DEVELOPMENT OF AN EFFICIENT TASK SCHEDULING ALGORITHM FOR RECONFIGURABLE COMPUTING SYSTEMS

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

MASTER OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

By

ARCHANA KOKKULA

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE-247 667 (INDIA)

JUNE, 2007

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled, "***Development of an Efficient Task Scheduling Algorithm for Reconfigurable Computing Systems***" being submitted in the partial fulfillment of the requirements for the award of degree of ***Master of Technology*** in ***Information Technology***, in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (INDIA), is an authentic record of my work, carried out from August 2006 to June 2007, under the guidance and supervision of **Dr. Durga Toshniwal**, Assistant Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India).

The matter embodied in the dissertation report to the best of our knowledge has not been submitted for the award of any other degree elsewhere.

Dated : 14-5-07.
Place : Roorkee

Archana.
(Archana Kokkula)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: 15/05/07
Place: Roorkee.

[illegible]
(Dr. Durga Toshniwal)
Assistant Professor,
Dept. of E &C Engg.
IITR, Roorkee - 247667,
INDIA.

Acknowledgements

First and foremost, my sincere regards to my supervisor Dr. Durga Toshniwal, Associate Professor, Department of Electronics and Computer Engineering, for her encouragement, valuable suggestions, amiable, amicable and caring nature. Her guidance was helpful throughout this dissertation work.

Special thanks to Prof. Kumkum Garg, Professor, Department of Electronics and Computer Engineering under whom I have carried out my Project.

I also thank Prof. D. K. Mehra, Head of the Department, Department of Electronics and Computer Engineering for providing the necessary facilities for carrying out this work.

I express my sincere thanks to Prof R. C. Joshi, Prof. Kuldip Singh, Department of Electronics and Computer Engineering, Engineering for their kind help and moral support throughout this dissertation work.

I am thankful to Mr. Kota Solomon Raju, Scientist, Central Electronics Engineering Research Institute (CEERI), Pilani, whose contribution was crucial in making this work come true.

My special sincere heartfelt gratitude to my family, and especially my loving brother Sridhar (SLB) for his help, whose sincere prayers, best wishes, support and encouragement has been a constant source of strength to me during the entire work.

Finally I would also like to thank all my friends for their support and valuable suggestions.

Archana

Archana Kokkula

Contents

Abstract

Reconfigurable Computing (RC) is an emerging paradigm of research that offers cost-effective solutions for computationally intensive applications through hardware reuse. To fulfill the gap between Application Specific Integrated Circuit (ASIC) and Application Specific Instruction Processor (ASIP) reconfigurable computing has been introduced. In reconfigurable computing environment one can have performance like ASIC while having general-purpose processor flexibility. There is a growing demand in this domain for techniques exploiting inherent parallelism in the target application and scheduling the parallelized application. There arises the need for scheduling and mapping of the tasks on to hardware resources. The Reconfigurable Logic Units (RLUs) i.e., hardware resources represent reconfigurable hardware modules on a reconfigurable System-on-Chip (rSoC).

In this thesis, the problem of scheduling and mapping of the tasks onto the several RLUs for a given application task graph is considered, where the RLUs vary in terms of chip area (henceforth referred as variable area RLUs) and each task can have multiple versions of implementations (configuration bit streams) having corresponding execution time. An efficient scheduling algorithm dealing with the above mentioned problem is developed using dynamic programming approach, with the objective of minimizing the total execution time. The algorithm takes into account the reconfiguration delay when assigning task to the RLU in addition to the task execution time. On the similar lines, another algorithm using the Greedy Heuristic approach is developed for performance comparison. The comparison studies are carried out, which show that our algorithm always performs better.

The algorithm is implemented using SystemC environment to support hardware as well as software co-simulation. The development of algorithms is done keeping in view the Virtex II Pro XUP FPGA development kit standards.

List of Figures

Chapter 1

Introduction

Reconfigurable Computing Systems (RCS) [2] combine a processor with reconfigurable hardware. This area of computing is consolidating itself as a real alternative to ASICs and general purpose processors. The main advantages of RC derive from its unique combination of broad applicability (like general-purpose systems), and achievable performance (comparable to special purpose circuitry). Recently this area of computing has reached performance figures that enable it to appear as a serious competitor in a Digital Signal Processing (DSP) and multimedia applications market. However, the scheduling of the different tasks must be carefully analyzed to efficiently exploit all the capabilities of a reconfigurable system. It is especially crucial in real time applications.

1.1 General Introduction

Microprocessor is the heart of most current high performance computing platforms. They provide a flexible computing platform and are capable of executing large class of applications. Software for microprocessors is developed by implementing higher level functions using the instruction set of the architecture. As a result, the same fixed hardware can be used for many general purpose applications. Unfortunately, this generality is achieved at the expense of performance. The software program stored in memory has to be fetched, decoded and executed. In addition to this, data is fetched from and stored back to memory. These conditions force explicit sequential execution of the program. Casting all complex functions into simpler instructions to be executed sequentially on the processor results in a degraded performance.

Application Specific Integrated Circuits (ASICs) [2] provide an alternative to address the performance issues of general purpose microprocessors. ASICs are designed for a specific application and hence, each ASIC has a fixed functionality and superior performance for a highly restricted set of applications. However, ASICs restrict the flexibility of the architecture and exclude any post-design optimizations and upgrades in features and algorithms.

A new computing paradigm using *Reconfigurable Computing* [2] promises an intermediate trade-off between flexibility and performance. Reconfigurable Computing utilizes hardware that can be adapted at run-time to facilitate greater flexibility without compromising performance. Reconfigurable architectures can exploit fine grain and coarse grain parallelism available in the application because of the adaptability. Exploiting this parallelism provides significant performance advantages compared to conventional microprocessors. The reconfigurability of the hardware permits adaptation of the hardware for specific computations in each application to achieve higher performance compared to software. Complex functions can be mapped onto the architecture achieving higher silicon utilization and reducing the instruction fetch and execute bottleneck.

Reconfigurable logic permits custom digital circuits to be dynamically created and modified via software. This ability to create and modify digital logic without physically altering the hardware provides a more flexible and lower cost solution to the implementation of custom hardware. This type of computer architecture is enabled by the availability of high density programmable logic chips, or FPGAs (Field Programmable Gate Arrays). FPGAs consist of a matrix of logic blocks and interconnection network. The functionality of the logic blocks and the connections in the interconnection network can be modified by downloading bits of configuration data onto the hardware. A new hardware program can be downloaded to these chips in a few milliseconds. Different bitstreams can be loaded during the execution of a program or to run a different program on the fly.

Currently, hybrid architectures which integrate programmable logic and interconnect together with a microprocessor on the same chip are being developed. The availability of increasingly larger number of transistors facilitates the integration of reconfigurable logic with other components on System-on-Chip (SoC) architectures. These newer FPGA-based architectures eliminate the need for a host processor by providing mechanisms to configure the device on boot from flash, and to directly support essential interfaces to memory and network resources via a bus configured in the device fabric to form a complete "System on a programmable Chip". Providing a stable and stateful computational platform within a reconfigurable device requires, however, partial reconfigurability, i.e., the ability to reconfigure only that portion of the device that implements an application, while leaving unchanged the rest of the portion(s) of the device that implements the platform, the memory and network interfaces, the device drivers, and so forth. The portion of the device which can be reconfigurable is known as Reconfigurable Logic Unit (RLU) i.e. one or more RLUs and fixed hardware portion constitutes the basic FPGA chip. RLU consist an array of multi-input and multi-output logic cells to be programmed. Examples of FPGA devices which allow partial reconfiguration are the Xilinx Virtex-II Pro and Virtex-4 devices [30], which include one or more PowerPC processors embedded within the FPGA's logic fabric. On-chip integration of reconfigurable logic reduces the memory access costs and the reconfiguration costs.

Applications are mapped onto reconfigurable architectures by analyzing the computations performed. Computations that can be speeded up by using reconfigurable hardware are identified and mapped onto the reconfigurable hardware. In the presence of a microprocessor, the computations which have complex control and data structures are executed on the microprocessor. The partitioning of the computations of an application between the microprocessor and the reconfigurable hardware is performed manually or by using automatic /semi-automatic tools. The partitioned computations are compiled into executable code on the microprocessor and hardware configurations on the reconfigurable hardware. The reconfigurable hardware needs to be configured using the configuration information before the actual execution can be performed. This configuration can be updated at run-time to execute a different set of computations from the application.

Development of systematic scheduling and mapping techniques for computing architectures require high level abstractions. Computing models at high level abstractions of the architectures can be utilized to develop algorithmic techniques for mapping applications onto the architectures. Reconfigurable computing is different from the Von-Neumann paradigm of computing and requires computational models different from conventional models. This gives rise to a design crisis in the tools for mapping applications onto reconfigurable architectures. Makimoto [3] predicts this changing paradigm in the wave illustrating technology trends in semiconductors (see Fig. 1.1).

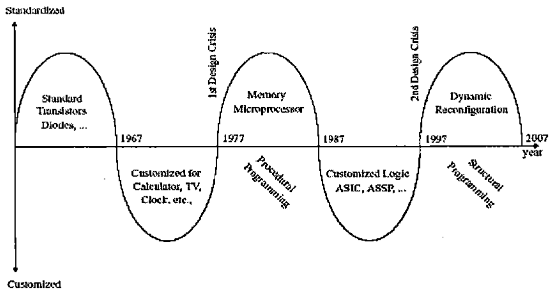

FIGURE 1.1: Makimoto's wave

There are several application areas where reconfigurable computing has been shown to achieve significant performance. These include long multiplication, cryptography, genetic algorithms, image processing, genomic database search, signal processing. The nature and diversity of the reconfigurable architectures results in a wide variety of implementation issues with respect to applications.

1.2 Motivation

To get the real advantage of the RCS, one has to provide general design methodology to explore different architectures which will support run-time configuration, efficient scheduling & mapping algorithms, reducing run-time configuration /reconfiguration overheads. Among these, Task scheduling becomes a very critical issue in achieving the high performance.

A lot of research is going on in this area of task scheduling for reconfigurable architectures. The previous works [15], [7], [24] are the significant contributions made in this area. The work in [15] has considered relatively better way for doing task scheduling for reconfigurable systems which takes into account the features of reconfigurable architectures. But it is targeted a particular coarse-grain reconfigurable device, MorphoSys [16], that imposes a set of architectural constraints, and provides dynamic reconfiguration. In [7], they proposed a dynamic scheduling algorithm for Sytem-on-Chip (SoC) platforms considering equal Dynamic Reconfiguration Logic (DRL) blocks. Using equal processing elements leads to inefficient hardware utilization as each processing element can execute all tasks. T. Wiangtong at al, [24] use the heuristic algorithm to map each task either to hardware or software. But heuristic approach always not gives the better results.

In this thesis, an application is modeled by a task graph. A task graph is a directed acyclic graph (DAG) in which nodes represent tasks and edges represent the data dependencies among the tasks. Although there are many heuristics proposed for scheduling DAG-type applications, most of them assume that the processors are equally capable, i.e. each processor can execute all the tasks. In real world application, this assumption usually does not hold. For example, software-defined radio (SDR) has to support multiple standards with seamless interconnection between two different standards in single device. Thus an electronic system has to support reconfiguration so that one can reconfigure different standards implementation in a single device to implement SDR in the system. In this work, variable size area RLUs have been considered instead of equal area RLUs and algorithm is developed for the RASIP for SDR architecture [1].

