: CFG of GCD program

For separating the control flow and data flow information the dot file was parsed and resultant is shown in Fig. 3.9 and 3.10. Figure 3.10 is automatically generated using the script and not meant for manual interpretation. For this a perl script was coded to read the edge information and command is given below:

$ perl Sep_CF and DF.pl main.dot

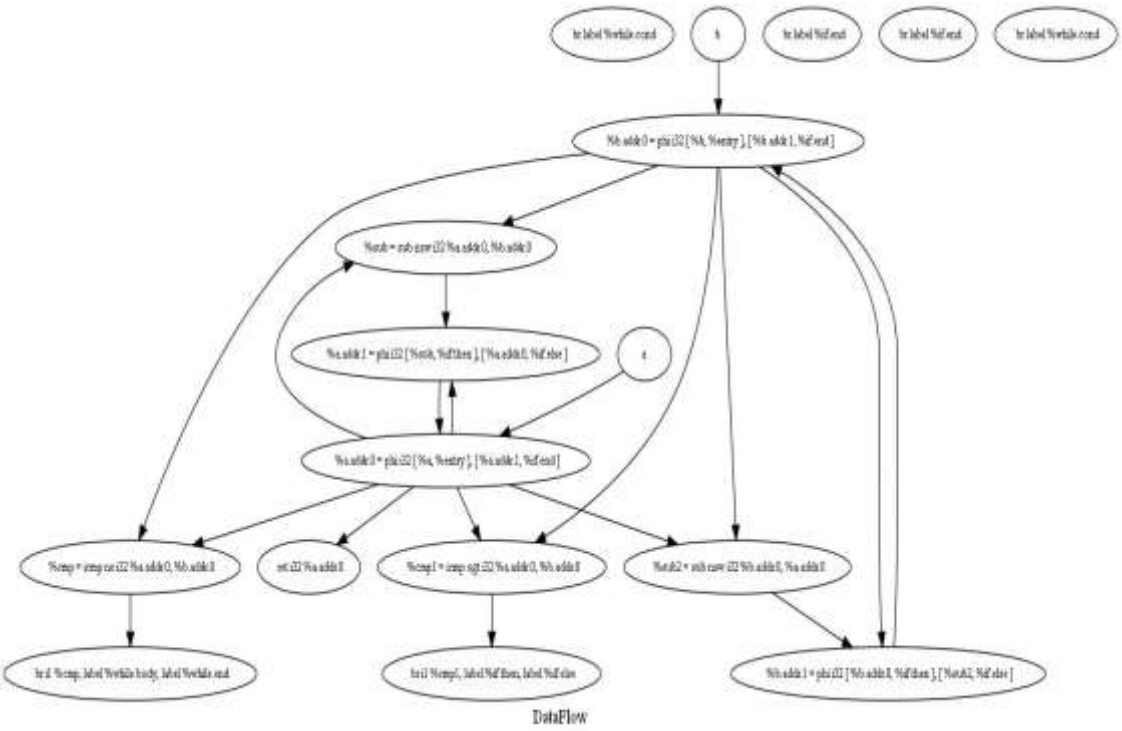


Figure 3.10: DFG of GCD Progrm

3.4.2 Library of Operators in LLVM Compiler

Now the generated control flow graph (Fig. 3.7) is used for generating the Verilog and estimate the resources consumed. An input specification written in simple C is taken, but the entire C specification which includes arrays, pointers is excluded here since our motive is to propose a estimation method and test its effectiveness. This research in not focused on high level synthesis but aims at providing an innovation that can be integrated to an existed tool.

Table 3.4: Library of operators

llvm instruction set: i32	Equivalent hardware unit	LUT
Phinode	Multiplexer +register	32
Phinode	Up/down counter Or acculumator	33
add	Adder	32
fadd	FP Adder	721
sub	Subtractor	32
fsub	FP Sub	874
mul	Multiplier	3/128 DSP
fmul	FP Multiplier	49
Udiv/urem	Unsigned Div	1098
Sdiv/srem	Signed Div	991
Fdiv/frem	FP divider	1941
shl	Shift left	93
ashr	Arithmetic	94
lshr	Logical Shift right	94
and	AND	32
or	OR	32
xor	XOR	32

icmp_eq	equal	11
icmp_ne	Not equal	11
icmp_sge	Signed GE	32
icmp_sgt	Signed GT	33
icmp_sle	Signed LE	32
icmp_slt	Signed LT	33
icmp_uge	Unsigned UGE	32
icmp_ule	Unsigned ULE	32
icmp_ult	Unsigned ULT	33

The Table 3.4 shows the LLVM instructions and their corresponding area. The Xilinx FPGA Virter-5 series having xc5vfx70t-1ff1136 chip on ML507 was used to compile the operators. The Table 3.4 shows the operators that are commonly used in the HDL design. Since the datapath contains arithmetic operators and consumes most of the area, its estimation is a prime concern. In this work we have focused on area hence we have not shown the delay values in the table. In the Table 3.4, column 1 is the LLVM instruction, column 2 is the equivalent hardware unit and column 3 is the area in terms of Look-up tables (LUTs). In order to estimate resources correctly we need so see the types and quantity of resources available in Xilinx Virtex-5. The various kinds of resources available in Virtex-5 are slice registers with LUTs and flip-plops, DSP slices and BRAMs. We need to create formula for slices registers and LUT which are the primary resources under the assumption that no operator sharing occurs i.e. one to one mapping for each instruction in CFG to its corresponding hardware block.

3.4.3. Converting C program to HDL

High-level synthesis [1.7], is the design flow to obtain hardware automatically from high level specification. These specifications are sequential in behavior hence the design flow is converted to concurrent execution in hardware. Parallelism is extracted and clock is added to the generated hardware. The generated hardware is in RTL form and described in Verilog or VHDL. The goal of HLS is to let hardware designers efficiently build and verify hardware, by giving them better control over optimization of their design architecture, and through the nature of allowing the designer to describe the design at a higher level of tools while the tool does the RTL implementation. Verification of the RTL is an important part of the process. A Perl script was coded to parse the LLVM IR and convert to Verilog.

The Perl parsing gives three files as outputs FSM, datapath and the top module. The FSM module preserves the control dependency and datapath preserves the dataflow. The top module instantiates the two modules and binds them together. The LLVM IR instructions can be divided into four categories [2.57].

1. Datapath instructions e.g. add, sub, mul,
2. Conditional jumps e.g. whilecond, ifcond.
3. Control flow e.g. br, ret
4. PHI nodes define the incoming branch.

By this classification, the instruction are handled differently during synthesis .

The above synthesis is discussed with respect to Fibonacci the C code as given below:

```
int fibo(int n)
{
 int prev = -1;
 int result = 1;
 int sum;int i;
 for(i = 0;i <= n;++ i)
 {
 sum = result + prev;
  prev = result;
  result = sum;
 }
return result;}
```

The CFG for the above program is shown in Fig. 3.12.

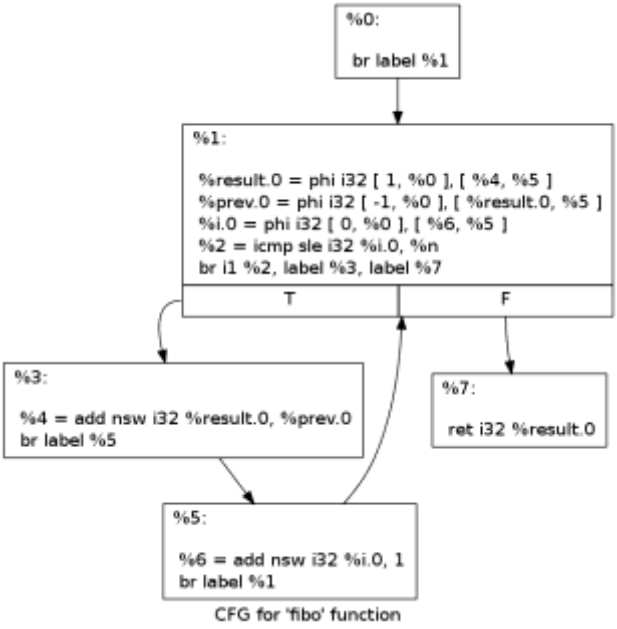


Figure 3.11: Fibonacci series C program and its CFG

The synthesis process starts by modeling each basic block as one FSM state. Each phi node defines the incoming branch which infers a MUX operation. The first line in %1 state means assign the variable *%result.0* takes a value of 1 from *%0* state or a value of *%4* from *%5* state. The select line of the MUX comes from FSM when in state *%1*. The output of the MUX goes to a register which has an enable line from FSM. The phi node with loop variable infers a counter in synthesis, so in Fig. 3.12 for control variable with constant adder is shown. The operators like *icmp* and *add* are synthesized as separate blocks. The output of the comparator is send to FSM to update the state. In state *%7* the program ends and result is the output. Signals generated in FSM are mapped to the datapath in the top level module. As shown in Fig. 3.11 the CFG has five states and so are in FSM diagram as shown Fig. 3.13. The various enable signals are generated when the state changes. Since this is first step for resource estimation, simple programs with no high level constructs have been used like no function calls, memory access.

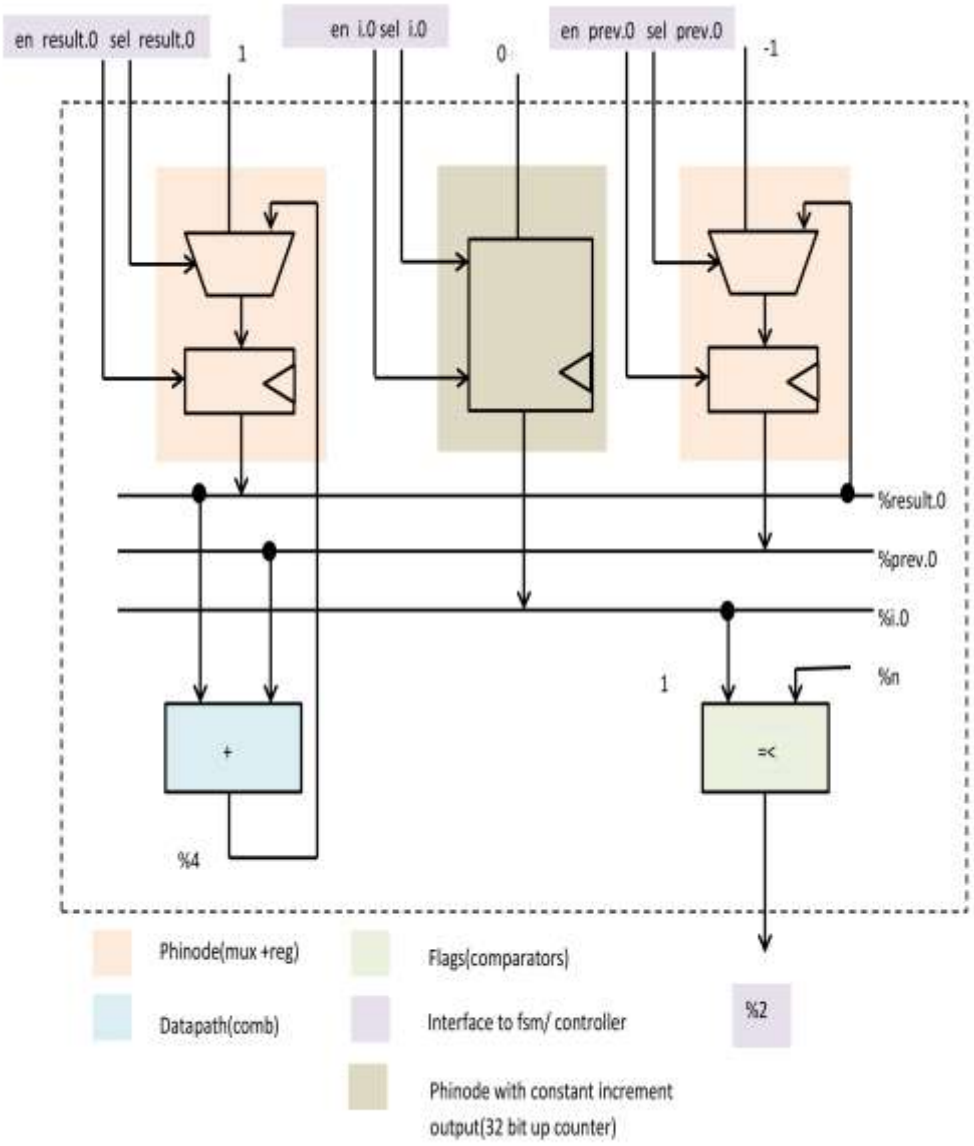


Figure 3.12: Datapath for Fibonacci program

From the CFG we can see that three phi nodes are in state %1, hence Fig. 3.12 shows two multiplexer and one counter. The output of MUX goes to a register, which has an enable signal. Four variable %result, %previous, %i.0 and %n are assigned values in different states. Other variables are temporary variable and are not generated. The datapath contains operators like adder shown in state %3 and is used to do computation. A data bit is also generated by comparator operator like icmp and is used as input in FSM machine.

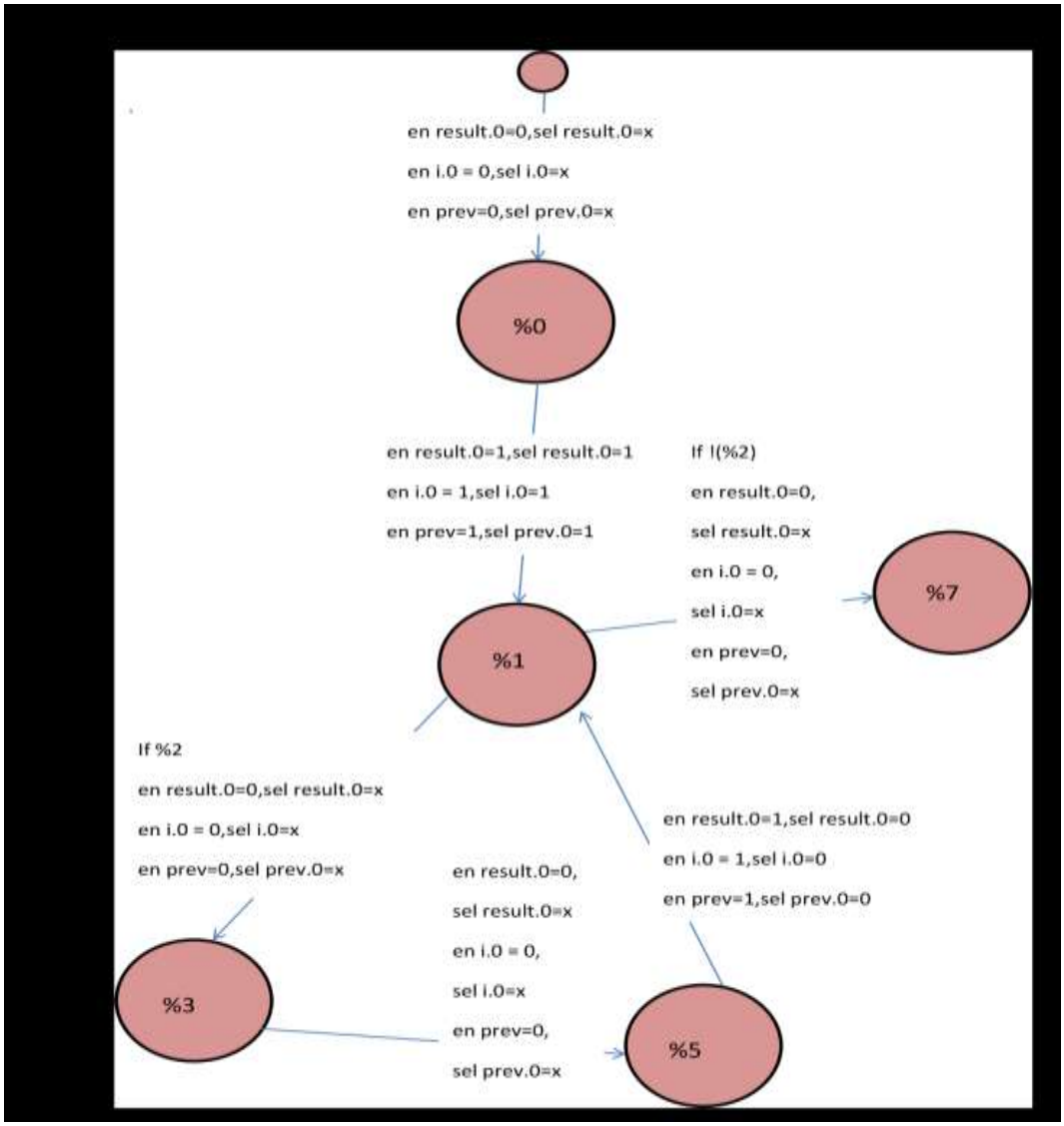


Figure 3.13: FSM of Fibonacci program

The aim of the FSM machine is to execute the code in correct order by generating control enable and select signal based on input coming from operators. Each node of basic block is taken as one state. The default entry for the FSM is state %0 which makes a state transition and goes to %1. In %1 state all the control signals are at the enabled as assignment to phi node happens. This state goes to %3 or %7 based on the condition flag generated from branch instruction which is an input to FSM machine.

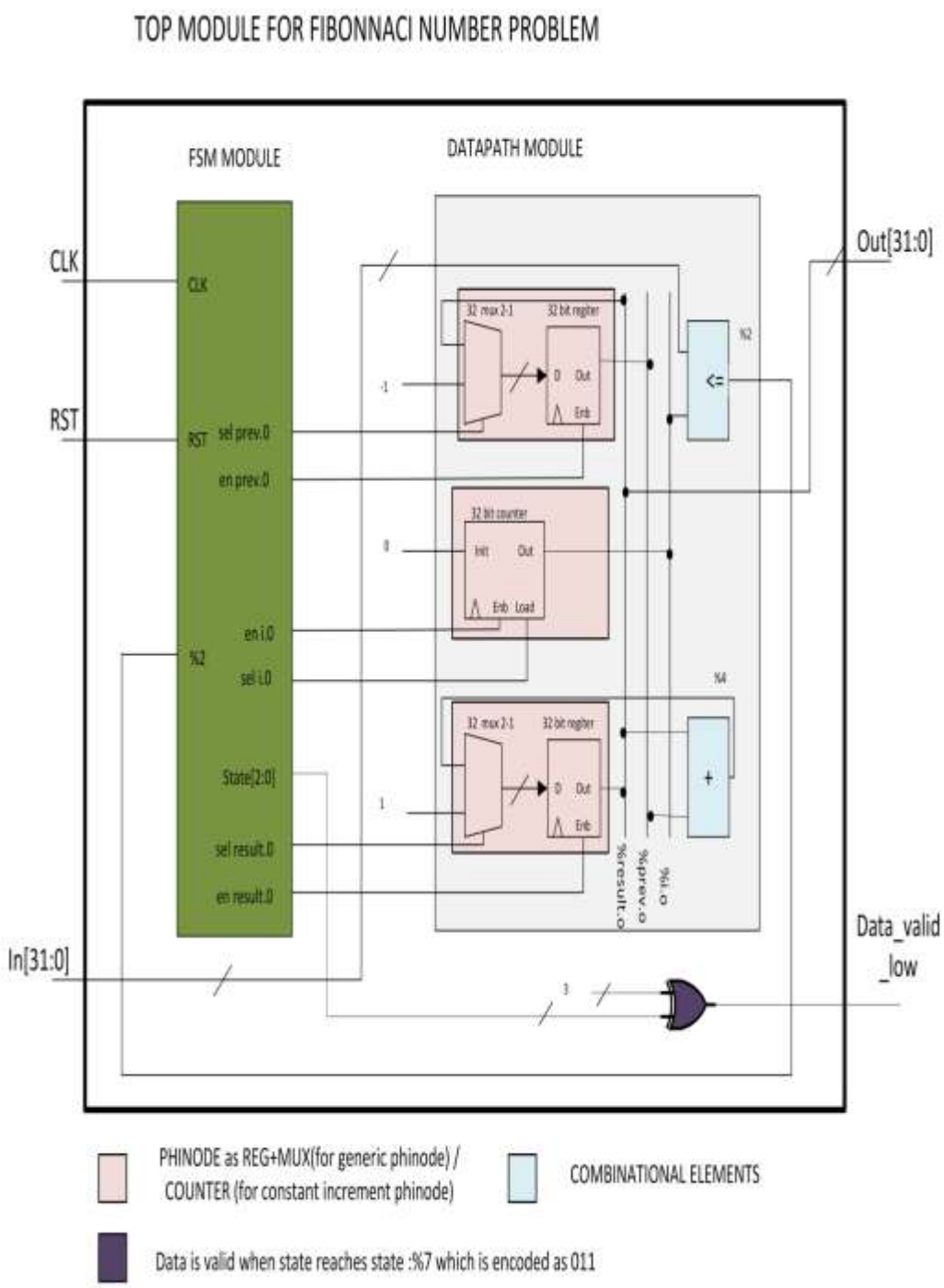


Figure 3.14: Top module for Fibonacci program

The top control flow and dataflow are combined and are shown in Fig. 3.14. This shows that resource consumed by the synthesis can be for the FSM, datapath and top entity combining them.

Many optimizations are can be carried out on the generated HDL. We next show the operator sharing optimization in this section. Operator sharing is used for using single hardware block and implementing multiple operations on it. So, a MUX is placed at each of the inputs of the operator block. The FSM controls the select pin of the MUX. If the next state uses the

operator, then corresponding lines are selected by FSM. Other than *icmp* instruction, all other operations are generally stored in a register through a *phinode*. If the next state to the current state where operation is used contains a *phinode*, then we need place a register after the operator. When this is not the case, then we have to place a register to temporarily store the value, which consumes Flip Flops.

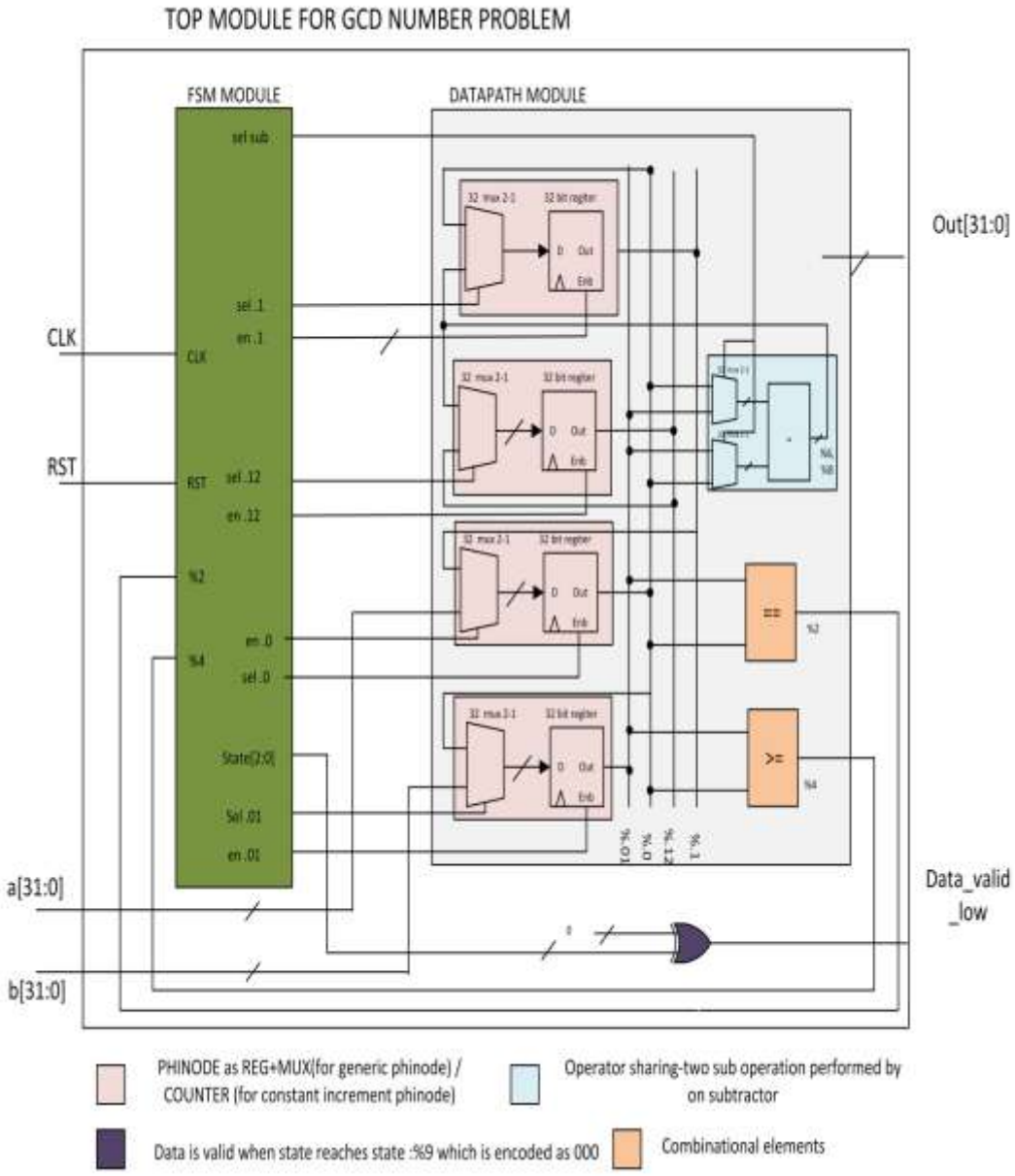


Figure 3.15: Sharing operators program for Fibonacci

The actual resource consumption is measured by number of LUT flip-flop pair used. Generally, number of LUT slices used is larger in number than FF used. So, to save resource usage on FPGA slice LUT utilization must be minimized. Worst case in operator sharing for a particular number of operations is when no inputs are shared between two operations. The implementation of gcd program with operator sharing is shown in Fig. 3.15. The operator

sharing has been compared for three operators and, *add, sub, shl.* From the Fig. 3.16 we conclude that with operator sharing, the slice usage is much less as compared to without sharing. The consumption of resources comes down and can be seen in Fig. 3.16 (a), (b), (c) and (d) as best case. This happens because of reusability of area, but this should only be used in case the tolerance in performance is acceptable.

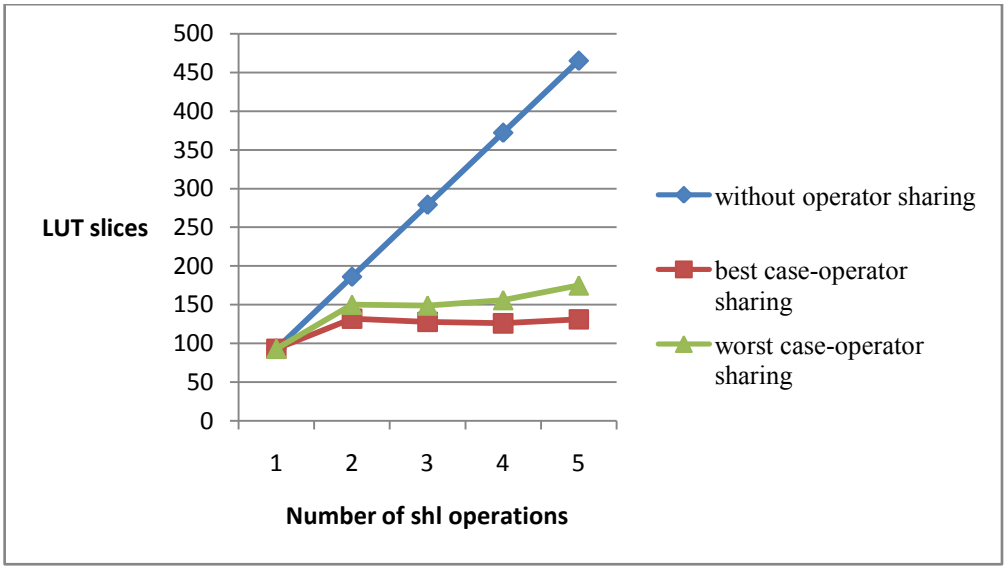


Figure 3.16(a): SHL operator sharing

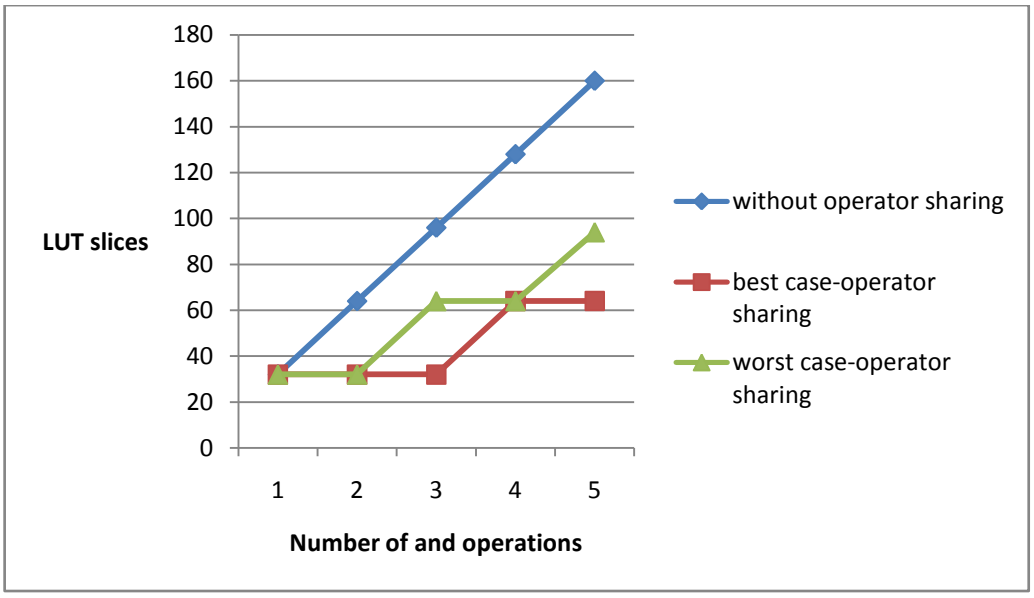


Figure 3.16(b): AND operator sharing

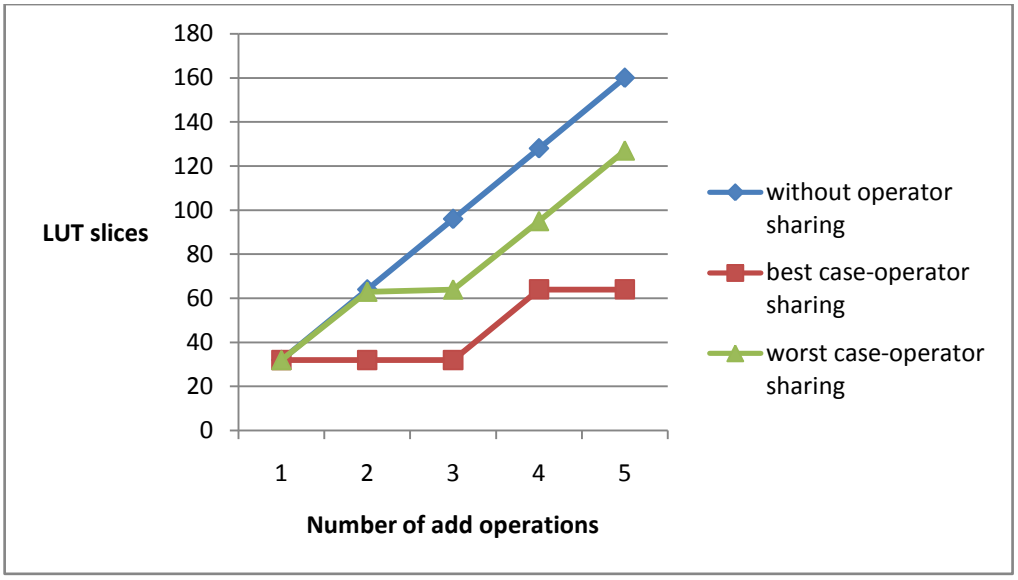


Figure 3.16(c): ADD operator sharing

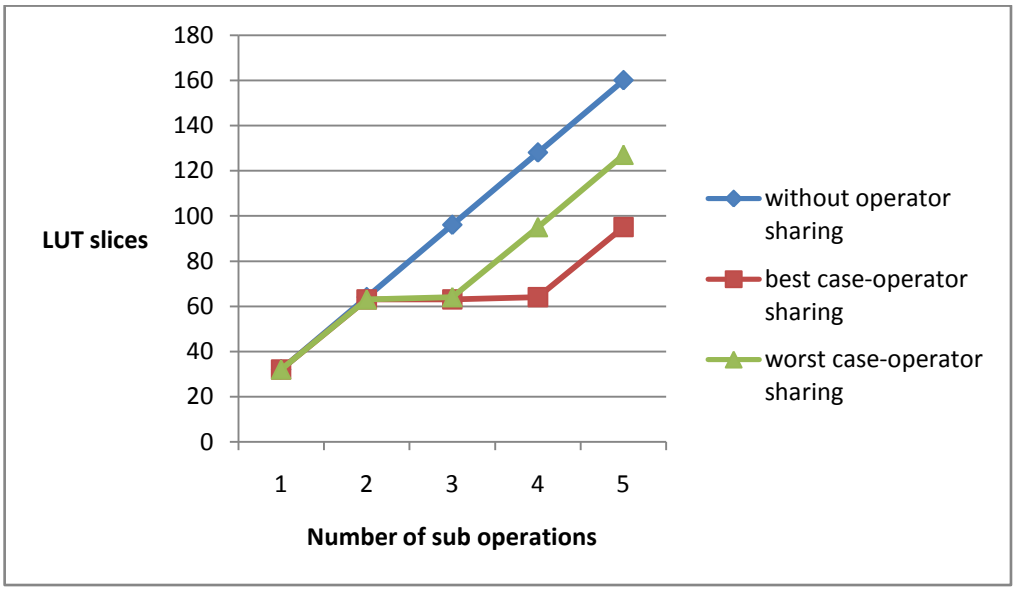


Figure 3.16(d): SUB operator sharing

The total estimated area depends on the area consumed by three entities FSM, datapath and top level, which corresponds to CFG, DFG and their connection. The area consumed by FSM and top level is insignificant and is ignored. The analysis is given below:

$$Estimated_{Total} = FSM_{area} + Datapath_{area} + Toplevel_{area} \quad \ldots(3.1)$$

We now propose equation that can estimate the resource consumption without doing the synthesis discussed above.

Proposed theoretical formula for resource estimation:

*No. of Registers = (number of phinodes) * (width of variable) + [log(number of nodes)/ log2]* …(3.2)

*No. of LUT slices = (LUT slices of hardware) * (occurrences) + LUT slices used in FSM* …(3.3)

Since datapath is 32 bit and the FSM signals are 1 bit wide, LUT slices used by FSM can be ignored without losing much in accuracy. Based on the Table 3.4, we have taken three simple programs: Fibonacci, GCD and factorial using Eq. 3.2 and 3.3:

Case a : Fibonacci series program

Expected slice register = 3*32 + 3 = 99

Expected LUTs = 2*(phi node) + 1*(constant increment phinode) + 1*(32-bit adder) + 1*(icmp_sle) = 161

Case b: GCD program,

Expected slice register = 4*32+3 = 131

Expected LUT = 4*(phi node) + 2*(32-bit subtractor) + 1*(icmp_eq) + (icmp_sgt) = 236

Case c: factorial program

Expected slice register=4*(phinode)+4=132

Expected slice LUT=1*(phi node) + 2*(constant increment phinode) + 2*(icmp_sle) = 194

Based on Eq. 3.3, the algorithm to calculate the LUT usage is shown in algorithm 3.1. We iterate through the CFG and make a entry for DSP slices and LUT usage by each basic block and finally add all the LUT slices used.

Algorithm 3.1: Resource Estimation Algorithm

Create library of resources
No.of LUTs = 0
No.of DSP slices = 0
For each line in LLVM IR
{
If (line==BasicBlock name)
{Make a new entry in the Resource requirement table
LUTs required for the previous block = no. of LUTs

DSP slices required for the previous block = no. of DSP slices

No.of LUTs=0

No.of DSP slices=0}

Else {

Search for matches with library

If a match is found {

No.of LUTs += Matched entry's resource requirement in the library

No.of DSP slices += Matched entry's resource requirement in the library}

}

}

Total LUTs required = Sum of LUTs required for each block

Total DSP slices required = Sum of DSP slices required for each block

End

3.4.4. Comparative Results of Theoretical and Synthesized Programs

In the previous section, we have proposed a formula which can estimate the resources used for implementing a CFG in HW. It is necessary to verify the formula with the real synthesis and compare the results for checking the correctness. The Verilog codes are generated and synthesized using Xilinx ISE. The output waveforms generated for the GCD program is in Fig. 3.17(result = 6).

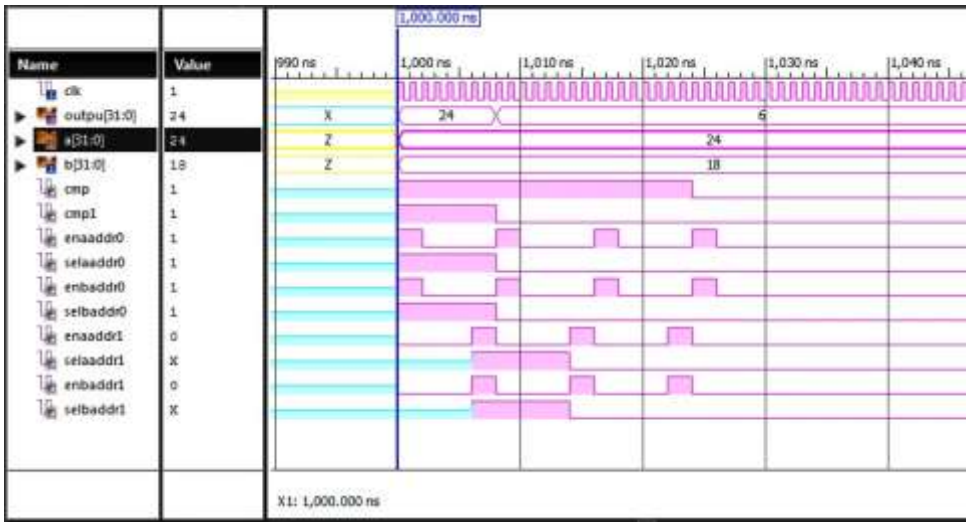


Figure 3.17: Verification of the GCD program

The LLVM IR can be extracted with different optimizations for the convenience of the programmer. It is found that different optimizations give different area.

We have applied various optimizations which are given below:

Optim1:

- -mem2reg, -instsimplify
- mem2reg pass considers memory as register
- instsimplify simplifies instruction and inserts phi nodes

Optim2:

- -mem2reg –lcssa -licm
- lcssa is loop closed single static asiignment form pass. It places phi nodes at the end of loops
- licm is loop invariant code motion. This pass identifies the statements which are inside the loop and whose values are not changing and keeps them outside the loop

Optim3:

- -mem2reg -loop-rotate -loop-reduce
- loop-rotate rotates loop and –loop-reduce reduces the strength of array references inside loops

Optim4:

- -mem2reg -loop-unswitch
- loop-unswitch creates multiple loops wherever it is necessary

Optim5:

- -mem2reg -loop-rotate -loop-unroll
- loop-unroll unrolls the loop. Here the unroll count used is 10. Table 3.5 shows the effect of these optimizations on the resources usage.

Table 3.5: Shows the LUT/DSP resource estimation for each of the optimizations

Function LUT/DSP	**Optim1**	**Optim2**	**Optim3**	**Optim4**	**Optim5**
Gcd	236/0	268/0	311/0	268/0	1859/0
Factorial	320/6	384/6	512/6	384/6	1376/60
Sum of Fibonacci series	192/0	224/0	288/0	224/0	1152/0

Fig. 3.18 shows the graphical representation of LUTs used for different optimizations. It is clear that loop unrolling increases the resource requirement. But with loop unrolling, concurrency can be achieved.

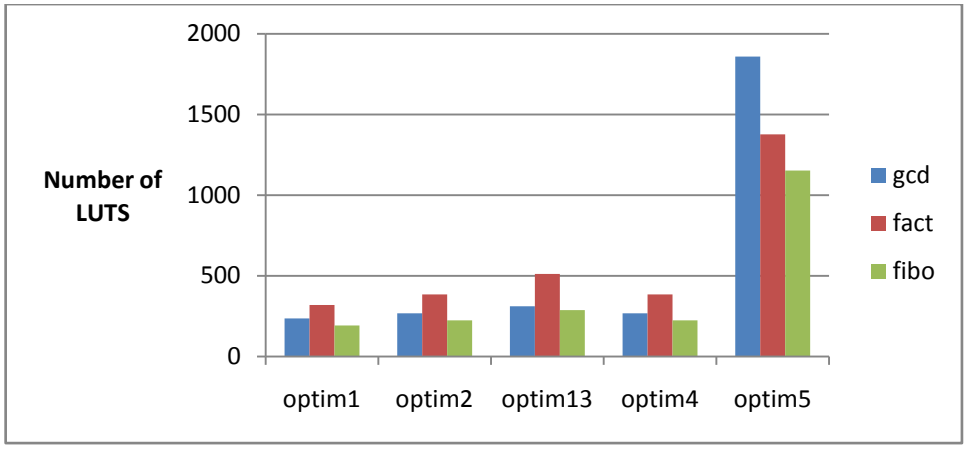


Figure 3.18: Comparison of LUT after optimizations on three programs

Table 3.6: Comparison of resources

Module(1)	Estimate(2)		Actual after synthesis(3)		Vivado-HLS Results
	Slice Lut	Slice register	Slice Lut	Slice register	Slice Lut, registers
Nth Fibonacci number	161	99	164	99	168,96
Gcd function	236	131	240	134	212, 65
Sum of all factorials upto ns	194 + 3/128 DSP	132	199 + 3/128 DSP	136	163 + 4/128 DSP

Table 3.6 shows the comparison of three works, estimate 2 refers to the amount of resources calculated based on formula proposed in Eq. 3.1, 3.2 and 3.3. Column 3 shows the result after the Verilog code generated from the synthesis proposed in section 3.4.3. Column 4 shows the resources used shows by Vivado HLS tool. The Vivado shows much lower values of resources results as compared to other. Hence we conclude that resources for a program can be estimated in different ways and using Vivado-HLS is a good option.

The resource estimation for dfiv program which was used in the previous section (time profiling) was carried out and the estimated results showed in Table 3.7. The callgraph depicting all the functions available in Dfdiv program is shown in Fig. 3.19. The main function contains the test data, hence the floa64_div has been taken as top level synthesizable functions.

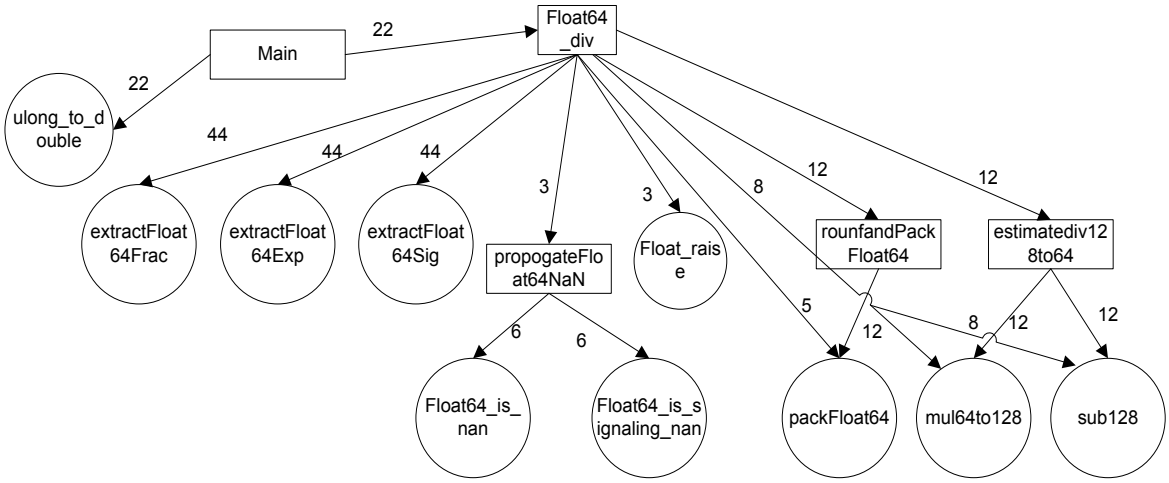


Figure 3.19: Callgraph for DfDiv

Table 3.7: Resource consumption of Dfdiv functions

Name (location)	Calls	DSP48(%)	FF(%)	LUT(%)
main	1			
estimateDiv128To64	12	8	14268	15464
mul64To128	20	The resources added to estimateDiv128To64 as they are sub functions		
sub128	20			
float64_is_nan	6	propogateFloat64NaN as they are sub functions		
float64_is_signaling_nan	6			
propagateFloat64NaN	3	0	0	329
extractFloat64Exp	44	0	168	428
extractFloat64Frac	44	0	296	642
extractFloat64Sign	44	0	258	546
float_raise	3	0	198	486
float64_div	22	24	17196	19824
packFloat64	17	The resources added to roundand PackFloat64 as it is sub function		
roundandPackFloat64	12	0	214	1180

Table 3.7 shows the resource consumed by functions in the dfdiv program(19824) using the estimation method presented in algorithm 3.1. The function estimateDiv128To64 consumes significant resources as compared to all other functions. In the time profiling results, it was shown that this function consumed most of the time in the code.

3.4.5. Creating Extended Basic Block for Task graph generation from CFG

CFG can also be combined together to form clusters known as extended basic blocks for increasing the granularity of a node. A basic block is a sequence of straight line code that can be entered only at the beginning and exited only at the end. To build basic blocks we first identify the leaders (the first instruction in a procedure, or the target of any branch, or an

instruction immediately following a branch (implicit target). Starting from a leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the starting leader. We formally define an Extended Basic Block [3.20] as follows: A maximal sequence of instructions that has no merge points in it (except perhaps in the leader). Extended basic blocks increase the scope for optimization and parallelization. Following is an algorithm that can be used to form extended basic block from a Control Flow Graph containing basic blocks. The comments are given at different places in the algorithm.

Algorithm 3.2: Extended Basic Block

```
all_nodes: set of all nodes
        EBB_roots=root                  //set of root nodes for EBB initially only root
        for every node 'n'
if |Predecessor(n)| >1
set EBB_roots += n

        for every 'x' in EBB_roots
                successor(x) = 'x';
                for every 'y' in successor(x)
                        's'=pop successor(x)
                        EBB(x)=s;
                        for every child 'c' of 's'
                                if 'c' is not in EBB_roots // add the nodes till the roots are found
                                        successor(x) = successor(x) +{c}

finally EBB(x) contains group of nodes making an EBB with entry node 'x';
```

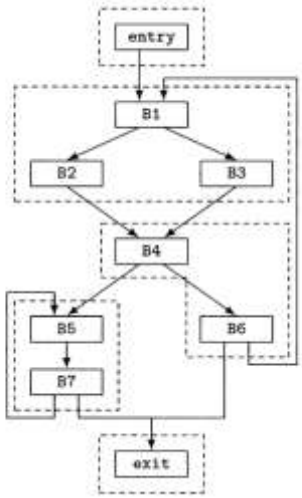


Figure 3.20: Identification of extended basic block [Source: 3.20]

For e.g. B1, B5 and B4 are the roots in Fig. 3.20. The algorithm starts from B1 and clusters B2 and B3 and stops at a root which is B4. To implement the algorithm we first create an

adjacency matrix of CFG of Basic blocks which is input to the algorithm. The result is an output file containing labels of Basic blocks that form Extended Basic Blocks. For the CFG given in Fig. 3.20, we used the dependence matrix given below:

0, 1, 1, 0, 0, 0, 0, 0
0, 0, 0, 1, 0, 0, 0, 0
0, 0, 0, 1, 0, 0, 0, 0
0, 0, 0, 0, 0, 1, 1, 0
0, 0, 0, 0, 0, 0, 0, 1
1, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 1, 0, 0

This gives the EBB = {*(B1, B2, B3), (B4, B6), (B5, B7)}*. Fig. 3.20 shows the CFG created after extended basic blocks are identified. We applied this algorithm in LLVM IR and following commnds and files wre used to generate the ouput graph:

- *llvm-gcc -emit-llvm -S -o program.ll dijkstra.c*
- *opt -disable-opt -dot-callgraph -dot-cfg -S -o p11.ll program.ll*
- *dot -Tjpg -o main.jpg cfg.main.dot*
- *gcc -o dotToMat.out dotToMat.c*
- *./dotToMat.out*
- *gcc -o ebb_final.out ebb_final.c*
- *./ebb_final.out*
- *gcc -o bbtoebb.out bbtoebb.c*
- *./bbtoebb.out*
- *gcc -o adjtodot.out adjtodot.c*
- *./adjtodot.out*
- *dot -Tps AMatrix.dot -o outfile.ps*

Figure 3.21: CFG for Dijkstra

The CFG for the Dijkstra algorithm is shown in Fig. 3.21 (It has many operations hence visibility is low and is parsed by program). Its EBB is shown in Fig. 3.22 and can be used to create coarse level graphs from C programs. These EBBs provide a higher level of abstraction for the algorithm design.

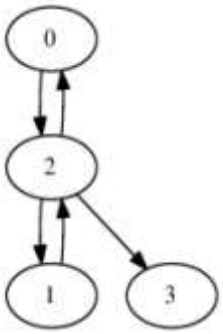


Figure 3.22: Extended basic block for the CFG

We continue the discussion on Vivado-HLS as it was released in 2012 and it need comparative analysis.

3.5. Resource Estimation using Vivado High Level Synthesis Tool

The Xilinx Vivado HLS [1.21] compiler interface is built very similar to eclipse interface which can convert a C/C++ program into HDL and SystemC. It has been developed over a decade of research work. It gives the idea of the amount of resources, latency, frequency of

operation of the application and exploitation of parallelism in HW. We have used Xilinx Vivado HLS version 2013.2 and chose the target product family as Virtex-5 and target device as xc5vfx70t-1ff1136. Many kinds of optimization can be applied during HLS which result in the area/delay trade-off. We discuss these optimizations in brief in the next section.

The synthesis of all the ten ChStone benchmarks [2.50], written in C, was done using the Vivado-HLS tool [2.60]. The results of synthesis are shown in Table 3.8 with respect to four resource components: BRAM, DSP slices(it contains MAC units), slice registers and slice LUT obtained from Xilinx ISE. It can be concluded from the Table 3.8, that AES, Dfsin, JPEG and DfDiv consume more than 40% of the slice LUTs.

Table 3.8: Synthesized results of ChStone benchmarks

Benchmark	Class	Lines of Code	BRAM (%)	DSP48 (%)	FF (%)	LUT (%)
adpcm	Media	550	9	48	5	17
aes	Encryption	289	4	2	12	41
blowfish	Arithmetic	1255	1	~0	1	5
dfadd	Arithmetic	441	~0	~0	2	26
dfdiv	Arithmetic	292	~0	10	18	50
dfmul	Arithmetic	270	~0	6	1	12
dfsin	Arithmetic	580	1	18	23	93
jpeg	Media	1073	24	38	12	57
mips	Processor	271	1	3	~0	4
sha	Encryption	1969	3	~0	1	6

We can infer from the Table 3.8 that many programs consume high resources using automatic synthesis. If area is the constraint for an application then designer may look for HW-SW partitioning of such benchmarks. We chose dfdiv for partitioning process described in chapter 4.

3.5.1. Optimizations in Vivado-HLS

Area and latency are two important parameters that the designer has to keep in mind while developing any digital hardware which are. In most cases the aim of the designer is to minimize area utilization and increase the throughput thus making it an efficient design. These two parameters are inversely proportional to each other and hence both the parameters cannot be optimized simultaneously. The area parameter is usually an optimization metrics, while performance is defined as a constraint metrics. Different kinds of optimizations [1.21] can be applied by the designer to achieve the area/delay trade-offs in a HLS methodology. Some of them are explained below:

- **Function Inlining**: This basically removes the functional hierarchy which saves the time spent in executing the call and return statements from the function every time it is called. This can be used at places where the function is called just once or twice or if there is some kind of dependency which is preventing the top function to be pipelined.

- **Function Dataflow Pipelining**: This is a very powerful technique used to increase the throughput by a huge margin. This basically breaks the sequential nature of the algorithm and performs tasks in parallel as much as possible, so that one function doesn't have to wait for the previous one to be executed completely before it can start. It checks for dependencies and overlaps the operations as much as possible.
- **Loop Unrolling**: This technique tries to carry out a certain number of iterations of the loop in one go unlike the unrolled case where it executes iteration in each clock. This increases the resources on the chip but can prove to be beneficial if the number of iterations is low.
- **Loop Dataflow Pipelining**: It is operated in a similar manner as the functions pipelining, by allowing the parts of the loop that is sequential in nature to occur concurrently at register transfer level (RTL) .
- **Array Partitioning**: Arrays can also be partitioned into smaller arrays. Memories only have a limited amount of read ports and write ports that can limit the throughput of a load/store intensive algorithm. The bandwidth can sometimes be improved by splitting up the original array (a single memory resource) into multiple smaller arrays (multiple memories), which effectively increases the number of ports.

These optimizations were applied to ChStone benchmark and results are discussed next:

In the **adpcm** benchmark, functions dataflow pipelining was applied on the encode function because it contributed to minimum latency. This led to a drop in the latency and interval of the design by almost 80% (((35654-7154)/35654)x100) as shown in Table 3.9(part-a). The resource used goes from 4577 FFs to 7293 FFs.

- Interval defines the number of cycles after which the IP can accept new values.
- Solution-1 is original version and solution 2 is optimized version.
- The clock has been kept at 100 MHz.

Table 3.9 (a): Performance comparison of original and optimized adpcm synthesis and

3.9 (b): Resource usage comparison of original and optimized adpcm synthesis.

	Latency (part-a) cycles		**Interval cycles**		**Resources (part-b)**			
	min	max	min	max	BRAM_18K	DSP48E	FF	LUT
Solution1	28254	35654	28255	35655	26	116	4577	11483
Solution2	7154	7154	7155	7155	24	242	7293	11483

In the **blowfish** benchmark, the array partitioning was applied on the array because it got synthesized into a dual port BRAM, which was constraining the number of reads and writes per cycle to two. Hence, complete partitioning of the array led to more number of reads and writes per cycle thus decreasing the overall latency and interval of the design as shown in Fig. 3.23 and Fig. 3.24. The iv_load variable which was sequentially accessed is converted into a parallel accessible array mapped to dual port RAM. Fig. 3.10 shows the comparison of resource (1090 to 2173) and latency (44002 to 1442) for blowfish.

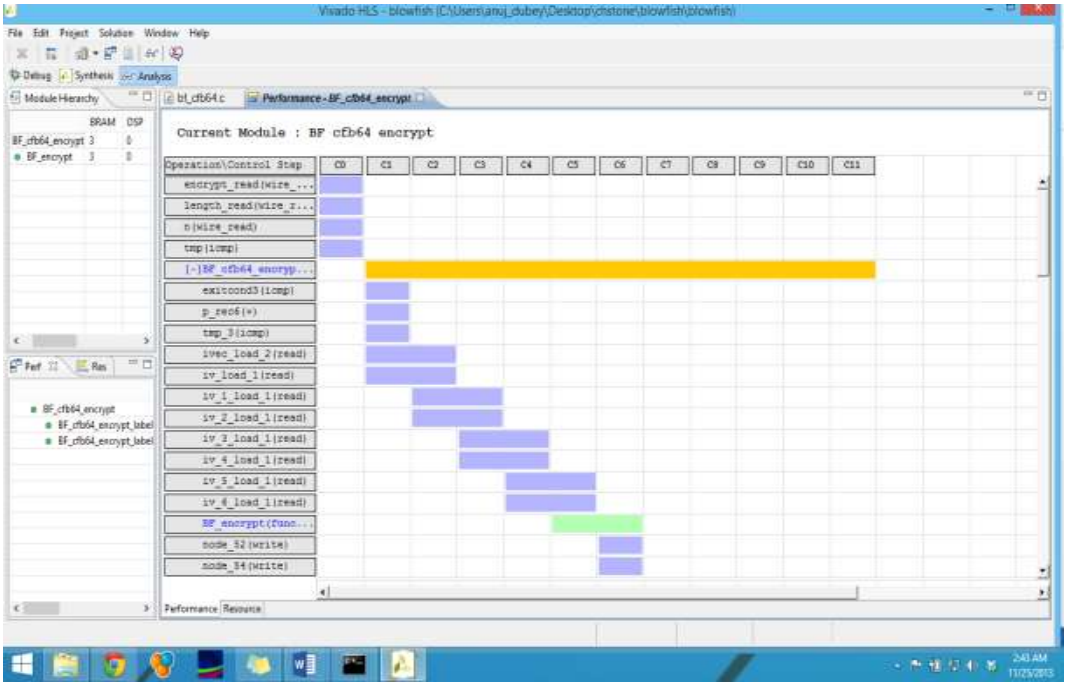


Figure 3.23: Sequential access of array

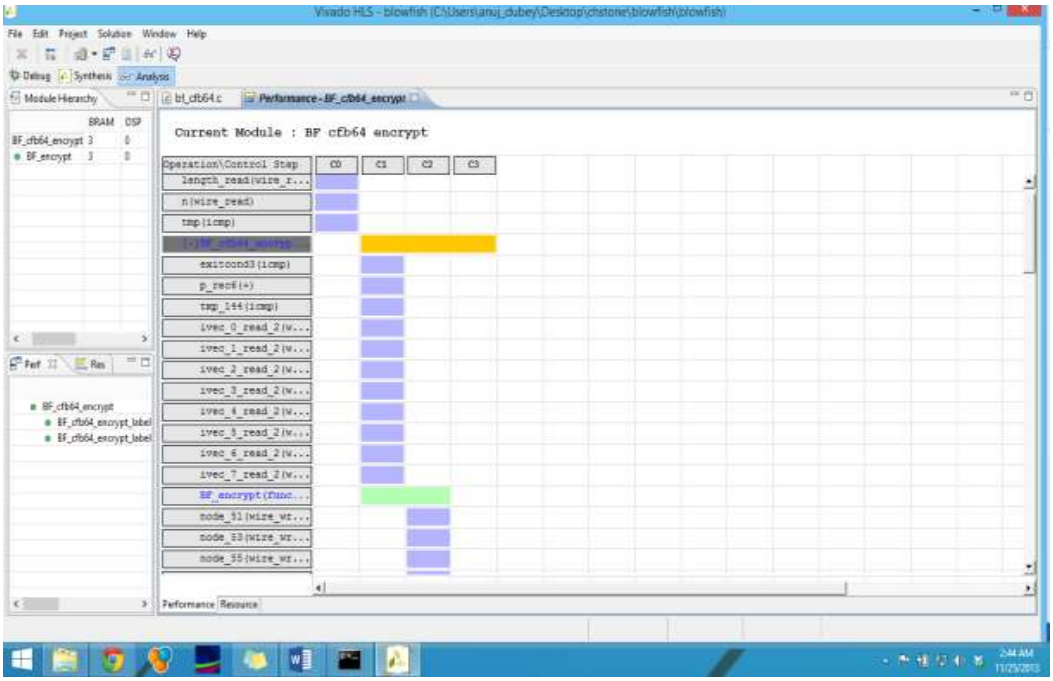


Figure 3.24: Parallel access of array

Table-3.10 (a): Performance comparison of original and optimized blowfish synthesis and

(b): Resource usage comparison of original and optimized blowfish synthesis

	Latency(part-a) cycles		**Interval cycles**		**Resources(part-b)**			
	min	max	min	max	BRAM_18K	DSP48E	FF	LUT
Solution1	2	44002	3	44003	3	0	1090	2173
Solution2	2	1442	3	1443	3	0	2173	1954

In the **dfmul** benchmark, the loop dataflow pipelining directive was applied to the float64_mul function and it led to a decrease in interval from 14 to 2 as shown in Table 3.11 (a) and resource went from 1338 to 1879.

Table 3.11 (a): Performance comparison of original and optimized dfmul synthesis

(b): Resource usage comparison of original and optimized dfmul synthesis

	Latency(part-a) cycles		**Interval cycles**		**Resources(part-b)**			
	min	max	min	max	BRAM_18K	DSP48E	FF	LUT
Solution1	1	14	2	14	1	16	1338	5213
Solution2	14	14	2	2	1	16	1879	5393

In **mips** benchmark, the loop unroll was applied to the three inner loops with constant bounds inside the infinite while loop. This led to a decrease of about 5% in latency. Further the reg array was completely partitioned leading to a further decrease of 8% in the overall latency. This also decreased the time period of each clock cycle increasing the frequency as shown in Table. 3.12.

Table 3.12 (a): Performance comparison of original and optimized mips synthesis and

(b) Resource usage comparison of original and optimized mips synthesis

	Latency(part-a) cycles		**Interval cycles**		**Resources(part-b)**			
	min	max	min	max	BRAM_18K	DSP48E	FF	LUT
Solution1	75	867	76	868	4	8	437	1900
Solution2	27	819	28	801	2	8	386	2120

In the **sha** benchmark, the dataflow pipelining directive was applied on sha_transform function since it majorly contributed to the latency. It led to a drastic decrease of 60% in the latency and interval of the design as shown in Table. 3.13.

Table 3.13 (a): Performance comparison of original and optimized sha synthesis

(b): Resource usage comparison of original and optimized sha synthesis

	Latency Cycles(part-a)		Interval cycles		Resources(part-b)			
	min	max	min	max	BRAM_18K	DSP48E	FF	LUT
Solution1	103587	151605	103588	151606	10	0	1315	2619
Solution2	11067	59085	11068	59086	9	0	10543	26709

In most of the optimized solutions as shown in tables above, more resources were consumed, but performance improved, hence it is the choice of the designer to select solution 1 or 2 based on constraints.

Table 3.14: Comparison with LegUP compiler synthesis results:

Benchmark	Latency (cycles)		Frequency(Mhz)	
	Vivado	LegUp	Vivado	LegUp
	Virtex-5, PowerPC		Cyclone IV, MIPS	
Adpcm	7154	10585	115.74	53
Blowfish	1442	196774	117.51	60
DfAdd	8	788	115.61	102
DfDiv	10	2231	115.74	71
DfMul	14	266	115.74	93
DfSin	NA	63560	114.68	46
JPEG	NA	1362751	115.21	37
MIPS	800	5184	124.84	78
SHA	59085	201746	139.08	58

Since the ChStone benchmark has been developed by LegUp group, it is necessary to compare the optimality of generated HDL from Vivado-HLS vs. LegUp. LegUp is an open source high level synthesis tool developed at the University of Toronto. The LegUp framework allows researchers to improve C to Verilog synthesis without building the infrastructure from scratch. It accepts a vanilla ANSI C program as input, i.e. no pragmas or special keywords are required, and produce a Verilog hardware description as output that can be synthesized onto an Altera FPGA. C printf statements are converted to Verilog $display statements that are printed during a Modelsim simulation, making it possible to compile the same C file with gcc and check its output in the simulation.

In LegUp, we can compile the entire C program to hardware, or we can also select one or more functions in the program to be compiled to hardware accelerators, with the remaining program segments runs in software on the MIPS soft core processor. Compiling the entire program to hardware can give us the most benefits in terms of performance and energy efficiency. However, there may be parts of the program which are not suitable for hardware, such as linked list traversal, recursion, or dynamic memory operations. Table 3.14 shows the

comparison with respect to latency and frequency of operation. It can be seen from the table that on Virtex-5 series of Xilinx, the Vivado performs much better as compared to LegUp. The loop bounds in the C code to be synthesized can either be constants or variable. For certain types of variable loop bounds Vivado can calculate the upper loop bound and give the latency of the design but for some it is unable to do so and hence the results are undefined. Such cases have been mentioned as NA in the Table 3.14.

From the above analysis, we obtained the resource consumption of functions and will be used in partitioning phase. The next section shows how an IP generated from Vivado-HLS can be directly be added to the EDK flow and minimizes the designer's effort in interfacing.

3.6. HW IP Design Integration of IP as a part of SOC

FPGAs are platforms that allow the implementation of the HW synthesized from HDL languages. The application is synthesized and interfaced to the bus in FPGA based SOC design as discussed in chapter 1. The SOC flow encompasses the computation and communication infrastructure for the generated HW. There can be various communication substructures that can be used such as peer to peer connection, network on a chip or a bus design. Bus design has been the most used commonly interface in FPGA IP interfacing technique. This is because it is simpler as compared to others and available in all the SOC platforms.

3.6.1. Case Study for Hardware, Software IP Core Integration Using Vivado-HLS and EDK

This sections aims to clarify the wrapper generation and bus bundling required for IP integration. The same method can be applied to any co-design process. A simple code for adding two numbers was written and its RTL description was generated using Vivado-HLS. Then, the generated RTL IP core was migrated to Xilinx EDK and interfaced [3.21, 3.22] with the Microblaze soft core processor. The hardware design was exported to SDK to create its board support package and user interface. The driver generated from Vivado-HLS where used as application programming interface and were used to write an application code for the generated hardware. The software computation was then complied on the hardware thus demonstrating the concept of usage of automatically generated IP core using Vivado-HLS.

***Source Code**: Input to Vivado*

```
#include<stdio.h>
#include "adder.h"
void adder(int *c,int a,int b)
```

```
{
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle slv0"
#pragma HLS RESOURCE variable=a core=AXI4LiteS metadata="-bus_bundle slv0"
#pragma HLS RESOURCE variable=b core=AXI4LiteS metadata="-bus_bundle slv0"
#pragma HLS RESOURCE variable=c core=AXI4LiteS metadata="-bus_bundle slv0"// AXI bundle for slave interface
*c=a+b;
}
```

A simple adder written in C is defined in the code above and is used input to Vivado-HLS. The pragma compiler directive tells the compiler to bundle the input signals in the AXI bus as slave inputs.

Testbench:

```
#include<stdio.h>
#include "adder.h"
int main()
{
int a,b,*c,result;
a=2;
b=3;
c=&result;
adder(c,a,b);
if(result==5)
   return 0;
else
   return 1;
}
```

The tool requires a test case in which the function to be synthesized is called and inputs are given. Such test case in shown above with a = 2 and b = 3. The generated IP core and its driver are migrated in EDK/SDK flow for achieving the SOC flow. The core generated by the Vivado-HLS tool was copied in the pcore directory of the EDK and interfaced with the Microblaze processor. Similarly the drivers were included in the SDK environments.

SDK Application Code:

```
#include <stdio.h>
#include "xadder_slv0.h"                    //generated from Vivado as header files
#include "xadder.h"                  //generated from Vivado as header files
#include "xparameters.h"
XAdder XA;
XAdder_Config XA_config={0,XPAR_ADDER_S_AXI_SLV0_BASEADDR};

void XAdderStart(void *InstancePtr){
```

```
XAdder *pAd = (XAdder *)InstancePtr;
//Enable ap_done as an interrupt source
XAdder_InterruptEnable(pAd,1);
//Enable the Global IP Interrupt
XAdder_InterruptGlobalEnable(pAd);
//Start the IP
XAdder_Start(pAd);
}

int main()
{
int result,resultm;
XAdder_Initialize(&XA,&XA_config);
XAdder_SetA(&XA,2);              //pass value of 2 to adder
XAdder_SetB(&XA,3);              // pass value of 3 to adder
XAdderStart(&XA);                //execute
result=XAdder_GetC(&XA);         // get the sum
resultm=result*result;           //square the value
xil_printf("%c2J",27);
xil_printf("Final result:%d",resultm);
return 0;
}
```

The above SDK code which uses the driver functions in given input 2 and 3. When the above code is run on the FPGA, the result = 25 is shown on the console screen. Here the addition is carried out by the hardware, whereas the squaring is done by the software as shown by two bold lines in the code.

The above example shows the generation of IP core for AXI bus. The same procedure can be adopted for PLB bus, but the PLB drivers are not generated automatically in the Vivado-HLS. The user has to write the API functions to read and write from the IP core. Clock and reset are manually connected for the core by the designer. The example given below shows the PLB bundle that is used for IP [3.23].

#pragma HLS RESOURCE core=PLB46S variable=a metadata="-bus_bundle CONTROL"

#pragma HLS RESOURCE core=PLB46S variable=b metadata="-bus_bundle CONTROL"

#pragma HLS RESOURCE core=PLB46S variable=return metadata="-bus_bundle CONTROL"

The PLB on-chip [3.24] bus is used in SOC integrated systems and it supports read/write data transfers between master and slave devices equipped with a PLB interface and connected through PLB signals. It is a 64-bit bus format that supports multiple master and slave devices. Listed below are some of its characteristics:

- It can support 32-bit devices.
- Each master device is assigned a priority so that the bus can arbitrate when multiple masters want to use the bus.
- Slave devices are assigned one or more regions of addresses that are responsible for handling read/ write requests.
- The read or write operation takes five bus clock cycles and the maximum clock frequency is 125 MHz.
- PowerPC 405(ML507 Board used in this work has 405) supports two PLB interfaces:
    i. Instruction-side PLB for loading instructions into cache.
    ii. Data-side PLB (read-write) for data cache.

The IP cores interfaced to the bus runs at a high frequency which is usually in MHz and hence the latency of computation is in the order of nanoseconds. This gives the designer a hope of very high throughput in HW. The data transfer takes place from memory to memory through the bus. This means that the processor takes data from cache, sends it to core, the core processes it and results are saved back to cache. This sending/receiving of data incurs a communication overhead. If such a overhead is large, then the advantage of using the HW-SW co-design approach may be lost. Hence it is necessary to understand the underlying bus and its overhead. When an IP core is designed for acceleration, many kinds of optimization and design strategies at different levels of the design flow can be applied to increase performance. Some of the strategies are discussed below:

- Using burst mode of transfer in which packets of data are transferred after the initial address setup.
- Using pipelined mode of transfer if the bus supports. In this mode address and data are overlapped.
- Using data pipelining in the design and pre-fetching the data.
- Allowing the core, bus and processor to run at their maximum allowable frequency.
- Using a direct memory transfer technique to release the processor during communication.

We conclude that migration of C/C++ based application to HW is now accelerated, but there is a need to recognize the performance gain. Next section elaborates the usage of HW timer for calculating the time taken by core for processing the data.

3.7. Hardware Timer

This section describes Xilinx XPS HW Timer [3.25] which has been used in this thesis for time measurement and has been used to find the running time of the applications. The XPS Timer connects as a 32-bit slave on processor local bus (PLB). It is organized as two identical timer modules. Each timer module has an associated load register that is used to hold either the initial value for the counter for event generation, or a capture value, depending on the mode of the timer. The top level block diagram of the XPS Timer/Counter is shown in Fig. 3.25. There are 3 modes in which the timer can operate:

Generate mode: The value of the load register is loaded into the counter. The counter begins the count and on reaching the value, it stops or automatically reloads the generate value. This generates an interrupt if enabled. This mode is useful for generating repetitive interrupts or external signals with a specified interval.

Capture mode: The value of the counter is stored in the load register when the external capture signal is asserted. This mode is useful for time tagging external events while simultaneously generating interrupts. This mode of the timer is used for capturing the number of cycles elapsed.

Pulse Width Modulation Mode: In this mode, the two timers are used as a pair to produce an output signal with a specified frequency and duty factor.

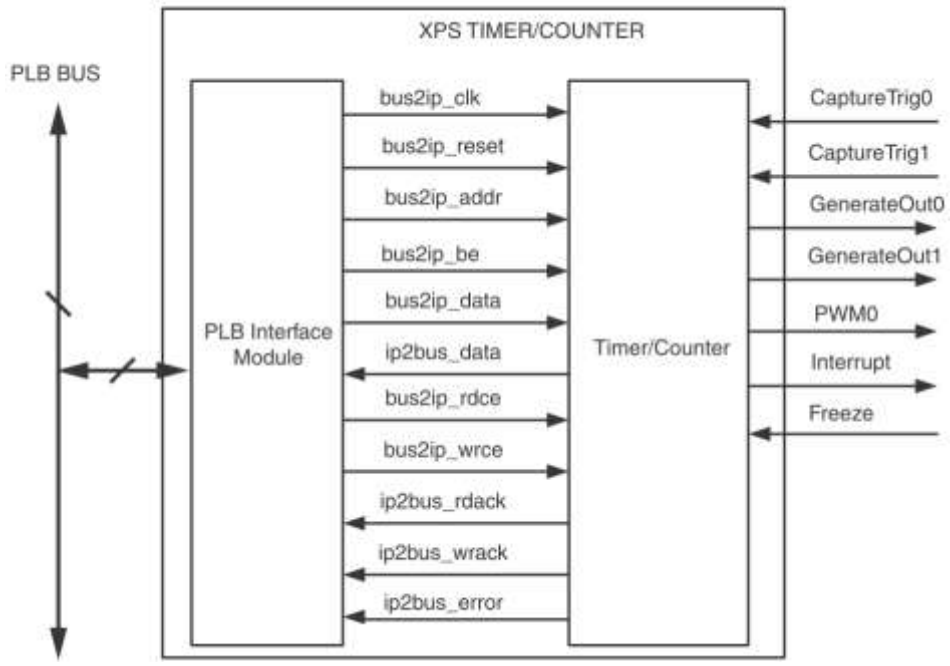


Figure 3.25: XPS timer interface with PLB bus [3.25]

There are three functions we need to use to capture the time. These are as follows:

- *XTmrCtr_Start() :– Starts the timer specified in the arguments;*
- *XTmrCtr_Stop() :– Stops the timer specified in the arguments;*
- *XTmrCtr_GetValue() :– Returns the value of the timer at that moment*;

We declare the following required global variables in the code for creating a timer object –

- *XTmrCtr TimerCounterInst;*
- *Unsigned long cycles;*

The first variable is required to be passed as an argument to the above functions. The second variable is used to capture the value returned by the get value function. After the above initialization, the functions must be called in the following order:

- *XTmrCrt_Reset() – To reset the timer before starting it to any other value;*
- *XTmrCtr_Start() – To start the timer;*
- *XTmrCtr_Stop() – To stop the timer;*
- *XTmrCtr_GetValue() – To get the number of cycles as captured by the timer;*

The events for which the time is required must be between the start and stop functions. It must be noted that the value captured is accurate to few clock ticks and hence, time captured at different runs can be different up to few nanoseconds. This inaccuracy is tolerable as order of execution time is more than milliseconds for critical and time consuming applications. The timer runs at 125 MHz and for all practical purposes, it is sufficient. Also, the operations of starting and stopping the timer take a few clock cycles. This overhead must be subtracted from the total time computation. So, for complete accuracy, the two functions (start and stop) must be written one below the other and the time must be captured. This gives the time taken to initialize the functions themselves. This value can be subsequently subtracted from total time values to get accurate results. For e.g. if the number of cycles = 000012FC and timer is running at 125MHz then the time would be 38880 ns. We found that timer takes 80 cycles for these start/stop functions, hence 80 x 8 ns = 640 ns. So the actual time comes out to be 38880 - 640 = 38240 ns. A demo function showing the timer code is presented in Appendix 1.

3.8. Results of Manual Interface of DfDiv Program as IP Core

In the last three sections we have discussed time profiling, Vivado-HLS and HW timer. We now choose a program from ChStone benchmark to compare its SW, HW and HW-SW

performance in Virtex-5 series on ML507 board. This board has been chosen since,

- It has a hard core PowerPC processor which can run at maximum frequency of 400 MHz
- It supports partial reconfiguration which will be used in chapter five for comparison.

DfDiv program was selected since it consumed 50% of resources of the chip and profiling results showed that it takes around 0.8 ms to execute in SW, which is large value for a double precision floating point operation and its callgraph is shown in Fig. 3.19.

From the Fig. 3.11, it is concluded that the best candidates for HW migration is estimateDiv128To64 as it consume more time (>400 us) thus being computationally intensive as compared to other functions. This function was synthesized manually in Vivado-HLS and its IP core was generated for achieving the co-design process. The IP core generated from Vivado-HLS was interfaced in the EDK flow and its execution time was measured using HW-timer as shown in Table 3.15.

Table 3.15: ChStone benchmarks timing results on ML506 Board

Benchmark (1)	LUT(%) (2)	SW time Using Profiling (gprof) (3)	SW Time Using XPS timer (4)	HW Time (5)	HW-SW Time (6)
DfDiv	**50**	**0.800 ms**	**0.728 ms**	**0.0294 ms**	**0.406 ms**

The above process was done manually and following are the results as given in Table 3.8:

- *SW time using the gprof profiling = 800 µsec*
- *SW time using XPS timer time = 728 µsec*
- *HW time as DfDiv generated using Vivado-HLS and interfaced with the PLB bus = 29.4 µsec*
- *HW-SW co-design time using XPS timer = 406 µsec*

From these values we have proved that SW/ HW acceleration = 728/29.4 = 24.76 times. But the co-design flow shows only a 1.79 speed-up due to bus interface overhead.

3.8.1. Comparison with LegUp

LegUp tool also supports co-design process and computationally intensive functions can be accelerated by hardware, while the remainder of the program runs in software. This allows supporting a wider range of applications and enables a broad exploration of the hardware/software co-design space. With the MIPS soft processor, we can also execute the entire program in software. First we compile the benchmark with gcc to verify the output.

Hybrid flow is a combination of SW and HW. All the three flows hardware, hybrid and software were performed on the DfDiv benchmark program and it has been found that the execution time was least in pure hardware flow but hybrid flow is far better compared to pure software flow. The same DfDiv benchmark was compiled in LegUp and the result of the simulation is shown in Table 3.16 for the MIPS (150 MHz) processor. For hybrid case the estimate64div function was migrated for co-design flow.

Table 3.16: Performance comparison of DfDiv in LegUp

Category	Pure SW	Hybrid	Pure HW
Clocks	94188	9980	1928
Fmax (MHz)	71.88	65.19	100.123
Execution Time (μs)	1310.35	153.09	19.123

Table 3.16 shows that DfDiv program takes 1310 microseconds in SW and 19.123 microseconds in HW. These results are simulation based and do not consider the bus overhead incurred during the real time execution.

Table 3.17: Area delay product comparison of DfDiv in LegUp

Category	Pure SW	Hybrid	Pure HW
Area(Logic elements)	4735	10884	4384
Execution time(us)	1310.35	153.09	19.123
Area delay product	62040507	1666231	83835

Table 3.17 shows that the area delay product is maximum for pure SW and best for HW. The drawback of the HW results is that it only shows the area taken by the IP, but does not includes the system area, which includes controllers and processor. Since there is big difference in the clock frequencies in the two platforms (Xilinx and Altera) a significant comparison cannot be made. The LegUp area and delay are for Altera FPGAs, the comparison with ML507 which has PowerPC (400 MHZ) processor is not justified. The hybrid implementation lies in between the HW and SW implementation in Table 3.17.

This section described the manual process of co-design and is static in nature. In such a design flow a function selected by the designer is used in SW migration and design space exploration cannot be explored. The next chapter uses genetic algorithm for answering such a crucial question.

3.9. Conclusions

In this chapter we have shown how real time profiling can be used to find the time consumed by each function. The principle behind the generation of HW from a high level specification has been discussed. An algorithm has been proposed that can find the amount of resources consumed by the program by iterating through the control flow graph. A standard program (Dfdiv) selected from ChStone benchmark has been used testing and verification. For the same program its HW, SW and co-design time has been compared on Xilinx ML507 board. Thus it can be ssummarized as:

a) Vivado-HLS tool was able to compile most of the programs present in ChStone benchmark and is user friendly.

b) The Vivado-HLS generated core when interfaced in the EDK design flow gave 25 times better performance to SW.

c) The real time profiling gave accurate results, when the sampling time was order of microseconds.

REFERENCES

3.1 Raj Kamal, Embedded systems: architecture, programming and design, Tata McGraw-Hill Education, June, 2008. (Chapter-3).

3.2 Raghavan, Amol Lad, Sriram Neelakandan, Embedded Linux System Design and Development, Auerbach Publications, December, 2005.(chapter-2,3,4)

3.3 V. G. Lesau, E. Chen, W. A. Gruver, D. Sabaz, Embedded Linux for Concurrent Dynamic Partially Reconfigurable FPGA Systems, 2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 99-106, June, 2012.

3.4 Muhammad Ali Mazidi, The 8051 Microcontroller and Embedded Systems, Pearson Education India, 2007. (chapter-2,3)

3.5 EDK User Peripheral Tutorial - Xilinx, japan.xilinx.com/direct/ise7_tutorials/import_peripheral_tutorial.pdf

3.6 De Micheli & Ernst & Wolf, Reading in Hardware software co-design, Morgan Kaufmann, 2001.

3.7 Profiling tools, vast.uccs.edu/~tboult/CS330/NOTES/profilers.ppt, J. Fenlason and R.Stallman.

3.8 GNU gprof, https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

3.9 Susan L. Graham Peter B. Kessler Marshall K. McKusick, gprof: a Call Graph Execution Profiler, SIGPLAN '82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction, Volume 39, Issue 4, April, 2004.

3.10 https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_6.html.

3.11 El- Sayed M. Saad, Medhat H. A. Awadalla, Kareem Ezz El-Deen, FPGA-based software profiler for Hardware/Software co-design, National Radio Science Conference (NRSC), pp. 1-8, March, 2009.

3.12 M. Aldham, J. Anderson, S. Brown, A. Canis, Low-Cost Hardware Profiling of Run-Time and Energy in FPGA Embedded Processors, ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 61-68, September, 2011

3.13 Using the RDTSC Instruction for Performance Monitoring, https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf , 1997

3.14 C Library - <time.h> - TutorialsPoint, http://www.tutorialspoint.com/c_standard_library/time_h.html

3.15 EDK Profiling User Guide Xilinx, www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/edk_prof.pdf

3.16 Clang "clang" C Language Family Frontend for LLVM, clang.llvm.org/

3.17 LLVM IR, LLVM Language Reference Manual — LLVM 3.9 documentation, llvm.org/docs/LangRef.htm

3.18 Visualizing code structure in LLVM - Institute of Computational Science, https://www.ics.usi.ch/images/stories/ICS/slides/llvm-graphs.pdf

3.19 http://www.llvmpy.org

3.20 S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, San Francisco, CA, 1997.

3.21 Vivado Design Suite Tutorial: High-Level Synthesis (UG871) - Xilinx, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug871-vivado-high-level-synthesis-tutorial.pdf

3.22 Xilinx Vivado tutorial, ese.wustl.edu/~xuan.zhang/ese566_files/tutorials/vivado_tutorial.pdf

3.23 AR# 59444: Vivado HLS 2013.4 - Example to generate PLB IP

3.24 Processor Local Bus (PLB) 4.6 - Xilinx, www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf

3.25 XPS Timer/Counter, http://www.xilinx.com/products/intellectual-property/xps_timer.htm last accessed on May, 2015.

Chapter 4

Design and Development of Efficient Hardware and Software Partitioning Algorithm

This chapter describes a framework which aims at design space exploration using HW-SW co-design approach. The framework presents an automated flow to generate design parameters using partitioning algorithm. Data generated from Vivado-HLS and time profiling has been further used as an input to genetic algorithm for partitioning the application and generates various implementations. This chapter proposes various solutions to overcome the co-design challenges, making the design flow feasible.

4.1. Frameworks for Reconfigurable Computing Systems

For migrating any design to FPGA, the amalgamation of both SW-HW and HW-HW can be applied to a design to explore the design space. We have broadly classified the framework into two classes:

Framework-1: Efficient automatic conversion method for C to HW-SW-co-design:

A complete HW solution of an application gives best performance, but in many cases, it may consume a large amount of chip resources. We may then, look for implementing only a part of the application into HW. In such a case, the option is to use a combination of HW and SW. Such a design flow requires to identity the part of the application that takes more time in a systematic way rather than estimating the parameters manually. The extra effort required in this case is of interface design development for the application to work correctly. The interface can be simple for bare metal design (no OS) and complex for OS based design as it requires device driver development for. Many parts of such design flow are manually done and require broad knowledge of the application, interface and FPGA. This research work proposes an automated co-design approach way by using profiling, partitioning and high level synthesis process. This chapter extensively covers this framework in detail.

Framework-2: Multiple IP cores can be designed for an application and interfaced to the bus. Usage of such core has been referred as HW-HW design flow in this thesis. This research work explores this framework which is HW-HW implementation of an application and is researched as two different approaches. In approach 1, reusable patterns in application are identified, and interfaced as static IP cores to the bus. Such reusable cores will allow the total

consumed resources to diminish and improve overall metrics of the design. Such similar patterns are created as clusters using isomorphism concept.

The second approach aims at partitioning an application into clusters using genetic algorithm. These clusters are then dynamically scheduled on a given area using partial reconfiguration design flow. This framework is discussed in detail in chapter 5. The next section presents framework 1 and the profiling, high level synthesis, partitioning and results are put together in the subsequent sections.

4.2. Hardware Software Co-design Partitioning Design Flow

The proposed framework that implements co-design and guides the designer with well defined steps is presented in this section. An efficient partitioning technique being the sole objective in design flow requires the HW-SW time and the area of each function on a real HW, for identifying the best solutions. Since the time and area parameters are not directly available from a single tool-chain, certain assistance from different tools is required. Here Vivado-HLS, gprof and time profiler were used to accumulate these values.

The framework starts from the specification written in C language. The code is synthesized using Vivado-HLS and resources are tabulated for each function in the program. For timing estimation, a callgraph is generated using gprof, which is a time profiler utility in gcc compiler and it returns the time taken by each function. The generated data from Vivado-HLS and gprof are given as input to the partitioning stage.

The partitioning step generates the various implementations of an application for a different deadline. The objective of this phase is to guide the designer about which functions should be in HW and which once should be in SW. When the C program is populated with too many functions and the value of time/area is close, answering this question is very tricky. Unless an initial implementation on real HW is available, this cannot be answered easily. An initial implementation which gives performance parameters requires extensive efforts in system-on-chip design flow. If this initial implementation can be bypassed, the job of the designer can be simplified.

The HW is then interfaced with the bus and bus wrapper is created. The SW part is compiled with Xilinx SDK and finally the entire design is tested for performance gain using HW timer. We propose a design flow as shown in Fig. 3.7, which is an extension of design flow discussed in [2.1] and aims at answering the issues discussed above.

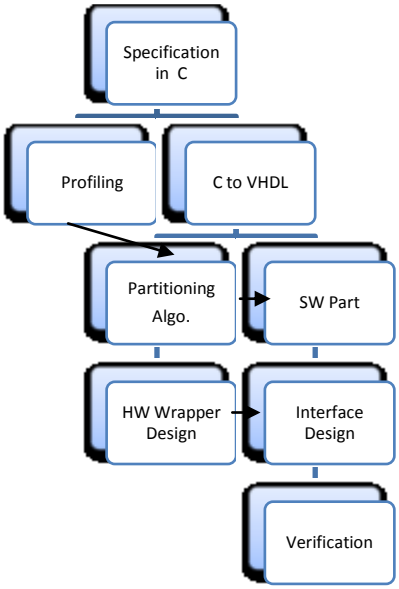


Figure 4.1: HW and SW Co-design flow

The flowchart in Fig. 4.1 describes the steps followed in detail:

- We start from a program written in C subset. Recursion and dynamic data structures are not supported in Vivado-HLS, hence should be avoided. LegUp group [2.5] has developed a standard benchmark known as ChStone which is written in C language and has been used in this work.
- The C code is profiled in gprof to generate callgraph and tabulate the call values and time consumed by each function. The call graph and the flat profile format are extracted for finding the number of calls. The child and parent relation is generated using pvtrace utility [2.18].
- Vivado-HLS is used to generate Verilog code of the benchmark. It can bundle the entire C code into an IP core that can be directly interfaced with the bus. However, it does not allow generating Verilog code of selected functions automatically. Though, we can choose the individual functions and migrate them to HW manually. This option is also possible in LegUp by defining a function in a Tcl script manually. This step identifies the resource taken by each C function and latency in terms of number of cycles consumed.
- The proposed partitioning algorithm is now applied to find the best solution. The functions, their dependency, resource table from Vivado and time of each function are given as input to the algorithm. The algorithm then generates the best HW-SW implementation for a given deadline.
- The previous stage identifies the functions that are supposed to be migrated to HW. These functions are taken from Vivado and a wrapper is created for interface design. The

required FIFO or slave registers are used for input/output data transfer as discussed in section 3.1.