1.3 Problem Statement

For a given parallel application which is represented using Directed Acyclic Graph (DAG), finding the order in which tasks, have to be instantiated on to the partially reconfigurable hardware resources by using the scheduling and mapping algorithm while considering resource and real-time constraints with the objective of minimizing the total execution time of the application.

In this work, an efficient static task scheduling algorithm for reconfigurable computing system with variable area RLUs is proposed. For a given application, the objective of this scheduling algorithm is to map parallel and independent tasks onto the multiple variable area RLUs and order their execution so that a minimum schedule length (execution time) is given under the limit of task precedence requirements and resource constraints. Directed Acyclic Graph (DAG) is a most popular way that is used for modeling the precedence constraints among tasks. In this work, main focus is on minimizing total execution time by effective allocation of tasks to reconfigurable logic units.

As the proposed scheduling algorithm is static, the task parameters and RLU parameters are known in advance. Each task has different version of implementations which are varying in terms of hardware area required and time to execute that task using specified hardware area. Each RLU is associated with hardware area and time to reconfigure the RLU.

1.4 Organization of the Thesis

The rest of the dissertation is organized as follows.

Chapter 2 provides the relevant background information about Reconfigurable Computing Systems and discusses the related work in task scheduling.

Chapter 3 discusses about the features of Vertex-II pro and SystemC development environment.

Chapter 4 describes the task scheduling problem in RCS where RLUs having different areas on reconfigurable System-on-Chip (rSoC) and describes target architecture.

Chapter 5 presents proposed scheduling algorithm for reconfigurable computing systems along with Greedy based algorithm.

Chapter 6 discusses the results based on the comparison graphs of different scheduling strategies and scope for the future work.

Chapter 2

Background and Literature Review

2.1 Reconfigurable Computing Systems (RCS)

The general definition of Reconfigurable Computing (RC) [2] is "Computing via a post-fabrication and spatially and temporally programmed connection of processing elements," that is computation in space and time, using hardware that can adapt at the logic level to solve specific problems. RC can also be defined as a new paradigm based on dynamically adapting the computations on hardware through reconfiguration of available hardware as well as communication structures of the chip through programming the processing elements and their interconnections. Now, one can define that the Reconfigurable Computing Systems is a system which will allow post- fabrication programming in spatial as well as temporal manner to adapt and implement any logic or algorithm. Thus, RCS composed of SRAM based FPGAs, memory, CPUs or DSPs. A reconfigurable computing system can be mounted into a host computer or have its own embedded CPU.

2.1.1 Classification of Reconfigurable Architectures

Over the years a large number of Reconfigurable Architectures (RAs) have been developed. Reconfigurable architectures can be classified based on several different parameters. Some of the most distinguishing architectural parameters which can be used to classify reconfigurable architectures listed as follow [2].

- *Granularity*:The granularity of the reconfigurable logic is the size of the smallest functional unit that is addressed by the mapping tools. The granularity expresses the level of the functionality encapsulated into one design object. Lower granularity provides more flexibility in adapting the hardware to the computation structure. However, lower granularity has a performance penalty due to larger delays when constructing computation modules of a larger size using smaller functional units. Some architectures implement features that are specifically targeted toward reducing these overheads. For example, some FPGAs implement fast carry chains to permit construction of larger arithmetic modules from small functional units. Typically, FPGAs have smaller granularity such as two-input and four-input functional units. Several reconfigurable architectures such as Chameleon implement coarse-grain arithmetic units of larger size such as 32 bits.

- *Host coupling*:A large fraction of reconfigurable logic is utilized as a processing fabric attached to a host processor. The host processor performs the control functions to configure the logic, schedule data input and output, and external interfacing, among other things. The type of coupling to such a host system dictates the overheads in utilizing reconfigurable logic to speed up computations. The degree of coupling affects the reconfiguration and the data access costs.

  The degree of coupling can be roughly partitioned into three classes:

  - *Loose system–level coupling*: this includes architectures which have reconfigurable logic communicating to the host through an I/O interface similar to a disk drive and other peripherals. A large number of initial FPGA-based boards were architected with this degree of coupling.
  - *Loose chip–level coupling*: these systems reduce the overheads in communicating to the host by using direct communication between the host and the reconfigurable logic. A large number of existing embedded architectures with reconfigurable logic are architected using this technique.
  - *Tight on–chip coupling*: the availability of a large number of transistors has resulted in the integration of reconfigurable logic on the same chip as a host processor, significantly reducing the communication overheads between different components of the architecture.

- *Reconfiguration methodology*: Typically, a reconfigurable device is configured by downloading a sequence of bits known as a bitstream onto the device. The speed and methodology of download depend on the interface supported by the device. Two possible interfaces are bit-serial and bit-parallel interface. The time for configuration is directly proportional to the size of the bitstream. Fine–grain and coarse–grain devices have differences in the configuration time because coarse–grain devices typically need smaller configuration bitstreams. The flexibility of reconfiguration is achieved at the expense of reconfiguration cost. Reconfigurable logic has to stop computation for initiating a new configuration. This reconfiguration time can be significant, especially for fine-grain multimillion gate FPGAs. Some architectures support partial and dynamic reconfiguration. Partial reconfiguration permits reconfiguration of the functionality of a portion the device while the remaining portion retains its functionality. Dynamic reconfiguration permits reconfiguration of a portion of the device while other portions of the device are performing computations.

- *Memory organization*: The computation performed on the reconfigurable logic needs to access data from memory. Intermediate results from computations also need to be stored before the logic can be reconfigured to perform the next computation. The organization of the memory affects the data access cost and is a significant fraction of the actual execution time. Currently, most reconfigurable architectures include large memory on the reconfigurable logic device. This memory can be implemented as large blocks of memory or as distributed memory blocks.

2.1.2 Characteristics of Reconfigurable Logic

Reconfigurable logic[1] can be defined as consisting of a matrix of programmable computational units with a programmable interconnection network superimposed on the computational matrix. The fundamental differences between reconfigurable logic and traditional processing architectures include the following [2]:

[1]Reconfigurable logic is defined as a device that can be reprogrammed at run-time, in between computations, in the field.

- *Spatial Computation*: The data is processed by spatially distributing the computations rather than temporally sequencing through a shared computational unit.
- *Configurable Datapath*: The functionality of the computational units and the interconnection network can be adapted at run-time by using a configuration mechanism.
- *Distributed Control*: The computational units process data based on local configuration rather than an instruction broadcast to all the functional units.
- *Distributed Resources*: The required resources for computation, such as computational units and memory are distributed throughout the device instead of being localized in a single location.

The spatial distribution of the computations and the distributed control and resources result in higher computational power efficiency for reconfigurable computing compared to microprocessors, DSPs and ASICs. Computational power efficiency is defined as ratio of the number of gates actively working in a clock cycle to solve a problem to the total number of gates in a device. In traditional architectures like microprocessors and DSPs, a large portion of the chip is utilized to support active computation in a much smaller portion of the chip. Reconfigurable computing can achieve significantly higher computational power efficiency compared with conventional microprocessors and ASICs.

2.2 Review of Scheduling Algorithms

The problem of mapping (including matching and scheduling) tasks and communications is a very important issue since an appropriate mapping method can truly exploit the parallelism of the system thus achieving large speedup and high efficiency. It deals with assigning (matching) each task to a processor and ordering (scheduling) the execution of the tasks on each processor in order to minimize some cost function. The most common cost function is the total schedule length. Mapping and scheduling are used interchangeably. Unfortunately, the scheduling problem is extremely difficult to solve and is proved to be NP-complete in general. Even problems constructed from the original mapping problem by making simplified assumptions still fall in the class of NP-hard problems. Consequently, many heuristics have been proposed to produce adequate yet sub-optimal solutions. In general, the objective of task scheduling is to minimize the completion time of a parallel application by properly mapping the tasks to the processors.

There are many criteria used to categorize the types of scheduling method used. By considering the input characteristics, scheduling can be divided into those with or without data/control dependency. From the system architecture point of view, it may be categorized into scheduling for single processor, or multiple homogeneous/non-homogeneous processors. Nonetheless, in a broad sense, scheduling exists in two forms: static and dynamic scheduling. In the static scheduling case, all the information regarding the application and computing resources such as execution time, communication cost, data dependency, and synchronization requirement is assumed to be available a priori [18] . Scheduling is performed before the actual execution of the application. Static scheduling offers a global view of the application thus usually generates high quality schedules. On the other hand, in the dynamic mapping a more realistic assumption is used. Very little a priori knowledge is available about the application and computing resources. Scheduling is done during run-time. In order to support load balancing and fault tolerance, tasks can be reallocated during the execution.

A popular parallel application model is the task precedence graph model [18]. In this model, an application can be represented by a Directed Acyclic Graph (DAG). In a DAG, nodes represent the tasks and the directed edges represent the execution dependencies as well as the amount of communication between the nodes. A node in the DAG represents an atomic task that is a set of instructions that must be executed sequentially without preemption on the same processor. The weight of the node reflects the amount of work associated with the task. But in our target architecture of RCS, execution time of a task is different for each RLU in the system.

Here, the focus is on static DAG scheduling for variable area RLU system which seems to be similar to the scheduling for heterogeneous multiprocessor systems. Even though, it seems to be similar, in addition to heterogeneity of the RLU, this algorithm considers the specific features of reconfigurable systems. Next section discusses the existing scheduling algorithms for heterogeneous multiprocessor system. After that past work about the scheduling in RCS has been discussed in section 2.2.2

2.2.1 Static DAG scheduling algorithms for Heterogeneous Environment

Kwok and Ahmad [18] give a survey of various static DAG scheduling algorithms. The authors classify the considered algorithms into different categories based on the assumptions used in the algorithms such as the task graph structure (arbitrary DAG or restricted structure such as trees), computation costs, communication cost, duplication (task duplication allowed or not), number of processors and connection type among the processors. However, the 27 algorithms surveyed are mainly designed for a homogeneous environment. For algorithms designed for heterogeneous systems, there are basically four types, namely list scheduling based algorithms, clustering heuristics, task duplication heuristics and random search based algorithms [19](illustrated in Fig. 2.1).

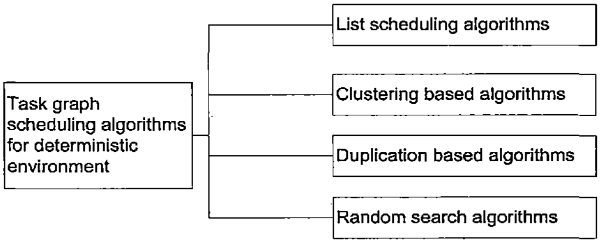
Task graph scheduling algorithms for deterministic environment
List scheduling algorithms
Clustering based algorithms
Duplication based algorithms
Random search algorithms

FIGURE 2.1: Taxonomy of static task scheduling algorithms for deterministic environment

List Scheduling: List scheduling [18] is a class of scheduling algorithms that assign tasks one by one according to their priorities. The essence of list scheduling is to make an ordered task list by assigning tasks some priorities and then repeatedly perform the following two steps until all the tasks in the list are scheduled:

1. Remove the first task from the list;
2. Allocate the task to a processor that will optimize some predefined cost function

The pseudo-code of list scheduling is presented in Alg. 1

There are two important questions in a list scheduling algorithm: (1) How to compute a task node's priority? (2) How to define the cost function? The first

Algorithm 1 List Scheduling()

```
Calculate the priority of each task according to some predefined formula
$PriorityList = \{v_1, v_2, \ldots, v_n\}$ is sorted by descending order of task priorities

while PriorityList is not empty do
  Remove the first task from the PriorityList and assign it to an appropriate
  processor in order to optimize a predefined cost function
end while
return (schedule)
```

question is related to the way the algorithm views the node's urgency of being scheduled. In the earlier list scheduling algorithms, the target computing systems are generally homogeneous. Some algorithms do not take into account the communication costs. Level-based heuristics are proposed for this case. For example, in the HLEFT algorithm [20], the level of a node denotes the sum of computation costs of all the nodes along the longest path from the node to an exit node.

Two important attributes [17, 18] used in the calculation of the priority of a task node are the t–level (top level) and b–level (bottom level). The t–level of a node is defined as the length of a longest path from an entry node to the node (excluding the node itself). The length of a path is the sum of all the node and edge weights along the path. As pointed out previously, the weights are approximations based on one of the criteria. The t-level is related to the earliest start time of the node. The b-level of a node is the length of a longest path from the node to an exit node. The critical path of a DAG is a longest path in the DAG. Clearly, the upper bound of a node's b-level is the critical path of the DAG. B–level and t–level can be computed with time complexity O(e + n), where e is the number of edges and n is the number of nodes in the DAG. The second question deals with the selection of "best" processor for a task. In homogeneous systems, a commonly used cost function is called earliest start time [21]. For example, the Earliest Time First (ETF) algorithm computes, at each step, the earliest start times for all ready nodes and then selects the one with the smallest earliest start time. When two nodes have the same value of their earliest start times, the ETF algorithm breaks the tie by scheduling the one with the higher static level. Some list scheduling based algorithms include Heterogeneous Earliest Finish Time (HEFT) [23], Dynamic Critical Path, Fast Critical Path (FCP) [22], and Insertion Scheduling Heuristic (ISH).

Clustering based heuristics: Another class of DAG scheduling algorithms is based on a technique called clustering. The basic idea of clustering based algorithm is to group heavily communicated tasks into the same cluster. Tasks grouped into the same cluster are assigned to the same processor in an effort to avoid communication costs. There are basically two types of clusters; linear and nonlinear. Two tasks are called independent if there are no dependence paths between them. A cluster is called nonlinear if there are two independent tasks in the same cluster, otherwise it is linear.

There are essentially two steps in a clustering based heuristic; grouping the nodes into clusters and mapping the clusters to processors. Initially, each task is assumed to be in a separate cluster. Then a series of refinements are performed by merging some existing clusters. A final clustering will be derived after certain steps. In order to avoid high time complexity, once the clusters have been merged they cannot be unmerged in the subsequent steps. During the mapping phase, sequences of optimizations are carried out: (1) Cluster merging: It is possible that the number of clusters is greater than the number of processors. Then it is necessary to further merge the clusters; (2) Task ordering: if the tasks in a cluster are related by precedence constraints, the execution order of the tasks is arranged based on such constraints.

Task duplication based heuristics:The basic idea behind task duplication based (TDB) scheduling algorithms is to use the idle time slots on certain processors to execute duplicated predecessor tasks that are also being run on some other processors, such that communication delay and network overhead can be minimized. In this way, some of the more critical tasks of a parallel program are duplicated on more than one processor. This can potentially reduce the start times of waiting tasks and eventually improve the overall completion time of the entire program. Duplication based scheduling can be useful for systems having high communication latencies and low bandwidths.

Guided random search algorithms: The task scheduling problem is a search problem where the search space consists of an exponential number of possible schedules with respect to the problem size. Guided random search algorithms are a class of search algorithms based on enumerative techniques with additional information used to guide the search. They have been used extensively to solve very complex problems. A common characteristic of these algorithms is that they are stochastic processes with the use of random probability. Evolution computation (ex: genetic algorithm) and stochastic relaxation (ex: simulated annealing) are the two major categories of guided random search algorithms.

2.2.2 Task scheduling in RCS

The scheduling problem in reconfigurable computing is relatively a new one. In comparison with multiprocessor systems, there exist a small number of works addressing the scheduling problem for RCS that involve reconfiguration time and hardware resources. Resource conflicts in shared bus or shared memory are usually ignored, making these works not applicable to real time systems.

In realistic systems, communication overhead, scheduling overhead and configuration overhead are all important. This makes the scheduling problem really hard to solve. Previous research such as [4] ignores some of these overheads to simplify the problem. It is well known that hardware/software partitioning and scheduling is a combinatorial optimization problem that is NP-complete. In addition, if FPGAs, which allow partial reconfiguration at run time, is employed, methods used to rearrange tasks are considered as an NP-hard problem.

Most of the approaches [8–11] are versions of existing high level synthesis (HLS) techniques extended in order to consider specific features of reconfigurable systems, such as the reconfiguration time.

A heuristic technique [10] based on static-list scheduling, enhanced to consider dynamic area constraints is proposed, while [11] presents a level-based scheduling algorithm. A new approach [12, 13] to the problem is presented, where an integrated linear programming (ILP) model is applied to the temporal partitioning of a task graph. Additionally, a technique [14] for loop fission that reduces the configuration overhead is proposed. All the related work discussed has not considered the features of the reconfigurable architectures.

Only the research work in [15] has considered relatively better way for doing task scheduling for reconfigurable systems. Given a task graph showing data dependencies, together with some additional information (task execution time, data sizes), the aim was to find the task schedule having the optimal execution time. But it is targeted a particular coarse-grain reconfigurable device, MorphoSys [16], that imposes a set of architectural constraints, and provides dynamic reconfiguration.

Dynamic techniques [5, 6], to schedule for reconfigurable architectures are proposed. Additionally, [7] introduced two different versions of dynamic reconfigurable architectures with or without a hardware prefetch unit and proposed dynamic scheduling algorithms for their architecture which tries to minimize the reconfiguration overhead by overlapping the execution of tasks with device reconfigurations. While such dynamic scheduling approaches ensure optimal resource utilization, they do not ensure real-time performance. Moreover, scheduling tasks during run-time is a costly overhead. Our approach is a static scheduling approach which gives near optimal solution by using dynamic programming concept.

The work done here differs from other reported works in many ways: one is it addresses scheduling problems for run-time reconfigurable computing system having variable area reconfigurable logic units and it also considers multiple implementations for each task. The architecture in this work considers Virtex-II Pro and Virtex-4 parameters, with their actual reconfiguration timings given in Table 3.1
.

Chapter 3

Hardware and Software Details

3.1 FPGA Development kits

Some of the FPGAs which allow the partial reconfiguration are Vertex-4 and Virtex-II Pro. In this thesis, the algorithm developed by considering the actual reconfiguration delays for these kits which are given in the Table. 3.1.

Features, benefits and applications of the Vertex-II pro [30] are discussed below:

Features:

- Industry's fastest FGPA fabric
- Up to four 300+ MHz, 420 +DMIPS IBM PowerPC 405 processors
- Up to twenty-four 3.125 Gbps full duplex RocketIO transceivers
- Over 10,008 Kbits Block RAM
- Up to 556 dedicated 18 x 18 Multipliers
- Up to 1200 users input/outputs
- Up to 12 Digital Clock Managers
- XCITE Digitally Controlled Impedance Technology

Benefits:

- On-Chip IBM PowerPC processors - Hard cores operating at peak efficiency, tightly coupled with all memory and programmable logic resources to maximize performance.
- High-Performance Connectivity Solution - Supports any of the existing single-ended and differential connectivity standards, such as PCI, HyperTransport, POS PHY, Flexbus, XSBI, and RapidIO, as well as all of the emerging serial connectivity standards such as XAUI, Fibre Channel, Serial ATA, InfiniBand, Serial RapidIO, and PCI Express (3GIO)
- Excellent Price/Performance Solution - Reduce your total bill of material cost and achieve higher performance, while reducing your overall development time. Plus, your product can easily adapt to new requirements, extending its profitability.
- On-Demand Architectural Synthesis - Specify high-level system requirements and generate architecture implementations. The flexibility of the Virtex-II Pro architecture allows you to partition the functionality of the hardware and software at any time during the design and development phase - or even after your product has shipped.
- Realtime Hardware and Software Debugging - Debug your processor software at full hardware speeds while you continue to optimize your hardware design.

Applications:

- Networking and Communications - Switch, router, network processor, MPLS
- Wireless Infrastructure - Packet switch (voice/data), base transceiver station, base controller, mobile switching center
- Storage Systems -Storage Area Networks and Storage subsystems/servers
- Professional Video: Camera to TV - Editing, storage, mixing, text/graphics effects, server, transmit to TV/STB
- Complex Embedded System - Industrial control, disk controller, medical instrumentation

TABLE 3.1: Reconfiguration Delay for Xilinx FPGA Devices in msec.

% of Device reconfiguration	**Virtex 4 (XC4VLX25)**	**Virtex II Pro (XC2VP30)**	**Virtex II Pro (XC2VP30)**
10	0.50	4.50	5.80
12	0.52	5.00	7.20
14	0.57	5.40	8.70
16	0.59	5.80	10.20
25	0.65	7.50	15.00
26	0.68	8.50	16.30
30	0.72	12.10	21.80
40	1.20	22.00	34.00
50	2.00	26.00	45.00
75	4.50	40.00	55.00
100	5.00	50.00	60.00

3.2 SystemC Language

SystemC [27] is a system design language that has evolved in response to a pervasive need for a language that improves overall productivity for designers of electronic systems. Typically, today's systems contain application-specific hardware and software. Furthermore, the hardware and software are usually co-developed on a tight schedule, the systems have tight real-time performance constraints, and thorough functional verification is required to avoid expensive and sometimes catastrophic failures. SystemC offers real productivity gains by letting engineers design both the hardware and software components together as these components would exist on the final system, but at a high level of abstraction. This higher level of abstraction gives the design team a fundamental understanding early in the design process of the intricacies and interactions of the entire system and enables better system trade offs, better and earlier verification, and over all productivity gains through reuse of early system models as executable specifications. It supports design abstraction at the RTL, behavioral, and system levels. SystemC consists of a class library and a simulation kernel. The language is an attempt at standardization of a C/C++ design methodology, and is supported by the Open SystemC Initiative (OSCI), a consortium of a wide range of system houses, semiconductor companies, IP providers, embedded software developers, and design automation tool vendors.

3.2.1 Why SystemC?

The systemC born because of the necessities of the current electronic industry: Electronic gadgets are incorporating greater and greater functionality today, but not compromising with the time to produce and market the gadgets. For example, you want your mobile handset to have internet facility but you are not ready to wait for one year for that facility to come. It is easy for you to demand, but it is not so easy for electronic design engineers who design the system. The greater complexity of the future systems is making the situation still worst. Previously, the C (or C++) was used to write the software part of the design. For hardware part any of the existing HDL's was used to design the hardware. It was very difficult to setup a test bench which is common for both, since they are entirely different languages. The introduction of SystemC solved many of these problems [28].

3.2.2 Language Comparison

Strictly speaking, SystemC is not a language, but rather a class library within a well established language, C++. SystemC is not a panacea that will solve every design productivity issue. However, when SystemC is coupled with the SystemC Verification Library, it does provide in one language many of the characteristics relevant to system design and modeling tasks that are missing or scattered among other languages. Additionally, SystemC provides a common language for software and hardware, C++.