- An upper level SW is now written in SDK, which can bind the functions together and the entire design is now built to test the performance gain and verify the functionality. Pure SW execution of the program on SDK gives the lower bound and pure HW as core interfaced to the bus gives the upper bound for comparison of HW, SW and hybrid partitioning approach. The above two steps have been discussed in previous chapter.

4.3. Partitioning Process Using Genetic Algorithm for HW-SW Co-design

In the previous section we have proposed the design flow and presented the manual approach of interfacing a predetermined function in a SOC flow. We have shown how time and area can be estimated using profiling and synthesis in Vivado-HLS for a program written in C. If any other function other than estimate64div is migrated to HW, what will be the performance can only be estimated after following the SOC flow. Such a flow is time consuming and error prone. What we need is a algorithm that will define which functions should be in HW and which once should be in SW, so that a given deadline is met. Hence we explore the process of partitioning and apply it to the domain of co-design. In subsequent sections we will show the process of partitioning on two different kinds of specification:

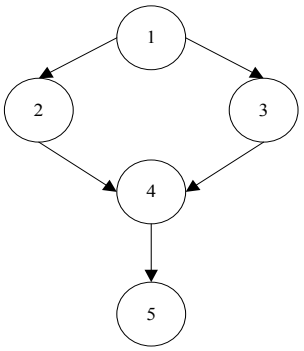
a) A C program: Partitioning at this level means deciding which functions should be migrated to HW. Since there can be data structures shared across these functions, they cannot be assumed to be independent of each other. Hence in this case serial execution of functions will be considered in proposing the time equation. For this we will assume one CPU and one HW area available for mapping.

b) A task graph: It is an acyclic graph in which data flows from one node to another and the assignment is continuous. Since there in no data structure sharing and continuous assignment, it is possible to run the operations in parallel, hence a different time equation will be proposed. Here we will extent the architecture to have multiple CPU and multiple HW area available for mapping.

Partitioning is a process by which we divide the input specification into disjoint subsets depending on the constraints which defines the number of subsets, maximum vertices running between the subsets and the number of functions or nodes in a subset. It has been extensively used in Electronics design automation in placement and routing algorithms for netlist processing. It has extended its span to domain of embedded systems where HW-SW co-design exploration can be done. Most of the applications can be described as dataflow and control

flow graphs which are inputs to the partitioning phase hence the roots of partitioning phase are in graph theory. We next introduce basic terms and jargons used in graph theory.

A graph G = (V, E) portrays nodes V as operators and edges E as connections. The graph size parameters: are defined as n (nodes) = |V|, m (edges) = |E|. Fig. 4.2 (a) shows the sample graph with its parameters in Fig. 4.2 (b).



V = {1,2,3,4,5}

E={{1,2},{1,3},{2,4},{3,4},{4,5}}

N=5

M=5

Fig. 4.2: (a) Sample graph

Fig. 4.2: (b) Parameters for sample graph

The graphs can be classified as undirected, directed, mixed and weighted. An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct. An undirected graph is a tree if it is connected and does not contains a cycle. Forest is s a union of trees. Given a tree (T), choose a root node (r) and orient each edge away from r defines a rooted tree. A shortest path between two vertices is a path of minimal length. Given below are parameters and traversals [4.1] that are utilized in algorithms applied to graph.

a. **Length** – number of edges in the graph.

b. **Distance** between u and v – the length of a shortest path between them (or ∞ if a path does not exist)

c. **Subgraphs**: *G'(V', E')* is subgraph of *G(V, E): V' ⊆ V, E' ⊆ E*

d. **Degree** of a vertex: the number of edges incident on the vertex (in undirected graphs)

e. **In-degree** and **out-degree** in directed graphs: the number of edges coming into/going out of the vertex.

f. **Adjacency matrix**: n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge. Fig. 4.3 (a) shows a sample graph with its adjacency matrix in Fig. 4.3 (b).

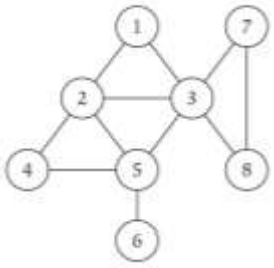


Figure 4.3: (a) Sample graph

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	1	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Figure 4.3: (b) Adjacency matrix

g. **Adjacency list**: Node indexed array of lists.

h. **Graph traversing and shortest path problems:** Given a graph G (V, E), explore every vertex and every edge using the adjacency using depth first search or breadth first search approach.

In the next section we present some vernaculars used in HW-SW partitioning that are used in the algorithm selection and design.

4.3.1. Hardware and Software Partitioning Issues

HW-SW partitioning is the process of mapping of application defined as graph nodes to either hardware or software components. The objective here is to minimize the execution time, while keeping the physical area within constraints. Before the designer can apply partitioning, the various issues have to be kept in consideration. These issues are briefly described below [2.19]:

I. **Abstraction Level:** The partitioning process can consider the input at various levels of abstraction. As the level increases the amount of logic amounting in the node also increases. The HW abstraction level can be:

a. **Netlist or DFG:** This is the lowest level and nodes are digital components with their interconnection.

b. **FSMD:** These are used for behavioural specification of the systems along with control structure.

c. **Modules:** These are entities that define functionality and can be treated as black boxes.

d. **Subsystems:** A complete system is a collection of sub-systems.

The SW abstraction level can be:

a. **CFG:** Control flow graph is used to describe the flow structure of any language and can be used as graphical notations for specification.

b. **Basic block:** Most compilers generate CFG at the basic block level. A basic block level is a piece of code which is entered at the top and exited at the bottom.

c. **Functions:** These are entities that define functionality and can be called as per requirement.

d. **Subsystems**: A complete SW is a collection of functions put together.

The partitioning process can be applied to any of these abstraction levels. In this chapter we are showing the process on function level and DFG level, although the proposed algorithm can be applied to any level.

II. **Objective Function:** The partitioning process is guided by a function which tells whether the algorithm movement is beneficial or not and it is known as objective function. This function can be modified as per the requirement and constraints in the design. For e.g. very simple objective function can be cost, which means for a given solution the one with minimum area cost will be chosen.

4.4. Genetic Algorithm for Co-design

Partitioning has been a matter of extensive research in the ASIC synthesis tools and aim at creating clusters, such that minimum wires are running between them. It can be divided into two categories: constructive and iterative based approach. Constructive partitioning aims at identifying an initial partition and iterative perform a certain number of iterations. Examples of constructive algorithm are cluster growth, hierarchical clustering, etc. Iterative algorithms are heuristic in nature that find solutions among all possible ones, but they do not guarantee that the best will be found, therefore they may be considered as approximately good but not optimal algorithms. Examples of iterative algorithm are Fiduccia Mattheyses (FM), simulated annealing (SA) and genetic algorithm (GA). The application of GA has been shown efficiently in VLSI domain [4.2-4.3] and further extended to co-design [4.4]. The current work uses GA [4.5] as the underlying partitioning algorithms since the literature survey shows the efficacy of the algorithm. Another approach known as Integer linear programming is a mathematical

approach that has been used for partitioning and scheduling solutions for smaller set of problems.

Genetic algorithm is a robust stochastic optimization technique that is inspired by the principle of survival of the fittest in nature [2.27]. GA gives good result because at a time it maintains a set of solutions, whereas algorithm like simulated annealing works with one solution. An initial population is randomly created at the starting phase of GA. This population is actually a set of solution for the problem under consideration. The fitness value of the objective function [4.6] for each individual in the set is the index of the goodness of that individual. GA evolves the population over generations with the use of operators such as selection, crossover, and mutation. In selection, individual with higher fitness value are selected and repeated in the next population. The newly selected population is subjected to a crossover operation where genetic information of the individuals, with better fitness value is exchanged. This enriches the population with better offspring. The final step is mutation where some bit of genetic information is complemented in the selected individuals. Finally the fitness value is compared and a new iteration is started until there is no change in the fitness value of the new generation. The pseudo code for GA is shown in algorithm 4.1 with explanation given in each line as comments.

Algorithm 4.1: Pseudo Code for Time and Cost Calculation

```
/* Create first generation with gen_size random partitions*/
G = NULL
for I in 1 to gen_size loop          //select a generation size and for each generation do
G = G U CreateRandomPart(O)          // create a random solution and add it to G
end loop
P_best = BestPart(G);                // Assign a solution to a variable.
/*Evolve generation */
While not Terminate loop // do some fixed number of iterations
G = Select(G, num_sel) U Cross(G, num_cross) // Apply selection and crossover
Mutate(G,num_mutate)                         // Apply mutation
If objfct(BestPart(G)) < Objfct(P_best ) then
 //find the value of objective function, reject it if more
P_best = BestPart(G)
end if
end loop
return P_best                                // return the solution
```

4.4.1. Sample Case Study using GA for Co-design Using Callgraph Model

We now present the partitioning phase of framework 1 using genetic algorithm. In this example, we extent the applications of partitioning to callgraph model generated from a C program. In order to prove the effectiveness of the partitioning process, a benchmark is required, which tabulates the SW execution time, SW area, HW execution time and HW area on a given CPU and FPGA. The availability of such benchmarks has not been reported, hence we start from a random specification and values as shown in Table 4.1. The C application consists of callgraph as shown in Fig. 4.4 and has seven functions. This was created for examining the proposed design flow and values were designed in such a way that algorithm has to search for a good solution. The main function is always implemented in SW because it is the function which calls all other functions. Only one level hierarchy has been taken since all the sub-callee will be the component of the callers. This assumption has been made since the HLS HW generation is usually done of the functions present at level 1. This means for Fig. 4.4, there will be six SW functions in SW or six modules as HW in HLS.

Since the C program executes serially, in the total time calculation, serial execution of functions has been considered. The function main has been given 0 time so that the algorithm takes it as SW. In a situation in which the functions have similar parameters we need a well defined partitioning algorithm that can give concrete answers about the co-design paradigm.

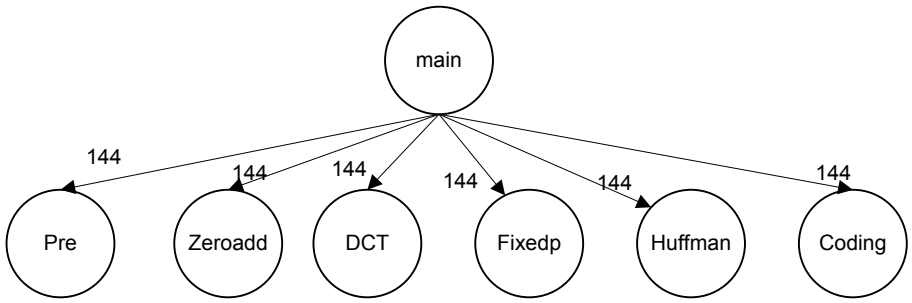


Figure 4.4: Callgraph of a random C program

Table 4.1 shows the dummy values taken for seven functions. From previous discussions, we can get the values from the Vivado-HLS, but this feature has to be inbuilt in compiler of Vivado, i.e. the tool itself should return the values in a tabulated form.

Table 4.1: SW and HW parameters

Node-Name(1)	CPU Time(2)	FPGA Time(3)	Calls(4)	FPGA Resources(5)	CPU Area(6)
Main	0	100	1	0	5000
Pre	90	1	144	10000	0
Zeroadd	20	2	144	2000	0
Fixedp	40	4	144	4000	0
DCT	100	10	144	5000	0
Huffman	50	5	144	6000	0
Coding	70	7	144	10000	0

Table 4.1 has following columns:

- Column 1: The name of the function.
- Column 2: The time taken by each function in SW. Main has been given zero value so that it is always admitted in SW by algorithm.
- Column 3: The time taken by each function in HW. These values are kept 10% faster as compared to SW values as we have shown in DfDiv example (section 3.8) that HW is 25 times faster to SW.
- Column 4: Number of times the main calls the other functions.
- Column 5: Area consumed by each functions on FPGA. For e.g. FPGA may have 44800 LUTs.
- Column 6: Area consumed by each functions in CPU. The main's function area is fixed to 5000 accounting to area taken by CPU on the chip and all other are kept as zero since functions will be stored in memory and will not consume area.

As shown in the algorithm 3.1 that the GA is guided by the objective function which defines the goodness of the solution. We have used the objective function (*objfct)* as shown in Eq. 4.1. It includes time and implementation cost along with a predetermined deadline [2.33].

$$Objfct = \begin{cases} k_1 * (t - t_r) + k_2 * c & if\ t > t_r \\ c & if\ t < t_r \end{cases} \quad \ldots(4.1)$$

In Eq. 4.1, t is total execution time of solution, c is total cost (area), t_r is deadline constraint and k_1, k_2 are constants. The objective is to find the minimum value of cost for a given time constraint. So, satisfying solutions are obtained under the condition $k_1 >> k_2$ because objective function value becomes large under the condition when execution time exceeds deadline. This will guide the GA and it will remove this solution in next iteration. For e.g. suppose, if k_1 =1000, k_2=1, t_r = 200, c = 500 then if t > t_r *objfct* = 1000(300 - 200) + 1*500 = 100500. But if

$t < t_r$, *objfct* = 500. Hence if the deadline is missed *objfct* takes a high value because of k_1 and k_2 and this solution will be avoided.

Similar Objective Function can be used when area is the constraint and it is shown in Eq. 4.2.

$$objfct = \begin{cases} k_1 * (a - a_r) + k_2 * t & if\ a > a_r \\ t & if\ a < a_r \end{cases} \quad \ldots(4.2)$$

$$F = \sum_{i=1}^{p} \text{ObjfctF}(V_i) \quad \ldots(4.3)$$

$$p_i = objfct\ (V_i)/F \quad \ldots(4.4)$$

We have used Roulette wheel [2.27-Appendix-2] implementation in GA and it requires that the objective function should be normalized and converted to numbers between 0-1. For this the Fitness of the population is calculated and is given by Eq. 4.3. For converting the number between 0-1, Eq. 4.4 is used. The detail GA algorithm is shown below with comments added in each line.

Algorithm 4.2: GA FOR HW SW Partitioning

BEGIN

p = population size; // decide the size of the initial population

n = no. of nodes; // number of nodes in the graph

G = Generate random population (0-1) of size = p x n; // for roulette wheel generate random no.

REPEAT

1. *FITNESS evaluation of each chromosome G*

   Calculate time;// find the time of each chromosome

   Calculate cost; //use HW-SW time and area to find the cost.

   Calculate Objfct; // use Eq. 4.1 to find the objective function

2. *PROBABILITY evaluation of each chromosome*

   Give max probability to min. OF;// using Eq. 4.4 find the probability of each chromosome.

3. *CUMULATIVE PROBABILTY (CP); //convert the above numbers to cumulative distribution.*

4. *SELECTION (Roulette Wheel) of better chromosomes*

   g1= generate random no (0 to 1) for size= p;

   Find nearest larger CP value for each g1;

   G = Generate new population;

   (max. chance of larger probability value)

5. *CROSSOVER between the pairs of parents*

   c = define crossover probability;

   g2= generate random no (0 to 1) for size= p;

   Select chromosome for g2 ≤ c;

   Generate random position (0 to n) for each p pair of selected chromosome;

   Interchange bit patterns between pairs after their generated position;

   G = Generate new population;

6. *MUTATION of the resulting offspring;*

   m = define mutation probability;

   g3= generate random no (0 to 1), size= p x n;

   Select position for g3 ≤ m;

   Toggle bit pattern;

   G = Generate new population;

   UNTIL TERMINATION CONDITION SATISFIED

Schedule optimum OF value pattern;

END

The algorithm proposed for cost calculation is shown in algorithm 3.3 with NC being the number of time the function is executed. The algorithm finds the total time taken for the chromosome.

Algorithm 4.3: Pseudo Code for Time and Cost Calculation

```
BEGIN
Take one chromosome
Cost = 0; dummy = 0;
for i=1:n
   If (pattern [I] == 1)
    execution_time[i] = hw_time[i] *NC[i]; //multiply the number of calls with the hw time.
         Cost = Cost + hw_cost[i];
   else
    execution_time[i] = sw_time[i] *NC[i];
    if (dummy == 0)
        {Cost = Cost + sw_cost[i];
              dummy = 1;}
Start_time[1] = execution_time[0];
End_time[1] = 0;
for i=2:n
```

End_time[i] = Start_time[i] + execution_time[i];// addition of total time of each node
end
Time = End_time[n];
Cost;
END

Following functions were written in Matlab to accomplish the GA execution, *random_population(), mutatiton(), crossover(), fitness_value(), tot_time(), tot_cost(), roulette_wheel(), pattern(), randpop(), cum_probability_value() and schedule().*

4.4.2. Experimental Results

The architecture used for verification is shown in Fig. 4.5 with one CPU and one FPGA. The values of each node are shown in Table 4.1 for CPU/FPGA time and CPU/FPGA area.

Figure 4.5: A SOC testing architecture

The process by which the values are given as input and the manner in which the outputs are obtained from one run of GA is shown below:

1. *For e.g. consider the deadline as 200:*
2. *enter deadline: 200 ; User defined time for application to complete*
3. *enter population size: 20 ; The size of initial population for GA*
4. *enter no of iteration: 50 ; The number of times the GA does the iterations*
5. *Do you want serial execution or parallel (s/p): s ; The execution is serial or parallel*
6. *PATTERN = 0 1 1 1 1 1 1 ; main in SW and all in HW*
7. *COST = 4018000 TIME = 4176*
8. *node 1 start = 0 end = 0*
9. *node 2 start = 0 end = 144*
10. *node 3 start = 144 end = 432*
11. *node 4 start = 432 end = 1008*
12. *node 5 start = 1008 end = 2448*
13. *node 6 start = 2448 end = 3168*
14. *node 7 start = 3168 end = 4176*

A 0 in the pattern means SW node and 1 means HW node shown at line number 6. A manual analysis gives that the pure HW time is 4176 and pure SW time is 53280. A hybrid implementation should fall in between these values.

Table 4.2: Partitioning results for different deadlines

Deadline	Opt. Cost	Pattern	Time	Area	Product
00200	4018000	0111111	04176	37000	154512000
01000	3218000	0111111	04176	37000	154512000
05000	0042000	0111111	04176	37000	154512000
10000	0038000	0110111	09360	33000	309870000
20000	0028000	0101110	18432	25000	460800000
30000	0020000	0100100	27504	15000	412456000
40000	0020000	0000101	31248	15000	468720000
50000	0009000	0001100	49536	09000	445824000
53280	0005000	0000000	53280	05000	266400000

The results for different deadlines are given in Table 4.2

- Column 1: The user defined deadline
- Column 2: Cost value returned by the GA
- Column 3: SW and HW pattern for the chromosome selected.
- Column 4: The time taken by the chromosome to complete.
- Column 5: Area consumed by the chromosome.
- Column 6: Area delay product for the chromosome.

For lower value of deadline upto 5000, all the functions are implemented in SW except the main. This is because the lowest possible time is 4176 with HW. As the deadline increases, the functions migrate to the SW and the cost comes down. For e.g., If the deadline is t = 200, k_1 = 1000 and k_2 =1, then from Eq. 4.1, objfct is given as:

objfct = 1000(4176-200) + (10000 + 2000 + 4000 + 5000 + 6000 + 10000) = 4018000;

Suppose the deadline given is 10000, then 0110111 chromosomes can be taken as the solution. Hence the choice of the solution is dependent on the designer. Many parameters like number of iterations, the size of the initial population, mutation and crossover probability, size of the input can impact the results and running time of GA. More iteration gives a better optimization results, but computations increases. Higher probability value gives faster convergence, but the possibility of missing out optimum value is possible. So, for a particular problem these parameters experimentally determined and are fixed. Table 4.3 represents different parameter values have been used in GA algorithm.

These values have been taken from [2.34], but population size and iterations are defined for our problem set. The constant terms allows to create a weight factor in the function, crossover probability and mutation probability are kept low.

Table 4.3: Parameter values used in GA

Parameter	Value
k_2 (constant in OF)	1
k_1 (constant in OF)	1000
c (Crossover probability)	0.5
m (Mutation probability)	0.02
p (Population size)	20
Iterations	50

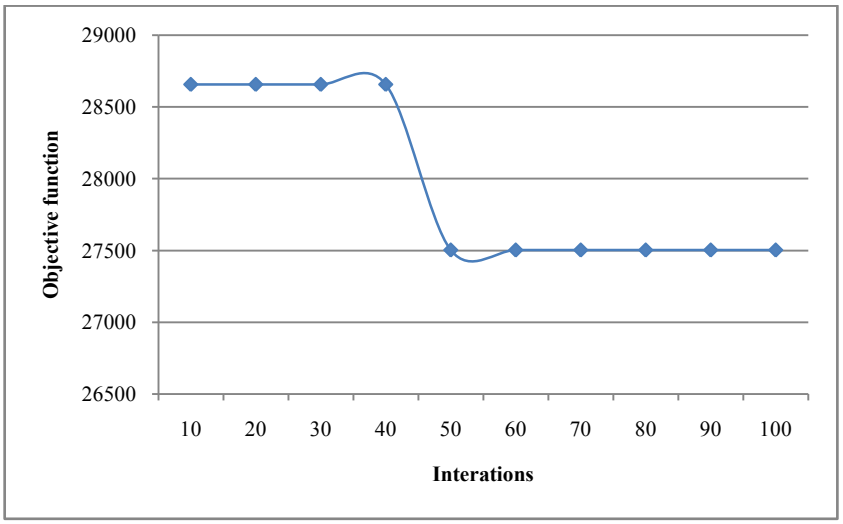


Figure 4.6: Fitness value optimized with iterations in GA for implementation

Fig. 4.6 shows how fitness value moves towards the optimum solution with increasing number of iterations in GA for CPU – ASIC implementation for deadline = 30000. Here for particular results, no. of iterations is set to 50. Once the algorithm finds a minimum cost solution, it will hold this solution till it finds a solution optimum to the current one. After certain no. of iterations, the fitness value reaches to its optimum value and GA will hold this value in next remaining iterations. Given graph is indicative only, for every run of algorithm curve will change, but the final optimization result will remain same after reasonable no. of iterations. This happens because every time GA starts with random populations, so in every run different initial starting and with a selection, crossover and mutation stages as algorithm achieves solutions from random numbers only.

After applying the GA to case study, we now take the same program Dfdiv which we chose in section 3.8, to show the HW and SW solutions generated using GA. In order to apply the GA

what we need is the callgraph, HW time, SW time and HW area of each function. The call graph generated is shown in Fig. 4.7 for DfDiv application using pvtrace utility in Linux.

The exact same functions shown in a C program callgraph are not synthesized as modules in Vivado-HLS. The sub-functions are synthesized as sub-modules. There are total 15 functions in the Dfdiv application. Floatdiv64_div is the function which calls all other functions and main has the testing data for verification. Now Vivado-HLS returned four modules(Float64_div, EstimateDiv128to64, propagateFloat64NaN and roundAndPackFloat64) as shown by circles in Fig. 4.7. Table 4.4 shows the resources returned by Vivado-HLS of these modules.

The time for each function can be accumulated from the profiling results. The FPGA time can be tabulated from the Vivado synthesis results, number of calls from the pvtrace utility (this needs modification similar to Vivado synthesized modules), FPGA resources as LUT slices from Vivado-HLS and CPU area from data sheet of Virtex-5. These values are shown in Table 4.4.

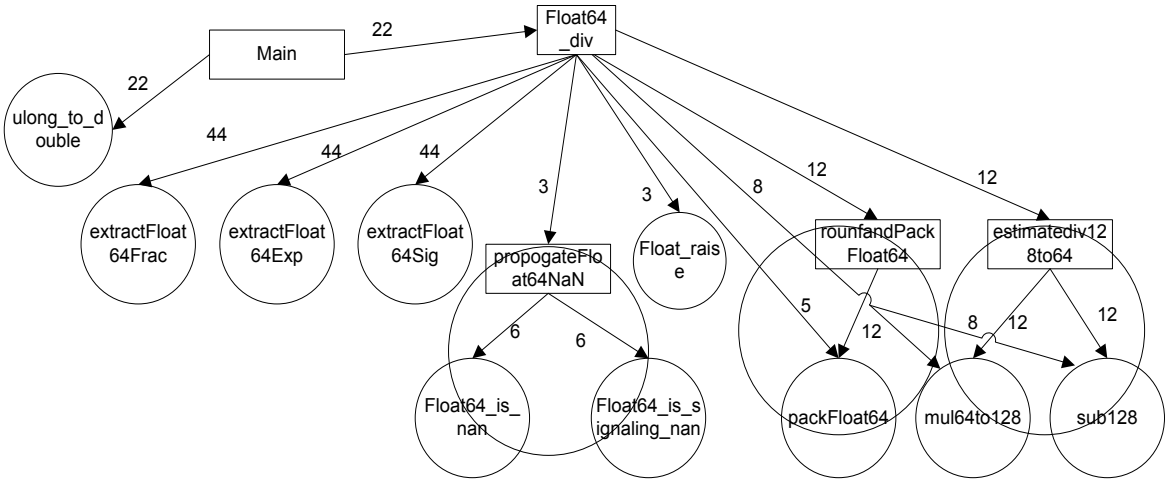


Figure 4.7: Callgraph for DfDiv

Table 4.4: Resource usage from Vivado HLS for DfDiv program

Function	BRAM_18K	DSP48E	FF	LUT
Available on Virtex-5	296	128	44800	44800
Float64_div	1	16	2812	4184
EstimateDiv128to64	**0**	**8**	**16988**	**16928**
propagateFloat64NaN	0	0	0	329
roundAndPackFloat64	0	0	367	1385
Total Used	1	24	20167	22826

For Float64_div the CPU time is taken to be less so that this function is always in SW, this is due to the fact that this function calls all other functions. We have proved that HW time is at least 25 better to SW time in Dfdiv example hence the SW time has been divided by 25 times in the Table 4.5.

Table 4.5: Parameters for DfDiv

Node Name (1)	CPU (Total time) Time (2) (usec) (profiling)	FPGA Time (3) (usec)	Calls (4)	FPGA Resources (5)	CPU Area (6)
Float64_div	Base as 210/0	0	1	4184	5000
EstimateDiv128to64	10.25	0.41	40	16928	0
propagateFloat64NaN	5	0.2	12	329	0
roundAndPackFloat64	40	1.6	3	1385	0

The execution of GA for generating the design space is shown as a flowchart in Fig.4.8.

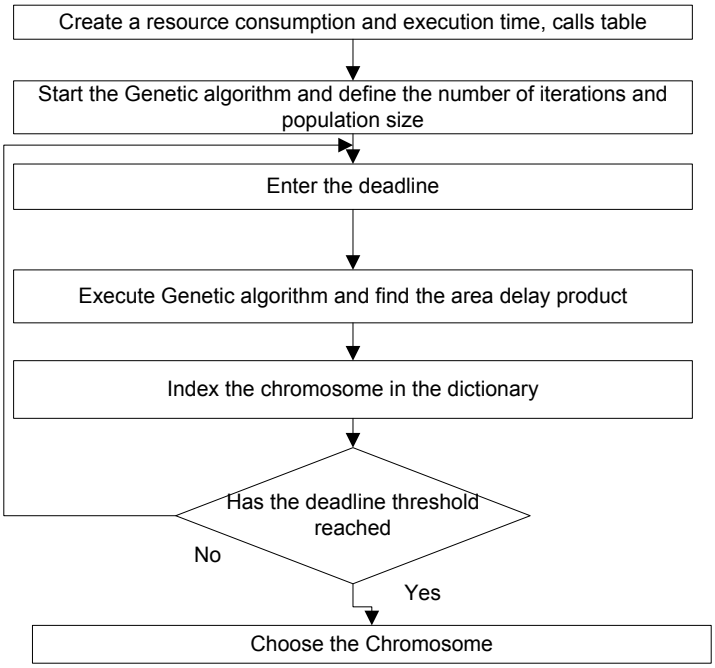


Figure 4.8: Flowchart for the algorithm execution

The execution steps of GA are shown in Fig. 4.8. After the inital parameters are set , different deadlines are entered until a upper threshold of deadline is reached. Table 4.6 shows the area and delay product for various solutions generated by GA for various deadlines.

Table 4.6: Results of DfDiv program

Deadline	Pattern	Time	Area	Product (Time x Area)
300	0101	291.2	23313	06788745.6
350	0110	348.8	22257	07763241.6
400	0100	406.4	21928	08911539.2
430	0111	233.6	23642	10251171.2
650	0011	627.2	06714	04211020.8
700	0001	648.8	06385	04142588.0
800	0000	800.0	05000	00400000.0
800	0010	794.4	05329	04233357.6

Designer can select the chromosome according to deadline. These values are plotted as a graph in Fig. 4.9. For e.g. if the deadline is 430, all the functions are in HW except float64_div.

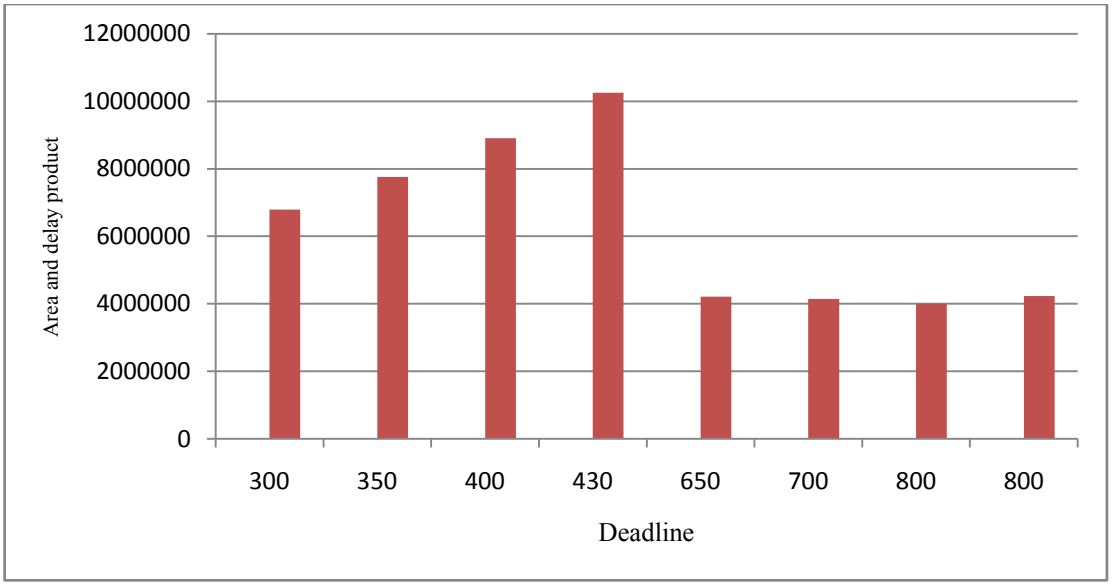


Figure 4.9: Deadline vs. Area/delay product for various genes

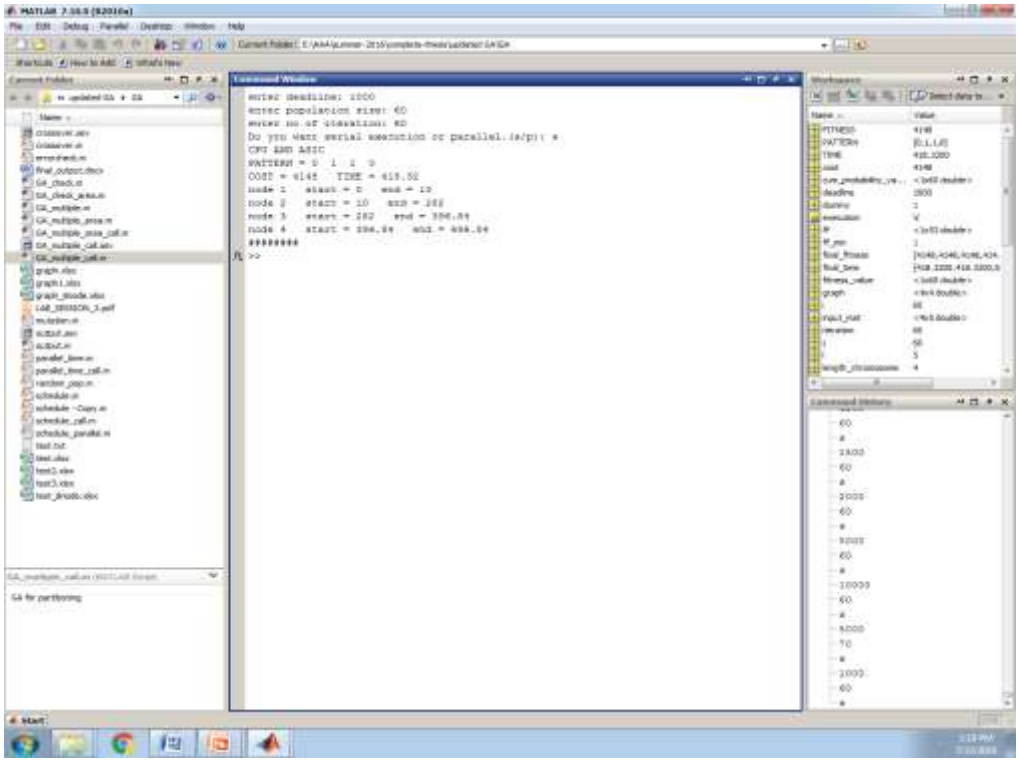


Figure 4.10: Snapshot of GA running in Matlab

Fig. 4.10 shows the snapshot of the GA program execution in Matlab.

4.5. Sample Case Study using GA for Co-design Using Task Graph Model

In the previous example call graph was taken as input to the genetic algorithm. In this section, a input specification and a architecture have been created for proving the effectiveness of the algorithm for acyclic graph specification. Task graph [4.7] with eight nodes has been presumed for demonstrating the GA partitioning approach as shown in Fig. 4.11. Task graph is an acyclic graph which represents an application as nodes and edges. There is no data

structure sharing among the nodes, i.e. there is continuous assignment of the output of one node to another. This means that the nodes can be executed in parallel, which was not the case with callgraphs.

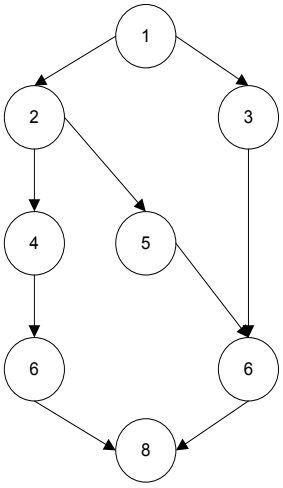


Figure 4.11: Sample task graph

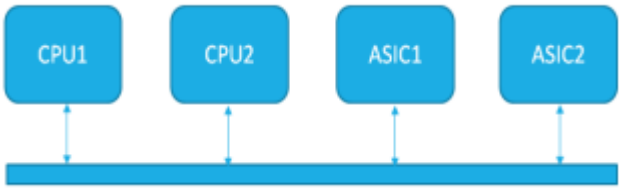


Figure 4.12: Multiple CPU and ASIC sample testing architecture

Fig. 4.12 shows the architecture assumed for implementing the task graph and contains two CPU and two accelerator area named as ASIC1 and ASIC2. This architecture is different from the call graph presumed architecture; hence this sample case study using GA is different from call graph approach. Table 4.7 represents time and cost for 8 tasks on 2 CPU and 2 ASIC. The various possible architectures can be CPU1, CPU2, CPU1-CPU2, CPU1-ASIC1, CPU1-ASIC2, CPU2-ASIC1, CPU2-ASIC2. We have imposed the restriction that at least one CPU and one ASIC should be present in the solution. This can be changed as per the requirements. Now the task of GA is to generate the best performance solution on any one of them: CPU1-ASIC1, CPU1-ASIC2, CPU2-ASIC1, CPU2-ASIC2 and show the node mapping and schedule on the component.

Table 4.7: Implementation parameters for different Tasks

Node	CPU1 Time	CPU2 Time	ASIC1 Time	ASIC2 Time	CPU1 Cost	CPU2 Cost	ASIC1 Cost	ASIC2 Cost
1	60	30	20	10	40	60	25	45
2	90	50	30	15	40	60	30	35
3	81	54	27	15	40	60	15	20
4	60	40	20	10	40	60	10	15
5	90	44	30	15	40	60	10	10
6	87	30	27	20	40	60	10	25
7	90	50	40	15	40	60	15	35
8	99	56	33	20	40	60	15	15

The GA will be using the execution to determine the objfct of a chromosome. Algorithm 4.4 shows the pseudo code used in finding the execution time of each task.

Algorithm 4.4: Pseudo Code of Time and Cost Calculation

```
BEGIN
Take one chromosome
Cost = 0; dummy = 0;
for i=1:n                          // for each node
   if (pattern[i] == 1)            // if the pattern is 1 then the node is in HW
        execution_time[i] = hw_time[i];
         Cost = Cost + hw_cost[i];
   else                            // else it in SW
        execution_time[i] = sw_time[i];
        if (dummy == 0)
            {Cost = Cost + sw_cost[i]; // add area
              dummy = 1;}
Start_time[1] = 0;
End_time[1] = execution_time[1];
for i=2:n                          // schedule the task
   Start1 = max{End_time[predecessor_task]}; // calculate the predecessor which has max time
   for j=1:i-1
        if(pattern[i] == pattern[j])        // if both are in on same component
           z=j;                             // use a temporary value to store
   end
   Start2 = End_time[z];                    // find the end time of z
   Start_time[i] = max(Start1 + Start2)     // add thhe
```

End_time[i] = Start_time[i] + execution_time[i]; // calculate the end time of node i

end

Time = End_time[n];

Cost;

END

Algorithm 4.4 shows pseudo code for parallel execution time and total cost calculation of one chromosome which is useful for finding Fitness Value of chromosome. In a particular algorithm, each chromosome is represented by a pattern of 0 and 1 with length equals to total no. of tasks (nodes) in task graph. Here task is represented by 0 if it is implemented in software and 1 if it is implemented in hardware. In total cost calculation all ASIC implementation costs are added, but only one time CPU cost is added because the CPU area remains same. Here communication cost between tasks while migrating from hardware to software or from software to hardware is not taken into consideration.

4.5.1 Results of Sample Case Study

For particular input values of the task graph various iterations have been made. For the given parameters and deadline= 275 time units optimum hardware-software partition for each possible combination is shown in Table 4.8 along with scheduling of each task in particular partition. For deadline = 275 optimum cost value is 80 and it is found in CPU2 and ASIC1 combination. Now for same task graph parameters if deadline = 200 optimum cost value of 115 is obtained in CPU2 and ASIC1 combination. The cost results and scheduling of tasks are shown in Table 4.9. For correctness of solutions manual crosscheck has been made with all possible combinations of partition (all possibilities) and it is found that given solutions are optimum, hence given algorithm gives optimal solution.

Table 4.8: Optimization results for deadline = 275

	CPU1 & ASIC1			**CPU1 & ASIC2**			**CPU2 & ASIC1**			**CPU2 & ASIC2**		
	Opt. Cost		Time	Opt. Cost		Time	**Opt. Cost**		**Time**	Opt. Cost		Time
	115		264	135		267	**80**		**270**	85		270
Node	**Map**	**Start**	**End**	**Map**	**Start**	**End**	**Map**	**Start**	**End**	**Map**	**Start**	**End**
1	CPU1	0	60	CPU1	0	60	CPU2	0	30	CPU2	0	30
2	ASIC1	60	90	CPU1	60	150	CPU2	30	80	CPU2	30	80
3	CPU1	60	141	ASIC2	60	75	CPU2	80	134	CPU2	80	134
4	ASIC1	90	110	ASIC2	150	160	ASIC1	80	100	ASIC2	80	90
5	ASIC1	110	140	ASIC2	160	175	ASIC1	100	130	ASIC2	90	105
6	ASIC1	140	167	CPU1	160	247	CPU2	134	164	CPU2	134	164
7	CPU1	141	231	ASIC2	175	190	CPU2	164	214	CPU2	164	214
8	ASIC1	231	264	ASIC2	247	267	CPU2	214	270	CPU2	214	270

Table 4.9: Optimization results for deadline = 200

	CPU1 & ASIC1			**CPU1 & ASIC2**			**CPU2 & ASIC1**			**CPU2 & ASIC2**		
	Opt. Cost		Time	Opt. Cost		Time	**Opt. Cost**		**Time**	Opt. Cost		Time
	155		200	175		176	**115**		**197**	120		190
Node	**Map**	**Start**	**End**	**Map**	**Start**	**End**	**Map**	**Start**	**End**	**Map**	**Start**	**End**
1	ASIC1	0	20	CPU1	0	60	CPU2	0	30	CPU2	0	30
2	ASIC1	20	50	ASIC2	60	75	CPU2	30	80	CPU2	30	80
3	CPU1	20	101	CPU1	60	141	ASIC1	30	57	ASIC2	30	45
4	ASIC1	50	70	ASIC2	75	85	ASIC1	80	100	ASIC2	80	90
5	ASIC1	70	100	ASIC2	85	100	CPU2	80	124	ASIC2	90	105
6	ASIC1	100	127	ASIC2	100	120	CPU2	124	154	CPU2	90	120
7	ASIC1	127	167	ASIC2	141	156	ASIC1	124	164	CPU2	120	170
8	ASIC1	167	200	ASIC2	156	176	ASIC1	164	197	ASIC2	170	190

Fig. 4.13 shows how fitness value moves towards the optimum solution with increasing number of iterations in GA for CPU2 – ASIC1 implementation for deadline = 275. Once the algorithm finds minimum cost solution, it will hold this solution till it finds a solution optimum to the current one. After certain no. of iterations fitness value reaches to its optimum value with deadline constraint algorithm will hold this value in next remaining iterations.

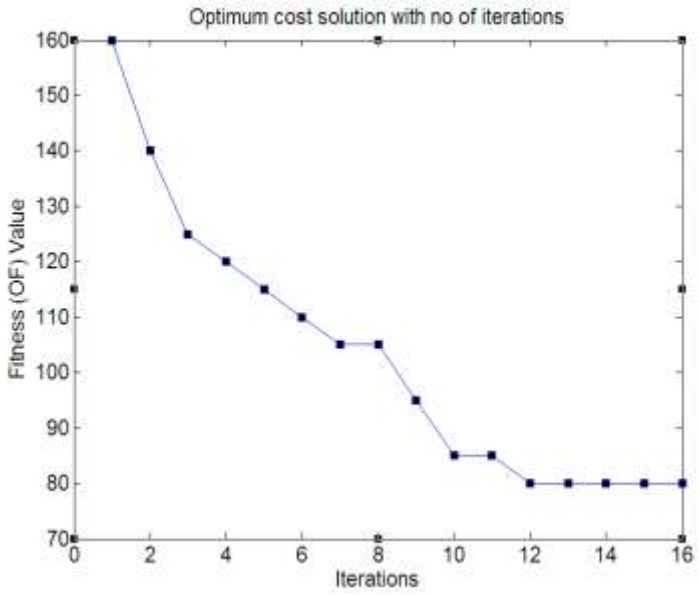


Figure 4.13: Fitness value optimized with iterations in GA for CPU2 - ASIC1

Table 4.10: Partitioning results for different deadlines

Deadline	Opt. Cost	CPU-ASIC	Pattern	Time
120	190	CPU2-ASIC2	01111011	120
140	155	CPU2-ASIC2	00111011	140
160	155	CPU2-ASIC2	00111011	140
180	140	CPU2-ASIC2	00111010	176
190	120	CPU2-ASIC2	00111001	190
200	115	CPU2-ASIC1	00110011	197
210	110	CPU2-ASIC1	00110101	207
220	105	CPU2-ASIC1 CPU2-ASIC2	00011101 00101001	217 220
230	95	CPU2-ASIC1	00011010	230
240	90	CPU2-ASIC1	00011100	240
260	85	CPU2-ASIC1	00110000	260
270	80	CPU2-ASIC1	00011000	270
320	70	CPU2-ASIC1 CPU2-ASIC2	00001000	314 310
360	60	CPU2	00000000	354
580	50	CPU1-ASIC1	00000100	570
660	40	CPU1	00000000	657

Table 4.10 represents optimum cost value and corresponding hardware, software partition for different deadlines. For lower value of a deadline (up to 200) CPU2 – ASIC2 implementation gives best choice because comparatively lower execution time in this implementation, but optimum cost value remains high because more no. of tasks scheduled are in hardware. After deadline of 200 CPU2 – ASIC1 implementation gives best choice and for a considerably larger amount of deadline only one CPU implementation gives optimum cost.