Several languages have emerged to address the various aspects of system design. Although Ada and Java have proven their value, C/C++ is predominately used today for embedded system software. The hardware description languages (HDLs), VHDL and Verilog, are used for simulating and synthesizing digital circuits. SystemVerilog is a new language that evolves the Verilog language to address many hardware-oriented system design issues. Matlab and several other tools and languages such as System Studio are widely used for capturing system requirements and developing signal processing algorithms.

Fig. 3.1 highlights the application of these and other system design languages. Each language occasionally finds use outside its primary domain, as the overlaps in fig. 3.1 illustrate.

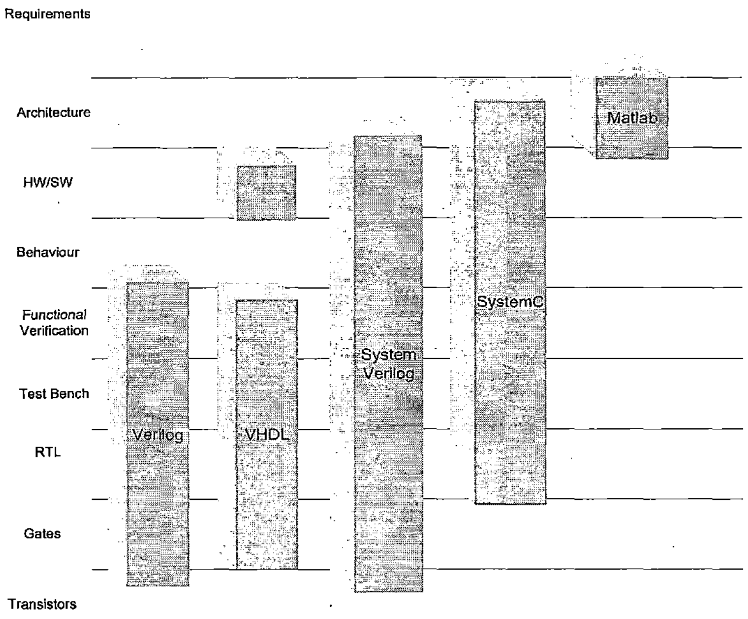

FIGURE 3.1: SystemC contrasted with other design languages [27]

3.2.3 Features of SystemC

1. It inherits all the features of C++, which is a stable programming language accepted all over the world. It has got large language constructs, which makes easier to write the program with less efforts.

2. Rich in data types: Along with the types supported by C++, SystemC supports the use of special data types which are often used by the hardware engineers.

3. It comes with a strong simulation kernel to enable the designers to write good test benches easily, and to simulate it. This is so important because the functional verification at the system level saves a lot of money and time.

4. It introduces the notion of time to C++, to simulate synchronous hardware designs. This is common in most of the HDL's.

5. While most of the HDL's support the RTL level of design, systemC supports the design at an higher abstraction level. This enables large systems to be modeled easily without worrying the implementation of it. It also supports RTL design, and this subset is usually called as systemC RTL.

6. *Concurrency*: To simulate the concurrent behavior of the digital hardware, the simulation kernel is so designed that all the "processes" are executed concurrently, irrespective of the order in which they are called.

3.2.4 SystemC Development Environment

Since SystemC is extension of C++, the development environment is the standard C/C++ development environment as shown in Fig. 3.2.

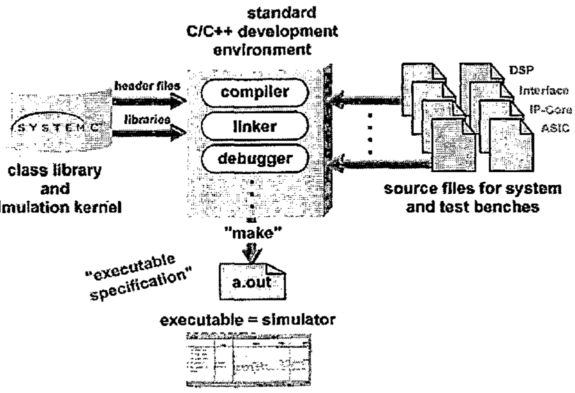
standard C/C++ development environment
header files
libraries
SYSTEMC
class library and simulation kernel
compiler
linker
debugger
DSP
Interface
IP-Core
ASIC
source files for system and test benches
"make"
"executable specification"
a.out
executable = simulator

FIGURE 3.2: SystemC Development Environment [30]

The designer writes the SystemC models at the system level, behavioral level, or RTL level using C/C++ augmented by the SystemC class library. The class library serves two important purposes. First, it provides the implementation of many types of objects that are hardware-specific, such as concurrent and hierarchical modules, ports, and clocks. Second, it contains a lightweight kernel for scheduling the processes. The user's SystemC code can now be compiled and linked together with the class library with any standard C++ compiler (such as GNU's gcc), and the resulting executable serves as the simulator of the user's design. The test bench for verifying the correctness of the design is also written in SystemC and compiled along with the design. The executable can be debugged in any familiar C++ debugging environment (such as GNU's gdb). Additionally, trace files can also be generated to view the history of selected signals using a standard waveform display tool.

Conceptually, the most powerful feature is that the hardware, software, and test bench parts of the design can be simulated in one simple and unified simulation environment without the need for clumsy co-simulations of disparate modeling paradigms.

Chapter 4

Task Scheduling in RCS Considering Variable Area RLUs

As described in chapter 2, the scheduling of task graphs is highly critical to the performance of reconfigurable computing systems. It deals with the allocation of individual tasks to suitable processing elements and proper order of task execution on each resource where the common objective is to minimize the overall completion time. In this chapter, we give the formal definition of task scheduling problem for reconfigurable system having variable area RLUs and discuss the target architecture for which algorithm is developed.

4.1 Problem Description

A scheduling system usually consists of three parts: application, computing environment, and scheduling goal. The application can be represented by a task graph.

4.1.1 Taskgraph

The DAG is a generic model of a workflow application consisting of a set of tasks (nodes) among which precedence constraints exist. It is represented by $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ represents the set of n tasks that can be executed on a subset of the available RLUs. E is the set of e directed arcs or edges between the tasks that maintain a partial order among them. The partial order introduces precedence constraints, i.e. if edge $e_{i,j} \in E$,then task v_j cannot start its execution before v_i completes. A node in the DAG represents an atomic task that is a set of instructions that must be executed sequentially without preemption on the same processor. A node without a parent is called an entry node, and a node without a child is called an exit node.

Each node (v_i) in the task graph is associated with multiple versions of implementations for that task. Each version (j) of implementation has their corresponding hardware requirement in terms of area or functional units ($a_{i,j}$) and time to execute that task ($w_{i,j}$). Normally, each task is associated with 3 different versions of implementations which will give the best, worst and moderate results in terms of hardware resources and execution time.

Furthermore, it is assumed in the following:

1. Tasks are non-preemptive: in a preemptive resource environment, a running task can be preempted from execution. Preemption is commonly used in priority-based or real-time systems. For example, when a pending task A's deadline is approaching, it is necessary to preempt one running job B, whose deadline is not as imminent as A's and assign the resource to task A. As a result of preemption, the scheduling is very complicated. In a non-preemptive computational resource, preemption is not allowed, i.e., once a task is started on such a resource, it cannot be stopped until its completion.

2. Only process level parallelism is considered. The application consists of a set of tasks (processes). Each task can only be assigned to one hardware module.

3. We assume that the task graph is a single-entry and single-exit one. If there is more than one exit or entry task, we can always connect them to a zero-cost pseudo exit or entry task with zero-cost edges. This will not affect the schedule.

4. The application in the form DAG is known in advance along with task parameters i.e. for each version corresponding area requirement and time to execute that task.

5. Each node in the DAG is functional unit at high level functionality implementation, rather than considering each instruction as node.

4.1.2 Computing Environment

Computing environment consists of m reconfigurable logic units (RLU) which are connected to a shared system bus as shown in Fig. 4.1. Each RLU r_i, where $i \in 1, \ldots, m$ has the parameters such as hardware area ra_i and the delay required to reconfigure the hardware area rd_i. For each RLU, the parameters such as area and reconfiguration delay are known in advance.

4.1.3 Performance Criteria

Before presenting the performance criteria, it is necessary to define a few attributes used in the algorithm.

- The execution time of task v_i on RLU r_j is denoted by $t_{i,j}$.
  Where,

$$\begin{aligned} t_{i,j} &= min\{et_k\} \\ & \quad k \in \{1, \ldots, q\} \\ et_k &= \infty,\ if\ ra_j < a_{i,k}; \\ et_k &= w_{i,k},\ if\ ra_j \geq a_{i,k}; \end{aligned} \tag{4.1}$$

  Here, q is the no.of versions for task v_i;
  If v_i cannot be processed on r_j then $t_{i,j} = \infty$. If two versions have the same execution time then the version with less hardware requirement is selected.

- We assume that the communication cost $c_{i,j}$ from task v_i to v_j as a constant K.

- $EST(v_i, r_j)$ and $EFT(v_i, r_j)$ are the earliest execution start time and the earliest execution finish time of task v_i on RLU r_j respectively. The entry task can start execution at time 0. Other task's EST can be computed by

$$EST(v_i, r_j) = max\{avail(v_i, r_j), \quad \underset{v_k \in pred(v_i)}{max}\{FT(v_k, r_{sk}) + c_{k,i}\}\} + rd_j \tag{4.2}$$

  where,

  $avail(v_i, r_j)$ is earliest time at which the RLU r_j is ready for task v_i's execution;

$pred(v_i)$ is the set of immediate predecessor tasks of task v_i;

rd_j is the time required to reconfigure the RLU j (reconfiguration delay for r_j);

The inner max block in the above Eq. 4.2. is the time that all the data needed to execute task v_i on RLU r_j is available, i.e. ready time. This is obtained by considering all immediate predecessors of task vi, the time they finish (FT) and the time needed to transfer the data from them.

The EFT is defined by

$$EFT(v_i, r_j) = t_{i,j} + EST(v_i, r_j) \tag{4.3}$$

The **schedule length** L of the DAG is the actual finish time of the exit task v_{exit}.

$$L = FT(v_{exit}) \tag{4.4}$$

The goal of the proposed scheduling algorithm is to minimize the scheduling length L.

4.2 Target System Architecture

The proposed scheduling algorithm is developed by considering reconfigurable application specific instruction-set processor for software-defined radio architecture model [1]. The architecture model consists of various reconfigurable logic units and fixed hardware units, system bus, local bus and shared memory with tightly coupled manner that is shown in Fig. 4.1. Fixed hardware has been divided into two parts say fixed hardware 1 (FHW1) and fixed hardware 2 (FHW2). Reconfigurable hardware has been divided into four reconfigurable logic units (RLUs), whenever tasks are available in the ready queue, the suitable RLU or fixed hardware unit has been selected.

The scheduler that is shown in the Fig. 4.2 takes the task graph, parameters for each task and RLU's information as input and produces the list specifying which task is mapped to which RLU considering RASIP architecture. The scheduler will select the task to be executed and on which RLU it has to execute. Then, resource manager accordingly allocates the specified RLU to the specified task. At the same

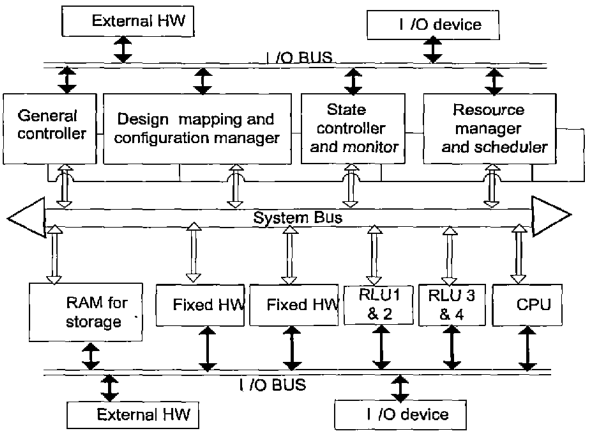

FIGURE 4.1: RASIP Architecture [1]

time, configuration manager will load the particular version of configuration for the task from the design library on to the RLU. Also the resource manager keeps track of information regarding free RLUs. The total time required to execute a task will also considers the time to reconfigure the hardware.