Table 4.11: Partitioning results for different area

Area constraint	Opt. Time	CPU-ASIC	Pattern	Area
40	657	CPU1	00000000	40
50	567 570	CPU1-ASIC2 CPU1-ASIC1	00001000 00000100	50 50
60	354	CPU2	00000000	60
70	310	CPU2-ASIC1 CPU2-ASIC2	00001000	70
80	270	CPU2-ASIC1	00011000	80
85	256	CPU2-ASIC1	00101000	85
90	256	CPU2-ASIC1 CPU2-ASIC2	00101000	85 90
95	230	CPU2-ASIC1	00011010	95
100	206	CPU2-ASIC1	00101010	100
105	220	CPU2-ASIC1 CPU2-ASIC2	00110010 00101001	100 105
110	207	CPU2-ASIC1	00011011	110
115	183	CPU2-ASIC1	00101011	115
135	184	CPU2-ASIC2	01011001	135
140	173	CPU2-ASIC1	10101011	140
155	140	CPU2-ASIC2	00111011	155
180	134	CPU2-ASIC2	01011011	170
190	120	CPU2-ASIC2	01111011	190
200	120	CPU2-ASIC2	01111011	190

		CPU1-ASIC2	11111111	200
210	120	ASIC2	11111111	200

Table 4.11 represents optimum cost value and corresponding hardware, software partition for different area constraints. For lower value of area CPU1 is the best choice because and for high area ASIC2 implementation gives best choice. Optimization of cost depends on no. of tasks, initial population, crossover, mutation probabilities and no. of iterations. Here for particular results no. of iterations are set to 40. More iteration gives a better optimization results, but computations increases. The data of running GA is given in Appendix 2.

4.6. Conclusion

This chapter shows the hardware-software co-design framework for C specification and task graph specification. The partitioning at functional level has been proposed and shown in the co-design flow. ChStone benchmark and a presumed architecture model were used as input for showing the effectiveness of the algorithms. The work starts from basic interfacing of an IP generated using Vivado-HLS. The profiling and genetic algorithm has been successfully shown to give good results. The genetic algorithm deciding parameters like number of iterations and population size were explored. This lays the basic foundation of comparison of the work with any new proposed model of analysis.

Summary:

a) A genetic algorithm was shown to be implemented in co-design flow and actual values obtained from Vivado-HLS and profiling were used.

b) It allowed to show various design chromosomes and guided the designer to chose the one with a given deadline.

REFERENCES

4.1 Narsingh Deo, Graph Theory With Applications To Engineering And Computer Science, Phi Learning, 2014

4.2 L. Jin and S. Chan, A New and efficient partitioning algorithm: genetic partitioning, Circuits and Systems, Proceedings of the 34th Midwest Symposium on, May, 1992.

4.3 R. Chandrasekharam, S. Subhramanian, S. Chaudhury, Genetic algorithm for node partitioning problem and applications in VLSI design, IEE Proceedings E - Computers and Digital Techniques, IEEE, proceedings. Vol. 140, No.5, September, 1993.

4.4 Pierre-Andre Mudry, Guillaume Zufferey, Gianluca Tempesti, A Dynamically Constrained Genetic Algorithm For Hardware-software Partitioning, Proceedings of the 8th annual conference on Genetic and evolutionary computation, pp. 769-776, ACM 2006.

4.5 M. Jagadeeswari, M. C. Bhuveneswari, A Fast Multi-Objective Genetic Algorithm for Hardware-Software Partitioning in Embedded System Design, ICGST-AIML, ICGST-AIML Journal, ISSN: 1687-4846, Volume 8, Issue II, September, 2008.

4.6 A. Bhattacharya, A. Konar, S. Das , Hardware Software Partitioning Problem in Embedded System Design Using Particle Swarm Optimization Algorithm, International Conference on Complex, Intelligent and Software Intensive Systems, March, 2008.

4.7 S. Tosun, Syracuse, N. Mansouri , E. Arvas, M. Kandemir, Reliability-centric hardware/software co-design, Sixth international symposium on quality electronic design (isqed'05), pp. 375-380, March, 2005.

Chapter 5

Static and Dynamic Hardware partitioning for Reconfigurable Computing Systems

In this chapter we present an elaborate approach used to develop framework 2. In this framework we focus on partitioning and scheduling of application which starts from a dataflow specification of a design. The dataflow specification represents the computational part of the design, where nodes are operators like adder etc. and edges represents the communication between them. For partitioning, graph isomorphism has been used to identify similar patterns in the design and partitioned graphs are scheduled based on dependence analysis. A new approach based on partial reconfiguration has been proposed for running a standalone application by creating it as clusters. We explore the effectiveness of genetic algorithm for partitioning and scheduling of such standalone application in partial reconfiguration flow. A modular random task graph generator has been design to generate random load and verify the running time of genetic algorithm. Both the isomorphic and genetic algorithm have been compared on a simulation benchmark. A DCT program has been used to show the design flow and results on Xilinx ML507 board.

5.1. Partitioning and Scheduling of Dataflow Graphs for Reconfigurable Computing Systems

In framework 1, we were working at high level of abstractions in the form of functions in an application. We now move to a lower level of abstraction and explore the design methodologies for RCS. In this chapter we have proposed a new design flow, which starts from the specification in the DFG format and implements a design by partitioning it into SW, HW or a combination of both. These partitions are interfaced and executed on the FPGA based SOC platform. The proposed design flow follows two different paths based on the condition whether static or dynamic scheduling of clusters is required.

The two proposed design flows for framework 2 are presented in this chapter and shown in Fig. 5.1. In this work, an input specification in the form of dataflow model is created manually. The objective of the design flow is to find the best methods for performance and area trade-offs. This is similar to the work done in the chapter 3, but the input specification is in the form of DFG. In approach one, isomorphic graphs are used to find the similar patterns

and these patterns are interfaced and executed statically as IP cores in HW for reusability. The second approach addresses partitioning a standalone design using GA and then these clusters are executed on partial reconfigurable region dynamically.

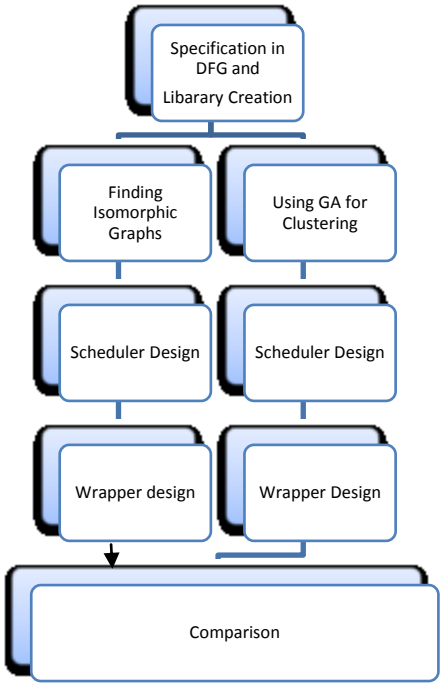


Figure 5.1: Framework 2 design flow

First we present the approach 1 and the step by step method used for partitioning and it is discussed in detailed in subsequent sections. In the Fig. 5.1, we can see the following listed steps:

- The design is specified in a data flow model of computation, which aims at the definition of data communicated between nodes. Dot language [2.39] has been used to describe the specification and capture the graph properties as it is a standard format in many compilers for graph representations.
- The data flow model is specified by arithmetic operators and the edges that represent communication between nodes. The partitioning stage should have information about each operator with it's area and delay on a given platform. For this a library is created and each operator has been written in VHDL language. Similarly the equivalent assembly language instruction for a given processor is considered such as *ADD/SUB/FADD/FSUB* and their time has been calculated on PowerPC processor. These operators have been taken as integer and float type in our design flow.
- The partitioning phase has to decide which nodes to club in a cluster so that minimum time is taken for the execution and various constraints are satisfied. For achieving

partitioning, two different methods have been adopted, one is based on graph isomorphism and the other one is based on genetic algorithm as shown in Fig. 5.1.

- The order of execution is controlled by a scheduler, which is designed to load the partitions in order.
- The generated clusters are then wrapped around the bus signals and interfaced with the bus.
- The design is tested for performance gain using a timer and area/delay product is compared for each solution.

The objective of proposing the design flow is to compare static and dynamic scheduling of an application by comparing the area delay product. The dynamic scheduling highlights the pros and cons of the partial reconfiguration feature available these days in FPGAs and compare it with other possible implementations. The remainder of the chapter is organized as follows: Section 5.3, 5.4 and 5.5 we discuss the design flow based on isomorphic identification. Section 5.6 to 5.10 discusses the use of GA for cluster creation. Section 5.11 proposes a random graph generation method that can be used for generating random loads. Section 5.12 the two approaches are compared. The DCT comparative results are discussed in section 5.13. Section 5.14 draws the conclusion made for framework 2.

5.2. Hardware and Software Synthesis of DFGs

Many C/DSP applications can be described in the form of DFG models for the computational part of the design. The DFG is a directed acyclic graph G (V, E) which contains vertices (V) that are operators and edges (E) that are data dependency. The DFGs have been used as a model of computation for computational intensive part of the application for verification of scheduling and partitioning algorithms [1.7]. DFG is a model which is independent of implementation which means a node can either in the SW or in HW. Fig. 5.2 shows how the nodes can be modeled, wherein each node is characterized by HW: area, delay, power and SW: area, delay and power.

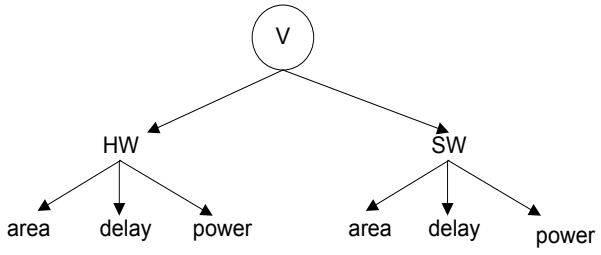


Figure 5.2: Node model

These properties such as area, delay, bit-width, memory usage etc., are obtained from HW or

SW target implementation. For e.g. if the node is an adder, the corresponding HDL synthesis on a tool can give the area, delay and power. Similarly the SW instruction for a processor can give its time, power and code memory occupied. These properties can be used as metrics in algorithmic approaches that define the goodness of a set of nodes. The most common property that is used for goodness is area-delay product; hence throughout the work we have used the same. The implementation of DFGs requires synthesis, which means a description in HW or SW. The synthesis of DFGs can be split into three major steps which are: *allocation, binding and scheduling*. Allocation refers to the process of selecting the components such as processors, ASIC, memory, accelerators, adders, RTL blocks, etc. Binding refers to the process of mapping the functional objects to the library of components. Scheduling step decides the order in which the operator will run. This work uses adder, multiplier, subtractor and divider through for allocation step. These units are designed as integer or floating point units. Hence for the binding phase the operator can be any of the two. Most of the work in this thesis uses floating point units for mapping.

A DFG can be implemented in various combinations of HW and SW. Synthesis of DFGs at very low level of granularity (operator level) may not be advantageous due to ineffective usage of resources. Hence it is required that these nodes are grouped together and form a cluster based on some nearness function. Such clusters should be created in such a manner that the system functional parameters are improved. One such parameter commonly used is area delay product. For comparison of each implementation (SW or HW or hybrid), the performance parameter is required, which can be obtained by timing analysis. To find the total SW time and HW time the following set of equations are proposed for five different implementations as shown in Table 5.1.

Table 5.1: Description of different implementations

Serial Number	Name	Description
1	*Critical_time*	Complete HW implementation of the graph
2	*Serial_time*	Complete SW implementation of the graph
3	*Hybird_time*	HW and SW implementation of the graph having similar patterns
4	*Par_time*	HW and HW implementation of the graph having similar patterns
5	*HW_time*	HW and HW implementation of the graph having no similar patterns

a) HW time referred as *Critical_time* is defined as the maximum time taken to evaluate all the output. Firstly, the end time of each node is required for calculation of critical time, this is given by Eq. 5.1 and is recursively used to calculate the time for each node given by Eq. 5.2. The process starts from the predecessor node and adds up the maximum time taken by each node similar to ASAP schedule. Then the maximum value among those

times is taken to be the critical time of the given output nodes.

$$\textit{Node end time, } T_i = max(T_j, T_k) + d_i \textit{ ...(5.1)} \quad \ldots(5.1)$$

T_j and T_k are the end time of the predecessor nodes, d_i is the delay of the T_i node

$$\textit{Critical\_time, } \quad T_{cr} = \max(t(v_0)) \quad \ldots(5.2)$$

Where, v_o= set of output nodes

b) SW time referred as *Serial_time* which is the time taken to evaluate a given output, when the nodes are running sequentially, such as add (time stamp = add (t_1), sub (t_2), add (t_3), add (t_4), sub (t_5), sub (t_6), add (t_7)) for Fig. 5.5.

$$\textit{Serial\_time, } T_{sr} = \sum_{i=1}^{m}(t_i) \quad \ldots(5.3)$$

Where m = total number of nodes, t_i = time of each node

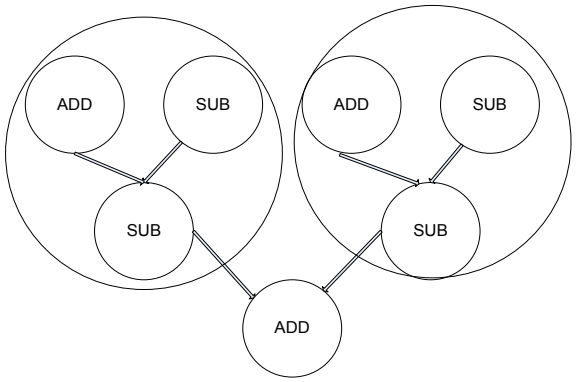


Figure 5.3: Node matching based isomorphic graphs

c) As stated previously, many applications show similar patterns in DFG models. Fig. 5.3 gives an idea of finding the similar patterns that can be used as an optimization of area in the design. For such patterns, HW-SW time is referred as hybrid_time, as proposed in Eq. 5.4. It is the time taken to evaluate a given output with similar part in HW and remaining part in SW. The time in this case is the sum of serial nodes in SW, sum of nodes of similar clusters in HW and communication overhead (bus overhead), which account to certain cycles taken by bus to put/get data. Communication overhead occurs because of the HW clusters are wrapped around the bus interface signals.

$$\textit{Hybird\_time, } T_{hr} = \sum_{i=1}^{n-m}(t_i) + \sum_{i=1}^{l} \sum_{i=1}^{c}(t_{si}) + \sum_{i=1}^{p}(t_{comm}) \quad \ldots(5.4)$$

Where n = total number of nodes, m = number of similar nodes, p = number of cluster edges, c = number of clusters, l is the number of levels, ti = SW time of each node and t_{si} = HW critical time of the similar patterns given by Eq. 5.2.

In Eq. 5.4 the first factor gives the SW time, the second one gives the similar pattern time and last factor contributes to the communication delay. These patterns can occur at any level of the design (many different similar patterns), hence the second summation is required.

d) If the entire implementation is done as HW with various clusters interfaced as IP core, then the HW-HW time referred as *Par_time* is the time taken to execute each cluster plus the communication overhead as proposed in Eq. 5.5.

$$Par\_time,\ T_{pr} = \sum_{i=1}^{n-m} max(t_i) + \sum_{i=1}^{l} \sum_{i=1}^{c} (t_{si}) + \sum_{i=1}^{p} (t_{comm}) \quad \ldots(5.5)$$

where n = total number of nodes, m = nodes of nodes similar t_{si}= time of each node, c = number of clusters, t_i = SW time of each node and t_{si} = HW critical time of the patterns.

e) If no similar pattern exists in the design and clusters are created with certain objectives like an area constraint, then the *HW_time* is used to calculate the total time taken as proposed in Eq. 5.6.

$$HW\_time,\ T_{hr} = \sum_{i=1}^{c} (t_{si}) + \sum_{i=1}^{p} (t_{comm}) \quad \ldots(5.6)$$

where t_{si}= critical time of each node, c = number of clusters and p = number of cluster edges.

Assuming a sample DFG is given as shown in Fig. 5.4 for the verification of equations proposed above. This sample shows the various sizes of patterns found in the graph. The Fig. 5.4(a), 5.4 (b) and 5.4(c) take into consideration the different sizes of the similar clusters.

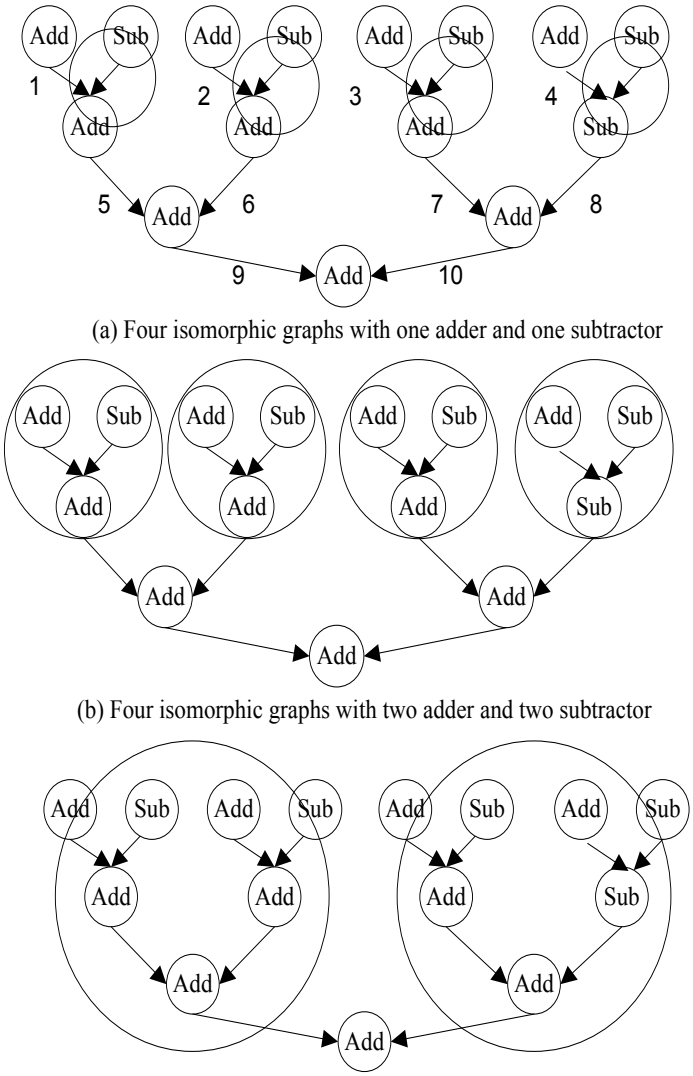


Figure 5.4: Creation of isomorphic clusters at different levels

Let us assume that an floating point adder/sub takes 10 ms in SW, 1 ms in HW (HW is at least 10 times faster than SW), 20 units of area, a basic area of 100 units for a processor for SW implementation, a communication overhead of 2 ms for each edge between SW-HW and 4 ms for HW-HW edge. We are taking these parameters based on the interfacing done in chapter 3. The communication overhead of HW-HW interface has been kept more due to penalty incurred by the bus. Using the data and equations discussed above, the various results of Fig. 5.4 specifications are summarized in Table 5.2.

Table 5.2: Comparison of time taken by the DFG in different implementations

Implementation	Time(ms)	Area	Product
SW	150	100	15000
HW	4	100 + 300 = 400	1600
SW-HW 5.4(a) - hybrid	7x10 + 2x4 + 10(edge)x2 = 98	100 + 40 = 140	13720
explanation	Seven adders are executed serially in SW(7x10), four isomorphic graphs having delay of four are executed in HW(4x2) and there are 10 edges between HW-SW(10x2).		
SW-HW 5.4(b) - hybrid	3x10 + 2x4 + 6x2 = 50	160	8000
	Cluster size increases		
SW-HW 5.4(c) - hybrid	10 + 3x2 + 2x2 =20	240	4800
HW-HW 5.4(a) - Par	1 + 2 + 2 10(edge)x4= 45	80 + 40 + 60 = 180	8100
	In this case three clusters will of HW will be created, C1= four adders(delay =1, area = 80), C2 = iso graphs(delay = 2, area = 40); C3 = three adders(delay = 2, area = 60);		
HW-HW 5.4(b) - Par	2 + 2 + 6x4 = 28	120	3360
	Cluster size increases		
HW-HW 5.4(c) - Par	3 + 1+ 2x4= 12	160	1920

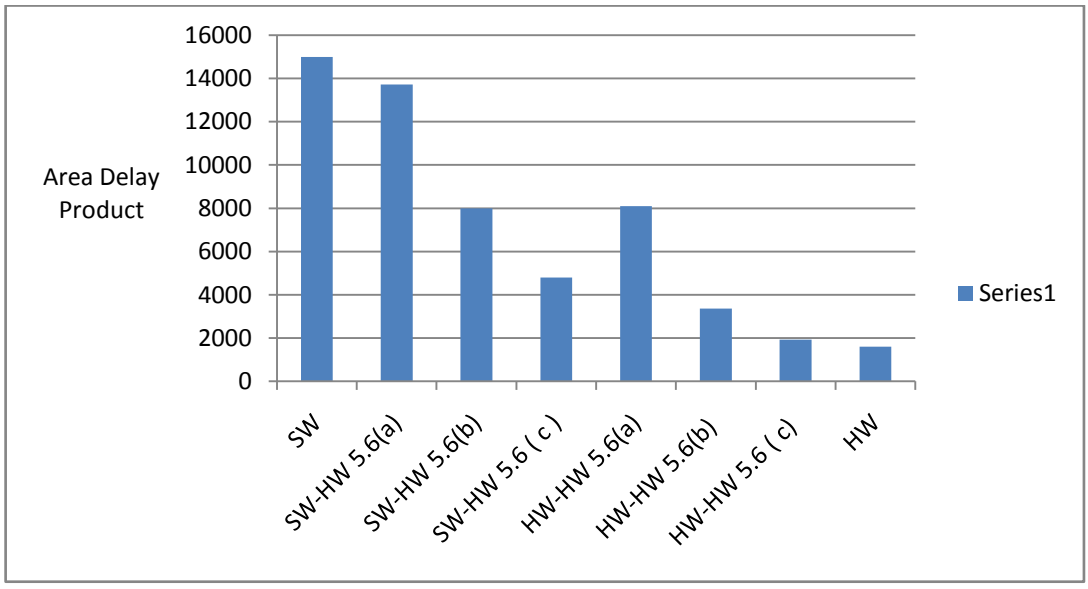


Figure 5.5: Comparison of various implementations

As we move towards the HW migration, the area delay product improves except in the case of HW-HW (a). This is because of delay caused by various HW-HW edges when clusters are small. Hence we can say that the size of the cluster should be adequate to exploit the usefulness of isomorphic graphs. These implementations show different design space for the designer. The Table 5.2 shows the best product for the HW solution and the worst solution in SW with various comparisons shown in Fig. 5.5. In between the two extremes, various other

implementations exist. HW-HW solution is implemented by creating IP cores of the clusters created. Hence we conclude that the idea of reusing the patterns as IP cores can result in a good area and time product. The HW-HW implementation case when no patterns are matching is discussed in genetic algorithm section.

5.3. Algorithmic Approach for Creating Isomorphic Graph

Many applications such as protein structures, image processing etc. show symmetrical structures in their composition or execution model. These structures can be exploited in different ways for improving the execution nature of the application. These repetitive node patterns are known as isomorphic graphs. This section focuses on the efficient development of algorithms for finding such graphs. The graph isomorphism allows finding the similar patterns and creating clusters of the required sizes. The following conditions define an isomorphic graph: Two graph G and H are isomorphic if H can be obtained from G by relabeling the vertices - that is, if there is a one-to-one correspondence between the vertices of G and those of H, such that the number of edges joining any pair of vertices in G is equal to the number of edges joining the corresponding pair of vertices in H. Additionally, for reconfigurable computing systems it is necessary that the node type should also match, which means that the adder should match with adder. Hence, next we propose a weighted method to find the subgraphs. Entire algorithm is divided into four parts and the input parameters of the algorithm are: matrix of graph G (V, E), type of the node, area/delay of the node and number of nodes. In part 1 (Weight algorithm), for each individual node in the graph, its level, type, degree, area, time and weight is calculated. In part 2 (Subgraphs algorithm) level based subgraphs are constructed using edge information. In part 3 (Iso algorithm), the weight of various subgraphs is compared and isomorphic subgraphs are found. In part 4 (Performance algorithm), the critical time for executing the graph is calculated. The algorithms are shown in Fig. 5.7 to Fig. 5.10.

5.3.1. Weight algorithm

The first step toward comparing the subgraph matching is by finding a unique method that simplifies the process. A weighted method [2.43] has been used to assign a unique number to each node, which is further used in the comparison process. What we have added is a type number to the weight function which differentiates between the nodes as adder, subtractor etc. The objective of the algorithm to find the various parameters such as the weight of a node which includes its level, type & total degree as defined by proposed in Eq. 5.7, Eq. 5.8, Eq. 5.9 and Eq. 5.10:

$$Weight[node] = k_1 * Level[node] + k_2 * type[node] + len(connectionsOfNodes[node]) * type[node]; \quad \ldots(5.7)$$

$$Weight\ of\ subgraph = \sum weight\ of\ its\ nodes \quad \ldots(5.8)$$

$$Area\ of\ subgraph = \sum Area\ of\ its\ nodes \quad \ldots(5.9)$$

$$Time\ of\ subgraph = Time\ for\ its\ critical\ path \quad \ldots(5.10)$$

k_1 and k_2 are the constants and have been assigned a value of k_1 =1000, k_2 =100. The weight equation has been proposed in such a way that at any point of time the weight of any node will be same only if they match in all ways. If more number of nodes are there, we can increase the value of constant but always keep $k_1 : k_2 >> 1$. We have used that ratio to be greater than one, so that the weight of a node is always unique. For a random sample shown in Fig. 5.6, the value for node 1 are level = 1, length =1, type =1 which gives weight as 1101.

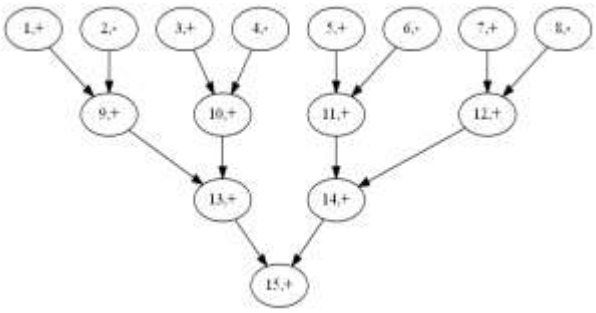


Fig. 5.6: Sample graph

The above graph contains 15 nodes and their weight values are specified in Table 5.3.

Table 5.3: Weight of the nodes

Node Number	Level	Length	Type	Weight Calculation	Weight
1	1	1	1	1000x1+100x1+1x1	1101
2	1	1	4	1000x1+100x4+1x4	1404
3	1	1	1	1000x1+100x1+1x1	1101
4	1	1	4	1000x1+100x4+1x4	1404
5	1	1	1	1000x1+100x1+1x1	1102
6	1	1	4	1000x1+100x4+1x4	1404
7	1	1	1	1000x1+100x1+1x1	1101
8	1	1	4	1000x1+100x4+1x4	1404
9	2	3	1	1000x2+100x1+3x1	2103
10	2	3	1	1000x2+100x1+3x1	2103
11	2	3	1	1000x2+100x1+3x1	2103
12	2	3	1	1000x2+100x1+3x1	2103
13	3	3	1	1000x3+100x1+3x1	3103
14	3	3	1	1000x3+100x1+3x1	3103
15	4	2	1	1000x4+100x1+2x1	4102

The specification of the input has been done in the dot language using the IBM Graphviz software. While traversing the dot file, nodes and their connections are identified. To determine the level of each node, traversing has been made to list the connection that contains

the source and destination node of an edge. If a node is encountered for the first time, then its level is set to 1. For the connection between two nodes, we set the level of the successor node as one more than the level of predecessor node. In case, if a successor has multiple source nodes, then the maximum value of the level of source node is considered. After determining the level of every node, the weight for each node is to be calculated based on the weight Eq. 5.7. The time complexity of this algorithm if O(n), since iteration is done for level and weight for each node. The step by step details of the algorithm is shown as a flow chart in Fig. 5.7.

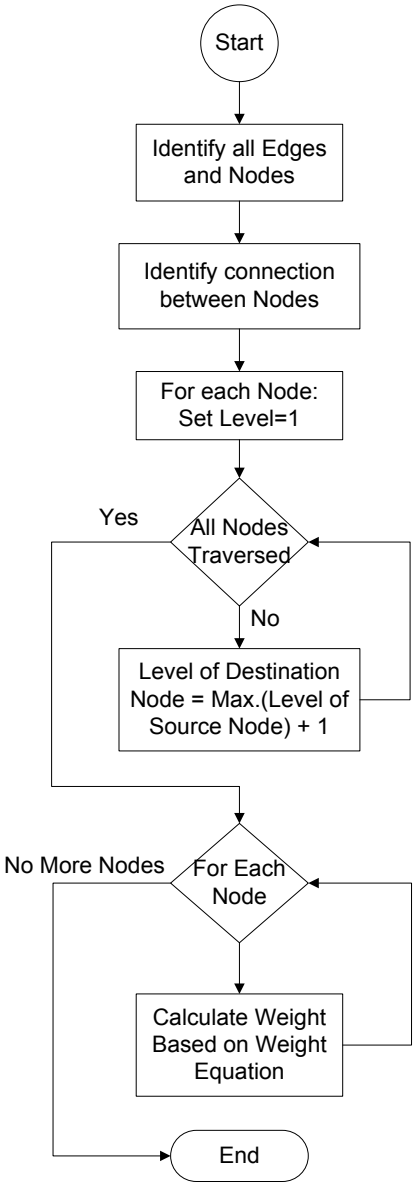


Figure 5.7: Weight algorithm

5.3.2. Subgraph Algorithm

Next stage of the algorithm proceeds by finding the subgraphs as shown in Fig. 5.8. We have taken two variables as i and j where range of i is 1 to the number of levels in the graph, let's say L, and range of j is i+1 to L. We traverse the graph level wise, and for each node in a

level, search for subgraphs. We add the current node to temporary subgraphs and to the list of added nodes. For all the other nodes that are connected to the nodes in the list 'added nodes' and the current node, we then check if they fulfil the criteria of the current value of i and j and add these nodes to the temporary subgraphs. If at any stage, any condition goes false, then add the temporary subgraphs to the list of all subgraphs. The above process is repeated for all the nodes traversing level wise in the graph recursively. The time complexity of this algorithm if $O(n(\log n)^2)$, since iteration is done for level and subgraphs, n is the number of nodes.

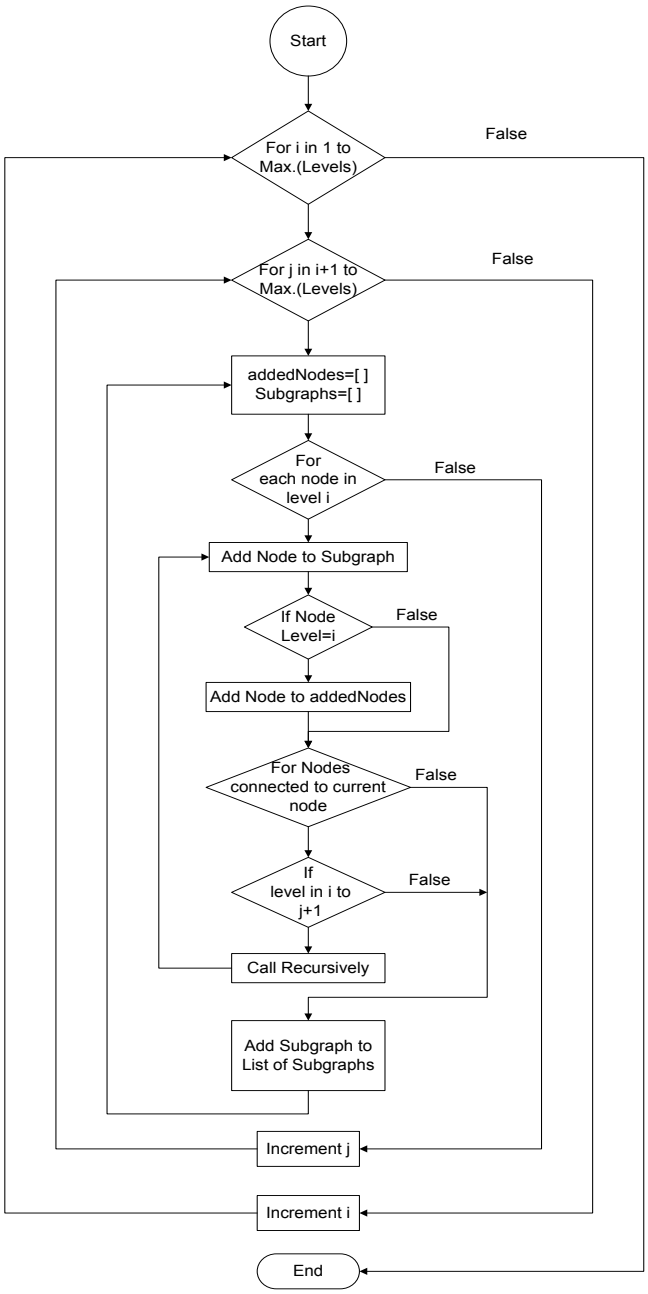


Figure 5.8: All subgraphs algorithm

5.3.3. Iso Algorithm

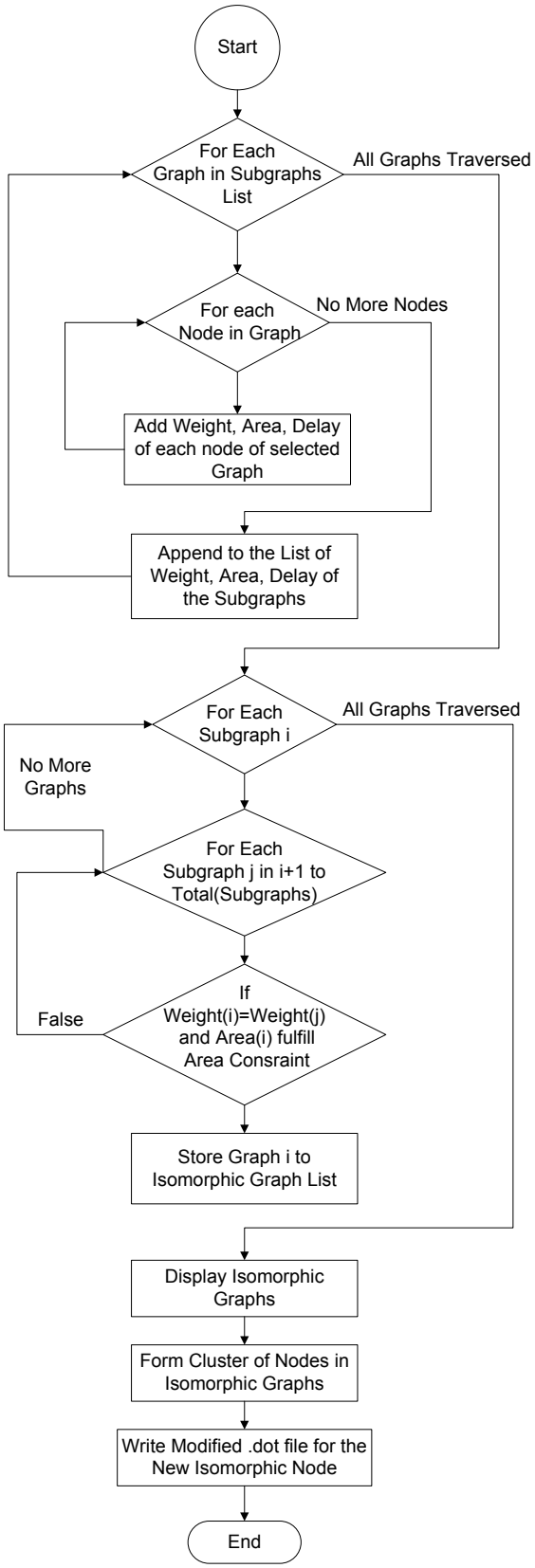


Figure 5.9: Iso algorithm

For each graph in the list of all subgraphs, we calculate the weight, area, delay of every graph from the node parameters as defined by Eq. 5.8. We make a record for every graph with its

weight, area and delay in a dictionary. Next we select any graph from the list of subgraphs and match its weight parameter with all the remaining graphs from the list. If the weight of two subgraphs matches, then both of them are added to the isomorphic graphs list. This is done for all the graphs present in the subgraphs list. The duplicate isomorphic graphs are sorted and removed. Then we create clusters of nodes that form an isomorphic graph with any other, and rename the cluster. At the end, we create a new dot file that contains all possible isomorphic graph nodes and other individual node connected to each other. The algorithm is shown in Fig. 5.9. The time complexity of this algorithm if $O(n^2)$, since iteration is done for subgraphs matching for each item in the list. The combined complexity of the previous three algorithm is $O(n) + O(n(\log n)^2)+O(n^2)$, which is much better as compared to brute force method.

5.3.4. Performance Algorithm

This algorithm finds the performance in terms of time after the isomorphic graphs are identified and condensed along with other nodes. For finding the critical time, source node should always be at level 1. For all such nodes that are at level 1, we find the outgoing edges and the corresponding destination node. Next we add the time of the source node at the initial level to a temporary variable which was initialized as 0 at the beginning. Then we add the time of the first destination node. Next step is to modify the destination node as the node connected to present destination node and adds its time to the temporary time variable. This process is repeated until the destination node reaches the maximum level of the graph. After this, we add the value of temporary time variable to a list of possible critical time.

This is done for all the destination nodes that were encountered at the beginning of the source node. If no more destination node is left, then change the source node if any, and repeat the above algorithm. The maximum value from the list of possible critical time will be the critical time of the graph. The detailed algorithm is shown in Fig. 5.10.

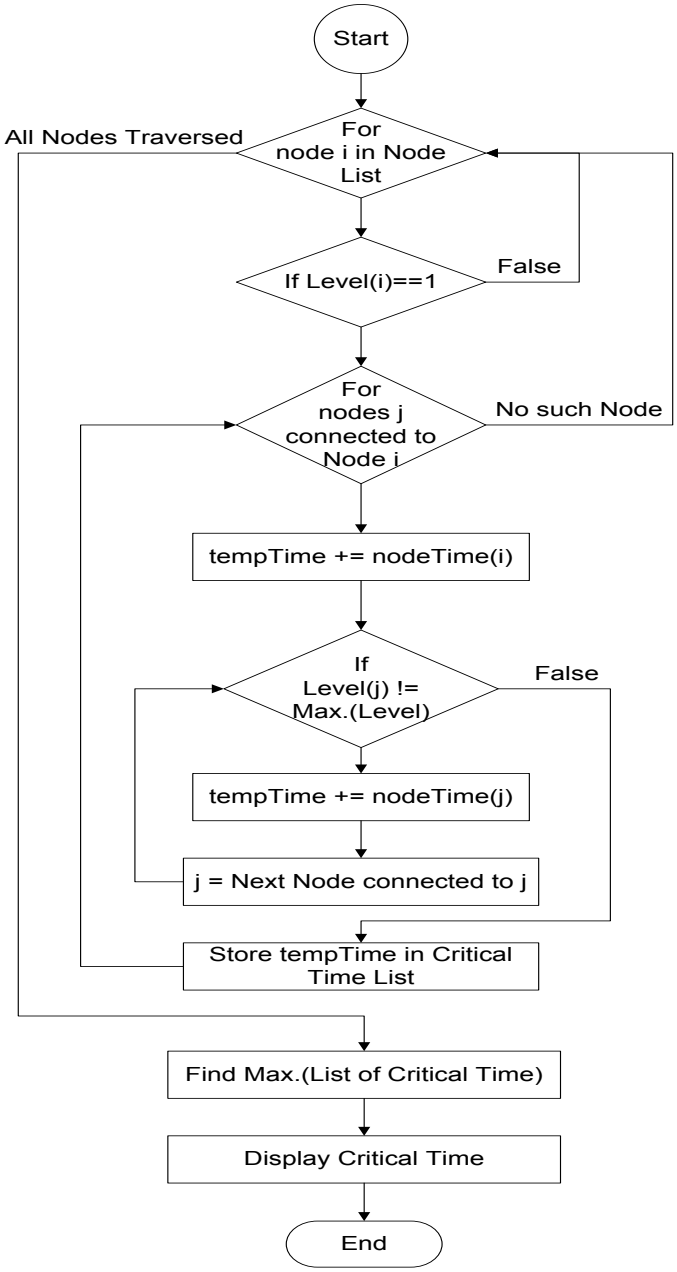


Fig. 5.10: Performance algorithm

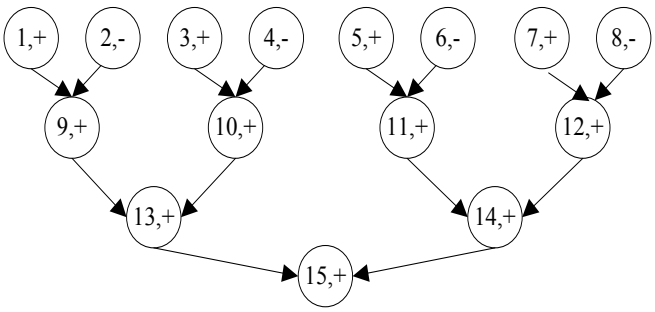


Figure 5.11: Sample graph

In order to verify the results of the above algorithms a sample graph as shown in Fig. 5.11 was taken for explanation. Each node shows the number and an operator in it such as node 1 is adder. There are 15 nodes in the graph with four kinds of operators (+, -, *, /). These four operators were written in VHDL language and complied in Virtex-5. The critical time and area of each operator is shown in Table 5.4.

Table 5.4: Parameters for sample graphs

Symbol	Type	Time (ns)	Area (Slice LUTs)
+	1	6.853	32
-	4	6.853	32
*	8	13.788	3 DSP slices
/	16	67.271	882

Table 5.5: Sample graph results

Subgraph1	Subgraph2	Time (ns)	Area (Slice LUTs)
1 9 2	3 10 4	13.706	96
1 9 2	5 11 6	13.706	96
1 9 2	7 12 8	13.706	96
3 10 4	5 11 6	13.706	96
3 10 4	7 12 8	13.706	96
5 11 6	7 12 8	13.706	96
11 14 12	10 13 9	13.706	96
1 9 2 13 10 3 4	5 11 6 14 12 7 8	20.559	224

Table 5.5 shows the results of the algorithm discussed. Node 1, 9, 2 is isomorphic with node 3, 10, 4 having a critical delay of 13.7 ns. Similarly, node 1, 9, 2, 13, 10, 3, 4 is isomorphic with 5, 11, 6, 14, 12, 7, 8, with an area of 224.

5.4. Scheduler Design

The previous stage gave information about which of the nodes are similar and form a cluster. It then gave the critical time of the cluster. The next step is to introspect two factors :

a) Whether a given node is isomorphic if yes make an entry in scheduling queue if not.

b) Create a cluster of a given size.

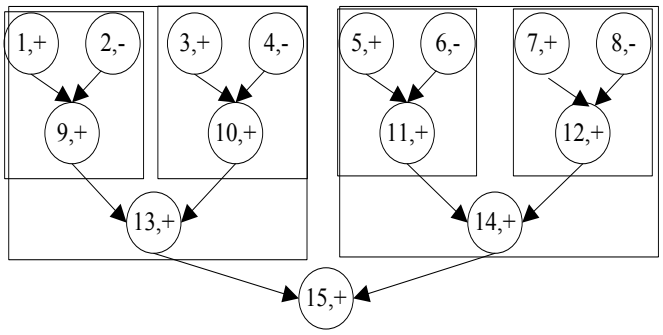


Figure 5.12: Various clusters in sample graphs

Fig. 5.12 shows the concept applied in this phase. We can run (1,2,9; 3,4,10; 5,6,11; 7,8,12) or (1,2,9,3,4,10; 5,6,7,8,11,12) based on the area constraint given. After making this decision, we need to put clusters in order of execution.