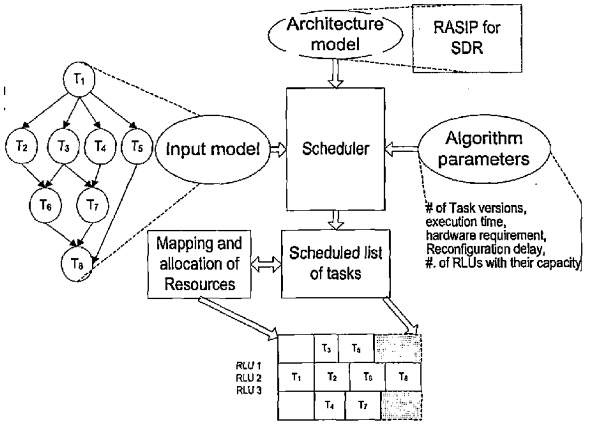
Architecture model
RASIP for SDR
T1
T2
T3
T4
T5
T6
T7
T8
Input model
Scheduler
Algorithm parameters
# of Task versions, execution time, hardware requirement, Reconfiguration delay, #. of RLUs with their capacity
Mapping and allocation of Resources
Scheduled list of tasks
RLU 1
RLU 2
RLU 3

FIGURE 4.2: Scheduler in RASIP Architecture

During the Scheduling process, RLU may be in one of the four possible states: idle, reconfiguration, task switch (context switch) or execution.

- IDLE state: the RLU is not performing any useful computation.
- RECONFIGURATION state: represents the state when the configuration manager is loading configuration bit stream of the task from the memory to the RLU i.e., reconfiguring the RLU.
- TASK (CONTEXT) SWITCH state: represents the state when the RLU must process a new task that uses the same reconfiguration context (task type) like the one currently loaded in the RLU. Before the new task execution can start, some registers need to be written which are required for the execution of task.
- EXECUTION state: represents the state when DRL device is doing some useful computation.

The state diagram for the RLU is shown in Fig 4.3. Initially the RLU will be in idle state. Whenever the scheduler selects a task to be executed on that RLU then, it may be either in the reconfiguration state or in the task switch state depending on the task to be executed. If the task to be executed has the same functionality as the previous task which has been executed on that RLU, then there is no need of configuring the RLU and directly it can go to the task switch state otherwise RLU requires the reconfiguration.

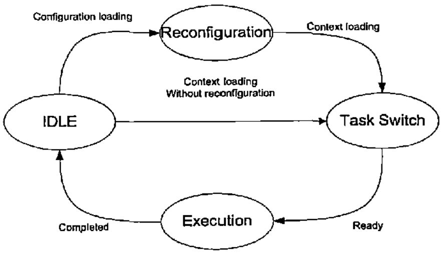

FIGURE 4.3: RLU State Diagram

During the reconfiguration state, configuration manager loads configuration bit stream of the task from the memory to the RLU. In the task switch state, all the parameters which are required for the execution of task are given. After all the parameters are passed, RLU enters into the execution state and starts the execution of the task.

Chapter 5

Scheduling Algorithms Developed for Variable Area RLUs

In this chapter, scheduling algorithms using greedy heuristic and dynamic programming are proposed for the system consisting of variable area RLUs.

5.1 The Greedy Technique

The greedy technique suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be feasible, locally optimal. Once a local optimum is picked, it is never changed nor is it re-examined. Greedy techniques require a correctness proof because there are times when a sequence of local optimums does not converge to a global optimum. In the next section, a scheduling algorithm using the greedy heuristic (SGH) is presented.

5.2 SGH Algorithm

SGH algorithm is developed based on static list scheduling algorithm. As with other list scheduling algorithms, the SGH algorithm has two major stages: a task prioritizing stage and a RLU selection stage. The first stage computes the priorities of all the tasks while the second one selects the tasks in the order of their priorities and assigns each selected task on its most desirable RLU, which minimizes the task's finish time.

5.2.1 Prioritizing the tasks

This step is essential for list scheduling algorithms. A task processing list is generated by sorting the task by decreasing order of some predefined rank function. In this work, level of a node is used as the rank function i.e., using the Highest Level First (HLF) heuristic of list scheduling to build the priority list of the tasks because the results of empirical performance studies [25] on list scheduling indicate that level-based heuristics are best at approximating the optimal schedule. The level of node v_i is the length of the longest path from v_i to the exit node. It can be obtained by recursively traversing the task graph from the exit node with time complexity $O(e+v)$.

$$\begin{aligned} LEV(v_i) &= \max\{LEV(v_j)\} \\ & \quad\; v_j \in succ(v_i) \qquad (5.1) \\ LEV(v_{exit}) &= 1 \qquad (5.2) \end{aligned}$$

where, $succ(v_i)$ is the set of immediate successors of v_i. The sorted list preserves the precedence constraints among tasks.

5.2.2 Selecting RLUs

In this step, tasks having highest level priority among all the ready tasks are selected for scheduling. Various criteria have been proposed to select suitable processor for a task. When scheduling in a homogeneous environment, EST is a popular choice. While in heterogeneous settings, using EFT as selection criteria gives better schedules [19]. SGH algorithm searches all the possible mappings to map the ready task set (T) efficiently on to free RLU set (R) with the objective of minimizing the schedule length at that stage. Here the selection function is to minimize the maximum function. For each mapping, maximum function returns the execution time of the task $v_i(v_i \in T)$, which takes more time than remaining tasks in that mapping. Task set T is allocated to R according to the mapping that minimizes the maximum function.

5.2.3 Pseudo–code

The pseudo-code for SGH algorithm is presented in Alg. 2.

Algorithm 2 SGH()

```
Build the priority list by computing levels for all tasks by traversing the graph
upward from the exit node
Sort the tasks in decreasing order of their priorities (level)
Initialize the ready queue with the tasks that has no immediate predecessors
repeat
  while hardware resources are available && ready queue is not empty do
    Sort the ready queue based on their priorities
    Get the information about the free RLUs from the Resource manager
    Add the ready tasks which are on same level to the task set $T$
    Add the free RLUs to resource set $R$
    for each task $v_i$ in $T$ do
      for each RLU capable $r_j$ in $R$ do
        Compute $EFT(v_i,r_j)$
      end for
    end for
    Find an efficient mapping from the available resources $R$ to ready tasks $T$
    that satisfies our objective. This can be achieved by searching all possible
    mappings
    Scheduler sends the resulted mapping details such as which ready task have
    to be allocated to which RLU to the Resource Manger
    Then Resource Manager will map the tasks on to the RLUs according to
    the mapping details given by Scheduler
    Remove the allocated tasks from the ready queue
  end while
  wait for a RLU free event from Resource manager
  Add ready tasks to the ready queue
until all tasks are scheduled
Display the scheduled list
```

5.2.4 Implementation Details

Functions that are used to implement the SGH algorithm are:

makePriorityList(): This function builds the priority list of tasks or nodes using HLF heuristic of List Scheduling algorithm.

sortPriorityList(): Function that sorts the task priority list in decreasing order of their priorities.

addReadyTasksToQueue(): This function adds the ready tasks to ready queue by checking the precedence constraints among the tasks.

allocateReadyTasks(): This function allocates all ready tasks to available hardware resources as well as maps the tasks onto the RLUs for execution using greedy heuristic i,e., local optimization. This function searches the all possible mappings from available resources to ready tasks and determines the best mapping which leads to the minimum execution time at that stage or meets our objective. This function also gets the free RLUs information from the resource manager and sends the information such as which task has been allocated to which RLU so that resource manager can keep track of the free RLUs.

5.3 Dynamic Programming

Dynamic Programming (DP) is an important optimization technique. It is efficient in finding optimal solutions for cases with many overlapping subproblems. It solves problems by successively recombining solutions to subproblems and sub-subproblems. In order to avoid solving these sub-subproblems several times, their results are gradually computed and memorized, starting from the simpler problems, until the overall problem itself is solved. Thus, dynamic programming is simply memorization of results of a recurrence, so that time is not spent trying to solve the same subproblem (or problem) repeatedly. Dynamic programming can only be applied when the problem under concern has optimal substructure. Optimal substructure means that the optimal solutions of local problems can lead to the optimal solution of the global problem. In simple terms, that means that the problem can be solved by breaking it down and solving the simpler problems. Thus one can say that, dynamic programming makes use of overlapping subproblems, optimal substructure and memorization.

Dynamic Programming differs from Greedy method in the fact that greedy method looks at the best possible choice at a particular point and uses it as a kind of local optimization. So, it may not generate optimal solution in some cases. Dynamic programming makes a sequence of decisions rather than step-wise decisions as a greedy method. In the next section, a scheduling algorithm using the dynamic programming (SDP) is presented.

5.4 SDP Algorithm

SDP algorithm uses the recursive function to produce the efficient solution to task scheduling problem for variable area RLUs.

5.4.1 Procedure

The recursive function f takes 2 parameters. First parameter represents the current state of all RLUs and the second parameter represents the queue of tasks waiting to be scheduled for execution. The second parameter is necessary since it is not possible to schedule all the functions which become candidates for execution on hardware resources after their precedence constraints are satisfied.

Current state of RLUs consists of which RLU is allocated to which task and at what time the RLU will complete the execution of that task and is represented by S.

$$S = \langle\langle v_{k1}, EFT(v_{k1}, r_1)\rangle, \langle v_{k2}, EFT(v_{k2}, r_2)\rangle, \ldots, \langle v_{km}, EFT(v_{km}, r_m)\rangle\rangle \quad (5.3)$$

where $v_{ij}, i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$ is the task that is being executed on j^{th} RLU.

Current queue consists of list of tasks that are waiting for execution and is represented by Q.

$$Q = \langle v_{q1}, v_{q2}, \ldots\rangle \quad (5.4)$$

where $v_{qi}, i \in \{1, \ldots, n\}$ is the task waiting for execution.

In each call to this function, it tries all the possible mappings of tasks in waiting queue to available resources and for each mapping it recursively calls this function with the state of RLUs modified to the selected mapping and queue is modified to ready tasks, until all the tasks are scheduled. While backtracking, it chooses the mapping that takes minimum time to execute.

5.4.2 Pseudo–code

The pseudo-code for recursive function *f(S,Q)* is presented in Alg. 3

Algorithm 3 SDP()

```
if $Q$ is empty then
  return
end if
for each possible mapping from $Q$ to free RLUs do
  Find the task that is going to finish first in selected mapping
  Remove that task and make that RLU free
  Find the tasks which are ready to execute after the completion of selected task
  Add the ready tasks to the queue $\acute{Q}$
  Change $S$ of the system to $\acute{S}$ where $\acute{S}$ represents the selected mapping
  Call the function f($\acute{S}$,$\acute{Q}$)
end for
```

5.4.3 Implementation Details

Important functions that are used to implement the SDP algorithm are:

Schedule(): This is base recursive function which takes current state of RLUs and queue of ready tasks and tries out the all possible mappings to produce the efficient schedule.

Map(): This function returns the all possible mappings from available resources set to ready task set.

Best(): This function returns the efficient mapping for a given state of RLUs and queue, from the stored efficient mappings.

Chapter 6

Conclusions

The Results of the various algorithms developed are compared in this chapter. The various conclusions and outlook is discussed.

6.1 Discussion of Results

To compare the proposed SDP algorithm, three more algorithms are developed in SystemC language. These algorithms are based on static list scheduling and prioritize the tasks using HLF method. The tasks on the same level are selected randomly for scheduling. The differences among three algorithms are in allocating the RLU for the selected task. The selected task is allocated to the large RLU (larger area fit) or to the small RLU (smaller area fit) or to the best RLU (best area fit) from the available RLUs. Here best RLU is in terms of execution time.

Performance comparison of Algortihms

Here we present a performance comparison of all algorithms, using execution time as parameter. For this purpose, a set of task graphs [26] as the workload for testing the algorithms is considered. The set contains regular task graphs representing various parallel algorithms and also synthetic task graphs representing commonly encountered algorithmic structures. The parallel algorithm considered is fast fourier transform (FFT) and synthetic task graphs include mean value analysis, cut-tree and fork-join.

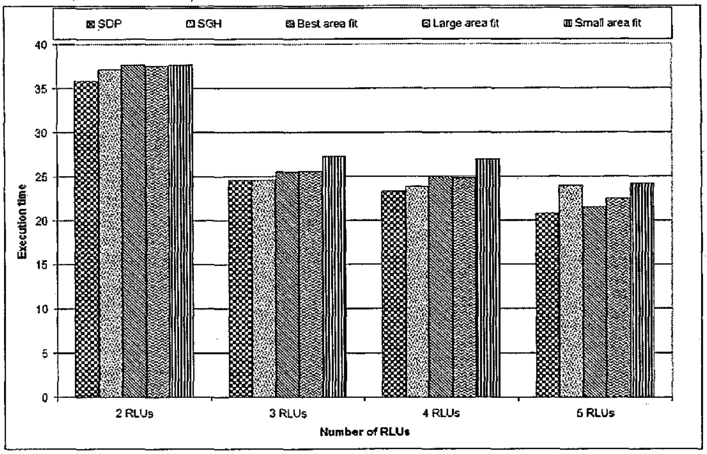

FIGURE 6.1: Performance comparison of developed algorithms for FFT task graph

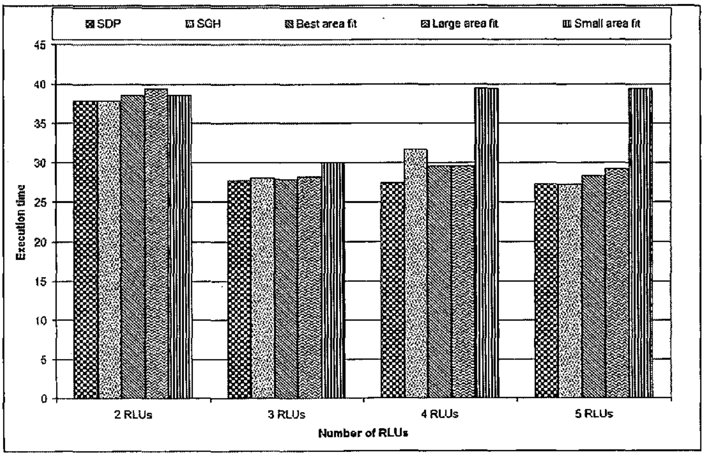

FIGURE 6.2: Performance comparison of developed algorithms for mean value task graph

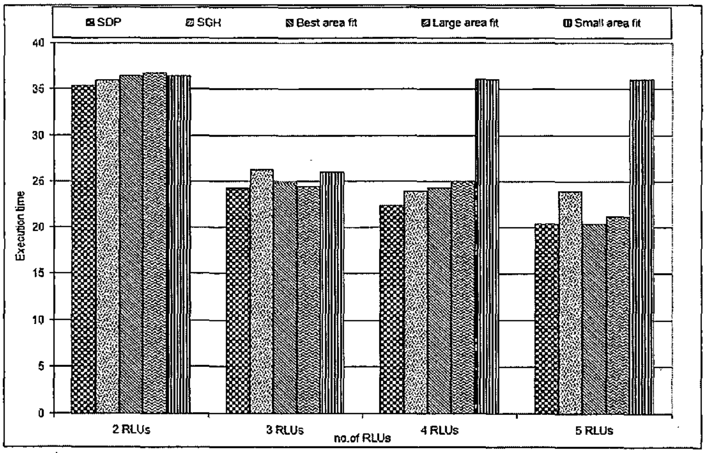

FIGURE 6.3: Performance comparison of developed algorithms for cut-tree task graph

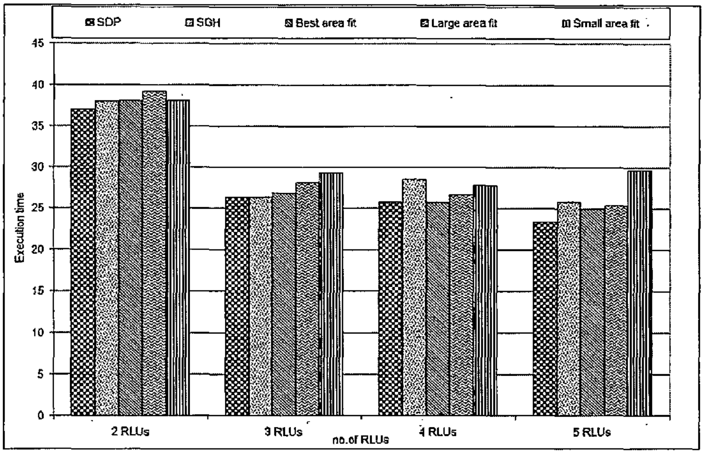

FIGURE 6.4: Performance comparison of developed algorithms for fork-join task graph

The execution times of the developed algorithms for all the above task graphs are shown in Fig. 6.1 to Fig. 6.4. Execution times for the task graphs are obtained by varying number of RLUs in the system.

In this thesis, a scheduling algorithm for reconfigurable System-on-Chip with variable area RLUs was developed using dynamic programming by considering the RASIP architecture [1]. This approach was an integrated approach which considers both scheduling and mapping of tasks together. The comparison shows that SDP algorithm always gives the best schedule minimizing the execution time when compared with the other developed algorithms. The Greedy heuristic (SGH and Best area fit) algorithms also produces a better schedule in almost all cases but lesser in number than the former. This is because the SDP algorithm searches the global optimal solution rather than local optimal solution as done by Greedy heuristic.

6.2 Suggestions for future work

There is a lot of scope for future work based on the work done in this thesis. These can be enumerated as follows:

1. The algorithms developed to perform scheduling can be integrated with the Design Library of the RASIP Architecture [1] which forms an integral part of this architecture.
2. In the algorithm developed, the tasks considered require lesser area than the maximum of the individual RLUs available. This algorithm can be extended to the tasks which requires more area than the maximum of individual RLUs.
3. Only Directed Acyclic Graphs are considered as input, this algorithm can also be modified to make it applicable for Directed Cyclic Graphs too.

Bibliography

[1] K. Solomon Raju, Chandra Shekhar and R. C. Joshi, "Design of Architecture and Instruction-set of RASIP for SDR," International Conference on Advanced Computing and Communications (ADCOM 2006), NIT Surathkal, December 2006, India.

[2] K. Bondalapati, V. Prasanna, "Reconfigurable Computing systems," Proceedings of IEEE, vol. 90, no. 7, pp.1201-1217, July 2002.

[3] R. Hartenstein, "The Microprocessor is no more General Purpose: why Future Reconfigurable Platforms will win; invited paper," Proceedings of International Conference on Innovative Systems in Silicon, ISIS'97, Austin, Texas, USA, October 8-10, 1997.

[4] A. Shrivastava and M. Kumar, "Optimal Hardware/Software Partitioning for Concurrent Specification using Dynamic Programming," International conference on VLSI design, pp. 110-113, 2000.

[5] J. Noguera and R. M. Badia, "Dynamic run-time HW/SW scheduling techniques for reconfigurable architectures," in Proceedings of the Tenth International Symposium on Hardware/Software Co-design, 2002.

[6] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable FPGAs," in IEE Proceedings - Computers and Digital Techniques, 2000.

[7] J. Noguera and R. M. Badia, "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling," ACM Transactions on Embedded Computing Systems, vol. 3, pp. 385-406, May 2004.

[8] I. Ouaiss, S. Govindarajan, V.Srinivasan, M. Kaul and R.Vemuri, "An Integrated Partitioning and Synthesis system for Dynamically Reconfigurable Multi-FPGA Architectures," 5th Reconfigurable Architectures Workshop, 1998.

[9] M. Vasilko and D. Ait-Boudaoud, "Architectural synthesis techniques for dynamically reconfigurable logic," in Proc. 6th Int. Workshop Field- Programmable Logic and Applications FPL'96, Darmstadt, Germany, pp. 290-296, Sept. 1996.

[10] M. Vasilko D, Ait-Boudaoud, "Scheduling for dynamically reconfigurable FP-GAs," in Proc.Int. Workshop on Logic and Architecture Synthesis, Grenoble, France, pp. 328-336, Dec. 1995.

[11] K. M. GajjalaPurna and D. Bhatia, "Temporal partitioning and scheduling for reconfigurable computing," Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, pp. 329-330, 1998.

[12] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," in Proc. Design, Automation, and Test in Eur. (DATE), Paris, France, pp. 389-396, Feb. 1998.

[13] Meenakshi Kaul, Ranga Vemuri, "Temporal partitioning combined with design space exploration for latency minimization of run-time reconfigured designs," in Proc. Design, Automation, and Test in Eur. (DATE), Munich, Germany,. pp. 202-209, Mar. 1999.

[14] M. Kaul, R.Vemuri, S. Govindarajan, and I. Ouaiss, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP application," in Proc. Design Automation Conf. (DAC), Atlanta, GA, pp. 616-622, Oct. 1999.

[15] Rafael Maestre, Fadi J. Kurdahi, Milagros Fernndez, Roman Hermida, Nader Bagherzadeh, and Hartej Singh, "A Framework for Reconfigurable Computing: Task Scheduling and Context Management," IEEE Transactions on VLSI Systems, vol. 9, no. 6, pp 858-873, December 2001.

[16] Hartej Singh, M.Lee, Fadi J. Kurdahi, and Nader Bagherzadeh, "MorphoSys: An Integrated Reconfigurable Architecture," Proc. of the NATO Symposium on System Concepts and Integration, Monterey, CA, April 1998.

[17] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," Parallel Distrib. Comput., 59(3):381-422, 1999.

[18] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," ACM Computing Surveys (CSUR), vol. 31, pp. 406-471, 1999.

[19] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," IEEE Trans. Parallel Distrib. Syst., 13(3):260-274, 2002.

[20] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," Commun. ACM, 17(12):685-690, 1974.

[21] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," SIAM J. Comput., 18(2):244-257, 1989.

[22] A. Radulescu and A. J. C. Van Gemund, "Fast and effective task scheduling in heterogeneous systems," In HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop, page 229, Washington, DC, USA, 2000. IEEE Computer Society.

[23] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," IEEE Transactions on Parallel Distributed Systems, vol. 13(3), pp 260-274, 2002.