In the scheduling phase, we need to construct clusters of a given size. Our algorithm is an extension of list scheduling algorithm [1.7] for reconfigurable computing. Since list scheduling does not consider the hierarchical structure of the program in terms of functions and instructions, it needs certain modifications. A better way to schedule this is using LIST scheduling [1.7], which takes the priority and then schedules accordingly. LIST scheduling still works at operator level and needs to be refined for partial reconfigurable systems. The next section aims at developing the mathematical background for DFGs as HW or SW implementation.

The first step towards applying partitioning is the creation of hierarchical clusters as the previous stage gives nodes which are isomorphic. The algorithm is described in algorithm 5.1 along with comments shown in each line.

Algorithm 5.1: Clustering Algorithm

Input : Graph as Dot files of the application

Given: The total area(TA) of the partition in terms of LUTs and ISO graphs. Select ISO clusters nearest to given area.

Output: partitions of blocks.

create a partition PA[j] := Ø *// create a NULL cluster*

for i =1 to n *// loop till all the nodes are covered*

if(Node[i] == ISO) *//check if the node is a isomorphic graph*

Add node to scheduling queue. *// update the scheduler queue*

else if ((area(PA[j])+ area(Node[i]) ≤ TA) //check if the cluster size has reached

PA[j] = PA[j] ∪ Node[i] *// if the node is not iso. and cluster area is there add node*

else

Add PA[j] to scheduling queue. *//Add the node to the scheduling queue*

Create a new partition j = j+1 *//create a new cluster*

Add Node[i] to PA[j].

end if

end for

Suppose according to the size given (in which two adder and one subtractor can be placed) four isomorphic graphs are identified that are (1,2,9), (3,4,10), (5,6,11), (7,8,12) and the remaining 13, 14 ,15 form one cluster. Then the scheduler will create a scheduling queue with order: (1,2,9), (3,4,10), (5,6,11), (7,8,12), (13, 14 ,15) as shown in Fig. 5.13.

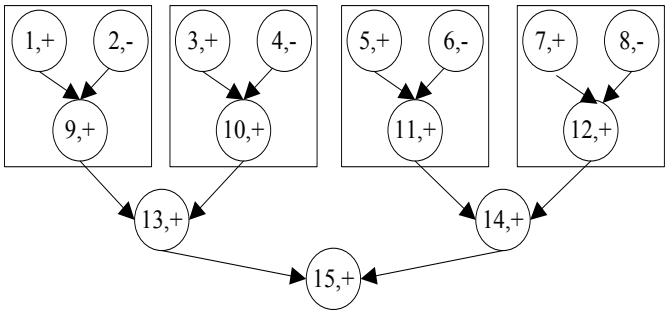


Figure 5.13: Sample graph with isomorphic clusters

After the creation of the clusters, the scheduling queue maintains the order in which the nodes will execute. The pseudo code for the calculation of execution time as proposed in Eq. 5.1, Eq. 5.2, Eq. 5.3 and Eq. 5.4 is shown in algorithm 5.2.

Algorithm 5.2: Scheduling Algorithm

Input: Scheduling queue, ISO nodes level (K)

Given: the performance algorithm proposed in 6(d)

Output: execution time, area and product.

j =0; t_{comm} *=0;area = 0,product=0;*

for i =1 to n

If(Node[i]= ISO or HW)

tc = Call critical time algorithm(Node[i])

area = area + area[i];

else

tc= Call serial time algorithm(Node[i])

end if
execution time T= tcomm + tc;
*product = area * T;*
end for

Let us consider an example to understand the algorithms presented above. Assume that an adder/sub takes 10 ms to run SW, 1 ms to run in HW, then for the sequence: (1,2,9), (3,4,10), (5,6,11), (7,8,12), (13, 14 ,15), the time will be : 30, 30, 30, 30, 3 resulting in the total time to be 123 ms. The above scheduling algorithm gives the clusters that are to be executed in a given order in a queue. This queue tags each cluster with either an ISO cluster, SW or HW cluster. It also indexes the ISO variation at each level. The algorithm picks each cluster at a time and finds whether it is an ISO, SW or HW cluster and respectively calls the algorithm to calculate the time taken by the cluster node. This provides the time taken for the execution of the application. This algorithms implements the Eq. given by 5.4, 5.5. The communication delay is then added to the total time. In the next section we will discuss the result on two grounds:

a) Simulation of the above algorithm on four self written programs
b) Using DCT program as case study on ML507 Board.

5.5. Results and Discussion for Isomorphic Design flow

In order to verify the effectiveness of the algorithm, four programs were created. These are listed in Table 5.6, which also highlights the total resource usage of the program. All the algorithms were written in python and test cases were written in dot language.

Table 5.6: Programs used for testing

Design	Kind	Nodes	Edges	Inputs	Outputs	Resources Used	
						LUTs	DSP
Cosine series(float)	Self written	15	31	5	1	7280	33
Exponent (float)	Self written	34	69	8	1	15300	75
Matrix Multi(3x3-Integer)	Self written	45	99	18	9	576	81
Sine series(float)	Self written	18	37	4	1	7427	42

These manually created benchmarks programs have four operators, which are adder, subtractor, divider and multiplier.

Table 5.7: Library of hardware blocks and their values on Xilinx ML507 board

Node Type(1)	Resource(2)					Delay(ns)-HW(3)	Delay(ns)-SW(PowerPC)(4)
	Slice Registers	Slice LUTs	LUT FF pair	IOBs`	DSP48E		
Integer adder	0	32	32	98	0	6.8	50.0 ***-add***
Integer subtractor	0	32	32	98	0	6.8	50.0
Integer multiplier	-	-	-	98	3	13.7	65.0
Integer divider	0	882	822	115	0	67.2	90.7
FP adder(FPU unit)	224	621	705	99	0	7.7	60.0-***fadd***
FP subtractor(FPU unit)	223	628	714	99	0	7.8	60.0
FP multiplier(FPU unit)	0	49	49	97	3	15.8	70.0
FP divider(FPU unit)	24	1654	1654	99	2	110.8	270.0

A library of these operators was created by writing VHDL code and tabulating their parameters on Virtex-5 [5.1] series. These actual values were used by the algorithms for generating partitions. Appendix 3 shows how the time for SW instructions was obtained. The Table 5.7 is divided into four major columns: name of the unit (integer or floating), resources, HW synthesis time and SW instruction time. The floating point unit was enabled in PowerPC for floating point instructions (fadd). The last column shows the SW execution time for PowerPC available on ML507 board. The SW execution time is tabulated by executing a floating point instruction in loop for 100 times and taking the average. The Table 5.7 highlights the two major resources that used by an application: which are LUT slices and DSP slices. In algorithmic design we need a unified value corresponding to the area consumed. These different resources are combined together in a single equation as proposed in Eq. 5.11.

$$Area = k_1 \, x \, LUT + k_2 \, x \, DSP \qquad \ldots(5.11)$$

Where k_1=1, k_2 = 50, since DSP slices are much less as compared to LUTs k_2 has been kept high since DSP slices are at least 50 times lesser to LUTs.

XPS Synthesis Summary (estimated values)				
Report	**Generated**	**Flip Flops Used**	**LUTs Used**	**BRAMS Used**
system	Mon Nov 7 16:53:20 2016	6498	7185	4
proc_sys_reset_0_wrapper	Mon Nov 7 16:52:30 2016	69	53	
jtagppc_cntlr_inst_wrapper	Mon Nov 7 16:52:26 2016		2	
clock_generator_0_wrapper	Mon Nov 7 16:52:22 2016	4	3	
ppc440_0_apu_fpu_virtex5_wrapper	Mon Nov 7 16:52:18 2016	2629	4204	
ppc440_0_fcb_v20_wrapper	Mon Nov 7 16:52:02 2016			
ddr2_sdram_wrapper	Mon Nov 7 16:51:59 2016	2355	1770	2
xps_timer_0_wrapper	Mon Nov 7 16:51:16 2016	357	290	
sram_wrapper	Mon Nov 7 16:51:04 2016	544	316	
rs232_uart_2_wrapper	Mon Nov 7 16:50:51 2016	144	127	
xps_bram_if_cntlr_1_bram_wrapper	Mon Nov 7 16:50:39 2016			2
xps_bram_if_cntlr_1_wrapper	Mon Nov 7 16:50:35 2016	255	203	
plb_v46_0_wrapper	Mon Nov 7 16:50:27 2016	139	214	
ppc440_0_wrapper	Mon Nov 7 16:50:18 2016	2	3	

Figure 5.14: Resource consumed by a basic design in ML507 Board

Figure 5.14 shows that a basic processor based design flow will take 7185 LUTs. Hence for pure SW implementation, we have taken a value of 8000 in area calculation. The SW to HW communication and HW to HW communication is taken to be negligible. These values have been used in calculating the performance of the design for simulation purpose.

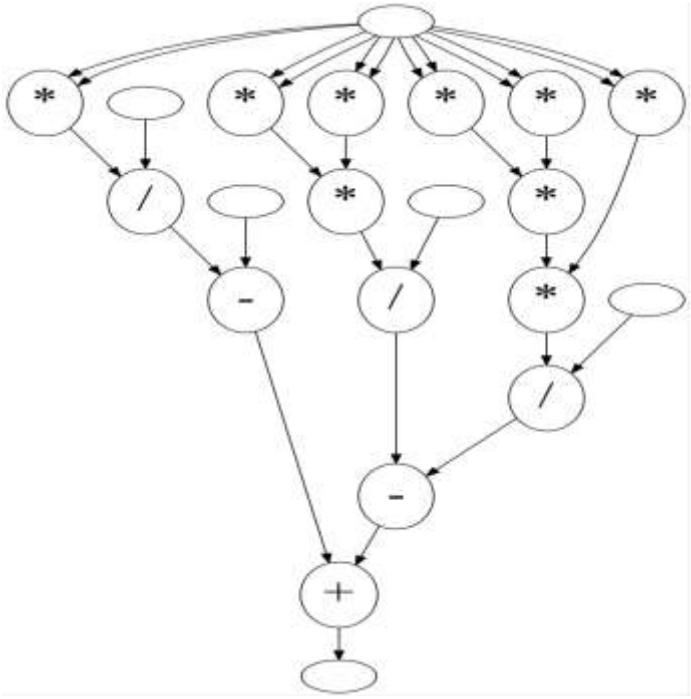


Figure 5.15: DFG of Cosine series *(1 - x^2/2! + x^4/4! - x^6/6!)*

Fig. 5.15 presents the dataflow graph for the cosine series upto four terms. The empty circles are representation of constant factorial terms which are pre-computed and stored in SW. Fig. 5.16 shows the isomorphs in the cosine series.

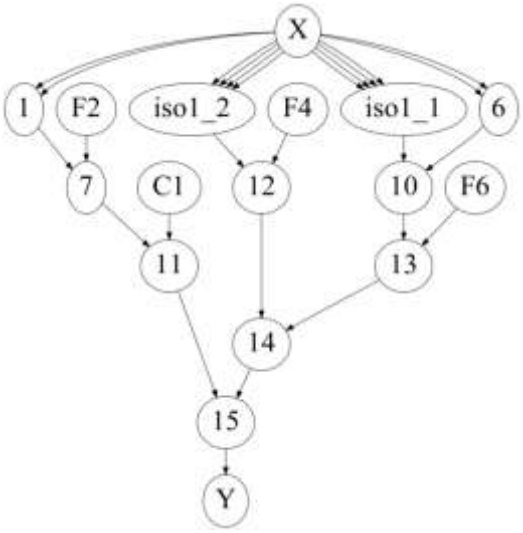


Figure 5.16: Cosine Series with Isomorphic Graphs

Table 5.8: Comparison of area delay product for Cosine function

Design	COSINE		
operators	1-Adder,1-Sub,9-Mul,3-Div		
	Time(ns)	Area	Product(LUT-sec) (x10^6)
SW Implementation	1560	8000	12.48
HW Implementation	173.7	16320	2.834
SW-HW Implementation	1155.8	8597	9.936
HW-HW Implementation	188.8	15705	2.965

Table 5.8 shows the time, area and product for four different implementations of cosine function. We can see that SW Implementation has the maximum product and HW Implementation has the minimum value. For all the benchmarks the SW time is the upper limit and HW is the lower limit. Cosine benchmarks show good results in HW-HW implementation where two isomorphic subgraphs are identified.

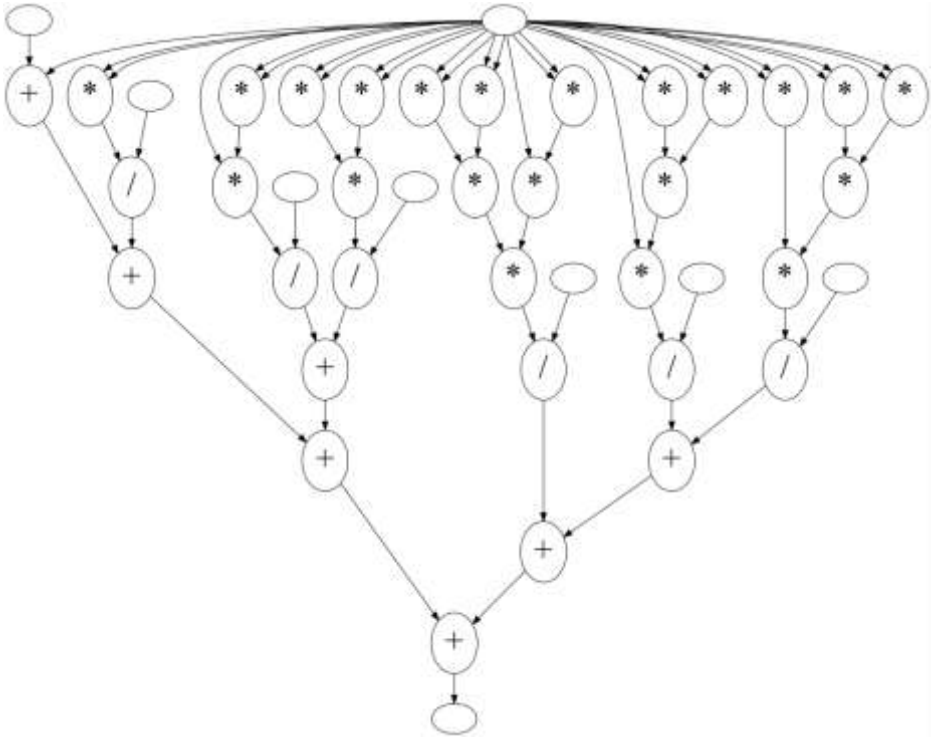


Figure 5.17: DFG of Exponent series *(1 + x + x^2/2! + x^3/3! + x^4/4! + x^5/5! + x^6/6!)*

Fig. 5.17 shows the dataflow graph for the exponent series upto seven terms. The empty circles are representation of constant factorial terms are pre-computed and stored.

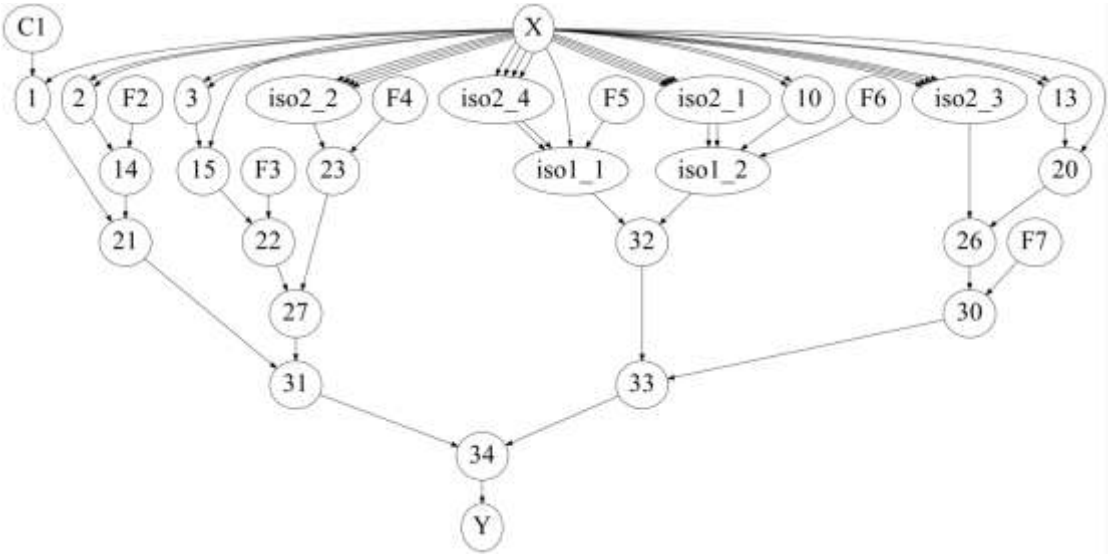


Figure 5.18: Exponent series with isomorphic graphs

From Fig. 5.18 we can see that four isomorphic graphs are there at level 1(iso2_2) and two at level 2(iso1_1).

Table 5.9: Comparison of area delay product for Exponent function

Design	EXPOENET		
Operators	7-Adder,21-Mul,6-Div		
	Time(ns)	Area	Product(LUT-sec) ($x10^6$)
SW Implementation	3510	8000	28.08
HW Implementation	181.3	27050	4.904
SW-HW Implementation	1871.4	12503	23.398
HW-HW Implementation	608.5(critical delay is long)	22908	13.939

Table 5.9 shows the time, area and product for four different implementations of exponent function. We can see that SW Implementation has the maximum product and HW Implementation has the minimum value. Exponent benchmarks show the good results in HW-HW implementation compared to software. When the density of isomorphic graphs is high, HW-HW implementation will give better results. Fig. 5.19 shows the dataflow graph for the matrix multiplication for 3x3 terms. Nine isomorphs for matrix multiplication series are shown in Fig. 5.20.

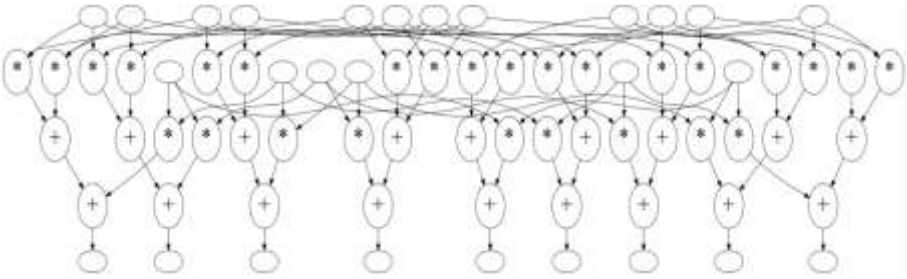


Figure 5.19: Matrix Multiplication for 3x3 elements

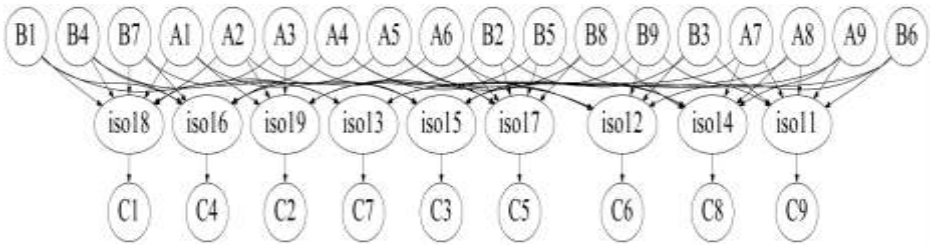


Figure 5.20: Matrix Multiplication for nine isomorphic graphs

Table 5.10: Comparison of area delay product for Matrix function

Design	MATRIX		
Operators	18-Adder, 27-Mul		
	Time	Area	Product($x10^6$)
SW Implementation	52655	8000	21.24
HW Implementation	27.3	12626	0.344
SW-HW	245.7	8514	2.091

Implementation			
HW-HW Implementation	245.7	8514	2.091

Table 5.10 shows the time, area and product for four different implementations of matrix multiplication function. We can see that SW Implementation has the maximum product and HW Implementation has the minimum value In the matrix multiplication density of isomorphic graphs is even higher, hence it is much better to pure SW implementation. In this case since one isomorphic graph can be used for entire execution, the SW-HW implementation is also giving same results.

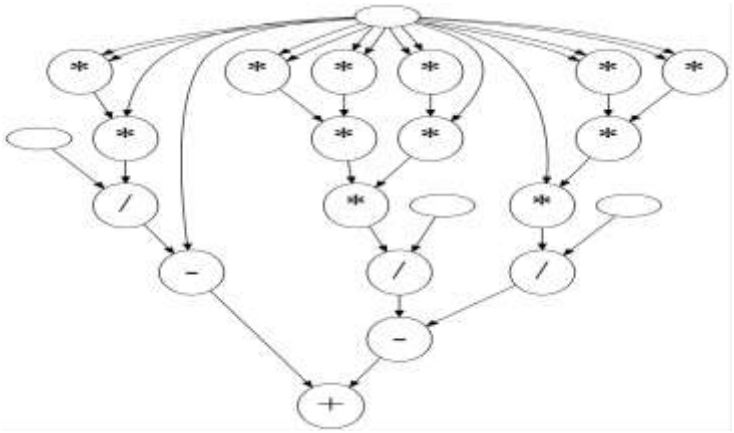


Figure 5.21: Sine Series with five elements$(x - x^3/3! + x^5/5! - x^7/7!)$

Fig. 5.21 shows the dataflow graph for the sine series upto four terms. The empty circles are representation of constant factorial terms are pre-computed and stored.

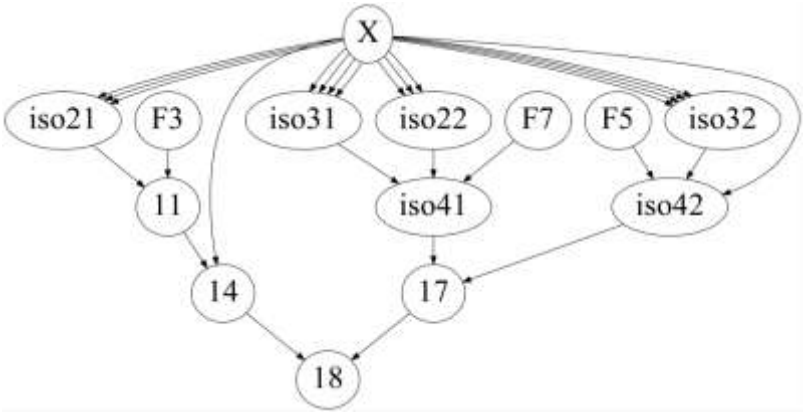


Fig. 5.22: Sine Series with three isomorphic graphs

From Fig. 5.22 we see that four isomorphic graphs are there at level 1 and two at level 2.

Table 5.11: Comparison of area delay product for Sine function

Design	SINE		
Operators	1-Adder, 2-Sub, 12-Mul,3-Div		
	Time	Area	Product(x10^6)
SW Implementation	1830	8000	14.65
HW Implementation	173.7	17527	3.05
SW-HW Implementation	814.8	12901	10.511
HW-HW Implementation	411.1	14579	5.991

Table 5.11 shows the time, area and product for four different implementations of sine function. We can see that SW Implementation has the maximum product and HW Implementation has the minimum value

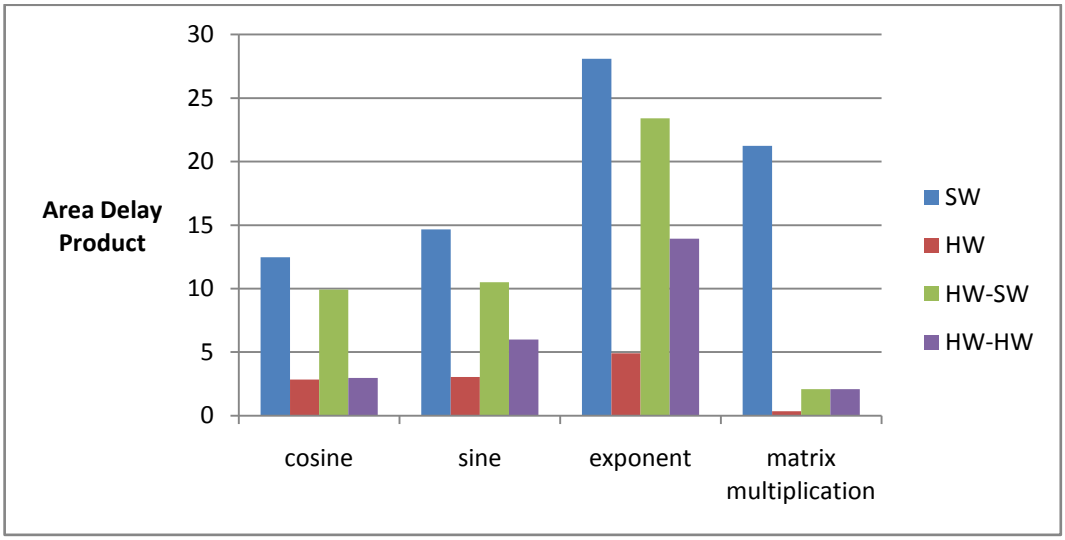


Figure 5.23: Comparison of four benchmark programs

Fig. 5.23 shows the comparison of four benchmarks programs. The HW-HW implementation the product is better than SW-HW but inferior to HW implementation. In order to create a clear practical approach of our implementation in real HW, we have taken a Discrete Cosine Transform (DCT) as a case study in result section.

5.6. Partitioning and Scheduling Problem for Partial Reconfiguration

The design flow of partial reconfiguration has been discussed in chapter 1. This section extends the implementation of DFG nodes on partial reconfiguration region. The partial reconfiguration technique allows running different modules on the same defined area, hence resulting in area reusability. This feature of reusability gives a new dimension to the allocation, binding and scheduling problem. The usage of the partial reconfiguration technique requires the modification to existing partitioning and scheduling solving approach. The

primary purpose of this design flow is to test the usefulness of partial reconfiguration in HW-HW implementation.

In this design flow the same application is partitioned and mapped to a fixed area for execution. This is explained with the help of the Fig. 5.24, which shows how a standalone design is partitioned into two clusters (C1 and C2) and bound to one PR region:

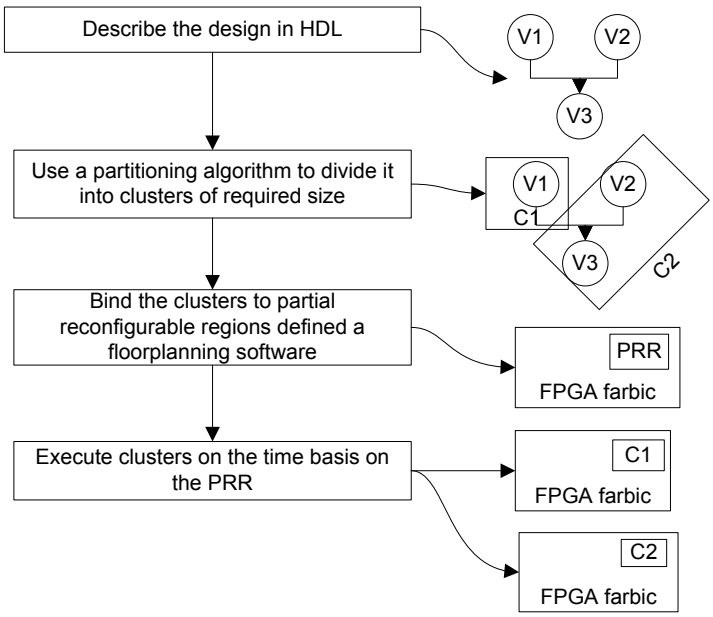


Figure 5.24: Partitioned design running on PRR

In the case of partial reconfiguration flow it is also possible to define many PR regions on which different modules can run concurrently making it an active area of research. The PR tutorial [2.64] gives a design flow of an adder/multiplier mapped to one PR region. Now let us assume that two PR regions (Mult/Mult or one Mult/Add or Add/Add) are available for mapping. The floorplan is visualized in Fig. 5.25 and the schedule is shown is Fig. 5.26 for the same graph discussed in the section 5.1.

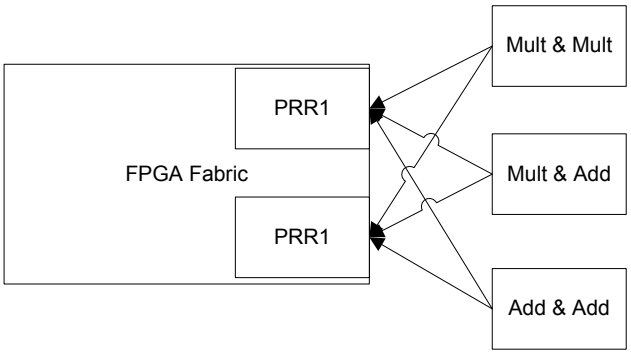


Figure 5.25: Two PR regions with three PRM

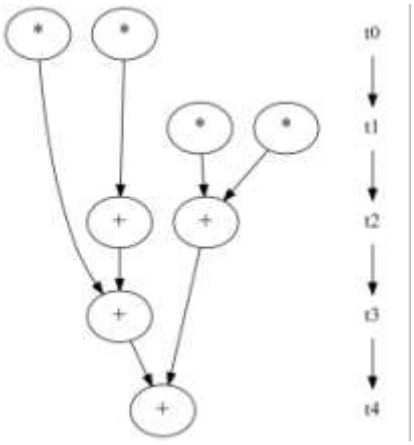


Figure 5.26: Two PR regions Schedule

From Fig. 5.26 we see that at each level two operators are scheduled. This becomes feasible by mapping two operators on PRR1 or PRR2. In the current work we have assumed only one PRR is available for a single standalone application. This requires partitioning the DFG and then effectively scheduling it on one PR region is required. An algorithm which can schedule DFG on this PR region with minimum reconfiguration overhead and minimum execution time is the call for of the design flow. The PR design flow requires swapping in and swapping out of the bit file of the operator. It incurs certain delay in the design flow as overhead, so for such an implementation, we need to find the overhead of the reconfiguration process. For this, we consider four time parameters to calculate the total time required for executing the DFG as defined in Eq. 5.12:

$$t_{conf} = \text{Loading time} + \text{input/output transfer time} + \text{execution time} \quad \text{...(5.12)}$$

Configuration time (t_{conf}), is the time it takes to place the bit file of the PR module on the chip. The bit files are usually placed outside the FPGA chip on non-volatile memory such as compact flash card, SD card etc. During the boot operation, these files can be brought to DDR or BRAM memory on the chip.

Execution time (t_{exe}) is defined as the time required to execute the operation on the HW. Input time t_i is defined as time to place input data. There is certain time elapsed when sending the inputs from I-cache/D-cache to the IP core interface to the bus. Similarly Output time t_o is defined as time to place output data. N is the number of times the reconfiguration is done.

The penalty of t_{conf} is very high, as the bit files are located outside the chip on a memory like CF card, DDR or Flash. Hence there is a need to minimize the number of times the reconfiguration is loaded.

Brining down the penalty has been analyzed extensively over the last decade and many works

exist in this respect as discussed in literaure survey [2.61-2-67]. For such a scenario where we want to minimize the number of times the reconfiguration is done, an algorithm can be proposed:

Starting with an area which implements an operation like adder and multiplier, we propose an algorithm to check if the next operator which is loaded is same as the current then we can save on loading a new configuration.

Algorithm 5.3: Minimum Time Reconfiguration List Scheduling (MTR)

Inputs : DFG and FPGA area constraints

Outputs : set of functions to be migrated

Begin :

Sort nodes in topological order;

Assign priority to nodes based on mobility;

label x: Place maximum priorty node // priority using levels

If next module can be same as present

$t = t + t_{exe} + t_i + t_o$

else

$t = t + t_{conf} + t_{exe} + t_i + t_o$

If DFG in not NULL go to x

The algorithm to schedule the operators on one PR with minimum latency is shown in algorithm 5.3 and is referred a MTR. The schedule is shown is Fig. 5.27. In this algorithm, it is checked whether the next node to be placed can be same as present if yes we can save on $t_{conf.}$

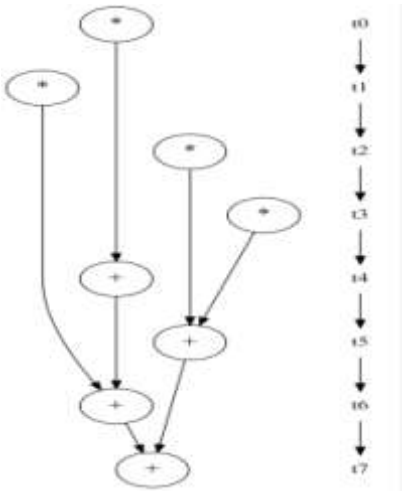


Figure 5.27: PR schedule

In the above DFG, reconfiguration will be done only once since the multiplier is replaced with

an adder only once. For e.g. if the multiplier HW time is 40 ns and adder HW time is 10 ns, the reconfiguration time is 100 ns and one PRR is given, the execution time of the graph will be equal to 40 + 40 + 40 + 40 + 100 + 10 + 10 +10 +10 = 300 ns.

The reconfiguration time depends on the size of partial bitstream, which depends on the area selected on the chip. In the above example, the size of the PRR is decided by the maximum of (Mult, Add) area. The detailed discussion of the reconfiguration time is discussed in subsequent sections.

5.6.1. Coarse Level Graph Creation

Performing the reconfiguration at very low level of granularity i.e. at operator level, may lead to huge reconfiguration penalty may be very high as applications can have many such operations and FPGA resources cannot be exploited to its best. Hence, it is required that we mobilize from fine grain DFG to coarse grain granularity, where clusters of a size given can be created. Given the size of the PR region as number of LUTs of the total device, we need to create partitions of the design accordingly. For e.g., the Xilinx FPGA series Virter-5 xc5vfx70t-1ff1136 present in ML507 [5.1] board has 44800 slices LUTs, 48 DSP slices, 128 BRAM memories. Since any logic can be implemented on LUTs, other resources can be ignored. When as embedded system is developed around the FPGA based design, the space is returned as a percentage of the total area available. The complete system design encompasses processor, memory controller, bus controller and IP cores etc. After creating a complete system around the ML507 board we found that the PR region can be allocated as large as 40% of the space. This was concluded by creating 10 sample projects in which device was not able to place and route the design which consumed more than 40% of the chip.

Once the size of PRR is given, we need two more parameters for creating clusters of nodes called as coarse level clusters. We again focus on four operators ADD, SUB, MUL and DIV modules for the DFG clustering similar to operators used in isomorphic design flow. In order to estimate the size of the cluster we should first know the area consumed by each operator on real device and library created in Table 5.7 will be used again. The coarse level graph can be created in different ways depending on the design of the cost function. The easiest method is to assign levels using ASAP. Fig. 5.28 shows level wise clustering with area constraints.

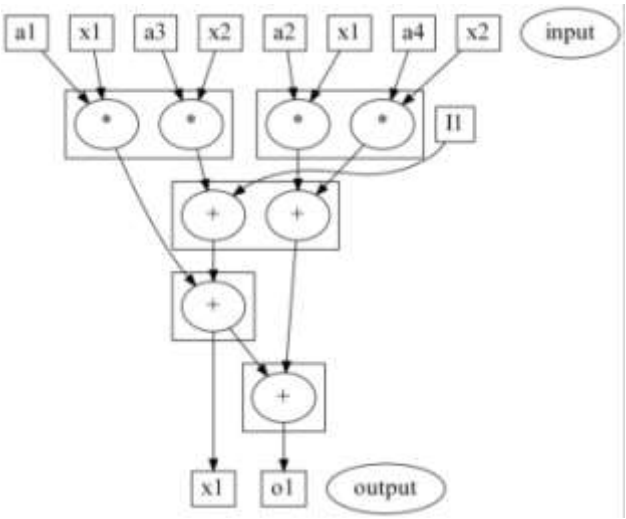


Figure 5.28: Level based clusters

The process of creating and executing partitions on time basis is known as temporal partitioning. In the next section we have used GA for the creation of clusters based on the partial reconfiguration design constraints. The effectiveness of GA was demonstrated in the chapter 4 and hence has been used.

5.7. Genetic Algorithm for RCS

In the chapter 3 we have used the GA for co-design process. The next requirement of our design flow requires the application of GA to RCS. Given a PRR region, we have to find the best partitions and schedule them for a minimum execution time. Given a partial reconfigurable region (PRR) with the resources (R) and a dataflow graph G with nodes N = $\{n_1,n_2,...n_n\}$, which are operators with a given area A = $\{ a_1,a_2,...a_n \}$ and delay D = $\{ t_1,t_2,...t_n \}$, partitioning is defined as the creation of disjoint clusters set C = $\{c_1,c_2,...c_n\}$, where $c_i \subseteq N$ satisfying the condition area(c_i) $\leq$ R. R is the maximum size of the defined area [1.7].

The above equation also imposes an ordering constraint on the execution model of the device which is defined as: a cluster c_i can only execute if all its predecessor nodes have executed. This condition defines the scheduling order of GA. The objective of the clustering algorithms is to find clusters which can fit into a given PR region. GA is the popular algorithm used in recent times and has been discussed in chapter 3. The *objfct* used in our algorithm is shown below; it includes *time, max_inputs, max_outputs, size and quality* [1.7]. In order to calculate the quality of the partitions we define connectivity of the graph,

a. Connectivity of a graph G = (V, E) is given is defined in Eq. 5.15 as:

$$con\ (G) = 2*|E|/(|V|^2 - |V|), \quad \ldots(5.13)$$

Where V is the number of nodes and E is the number of edges.

b. Quality of partitioning P = {P1,…, Pn} is given by Eq. 5.14 as:

$$\text{Average connectivity over P: } Q(P) = 1/n \sum_{i=1,\ldots,n} con(P_i) \quad \ldots(5.14)$$

The objective function is given below (Eq. 5.15) and combines all the constraints imposed by the algorithm. The partial reconfiguration design flow imposes many constraints (inputs/outputs pins are fixed, area are constant) on the design flow and was presented in literature survey [2.66]:

$$f\_value = \begin{cases} (k_1 * \max\_time) + (k_2 * (1/(\max\_inp - inp\_num))) \\ \quad + (k_3 * (1/(\max\_op - op\_num))) + \\ k_4 * (1/quality)) + \left(k_5 * \left(max_{size} - size_{partition}\right)\right) & if\ t > tr \\ c & if\ t < tr \end{cases} \quad \ldots(5.15)$$

Five parameters decide the value of the function and each is weighted by multiplying with a constant. The higher the value of a constant, higher will be the effect of the parameter. The detailed algorithm with modification of our requirement is given below:

Algorithm 4.4: Genetic Algorithm for Reconfigurable Systems

BEGIN

p = population size;

t_n = no. of nodes;

b_n = number of bits to represent a partition; // define a variable to define the partitions

n= total number of bits in a chromosome;

G = generate random population (0 or 1) of size = p x n;

REPEAT

1. *Evaluate FITNESS of each chromosome G by finding the following*
   Calculate time; Calculate cost; Calculate quality; Calculate objective function (OF);
2. *Evaluate PROBABILITY of each chromosome by,*
   Give max probability to min. OF;
3. *Calculate CUMULATIVE PROBABILTY (CP);*
4. *Evaluate Roulette Wheel for SELECTION of better chromosomes by,*
   g1= generate random no. (0 to 1) for size = p;
   Find nearest larger CP values for each g1;
   Generate new population;
5. *Do CROSSOVER between the pairs of parents*
   c = define crossover probability;

g2= generate random no (0 to 1) for size = p;

Select chromosome for g2 ≤ c;

Generate random position (0 to n) for each pair of selected chromosome;

Interchange bit patterns between pairs after their generated position;

Generate new population;

6. *Do MUTATION of the resulting offspring;*

m = define mutation probability;

g3 = generate random no. (0 to 1) for size= p;

generate g4(1 to n)

Select chromosome for g4 ≤ m;

Toggle bit pattern;

G = Generate new population;

UNTIL TERMINATION CONDITION SATISFIED

Schedule optimum OF value pattern;

END

The GA starts with a population which contains chromosome. The chromosome can be visualized containing n bits given by Eq. 5.16 as:

$$n = t_n * b_n \quad \ldots(5.16)$$

Where t_n is the total number of nodes present in a chromosome and b_n is the number of bits taken to represent the partition to which a node belongs. The value of b_n is computed by calculating the number of partitions required and the number of bits to represent this number. Number of partitions required is the maximum of: *minimum PRR size (min_PR_size), minimum PRR inputs (min_PR_inp) and minimum PRR outputs (min_PR_op).* These three parameters are required since the created partition should not violate any of these values as given by Eq. 5.17, Eq. 5.18 and Eq. 5.19.

$$n\_PR\_size = ceil\ (req\_size/max\_size) \quad \ldots(5.17)$$

$$min\_PR\_inp = ceil(inp\_num/max\_inp) \quad \ldots(5.18)$$

$$min\_PR\_op = ceil(op\_num/max\_op) \quad \ldots(5.19)$$

Where *req_size* is the amount of partition space required for implementing given function and *max_size* is the amount of space available on the FPGA. Similar is the case with *min_PR_inp and min_PR_op*. The maximum of these values is taken to be *min_PR* which represents the number of partitions to be made as given by Eq. 5.20.

$$min\_PR = mini(min\_PR\_size, min\_PR\_inp, min\_PR\_op) \quad \ldots(5.20)$$

Let us assume that an PRR has 10 inputs, 5 outputs and 200 units of area. Now as application has 100 inputs, 20 outputs and 600 units of area required. This gives min_PR_size = 3, min_PR_inp = 10, min_PR_op = 4, hence number of partitions required will be 3. The parameter b_n can be calculated from mini_PR with a ceiling function as given by Eq. 5.21.

$$b_n = ceil(log(\$min\_PR)/log(2)) \quad ...(5.21)$$

For generating the initial population, *initialize_population ()* function has been written that creates an initial random population for the next stages of the algorithms. After the population has been generated, the task of next stage (Fitness) is to find, the better chromosome for which an objective function is used. An objective function (Objfct) is defined to guide genetic algorithm to optimize a partition for minimizing some attributes for given constraints parameters. As a result, some form of metrics needs to be created. Some possible minimization attributes are power, delay, hardware, cost, silicon area etc.

Calculate_chromosome_f() function calculates the *f_value* that accumulates the effects of all the parameters like num_inputs, num_outputs, max_time, size, quality and validity of the order. The aim of the genetic algorithm would be to keep the chromosomes with minimum *f_value*. So, by increasing the *f_value* of a particular chromosome, the probability of it making to the next generation decreases. Since we need to minimize the *max_time* parameter, a value proportional to *max_time* is added to *f_value.* The proportionality constant is k_1.

The *num_inputs* are taken to be the difference between *maximum inputs* available in the partition space of the FPGA and the number of inputs present in a given partition. Similar is the case with *num_outputs*. Since this *num_inputs* should be minimized, a value proportional to *num_inputs* and *num_outputs* is added to *f_value* in case these are positive. Proportionality constants are taken to be *k2_p* and *k3_p* respectively. In cases where the *num_inputs* or *num_outputs* are negative, those particular chromosomes should be neglected as those are not possible to be placed on the partition space of the FPGA. So, the k2_n and k3_n which are the proportionality constants in cases when *num_inputs* and *num_outputs* are negative are taken to be very high.

Since the difference between the maximum partition space available on the FPGA and space that will be occupied a given partition should also be minimized, we add to *f_value* a value proportional to the difference. Proportionality constants are *k5_p* and *k5_n* for the positive and negative cases similar to *num_inputs* and *num_outputs.*

The quality of a chromosome indicates how many edges are being cut between the partitions of a chromosome. Lesser the number of edges cut, higher will be the quality. Since the

quality is to be maximized, a value inversely proportional to the quality is to be added to *f_value*. But since the change in the quality is so less, a value that is normally observed to be the average of the qualities of chromosomes of few iterations of best populations subtracts from the quality and then the inverse is taken. The proportionality constant here is taken to be *k4_p* and *k4_n* in the positive and negative cases respectively.

As shown by Eq. 4.21, the objective function calculation requires many constants. Table 5.12 shows the value of constants used in our objective function calculation. Higher the value of the constant, less will be the effect of it in the function, hence the values have been set in such a way that time gets the highest contribution.

Table 5.12: Constants used in genetic algorithm

Type	Name	Value
Time	$k1	10
Input	$k2_p	1
Input	$k2_n	100
Output	$k3_p	1
Output	$k3_n	100
Quality	$k4_p	0.001
Quality	$k4_n	10000
Size	$k5_p	0
Size	k5_n	100

Following functions were written for calculating the parameters for GA and are called by *Calculate_chromosome_f() :*

- *Critical_time()* function calculates the maximum time taken to evaluate a given output in the specified partition. This recursive function starts from the input node and adds up the time taken by each node till an ouput to the partition is encountered. Then, maximum value among those times is taken to be the critical time of the given output node.
- *Calculate_partition_time()* function calculates the critical time for all the available outputs and temp_outputs in the given partition by calling critical_time function.
- *Calculate_chromosome_time()* function calculates the maximum time taken by a chromosome by adding the times taken by partitions in that chromosomes which are calculated by calling calculate_partition_time function.
- *Calculate_partition_connectivity ()* function calculates the connectivity of a partition.
- *Calculate_chromosome_quality ()* function calculates the quality of partitioning of the chromosome.

The process of selection, crossover and mutation given in algorithm are implemented using Roulette wheel [2.27]. A function name *GA_generation ()* does this. The function

GA_generation calls the function *partition()* takes the initial population and creates the partitions. After the partitioning process, the order in which the partitions should run must be decided, a function named *depict_partitions()* checks the dependency order and outputs the partitions with a specific naming convention. Each partition is then converted to HDL by a function named *write_to_verilog()* in dataflow semantics. The order of complexity of all the functions is known in Table 5.13 where N is the number of nodes, G is the number of generations and P is the number of population.

Table 5.13: Order of complexity of functions in GA

Calculate_chromosome_f	$O(N^2)$	Partition	$O(N^2)$
Calculate_chromosome_quality	$O(N)$	Permute	$O(N^2)$
Calculate_chromosome_time	$O(N^2)$	Validate_order	$O(N)$
Calculate_partition_connectivity	$O(1)$	Write_to_dot	$O(N^2)$
Calculate_partition_time	$O(N)$	Write_to_verilog	$O(N^2)$
Critical_time	$O(N)$	Depict_partitions	$O(N^2)$
Find_valid_order	$O(N^2)$	Main_control	$O(N^2G)$
GA_generation	$O(N^2)$	Partition	$O(N^2)$
Initialize_population	$O(N^2)$	Permute	$O(N^2)$

Explanation for Overall time complexity:

N -> No. of nodes (Only input of algorithm)

G -> No. of generations (May need to be increased as the number of nodes is increased)

The critical operation leading to the mentioned overall time complexity is the partitioning of nodes. Since this is done for each generation, O(G) is the complexity. And for each generation, partition function needs to loop through number of partitions and through all the number of nodes for each partition, the complexity $O(N^2)$is used. Hence overall time complexity is: $O(N^2G)$.

5.8. Wrapper Design and Scheduler Design

The next step in design flow requires the partitions to be scheduled in precedence order in SW. The GA discussed in previous section returns the order of execution of partitions which can be used in designing a static scheduler. The function *Find_valid_order* checks the dependency of the nodes and returns the cluster as C1, C2 and so on. These clusters are wrapped around in a common interface. The obtained partitions are supposed to run in one PRR, which requires the interface should be the same, since this a constraint in the PR design flow as discussed in chapter 1. This is a restriction on the design in terms of inputs and outputs channels cannot vary. The wrapper design requires that all the partitions and their ports be scanned order and type.

Suppose that two clusters given below are supposed to be designed for one partition.

module prr1(a,b,c) ;

input [3:0] a, [3:0] b;

output [7:0] c;

module prr1(x, y, z);

input [7:0] x, [7:0] y;

output [15:0] z;

The created wrapper is

module prr(a,b,c,x,y,z);

This wrapper is then instantiated as a component in the bus wrapper file and can be tested. Although this technique uses extra ports for each partition, but is necessary, since the PR tools searches the nets and if the same name net is not found it gives an error. When creating a partition of size around 20%, the region selection becomes crucial. In many cases the design is not routable and gives an error. The constraints discussed in section 2.5.2 (Restrictions in design flow) about the number of clock regions may fail as number of clocks available are restricted. In this situation the resulting partitions may be of no use and require modification in the HDL code of the design. It was found that left side upper region of the Virtex-5 gave better results in terms of PR region. This may be due to other regions are better supported for specific controllers like DDR memory.

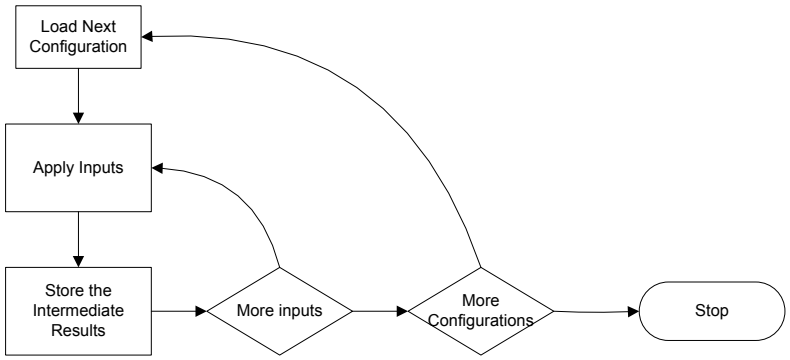


Figure 5.29: Scheduler design in SDK

After the clusters are wrapped, we need to run them in correct order. This step demands a robust scheduler, which configures the memory by picking up partial bit files from external memory. The Fig. 5.29 shows the flow chart for the scheduler designs.

The scheduler designed has following steps:

- A new configuration is loaded into the FPGA
- For this configuration inputs are applied

- The results of the configuration are saved in temporary registers.
- If the same existing configuration can be used, then new inputs are applied
- If more configurations exist then the process is repeated.

The code given below shows the snippet of the scheduler used in Xilinx SDK:

```
xil_printf("\r\n Performing Reconfiguration for b1\r\n ");
 XHwIcap_FLASH2Icap (XPAR_FLASH_MEM0_BASEADDR+0x01f20000);
 xil_printf("\r\n done b1\r\n ");
TEST_mWriteReg (baseaddr, TEST_SLV_REG24_OFFSET, input [16]);
xil_printf("\r\n----------\r\n Writing Data Set: %d\r\n",i);
...
xil_printf("\r\n b1_output[1] = %X",b1_output[1]);
TEST_mWriteReg (baseaddr, TEST_SLV_REG8_OFFSET, input[8]);
...
b1_output [2] = TEST_mReadReg (baseaddr, TEST_SLV_REG25_OFFSET);
xil_printf("\r\n b1_output[2] = %X",b1_output[2]);
xil_printf("\r\n Performing Reconfiguration for b2\r\n ");
XHwIcap_FLASH2Icap(XPAR_FLASH_MEM0_BASEADDR+0x01e00000);
...
```

In the above code, XHwICAP_FLASH function is used to load partial bits files from flash memory using the ICAP controller.

5.9. Reconfiguration Time Analysis

The design space exploration starts from SW implementation and can also look for HW, SW-HW and HW-HW variations depending on the performance and area constraints. Our proposed HW-HW approach should be compared in terms of design flow complexity, performance gain and running time. The design flow complexity involved in this process is the creation of PlanAhead project, which takes specialized knowledge and understanding. For performance comparison, the design flow adopted here should be compared in terms of timing with respect to

i. Total SW implementation of design on PowerPC processor in Xilinx SDK.

ii. Total HW implementation of the design as IP core interfaced to the PowerPC processor in Xilinx XPS.

iii. Proposed PR design flow (HW-HW): using compact flash (CF) card, double data rate (DDR) and Flash memory, PowerPC processor and PlanAhead SW.

Since the proposed design flow is new hence arises the need for a time estimation model, which can compare the obtained results and theoretical results. Estimated time for reconfiguration of the FPGA is resolved using the equation [2.73]:

$$Config\_time[s] = Bitsize /(Cclk* Buswidth) \quad ...(5.22)$$

Eq. 5.22 takes Bitsize, input clock and bus width to calculate the configuration time, but does not includes inputs, outputs and execution time required to run the IP core. The Eq. 5.23 which is below, includes execution time, input, output and reconfiguration time can be formulated as

$$t_{tot} = n \cdot t_{conf} + \sum_{i=1}^{n} \left(t_i(i) + t_{exe}(i) + t_o(i) \right) \quad \ldots(5.23)$$

Where n is the number of time the configuration is done, t_{conf} as the time required to configure the FPGA, t_i as the time to apply the input to the configuration, t_{exe} as the time required to execute the inputs and t_o as the time required to get the output in registers. Eq. 5.23 describes the total execution time of an application to run using clustering approach. For our HW-HW model, it is used for calculating the total time required for execution. Since we are using only one PR region, the t_{conf} will remain constant as it is dependent only on PR region size not on the bitstream size. The smaller the value of n, lower will be the reconfiguration overhead. This equation can be applied when the reconfiguration time for benchmark programs are measured on HW using a timer. The values used in the equation were measured using the hardware *xps* timer, which is a core available in repository and has been described in chapter 3.

After creating the a sample design in PR flow, it was found that partial bitstream for 40% area consumption is of 358 KB for and a simple adder file was 128 KB. Hence it was concluded that the size of the partial bit files is of the order of KBs. The maximum throughput of the PR design flow reported as discussed in literature survey [2.67] is 400 Mbps using DMA controller and double clocking. This concludes that a file of 100 KB will take 250 microseconds for reconfiguration. This overhead in writing a new bitstream is of the order of microseconds, which is large as compared to applications which are completely their execution in microseconds. Hence this technique can only be useful if either this time is diminished or application can tolerate this much overhead for saving significant area. Hence as a thumb rule we have assumed a throughput of 400 Mbps for reconfiguration time calculations in subsequent sections.

5.10. Parameters and Results for Genetic Algorithm

The constants required for the algorithm to run are the maximum size of the PRR in terms of LUTs, maximum inputs, maximum outputs, population size and number of generations. The parameters used for the GA execution are shown in Table 5.14 for cosine1 and cosine2.

Table 5.14: PR and GA parameters

PR Parameters		GA Parameters	
Max_size	2200 LUTs	Population size	30
Max_inputs	30	Num of generations	300
Max_output	25	Mutation probability	0.1
		Crossover probability	0.7

GA has been applied to Express benchmark [2.38] available as dataflow graph for cosine1 and cosine2 functions. The dataflow graphs of the two programs are shown in Fig. 5.30 and Fig. 5.31. Table 5.15 shows the number of nodes, number of edges, inputs, outputs and resource consumed by the two benchmarks programs. The two differ in the number of inputs.

Table 5.15: Express benchmark programs used for testing GA

Design	Kind	Nodes	Edges	Inputs	Outputs	Resources Used	
						LUTs	DSP
Cosine1	Express	66	76	16	8	17021	48
Cosine2	Express	82	91	32	8	17021	48

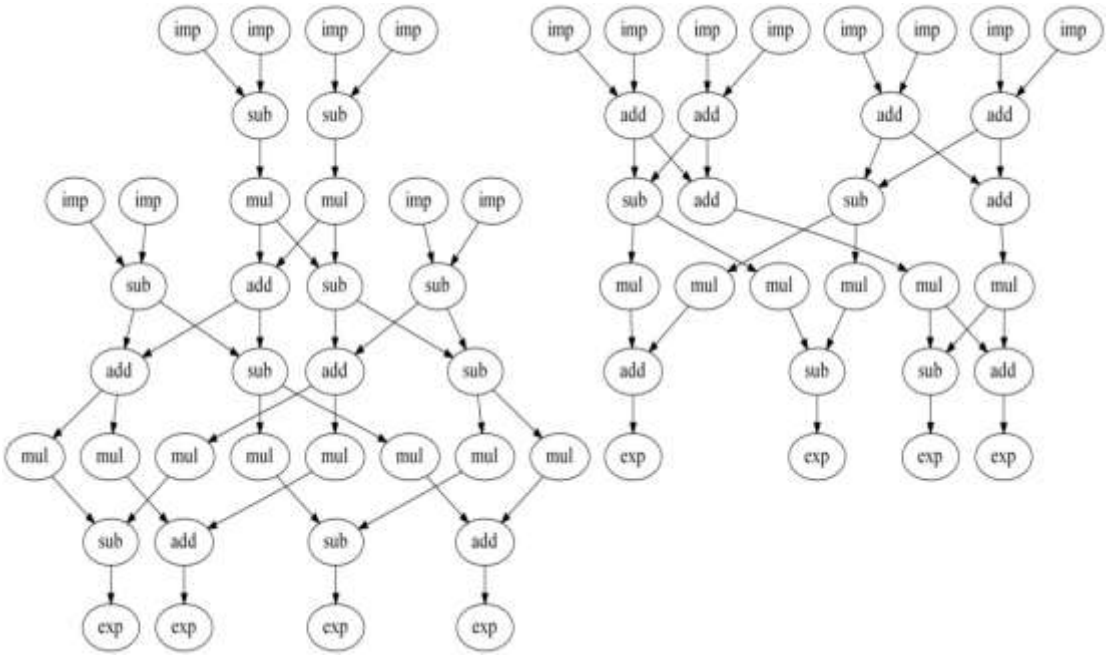


Figure 5.30: DFG of Cosine 1

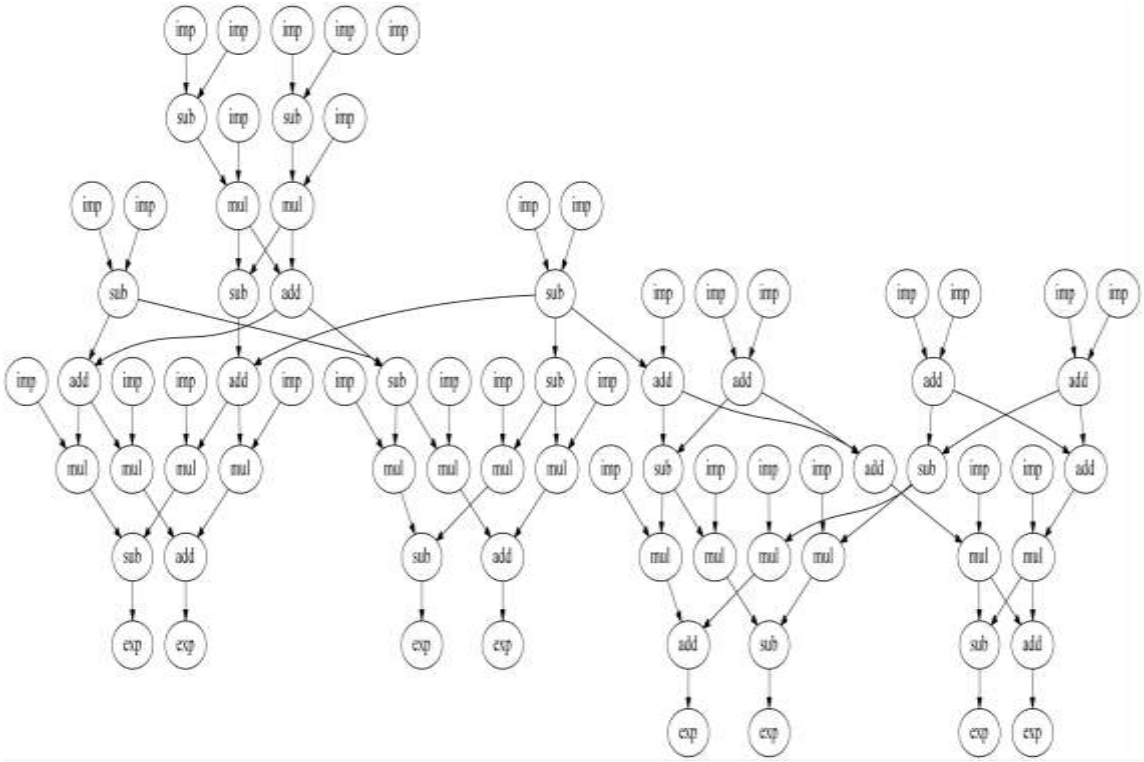


Figure 5.31: DFG of Cosine 2

The number of times the iterations of the algorithm is executed decides the running time of the GA.

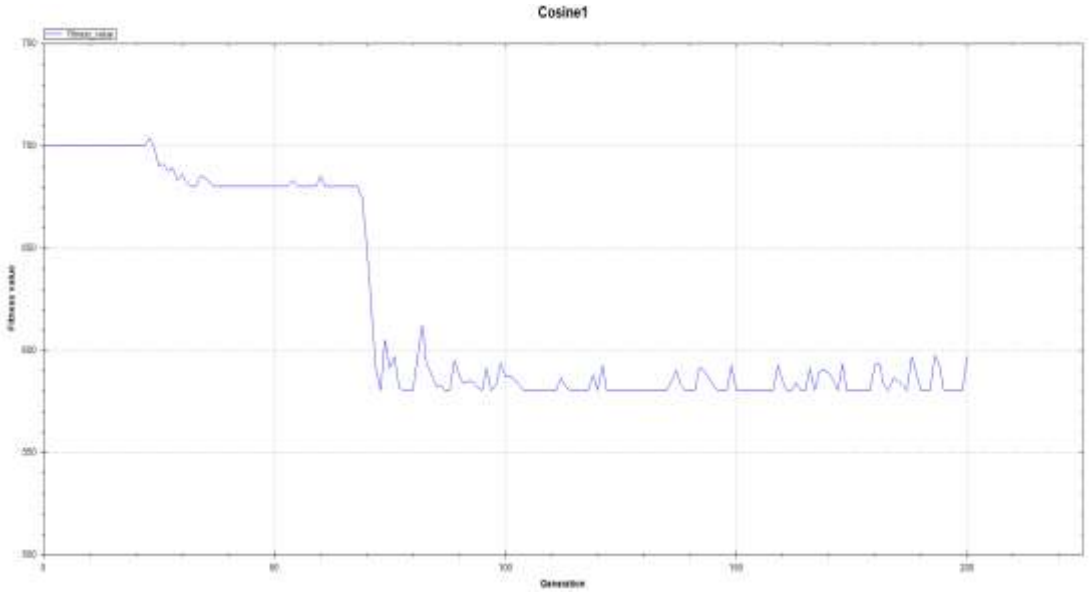


Figure 5.32: Fitness vs. generations

Fig. 5.32 shows the plot of fitness value against its generation for cosine 1. Algorithm has the initial fitness value to be around 700 which then reached the global optimum solution (having fitness value as 580 in this case) by traversing through one local optimum solution (fitness value as 680). This being a simpler graph, could be seen to have converged to final optimal solution in around 70 generations.

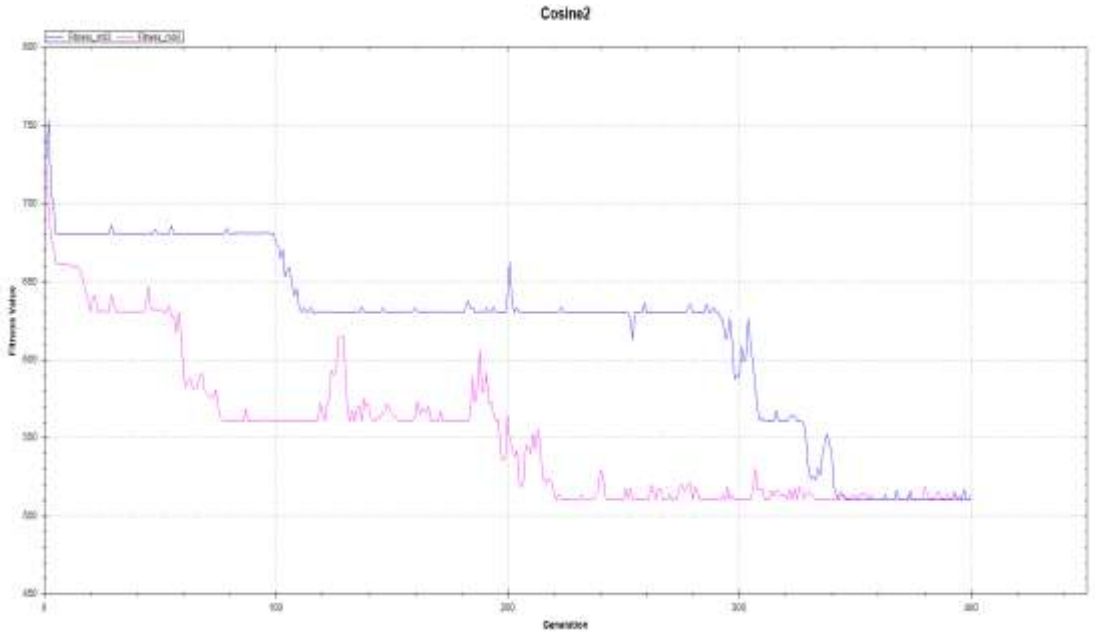


Figure 5.33: Fitness value vs. generation for mutation value as 0.2(Blue) and 0.4(Pink)

Fig. 5.33 shows plots of fitness value against its generation for mutation value as 0.2(Blue) and 0.4(Pink). Rate of convergence could be seen to be different for both the plots as expected. With mutation rate as 0.4, optimal solution is reached in around 225 generations while it took around 350 generations for the case using 0.2 as mutation rate. The algorithm with higher mutation value explores more of new solutions and hence reaches the optimal solution (which is approximately at fitness value of 510 in this case) faster than the algorithm with lower mutation value. On the other hand, too high mutation value leads to losing of the essence of genetic algorithm making it a random search algorithm. Spikes on the graph are closely proportional to exploration rate and hence to the mutation rate. Algorithm has the initial fitness value to be around 730 which then reached the global optimum value of 510 by traversing through the local optimal values – 680, 630, 560.

In the next section we have developed a task graph generator for examining the strength and running time of GA.

5.11. Random Task Graph Generation

Validation of the robustness, efficiency of allocation and scheduling heuristics in large scale parallel and distributed systems is usually done using synthetic randomly generated workloads, represented by task graphs. We present a modular approach to the problem of generating random directed acyclic graphs (DAGs), called as modular random task graph generator (MRTG), making it very flexible for the researchers to use it. Modular based approach provides a great advantage for future development as more modules can be added without disturbing the existing stable software. The task nodes are placed randomly using a layer-by-layer approach and then connected randomly. Paramount importance has been given to user-controlled randomness in developing this algorithm. The MRTG can generate task sets containing several different types of task graphs like rooted trees, isomorphic graphs and random graphs with same node placement but different connections, with the flexibility to dictate the type of graph generated. We also present a comparison of MRTG with existing solutions to the random task graph generation problem.

5.11.1. Random Graph Generators

Research in real-time embedded systems, operating systems and hardware software co-design, as well as in more general allocation and scheduling fields, is hampered by the lack of a common base of benchmarks. In general, any example used in allocation and scheduling research consists of a task set and a database of processors plus several communication resources. A task set is a collection of task graphs, each of which is a directed acyclic graph (DAG) of communicating nodes. Generation of sample task sets is often a requirement when comparing allocation or scheduling methods with each other. The existing solutions are of limited relevance in today's scheduling problems in parallel, distributed systems and fields like hardware software co-design, which require a clear definition of the size of the critical path and also cater to the possibility of defining different types of task nodes with independent parameters. MRTG accomplishes these and also gives researchers an opportunity to clearly define the number of inputs to each type of task node, which is necessary in today's computer science scheduling problems which require that all inputs arrive for the task node to give an output.

MRTG is highly valuable for simulation of scheduling problem on many core processors in order to choose how to divide the work that must be done among such large number of processing cores. It is of particular importance to researchers working in areas like reconfigurable computing and System-on-Chip. Because of its layer-by-layer [5.2] approach

as researchers can define graphs according to the requirements. As MRTG has been created with computer hardware scheduling problems in mind, the constraints are defined in terms of the silicon area consumed in executing each task and the delay across that task node. But, these definitions are flexible and we can interpret these constraints in the way that they are relevant to any research. For easy analysis, MRTG currently give output graphs in two formats a text file with the list of node placement, an array of ordered pairs describing the connections and in GraphViz's dot format.

We now present a thorough comparison of MRTG with popular existing random task graph generators. Task Graphs for Free (TGFF) [5.3] is one of the oldest and most popular algorithms for generating user-controllable, general-purpose, pseudorandom task graphs. The original TGFF algorithm iteratively adds nodes to construct a graph using limits on the maximum in and out degrees of each node, this reduces the randomness of the task graph generated, making it a pseudo-random task graph generator [5.4]. Fig. 5.34 shows the sample graph generated using TGFF.

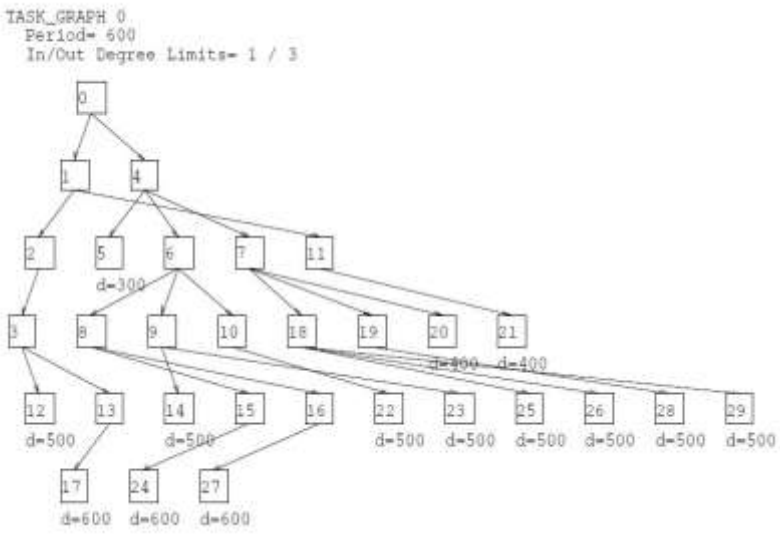


Figure 5.34: A graph generated using TGFF

A more recent version provides an option to generate also series-parallel DAGs. MRTG on the other hand, keeps the node placement, connection steps separate and adds randomness at every level making it a truly random task graph generator. This method of iteratively adding nodes, leads to generation of only rooted task graphs by TGFF, while MRTG can produce graphs with any number of input nodes, including rooted trees if the number of input nodes specified by the user is one.

MRTG creates a graph with the exact number of nodes as specified by the user, while TGFF takes the average and multiplier from the user, for the lower bound on the number of nodes in a graph and creates a graph with number of nodes that are randomly greater than this lower bound. Another major difference between TGFF and MRTG is that, TGFF uses a concept of depth to decide the communication delay of the graph generated. In MRTG we have a concept of levels, where we assume that any task in the next level does not start unless all the tasks in the previous level are completed. This is of particular importance to researchers in fields like reconfigurable computing, as now they can define different levels in random task scheduling to take into account reconfiguration of the hardware between levels.

TGFF is flexible with the number of inputs of each node and the user specifies the degree of inputs. The TGFF algorithm results in a graph that has nodes in which the number of inputs connected is any number less than the degree specified, while MRTG takes the exact number of inputs required by each task from the user and ensures that they are connected.

Even so, MRTG and TGFF are similar with respect to the flexibility of the outputs of a node as both take the maximum number of outputs and connect any number below this. Also, in TGFF only one type of node is specified, while in MRTG, multiple types of nodes can be specified with their own individual constraints. The two other generators commonly available are Graph Generation for Scheduling Simulations (GGEN) [5.5] and Random Task Resource Graphs (RTRG). GGEN provides a very thorough coverage of the different task scheduling algorithms developed over the years. It uses the libraries of the existing solutions and provides the developer with a single tool to exploit them. But, in doing so it gets limited by the shortcomings of those algorithms. It becomes a one stop tool for developers, but has limited additions of its own. Different types of nodes with their own individual parameters and constraints can be specified in MRTG, while this is not an option in GGEN. One aspect where GGEN and MRTG are similar is that both generate an output in Graphviz's dot language. RTRG is a simple and effective tool, but provides very limited flexibility to the developer. It defines resources used by a task node, which is similar to the concept of area used in MRTG. It offers different types of nodes but the constraints of each node have to be defined individually, which presents a problem when the number of nodes increases, while the node type definitions in MRTG are grouped making it easier for the user. The output file generated by RTRG is in .rtg format, which is difficult to analyze, while MRTG generates a dot file as output along with a text file showing the connections and node placement in matrix form.

MRTG differs from all existing solutions in the respect that it is divided into self-contained modules and changes made in one module don't affect the functioning of another. This

modular nature makes the code far more reusable than a conventional monolithic design. Future improvements are easier to make, as additional modules can be added without disturbing the functionality of the original stable software. It also makes the program very flexible to use, as now researchers can choose to run only those modules that they require and also change the order of execution of modules to suit their needs. We now describe the working of MRTG, which is currently divided into four modules assignLevels, connectNodes, isomorphize and plotGraph.

5.11.2. Algorithmic Design of MRTG

MRTG primarily generates a specified number of random task graphs, where the graph nodes are tasks and the graph edges depict the communication between tasks. In the algorithm, we first decide the total number of task nodes needed, the number of levels in which to place them, number of input nodes, the maximum area that each level can accommodate and the total delay constraint. Along with this, we can specify different types of task nodes here, by giving the number of inputs, the fan-out, i.e. degree of the output, area consumed, delay and the total number of nodes of that type. Here, we can also give the number of isomorphic graphs required. Rooted graphs can also be generated by specifying the number of input nodes as one. Given below is the sample input specification file:

isomorphicCount
numberOflevel areaPerLevel MaxDelay
TypesOfInputNodes CountOfTotalInputNodes
CountOfType1 AreaofType1 DelayOfType1 OutputsOfType1 SymbolOfType1
CountOfType2 AreaofType2 DelayOfType2 OutputsOfType2 SymbolOfType2
CountOfType3 AreaofType3 DelayOfType3 OutputsOfType3 SymbolOfType3
....
TypesOfTaskNodes CountOfTotalTaskNodes
CountOfType1 AreaofType1 DelayOfType1 InputsofType1 OutputsOfType1 SymbolOfType1
CountOfType2 AreaofType2 DelayOfType2 InputsofType2 OutputsOfType2 SymbolOfType2
CountOfType3 AreaofType3 DelayOfType3 InputsofType3 OutputsOfType3 SymbolOfType3
....

The seed for randomness in MRTG, which decides the structure and other aspects of the generated task graphs, is the current system time making it highly unlikely for any two graphs generated at different times to be similar. But if similar task graphs are required, we can specify a user-defined seed and share it with another researcher. We have defined four rules in this algorithm:

a. Every type of node has only specified number of inputs, but the output can go to any number of nodes less than the fan-out as input.
b. The output of every node, except the ones the bottom level, is connected to at least one input.
c. All connections are downward directional, so the output of a lower node cannot connect to an input of an upper node and any node's output can go into the input of a node from any level below its own.
d. Tasks in a level start only after all the tasks in the previous level are completed.

5.11.2.1 Module 1: assignLevels

This module develops on the layer by layer method proposed by Tobita and Kasahara [5.6]. Here, we randomly place node in different levels, without violating the maximum area that each level can accommodate and the total delay constraint. This module only decides the node placement and makes no connections between the nodes. We first create a list, which stores the node placement information. Then, we repeatedly select a random level and put a randomly selected node in it, while ensuring the maximum area in that level is not exceeded. This process continues till all nodes are placed. Lastly, we calculate the total delay of this particular node placement by adding the maximum delay at each level and check against the delay constraint. If violated, the process repeats from the start. Although very simple, this method is very useful in practice because by limiting the number of levels we can limit the size of the critical path.

Algorithm 5.5: Module 1 - assignlevels

1. Name:assignLevels
2. Function: Randomly place node in different levels, without violating constraints
3. Input: numberOfLevels = The total number of levels in the graph
4. areaPerLevel = The maximum area that each level can accommodate
5. delayLimit = The delay constraint
6. inputNodes = Array containing all the input nodes
7. taskNodes = Array containing all the task nodes
8. nodeList = List defining node placement, with the first index holding the level number
9. Output: Passed by reference
10. Boolean value returned
11. Algorithm:
12. / assume nodes sorted already by type */*
13. Define an array freeArea[] and put the value of areaPerLevel for each level.
14. Insert all input nodes at level 0 of nodeList and subtract area of each from freeArea[0].
15. iffreeArea[0] is less than 0

```
16.             then say "Input nodes occupy too much area" and assert false
17.     endif
18.19.  /* fulfill area constraints */
20.     for i=0 to taskNodes.size()
21.             Initialize chosenLevel to -1
22.             Initialize numIterations to 0
23.             do
24.                     set chosenLevel as 1 + modulus of random number with numberOfLevels
25.                     ifnumIterations is 1e6
26.                             then say "Unable to find suitable level in time" and return false
27.                     endif
28.             whilefreeArea[chosenLevel] is less than area at taskNodes[i]
29.             enddowhile
30.             assert false if chosenLevel is equal to -1
31.             Push back the node at taskNodes[i] into nodeList[chosenLevel].
32.             Subtract the area of taskNodes[i] from freeArea[chosenLevel].
33.     endfor
34.35.  /* fulfill delay constraints */
36.     Initialize totalDelay to 0
37.     for i=0 to numberOfLevels
38.             Initialize maxDelay to 0
39.             for j=0 to size of nodeList[numberOfLevels]
40.                     setmaxDelay to maximum of maxDelay and delay at nodeList[numberOfLevels][j]
41.             endfor
42.             Add maxDelay to totalDelay
43.     endfor
44.     iftotalDelay is greater than delayLimit
45.             then return false
46.     else
47.     return true
48.     endif
```

5.11.2.2. Module 2: connectNodes

Here, we take the node placement information from the previous level and randomly make downward directional connections and store them in an array of ordered pairs, while satisfying the rules of the algorithm. We start moving up from the second level from the bottom and randomly connect each output only once to an input below. After all the outputs have been connected once, we start moving down from the level below the input level and randomly connect all unconnected inputs to an output above, while ensuring that the fan-out is not violated. A totally random task graph, subject to input constraints, is generated at the end of this module. As a fail-safe, if the algorithm gets stuck at any point and is not able to place the nodes or make connections, it will automatically show an error after trying for a hundred thousand times. A sample of rooted graph generated from MRTG is shownin Fig. 5.35.

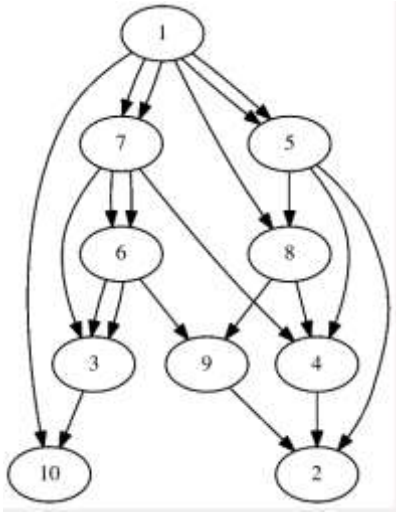


Figure 5.35: A rooted graph generated using MRTG

Algorithm 5.6: Module 2- connectnodes

1. Name: connectNodes

2. Function: Make random downward directed connections subject to number of inputs and fanOut of outputs of each node

3. Input: nodeList = List defining node placement, with the first index holding the level number

4. connections = Array of ordered pairs defining connections between nodes

5. Output: Passed by reference

6. Boolean value returned

7. Algorithm:

8. Clear any previous connections.

9./ ensure every input node is utilized atleast once:: bottom to top*/*

10. Initialize array of ordered pairsfreeInput

11.for i=0 to size of nodeList[numberOfLevels]

```
12.  for j=0 to number of inputPins of nodeList[numberOfLevels][i]
13.       Push back (numberOfLevels,i) into freeInput
14.  endfor
15.endfor
16.for level=numberOfLevels-1 to 0
17.  Randomly shuffle the array freeInput
18.  for i=0 to size of nodeList[level]
19.       Initialize a node id to nodeList[level][i]
20.       if size of freeInput is 0
21.            then return false
22.       endif
23.       Initialize choice to freeInput.size()-1.
24.       Initialize newLevel to the first value in the pair freeInput[choice]
25.       Initialize indexInNewLevel to the second value in the pair freeInput[cho.ice]26
26.       Initialize node newId to nodeList[newLevel][indexInNewLevel]
27.       Push back (id,newId) into connections
28.       Increment the value of connectedOutputs of nodeList[level][i]
29.       Increment the value of usedInputPins of nodeList[newlevel][indexInNewLevel]
30.       Pop back freeInput
31.  endfor
32.  for i=0 to size of nodeList[level]
33.       for j=0 to number of inputPins of nodeList[level][i]
34.            Push back (level,i) into freeInput
35.       endfor
36.  endfor
37.endfor
38.Initialize array of ordered pairsfreeOutput
39.for i=0 to size of nodeList[0]
40.  for j = connectedOutputs of nodeList[0][i] to fanOut of nodeList[0][i]
41.       Push back (0,i) into freeOutput
42.  endfor
43.endfor
44.Initialize highestLevelInFreeOutput to 0
45.Sort the elements in freeInput
46.
47./* now process all the free inputs top to bottom*/
48.for i=0 to size of freeInput
49.  Initialize level to the first value of the pair freeInput[i]
50.  Initialize indexInLevel to second value of the pair freeInput[i]
51.  Initialize node id to nodeList[level][indexInLevel]
52.  while level - highestLevelInFreeOutput is greater than 1
53.       Increment highestLevelInFreeOutput
```

54. if level is equal to highestLevelInFreeOutput

55. then return false

56. endif

57. for ii=0 to size of nodeList[highestLevelInFreeOutput]

58. for j= number of connectedOutputs of nodeList[highestLevelInFreeOutput][ii] to fanOut of nodeList[highestLevelInFreeOutput][ii]

59. Push back (highestLevelInFreeOutput,ii) into freeOutput

60. endfor

61. endfor

62. Randomly shuffle the array freeOutput

63. endwhile

64. if size of freeOutput is 0

65. then say "Insufficient outout pins" and return false

66. endif

67. Initialize choice to size of freeOutput -1

68. Initialize newlevel to the first value in the pair freeOutput[choice]

69. Initialize indexInNewLevel to the second value in the pair freeOutput[choice]

70. Pop back freeOutput

71. Initialize node newId to nodeList[newlevel][indexInNewLevel]

72. Push back (newId,id) into connections

73. Increment the usedInputPins of nodeList[level][indexInLevel]

74. Increment the connectedOutputs of nodeList[newLevel][indexInNewLevel]

75.endfor

76.Return true

5.11.2.3. Module 3: isomorphize

A graph G is isomorphic to a graph H if there exists a one-to-one function, called an isomorphism, from V(G) (the vertex set of G) onto V(H) such that (u_1,v_1) is an element of E(G) (the edge set of G) if and only if (u_2,v_2) is an element of H [2.43]. In simpler terms, two graphs are isomorphic when the vertices of one can be re labeled to match the vertices of the other in a way that preserves adjacency. This module is used to generate graphs that are isomorphic to the one generated above. We randomly select a type of node and swap the identification numbers of any two nodes of that type. The number of times this process is repeated for each isomorphic graph is also random.

Algorithm 5.7: module : assignlevels

---

*1. **Name: isomorphize***

*2. **Function:** Generate isomorphic graphs*

*3. **Input:**typeList = List defining the type of node, with the first index holding type and the second one holding the node information.*

*4. **Output:** Passed by reference.*

*5. **Algorithm:***

6. for i=0 to size of typeList

7. while generated random number %10 is not 0

8. int u = modulus of randomly generated number with size of typeList[i];

9. int v = modulus of another randomly generated number with size of typeList[i];

10. define Node n1 as typeList[i][u] and Node n2 as typeList[i][v];

11. swap the ID of n1 and n2 by reference;

12. endwhile

13. endfor

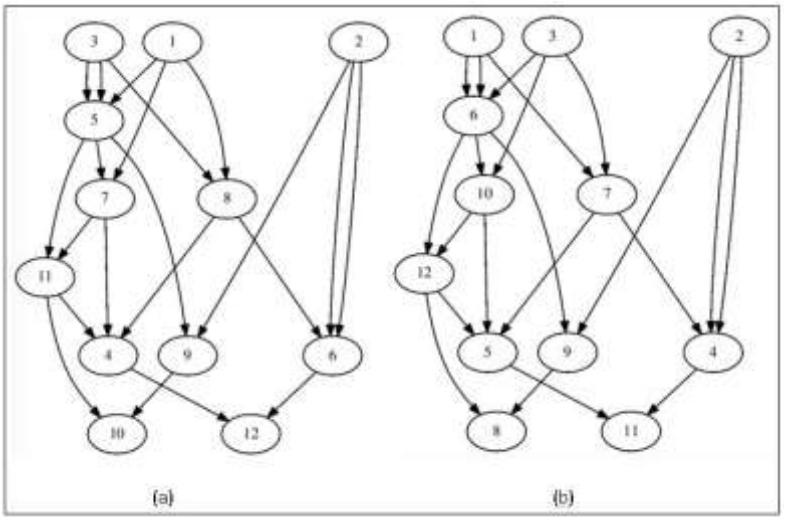


Figure 5.36: Two isomorphic graphs generated using MRTG

A sample of two isomorphic graphs generated using MRTG is shown in Fig. 5.36.

5.11.2.4. Module 4: plotGraph

The DOT language provides syntax for describing graphs, edges, nodes and the properties associated with the graph components in simple text format. We have chosen Graphviz's DOT language as the default format for graph representation for MRTG, to make it compatible with most of the available tools for graph analysis. In this module, we create graphs in DOT file format from the list of nodes and array of connections created above. MRTG being modular gives a lot of flexibility and control to the researcher. We can run *assignLevels* module once and c*onnectNodesmodule* multiple times to generate similar graphs that have the same node

place but different connections between the nodes. Rooted trees can be generated by specifying only one input node.

Being modular, MRTG can have future additions in the form of modules, which can be added without disturbing the original stable software. We plan to make it open source so that researchers who really need it, can develop modules they need and add them to the project so the whole community can use them. We plan to develop a module to add weights to the connections too. This will be very useful for researchers who need to do scheduling while taking into account the communication delay and resource expenditure. After that we also plan to add a concept of depth, as proposed in TGFF. MRTG provides a modular approach for generating user-controlled, truly random task graphs that find relevance in simulating today's scheduling problems in parallel, distributed systems and fields like hardware software co-design**.** This modular nature makes the program code far more reusable than a conventional monolithic design. It also makes the program very flexible to use, as now researchers can choose to run only those modules that they require and also change the order of execution of modules to suit their needs. The layer-by-layer approach followed in MRTG, with the ability to define different types of nodes with their individual parameters separates it from existing available solutions and makes it highly valuable for researchers working in areas like reconfigurable computing, System on Chip and for scheduling simulation in the problem of many core processors, to choose how to spread the work among such large number of processing cores. A sample of two operator based graph generated using MRTG is shown in Fig. 5.37.

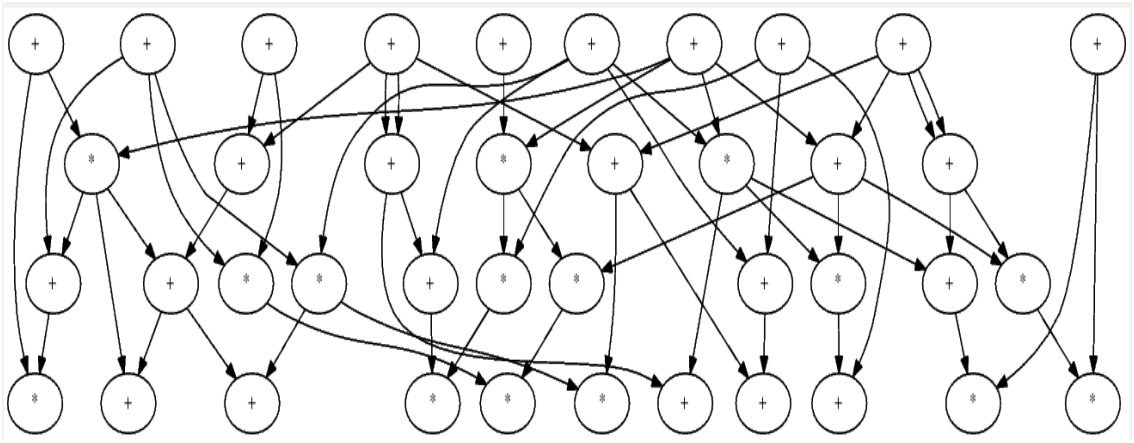


Figure 5.37: Nodes with operators generated using MRTG

The sample program used in the analysis of the GA algorithm contains nodes up to 100. In real practical applications such nodes can go up to thousands hence arises the need for handling large number of nodes. Results of GA applied to MRTG graphs, for number of

iterations = 10, Population size = 30 on a desktop machine having i5-2400 CPU @ 3.10GHz 3.10 GHz are shown in Table 5.16.