[24] T. Wiangtong, P.Y.K. Cheung, W. Luk, "Cluster-Driven Hardware/Software Partitioning and Scheduling Approach for a Reconfigurable Computer System," Field-Programmable Logic and Applications (FPL), pp. 1071-1074, 2003.

[25] L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," ACM Transactions, V01. 17, pp.685-690, 1974.

[26] I. Ahmed and Kwok, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," IEEE Transaction on Parallel and Distributed Systems, vol. 7, 1996.

[27] David C.Black and Jack Donovan, "SystemC from the ground," Kluwer Academic Publishers, Boston, 2004.

[28] J.Bhasker, "A SystemC Primer," Star Galaxy Publishers, USA, 2004.

[29] www.systemc.org

[30] www.xilinx.com

Appendix A

Code Listing

<u>**Main.cpp:**</u>

```
#include<systemc.h>
#include"driver.h"
#include"rmgr.h"
#include"functions.h"
#include"schedular.h"
#include<iostream>
#include<string>

sc_event allocate_RLU;
sc_event RLU_allocated;
sc_event RLU_free;
sc_event free_RLU1;
sc_event free_RLU2;
sc_event free_RLU3;


int sc_main (int argc,char* argv[]) {
    sc_clock t_clk( "clk",10,SC_NS);
    sc_signal<int> t_noSarea,t_noTask,t_noNext[48],
    t_nextID[48][48],t_noVersions;
    sc_signal<float>t_task[48][10][2];
    sc_signal<float>t_sArea[10][3];
    sc_signal<int>t_noCtask;
    sc_signal<float>t_cTask[48][6];
    sc_signal<bool>t_t[48];
    sc_signal<int>t_t1[48][2];
    int i,j;

    rmgr r("Read informationg Regarding RLU's");
    r.r_clk(t_clk);
```

```
r.r_noSarea(t_noSarea);
for(i=0;i<10;i++)
{
    r.r_sArea[i][0](t_sArea[i][0]);
    r.r_sArea[i][1](t_sArea[i][1]);
    r.r_sArea[i][2](t_sArea[i][2]);
}
for(i=0;i<48;i++)
{
    r.r_t[i](t_t[i]);
    r.r_t1[i][0](t_t1[i][0]);
    r.r_t1[i][1](t_t1[i][1]);
}
r.r_noCtask(t_noCtask);
for(i=0;i<48;i++)
{
    for(j=0;j<6;j++)
        r.r_cTask[i][j](t_cTask[i][j]);
}
driver d("Read TaskGraph");
d.d_clk(t_clk);
d.d_noTask(t_noTask);
d.d_noVersions(t_noVersions);
for(i=0;i<48;i++)
{
    for(j=0;j<10;j++)
    {
        d.d_task[i][j][0](t_task[i][j][0]);
        d.d_task[i][j][1](t_task[i][j][1]);
    }
}
for(i=0;i<48;i++)
    d.d_noNext[i](t_noNext[i]);
for(i=0;i<48;i++)
{
    for(j=0;j<48;j++)
        d.d_nextID[i][j](t_nextID[i][j]);
}
schedular sh("Allocate");
sh.clk(t_clk);
sh.noSarea(t_noSarea);
for(i=0;i<10;i++)
{
    sh.sArea[i][0](t_sArea[i][0]);
    sh.sArea[i][1](t_sArea[i][1]);
    sh.sArea[i][2](t_sArea[i][2]);
}
```

```
    sh.noTask(t_noTask);
    sh.noVersions(t_noVersions);
    for(i=0;i<48;i++)
    {
        for(j=0;j<10;j++)
        {
            sh.task[i][j][0](t_task[i][j][0]);
            sh.task[i][j][1](t_task[i][j][1]);
        }
    }
    for(i=0;i<48;i++)
        sh.noNext[i](t_noNext[i]);
    for(i=0;i<48;i++)
    {
        for(j=0;j<48;j++)
            sh.nextID[i][j](t_nextID[i][j]);
    }
    sh.noCtask(t_noCtask);
    for(i=0;i<48;i++)
    {
        for(j=0;j<6;j++)
            sh.cTask[i][j](t_cTask[i][j]);
    }
    functions f("functionss");
    for(i=0;i<48;i++){
        f.f_t1[i][0](t_t1[i][0]);
        f.f_t1[i][1](t_t1[i][1]);
        f.f_t[i](t_t[i]);
        //f.f_time[i](t_task[i][1]);
    }
    f.f_noCtask(t_noCtask);
    for(i=0;i<48;i++)
    {
        for(j=0;j<6;j++)
            f.f_cTask[i][j](t_cTask[i][j]);
    }
    sc_start(100,SC_NS);
    sc_close_vcd_trace_file(tfile);
    return 0;
}
```

<u>Driver.cpp:</u>

```
#include"driver.h"
#include <iostream>
#include <fstream>
#include <string>
```

```
void driver::prc_driver() {

    wait(5,SC_NS);
    int t_noTask,t_noNext[48],t_nextID[48][48],next,t_noVersions;
    float t_task[48][10][2];
    int i,j;

    fstream infile;
    infile.open("taskGraph.txt",ios::in);
    if(infile.fail())
    {
         cout << "*****Error in Opening the File Task graph****";
         sc_stop();
    }

    infile >>t_noTask;
    infile >>t_noVersions;
    for(i=0;i<t_noTask;i++)
    {
        for(j=0;j<t_noTask;j++)
            d_nextID[i][j]=0;
    }
    d_noTask=t_noTask;
    d_noVersions=t_noVersions;
    for(i=0;i<t_noTask;i++)
    {
        for(j=0;j<t_noVersions;j++)
        {
            infile>>t_task[i][j][0];
            infile>>t_task[i][j][1];
            d_task[i][j][0]=t_task[i][j][0];
            d_task[i][j][1]=t_task[i][j][1];
        }
        infile>>t_noNext[i];
        d_noNext[i]=t_noNext[i];

        for(j=0;j<t_noNext[i];j++)
        {
            infile>>next;
            d_nextID[i][next-1]=1;
        }
    }
}
```

Rmgr.cpp:

```
#include"rmgr.h"
#include <iostream>
#include <fstream>
#include <string>
#include<conio.h>
int maxarea;

void rmgr::prc_readRLU()
{
    int i,t_noSarea;
    float t_sArea[10][3];
    maxarea=0;

    fstream file1;
    file1.open("rlu.txt",ios::in);
    if(file1.fail())
    {
         cout << "******Error in Opening the File RLU******";
         sc_stop();
    }
    file1>>t_noSarea;
    r_noSarea=t_noSarea;
    for(i=0;i<t_noSarea;i++)
    {
        file1>>t_sArea[i][0];
        file1>>t_sArea[i][2];
        r_sArea[i][0]=t_sArea[i][0];
        r_sArea[i][1]=0;
        //r_sArea[i][2]=0;
        r_sArea[i][2]=t_sArea[i][2];
        if(t_sArea[i][0]>maxarea)
        maxarea=t_sArea[i][0];
    }
    wait(5,SC_NS);
}
void rmgr::prc_allocateRLU()
{
    while(1)
    {
        wait(allocate_RLU);
        int i;
        //cout<<"\n*******PROCEDURE allocateRLU started";
        for(i=0;i<r_noCtask;i++)
        {
            if(r_cTask[i][2]==0)
            {
                r_sArea[(int)r_cTask[i][1]][1]=r_cTask[i][0]+1;
                r_t[(int)r_cTask[i][0]]=1;
                r_t1[(int)r_cTask[i][0]][0]=r_cTask[i][1]+1;
                r_t1[(int)r_cTask[i][0]][1]=i;
                r_cTask[i][4]=sc_time_stamp()
                .to_default_time_units()-sim_time;
                r_cTask[i][2]=2;//0-scheduled
                                //1-completed
                                //2-resource allocated
            }
        }
        RLU_allocated.notify(SC_ZERO_TIME);
        //cout<<"\n*******PROCEDURE allocateRLU ended";
    }
}
void rmgr::prc_freeRLU1()
{
    while(1)
    {
        wait(free_RLU1);
        //cout<<"\n*******PROCEDURE freeRLU1 started";
        int i;
        r_sArea[0][1]=0;
        //cout<<"\n*******PROCEDURE freeRLU1 ended";
        RLU_free.notify(SC_ZERO_TIME);
    }
}
void rmgr::prc_freeRLU2()
{
    while(1)
    {
        wait(free_RLU2);
        //cout<<"\n*******PROCEDURE freeRLU2 started";
        int i;
        r_sArea[1][1]=0;
        //cout<<"\n*******PROCEDURE freeRLU2 ended";
        RLU_free.notify(SC_ZERO_TIME);
    }
}
void rmgr::prc_freeRLU3()
{
    while(1)
    {
        wait(free_RLU3);
        //cout<<"\n*******PROCEDURE freeRLU3 started";
        int i;
        r_sArea[2][i]=0;
        //cout<<"\n*******PROCEDURE freeRLU3 ended";
        RLU_free.notify(SC_ZERO_TIME);
    }
}
```

Schedular.cpp:

```
#include"schedular.h"
#include <iostream>
#include<conio.h>
#include <fstream>
#include <string>
double sim_time=10;


Rset Rset::operator-(int pos)
{
    int i;
    Rset r1;
    r1.noR=noR-1;
    for(i=0;i<pos;i++)
        r1.R[i]=R[i];
    for(i=pos;i<noR-1;i++)
    {
        r1.R[i]=R[i+1];
    }
    return r1;
}
Tset Tset::operator-(int pos)
{
    int i;
    Tset t1;
    t1.noT=noT-1;
    for(i=0;i<pos;i++)
        t1.T[i]=T[i];
    for(i=pos;i<noT-1;i++)
    {
        t1.T[i]=T[i+1];
    }
    return t1;
}
void schedular::prc_initialise()
{
    wait(sim_time,SC_NS);
    //cout<<"\n*******PROCEDURE initialise started";
    int i,j;
    cout<<"\n\n*******************************************";
    cout<<"\n\n\t\t\t\tSILICON AREAS";
    cout<<"\n\n*******************************************";
    cout<<"\n\n\t\tArea\tReconfiguration Delay";
    for(i=0;i<noSarea;i++)
        cout<<"\n\nRLU "<<i+1<<":  \n\t\t"<<sArea[i][0]
        <<"\t"<<sArea[i][2];
    cout<<"\n\n*********************************************";
    cout<<"\n\n*********************************************";
    cout<<"\n\n\t\t\t\tTASK GRAPH";
    cout<<"\n\n*********************************************";
    for(i=0;i<noTask;i++)
    {
        cout<<"\n\n\t\t\t\tDetails of TaskID: "<<i+1;
        cout<<"\n\t\t\t\t*************************";
        for(j=0;j<noVersions;j++)
        {
            cout<<"\n\nVersion "<<j+1<<": ";
            cout<<"\n\n\t\tArea           : "<<task[i][j][0];
            cout<<"\n\n\t\tExecution time : "<<task[i][j][1];
        }
    }
    getch();
    cout<<"\n\n\t\tTASK ADJACENCY MATRIX\n\n";
    for(i=0;i<noTask;i++)
        cout<<"\t"<<i+1;
    for(i=0;i<noTask;i++)
    {
        cout<<"\n"<<i+1<<"\t";
        for(j=0;j<noTask;j++)
            cout<<nextID[i][j]<<"\t";
        cout<<"\n";
    }
    cout<<"\n\n********************************************";
    getch();
    //Initialisation
    noCtask=0;
    noQ=0;
    makePriorityList();
    sortPriorityList();
    prc_allocation();
}
void schedular::prc_allocation()
{
    cout<<"\n\n****SCHEDULING & ALLOCATION OF TASKS*****\n\n";
    addReadyTaskstoQueue();
    int i,j;
    do
    {
        for(;;)
        {
            if(resourcesAvailable() && noQ>0)
            {
                if(allocateReadyTask());
                else break;
            }
            else
                break;
        }
        wait(RLU_free);
        addReadyTaskstoQueue();
    }while(noCtask!=noTask);
    wait();
    cout<<"\n\n*******************************";
    cout<<"\n\n\t\tSCHEDULED LIST OF TASKS";
    cout<<"\n\n*******************************";
    cout<<"\n\n\tTaskID\tRLU ID\tTime\tSTime\tETime";
    for(i=0;i<noCtask;i++)
    {
        cout<<"\n\n\t"<<cTask[i][0]+1<<"\t"<<cTask[i][1]+1<<"\t"
        <<cTask[i][3]<<"\t"<<cTask[i][4]<<"\t"<<cTask[i][5];
    }
    cout<<"\n\n*********************************";
    getch();
    area_used();
}
void schedular::addReadyTaskstoQueue()
{
    int level=0;
    for(int i=0;i<noTask;i++)
    {
        if(isReady(List[i][0]) && List[i][2]==0)
        {
            if(level==0)level=List[i][1];
            readyQueue[noQ][0]=List[i][0];
            readyQueue[noQ][1]=0;
            readyQueue[noQ++][2]=List[i][1];
            List[i][2]=1;
        }
    }
}
void schedular::deleteTaskfromQueue(int TID)
{
    int pos;
    for(int i=0;i<noQ;i++)
    {
        if(TID==readyQueue[i][0])
            pos=i;
    }
    for(int i=pos;i<noQ-1;i++)
    {
        readyQueue[i][0]=readyQueue[i+1][0];
        readyQueue[i][1]=readyQueue[i+1][1];
        readyQueue[i][2]=readyQueue[i+1][2];
    }
    noQ--;
}
void schedular::makePriorityList()
{
    int i,j,level;
    level=1;
    for(i=0;i<noTask;i++)
    {
        for(j=0;j<3;j++)
            List[i][j]=0;
    }
    for(i=0;i<noTask;i++)
    {
        if(noNext[i]==0)
        {
            List[i][0]=i;
            List[i][1]=level;
        }
    }
    int cond=0;
    do
    {
        cond=0;
        for(i=0;i<noTask;i++)
        {
            if(List[i][1]==level)
            {
                for(j=0;j<noTask;j++)
                {
                    if(nextID[j][List[i][0]]==1)
                    {
                        List[j][0]=j;
                        if(List[j][1]<(level+1))
                            List[j][1]=level+1;
                        cond=1;
                    }
                }
            }
        }
        level++;
    }while(cond==1);
}
void schedular::sortPriorityList()
{
    for(int i=0;i<noTask;i++)
    {
        for(int j=i+1;j<noTask;j++)
        {
            if(List[i][1]<List[j][1])
            {
                int temp;
                temp=List[i][0];
                List[i][0]=List[j][0];
                List[j][0]=temp;
                temp=List[i][1];
                List[i][1]=List[j][1];
                List[j][1]=temp;
            }
        }
    }
    cout<<"\n\n\tPriority List for the TaskGraph";
    cout<<"\n\t-----------------------------------";
    cout<<"\n\n\tTaskID\tPriority";
    for(int i=0;i<noTask;i++)
    {
        cout<<"\n\t"<<List[i][0]+1<<"\t"<<List[i][1];
    }
    getch();
}
void schedular::sortReadyQueuebasedonPriority()
{
    for(int i=0;i<noQ;i++)
    {
        for(int j=i+1;j<noQ;j++)
        {
            if(readyQueue[i][2]<readyQueue[j][2])
            {
                int temp;
                temp=readyQueue[i][0];
                readyQueue[i][0]=readyQueue[j][0];
                readyQueue[j][0]=temp;

                temp=readyQueue[i][1];
                readyQueue[i][1]=readyQueue[j][1];
                readyQueue[j][1]=temp;

                temp=readyQueue[i][2];
                readyQueue[i][2]=readyQueue[j][2];
                readyQueue[j][2]=temp;
            }
        }
    }
}
float schedular::getRLU(int TID,int RLUid)
{
    int cond=0;
    int ID;
    for(int i=0;i<noQ;i++)
    {
        if(readyQueue[i][0]==TID)
            ID=i;
    }
    for(int i=0;i<noVersions;i++)
    {
        if(task[readyQueue[ID][0]][i][0]<=sArea[RLUid][0])
        {
            if(cond==0)
            {
                cond++;
                readyQueue[ID][1]=i;
            }
            else if(task[readyQueue[ID][0]]
            [readyQueue[ID][1]][1]>task[readyQueue[ID][0]][i][1])
                readyQueue[ID][1]=i;
            else if(task[readyQueue[ID][0]]
            [readyQueue[ID][1]][1]==task[readyQueue[ID][0]][i][1]
            && task[readyQueue[ID][0]]
            [readyQueue[ID][1]][0]>task[readyQueue[ID][0]][i][0])
                readyQueue[ID][1]=i;
        }
    }
    if(cond>0)
        return task[readyQueue[ID][0]][readyQueue[ID][1]][1]+sArea[RLUid][2];
    else
        return -1;
}
float schedular::gen(Rset rs,Tset ts,int pos)
{
    int i,j;
    float maxv=-1;
    //cout<<"\nfun called with noR="<<rs.noR<<" and noT="<<ts.noT;
    if(rs.noR==0 || ts.noT==0)
    {
        maxv=-1;
        for(i=0;i<notS;i++)
        {
            //cout<<"\n\t\t"<<tmps[i][0]<<"\t"<<tmps[i][1];
            maxv=max(maxv,c[tmps[i][0]][tmps[i][1]]);
        }
        if((minv==200.0 || minv>maxv) && maxv>0)
        {
            for(i=0;i<notS;i++)
            {
                s[i][0]=tmps[i][0];
                s[i][1]=tmps[i][1];
            }
            noS=notS;
            minv=maxv;
        }
        return -1;
    }
    else if(rs.noR<ts.noT)
    {
         for(j=0;j<ts.noT;j++)
           {
            tmps[pos][0]=rs.R[0];
            tmps[pos][1]=ts.T[j];
            notS=pos+1;
            float cost=c[rs.R[0]][ts.T[j]];
            gen(rs-0,ts-j,pos+1);
           }
    }
    else
    {
           for(i=0;i<rs.noR;i++)
            {
            tmps[pos][0]=rs.R[i];
            tmps[pos][1]=ts.T[0];
            notS=pos+1;
            float cost=c[rs.R[i]][ts.T[0]];
            gen(rs-i,ts-0,pos+1);
            }
    }
    return maxv;
}
int schedular::allocateReadyTask()
{
```

```
int i,j;
notS=noS=0;
minv=200.0;
Rset rs;Tset ts;
int R[48],T[48];
rs.noR=0;
for(i=0;i<noSarea;i++)
{
    if(sArea[i][1]==0)
    {
        rs.R[rs.noR]=rs.noR;
        R[rs.noR++]=i;
    }
}
cout<<"\n\n --------------";
cout<<"\n|               \\";
cout<<"\n|      INPUT     \\";
cout<<"\n|                /";
cout<<"\n|               /";
cout<<"\n ---------------";
cout<<"\n\n\tFREE RLUs: \n";
for(i=0;i<rs.noR;i++)
    cout<<"\n\t\t"<<R[i]+1;
sortReadyQueuebasedonPriority();
int level=readyQueue[0][2];
ts.noT=0;
for(i=0;i<noQ;i++)
{
    if(readyQueue[i][2]>=level)
    {
        ts.T[ts.noT]=ts.noT;
        T[ts.noT++]=readyQueue[i][0];
    }
}
cout<<"\n\n\tTASKS: ";
for(i=0;i<ts.noT;i++)
    cout<<"\n\t\tTaskID: "<<T[i]+1;
//calculating cost matrix rows-resourses and cols-Tasks
for(i=0;i<rs.noR;i++)
{
    for(j=0;j<ts.noT;j++)
        c[i][j]=getRLU(T[j],R[i]);
}
cout<<"\n\n\tCOST matrix: \n";
cout<<"\n\t\tTaskID \t";
for(i=0;i<ts.noT;i++)
    cout<<"\t"<<T[i]+1;
    for(i=0;i<rs.noR;i++)
    {
        cout<<"\n\n\t\tRLU "<<R[i]+1<<"\t\t";
        for(j=0;j<ts.noT;j++)
            cout<<c[i][j]<<"\t";
        cout<<"\n";
    }
    getch();
    gen(rs,ts,0);

    cout<<"\n\n\t\t\t\t\t ------------";
    cout<<"\n\t\t\t\t\t\t|             \\";
    cout<<"\n\t\t\t\t\t\t|     OUTPUT   \\";
    cout<<"\n\t\t\t\t\t\t|              /";
    cout<<"\n\t\t\t\t\t\t|             /";
    cout<<"\n\t\t\t\t\t\t -------------";
    cout<<"\n\n\tBEST WAY OF ALLOCATING THE
    READY TASKS ONTO THE FREE RLUs is: \n";
    if(noS==1 && c[s[0][0]][s[0][1]]==-1)
        return 0;
    for(i=0;i<noS;i++)
    {
        //cout<<"\n\t\t"<<s[i][0]<<"\t"<<s[i][1];
        cTask[noCtask][0]=T[s[i][1]];
        cTask[noCtask][1]=R[s[i][0]];
        cTask[noCtask][2]=0;
        cTask[noCtask][3]=c[s[i][0]][s[i][1]];

        noCtask=noCtask+1;

        allocate_RLU.notify(SC_ZERO_TIME);
        wait(RLU_allocated);
    }
    getch();
    for(i=0;i<noS;i++)
        deleteTaskfromQueue(T[s[i][1]]);
    return 1;
}
int schedular::resourcesAvailable()
{
    for(int i=0;i<noSarea;i++)
    {
        if(sArea[i][1]==0)
            return 1;
    }
    return 0;
}
```

```
int schedular::isTaskScheduled(int ID)
{
    for(int i=0;i<noCtask;i++)
    {
        if(cTask[i][0]==ID)
            return 1;
    }
    return 0;
}
int schedular::isReady(int ID)
{
    for(int i=0;i<noTask;i++)
    {
        if(nextID[i][ID]==1)
        {
            if(isTaskScheduled(i)==0)
                return 0;
            else if(isTaskScheduled(i)==1)
            {
                if(isTaskCompleted(i)==0)
                    return 0;
            }
        }
    }
    return 1;
}
int schedular::isTaskCompleted(int ID)
{
    for(int i=0;i<noCtask;i++)
    {
        if(cTask[i][0]==ID && cTask[i][2]==1)
            return 1;
    }
    return 0;
}
void schedular::area_used()
{
    int *count=new int[noSarea];
    int i;
    for(i=0;i<noSarea;i++)
        count[i]=0;
    for(i=0;i<noCtask;i++)
    {
       count[(int)cTask[i][1]]++;
    }
    cout<<"\n\n\tArea\tNo.of Times Used";
    for(i=0;i<noSarea;i++)
```

```
        cout<<"\n\n\tRLU "<<i+1<<"\t"<<count[i];
}
```