Table 5.16: Number of nodes vs. time taken

Number of nodes	Time Taken(minutes)
100	4.32
250	22.70
500	98.43

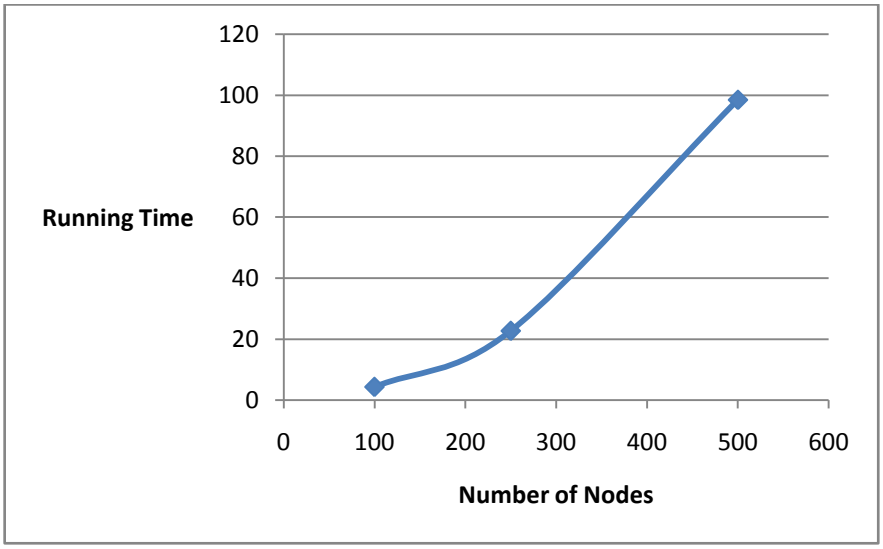


Figure 5.38: Number of Nodes vs. time taken

Figure 5.38 shows a nearly linear behavior of the genetic algorithm for nodes ranging from 100 to 500.

5.12. Results and Discussion: Comparing ISO and GA Approaches

This section focuses on a results based on simulation and experiments done to justify the frameworks proposed in chapter 3 and 4. Firstly the results of the GA are discussed and then a DCT design has been taken for experimental verification and comparison.

The results of GA were further applied to the four programs and are tabulated in Table 5.17. The partial reconfiguration overhead is neglected in the simulation since the order of time was reconfiguration is microseconds as discussed in literature survey [2.61]. The table shows the PRR defined and the corresponding partitions and execution time obtained from GA execution.

Table 5.17: GA applied to four benchmarks

Design (1)	Nodes /edges (2)	Inputs /outputs (3)	Resources Used (4)	GA Running time Seconds (5)	PRR Area Given (6)	No. of Partitons (7)	Execution Time (8)
Cosine series(float)	15/31	5/1	8302	8	5000	2	276.5
Exponent (float)	34/69	8/1	19050	8	5000	4	497.9
Matrix Multi**(3x3-Integer)**	45/99	18/9	4626	4	2500	2	75.3
Sine series(float)	18/37	4/1	9527	2	5000	2	284.5

The results in Table 5.17 are calculated in a python script and hence are simulation based only.

- Column 1 : name of the program
- Column 2 : number of nodes and edges
- Column 3 : number of inputs and outputs
- Column 4 : the resource consumed by the program
- Column 5 : runing time of GA
- Column 6 : partial area constraints given
- Columns 7 : Number of partitions returnedby GA
- Columns 8 : Execution time of program

Now that two different approaches have been discussed (GA and ISO), the Fig. 5.39 shows an algorithm to decide which one to chose, given a dataflow specification. This method helps the designer to follow a path based on requirement.

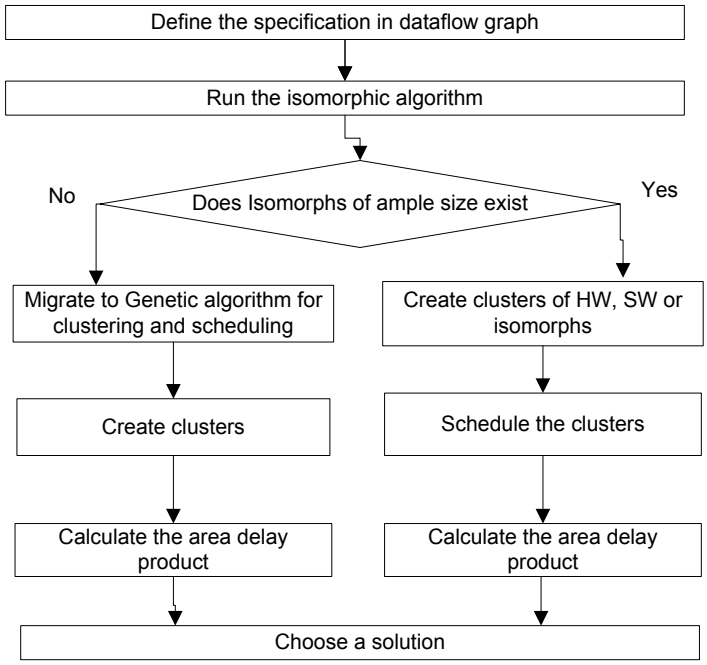


Figure 5.39: Flowchart for comparing the ISO and GA approach

Table 5.18: Comparison of simulations time

	Design	Area Delay Product GAx(1000)	Area Delay Product ISOx(1000)
1	Cosine series	1382.5	1442.05
2	Exponent Series	2489.5	3453.76
3	Sine series	1422.5	1654.83
4	Matrix	183.75	126.28

The comparison of GA and ISO is shown in Table 5.18. In case 1, 2 and 3 the performance of GA is better as compared to ISO approach. This is because of the same area used in the GA design. If we add the reconfiguration overhead in the GA design flow, the performance of ISO flow will be much better. Hence we can say that ISO is a more feasible flow for efficient implementation. If the reconfiguration overhead can be neglected (like concurrently loading of bitstream) then GA is much better.

5.13. DCT Case Study for HW Isomorphic Design flow Based on Experimental Work

Discrete cosine transform is one of the common applications that has been migrated to HW over the last two decades. This is due to the fact that it involves computation and processing. We now present a basic introduction to DCT:

5.13.1. Implementations of Discrete Cosine Transforms

DCT is a basic coding method to transform the image from the spatial domain to the frequency domain and works by separating the images into regions of differing frequencies. The 1-D DCT of *N* real numbers *x(n)*, *n* = 0, ..., *N*-1, is the list of length *N* given by [5.7] as given by Eq. 5.24:

$$X(k) = \sqrt{\frac{2}{N}} C(k) \sum_{n=0}^{N-1} x(n) \cos\left(\frac{(2n+1)k\pi}{2N}\right) \qquad k = 0, \ldots N-1 \qquad \ldots(5.24)$$

The list *x(n*) can be recovered from its transform by applying the inverse cosine transform (IDCT) as defined Eq. 5.25:

$$x(n) = \sqrt{\frac{2}{N}} C(k) \sum_{k=0}^{N-1} X(k) \cos\left(\frac{(2n+1)k\pi}{2N}\right) \qquad n = 0, \ldots N-1 \qquad \ldots(5.25)$$

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & k = 0 \\ 1 & otherwise \end{cases} \qquad \ldots(5.26)$$

The constant C(k) is defined by Eq. 5.26. If the input sequence has more than *N* sample points, then it can be divided into sub-sequences of length *N* and DCT can be applied to these chunks independently. In each such computation, the values of the basic function points does not change, but the values of *x*(*n*) changes in each sub-sequence, enabling the basic functions to be pre-computed offline and then multiplied with the sub-sequences. This reduces the number of mathematical operations (i.e., multiplications and additions) thereby rendering computation efficiency. The sample C code used for testing the 1D DCT is shown in Optimization 1.

Optimization 1: 1D DCT

```
void dct(float **DCTMatrix, float **Matrix, int N){
    DCTMatrix[u] = 0;
     for (i = 0; i < N; i++) {
        { DCTMatrix[u] + = Matrix[i] * cos(M_PI/((float)N) * (i+1./2.)*u) ;   }  } } } }
```

We can look at the DCT as a matrix multiplication [5.8] Where the inputs and outputs are row-vectors: X=x×M, where M is the cosine coefficient matrix. Optimization 2 shows the representation as matrix form.

Optimization 2: 1D DCT [4.4]

```
static const double c0 = 1. / sqrt(2.) * sqrt(2. / 8.);
static const double c1 = cos(M_PI * 1. / 16.) * sqrt(2. / 8.);
static const double c2 = cos(M_PI * 2. / 16.) * sqrt(2. / 8.);
static const double c3 = cos(M_PI * 3. / 16.) * sqrt(2. / 8.);
static const double c4 = cos(M_PI * 4. / 16.) * sqrt(2. / 8.);
static const double c5 = cos(M_PI * 5. / 16.) * sqrt(2. / 8.);
static const double c6 = cos(M_PI * 6. / 16.) * sqrt(2. / 8.);
static const double c7 = cos(M_PI * 7. / 16.) * sqrt(2. / 8.);

#define a x[0]
// etc

void dct_ii_8a(const double x[8], double X[8]) {
  X[0] = a*c0 + b*c0 + c*c0 + d*c0 + e*c0 + f*c0 + g*c0 + h*c0;
  X[1] = a*c1 + b*c3 + c*c5 + d*c7 - e*c7 - f*c5 - g*c3 - h*c1;
  X[2] = a*c2 + b*c6 - c*c6 - d*c2 - e*c2 - f*c6 + g*c6 + h*c2;
  X[3] = a*c3 - b*c7 - c*c1 - d*c5 + e*c5 + f*c1 + g*c7 - h*c3;
  X[4] = a*c4 - b*c4 - c*c4 + d*c4 + e*c4 - f*c4 - g*c4 + h*c4;
  X[5] = a*c5 - b*c1 + c*c7 + d*c3 - e*c3 - f*c7 + g*c1 - h*c5;
  X[6] = a*c6 - b*c2 + c*c2 - d*c6 - e*c6 + f*c2 - g*c2 + h*c6;
  X[7] = a*c7 - b*c5 + c*c3 - d*c1 + e*c1 - f*c3 + g*c5 - h*c7;
}
```

Optimization 2 can be further optimized after factoring leading to Optimization 3.

Optimization 3: 1D DCT

```
void dct_ii_8b(const double x[8], double X[8]) {
  double c0 = 1. / sqrt(2.) * sqrt(2. / 8.);
 double c1 = cos(M_PI * 1. / 16.) * sqrt(2. / 8.);
 double c2 = cos(M_PI * 2. / 16.) * sqrt(2. / 8.);
 double c3 = cos(M_PI * 3. / 16.) * sqrt(2. / 8.);
```

```
double c4 = cos(M_PI * 4. / 16.) * sqrt(2. / 8.);
double c5 = cos(M_PI * 5. / 16.) * sqrt(2. / 8.);
double c6 = cos(M_PI * 6. / 16.) * sqrt(2. / 8.);
double c7 = cos(M_PI * 7. / 16.) * sqrt(2. / 8.);
 double ah = a - h;
 double bg = b - g;
 double cf = c - f;
 double de = d - e;
 double adeh = a - d - e + h;
 double bcfg = b - c - f + g;
 X[0] = (a + b + c + d + e + f + g + h)*c0;
 X[1] = ah*c1 + bg*c3 + cf*c5 + de*c7;
 X[2] = adeh*c2 + bcfg*c6;
 X[3] = ah*c3 - bg*c7 - cf*c1 - de*c5;
 X[4] = (a - b - c + d + e - f - g + h)*c4;
     X[5] = ah*c5 - bg*c1 + cf*c7 + de*c3;
     X[6] = adeh*c6 - bcfg*c2;
     X[7] = ah*c7 - bg*c5 + cf*c3 - de*c1;
   }
```

Chen et al. algorithm [5.9] represents the 8-point DCT with matrix transforms. Loffler [5.10] proposed a new class of a fast 1D-DCT algorithm using 11 multiplications and 29 additions. Lee algorithm is also based on the matrix representation [5.11] using butterfly decomposition yielding to an even and an odd part. AAN algorithm [5.12] being the most efficient technique is discussed next.

5.13.2. Pipelining Approach and Implementation of DCT based on AAN algorithm

A fast DCT algorithm commonly known as AAN, named after its authors: Arai, Agui and Nakajima is a optimum algorithm. Their algorithm uses five multiplies and eight post-multipliers.

Table 5.19: Computational steps in AAN algorithm

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
b0=a0+a7	c0=b0+b5	d0=c0+c3	e0=d0	f0=e0	s0=f0
b1=a1+a6	c1=b1-b4	d1=c0-c3	e1=d1	f1=e1	s1=f4+f7
b2=a3-a4	c2=b2+b6	d2=c2	e2=m3*d2	f2=e5+e6	s2=f2
b3=a1-a6	c3=b1+b4	d3=c1+c4	e3=m1*d7	f3=e5-e6	s3=f5-f6
b4=a2+a5	c4=b0-b5	d4=c2-c5	e4=m4*d6	f4=e3+e8	s4=f1
b5=a3+a4	c5=b3+b7	d5=c4	e5=d5	f5=e8-e3	s5=f5+f6
b6=a2-a5	c6=b3+b6	d6=c5	e6=m1*d3	f6=e2+e7	s6=f3
b7=a0-a7	c7=b7	d7=c6	e7=m2*d4	f7=e4+e7	s7=f4-f7

Constants:

m1=cos(4*pi/16); m2=cos(2*pi/16)-cos(6*pi/16); m3=cos(6*pi/16); m4=cos(2*pi/16)+cos(6*pi/16);

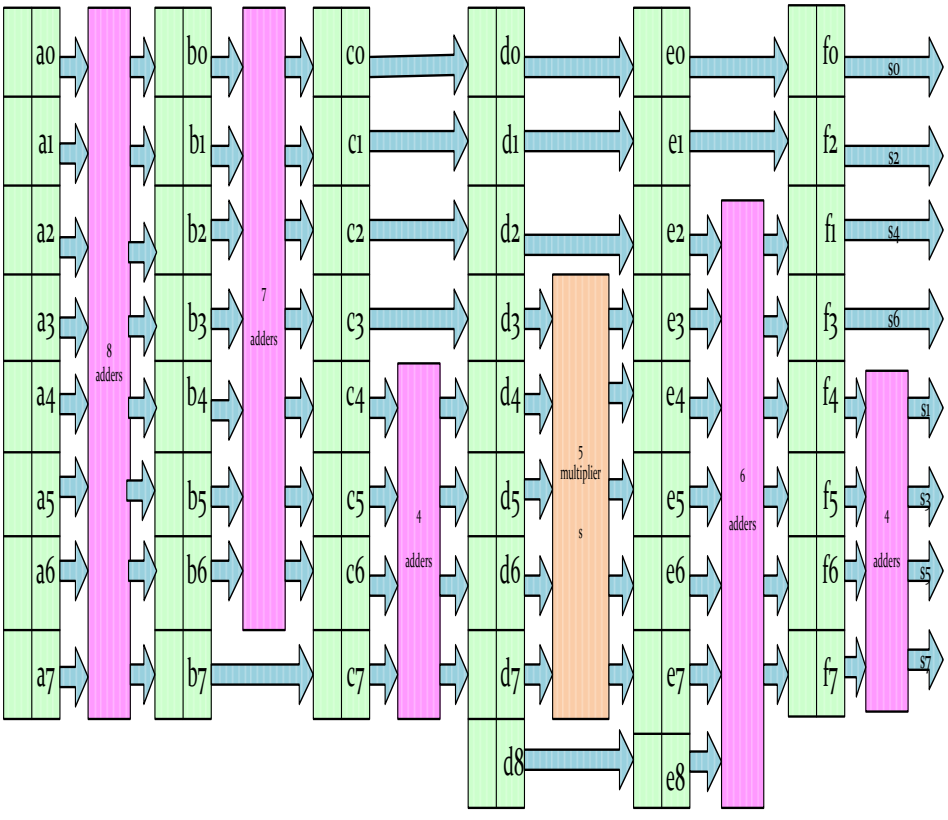


Figure 5.40: Pipeline architecture

The 1D-DCT architecture described in Fig. 5.40 is based on the AAN algorithm. The algorithm described above has six steps, so the pipeline will have six stages. The first stage has eight adders, second stage has seven adders, third stage has four adders, fourth stage five multipliers, fifth stage has six adders, and sixth stage has four adders. Here twenty nine adders,

five multipliers, in each stage eight pipeline registers except stage D and stage E which is having nine registers. AAN architectures were written in VHDL and floating point arithmetic was used to get precise results.

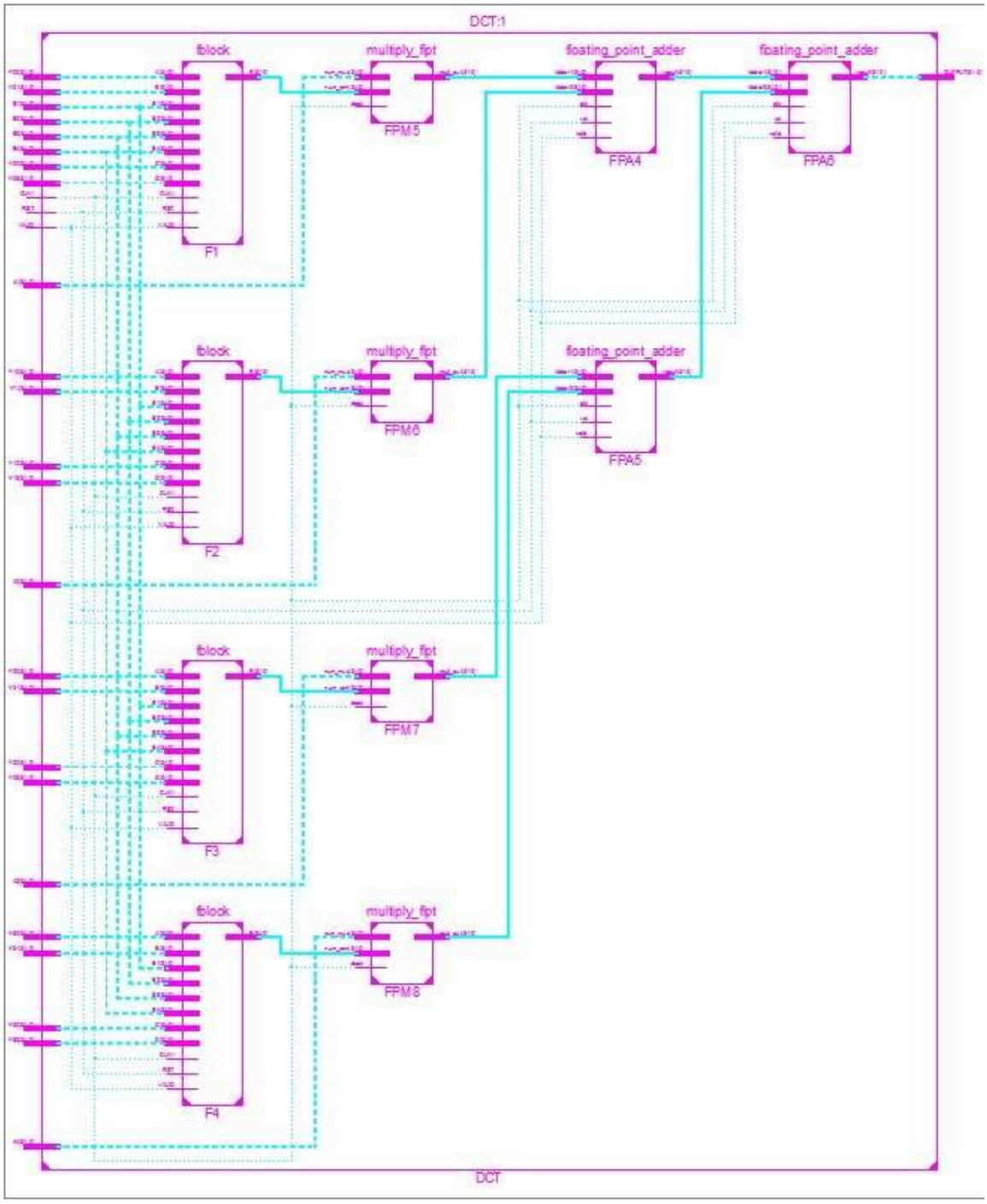


Figure 5.41: DCT netlist in Xilinx ISE

A C program and VHDL for 1-D floating point DCT was written for implementation. Fig. 5.41 shows the netlist obtained from Xilinx ISE of the Discrete Cosine Transform (DCT) architecture taken as a sample. It has four fblocks (F), four multiply_fpt (FPM) and three floating_point_adder (FPA) as shown in Fig. 5.42.

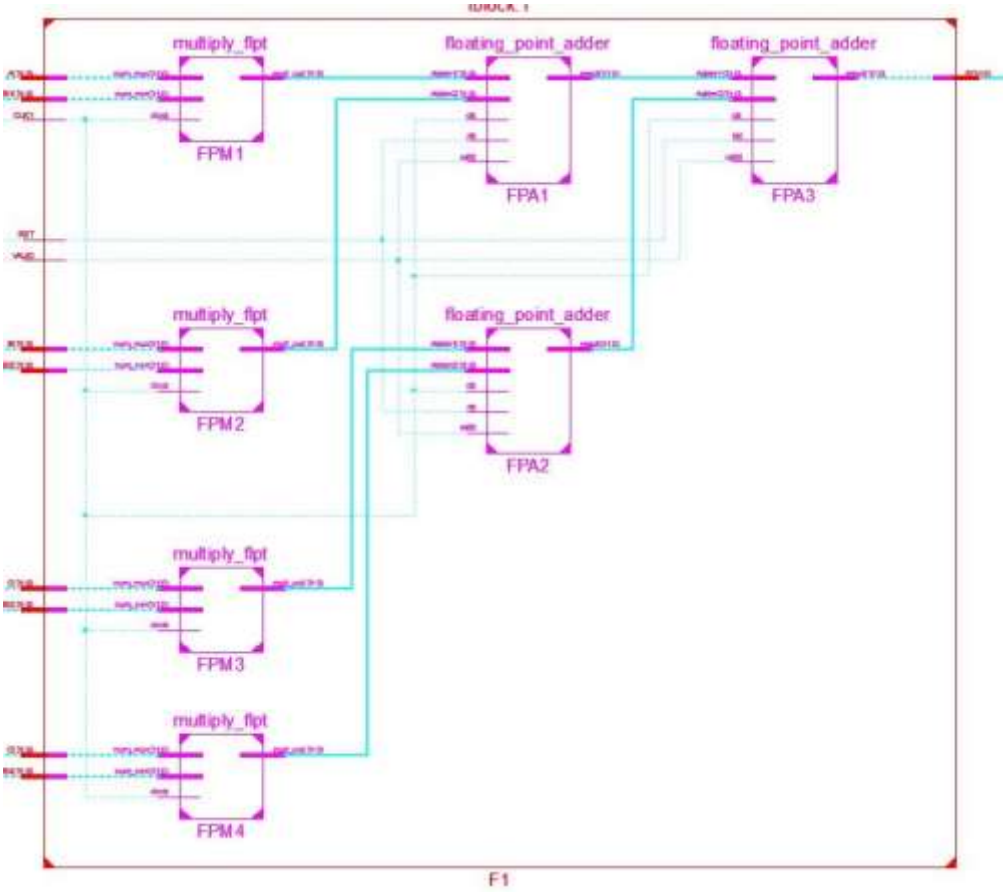


Figure 5.42: F block netlist in Xilinx ISE

5.13.3. HW SW Co-design of DCT

In the previous section we have selected and implemented DCT architecture in HW. Now our goal is to implement DCT as a co-design flow as presented in chapter 3. For the co-design flow, the application should partly run in HW and partly in SW. The optimization 2 discussed above shows that for each X, we need eight multipliers and seven adder/sub. Hence the co-design flow implements them in HW and for eight vectors(X) a SW loop is used. The co-efficient used in the sample 3 are computed in SW and shown in Table. 5.20.

Table 5.20: Matrix as Coefficient

For creating a dataflow model, we wrote codes for floating point multiplier and floating point adder in Verilog.

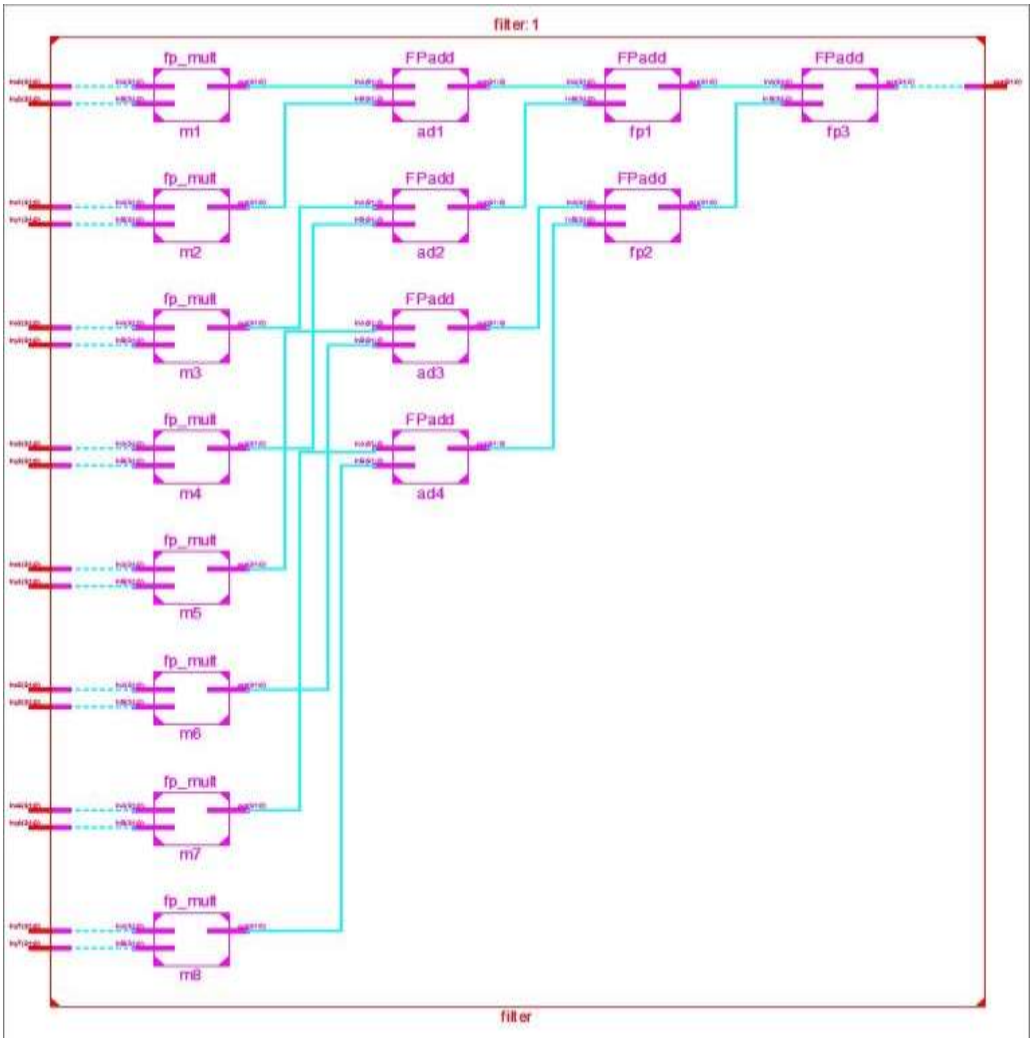


Figure 5.43: Netlist diagram for dataflow model

The dataflow graph for DCT design with floating point adder and multiplier is shown in Fig. 5.43.

Table 5.21 Resources consumed by floating point dataflow model of DCT

Number of Slice Registers	175	44800	0%
Number of Slice LUTs	4677	44800	10%
Number of fully used LUT-FF pairs	175	4677	3%
Number of bonded IOBs	544	640	85%
Number of BUFG/BUFGCTRLs	7	32	21%
Number of DSP48Es	16	128	12%

For calculating the area and delay product, we need the resources consumed. After compiling the DFG DCT design in Xilinx ISE, the resources were tabulated and are shown in Table 5.21 an are compared in next section.

5.13.4. Synthesis and Simulation results:

In this section we compare all the implementations discussed, which include, AAN design as HW, isomorphic design, Co-design and partial reconfiguration design.

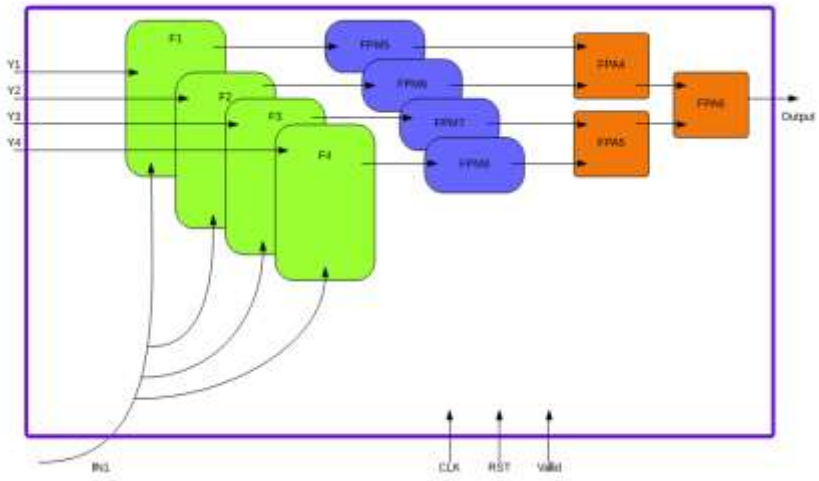


Figure 5.44: Redrawn DCT netlist showing isomorphic modules

During the synthesis we found that some isomorphic sets in DCT design (as shown in Fig. 5.44). The algorithm discussed for identification of isomorphic graphs gives us many sets:

a) {(F1, F2, FPM5, FPM6, FPA4) (F3, F4, FPM7, FPM8, FPA5)},

b) {(F1, F2, FPM5, FPM6,) (F3, F4, FPM7, FPM8)},

c) {(FPM5, FPM6,) (FPM7, FPM8)}, {(F1, F2, F3, F4)},

d) {(FPM5, FPM6, FP7)}, {(FPA4, FPA5, FPA6)}.

The time obtained from the proposed design flow using xps_timer is given in Table 5.22.

Table 5.22: Area and delay of each node

name	Time	Area
F Block	4.998 ns	3087 (6%) + 8 DSP(6%)
FPA	4.921 ns	868(1%) + 0
FPM	6.539 ns	91(0%) + 1 DSP

From Table 5.22 we see that F block takes 6% of the resource, which means it is a bigger block as compared to FPA and FPM. Hence we choose ((FPM5, FPM6,) (FPM7, FPM8)}, {(F1, F2, F3, F4)} as implementation, which means one IP core of F black and remaining as another IP core. The Fig. 5.45 shows a sample design in ML507 board (With PPC@200 MHz and PLB@125 MHz and Timer@125 MHz)

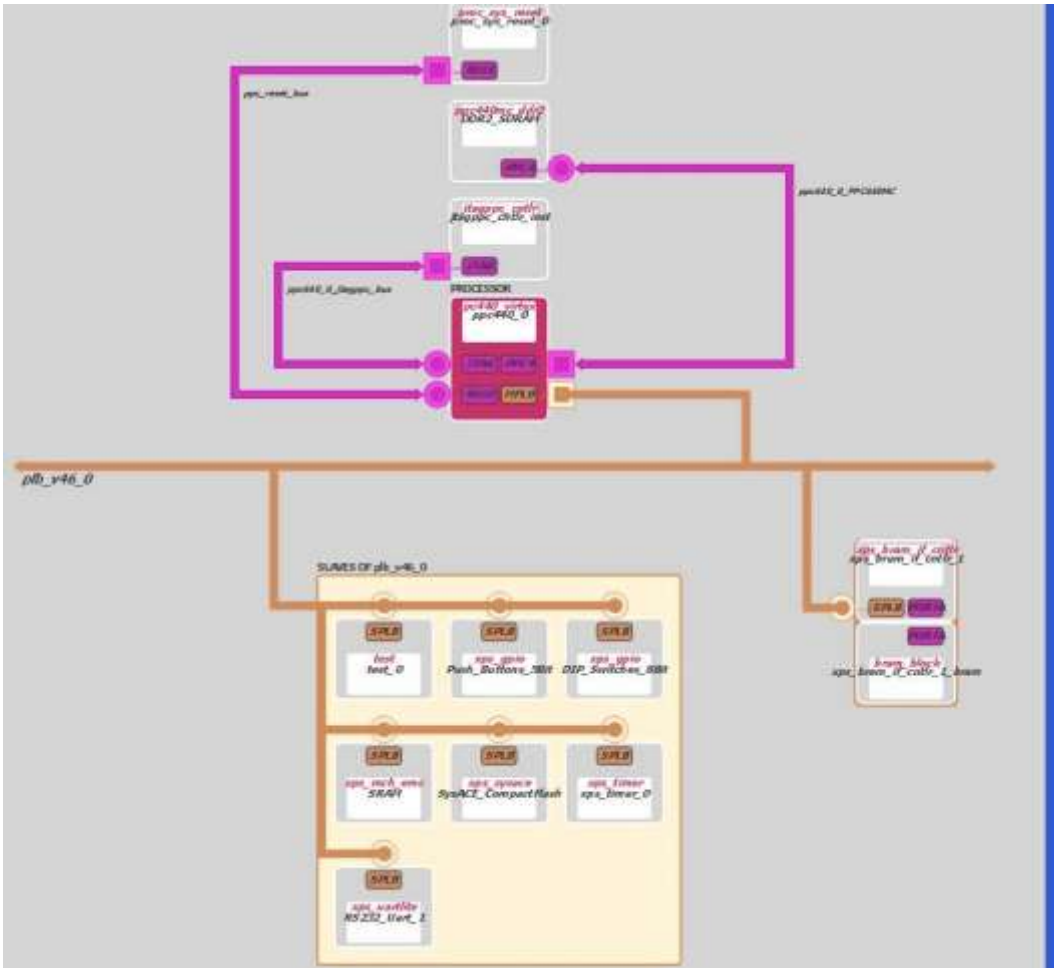


Figure 5.45: EDK components used in implementation

The AAN architecture was interfaced with the bus and is shown in Fig. 5.33 with test as the name of the block.

Table 5.23: Showing the resources for the AAN and DFG design flow as highlights

Report	Flip Flops Used	LUTs Used	BRAMS Used	Errors
system	**11376**	***26396***	**6**	0
dfgdct_0_wrapper	**965**	***7426***		**0**
clock_generator_0_wrapper	4	3		0
plb_v46_0_wrapper	144	364		0
xps_bram_if_cntlr_1_bram_wrapper			4	0
xps_bram_if_cntlr_1_wrapper	255	202		0
test_0_wrapper	**6101**	***15622***		**0**
proc_sys_reset_0_wrapper	69	53		0
jtagppc_cntlr_inst_wrapper		2		0
dip_switches_8bit_wrapper	124	64		0
xps_timer_0_wrapper	357	290		0
sysace_compactflash_wrapper	209	99		0
ddr2_sdram_wrapper	2355	1770	2	0
sram_wrapper	544	316		0
push_buttons_5bit_wrapper	103	55		0
rs232_uart_1_wrapper	144	127		0
ppc440_0_wrapper	2	3		0

Table 5.23 shows the resources used by DCT as HW IP and DFG as HW IP. These values are used in Table 5.24.

Table 5.24: Comparison of various implementations done

	Design		DCT		
			Cycles(8 ns)	Area	Product
a	SW Implementation	simulation	7840(8M+7A)	8000	12.48 x10^6
b	HW Implementation	simulation	173.894	12736	2.622x10^6
c	SW-HW Implementation	simulation	2000(250x8)	13939	10.15x10^6
1	SW Implementation-1	Sample -2	747070	26396	19719659720(11 digits)
2	SW Implementation-2	Sample -3	78574	26396	2074039304(10 digits)
3	SW Implementation-3	Sample -4	54260	26396	1432246960(10 digits)
4	SW Implementation-4	AAN Algo.	12578	26396	332008888(9 digits)
5	HW Implementation	AAN Complete IP	835	26396 + 15622 = 42018	35085030 (8 digits)
6	SW-HW Implementation	Data flow model	2720	26396 + 7426 = 33822	91995840(8 digits)
7	HW-HW Implementation	One Fblock + one core of(3 FPA + 4 FPM)	1459	26396 + 4096 = 30492	44487828(8 digits)

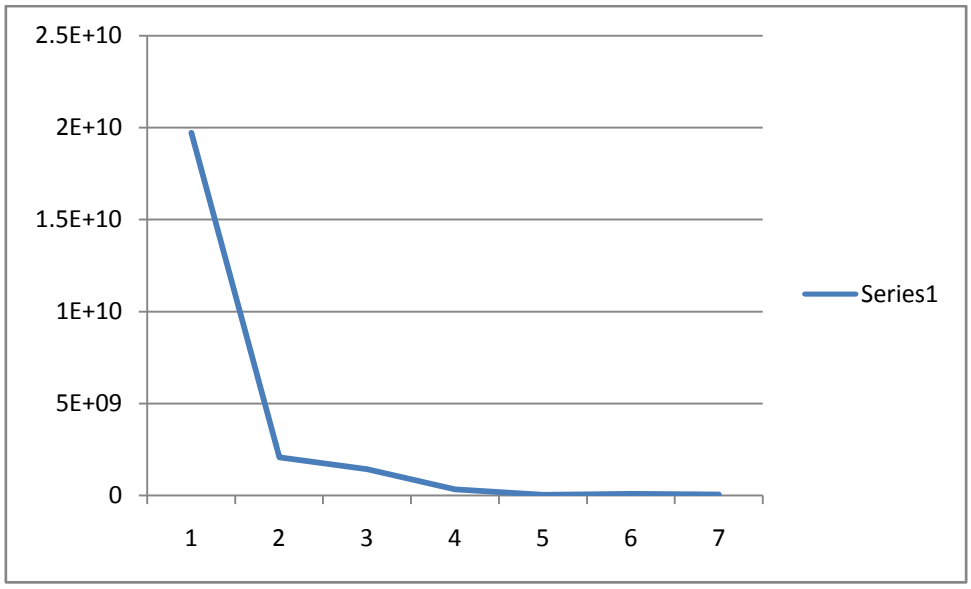


Figure 5.46: Comparison of area and delay product

The area delay product has been shown for seven different implementations. Fig. 5.46 shows as we move toward the HW implementations the product improves. The HW implementation gives the best results. Any kind of HW communication improvement was not applied like direct memory transfer or block RAM for local data. This leads to higher communication overhead and lower performance. In order to further improve the time such improvement

schemes should be used. The next section explores a new HW-HW design flow based on partial reconfiguration.

5.13.5. Synthesis and Simulation Results of PPR Design Flow

The proposed design flow has been successfully tested with the help of DCT design. The work highlights the use of reconfiguration for implementing any design on FPGA without bothering about resources. When creating a partition of size around 20%, the region selection becomes crucial. In many case the design is not routable and gives error. The constraints discussed previously about the number of clock regions may fail as number of clocks available are restricted. In this situation the resultant partitions may be of no use and require modification in the HDL code of the design. It was found that left side upper region of the Virtex-5 as shown in Fig. 5.47 available in ML507 board gave better results in terms of PR region. This may be due to fact tht other regions are better supported for specific controllers like DDR memory.

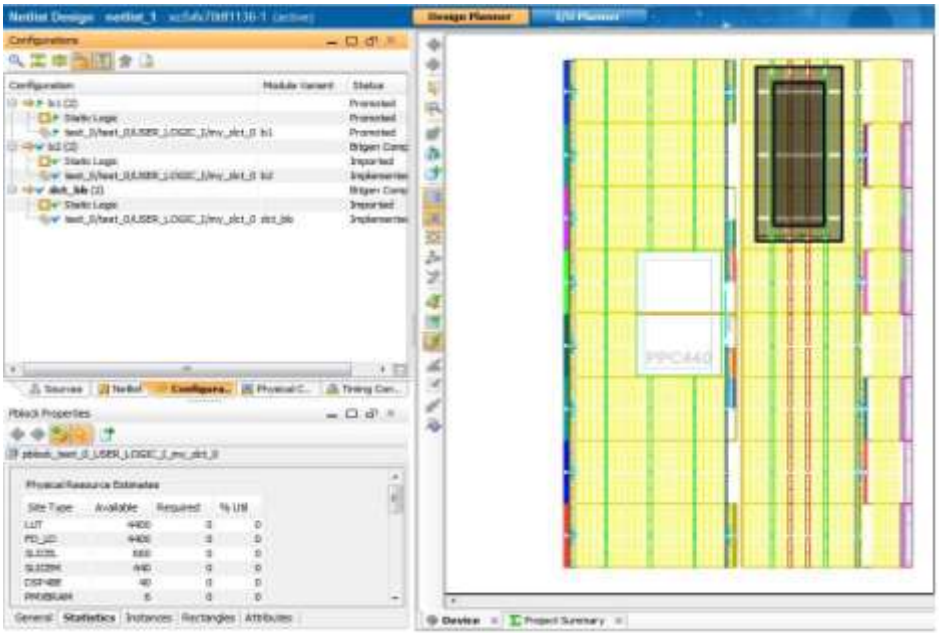


Figure 5.47: Floorplan for DCT having one PRR

During the synthesis we found that (F1, F2, F3, F4), (FPM5, FPM6, FPM7),(FPA4, FPA5, FPA6) were similar. From the synthesis report, we found the resource requirement of the F modules was equivalent to (FPM5, FPM6, FPM7),(FPA4, FPA5, FPA6). Hence two partitions were created one of F and second of (FPM5, FPM6, FPM7),(FPA4, FPA5, FPA6). The scheduler design discussed in the previous section was used to execute the design. For one set of input PR1 should execute four times and PR2 should be called one time. For each iteration of PR loading the intermediate results are stored and passed to the next stage. The time obtained from the proposed design flow using xps_timer is given in Table 5.25.

Table 5.25: Comparison of reconfiguration time

	CF (msec)	Flash (msec)	DDR (msec)	HW-time (usec)	SW-time (msec)
Measured	41.32	32.44	22.68	6.68	5.97656

This time turns out to be very large since we are loading the bitstream from the CF card. This time was further improved by loading the bitstream from DDR memory. There was a drastic improvement in time since the DDR memory is much faster. Many research works have shown the reconfiguration throughput to around 400 Mbps by using direct memory based bitstream access. If the throughput is assumed to be 400 Mbps then the DCT of 258 Kb will transfer in 647.5 microseconds. This will be a significant improvement in the total time which will be 647.5 + 6.68 = 654.18 microseconds.

5.14. Conclusions

- The software implementation takes the maximum time for execution and defines the upper bound for the performance, but it takes less silicon are and less expertise, hence a preferred choice.
- The software implementation can be manually optimized to give better results by converting the computations values to minimum by defining them as constants.
- The hardware implementation gives the best performance and defines the lower bound of the performance, but takes more silicon area and expertise in the implementation.
- An implementation between these two, which is hybrid approach aims at identifies the modules which are similar in the HW description for reusability can be a better approach.
- PR design is not very user friendly and consumes significant time in design cycle.
- A new implementation based on partial reconfiguration can only be useful if the reconfiguration throughput is very high, order of 400 Mbps. The conventional ICAP controller cannot be useful with its current implementation.
- A generable clustering approach based on genetic algorithm can be very effective for large graphs.
- MRTG provides a modular approach for generating user-controlled, truly random task graphs that find relevance in simulating today's scheduling problems in parallel, distributed systems and fields like hardware software co-design**.** This modular nature makes the program code far more reusable than a conventional monolithic design. Future improvements are easier to make, as additional modules can be added without disturbing the functionality of the original stable software. It also makes the program very flexible to use, as now researchers can choose to run only those modules that they require and also

change the order of execution of modules to suit their needs. The layer-by-layer approach followed in MRTG, with the ability to define different types of nodes with their individual parameters separates it from existing available solutions and makes it highly valuable for researchers working in areas like reconfigurable computing, System on Chip and for scheduling simulation in the problem of many core processors, to choose how to spread the work among such large number of processing cores. In this chapter the partitioning and scheduling was applied to RC systems.

REFERENCES

5.1 ML507 Reference Designs : ML507 Memory Interface Generator Design, http://www.xilinx.com/products/boards/ml507/reference_designs.htm

5.2 E. R. Gansner and S. C. North, An open graph visualization system and its applications to software engineering, Software: Practice and Experience, 30(11):1203–1233, 2000.

5.3 Robert P. Dick, David L. Rhodes, and Wayne Wolf, TGFF: Task Graphs for Free, (CODES/CASHE'98), Proceedings of the Sixth International Workshop on Hardware/Software Co-design, 1998.

5.4 Random Task and Resource Graph Tool, users.ecs.soton.ac.uk/ras1n09/rtrg/index.html.

5.5 Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J. M., & Wagner, F. (2010, March), Random graph generation for scheduling simulations, In Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (p. 60), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

5.6 T. Tobita and H. Kasahara, A standard task graph set for fair evaluation of multiprocessor scheduling algorithms, Journal of Scheduling, 5(5):379–394, 2002.

5.7 Discrete Cosine Transform , https://unix4lyfe.org/dct-1d/

5.8 Hassan EL-Banna, Alaa A. EL- Fattah, An efficient implementation of 1 DCT using FPGA Technology, Proceedings of the 15th International Conference on Microelectronics, 2003. ICM 2003, pp. 278-281.

5.9 W. Chen, C. H. Smith, And S. C. Fralick, Fast Computational Algorithm for the Discrete Cosine Transform, IEEE Transactions On Communications, Vol. Com-25, No. 9, September 1977.

5.10 C. Loeffler, A. Ligtenberg and G. S. Moschytz, Practical Fast 1-D DCT algorithm with 11 multiplications, Proceedings of ICASSP, vol.2, pp. 988-991, 1989.

5.11 Y.P Lee, A cost effective architecture for 8x8 two-dimensional DCT/IDCT using direct method, IEEE Transactions on circuit and system for video technology Vol. 7, No.3, June, 1997.

5.12 Y. Arai, T. Agui and M. Nakajima, A fast DCT-SQ scheme for images, IEICE transaction, Vol. E71, No. 11, pp 1095-1097, 1998.

Chapter 6

Conclusions

This thesis addresses the various issues involved and parameters to be considered in the development and verification of a framework for reconfigurable computing systems designed for FPGAs to improve the area delay characteristics of various applications. The knowledge of system parameters allows the designer to explore the design space and select a solution satisfying the given constraints. This chapter summarizes the various contributions made and also points out the some of the possible extensions of the proposed methodology as future work.

We have proposed two design flows, each of which takes the input in different formats. The first approach exploits profiling, high level synthesis and genetic algorithm for partitioning a C specification into HW and SW. ChStone benchmark written in C language (developed by University of Toronto) has been selected and compiled using gcc-powerpc tool chain and Vivado-HLS tool. Various optimizations were applied in HLS flow for latency and area trade-offs. One of the program (DfDiv), which is computationally intensive was selected from ChStone benchmark and its processor local bus (PLB) based IP core was migrated in HW, SW and hybrid approaches. The results of real time performance were tested on ML507 board. After real time performance comparison in both SW and HW, various hybrid implementations of Dfdiv program were also generated using genetic algorithm and compared with area-delay product for various solutions.

The second flow of the work is again divided into two approaches. The first one uses dataflow models for mapping applications as HW clusters. For this graph isomorphism was explored to find the clusters and a scheduler was designed to place them in correct order. The results show that this framework is useful for applications where similar patterns exist. In many applications it is also possible that the patterns are dissimilar, so we considered a second approach which is built on genetic algorithm to guide the creation of clusters as IP cores on partial reconfigurable regions. For this the Express benchmark from University of California Santa Barbara was used. This benchmark describes the application as dataflow model and contains nodes ranging from 50 to 100. To simulate the performance easily four programs (sine, cosine, matrix multiplication and exponent) were written which have nodes ranging

from 20 to 120. The performance of the above two approaches (ISO and GA) were compared for the four programs and it was proved that ISO gave better results as compared to GA. After the simulation of GA and ISO, HW experimentation was required to test the feasibility. To check the effectiveness of the above proposed approaches (Co-design, ISO and GA) on real applications, DCT was chosen and tested on ML507 board. The results offer a wide spectrum of design space implementations to the designer with area-delay parameter as the criteria to choose among them.

6.1. Contributions of the Thesis

The System-on-chip flow based on FPGAs was introduced in Chapter 1. The partial reconfiguration feature available in Xilinx FPGA was explained in brief. Chapter 1 described the overall frameworks presented in this thesis, and hence a basic foundation was laid.

In chapter 2 an elaborate literature survey presented which focused on various domains which include:

a. Various frameworks available for HW, SW co-design such as LegUp, ASSET etc.

b. The concept of isomorphic graphs for identifying the similar clusters.

c. Usage of Genetic algorithm for scheduling of dataflow graphs.

d. The partial reconfiguration time overhead.

Resource estimation technique using LLVM compiler was presented in chapter 3. It was successfully demonstrated how resources can be estimated without synthesis for a given C specification. In this thesis work we have efficiently shown the process of resource estimation by creating a library. We have verified the proposed formula by generating HDL code and synthesizing it on Xilinx.

Chapter 4 presented the ChStone benchmark analysis and one of its programs named DfDiv was used as a case study for HW SW co-design. Further GA was applied to a dummy case study and its usefulness in co-design was proved. In this work, we have presented a design flow to partition an application described in the high level specification into HW and SW. The design flow is based on a practical approach, starting from a Vivado compiler. We have successfully demonstrated the partitioning of a program and tabulated the time results of a benchmark program. The approach discussed opens up new horizon for electronic design automation in the field of FPGAs.

A new framework was proposed in chapter 5 based on similar patterns found in data flow graphs. A detailed design flow was presented describing each stage like specification,

clustering and scheduling. Four programs were written to find the effectiveness of the algorithm. The entire algorithm was explained with examples. The HW implementation showed the best result and SW showed the worst. The results of SW-HW were somewhat intermediate. HW-SW implementation should have shown better results, but since any kind of optimization like DMA was not used hence communication overhead was very high leading to intermediate results. The DCT design was used as a case study for proving the effectiveness of the flow. It was tested on ML507 board, which also showed the same results as simulation and was best implemented in HW.

Further GA was also used for clustering of dataflow graphs for partial reconfiguration feature. For this Express benchmark and four programs were written. The proposed design flow has been successfully tested with the help of DCT design. After extensive literature survey AAN algorithm for DCT was selected and coded in VHDL. The design was extended with floating point adder and multiplier units. An efficient pipelined AAN architecture was chosen for partitioning process. DCT was partitioned using the proposed algorithm and implemented in Xilinx PlanAhead software which was very challenging. The design flow was compared in SW and HW with different memory (CF, DDR) implementations. The results clearly show the design flow can be very useful for complex design and open up a new horizon for more automation opportunity in future IP designing. So this thesis highlights the use of reconfiguration for implementing any design on FPGA without bothering about resources.

A modular dataflow graph generator is designed for generating large graphs. MRTG provides a modular approach for generating user-controlled, truly random task graphs that find relevance in simulating today's scheduling problems in parallel, distributed systems and fields like hardware software co-design. This modular nature makes the program code far more reusable than a conventional monolithic design. Future improvements are easier to make, as additional modules can be added without disturbing the functionality of the original stable software. It also makes the program very flexible to use, as now researchers can choose to run only those modules which they require and also change the order of execution of modules to suit their needs. The layer-by-layer approach followed in MRTG, with the ability to define different types of nodes with their individual parameters separates it from existing available solutions and makes it highly valuable for researchers working in areas like reconfigurable computing, System-on-chip and for scheduling simulation in the problem of many core processors, by enabling them to choose how to divide the work among such large number of processing cores.

The results show that the proposed design flow is a very useful extension to currently existing tools that will allow any program to migrate on FPGA irrespective of amount of resources it can use. So it gives the designer a broader overview of how to proceed, plan the resources and chose between HW, SW or hybrid approach to suit his requirements.

The key contributions of this thesis are:

- An efficient HW-SW co-design flow is proposed for analysis, comparison and effectiveness of an application. The proposed flows have been tested using time analysis and resource usage of the function. Using execution time and resource consumption data, how an algorithm like GA can partition the system into HW and SW is vividly demonstrated.
- The amount of resources consumed by a C program can be estimated even without synthesis and compilation. Such a process has been evidenced by using LLVM compiler and generating a library of operators.
- A new framework for HW-HW implementation of an application described in DFG has been proposed. For area reusability, graph isomorphism has been used to identify the similarity in the design and interfaced them as static modules on the bus.
- We have proposed a new framework that partitions the dataflow graphs (DFG) models and executes them as partial modules. The advantage of this flow is that it allows the design to be mapped onto FPGA irrespective of amount of resources it consumes. The constraints imposed by partial reconfiguration flow have been used to design as efficient GA that can produce clusters of required size.
- A modular random task graph generator has been designed for generating heavy loads, giving dot format files as input to partitioning process. This is a versatile generator through which different graphs can be generated by modifying an input specification required.
- The developed frameworks have been tested on ML507 with the DCT design.
- The comparison of the proposed design flow with HW-SW implementation is presented and we have highlighted the pros-cons of the two approaches.

6.2. Limitations of the Work Done

Here we outline the problems and challenges that are encountered while adopting a partial reconfiguration design flow. Since all the experiential work has been tested with Xilinx PR flow, the same will be used as reference in the problems given below.

A. Complex Design Process

The PR design flow starts from a writing module and testing it in Xilinx ISE. The module is then interfaced with the bus as an empty box with only entry and exit ports. After compiling it in XPS and generating the netlist, the design is exported to SDK where the scheduler is written in C language. In the same flow, the PlanAhead floor planning SW is used for creating the PRR and various modules are bound to these regions. Since the PRR is manually selected it was found that certain regions are more favorable than others. Hence the design flow becomes a five step process which takes several hours to complie. If any point is missed out like incorrect region, time closure failure or incorrect output, then the process has to be repeated. This concludes the complexity of the design flow and it was seen that out of 20 projects only 5 were a success. The design flow should be optimized by creating a PR region automatically by the tools in the ISE design flow and when the design is synthesized, the tool should guide the design which regions are better for PR flow.

B. Reconfiguration Overhead

The partial bitstreams are stored in memory like SD card, CF card or Flash for permanent storage. Loading the bit file from these memories takes significant time. ICAP controller loads the bitstream and has a maximum throughput of about 400 Mbps. But in reality the throughput comes out to be around 4-10 Mbps. This problem can be solved by creating a specialized HW for Reconfigurable architectures in the future.

From the above discussion the following limitations were identified while working on the proposed flows:

1. The entire work uses Virtex-5 available on ML507 board. The library tables are corresponding to this series. Hence for the migration of the framework to the other platforms requires that all the HDL library operates should be compiled again. Though this process is not time consuming and can easily be done but it needs manual intervention.
2. The work has been shown with PLB bus IP cores, which is becoming obsolete and is replaced by AXI bus. However, this does not affect the presented framework.
3. Comparison of work with other work in the case of partial reconfiguration has not been done. This is due to the fact the design flow is novel and a standard design for comparison is missing.
4. DCT was used to show the results, which is a sub module in most of the signal processing applications. The framework can be extended to large end to end solutions for better results.

6.3. Future Scope

The process of partitioning has been shown on a dummy specification since a benchmark program was not available with tabulated HW-SW area and delay. So as a future work the benchmark presented in this thesis can be converted to a standard for this purpose. Moreover, currently the entire process has been shown outside the EDA tool. But if the same is tested within the tool then automation can be inbuilt in the commercial tool. The parameters of GA are decided based on the domain of design. So, automatically defining the optimum iterations based on a given a task graph is also challenging. During timing analysis we have assumed the bus latency of the order of milliseconds for simulating results. A mathematical model can be developed to compute the execution overheads incurred during bus transaction for hybrid design.

The presented thesis work uses dataflow models for experimentation and algorithm verification. The same work can be extended with netlist partitioning that can be directly be used in ASIC design flow. The isomorphic graphs computation has been applied to operator based clustering and the effectiveness of the proposed isomorphic algorithm can be tested with other domains also. A sample design of DCT was created for verification of the algorithm. Many other high level synthesis tools which generate HW can be used for the verification of the algorithm. As mentioned previously, the use of DMA controller and BRAM for data transfer in IP core can considerably improve the performance. Hence they can also be tested and compared for performance.

In this work, we have designed the DCT application and shown the software, hardware and partitioning time on one PR region. Such partitioning of a standalone design has not been reported on real HW. Among the various steps for the design, the most crucial step is partitioning the design so that overall performance is good. In future, we plan to propose a partitioning model so this design flow becomes easy for the designer. In addition, the scheduler design is complicated and depends on the partitioning of ports, number of partitions and a sequence of partitions. It is possible to generate this scheduler automatically which will bring down the time taken for the design flow. Many steps done in the design flow can be can be automated to bring reconfiguration to real world devices.

Being modular, MRTG can have future additions in the form of modules, which can be added without disturbing the original stable software. We plan to make it open source so that researchers who really need it, can develop modules they need and add them to the project so the whole community can use them. We also plan to develop a module to add weights to the connections too. This will be very useful for researchers who need to do scheduling while

taking into account the communication delay and resource expenditure. As an extension we also plan to add a concept of depth.

In this work we were able to estimate the resource requirement of the program on the reconfigurable hardware. Depending on the estimated values, the program can be now partitioned into clusters and executed on partial reconfigurable HW. By estimation technique it is possible to create clusters of the required size. For mapping the partitioned design to one PR region a wrapper generation is required which will interface to the bus. This wrapper should be automatically generated for each partition created. A significant research work is required to generate the scheduler automatically depending on the control flow of the program. The process can be applied to EDA tools on the netlist specifications giving better options for the designer. Further DFG can be created for equations directly giving a new automated flow. The current work will continue in this direction and we propose to bring up a robust scheduler.

APPENDIX-1

The code given below is used to find the time consumed by the program using XPS_timer

Timer Code

```
float time = 0;
float roll_back_time = 0;

int main()
{


  XCACHE_ENABLE_ICACHE();
  XCACHE_ENABLE_DCACHE();

  print("---Entering main---\n\r");

int i,j,k,r;

int data_to_local_link[] = {
     23170, 23170, 23170, 23170, 23170, 23170, 23170, 23170,
     32138, 27246, 18205,  6393, -6393,-18205,-27246,-32138,
     30274, 12540,-12540,-30274,-30274,-12540, 12540, 30274,
     27246, -6593,-32138,-18205, 18205, 32138,  6393,-27246,
     23170,-23170,-23170, 23170, 23170,-23170,-23170, 23170,
     18205,-32138,  6393, 27246,-27246, -6393, 32138,-18205,
     12540,-30274, 30274,-12540,-12540, 30274,-30274, 12540,
      6393,-18205, 27246,-32138, 32138,-27246, 18205, -6393
   };

int data_back_local_link[64];

static XTmrCtr xps_timer_0_Timer;
         u32 CounterControlReg;
         u32 cyclestart;
         u32 cycleend;
         u32 roll_back_count;
         u32 roll_back_cycle_end;
             {
     int status;
       status = XTmrCtr_Initialize(&xps_timer_0_Timer, TIMER_CNTR_0);
                        if (status != XST_SUCCESS) {
                        return XST_FAILURE;}


        XTmrCtr_SetOptions(&xps_timer_0_Timer,      TIMER_CNTR_0,XTC_INT_MODE_OPTION      |
XTC_AUTO_RELOAD_OPTION);
        XTmrCtr_SetResetValue(&xps_timer_0_Timer, TIMER_CNTR_0, RESET_VALUE);
        XTmrCtr_Reset(&xps_timer_0_Timer, TIMER_CNTR_0);

        cyclestart=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);

        XTmrCtr_Start(&xps_timer_0_Timer, TIMER_CNTR_0);

    k = 0; r = 0;
for (j=0;j<8;j++)
    {
if (j==0)
       xil_printf("Perform %dst Datablock out of 8\n\r",j+1);
elseif (j==1)
```

```
        xil_printf("Perform %dnd Datablock out of 8\n\r",j+1);
elseif (j==2)
        xil_printf("Perform %drd Datablock out of 8\n\r",j+1);
else
        xil_printf("Perform %dth Datablock out of 8\n\r",j+1);


      print("\n\r");
      print("Write input values to FSL Channel\n\r");
for (i=0;i<8;i++){
        nputfsl(data_to_local_link[k],0);
        xil_printf("%d; ",data_to_local_link[k]);
        k++;
      };
      print("\n\r\n\r");
      print("Read transformed values back from FSL Channel bus\n\r");
for (i=0;i<8;i++){
        ngetfsl(data_back_local_link[r],0);
        xil_printf("%d; ",data_back_local_link[r]);
        r++;
      };
      print("\n\r\n\r");
      print("-----------------------------------------------------------");
      print("\n\r\n\r");
      CounterControlReg = XTmrCtr_GetControlStatusReg(&xps_timer_0_Timer,TIMER_CNTR_0);
if ((CounterControlReg) == XTC_CSR_INT_OCCURED_MASK)
        { print("\n\r\n\r Timer rolled under!");
          roll_back_count = roll_back_count + 1;
      if (roll_back_count == 1)
          {
                roll_back_cycle_end=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);
                roll_back_time = (cycle_start - roll_back_cycle_end )*0.000000008;
                xil_printf("\r\n roll back time= %ld",roll_back_time);
        }

    };
    XTmrCtr_Stop(&xps_timer_0_Timer, TIMER_CNTR_0);
            }
            xil_printf("\r\n %ld",cyclestart);
        cycleend=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);
        xil_printf("\r\n %ld",cycleend)
      if (roll_back_count>0)
                {
                        time = (roll_back_count*roll_back_time) + (cyclestart-cycleend)*0.000000008;
                  xil_printf("\r\n time= %ld",time);
                }
      else
                {
                        time = (cyclestart-cycleend)*0.000000008;
                        xil_printf("\r\n time= %ld",time);
                }
      if (roll_back_count == 0)
                {
                        XTmrCtr_Reset(&xps_timer_0_Timer, TIMER_CNTR_0);
                  cyclestart=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);
                  XTmrCtr_Start(&xps_timer_0_Timer, TIMER_CNTR_0);
                  CounterControlReg                                                    =
XTmrCtr_GetControlStatusReg(&xps_timer_0_Timer,TIMER_CNTR_0);
              while ((CounterControlReg) != XTC_CSR_INT_OCCURED_MASK)
                        {
                        CounterControlReg                                            =
XTmrCtr_GetControlStatusReg(&xps_timer_0_Timer,TIMER_CNTR_0);
```

```
                        }
                        }
                    XTmrCtr_Stop(&xps_timer_0_Timer, TIMER_CNTR_0);
                    cycleend=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);
                    roll_back_time = (cycle_start - cycleend)*0.000000008;
                    xil_printf("\r\n roll back time = %ld\n",roll_back_time);
                                }
  print("---Exiting main---\n\r");
  XCACHE_DISABLE_ICACHE();
  XCACHE_DISABLE_DCACHE();
return 0;
}
```

APPENDIX-2

The data values given below show the way the GA produces the iterations.

enter deadline: 250

enter population size: 10

enter no of iteration: 4

random_population1 =

1	1	0	1	1	0	1	0
1	0	1	0	1	0	1	0
1	1	1	1	1	1	0	1
1	1	0	1	0	0	1	0
0	0	0	1	0	1	1	1
0	0	0	1	0	0	1	1
1	1	0	0	0	0	1	1
0	0	0	0	1	1	1	1
1	1	0	0	1	0	1	0
1	0	1	0	0	1	0	1

random_population2 =

1	1	0	1	1	0	1	0
1	0	1	0	1	0	1	0
1	1	1	1	1	1	0	1
1	1	0	1	0	0	1	0
0	0	0	1	0	1	1	1
0	0	0	1	0	0	1	1
1	1	0	0	0	0	1	1
0	0	0	0	1	1	1	1
1	1	0	0	1	0	1	0
1	0	1	0	0	1	0	1

random_population3 =

1	1	0	1	1	0	1	0
1	0	1	0	1	0	1	0
1	1	1	1	1	1	0	1
1	1	0	1	0	0	1	0
0	0	0	1	0	1	1	1
0	0	0	1	0	0	1	1
1	1	0	0	0	0	1	1
0	0	0	0	1	1	1	1
1	1	0	0	1	0	1	0
1	0	1	0	0	1	0	1

random_population4 =

1	1	0	1	1	0	1	0
1	0	1	0	1	0	1	0
1	1	1	1	1	1	0	1

```
1   1   0   1   0   0   1   0
0   0   0   1   0   1   1   1
0   0   0   1   0   0   1   1
1   1   0   0   0   0   1   1
0   0   0   0   1   1   1   1
1   1   0   0   1   0   1   0
1   0   1   0   0   1   0   1
```

fitness_value =

Columns 1 through 8

67130 106105 155 157120 171090 191080 151125 141090 127120 133105

42180 96150 205 132170 126130 178105 113170 96125 102165 110145

150 125 175 140 28110 100 1145 24110 140 125

200 170 225 190 150 125 190 145 185 165

tot_time =

317 356 200 407 421 441 401 391 377 383

292 346 150 382 376 428 363 346 352 360

200 223 187 234 278 241 251 274 230 237

165 186 110 209 233 228 213 229 205 214

probability_value =

Columns 1 through 5

0.176738929286603 0.155299083818076 0.197560360087906 0.045259115745567 0.022788156046132

0.176591324708580 0.133358747577602 0.197140185340663 0.023237851213118 0.045566961851729

0.089963068055363 0.162325164546130 0.071860502351513 0.126148268570111 0.000000332181549

0.022118533805913 0.109033604019920 0.000011216295033 0.065660191125667 0.151958365112836

Columns 6 through 10 =

0.000000015866957 0.067508414049207 0.089662589942840 0.133525794238406 0.111657540918305

0.000000019782754 0.067776584657921 0.154975283427512 0.111623218464114 0.089729822976007

0.180411121172521 0.053435720544964 0.027382389461607 0.126148268570111 0.162325164546130

0.194434474404415 0.065660191125667 0.173224460496209 0.087374938310377 0.130524025303962

cum_probability_value =

Columns 1 through 5

0.176738929286603 0.332038013104679 0.529598373192585 0.574857488938152
0.597645644984284
0.176591324708580 0.309950072286182 0.507090257626845 0.530328108839963
0.575895070691692
0.089963068055363 0.252288232601493 0.324148734953006 0.450297003523118
0.450297335704667
0.022118533805913 0.131152137825833 0.131163354120867 0.196823545246534
0.348781910359370

Columns 6 through 10

0.597645660851242 0.665154074900449 0.754816664843289 0.888342459081695
1.000000000000000
0.575895090474446 0.643671675132366 0.798646958559879 0.910270177023993
1.000000000000000
0.630708456877188 0.684144177422152 0.711526566883758 0.837674835453869
1.000000000000000
0.543216384763785 0.608876575889452 0.782101036385661 0.869475974696038
1.000000000000000

roulette_wheel =

Columns 1 through 5

0.660393951257713 0.127848763367490 0.332527054109658 0.394632113574479
0.264216402533851
0.088531581788494 0.490531413270812 0.412561610870134 0.335599558553777
0.045696180184506
0.723845934535098 0.579134452532590 0.441270757276153 0.978874376391397
0.851882048321298
0.962940347350769 0.052232229893487 0.436149459040004 0.292011418871669
0.694746803353756

Columns 6 through 10 =

0.763810931237840 0.841540169840319 0.469050001539782 0.538066460330539
0.237809971328859
0.688557909510549 0.285173447520110 0.180361835941481 0.450591840233820
0.108927749991106
0.774427087251813 0.059198299574430 0.569443789123108 0.393090112338098
0.548257380847610
0.961182516696144 0.181149785587021 0.517754273201131 0.059416332311161
0.779767512749519

CPU1 AND ASIC1

PATTERN = 1 1 1 1 0 1 0 1

```
COST = 145   TIME = 233
node 1   start = 0   end = 20
node 2   start = 20   end = 50
node 3   start = 50   end = 77
node 4   start = 77   end = 97
node 5   start = 20   end = 110
node 6   start = 110   end = 137
node 7   start = 110   end = 200
node 8   start = 200   end = 233
########
CPU1 AND ASIC2
PATTERN = 0  1  1  0  1  1  0  1
COST = 145   TIME = 245
node 1   start = 0   end = 60
node 2   start = 60   end = 75
node 3   start = 75   end = 90
node 4   start = 75   end = 135
node 5   start = 90   end = 105
node 6   start = 135   end = 155
node 7   start = 135   end = 225
node 8   start = 225   end = 245
########
CPU2 AND ASIC1
PATTERN = 0  0  0  1  0  0  1  1
COST = 100   TIME = 241
node 1   start = 0   end = 30
node 2   start = 30   end = 80
node 3   start = 80   end = 134
node 4   start = 80   end = 100
node 5   start = 134   end = 178
node 6   start = 178   end = 208
node 7   start = 134   end = 174
node 8   start = 208   end = 241
########
CPU2 AND ASIC2
PATTERN = 0  0  0  1  0  0  1  1
COST = 125   TIME = 228
node 1   start = 0   end = 30
node 2   start = 30   end = 80
node 3   start = 80   end = 134
node 4   start = 80   end = 90
node 5   start = 134   end = 178
node 6   start = 178   end = 208
node 7   start = 134   end = 149
node 8   start = 208   end = 228
>>
```

APPENDIX-3

The code given below shows how the time for SW instruction was calculated to create the library

```
#include <stdio.h>
#include "xparameters.h"
#include "xenv_standalone.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "gpio_header.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "gpio_header.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "gpio_header.h"
#include "xtmrctr.h"
#include "tmrctr_header.h"

#define TIMER_CNTR_0        0
#define TMRCTR_DEVICE_ID  XPAR_TMRCTR_0_DEVICE_ID
#define RESET_VALUE         0xF0000000
int main()
{


  XCACHE_ENABLE_ICACHE();
  XCACHE_ENABLE_DCACHE();

  print("---Entering main---\n\r");
  static XTmrCtr xps_timer_0_Timer;

  u32 cyclestartadd,cyclebitloadcompadd,cycleresultadd;
      u32 cycleendadd;
      u32 cyclestartmult,cyclebitloadcompmult,cycleresultmult;
          u32 cycleendmult;

       int status;

       status = XTmrCtr_Initialize(&xps_timer_0_Timer, TIMER_CNTR_0);
                        if (status != XST_SUCCESS) {
                        return XST_FAILURE;}
                xil_printf("start program\r\n");

       XTmrCtr_SetOptions(&xps_timer_0_Timer,       TIMER_CNTR_0,XTC_INT_MODE_OPTION      |
XTC_AUTO_RELOAD_OPTION);
       XTmrCtr_SetResetValue(&xps_timer_0_Timer, TIMER_CNTR_0, RESET_VALUE);
       XTmrCtr_Reset(&xps_timer_0_Timer, TIMER_CNTR_0);

     float a = 0.0 , b= 5.0 ,c = 6.0 ;
     xil_printf("\Multiplication\n\r");
                                                XTmrCtr_Reset(&xps_timer_0_Timer, TIMER_CNTR_0);

     cyclestartmult=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);
                                                XTmrCtr_Start(&xps_timer_0_Timer, TIMER_CNTR_0);
                                //          __asm__ __volatile__ ("lwarx %0, 0, %1 \n\t" : "=&r"(ret) : "r"(p));
                                //          asm("lwi r3, r0, 292"); // load word immediate
                                //          asm("addik r3, r3, 1"); // immediate add w/ keep carry
```

```
//		asm("swi r3, r0, 292"); // store word immediate

//		asm ( " addc.	%0,%1,%2	\n" : "=r"(res): "b"(op1), "r"(op2):
"r0"							);

 // asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 // asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 // asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//     asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //    asm ("add %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 // asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 //asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;

//  asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
 // asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
// asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //   asm ("mullw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
  //		asm ("fadd %1,%2,%0": "=f" (a): "f" (b), "f" (c)) ;
// asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
 //asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
//asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
//asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
//asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
//asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
//asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
//asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
//asm ("fadd %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;

 //asm ("fmul %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
asm ("fdiv %0,%1,%2": "=f" (a): "f" (b), "f" (c)) ;
 //  asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
   //			asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
   //				asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
```

```
//                                asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//                          asm ("divw %0,%1,%2": "=fr" (a): "r" (b), "r" (c)) ;
//                          asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//                          asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//                          asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//                          asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;
//                          asm ("divw %0,%1,%2": "=r" (a): "r" (b), "r" (c)) ;

    cyclebitloadcompmult=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);


    //cycleresultmult=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);


    //cycleendmult=XTmrCtr_GetValue(&xps_timer_0_Timer,TIMER_CNTR_0);
                                        XTmrCtr_Stop(&xps_timer_0_Timer, TIMER_CNTR_0);


                                        print("-- Exiting main() --\r\n");
                                          print("-- ---------------for mult ----------------------\r\n");
                                          xil_printf("\r\n cyclestart= %x\r\n",cyclestartmult);
                                          xil_printf("\r\n                                cyclebitloadcomp=
%x\r\n",cyclebitloadcompmult);

                                          //xil_printf("\r\n cycleresult= %x\r\n",cycleresultmult);
                                         // xil_printf("\r\n cycleend= %x\r\n",cycleendmult);
                                          //print("-------------------- for adder ----------------------\r\n");
                                          //xil_printf("\r\n cyclestart= %x\r\n",cyclestartadd);
                                          //xil_printf("\r\n                                cyclebitloadcomp=
%x\r\n",cyclebitloadcompadd);

                                          //xil_printf("\r\n cycleresult= %x\r\n",cycleresultadd);
                                          //xil_printf("\r\n cycleend= %x\r\n",cycleendadd);

  XCACHE_DISABLE_ICACHE();
  XCACHE_DISABLE_DCACHE();

  return 0;
}
```

Brief Biography of the Candidate

Ashish Mishra received his M.E. (Embedded Systems) degree from the Birla Institute of Technology and Science, Pilani, India, in 2008 and is pursuing his Ph.D. degree from BITS-Pilani in the area of Reconfigurable computing. Currently he is working as lecturer in the Department of Electrical and Electronics Engineering, BITS-Pilani. His main area of interest includes high level synthesis and reconfigurable computing systems. He has published ten journals and eight conference papers in the area of embedded systems. He has taught courses like reconfigurable computing, microprocessor and embedded systems and has a total of 14 years of teaching experience.

Brief Biography of the Supervisor

Dr. Kota Solomon Raju is Principal Scientist in Digital Systems Group, CSIR -Central Electronics Engineering Research Institute (CSIR-CEERI), Pilani, Rajasthan, India. Apart from R&D he teaches at CSIR-CEERI, Pilani (part of AcSIR, Chennai) and is also a visiting professor at BITS-Pilani. He received the B.E. from Andhra University, M.E. from BITS-Pilani and Ph.D. from IIT-Roorkee. His research focus includes Advanced Embedded Systems and Architectural Design for various applications, particularly for Software defined radios (SDR), Wireless Sensor Networks, Internet of Things, Network routing protocols, Protocol stack design for hybrid communication and Multimedia data acquisition / compression techniques. He has published more than 100 scientific papers in peer-reviewed international journals and conferences. He has completed 6 R& D projects and currently leading 3 projects as Principal Investigator. He has played the key role in materializing the MOU between CSIR-CEERI, Pilani and Hiroshima University, Japan.

Framework for Translation of C/C++ Applications on Reconfigurable Computing Systems

THESIS

Submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

ASHISH MISHRA

ID No. 2009PHXF038P

Under the Supervision of

Dr. Kota Solomon Raju

Dr. Abhijit Rameshwar Asati

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
PILANI - 333031 (RAJASTHAN) INDIA
2016

Chapter 6

Conclusions

This thesis addresses the various issues involved and parameters to be considered in the development and verification of a framework for reconfigurable computing systems designed for FPGAs to improve the area delay characteristics of various applications. The knowledge of system parameters allows the designer to explore the design space and select a solution satisfying the given constraints. This chapter summarizes the various contributions made and also points out the some of the possible extensions of the proposed methodology as future work.

We have proposed two design flows, each of which takes the input in different formats. The first approach exploits profiling, high level synthesis and genetic algorithm for partitioning a C specification into HW and SW. ChStone benchmark written in C language (developed by University of Toronto) has been selected and compiled using gcc-powerpc tool chain and Vivado-HLS tool. Various optimizations were applied in HLS flow for latency and area trade-offs. One of the program (DfDiv), which is computationally intensive was selected from ChStone benchmark and its processor local bus (PLB) based IP core was migrated in HW, SW and hybrid approaches. The results of real time performance were tested on ML507 board. After real time performance comparison in both SW and HW, various hybrid implementations of Dfdiv program were also generated using genetic algorithm and compared with area-delay product for various solutions.

The second flow of the work is again divided into two approaches. The first one uses dataflow models for mapping applications as HW clusters. For this graph isomorphism was explored to find the clusters and a scheduler was designed to place them in correct order. The results show that this framework is useful for applications where similar patterns exist. In many applications it is also possible that the patterns are dissimilar, so we considered a second approach which is built on genetic algorithm to guide the creation of clusters as IP cores on partial reconfigurable regions. For this the Express benchmark from University of California Santa Barbara was used. This benchmark describes the application as dataflow model and contains nodes ranging from 50 to 100. To simulate the performance easily four programs (sine, cosine, matrix multiplication and exponent) were written which have nodes ranging

from 20 to 120. The performance of the above two approaches (ISO and GA) were compared for the four programs and it was proved that ISO gave better results as compared to GA. After the simulation of GA and ISO, HW experimentation was required to test the feasibility. To check the effectiveness of the above proposed approaches (Co-design, ISO and GA) on real applications, DCT was chosen and tested on ML507 board. The results offer a wide spectrum of design space implementations to the designer with area-delay parameter as the criteria to choose among them.

6.1. Contributions of the Thesis

The System-on-chip flow based on FPGAs was introduced in Chapter 1. The partial reconfiguration feature available in Xilinx FPGA was explained in brief. Chapter 1 described the overall frameworks presented in this thesis, and hence a basic foundation was laid.

In chapter 2 an elaborate literature survey presented which focused on various domains which include:

a. Various frameworks available for HW, SW co-design such as LegUp, ASSET etc.

b. The concept of isomorphic graphs for identifying the similar clusters.

c. Usage of Genetic algorithm for scheduling of dataflow graphs.

d. The partial reconfiguration time overhead.

Resource estimation technique using LLVM compiler was presented in chapter 3. It was successfully demonstrated how resources can be estimated without synthesis for a given C specification. In this thesis work we have efficiently shown the process of resource estimation by creating a library. We have verified the proposed formula by generating HDL code and synthesizing it on Xilinx.

Chapter 4 presented the ChStone benchmark analysis and one of its programs named DfDiv was used as a case study for HW SW co-design. Further GA was applied to a dummy case study and its usefulness in co-design was proved. In this work, we have presented a design flow to partition an application described in the high level specification into HW and SW. The design flow is based on a practical approach, starting from a Vivado compiler. We have successfully demonstrated the partitioning of a program and tabulated the time results of a benchmark program. The approach discussed opens up new horizon for electronic design automation in the field of FPGAs.

A new framework was proposed in chapter 5 based on similar patterns found in data flow graphs. A detailed design flow was presented describing each stage like specification,

clustering and scheduling. Four programs were written to find the effectiveness of the algorithm. The entire algorithm was explained with examples. The HW implementation showed the best result and SW showed the worst. The results of SW-HW were somewhat intermediate. HW-SW implementation should have shown better results, but since any kind of optimization like DMA was not used hence communication overhead was very high leading to intermediate results. The DCT design was used as a case study for proving the effectiveness of the flow. It was tested on ML507 board, which also showed the same results as simulation and was best implemented in HW.

Further GA was also used for clustering of dataflow graphs for partial reconfiguration feature. For this Express benchmark and four programs were written. The proposed design flow has been successfully tested with the help of DCT design. After extensive literature survey AAN algorithm for DCT was selected and coded in VHDL. The design was extended with floating point adder and multiplier units. An efficient pipelined AAN architecture was chosen for partitioning process. DCT was partitioned using the proposed algorithm and implemented in Xilinx PlanAhead software which was very challenging. The design flow was compared in SW and HW with different memory (CF, DDR) implementations. The results clearly show the design flow can be very useful for complex design and open up a new horizon for more automation opportunity in future IP designing. So this thesis highlights the use of reconfiguration for implementing any design on FPGA without bothering about resources.

A modular dataflow graph generator is designed for generating large graphs. MRTG provides a modular approach for generating user-controlled, truly random task graphs that find relevance in simulating today's scheduling problems in parallel, distributed systems and fields like hardware software co-design. This modular nature makes the program code far more reusable than a conventional monolithic design. Future improvements are easier to make, as additional modules can be added without disturbing the functionality of the original stable software. It also makes the program very flexible to use, as now researchers can choose to run only those modules which they require and also change the order of execution of modules to suit their needs. The layer-by-layer approach followed in MRTG, with the ability to define different types of nodes with their individual parameters separates it from existing available solutions and makes it highly valuable for researchers working in areas like reconfigurable computing, System-on-chip and for scheduling simulation in the problem of many core processors, by enabling them to choose how to divide the work among such large number of processing cores.

The results show that the proposed design flow is a very useful extension to currently existing tools that will allow any program to migrate on FPGA irrespective of amount of resources it can use. So it gives the designer a broader overview of how to proceed, plan the resources and chose between HW, SW or hybrid approach to suit his requirements.

The key contributions of this thesis are:

- An efficient HW-SW co-design flow is proposed for analysis, comparison and effectiveness of an application. The proposed flows have been tested using time analysis and resource usage of the function. Using execution time and resource consumption data, how an algorithm like GA can partition the system into HW and SW is vividly demonstrated.
- The amount of resources consumed by a C program can be estimated even without synthesis and compilation. Such a process has been evidenced by using LLVM compiler and generating a library of operators.
- A new framework for HW-HW implementation of an application described in DFG has been proposed. For area reusability, graph isomorphism has been used to identify the similarity in the design and interfaced them as static modules on the bus.
- We have proposed a new framework that partitions the dataflow graphs (DFG) models and executes them as partial modules. The advantage of this flow is that it allows the design to be mapped onto FPGA irrespective of amount of resources it consumes. The constraints imposed by partial reconfiguration flow have been used to design as efficient GA that can produce clusters of required size.
- A modular random task graph generator has been designed for generating heavy loads, giving dot format files as input to partitioning process. This is a versatile generator through which different graphs can be generated by modifying an input specification required.
- The developed frameworks have been tested on ML507 with the DCT design.
- The comparison of the proposed design flow with HW-SW implementation is presented and we have highlighted the pros-cons of the two approaches.

6.2. Limitations of the Work Done

Here we outline the problems and challenges that are encountered while adopting a partial reconfiguration design flow. Since all the experiential work has been tested with Xilinx PR flow, the same will be used as reference in the problems given below.

A. Complex Design Process

The PR design flow starts from a writing module and testing it in Xilinx ISE. The module is then interfaced with the bus as an empty box with only entry and exit ports. After compiling it in XPS and generating the netlist, the design is exported to SDK where the scheduler is written in C language. In the same flow, the PlanAhead floor planning SW is used for creating the PRR and various modules are bound to these regions. Since the PRR is manually selected it was found that certain regions are more favorable than others. Hence the design flow becomes a five step process which takes several hours to complie. If any point is missed out like incorrect region, time closure failure or incorrect output, then the process has to be repeated. This concludes the complexity of the design flow and it was seen that out of 20 projects only 5 were a success. The design flow should be optimized by creating a PR region automatically by the tools in the ISE design flow and when the design is synthesized, the tool should guide the design which regions are better for PR flow.

B. Reconfiguration Overhead

The partial bitstreams are stored in memory like SD card, CF card or Flash for permanent storage. Loading the bit file from these memories takes significant time. ICAP controller loads the bitstream and has a maximum throughput of about 400 Mbps. But in reality the throughput comes out to be around 4-10 Mbps. This problem can be solved by creating a specialized HW for Reconfigurable architectures in the future.

From the above discussion the following limitations were identified while working on the proposed flows:

1. The entire work uses Virtex-5 available on ML507 board. The library tables are corresponding to this series. Hence for the migration of the framework to the other platforms requires that all the HDL library operates should be compiled again. Though this process is not time consuming and can easily be done but it needs manual intervention.
2. The work has been shown with PLB bus IP cores, which is becoming obsolete and is replaced by AXI bus. However, this does not affect the presented framework.
3. Comparison of work with other work in the case of partial reconfiguration has not been done. This is due to the fact the design flow is novel and a standard design for comparison is missing.
4. DCT was used to show the results, which is a sub module in most of the signal processing applications. The framework can be extended to large end to end solutions for better results.

6.3. Future Scope

The process of partitioning has been shown on a dummy specification since a benchmark program was not available with tabulated HW-SW area and delay. So as a future work the benchmark presented in this thesis can be converted to a standard for this purpose. Moreover, currently the entire process has been shown outside the EDA tool. But if the same is tested within the tool then automation can be inbuilt in the commercial tool. The parameters of GA are decided based on the domain of design. So, automatically defining the optimum iterations based on a given a task graph is also challenging. During timing analysis we have assumed the bus latency of the order of milliseconds for simulating results. A mathematical model can be developed to compute the execution overheads incurred during bus transaction for hybrid design.

The presented thesis work uses dataflow models for experimentation and algorithm verification. The same work can be extended with netlist partitioning that can be directly be used in ASIC design flow. The isomorphic graphs computation has been applied to operator based clustering and the effectiveness of the proposed isomorphic algorithm can be tested with other domains also. A sample design of DCT was created for verification of the algorithm. Many other high level synthesis tools which generate HW can be used for the verification of the algorithm. As mentioned previously, the use of DMA controller and BRAM for data transfer in IP core can considerably improve the performance. Hence they can also be tested and compared for performance.

In this work, we have designed the DCT application and shown the software, hardware and partitioning time on one PR region. Such partitioning of a standalone design has not been reported on real HW. Among the various steps for the design, the most crucial step is partitioning the design so that overall performance is good. In future, we plan to propose a partitioning model so this design flow becomes easy for the designer. In addition, the scheduler design is complicated and depends on the partitioning of ports, number of partitions and a sequence of partitions. It is possible to generate this scheduler automatically which will bring down the time taken for the design flow. Many steps done in the design flow can be can be automated to bring reconfiguration to real world devices.

Being modular, MRTG can have future additions in the form of modules, which can be added without disturbing the original stable software. We plan to make it open source so that researchers who really need it, can develop modules they need and add them to the project so the whole community can use them. We also plan to develop a module to add weights to the connections too. This will be very useful for researchers who need to do scheduling while

taking into account the communication delay and resource expenditure. As an extension we also plan to add a concept of depth.

In this work we were able to estimate the resource requirement of the program on the reconfigurable hardware. Depending on the estimated values, the program can be now partitioned into clusters and executed on partial reconfigurable HW. By estimation technique it is possible to create clusters of the required size. For mapping the partitioned design to one PR region a wrapper generation is required which will interface to the bus. This wrapper should be automatically generated for each partition created. A significant research work is required to generate the scheduler automatically depending on the control flow of the program. The process can be applied to EDA tools on the netlist specifications giving better options for the designer. Further DFG can be created for equations directly giving a new automated flow. The current work will continue in this direction and we propose to bring up a robust scheduler.